Berlin United - Nao Team Humboldt Team Report 2015 - RC2

Heinrich Mellmann
Thomas Krause
Claas-Norman Ritter
Steffen Kaden
Tobias Hübner
Benjamin Schlotter
Schahin Tofangchi

7th December 2015

Contents

Chapter 1

Introduction

Our team is part of the multi-league joint research group *Berlin United* between the RoboCup research group of the Humboldt-Universität zu Berlin and the Freie Universität Berlin (FUmanoids, KidSize League). The research group *NaoTH* was founded at the end of 2007 and consists of students and researchers at the Humboldt-Universität zu Berlin. The team is part of the research lab for Adaptive Systems at Humboldt-Universität which is headed by Prof. Verena Hafner. The team was established at and evolved from the AI research lab headed by Prof. Hans-Dieter Burkhard, and is led by Heinrich Mellmann and Marcus Scheunemann. At the current state the core team consists of two PhD, four Master/Diploma, and six Bachelor students. Additionally, we provide courses and seminars where the students solve tasks related to RoboCup and other problems of Cognitive Robotics and AI.

The team currently consists of Heinrich Mellmann, Marcus Scheunemann, Thomas Krause, Claas-Norman Ritter, Steffen Kaden, Peter Woltersdorf, Tobias Hübner, Benjamin Schlotter, Schahin Tofangchi, Maximilian Bielefeld, Alexander Berndt, and Carolin Matthie.

We have a long tradition within the RoboCup by working for the Four Legged League as a part of the GermanTeam in recent years, with which we won the competition three times. We started working with Naos in May 2008 and achieved the 4th place at the competition in Suzhou in the same year. In 2010, we simultaneously participated in the SPL and the 3D Simulation League for the first time with the same code. In the 3D Simulation, we won the German Open and the AutCup competitions and achieved the 2nd place at the RoboCup World Championship 2010 in Singapore. In 2011, we won the Iran Open competition in SPL and started a conjoint team *Berlin United* with the FUmanoids from Berlin who participated in the KidSize League. In

the world cup 2012 in Mexico, we won the technical challenge with an extension for the SimSpark Simulator, used in the *3D Simulation League*, to get closer to achieve our long-term goal to narrowing the gap between the simulation and real robots league.
With our efforts in these three leagues, we hope to foster the cooperation between them and enhance results in all of those leagues with perspective change. In cooperation with FUmanoids, we applied for a RoboCup project to investigate a common communication protocol to hold matches with different robot platforms and software in one team. Another RoboCup project of ours dealt with the topic of an extension for SimSpark for SPL. We informed about results of this extension during the symposium 2013 in Eindhoven. Our general research fields include agent-oriented techniques and Machine Learning with applications in Cognitive Robotics. Currently, we mainly focus on the following topics:

- Software architecture for autonomous agents (section 2)
- Narrowing the gap between simulated and real robots (section 4)
- Dynamic motion generation (section 7)
- World modeling (section 6)

The release of the NaoTH code base accompanying this report and the according documentation can be found under the following links:

Documentation: https://github.com/BerlinUnited/NaoTHDoc/wiki

Code: https://github.com/BerlinUnited/NaoTH

Chapter 2

Architecture

An appropriate architecture is the base of each successful software project. It enables a group of developers to work at the same project and to organize the solutions for their particular research questions. From this point of view, the artificial intelligence and/or robotics related research projects are usually more complicated than commercial product development, since the actual result of the project is often not clear. Since we use this project also in education, a clear organization of the software is necessary to achieve a fast familiarization with the software. Our software architecture is organized with the main focus on modularity, easy usage, transparency and convenient testing capabilities.

In the following subsections we describe the design and the implementation of different parts of the architecture. A detailed description of the principles we used can be also found in [10]

2.1 NaoSMAL

In our architecture we don't use the NAOqi API directly but use our own so-called NaoSMAL (Nao Shared Memory Abstraction Layer) NAOqi-module. This calls the DCM API of NAOqi[1] and makes it accessible for other processes via a shared memory interface. Thus we can implement our own code as a complete separated executable that has no dependencies to the NAOqi framework. The benefits are a safer operation of the Nao on code crashes (NaoSMAL will continue to run and ensures the robot will go in a stable position), faster redeploy of our binary without restarting NAOqi and a faster compilation since we have lesser dependencies.

[1] http://doc.aldebaran.com/1-14/naoqi/sensors/dcm-api.html

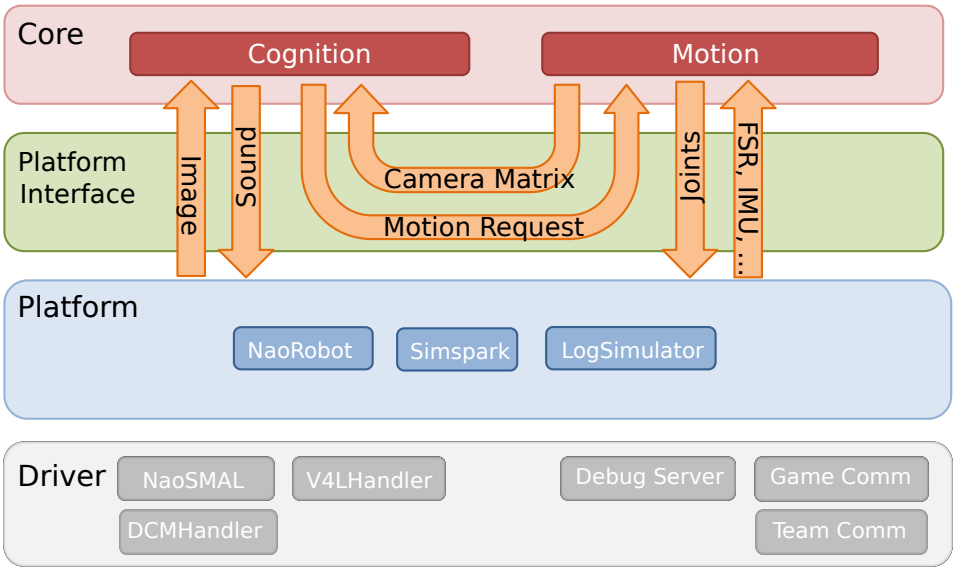


Figure 2.1: Platform Interface is responsible for data transferring and execution of the Cognition and Motion processes.

2.2 Platform Interface

In order to integrate different platforms, our project is divided into two parts: a platform independent one and platform specific one. The platform specific part contains code which is applied to the particular robot platform. We support the Nao hardware platform, the SimSpark simulator[2] and a logfile based simulator. While the platform specific part is a technical abstraction layer the platform independent part is responsible for implementing the actual algorithms. Both parts are connected by the *platform interface*, which transfers data between the platform independent and specific part (see Fig. 2.1).

[2] http://simspark.sourceforge.net/

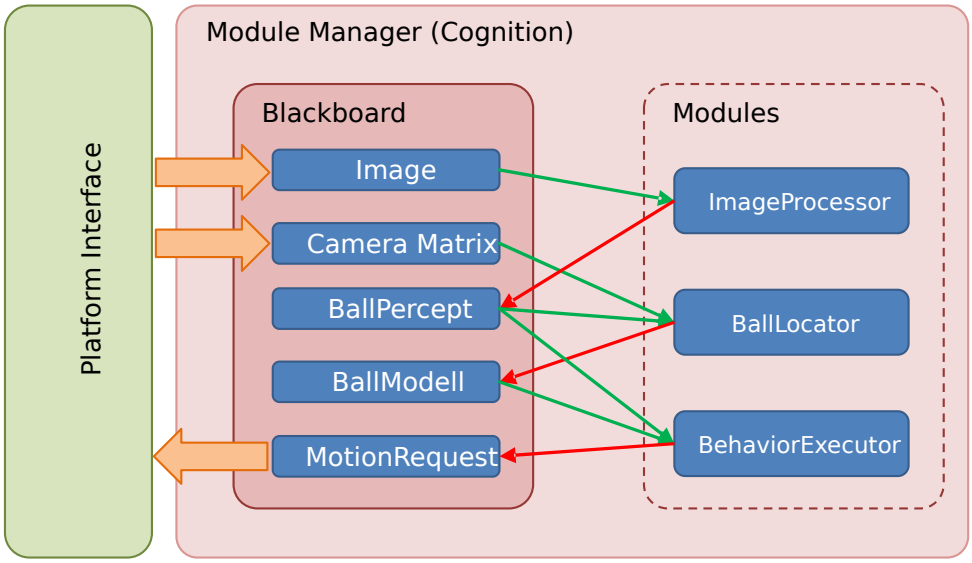


Figure 2.2: Overview about the different components of the module framework.

2.3 Module framework

Our module framework is based on a *blackboard architecture*. The framework consists of the following basic components:

Representation (objects carrying data and simple manipulation functions),

Blackboard (container storing representations as information units),

Module (executable unit, has access to the blackboard and can read and write representations),

Module Manager (manage the execution of the modules).

Figure 2.1 describes the interaction between this components. A module may *require* a representation, in this case it has a read-only access to it. A module *provides* a representation, if it has a writing access. In our design we consider only sequential execution of the modules, thus there is no handling

for concurrent access to the blackboard necessary. We decide which representation is required or provided due compilation time. Different modules can implement similar functionality and provide the same representations. You can configure which of the modules should be executed at runtime and it is also possible to dynamically change this for debugging purposes.

2.3.1 Example module

A module is a C++ class which inherits a base class which is created with the help of some macros defining the interface of the the module.

```
#ifndef _MyModule_H
#define _MyModule_H

#include <ModuleFramework/Module.h>
#include <Representations/DataA.h>
#include <Representations/DataB.h>

BEGIN_DECLARE_MODULE(MyModule)
 REQUIRE(DataA)
 PROVIDE(DataB)
END_DECLARE_MODULE(MyModule)

class MyModule: public MyModuleBase
{
public:
 MyModule();
 ~MyModule();

 virtual void execute();
};

#endif  /* _MyModule_H */
```

Listing 2.1: MyModule.h

The `MyModule` class inherits the `MyModuleBase` class which was defined with the `BEGIN_DECLARE_MODULE` macro. Each representation which is needed by the module is either declared as provided or required with the corresponding macro. After declaring a representation it is accessible with a getter function, which has the name of the representation prefixed with "get', e.g. `getDataA()` for the representation `DataA`. The actual implementation of the functionality of a module must be in the `execute()` function.

```
#include "MyModule.h"
```

```
MyModule::MyModule()
{
  // initialize some stuff here
}

MyModule::~MyModule()
{
  // clean some stuff here
}

void MyModule::execute()
{
  // do some stuff here
  getDataB().x = getDataA().y + 1;
}
```

Listing 2.2: MyModule.cpp

A representation can be any C++ class, it does not need to inherit any special parent class.

```
class DataA
{
public:
  DataA(){}

  int y;
};

class DataB
{
public:
  DataB(){}

  int x;
};
```

Listing 2.3: DataA.h/DataB.h

A module must be registered in the cognition process by including it in the file `NaoTHSoccer/Source/Core/Cognition/Cognition.cpp`.

```
#include "Modules/Experiment/MyModule/MyModule.h"
```

In the `init` method add the line:

```
REGISTER_MODULE(MyModule);
```

The order of registration defines the order of execution of the modules.

2.4 Serialization

As described in the Section 2.3 the core of the program is structured in modules which are responsible for different tasks like image processing, world modeling etc.. The modules communicate with each other through the *blackboard* by writing their results to *representations*. The *representations* are mainly pieces of data and have no significant functionality. These representation can be made *serializable*, which is mainly used in two cases: logging to a file and sending over the network for debug or monitoring reasons.

