

# Berlin United - Nao Team Humboldt Team Report 2014

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

11th January 2015

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Architecture</b>	<b>5</b>
2.1	NaoSMAL . . . . .	5
2.2	Platform Interface . . . . .	6
2.3	Module framework . . . . .	7
<b>3</b>	<b>Debugging and Tools</b>	<b>10</b>
3.1	Concepts . . . . .	10
3.2	RobotControl Dialogs . . . . .	11
<b>4</b>	<b>Simulation</b>	<b>17</b>
<b>5</b>	<b>Visual Perception</b>	<b>19</b>
5.1	HistogramProvider . . . . .	19
5.2	SimpleFieldColorClassifier . . . . .	20
5.3	ScanLineEdgeDetector . . . . .	20
5.4	FieldDetector . . . . .	21
5.5	LineGraphProvider . . . . .	21
5.6	GoalFeatureDetector . . . . .	22
5.7	GoalDetector . . . . .	22
5.8	BallDetector . . . . .	23
<b>6</b>	<b>Modeling</b>	<b>25</b>
6.1	Probabilistic Compass . . . . .	25
6.2	Multi-Hypothesis Goal Model (MHGM) . . . . .	26
<b>7</b>	<b>Motion Control</b>	<b>28</b>
<b>8</b>	<b>Behavior</b>	<b>29</b>
8.1	Strategy . . . . .	30
8.2	Role Change . . . . .	31

<i>CONTENTS</i>	2
8.3 Voronoi Based Strategic Positioning . . . . .	31

# 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 (FUManooids, 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 FUManooids 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 FUmoids, 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:

- Narrowing the gap between simulated and real robots (section 4)
- Software architecture for autonomous agents (section 2)
- 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 easy testing.

In the following subsections we describe the design and the implementation of the different parts of the architecture. A detailed description of the principles we used can be also found in [13]

### 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<sup>1</sup> 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.

---

<sup>1</sup><http://doc.aldebaran.com/1-14/naoqi/sensors/dcm-api.html>

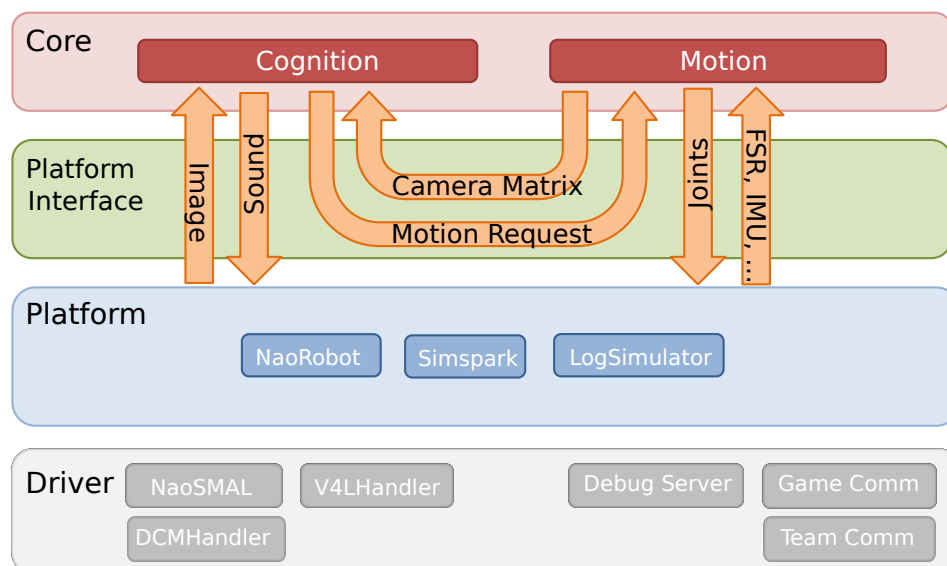


Figure 2.1: Platform Interface is responsible for data transferring and execution of the Cognition and Motion processes.

## 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<sup>2</sup> 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).

