GERMANTEAM

RoboCup 2002

Hans-Dieter Burkhard
Uwe Düffert
Jan Hoffmann
Matthias Jüngel
Martin Lötzsch

Ronnie Brunn
Martin Kallnik
Nicolai Kuntze
Michael Kunz
Sebastian Petters
Max Risler
Oskar von Stryk

Institut für Informatik,
LFG Künstliche Intelligenz,
Humboldt-Universität zu Berlin,
Rudower Chaussee 25,
12489 Berlin, Germany

Fachgebiet Simulation und Systemoptimierung,
FB 20 Informatik,
Technische Universität Darmstadt,
Alexanderstr. 10,
64283 Darmstadt, Germany

Nils Koschmieder
Tim Laue
Thomas Röfer
Kai Spiess

Arthur Cesarz
Ingo Dahm
Matthias Hebbel
Walter Nowak
Jens Ziegler

Center for Computing Technology,
FB 3 Informatik,
Universität Bremen,
Postfach 330440,
28334 Bremen, Germany

Computer Engineering Institute,
Chair of Systemanalysis,
University of Dortmund,
Otto-Hahn-Strasse 4,
44221 Dortmund, Germany

Abstract

The GermanTeam is a joint project of several German universities in the Sony Legged Robot League. This report describes the software developed for the RoboCup 2002 in Fukuoka. It presents the software architecture of the system as well as the methods that were developed to tackle the problems of motion, image processing, object recognition, self-localization, and robot behavior. The approaches for both playing robot soccer and mastering the challenges are presented. In addition to the software actually running on the robots, this document will also give an overview of the tools the GermanTeam used to support the development process.

In an extensive appendix, several topics are described in detail, namely the installation of the software, how it is used, the implementation of inter-process communication, streams, and debugging mechanisms, and the approach of the GermanTeam to model the behavior of the robots.

Contents

Chapter 1

Introduction

1.1 History

The GermanTeam is the successor of the Humboldt Heroes who already participated in the Sony Legged League competitions in 1999 and 2000. Because of the strong interest of other German universities, in March 2001, the GermanTeam was founded. It consists of students and researchers of five universities: Humboldt-Universität zu Berlin, Universität Bremen, Technische Universität Darmstadt, Universität Dortmund and Freie Universität Berlin. For the RoboCup 2001, the Humboldt Heroes only had reinforcements from Bremen and Darmstadt during the last two or three month before the world championship in Seattle took place.

In 2002, only the Freie Universität Berlin provided no active team members. Therefore, the system presented in this document is the result of the work of the team members from the other four universities. Each of these four groups has its own team in the Sony Legged RoboCup League, but they only participated separately in the German Open 2002 in Paderborn and formed a single national team in Fukuoka. The four teams are the *Bremen Byters*, the *Darmstadt Dribbling Dackels*, *Humboldt 2002*, and the *Ruhrpott Hellhounds* (Dortmund).

1.2 Scientific Goals

All the universities participating have certain special research interests, which they try to carry out in the GermanTeam's software.

1.2.1 Humboldt-Universität zu Berlin

The main interests of Humboldt University's researchers are robotic architectures for autonomous robots based on mental models and the development of complex behavior control architectures.

Even simple environments and tasks for autonomous robots require sophisticated robot architectures that satisfy real-time conditions. Such designs need to manage the sensor readings, the actuator control, the representation of internal and external objects, the planning system, and

the debugging and visualization. This leads to the specification and development of a robust and extensible software model that can be applied to different robotic platforms. The software architecture's modularity allows for different solutions to be developed independently and in parallel. For a given task, e. g. image processing, modules that fulfill this task can be interchanged at run-time (e. g. switch from "grid based image processing" to "flood fill image processing"). This leads to the development of a wide variety of approaches within the software project and enables us to easily benchmark and compare these solutions. For robots with complex tasks in natural environments, it is necessary to equip them with behavior control architectures that allow planning based on fuzzy and incomplete information about the environment, cooperation without communication, and actions directed to different goals. These architectures have to integrate both long-term plans and short-term reactive behaviors. On the one hand, they should not stick to dead end decisions, but on the other hand, they should also not change their intentions too frequently for a goal to be reached.

The behavior architecture first developed in 2001 was improved this year and an XML dialect for describing behaviors was added. It will ultimately be joined with the case based reasoning approach developed in our Simulation League team. This will give us a lot more flexibility in testing behaviors and knowledge transfer between the different leagues.