The backbone of the serialization framework is formed by the Google *Protocol Buffers*[3] (protobuf) library. For a representation to be serialized (which is described by a C++ class) an according protobuf message is defined. Please refer to the documentation page of protobuf for more details on this part. The serialization procedure is then performed in two steps: first the data is copied from the object which is to be serialized to the according message object; in the second step th message object is serialized by a protobuf serializer to a byte stream. The deserialization process works in reverse order. The second step is entirely done by the protobuf library. The copy procedure in the first step, however, has to be defined explicitly. This procedure is described in the `serialize()` and `decerialize()` functions of the template class `Serializer` which has to be specified for each representation to be serialized.

The following listings illustrate the whole code necessary for serialization of a representation. Listing 2.4 shows the header file *MyRepresentation.h* containing the declaration of the representation class `DataA` and the according specialization of the serializer `Serializer<DataA>`. Listing 2.5 contains the probobuf message for `DataA`. Listing 2.6 illustrates the implementation of the serialization functions in the file *MyRepresentation.cpp*.

```
#include <Tools/DataStructures/Serializer.h>

class DataA
{
public:
  DataA()
    :
    y(0),
    time(0.0)
  {}

  int y;
  double time;
```

[3] https://developers.google.com/protocol-buffers

```
};

namespace naoth {
template<>
class Serializer<DataA>
{
public:
  static void serialize(const DataA& representation, std::
     ostream& stream);
  static void deserialize(std::istream& stream, DataA&
     representation);
};
}
```

Listing 2.4: MyRepresentation.h

```
package mymessages;

message DataA {
  required double time = 1;
  required int32 y = 2;
}
```

Listing 2.5: messages.proto

```
#include "MyRepresentation.h"
#include "Messages/mymessages.pb.h"
#include <google/protobuf/io/zero_copy_stream_impl.h>

using namespace naoth;

void Serializer<DataA>::serialize(const DataA& data, std::
    ostream& stream)
{
  // create a new message
  messages::DataA msg;

  // copy data from the representation to the message
  msg.set_y(data.y);
  msg.set_time(data.time);

  // serialize the message to stream
  google::protobuf::io::OstreamOutputStream buf(&stream);
  msg.SerializeToZeroCopyStream(&buf);
}
```

```
void Serializer<DataA>::deserialize(std::istream& stream,
    DataA& data)
{
  // create a new message
  messages::DataA msg;

  // decerialize the message from stream
  google::protobuf::io::IstreamInputStream buf(&stream);
  msg.ParseFromZeroCopyStream(&buf);

  // copy data from the message to the data
  data.y = msg.y();
  data.time = msg.time();
}
```

Listing 2.6: MyRepresentation.cpp

Chapter 3

Debugging and Tools

In order to develop a complex software for a mobile robot, we require means for high-level debugging and monitoring (e. g., visualization of the robot's posture or its position on the field). Since we do not exactly know which kind of algorithms will be debugged, there are two aspects of high importance: accessibility at runtime and flexibility. The accessibility of the debug construct is realized based on our communication framework. Thus, they can be accessed at runtime by using visualization software like RobotControl, as shown in Figure 3.1).

3.1 Concepts

Some of the ideas were evolved from the GT-Architecture [12]. The following list illustrates some of the debug concepts:

debug request (activates/deactivates code parts),

modify allows modification of a value (in particular local variables)

stopwatch measures the execution time

parameter list allows to monitor and modify lists of parameters

drawings allows visualization in 2D/3D; thereby it can be drawn into the image or on the field (2D/3D)

plot allows visualization of values over time

As already mentioned, these concepts can be placed at any position in the code and can be accessed at runtime. Similar to the module architecture,

the debug concepts are hidden by macros to allow simple usage and to be able to deactivate the debug code at compilation time, if necessary.

In order to use a debug request in the code you have to register it once with the DEBUG_REQUEST_REGISTER macro:

```
DEBUG_REQUEST_REGISTER("My:Debug:Request", "Description of
    the debug request", true);
```

After that, you can use the DEBUG_REQUEST macro to wrap code that should be only executed when the debug request is active.

```
DEBUG_REQUEST("My:Debug:Request",
    std::cout << "This code is not executed normally" << std
        ::endl;
    ++c;
);
```

MODIFY works in a similar way, but does not need any registration. By, for example, wrapping a variable and defining an identifier, this variable can be changed later from RobotControl.

```
double yaw  = 0;
MODIFY("BasicTestBehavior:head:headYaw_deg",yaw);
```

In addition to these means for individual debugging, there are some more for general monitoring purposes: the whole content of the blackboard, the dependencies between the modules and representations, and execution times of each single module. The Figure 3.1 illustrates visualizations of the debug concepts. In particular a field view, 3D view, behavior tree, plot and the table of debug requests are shown.

3.2 RobotControl Dialogs

The various debugging possibilities are organized in different dialogs. The following list consists of our most used RobotControl Dialogs.

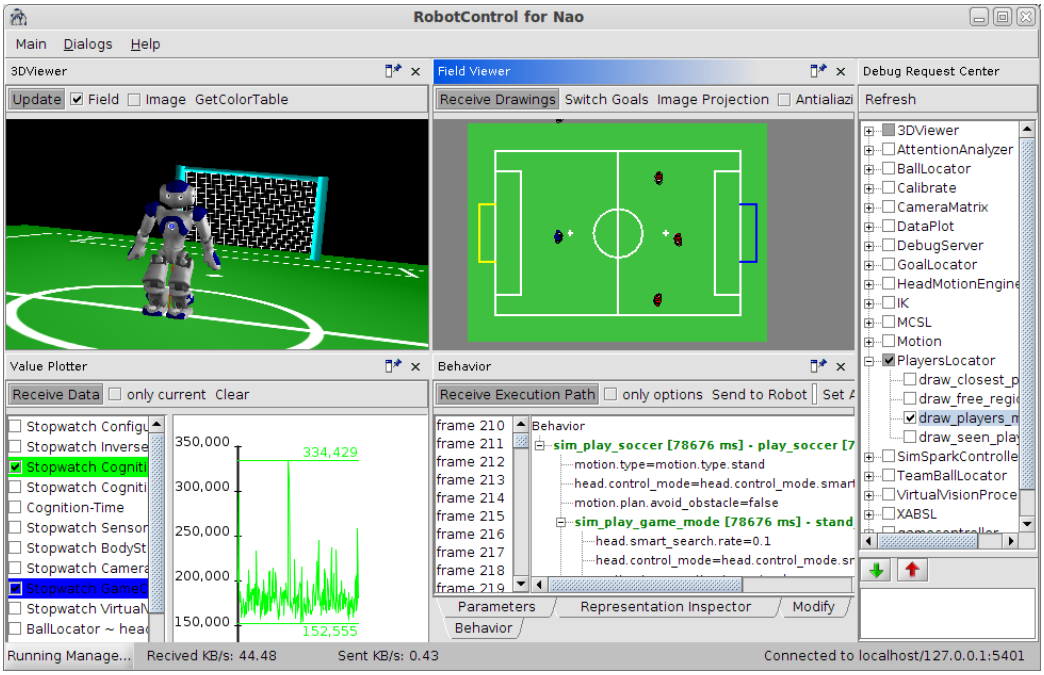
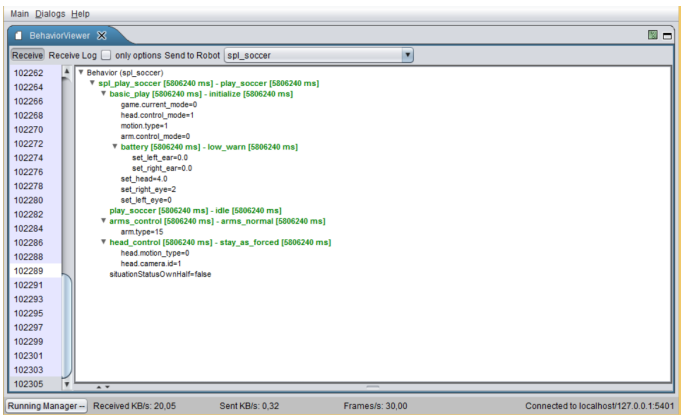


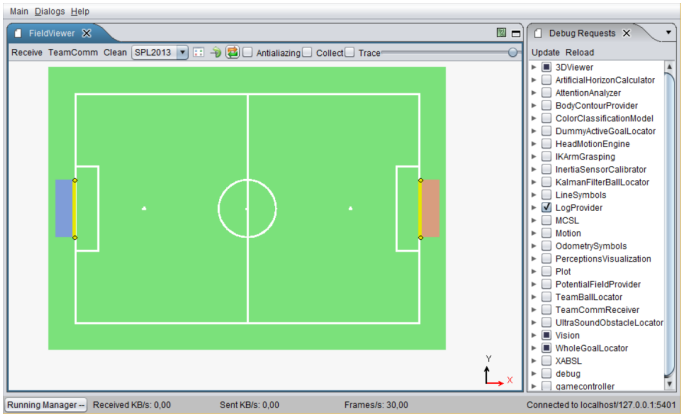
Figure 3.1: The RobotControl program contains different dialogs. The 3DViewer (top left) is used to visualize the current state of the robot; the Value Plotter dialog (bottom left) plots some data; the Field Viewer dialog (top center) draws the field view; the Behavior dialog (bottom center) shows the behavior tree; the Debug Request Center dialog (right) is for enabling/disabling debug requests.

Behavior Viewer



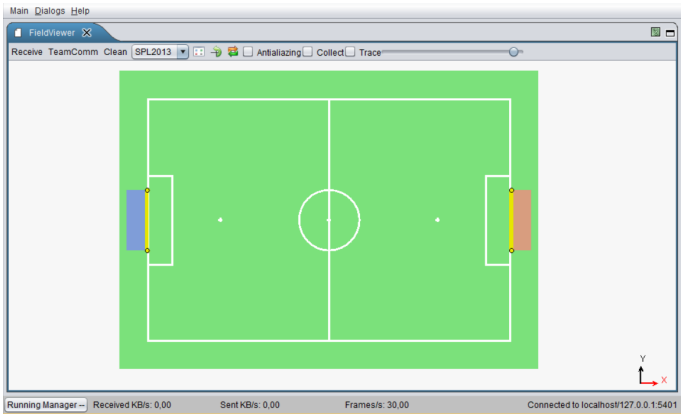
Shows the behavior tree for the current behavior. The compiled XABSL behavior needs to be sent to the robot first and then an agent can be selected to be executed. With 'Add Watch' you can track XABSL input and output symbols.

Debug Requests



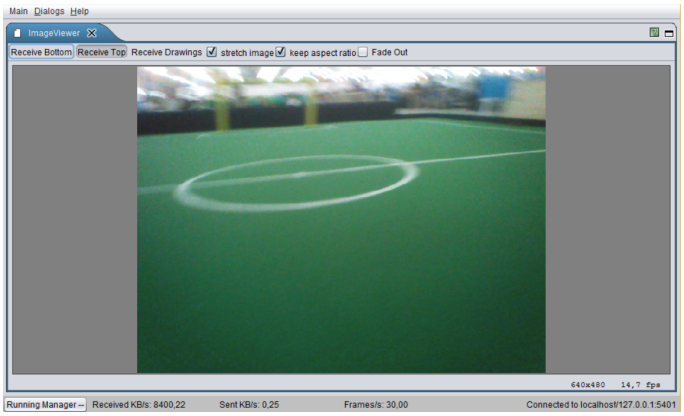
(De-)activates the debug request code. Usually a debug request draws something on the field viewer or on the camera images. For further information about individual debug requests, have a look at the source code.