<sup>2</sup><http://simspark.sourceforge.net/>

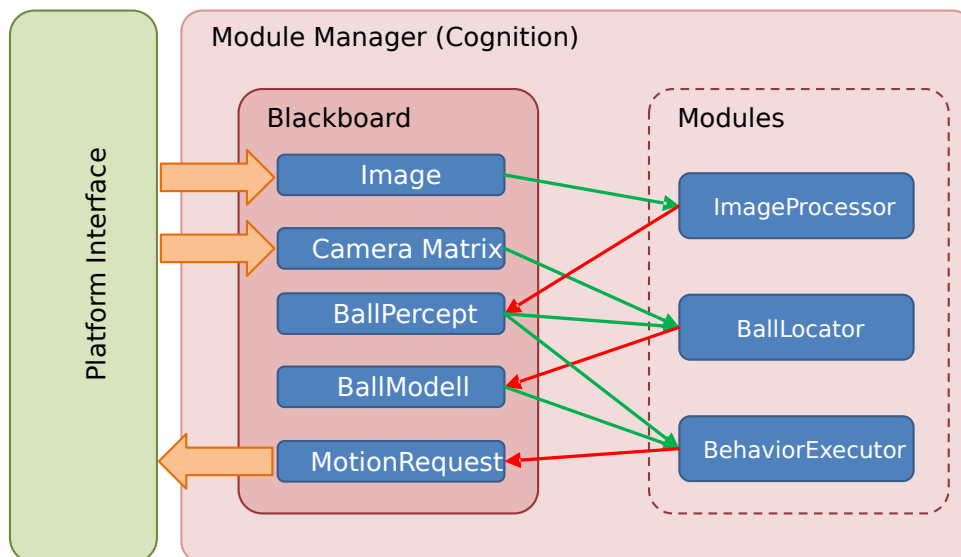


Figure 2.2: Overview about the different components of the module framework.

## 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 (can read and write representations)

**Module Manager** manage the execution of the modules

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

# 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 don't 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 Fig. 3.1).

### 3.1 Concepts

Some of the ideas were evolved from the GT-Architecture [16]. 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, e.g., 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 Fig. 3.1 illustrates some of the visualizations of the debug concepts. In particular a field view, 3D view, behavior tree, plot and the table of debug requests are shown.

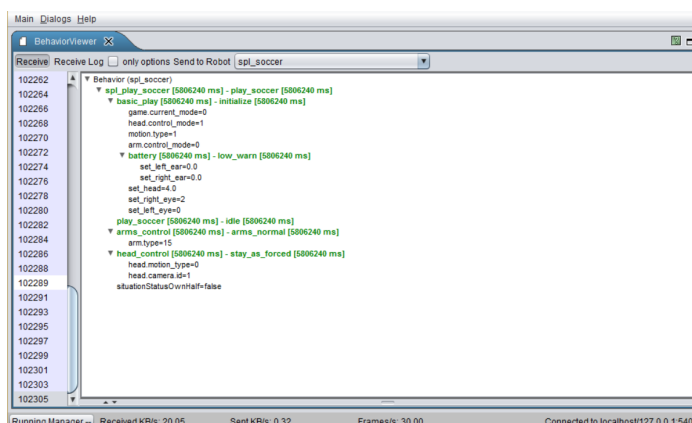
## 3.2 RobotControl Dialogs

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



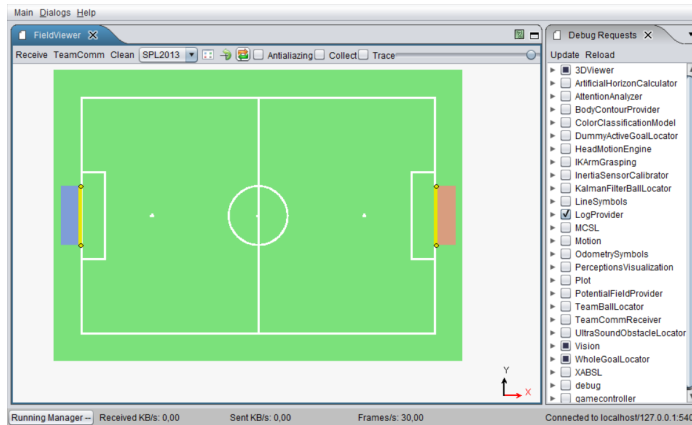
Figure 3.1: The RobotControl program contains different dialogs. The 3DViewer (top left) is used to visualize the current state of the robot; the Value Plotter dialog (bottom left) plots some data; the Field Viewer dialog (top center) draws the field view; the Behavior dialog (bottom center) shows the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