In addition, guided means of information gathering (e. g. active vision) have been found to be important in the context of the Sony Four Legged League. The limited resources of the robot (small field of vision, limited computing power) make it necessary to optimize the use of these resources and prohibit the use of brute force algorithms (especially in image processing). As a result of this, image acquisition and image analysis need to be closely linked with information processing and world modeling. These formerly separate, independent processes will need to become more closely coupled.

1.2.2 Technische Universität Darmstadt

The long-term goals of the team in Darmstadt are conceptual and algorithmic contributions to all sub-problems involved for a successful autonomous team of soccer playing legged robots (perception, localization, locomotion, behavior control).

The group in Darmstadt is developing tools for an efficient kinetic modeling and simulation of legged robot dynamics taking into account masses and inertias of each robot link as well as motor, gear and controller models of each controlled joint. Based on these nonlinear dynamic models computational methods for simulation, dynamic off-line optimization and on-line stabilization and control of dynamic walking and running gaits are developed and applied. These methods will be applied to develop and implement new, fast, and stable locomotion for legged robots, namely the Sony four-legged robots and a new humanoid robot under development. Until June 2002 a complete three-dimensional dynamical model of the Sony four-legged robot has been implemented based on the kinematical and kinetical data provided by Sony. This model has been used to optimize a fast trot gait in simulation using numerical optimal control methods. The simulation results still must be validated through experiments as well as model uncertainties concerning dynamic models and data of motors, gears, controllers and of the ground contact models.

However, new methods for planning and controlling legged locomotion of an *autonomous* robot cannot be investigated independently from the limited hard- and software resources which must be shared with other modules under real-time constraints. Thus, for the competitions in 2002, a library for fast and stable evaluation of math functions, a fast, single landmark self-location algorithm, a rapid XML/XSL state machine modeler as well as a walking engine have been newly developed in addition to the modules contributed by the other German universities. The new modules have been tested successfully during the RoboCup German Open in April.

1.2.3 Universität Bremen

The main research interest of the group in Bremen is the automatic recognition of the strategies of other agents, in particular, of the opposing team in RoboCup. A new challenge in the development of autonomous physical agents is to model the environment in an adequate way. In this context, modeling of other active entities is of crucial importance. It has to be examined, how actions of other mobile agents can be identified and classified. Their behavior patterns and tactics should be detected from generic actions and action sequences. From these patterns, future actions should be predicted, and thus it is possible to select adequate reactions to the activities of the opponents. Within this scenario, the other physical agents should not to be regarded individually. Rather it should be assumed that they form a self-organizing group with a common goal, which contradicts the agent's own target. In consequence, an action of the group of other agents is also a threat against the own plans, goals, and preferred actions, and must be considered accordingly. Acting under the consideration of the actions of others presupposes a high degree of adaptability and the capability to learn from previous situations. Thus these research areas will also be emphasized in the project.

The research project focuses on strategy detection of agents in general. However, the RoboCup is an ideal test-bed for the methods to be developed. The technology of strategy recognition is of large interest in two research areas: on the one hand, the quality of forecasting the actions of physical agents can be increased, which plays an important role in the context of controlling autonomous robots, on the other hand, it can be employed to increase the robustness and security of electronic markets. This project is also part of the priority program "Cooperating teams of mobile robots in dynamic environments" funded by the Deutsche Forschungsgemeinschaft (German Research Foundation).

In the Sony Legged Robot League, it is the goal of the group from Bremen to establish a robust and stable world model that will allow techniques for opponent modeling developed in the simulation league to be applied to a league with real robots.

1.2.4 Universität Dortmund

The team of the University of Dortmund focuses its research interests on two separate fields: The first is the combination of different sensor data information in order to build a sensor-fusion based world model. Therefore, we develop methodologies to classify objects that consider the specific problems of that challenge. On the other hand, we deal with the problem of developing gait patterns for mobile robots.

Fast and robust locomotion modules are crucial for the reliable movement of biped and quadruped robots. Developing parameter sets for gait patterns by hand is complex, time-consuming, and therefore expensive. Thus, methods that generate gait patterns (semi-) automatically by trial-and-error techniques on the robot are expected to lower the development costs. Unfortunately, such methods are typically characterized by a heavy wearout. Therefore, we examine techniques that are capable of reducing the wearout by classifying the fitness of the particular gait pattern offline.