Field Viewer



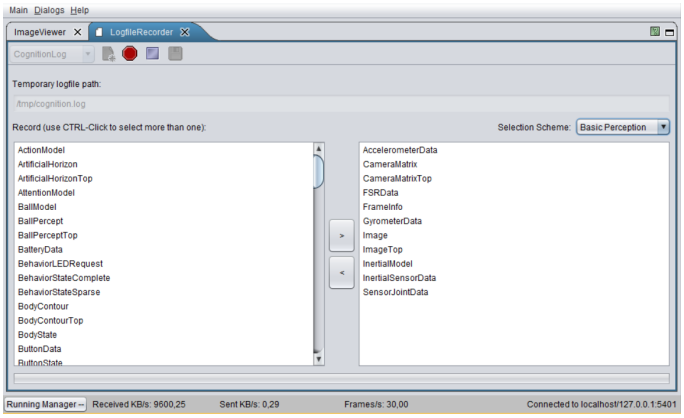
There are views for different field sizes and a local view. Certain debug requests draw on these views. For example, you could draw the robots' positions on the field by activating the corresponding debug request.

Image Viewer



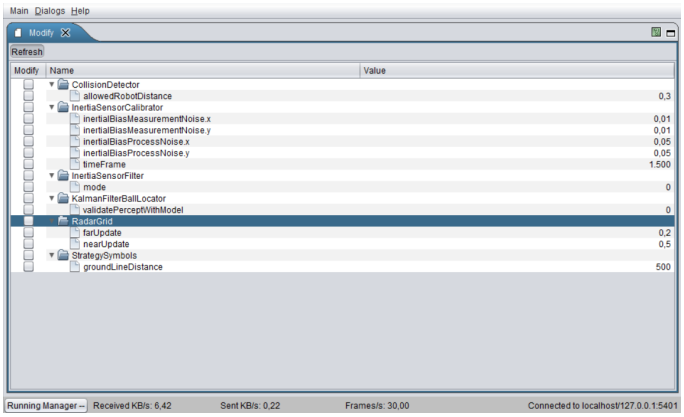
Can show the top and bottom images. There are debug requests that draw on the camera images, if they are active.

Logfile Recorder



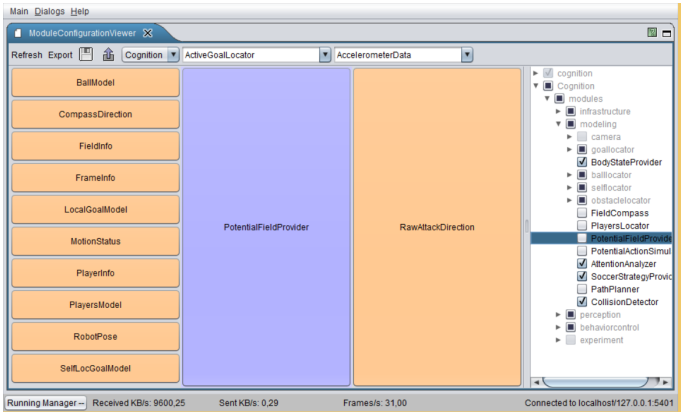
Records a logfile with selected data.

Modify



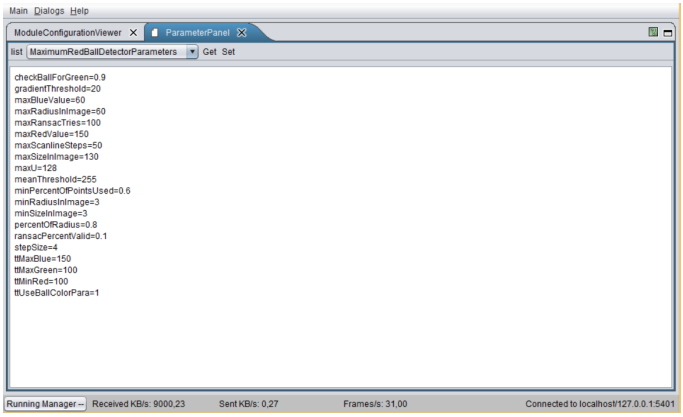
The Modify macro allows changing values of variables declared within this macro at runtime.

Module Configuration Viewer



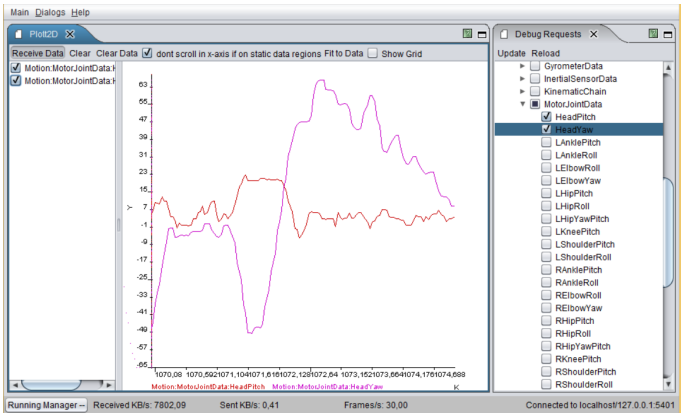
Shows which modules are currently (de-)activated. Also indicates, which other modules are required (left) and provided (right) by each module.

Parameter Panel



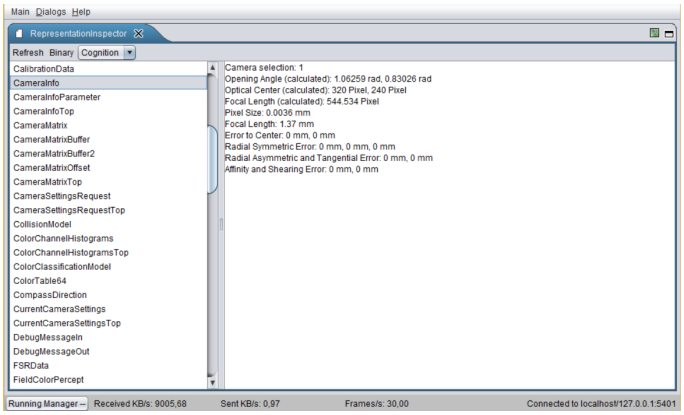
Shows parameters defined in our configuration files. It is possible to change the values at runtime. The variables must be registered as parameters in the code.

Plot 2D



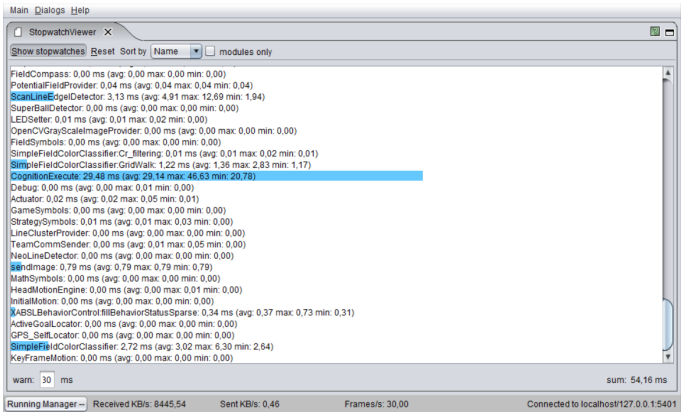
Shows plots activated by plot debug requests.

Representation Inspector



Shows the data that is written to the blackboard by each representation.

Stopwatch Viewer



Shows the execution time for each module.

3.3 NaoSCP

NaoSCP is a setup and deployment tool. It has primarily three tasks: (1) initialize a new robot, e.g., copy libraries and scripts, (2) set the network configuration and (3) deploy naoth binary and configurations to the robot. All these tasks can be easily done on a command line as well, the main aim for designing *NaoSCP* were simplification of the deployment process, ensured backup of deployed binaries and reduction of the chance of mistakes during setup in critical situations, e.g., before a game at the world championship.

3.3.1 Deployment Procedure

The deployment procedure is divided in two phases: in the first phase a deployment directory is assembled containing all files and configurations to be deployed as well as a deployment shell script; in the second phase this package is copied to the robot and the deployment script is executed. Using the *Write to Stick* button the deployment directory can be written to a target path, e.g., a USB stick, or sent directly to the robot via scp using the *Send to Robot* button. When a USB stick with a deployment directory is connected to the robot, it is mounted automatically and the setup script is executed. The begin and end of the deployment procedure are indicated by according sounds. This way is preferred when deploying software on several robots, e.g., setting up a team before a game.

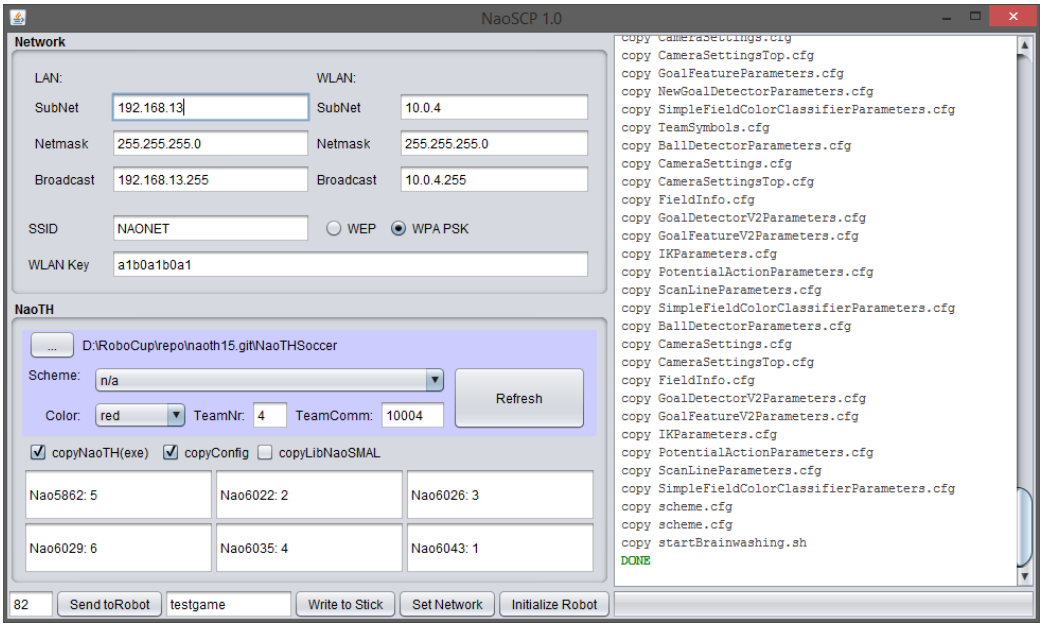


Figure 3.2: NaoSCP user interface. The log panel on the right display status of the deployment process. The left side contains the panels for the configuration of the deployment / setup process: *Network* configures the network setup; *NaoTH* is used to adjust the configuration relevant for the deployed binary, e.g., player numbers. The buttons in the left bottom tool bar trigger particular deployment and setup actions like writing the network configuration the the robot or copying a new binary to a deployment USB stick.

3.3.2 Usage Remarks

The following describes the particular components of the NaoSCP user interface as illustrated in the Figure 3.2.

Log Window (right) gives prints information regarding the progress of the deployment process, e.g., copied files, connection errors and such.