### 3.2.1 Behavior Viewer



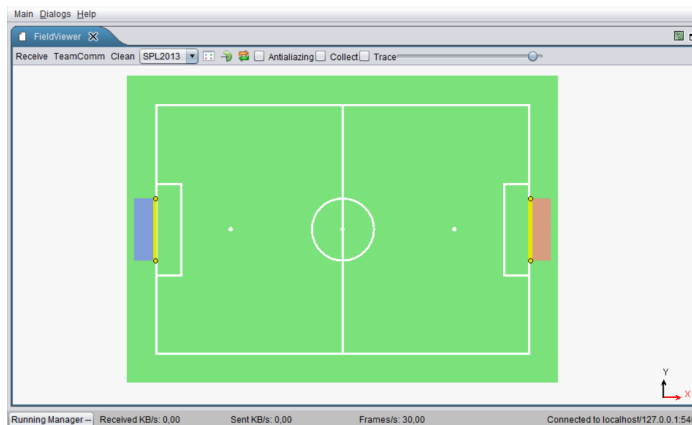
Shows the Behavior tree for the current behavior. The compiled XABSL behavior needs to be sent to the robot first and then an agent can be selected to be executed. With 'Add Watch' you can track XABSL input and output symbols.

## Debug Requests



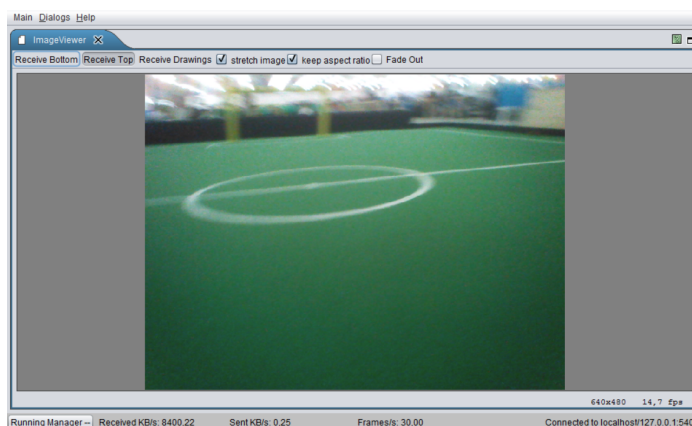
(De-)activates the debug request code. Usually a debug request draws something on the field viewer or on the camera images. For further information about individual debug requests, have a look at the source code.

## Field Viewer



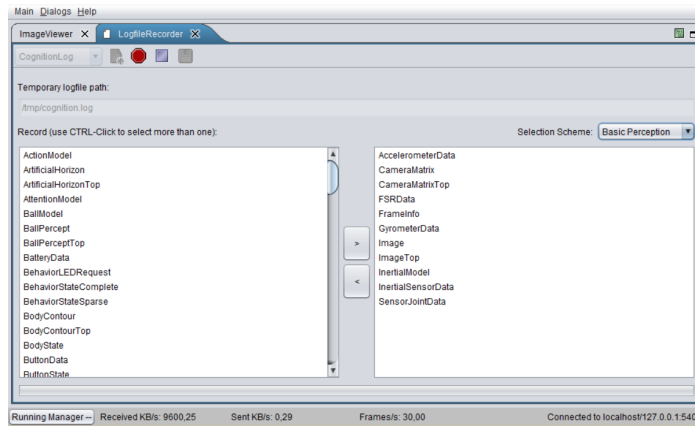
There are views for different field sizes and a local view. Certain debug requests draw on these views. For example, you could draw the robots' positions on the field by activating the corresponding debug request.

## Image Viewer



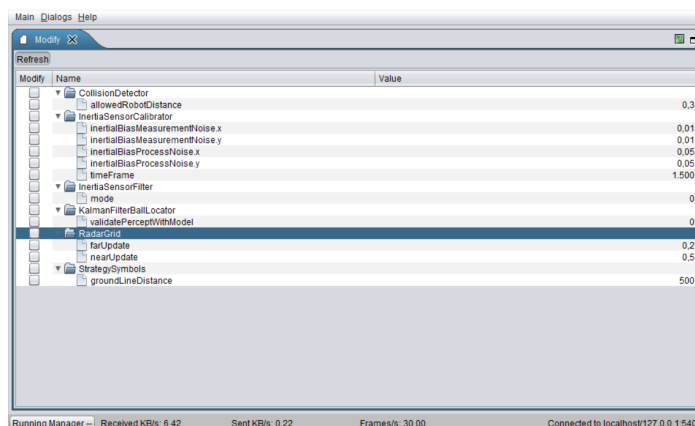
Can show the top and bottom images. There are debug requests that draw on the camera images, if they are active.