The second research interest deals with sensor fusion. We combine the sensor information of a set of robots in order to generate a world model that goes beyond the quality of a single-robot world model. For that, reliability information about sensor input, classification accuracy, and other a priory information must be extracted, exchanged between the individuals, and taken into account when merging all additional information available.

1.3 Contributing Team Members

At the four universities providing active team members, many people contributed to the GermanTeam:

1.3.1 Humboldt-Universität zu Berlin

Graduate Students. Uwe Düffert, Matthias Jüngel, Martin Lötzsch.

PhD Students. Joscha Bach, Jan Hoffmann.

Professor. Hans-Dieter Burkhard.

1.3.2 Technische Universität Darmstadt

Graduate Students. Ronnie Brunn, Martin Kallnik, Michael Kunz, Nicolai Kuntze, Sebastian Petters, Max Risler.

Post-Doc Researcher. Michael Hardt.

Professor. Oskar von Stryk.

1.3.3 Universität Bremen

Graduate Students. Denny Koene, Jürgen Köhler, Nils Koschmieder, Lang Lang, Tim Laue, Kai Spiess, Andreas Sztybryc, Hong Xiang.

Assistant Professor. Thomas Röfer.

1.3.4 Universität Dortmund

Graduate Students. Arthur Cesarz, Oliver Giese, Matthias Hebbel, Holger Hennings, Simon Fischer, Phillip Limburg, Marc Malik, Patrick Matters, Markus Meier, Ingo Mierswa, Walter Nowak, Christian Neumann, Denis Piepenstock, Jens Rentmeister, Lars Schley.

PhD Students. Ingo Dahm, Jens Ziegler.

1.4 Structure of this Document

This document gives a complete survey over the software of the GermanTeam, i. e. it does not just describe the differences between last year's version and the actual one. Thus new teams only have to read this year's documentation. However, people who already read the report from the year ago will find some repetitions.

Chapter 2 describes the software architecture implemented by the GermanTeam. It is motivated by the special needs of a national team, i.e. a "team of teams" from different universities in one country that compete against each other in national contests, but that will jointly line up at the international RoboCup championship. In this architecture, the problem of a robot playing soccer is divided in several tasks. Each task is solved by a *module*. The implementations of these modules for the soccer competition are described in chapter 3. Chapter 4 describes the solutions for the three so-called challenges.

Only half of the approximately 200,000 lines of code that were written by the GermanTeam for the RoboCup 2002 are actually running on the robots. The other half was invested in powerful tools that provide sophisticated debugging possibilities including the first 3-D simulator for the Sony Legged Robot League. These tools are presented in chapter 5.

The main part of this report is finished by concluding the results achieved in 2002 and giving an outlook on the future perspectives of the GermanTeam in 2003 in chapter 6.

In the appendix, several issues are described in more detail. It starts with an installation guide in appendix A. Appendix B is a quick guide how to setup the robots of the GermanTeam to play soccer and how to use the tools. Then, the GermanTeam's abstraction of *processes*, *senders*, and *receivers* is presented in appendix C, followed by appendix D on *streams* and appendix E on the debugging support. The appendices F, G, and H are a detailed documentation of the behavior engine used by the GermanTeam in Fukuoka. Finally, the appendices I and J describe the usage of SimGT2002 and RobotControl, the two main tools of the GermanTeam.

Chapter 2

Architecture

The GermanTeam is an example of a national team. The members participated as separate teams in the German Open 2002, but formed a single team at Fukuoka. Obviously, the results of the team would not have been very good if the members developed separately until the middle of April, and then tried to integrate their code to a single team in only two months. Therefore, an architecture for robot control programs was developed that allows implementing different solutions for the tasks involved in playing robot soccer. The solutions are exchangeable, compatible to each other, and they can even be distributed over a variable number of concurrent processes. The approach will be described in section 2.2. Before that, section 2.1 will motivate why the robot control programs are implemented in a platform-independent way, and how this is achieved.

2.1 Platform-Independence

One of the basic goals of the architecture of the GermanTeam 2002 was *platform-independence*, i. e. the code shall be able to run in different environments, e. g. on real robots, in a simulation, or—parts of it—in different RoboCup leagues.

2.1.1 Motivation

There are several reasons to enforce this approach:

Using a Simulation. A Simulation can speed up the development of a robot team significantly. On the one hand, it allows writing and testing code without using a real robot—at least for a while. When developing on real robots, a lot of time is wasted with transferring updated programs to the robot via a memory stick, booting the robot, charging and replacing batteries, etc. In addition, simulations allow a program to be tested systematically, because the robots can automatically be placed at certain locations, and information that is available in the simulation, e. g. the robot poses, can be compared to the data estimated by the robot control programs.