Network configuration (top left) is used to setup the LAN and WLAN;

NaoTH dialog (left) configures the deployment of the game binary and contains such things like the path to the source where the binaries can be found, used configuration scheme scheme and player numbers for each robot based on their serial number;

Action toolbar (bottom left) contains the buttons for the four different deployment / setup actions: *Send to Robot* deploy the complied code

and configuration to a particular robot via network. The text field left of the button defines the last byte of the ip address of the target robot. The network configuration from the dialog *Network* is used to determine the complete address. In this particular example the LAN target address would be 192.168.13.82. Thereby LAN is tried first and in case of failure WLAN is tried; *Write to stick* writes the deployment directory to a USB stick. If the stick already contains a deployment directory, a backup version of it is created. The text field left to the button hold an optional tag, which is used to organize the backups on th stick; *Set Network* configures the robots network according the the settings in the dialog *Network*; *Initialize Robot* will initialize a new robot, e.g., after a factory reset. This action will copy additional libs, configure the NaoQi modules, necessary starting scripts for binaries and for automatic mounting and running of USB sticks. Additionally the network is configured and the binary is deployed like previously described;

Chapter 4

Simulation

As a common experience, there are big gaps between simulation and reality in robotics, especially with regards to basic physics with consequences for low level skills in motion and perception. There are some researchers who have already tried to narrow this gap, but there are only few successful results so far. We investigate the relationships and possibilities for methods and code transferring. Consequences can lead to better simulation tools, especially in the 3D Simulation League. At the moment, we use the SimSpark simulator from the 3D Simulation League with the common core of our program, see Figure 4.1. As already stated, therewith, we want to foster the cooperation between the two leagues and to improve both of them.

When compared to real Nao robots, some devices are missing in the SimSpark, as LEDs and sound speakers. On one hand, we extended the standard version of SimSpark by adding missing devices like camera, accelerometer, to simulate the real robot. On the other hand, we can use a virtual vision sensor which is used in 3D simulation league instead of our

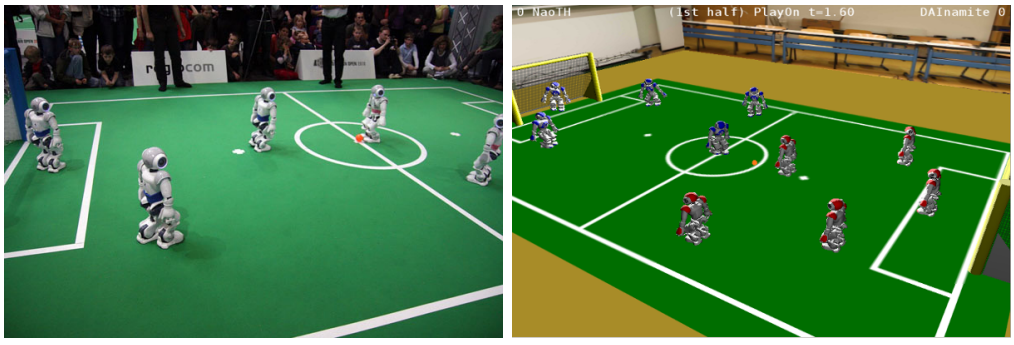


Figure 4.1: NAO robots run in Standard Platform League(left) and 3D Simulation League(right).

image processing module. This allows us to perform isolated experiments on low level (e. g., image processing) and also on high level (e. g., team behavior). Also we developed a common architecture [10], and published a simple framework allowing for an easy start in the Simulation 3D league.

Our plan is to analyze data from sensors/actuators in simulation and from real robots at first and then to apply machine learning methods to improve the available model or build a good implicit model from the data of real robot. Particularly, we plan to:

- improve the simulated model of the robot in SimSpark,
- publish the architecture and a version of SimSpark which can be used for simulation in SPL,
- transfer results from simulation to the real robot (e. g., team behavior, navigation with potential field).

So far, we have developed walking gaits through evolutionary techniques in a simulated environment [6, 5]. Reinforcement Learning was used for the development of dribbling skills in the 2D simulation [14], while Case Based Reasoning was used for strategic behavior [4, 2]. BDI-techniques have been investigated for behavior control, e. g., in [1, 3].

Chapter 5

Visual Perception

In order to realize a complex and successful cooperative behavior it is necessary to have a appropriate model of the surrounding world. Thus, one of the main focuses of our current research is the improvement of the perceptional abilities of the robot and its capabilities to build a world model.

Actually we do not use fixed color class based methods and color tables anymore. The main tasks of our vision system is detecting the field (including field borders), the field lines, the ball and the goal. Others, like a visual robots detection are not implemented yet. We detect the objects in a specific order, which makes some computations easier for each following object detector. First, we compute some statistical informations for each color channel and use this to classify the fields color. This approach is based on ideas from [11]. After that, we use this to validate, that the goal posts are grounded in the field, that lines are within the field, that a ball must be within the field and to calculate the field borders.

5.1 HistogramProvider

This module scans the top and bottom image, to calculate the statistics for each color channel. Since we use the YUV color space, this module calculates three histograms for the top and three for the bottom image. To calculate the histograms only every 6th pixel is used. In other words, the histogram is taken from an subsampled image, which is six times smaller. This does not change much for the distribution information of colors of the original image. The statistics are similar except for a small error.

5.2 SimpleFieldColorClassifier

In this module we estimate the field color as a cubic area in the YUV color space. For this we use statistical information of the distribution of gray level values, of each color channel. The basic assumption is, that in a robot soccer environment both (bottom and top) images are mostly covered by the field. In [11] this is the main assumption too.

Figure 5.1: The color of the field, which is covering most of the image pixels (left), is estimated by using statistics (right).

Since this algorithm of [11] has some problems, we modified it to cover our needs. Our approach is slightly different. We do not correct vignetting. We use statistical information of more than one succeeding frame. As first step we constrain the brightness. And we use only every 6th pixel to calculate the color channel statistics. One disadvantage is, that we sometimes have to tune the parameters to get good results, but the classification algorithm is still able to adapt to changing conditions.

5.3 ScanLineEdgelDetector

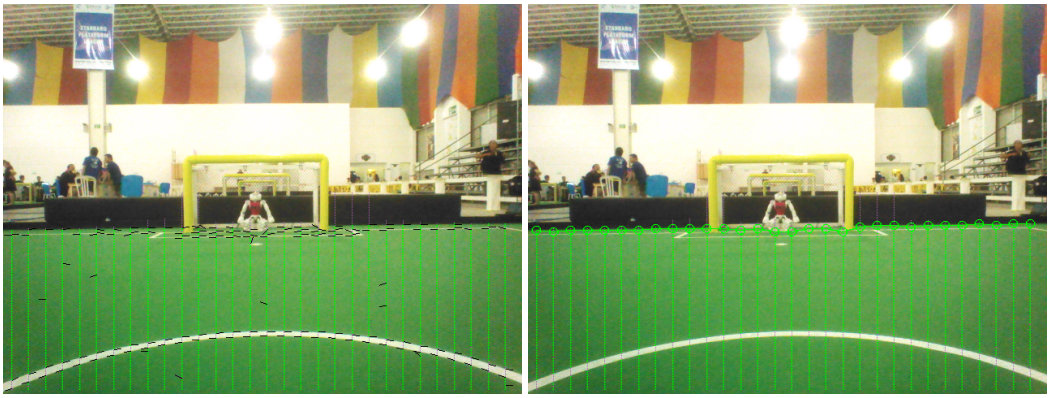


Figure 5.2: With top to down scanlines [green lines] the edges of possible field lines [black lines] including their orientation are detected (left) and the last field colored points are assumed as endpoints of the field [green circles] (right).

With this module we detect field line border points and estimate some points of the field border. To do this, we use scanlines, but only vertical ones. Along every scanline jumps are detected in the Y channel, using a 1D-Prewitt-Filter. A point of the field lines border is located at the maximum of the response of that filter. We estimate with two 3x3-Sobel-Filters (horizontal and vertical) the orientation of the line. With the result of the field color classification we detect along every scanline a point, which marks the border of the field.

5.4 FieldDetector

With the field border points, estimated with the *ScanLineEdgelDetector*, we calculate for each image a polygon, which is representing the border of the field in the image.

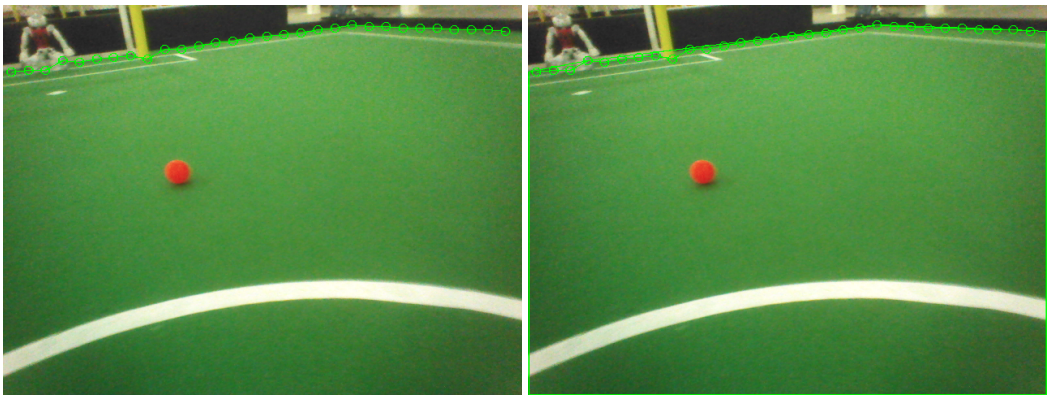


Figure 5.3: The endpoints provided by the *ScanLineEdgelDetector* (left) are used to calculate the field border (right).

5.5 LineGraphProvider

This module clusters neighbouring line border points, detected by *ScanLineEdgelDetector*.

5.6 GoalFeatureDetector

This module is the fist step of the goal post detection procedure. To detect the goal posts we scan along the horizontal scan lines parallel to the artificial horizon estimated in *ArtificialHorizonProvider*. Similar to the detection of the field line described in Section 5.3 we detect edgels characterized by the jumps in the pixel brightness. These edgels are combined pairwise to goal features, which are essentially horizontal line segments with rising and falling brightness at the end points. Figure 5.4 illustrates the scan lines as well as detected edgels (left) and resulting goal post features (right).

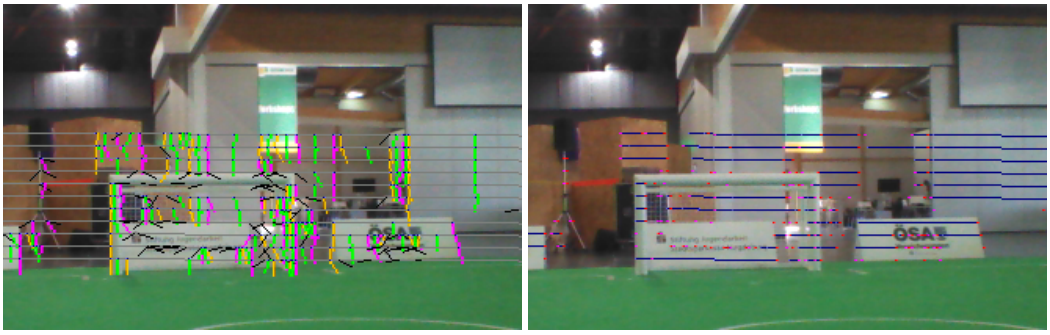


Figure 5.4: The scan lines [grey lines] above and below the estimated horizon are used to detect the goal post border points and the orientation of the corresponding edges [colored and black segments] (left). The results are features of possible goal posts [blue line segments with red dots] (right).