## Logfile Recorder



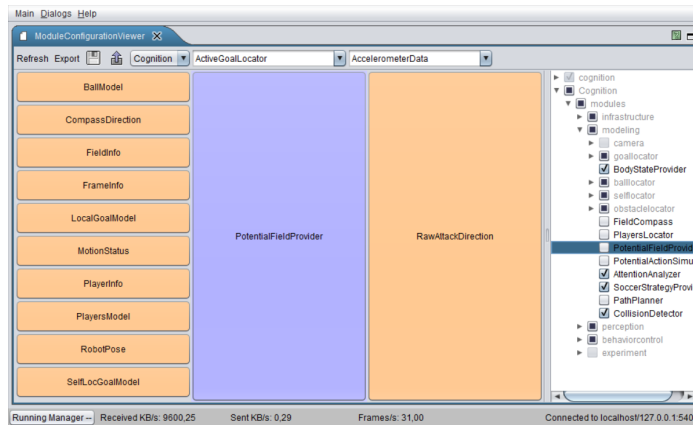
Records a Logfile with selected data.

## Modify



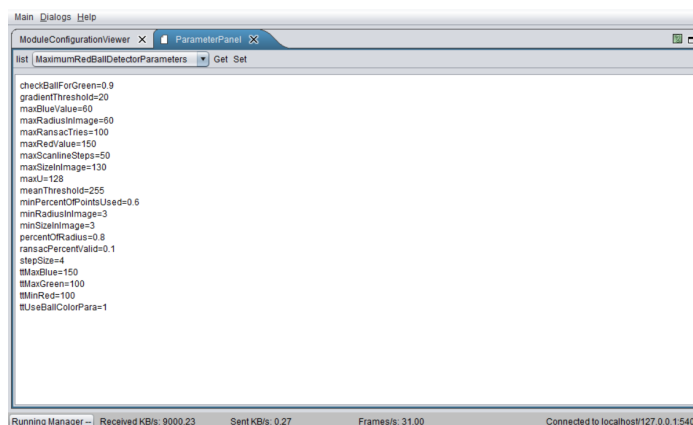
The Modify macro allows changing values of variables declared within this macro at runtime.