5.7 GoalDetector

The *GoalDetector* clusters the features found by the *GoalFeatureDetector*. The main idea here is, that features, which represent a goal post, must be located underneath of each other. We begin with the scan line with the lowest y coordinate and go through all detected features. Than the features of the next scan lines (next higher y coordinate) are checked against these features. Features of all scan lines, which are located underneath of each other, are collected into one cluster. Each of this clusters represents a possible goal post.

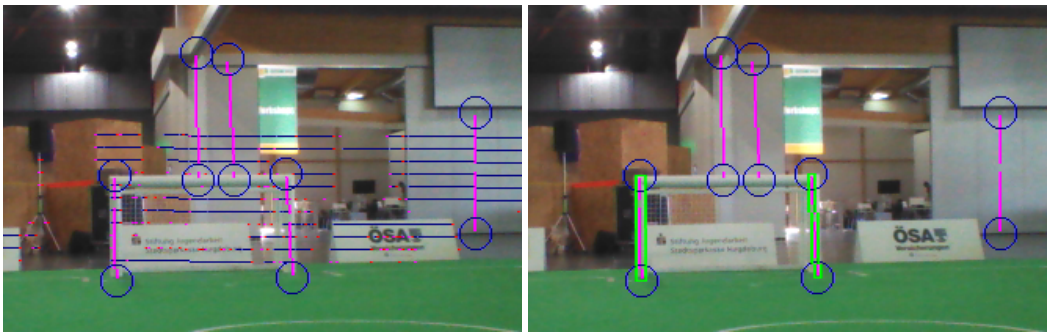


Figure 5.5: Goal features detected as described in 5.6 are clustered to form candidates for the goal posts (left). These candidates are evaluated regarding expected dimensions as well as their relation to the field. The candidates fulfilling all necessary criteria are selected as goal post percepts (right green boxes).

From the features of a cluster, the orientation of the possible goal post is estimated and used to scan up and down along the estimated goal post. This is done to find the foot and the top point of that goal post. A goal post is seen as valid, if its foot point is inside of the field polygon as described in the Section 5.4. Using the kinematic chain the foot point is projected into the relative coordinates of the robot. Based on this estimated position the expected dimensions of the post are projected back into the image. To be accepted as a goal post percept a candidate cluster has to satisfy those dimensions, i.e., the deviation should not exceed certain thresholds. The Figure 5.5 illustrates the clustering step and the evaluation of the candidate clusters. Although there seem to be a considerable amount of false features, both posts of the goal are detected correctly.

5.8 BallDetector

This module scans all pixels in the image, which are covered by the field.

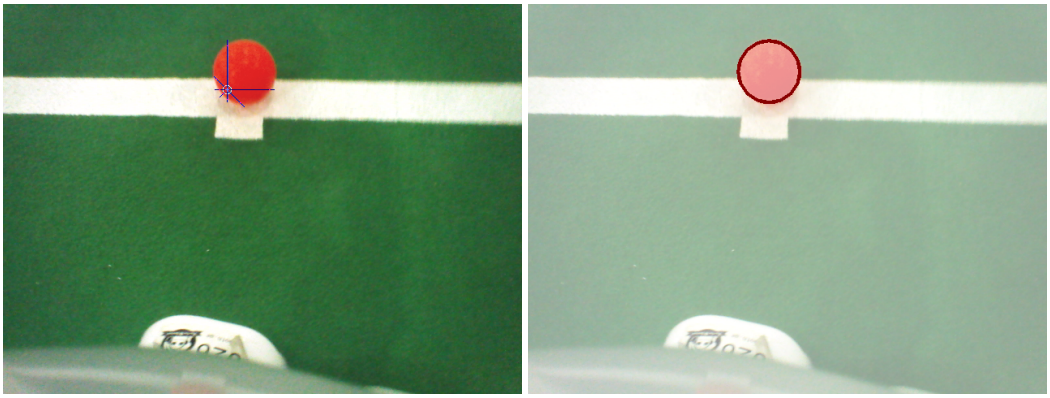


Figure 5.6: Scan lines are starting from the most red point found by the *BallDetector* (left) and the ball shape is estimated using the border pixels (right) found at the scan lines.

The pixel with the most red appearance is taken as a possible central point inside the ball. This pixel must be more red than any field colored pixel. Eight scan lines, beginning in this pixel and directing in eight different directions, are used the find border pixels of the assumed ball. The eight scan directions cover equally distributed 360 degrees. The resulting border pixels are used to estimate a circle, which represents the estimated ball shape in the image. This estimated ball shape in the image is projected to the ground and checked for its size.

5.9 Green Detection [outlook]

This section describes a new approach to classify the field color which has not been used at the RoboCup 2015.[1] This constitutes the fist step in the attempt for a automatic field color detection. Thereby we analyze the structure of the color space perceived by the robot NAO and propose a simple yet powerful model for separation of the color regions, whereby green color is of a particular interest.

To illustrate our findings we utilize a sequence of images from recorded by a robot during the Iran Open 2015. Figure 5.7 (left) shows a representative image from this sequence. To analyze the coverage of the color space

[1]For the first time this approach has been presented in November 2015 at the RoHOW workshop in Hamburg.

we calculate two color histograms over the whole image sequence. In the Figure 5.8 (left) you can see the uv-histogram, which is basically a projection of the yuv space onto the uv plane. The light green points indicate the frequency of a particular uv-value (the brighter the more). One can clearly recognize three different clusters: white and gray colors in the center; green cluster oriented towards the origin; and a smaller cluster of blue pixels in the direction of the u-axis which originate from the boundaries around the field. For the second histogram we choose a projection plane along the y-axis and orthogonal to the uv-plane which is illustrated in the Figure 5.8 (left) by the red line. This plane is chosen in a way to illuminate the relation between the gray cluster in the center and the green cluster. Figure 5.8 (middle) illustrates the resulting histogram. Here we clearly see the gray and the green cluster.

Figure 5.7: (left) Example image from the Iran Open 2015. (right) Pixels classified as green are marked green; pixels with too low chroma marked red;

From these two histograms we can make following observations: all colors seem to be concentrically organized around the central brightness axis , i.e., gray axis $(128, 128, y)$, which corresponds to the general definition of the yuv space; the colors seem to be pressed closer to the gray axis the darker they are. In particular all colors below a certain y-threshold seem to be collapsed to the gray axis. So we can safely claim that for a pixel (y, u, v) always holds $y = 0 \Rightarrow u, v = 128$. On the contrary the spectrum of colors gets wider with the rising brightness. Speculatively one could think that the actual space of available colors is a hsi-cone fitted into the yuv-cube. The collapse of the colors towards the gray axis might be explained by an underlying noise reduction procedure of the camera.

Based on these observations we can divide the classification in two steps: (1) separate the pixels which do not carry enough color information, i.e., these

which are too close to the gray axis. Figure 5.8 (middle) illustrates a simple segmentation of the gray pixels with a cone around the center axis illustrated by the red lines; (2) classify the color in the projection onto the uv-plane. Figure 5.8 (right) shows the uv-histogram without the gray pixels. Red lines illustrate the separated uv-segment which is classified as green. This way we ensure independence from brightness. The equation 5.3 illustrates the three conditions necessary for a pixel (y, u, v) to be classified as green. The five parameter are $b_o \in [0, 255]$ the back cut off threshold, $b_m, b_M \in [0, 128]$ with $b_m < b_M$ the minimal and the maximal radius of the gray cone, and finally $a_m, a_M \in [-\pi, \pi]$ defining the green segment in the uv-plane.

$$(u - 128)^2 + (v - 128)^2 > \max\left(b_m, b_m + (b_M - b_m) \cdot \frac{y - b_o}{255 - b_o}\right) \tag{5.1}$$

$$atan2(u - 128, v - 128) > a_m \tag{5.2}$$

$$atan2(u - 128, v - 128) < a_M \tag{5.3}$$

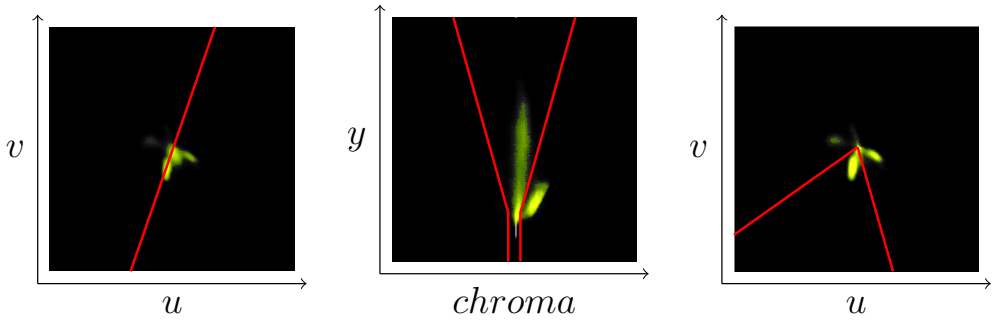


Figure 5.8: (left) UV-histogram for a log file taken at the Iran Open 2015. Red line illustrates the projection plane along the green region for the Y-Chroma histogram (middle); (middle) Y-Chroma-histogram along the projection plane illustrated in (left) figure. Red lines illustrate the gray-cone, i.e., area with not enough color information to be classified as a particular color; (right) UV-Histogram without pixel falling into the gray-cone as illustrated in the (middle) figure. Red lines illustrate the segment to be classified as green.

The classification itself doesn't require an explicit calculation of histograms. At the current state it's a static classification depending on five parameters to define the separation regions for the gray and green colors. These parameters can be easily adjusted by inspecting the histograms as

shown in the Figure 5.8 and have proven to be quite robust to local light variation.

The structure of the color space depends of course largely on the adjustments of the white balance. We suspect a deviation from a perfect white balance adjustment results a tilt of the gray cluster towards blue region if it's to cool and towards red if it's to warm. The tilt towards blue can be seen in the Figure 5.8 (middle). This might be a queue for an automatic white balance procedure which would ensure an optimal separation between colored and gray pixels. The green region shifts around the center depending on the general lighting conditions, color temperature of the carpet and of course white balance. In the current example the green tends rather towards the blue region. Tracking these shifts might be the way for a fully automatic green color classifier which would be able to cover the variety of the shades to enable a robot to play outside.

Chapter 6

Modeling

In order to realize a complex and successful cooperative behavior it is necessary to have a appropriate model of the surrounding world. In our approach we focus on local models of particular aspects of the environment. In this section we present two local models: a compass and a goal model.

6.1 Probabilistic Compass

We estimate the orientation of the robot on the field based on the detected line edgels utilizing the fact, that all field lines are either orthogonal or parallel to the field. Based on the orientations of the particular projected edgels it is possible to estimate the rotation of the robot up to the π symmetry. We calculate the kernel histogram over the orientations of the particular projected edgels, i.e., edgels in the local coordinates of the robot. To utilize the symmetry of the lines we use sin as distance measure. Let $(x_i)_{i=1}^n$ be the set of edgel orientations. We calculate the likelihood $S(x)$ for the robot rotation $x \in [-\pi, \pi)$ as shown in the equation 6.1.

$$S(x) = \sum_{i=1}^{n} \exp\left\{-\frac{\sin^2(2 \cdot (x - x_i))}{\sigma^2}\right\} \tag{6.1}$$

This compass is calculated in each frame where enough edgels have been detected. It has shown to be robust regarding outliers, e. g., when some edgels are detected in a robot. It can be directly used to update the likelihood of particles in the self locator. Figure 6.1 shows a set of edgels detected in a particular frame on the left side. On the right side the according histogram is plotted.