## Module Configuration Viewer



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

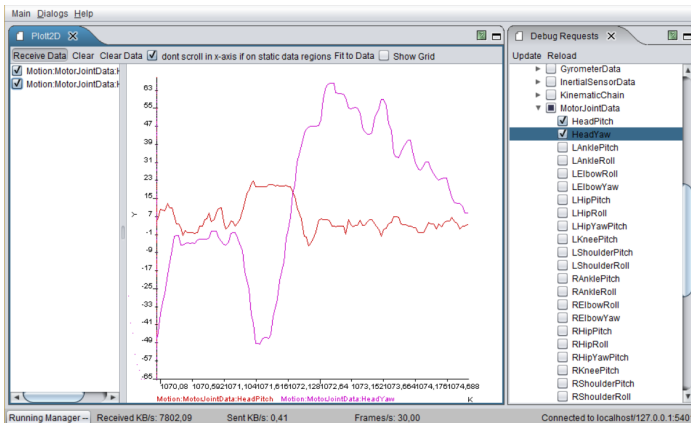
## Parameter Panel



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

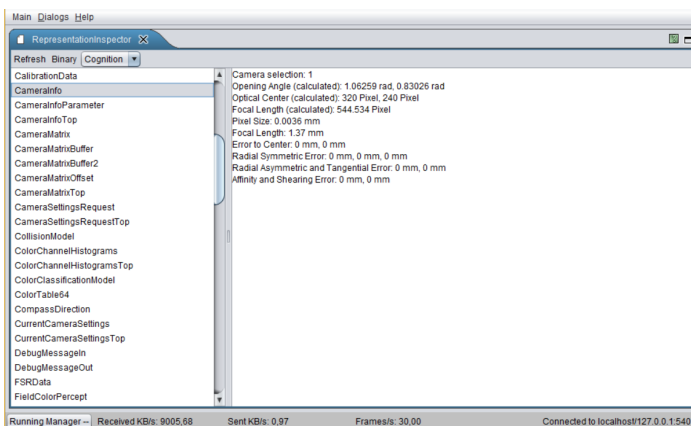


## Plot 2D



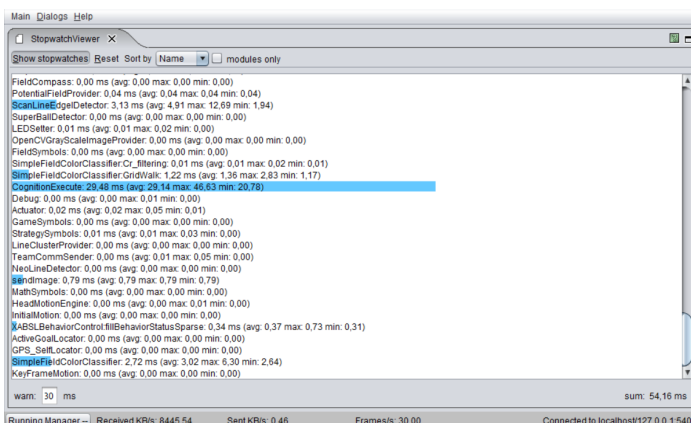
Shows plots activated by plot debug requests.

## Representation Inspector



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

## Stopwatch Viewer



Shows the execution time for each module.

# 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 Fig. 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 [13], 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 [7, 6]. Reinforcement Learning was used for the development of dribbling skills in the 2D simulation [19], 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 an appropriate model of the surrounding world. Thus, one of the main focuses of our current research is the improvement of the perceptual 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 [14]. 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 [14] this is the main assumption too.



Since this algorithm of [14] 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 ScanLineEdgeDetector



With this module we detect line border points and estimate some points of the field border. To do this, we use scan lines, but only vertical ones. Along every scan line jumps are detected in the Y channel, using a 1D-Prewitt-Filter. A point of the lines border is located at the maximum of the response of that filter. We estimate with 2 3x3-Sobel-Filters (horizontal and vertical) the orientation of the line. While the result of the field color classification we detect along every scan line a point which marks the border of the field.

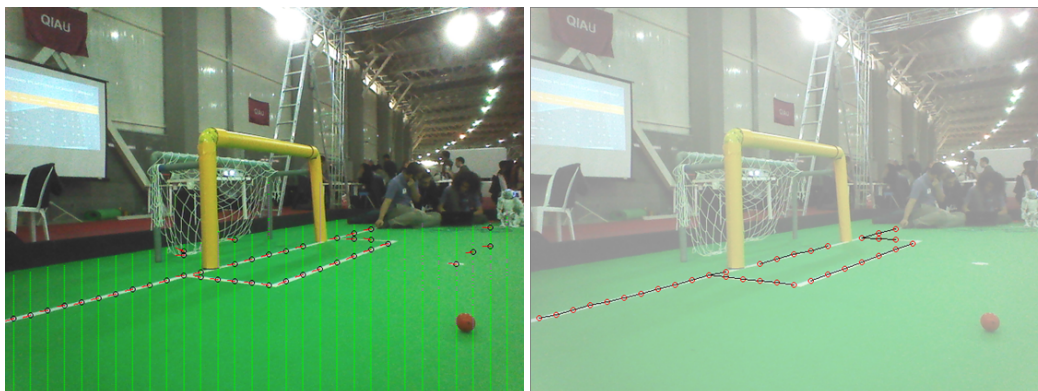
## 5.4 FieldDetector

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



## 5.5 LineGraphProvider

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



## 5.6 GoalFeatureDetector

In this module we use some scan lines around the artificial horizon, calculated by *ArtificialHorizonProvider*, to find groups of pixels, which are likely goal post pixels. We can choose between absolute value detection or gradient jump detection.



Till now, the detection using absolute values above a certain threshold seems to be the best. The scan for absolute values above a certain threshold is done with a  $5 \times 1$  gauss kernel, which is moved along a vertical scan line in the image. For the gradient jump scan a  $3 \times 1$  Prewitt-Filter is used.