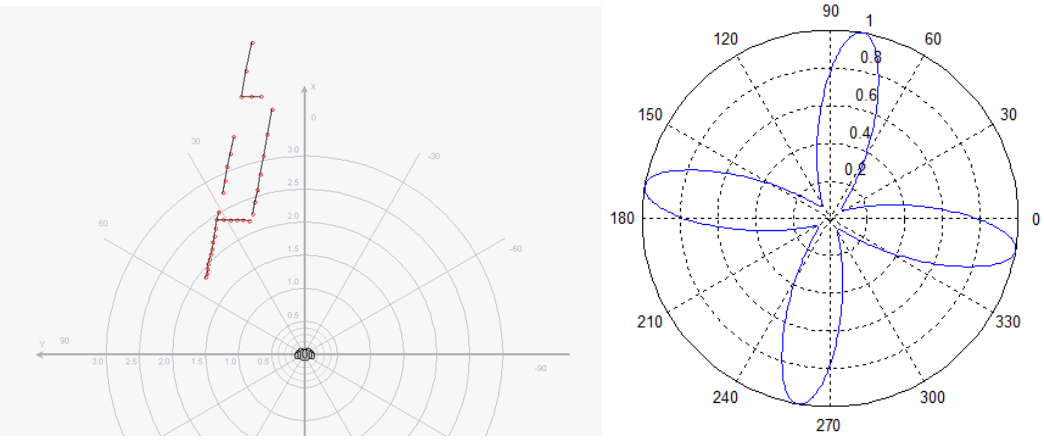


Figure 6.1: Left figure visualizes the edgel graph in local coordinates of the robot in a particular frame. Right illustrates the kernel histogram over the orientations of edgels shown left, calculated with formula 6.1.

6.2 Multi-Hypothesis Goal Model (MHGM)

In this section we describe a multi-hypothesis approach for modeling a soccer goal within the RoboCup context. The whole goal is rarely observed and we assume the image processing to detect separate goal posts. So we represent the goal by its corresponding posts. To reduce complexity of the shape of uncertainty we model the separate goal posts in local robot coordinates. The ambiguous goal posts are tracked by a multi-hypothesis particle filter. The actual goal model is extracted from the set of post hypotheses.

The joint uncertainty can be subdivided in *noise*, *false detections* and *ambiguity*. Each of this components is treated separately in our approach. The multi-hypothesis filter has to take care of noise and false detections, but it does not resolve the ambiguity of the goal posts. Instead, all occurring goal posts are represented by corresponding hypotheses and the ambiguity is solved on the next level when the goal model is extracted. Particle filters are great in filtering noise and are shown to be very effective for object tracking. To deal with sparse false positives we introduce a delayed initialization procedure. We assume a false positive to result in an inconsistency, i. e., it cannot be confirmed by any existing goal post hypothesis. In this case the percept is stored in a short time buffer for later consideration. This buffer is checked for clusters, in case a significant cluster of goal post percepts accumulated during a short period of time, a new hypothesis is initialized based on this cluster. The dense false detections result in post hypotheses, which is later ignored while extracting the goal.

More detailed description of the algorithm as well as the experimental results can be found in [13].

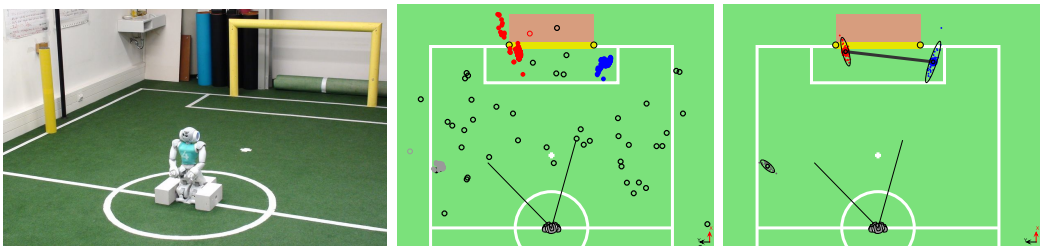


Figure 6.2: The left figure illustrates the experiment setup. The robot faces the goal and an additional goal post is placed to its right side. From the object recognition perspective, this post is identically to the *real* goal posts. The figure in the center visualizes all percepts collected during the course of the experiment. The full circles illustrate perceived goal posts, whereby their color indicates the classification by the MHGM: red - left post, blue - right post, gray - unknown post, black - none (percept buffer). The circles with holes stand for artificially generated sparse false positive perceptions. The right figure illustrates a snapshot of the state modeled by the MHGM at the end of the experiment. Drawn are the particle filter representing the goal posts with corresponding deviations as well as the extracted goal model. Similar to the figure in the center, the colors of the particles indicate the classification of the hypotheses.

6.3 Internal Models for Action Selection

The robot is capable of different kicks and should given a particular situation, e.g., the robot's position, the position of the ball and obstacles, determine which kick is the optimal kick to perform in this situation. A naive geometric solution which selects a kick based on the robot's direction towards the opponent goal does not account for uncertainty of the actual execution of the kick. Furthermore the distance of the kick is not considered in this approach. An improved kick selection algorithm was developed which is based on a forward simulation of the actions. Thereby each possible kick is simulated and the best kick is chosen based on the outcome, i.e., the position of the ball after the kick. Uncertainty and additional constraints can be integrated in a straight forward way.

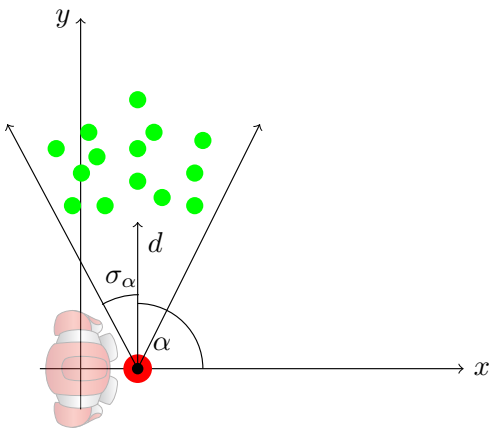


Figure 6.3: Distance d as defined by the velocity and friction parameters. Angle α from walking direction. Standard deviations for velocity and angle. This example shows a sidekick.

6.3.1 Definition of an Action

An Action is a set of parameters which describe a probability distribution of the possible ball location after the execution of a kick. Currently there are 4 kicks, two forward kicks and two sidekicks as well as the special case turn around the ball. The probability distribution is modelled as a gaussian distribution. The parameters which describe the distribution for one action are velocity, angle and their standard deviations.

6.3.2 Determine the parameters

To calculate the initial velocity of a kick the distance the ball rolled after a kick was measured in an experiment. By using the stopping distance formula the initial velocity of one kick can be calculated by

$$v_0 = \sqrt{d \cdot 2c_R \cdot g} \tag{6.2}$$

where v is the initial velocity of the ball. c_R the rolling resistance coefficient and g the gravitational constant. The mean of v_0 of multiple repetitions defines the initial velocity of this action. The standart deviation of the repetitions defines the standart deviation for the velocity of the kick. The parameter for angle is predefined for every action, e.g., it's zero for forward kicks and 90 degrees for left sidekick. The standart deviation for the angle is the standart deviation of the angle measurements from the previous experiment. The coefficient of friction is calibrated to a real surface from rolled

distances of the ball rolling on this surface with a known initial velocity. For this, we performed multiple experiments with an inclined plane starting at different heights. From these heights we could determine the initial potential energy of the ball, which was converted to kinetic energy by rolling down the inclined plane. At the end of the inclined plane (not taking into account the friction of the inclined plane), the initial velocity of the ball could thus be determined. We then measured the distance in multiple experiments. By transposing the rolling distance formula the rolling resistance coefficient can be calculated.

$$c_R = \frac{1}{2} \cdot \frac{v_0^2}{g \cdot d} \tag{6.3}$$

where v_0 is the starting velocity, g the gravitational constant, and d the total distance the ball traveled. The mean of the calculated coefficients is used as the rolling resistence coefficient for the other calculations. In the algorithm the position of the ball after the execution of an action is needed. To calculate this, the formula is transposed to calculate the distance the ball rolls after the execution of an action:

$$d = \frac{v_0^2}{2c_R \cdot g} \tag{6.4}$$

where v is the initial velocity of the ball. c_R the rolling resistance coefficient and g the gravitational constant. Figure 6.3 shows a resulting end position cloud of a hypothetical kick. The end points are calculated by drawing a sample from both the angle and kick speed distribution and plugging these values in equation 6.4. For detail, refer to section 6.3.3.

6.3.3 The algorithm

The whole simulation is divided into three steps: simulate the consequences, evaluate the consequences and decide the best action.

Simulating the consequences

Each action is simulated a fixed number of times. The resulting ball position of one simulation for an action is referred to as particle. The position of the particles are calculated according to the parameters of the action with applied standard deviations as shown in figure 6.3. The algorithm checks for possible collisions with the goal box and in case there are any the kick distance gets shortened appropriately. Collisions with the obstacle model are handled the same way.

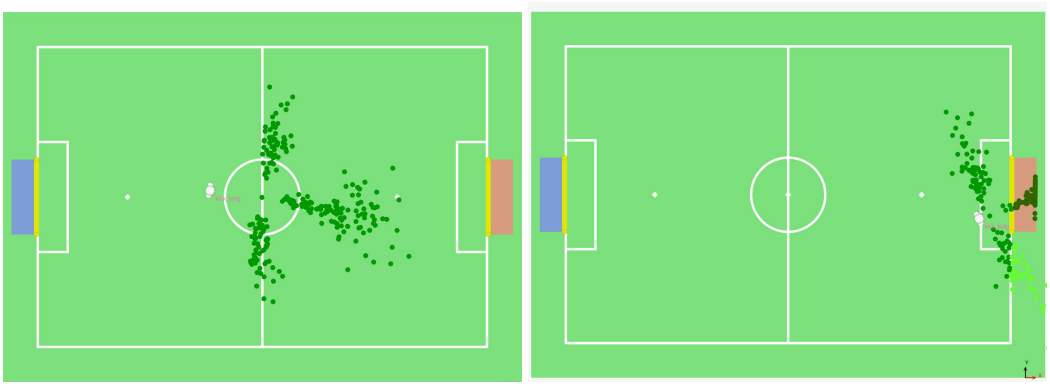


Figure 6.4: Left: All Actions with 30 particles each are simulated from a kickoff position. Right: The dark green particles result in a goal. The bright green particles are outside the field. In both cases the algorithm calculated that the long forward kick is the best action.

Evaluation

Each particle is sorted in different categories based on where on the field it is, e.g., inside the field, inside the own Goal, outside the field. If a particle lands outside the field it is sorted in the category according where it went out, e.g., left sideline or opponent ground line. This is repeated for every particle of every Action that is defined. The algorithm then counts the number of particles of each action that is either inside the field or inside the opponent goal.

Decision

If an action has less than the defined threshold of particles either inside the field or inside the opponent goal the action is discarded. For the remaining actions the one with the most particles inside the opponent goal is calculated. If there are two actions with the most particles inside the goal the best action is determined by evaluating the particles of each actions with the potential field. The action with the smaller sum is selected. If at least one particle of an action is inside the own goal the action will not be chosen. If no action has a particle inside the opponent goal the potential field is used to rank the actions. In this case all particles from one action are evaluated by the potential field and the mean of these values is calculated. The action with the highest mean is selected and executed. If no actions has enough good particles. The best action is to turn towards the opponent goal.

6.3.4 Potential field

Legal ball positions are evaluated using an potential field to assign scores to be used to decide on which kick to perform. The potential field

$$P(x,y) = P_{\text{slope}}(x,y) + P_{\text{own goal}}(x,y) + P_{\text{opp goal}}(x,y) \tag{6.5}$$

consists of three parts P_{slope}, $P_{\text{own goal}}$, and $P_{\text{opp goal}}$, which are modeled as

$$P_{\text{slope}}(x,y) = -\frac{x}{x_{\text{opp}}}, \tag{6.6}$$

$$P_{\text{own goal}}(x,y) = Exp\left(-\left(\frac{(x-x_{\text{own}})^2}{2\sigma_{x,\text{own}}^2} + \frac{y^2}{2\sigma_{y,\text{own}}^2}\right)\right), \tag{6.7}$$

$$P_{\text{own goal}}(x,y) = -Exp\left(-\left(\frac{(x-x_{\text{opp}})^2}{2\sigma_{x,\text{opp}}^2} + \frac{y^2}{2\sigma_{y,\text{opp}}^2}\right)\right) \tag{6.8}$$

with parameters x_{opp}, $\sigma_{x,\text{own}}$, $\sigma_{y,\text{own}}$, x_{opp}, $\sigma_{x,\text{opp}}$, and $\sigma_{y,\text{opp}}$ respectively. An example configuration of these parameters is

$$\begin{aligned} x_{\text{opp}} &= 3000, \\ \sigma_{x,\text{own}} &= 2250, \\ \sigma_{y,\text{own}} &= 800, \\ x_{\text{opp}} &= -3000, \\ \sigma_{x,\text{opp}} &= 1500, \\ \sigma_{y,\text{opp}} &= 800. \end{aligned}$$

The potential field described by these parameters is depicted in figure 6.5.

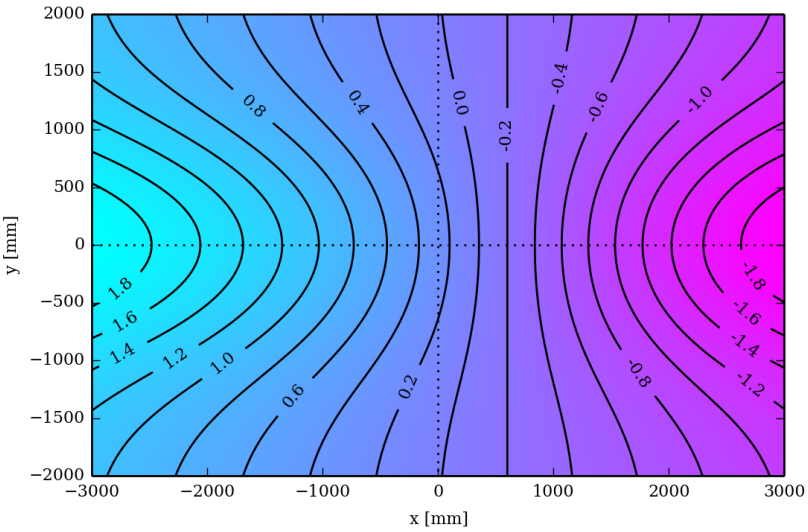


Figure 6.5: Potential field used to evaluate ball positions.

Chapter 7

Motion Control

The performance of a soccer robot is highly dependent on its motion ability. Together with the ability to walk, the kicking motion is one of the most important motions in a soccer game. However, at the current state the most common approaches of implementing the kick are based on key frame technique. Such solutions are inflexible and costs a lot of time to adjust robot's position. Moreover, they are hard to integrate into the general motion flow, e. g., for the change between walk and kick the robot has usually to change to a special stand position.

Fixed motions such as keyframe nets perform well in a very restricted way and determinate environments. More flexible motions must be able to adapt to different conditions. There are at least two specifications: Adaption to control demands, e. g., required changes of speed and direction, omnidirectional walk, and adaptation to the environment, e. g., different floors. The adaptation of the kick according to the ball state and fluent change between walk and kick are another example.

At the current state we have a stable version of an omnidirectional walk control and a dynamic kick, both used in our gameplay. Along with further improvements of the dynamic walk and kick motions our current research focuses in particular on *integration* of the motions, e. g., fluent change between walk and kick.

Adaptation to changing conditions requires feedback from sensors. We experiment with the different sensors of the NAO. Especially, adaptation to the visual data, e. g., seen ball or optical flow, is investigated. Problems arise from sensor noise and delays within the feedback loop. Within a correlated project we also investigate the paradigm of local control loops, e. g., we extended the Nao with additional sensors.

7.1 Walk

The algorithm we use to accomplish a walking motion can be subdivided into the three components: the step planer, the preview controller and stabilization.

At first the step planer determines the target position for the next step considering the walk request and various stability criteria. After that a sequence of desired ZMPs (zero moment points) is planned for each execution cycle of that step. This sequence of ZMPs is used by the preview controller to compute the trajectory of the COM (center of mass) during the execution of the step assuming a linear inverted pendulum model. While the step is executed the foot's 3-d trajectory is calculated on demand and combined with the corresponding COM pose to finally determine the target joint configuration using inverse kinematics.

7.1.1 Step Planner

The step planer calculates the next 2D positions for the feet based on the motion request.

The Motion Request contains the Walk Request as an optional part. A walk request contains information about the destination of the walk and is defined by a target pose (x, y, θ) and the frame of reference of the destination (the left, the right foot or the hip). Therefore the Walk Request is transformed into a virtual origin of the supporting foot-to-be. Virtual means that no physical counterpart exists. In this coordinate system the Walk Request is applied resulting in the virtual target origin for the moving foot. From this virtual target origin the target pose for the step is determined.

Our walk supports two different types of steps which can be requested. The normal walk step is interpolated lineary between start and target foot position. The Step-Control step can be used to realize more complicated trajectories, like arcs.

The requested steps are restricted due to anatomic constraints and increasing the walk's stability. A step is resticted elliptically in x-y-plane in general. The normal step's final dimensions are scaled by the cosinus of the requested rotation. So if a huge rotation is requested the translation will be small. In addition, the change in the step size is also restricted for normal steps to increase stability. Therefore the robot won't begin to walk with the maximal possible step size using normal steps. After applying these restrictions the step is finally added to the step buffer.

Independent of the requested steps the step planner might insert Zero-Steps for increasing the stability of the walk. A Zero-Step is a step in which

no foot is moved.

7.1.2 Preview Control

The Preview Controller calculates the trajectory for the COM based on planed ZMPs. For estimating a stable trajectory for the COM we assume a linear inverted pendulum model with constant height. In each planning cycle of a step a target ZMP is added to the ZMP-buffer. The ZMP-buffer is used by the preview controller to calculate the target position, velocity and acceleration of the COM during a step. The following equation is used to determine the control vector [15]:

$$u = - \underbrace{K_x x_k}_{state feedback} - K_I \underbrace{\sum_{i=0}^{k} (p_k - p_k^{ref})}_{accumulated ZMP error} - \underbrace{\begin{bmatrix} f_1, f_2, \ldots, f_N \end{bmatrix}}_{preview\ gain} \underbrace{\begin{bmatrix} p_{k+1}^{ref} \\ p_{k+2}^{ref} \\ \vdots \\ p_{k+N}^{ref} \end{bmatrix}}_{future ZMP} \tag{7.1}$$

Where x_k is a vector describing the location, velocity and acceleration of the COM at time k. p_k is the ZMP and p_k^{ref} the target ZMP at time k. K_x, K_I and $f_1, \cdots, f_N$ are the parameters of the preview controller and are precalculated. The next target COM x_{k+1} can be calculated using a linear motion model:

$$x_{k+1} = Ax_k + ub \tag{7.2}$$

7.1.3 Stabilization

The simplified model can easily affected by disturbances in the environment. Therefore a close loop stabilization is required.

Different control techniques are used during step creation and execution to accomplish a stable walk.

During step creation the target step is adapted by a P-D-Controller mechanism to compensate small errors in the COM's position. Another mechanism uses the average COM-Error. If the average COM-Error exceeds a threshold a emergency stop is performed. This emergency stop is realized by zero steps. As long as the COM-Error doesn't drop below a threshold the robot won't execute a step which is requested by a Walk-Request.

During the execution of a step three stabilization mechanisms are used. At first the height of the hip and its rotation around the x axis are adapted to compensate the moments appearing while a foot is lifted. A second stabilizer tries to keep the upper body in a upright position the whole time. And a

third controller adapts the ankles according to the current orientation of the robot's body and its change in orientation.

Chapter 8

Behavior

The Extensible Agent Behavior Specification Language — *XABSL* cf. [9] is a behavior description language for autonomous agents based on hierarchical finite state machines. XABSL is originally developed since 2002 by the *German Team* cf. [8]. Since then it turned out to be very successful and is used by many teams within the RoboCup community. We use XABSL to model the behavior of single robots and of the whole team in the Simulation League 3D and also in the SPL.

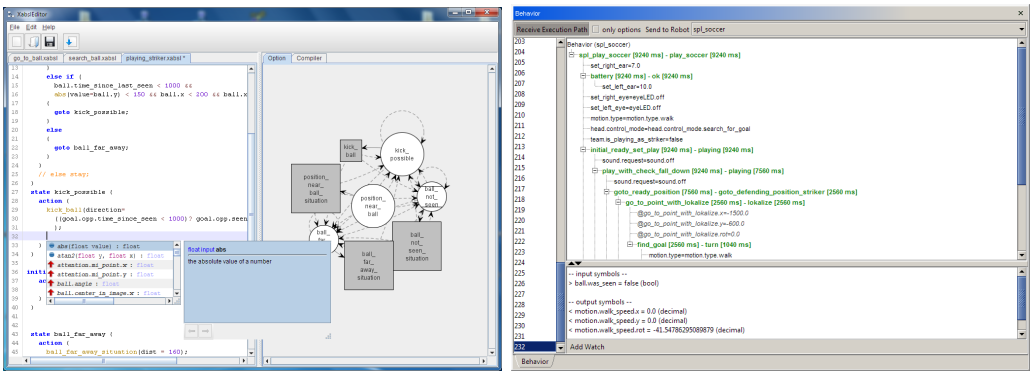


Figure 8.1: (left) XabslEditor: On the left side, you see the source code of a behavior option. On the right side the state machine of this option is visualized as a graph (right). In the main frame the execution path is shown as a tree; at the bottom, some monitored symbols can be seen, the developer can decide which symbols should be monitored; On the left side, there is a list of buffered frames, which is very useful to see how the decisions changed in the past.

In order to be platform independent, we develop our tools in Java. In particular we are working on a Java based development environment for

XABSL, named *XabslEditor*. This tool consists of a full featured editor with syntax highlighting, a graph viewer for visualization of behavior state machines and an integrated compiler. Figure 8.1 (left) illustrates the *XABSL Editor* with an open behavior file.

Another useful tool we are working on is the visualizer for the XABSL execution tree, which allows monitoring the decisions made by the robot at runtime. At the current state, this visualizer is part of our debugging and monitoring tool *RobotControl*. Figure 8.1 (right) illustrates the execution tree of the behavior shown within the visualizer.