## 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. Then 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.



From the features of a cluster, the orientation of the possible goal post is estimated and used to do scan up and down along the estimated goal post. This is done to find the foot and head point of that post. A goal post is seen as valid, if the foot point is inside of the field and if some of the pixels under the goal post (in image coordinates: over the post) are classified as field color. If a goal post is valid, then the statistical information, collected while scanning for the foot and head point, is used to update three histograms, which describe the distribution of the posts color in the YUV color space.

## 5.8 BallDetector

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





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 8 different directions, are used to find border pixels of the assumed ball. The 8 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.

# Chapter 6

## Modeling

In order to realize a complex and successful cooperative behavior it is necessary to have an 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  $\pi$  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.

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

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.

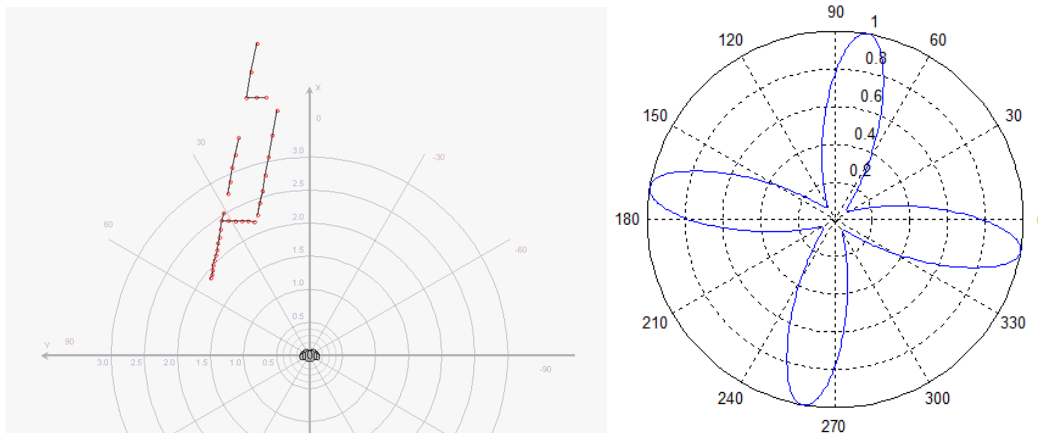


Figure 6.1: Left figure visualizes edge graph in the local coordinates of the robot in a particular frame. Right illustrates the kernel histogram over the orientations of edges shown left calculated with the formula 6.1.

## 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 [17].

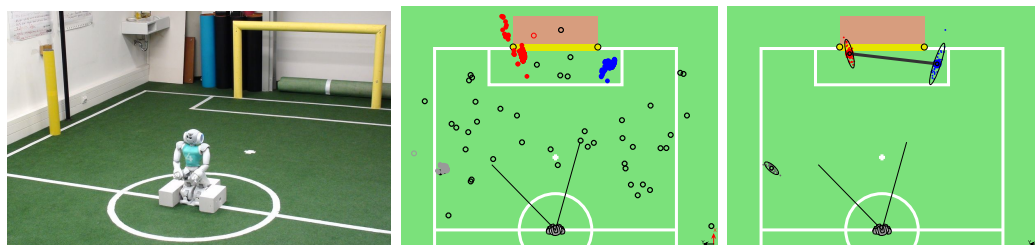


Figure 6.2: The left Figure illustrates the experiment setup. The robot faces the goal and an additional goal post is placed to its right side. From the object recognition perspective, this post is identically to the *real* goal posts. The Figure in the center visualizes all percepts collected during 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.

# 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 examples.

At the current state we have a stable version of an omnidirectional walk control and a dynamic kick which are used in the games. 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.

# Chapter 8

## Behavior

The Extensible Agent Behavior Specification Language — *XABSL* cf. [10] 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. [9]. Since then it turned out to be very successful and is used by many teams withing 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.

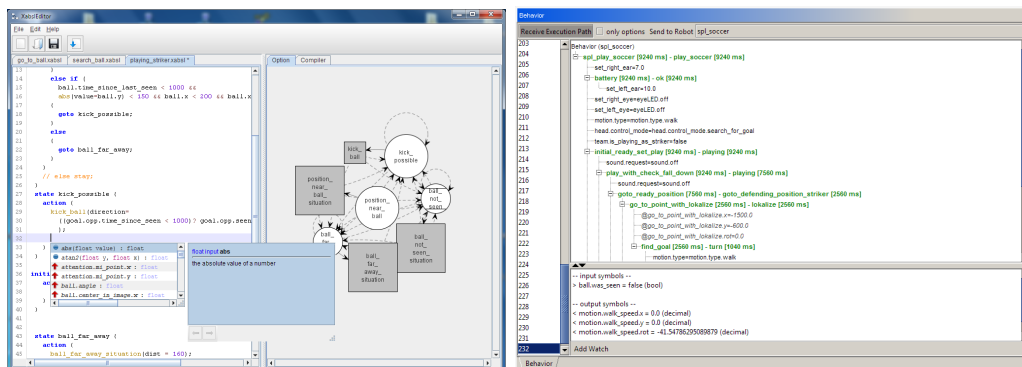


Figure 8.1: (left) XabslEditor: On the left side, you see the source code of a behavior option. On the right side the state machine of this option is visualized as a graph (right). In the main frame the execution path is shown as a tree; at the bottom, some monitored symbols can be seen, the developer 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 independent from the platform, 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

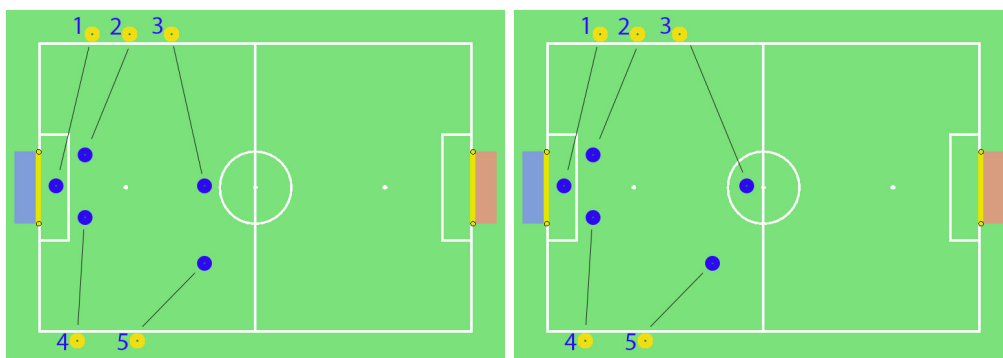


Figure 8.2: (left)The initial and kickoff positions when the opponent team will kickoff  
(right)The initial and kickoff positions when our team will kickoff

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. A robot becomes striker and is therefore not in

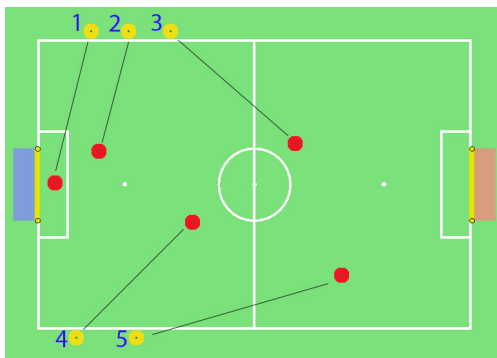


Figure 8.3: The Passive Positions

passive play anymore if all the robots calculated that this robot should be the striker. 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 its 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 Fig. 8.4 (right) illustrates the resulting tessellation. [8]