8.1 Strategy

We've only implemented a rather simple strategy so far. Our strategy is based on kickoff positions, passive positions and the use of only one striker. Every robot has a unique kickoff position. We distinguish the cases "opponent kickoff" and "own kickoff". The kickoff position depends on the player number. In our strategy, only one robot is allowed to go to the ball. This

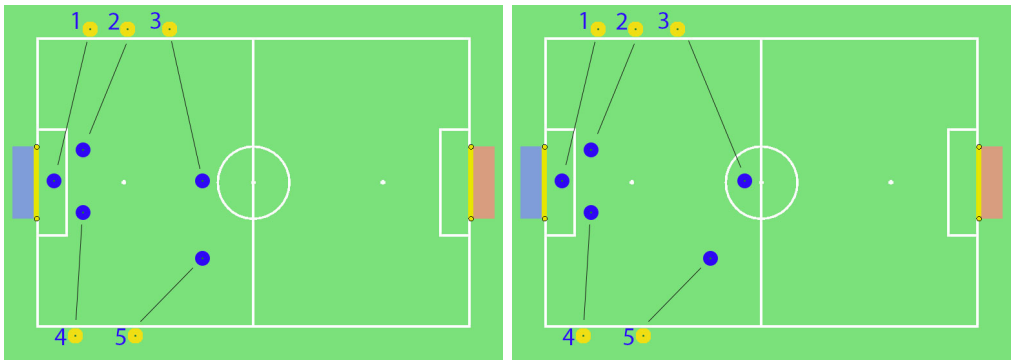


Figure 8.2: (left)The initial and kickoff positions when the opponent team will kickoff
(right)The initial and kickoff positions when our team will kickoff

robot has the striker role. All other robots are in passive mode. Passive means that the robot will look for the ball and, if it doesn't find the ball, it will go to the passive position according to its player number. While going to the passive position, the robot continues to look for the ball. When the robot finds the ball, it will look at it and turn toward the ball. When the robot is at its passive position, it will do the same. If the ball is moved, the passive robots will adjust. If all the robots calculated that a specific robot should

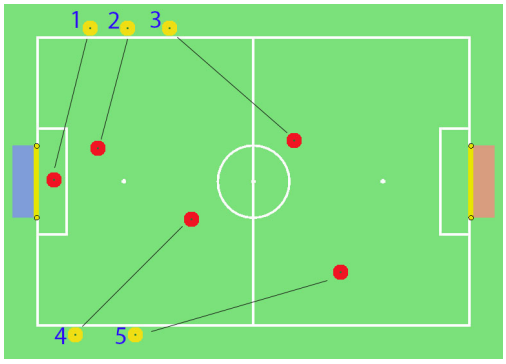


Figure 8.3: The Passive Positions

be the striker, this robot becomes striker and is therefore not in passive play anymore. The Goalie becomes striker if the ball is near the own goal or all the other robots are not in play anymore.

8.2 Role Change

Each robot communicates it's estimated distance to the ball. The robot with the shortest distance becomes striker. This is implemented in a way that oscillations of the role change are prevented.

8.3 Voronoi Based Strategic Positioning

Strategic positioning is a decisive part of the team play within a soccer game. In most solutions the positioning techniques are treated as a constituent of a complete team play strategy.

In our approach, based on the conditions of a specific strategy, the field is subdivided in regions by a Voronoi tessellation and each region is assigned a weight. Those weights influence the calculation of the optimal robot position as well as the path. A team play strategy can be expressed by the choice of the tessellation as well as the choice of the weights. This provides a powerful abstraction layer simplifying the design of the actual play strategy.

The *Voronoi tessellation* is used to separate the field in regions and is defined by a set of points, called *Voronoi sites*, distributed over the field. The area around the robot is divided in higher-resolved regions. With this we can easily construct very complex tessellations based on the conditions given by our strategy. Apart from a set of regions, we also get a graph, called

Delaunay graph, which is defined by the cells as nodes and the neighborhood as edges. This graph gives us a possibility for efficient search within the tessellation.

Scalar fields are used to formulate strategies and to express it in terms of weights of the VBSM. Thereby, the target position is modeled as the global minimum of a scalar field. The striker, goal posts as well as the line between ball and opponent goal should be avoided and therefore are modeled as maxima of the scalar field. In a different way from the target position, the objects should have a limited range of influence. For each Voronoi cell we define the weight as a sum of the scalar fields at the Voronoi site p defining the cell.

The whole *situation map* is defined by this Voronoi tessellation and positive weights assigned to each cell. Thus, the map consist of the spatial separation of the field in regions and a graph structure over the defining nodes. Basically, we can consider this map as a *weighted undirected graph* where the weights of the nodes are given directly by the definition and the weights for the edges are determined as a combination of the metric distance between the defining points and the weights of the nodes.

To solve the positioning task the A* algorithm is employed to find the shortest path. Thereby the *start node* is the region containing the position of the robot and the target node defined by the minimal weight.

Note that the geometry of the tessellation changes over time depending on the position of the player. The path calculated in one frame gives only a rough direction for the movement. The resulting path which emerges through the robot following the given directions will be much smoother as the higher resolution around the robot moves with it. The Figure 8.4 (right) illustrates the resulting tessellation. [7]

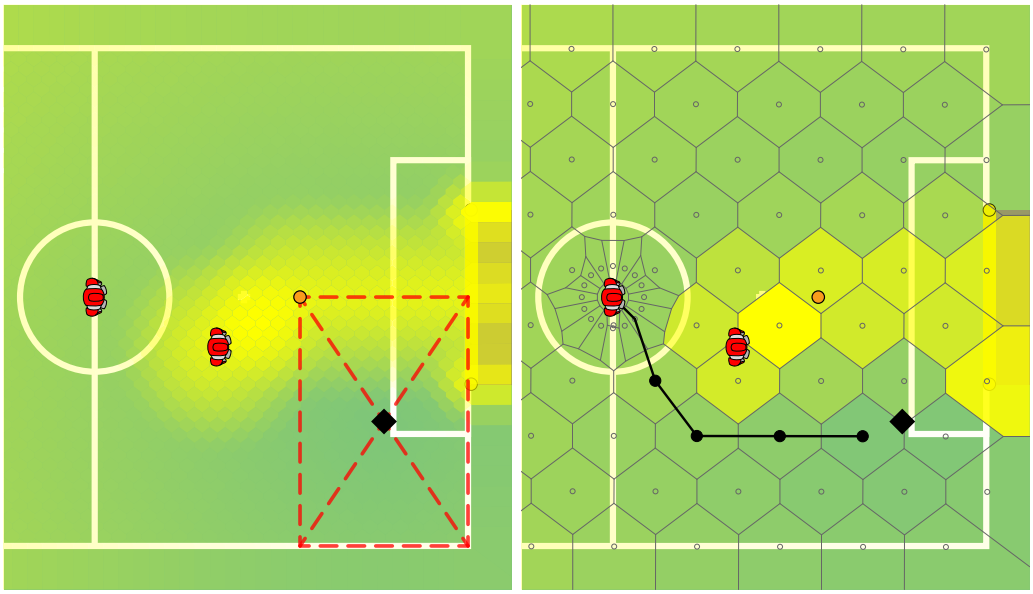


Figure 8.4: An example situation: (left) initial positions of the supporter (center) and the attacker (closer to the ball); the center (black diamond) of the red dashed rectangle illustrates the target position for the supporter; the scalar field encoding the strategy is depicted by the intensity of the yellow glow (the global minimum is at the diamond); (right) the Voronoi tessellation with the weights of the regions depicted by the intensity of the yellow color; path calculated by the A*.

Bibliography

[1] Ralf Berger. Die Doppelpass-Architektur. Verhaltenssteuerung autonomer Agenten in dynamischen Umgebungen (in German). Diploma thesis, Humboldt-Universität zu Berlin, Institut für Informatik, 2006.

[2] Ralf Berger and Gregor Lämmel. Exploiting Past Experience. Case-Based Decision Support for Soccer Agents. In *Proceedings of the 30th Annual German Conference on Artificial Intelligence (KI'07)*. Springer, 2007.

[3] Hans Dieter Burkhard. Programming Bounded Rationality. In *Proceedings of the International Workshop on Monitoring, Security, and Rescue Techniques in Multiagent Systems (MSRAS 2004)*, pages 347–362. Springer, 2005.

[4] Hans-Dieter Burkhard and Ralf Berger. Cases in robotic soccer. In Michael M. Richter Rosina O. Weber, editor, *Case-Based Reasoning Research and Development, Proc. 7th International Conference on Case-Based Reasoning, ICCBR 2007*, Lecture Notes in Artificial Intelligence, pages 1–15. Springer, 2007.

[5] Daniel Hein. Simloid – evolution of biped walking using physical simulation. Diploma thesis, Humboldt-Universität zu Berlin, Institut für Informatik, 2007.

[6] Daniel Hein, Manfred Hild, and Ralf Berger. Evolution of biped walking using neural oscillators and physical simulation. In *RoboCup 2007: Robot Soccer World Cup XI*, Lecture Notes in Artificial Intelligence. Springer, 2007.

[7] Steffen Kaden, Heinrich Mellmann, Marcus Scheunemann, and Hans-Dieter Burkhard. Voronoi based strategic positioning for robot soccer. In Marcin S. Szczuka, Ludwik Czaja, and Magdalena Kacprzak, editors,

Proceedings of the 22nd International Workshop on Concurrency, Specification and Programming (CS&P), volume 1032 of *CEUR Workshop Proceedings*, pages 271–282, Warsaw, Poland, 2013. CEUR-WS.org.

[8] M. Lötzsch, M. Risler, and M. Jüngel. Xabsl - a pragmatic approach to behavior engineering. In *Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS)*, pages 5124–5129, Beijing, China, October 9-15 2006.

[9] Martin Lötzsch, Matthias Jüngel, Max Risler, and Thomas Krause. XABSL web site. 2006. http://www.ki.informatik.hu-berlin.de/XABSL.

[10] Heinrich Mellmann, Yuan Xu, Thomas Krause, and Florian Holzhauer. Naoth software architecture for an autonomous agent. In *International Workshop on Standards and Common Platforms for Robotics (SCPR 2010)*, Darmstadt, November 2010.

[11] Thomas Reinhardt. Die Kalibrierungsfreie Bildverarbeitungsalgorithmen zur echtzeitfähigen Objekterkennung im Roboterfussball (in German). Master thesis, Hochschule für Technik, Wirtschaft und Kultur Leipzig, Fakultät für Informatik, Mathematik und Naturwissenschaften, 2011.

[12] T. Röfer, J. Brose, D. Göhring, M. Jüngel, T. Laue, and M. Risler. GermanTeam 2007 - The German national RoboCup team. In *RoboCup 2007: Robot Soccer World Cup XI Preproceedings*. RoboCup Federation, 2007.

[13] Marcus M Scheunemann and Heinrich Mellmann. Multi-hypothesis goal modeling for a humanoid soccer robot. In *Proceedings of the 9th Workshop on Humanoid Soccer Robots, 14th IEEE-RAS International Conference on Humanoid Robots (Humanoids), Madrid, Spain.*, 2014.

[14] Vìctor Uc-Cetina. *Reinforcement Learning in Continuous State and Action Spaces*. PhD thesis, Humboldt-Universität zu Berlin, 2009.

[15] Yuan Xu. *From simulation to reality – migration of humanoid robot control*. PhD thesis, Humboldt-Universität zu Berlin, 2014.