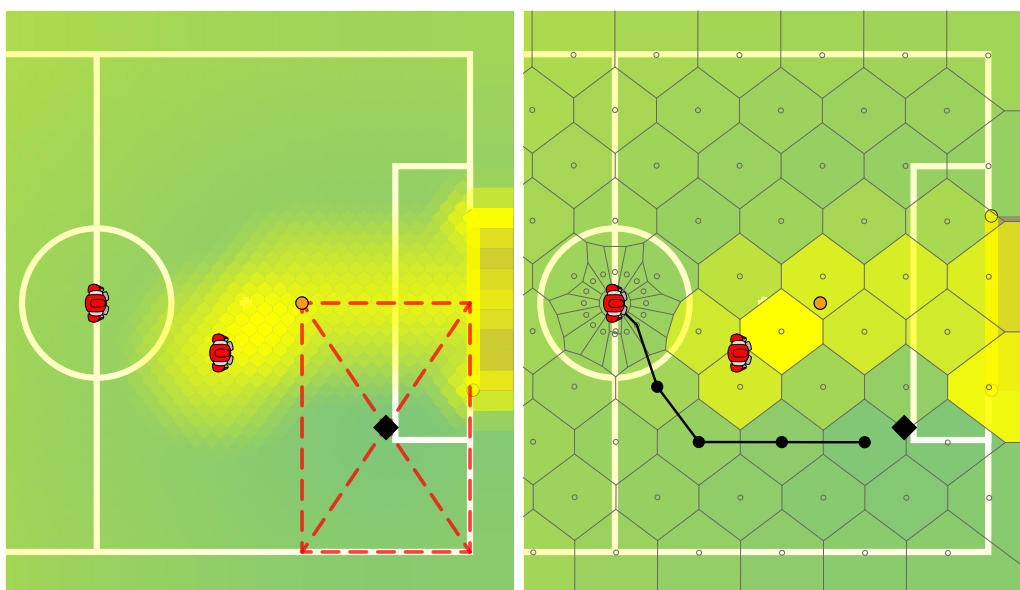


Figure 8.4: An example situation: (left) initial positions of the supporter (center) and the attacker (closer to the ball); the center (black diamond) of the red dashed rectangle illustrates the target position for the supporter; the scalar field encoding the strategy is depicted by the intensity of 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] Giuseppe Cotugno and Heinrich Mellmann. Dynamic motion control: Adaptive bimanual grasping for a humanoid robot. In *Proceedings of the Workshop on Concurrency, Specification, and Programming CS&P 2010*, volume Volume 2, Börnicke (near Berlin), Germany, September 2010.
- [6] Daniel Hein. Simloid – evolution of biped walking using physical simulation. Diploma thesis, Humboldt-Universität zu Berlin, Institut für Informatik, 2007.
- [7] 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.

- [8] 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.
- [9] M. Löttsch, 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.
- [10] Martin Löttsch, Matthias Jüngel, Max Risler, and Thomas Krause. XABSL web site. 2006. <http://www.ki.informatik.hu-berlin.de/XABSL>.
- [11] Heinrich Mellmann. Ein anderes Modell der Welt. Alternative Methoden zur Lokalisierung mobiler Roboter. Diploma thesis, Humboldt-Universität zu Berlin, Institut für Informatik, 2010.
- [12] Heinrich Mellmann and Yuan Xu. Adaptive motion control with visual feedback for a humanoid robot. In *the Proceedings of the 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Taipei, 2010.
- [13] 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.
- [14] Thomas Reinhardt. Die Kalibrierungsfreie Bildverarbeitungs-Algorithmen zur echtzeitfähigen Objekterkennung im Roboterfußball (in German). Master thesis, Hochschule für Technik, Wirtschaft und Kultur Leipzig, Fakultät für Informatik, Mathematik und Naturwissenschaften, 2011.
- [15] Thomas Röfer and Matthias Jüngel. Vision-based fast and reactive monte-carlo localization. In Daniel Polani, Andrea Bonarini, Brett Browning, and Kazuo Yoshida, editors, *Proceedings of the 2003 IEEE International Conference on Robotics and Automation (ICRA)*, pages 856–861. IEEE, 2003.
- [16] 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.
- [17] 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.
- [18] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. The MIT Press, 2006.
- [19] Victor Uc-Cetina. *Reinforcement Learning in Continuous State and Action Spaces*. PhD thesis, Humboldt-Universität zu Berlin, 2009.
- [20] Yuan Xu. *From simulation to reality – migration of humanoid robot control*. PhD thesis, Humboldt-Universität zu Berlin, 2014.
- [21] Yuan Xu and Hans-Dieter Burkhard. Narrowing reality gap and validation: Improving the simulator for humanoid soccer robot. In *Concurrency, Specification and Programming CS&P'2010*, Helenenau, Germany, September 2010.
- [22] Yuan Xu and Heinrich Mellmann. Adaptive motion control: Dynamic kick for a humanoid robot. In *the Proceedings of the 33rd Annual German Conference on Artificial Intelligence*, Karlsruhe, Germany, 2010.
- [23] Yuan Xu, Heinrich Mellmann, and Hans-Dieter Burkhard. An approach to close the gap between simulation and real robots. In Ando, Balakirska, Reggiani, and von Stryk, editors, *2nd International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR)*, Darmstadt, November 2010.