Sharing Code between the Leagues. Some of the universities in the GermanTeam are also involved in other RoboCup leagues. Therefore, it is desirable to share code between the leagues, e. g. the behavior control architecture between the Sony Legged Robot League and the Simulation League.

Non Disclosure Agreement. The participants in the Sony Legged Robot League got access to internal information about the software running on the Sony AIBO robot. Therefore, the universities of all members of the league signed a non disclosure agreement to protect this secret information. As a result and in contrast to other leagues, the code used to control the robots during the championship is usually only made available to the other teams in the league, but not to the public. This might change now, because Open-R is publically available, but already before June 2002, the GermanTeam wanted to be able to publish a version of the system without violating the NDA between the universities and Sony by encapsulating the NDA-relevant code and by the means of the simulator (cf. Sect. 5.1).

2.1.2 Realization

It turned out that platform-dependent parts are only required in the following cases:

Initialization of the Robot. Most robots require a certain kind of setup, e. g., the sensors and the motors have to be initialized. Most parameters set during this phase are not changed again later. Therefore, these initializations can be put together in one or two functions. In a simulation, the setup is most often performed by the simulator itself; therefore, such initialization functions can be left empty.

Communication between Processes. As most robot control programs will employ the advantages of concurrent execution, an abstract interface for concurrent processes and the communication between them has to be provided on each platform. The communication scheme used by the GermanTeam 2002 is illustrated in section 2.2.3.1 and in appendix C.

Reading Sensor Data and Sending Motor Commands. During runtime, the data to be exchanged with the robot and the robot's operating system is limited to sensor readings and actuator commands. In case of the software developed by the GermanTeam 2002, it was possible to encapsulate this part as a communication between processes, i. e. there is no difference between exchanging data between individual processes of the robot control program and between parts of the control program and the operating system of the robot.

File System Access. Typically, the robot control program will load some configuration files during the initialization of the system. In case of the system of the GermanTeam 2002, information as the color of robot's team (red or blue), the robot's initial role (e. g. the goalie), and several tables (e. g. the mapping from camera image colors to the so-called color classes) are loaded during startup.

2.1.3 Supported Platforms

Currently, the architecture has been implemented on three different platforms:

Sony AIBO Robots. The specialty of the Sony Legged Robot League is that all teams use the same robots, and it is not allowed changing them. This allows teams to run the code of other teams, similar to the simulation league. However, this only works if one uses the complete source code of another team. It is normally not possible to combine the code of different teams, at least not without changing it. Therefore, to be able to share the source code in the GermanTeam, the architecture described above was implemented on the Sony AIBO robots. The implementation is based on the techniques provided by Open-R and Aperios that form the operation system that natively runs on the robots.

Microsoft Windows. The platform independent environment was also implemented on Microsoft Windows as a part of a special controller in *SimRobot* (cf. Sect. 5.1) and, sharing the same code, in the general development support tool *RobotControl* (cf. Sect. 5.2). Under Windows, the processes are modelled as threads, i. e. all processes share the same address space. This caused some problems with global variables, because they are not shared on the real robots, but they are under Windows. As there is only a small amount of global variables in the code, the problem was solved "manually" by converting them into arrays, and by maintaining unique indices to address these arrays for all threads.

Open-R Emulator under CygWin. The environment was also implemented on the so-called Open-R emulator that allows parts of the robot software to be compiled and run under Linux and CygWin. The *router* (section 5.3) that functions as a mediator between the robots and RobotControl was implemented on this platform.

2.1.4 Math Library

Although the mathematical functions provided on different platforms are the same, it turned out that some functions are not working correctly on the AIBO robots. In addition, the runtime of mathematical functions is more crucial on embedded systems as the AIBO robots. Therefore, a math library was implemented that completely encapsulated the C++ math functions. This math library is called *GTMath*. The goals of the library were to be able to replace erroneous functions by working ones and to have the ability to test alternative implementations for trigonometric functions, which are less precise but faster. In addition, *GTMath* provides some higher data types for angles, vectors (two and three dimensional), matrices (three dimensional), rotation matrices, and translation matrices (two and three dimensional).

2.1.4.1 Class Hierarchy

In order to be able to change the implementation of fundamental data types such as *Angle* and *Value* as well as the implementation of trigonometric functions without editing all files in the

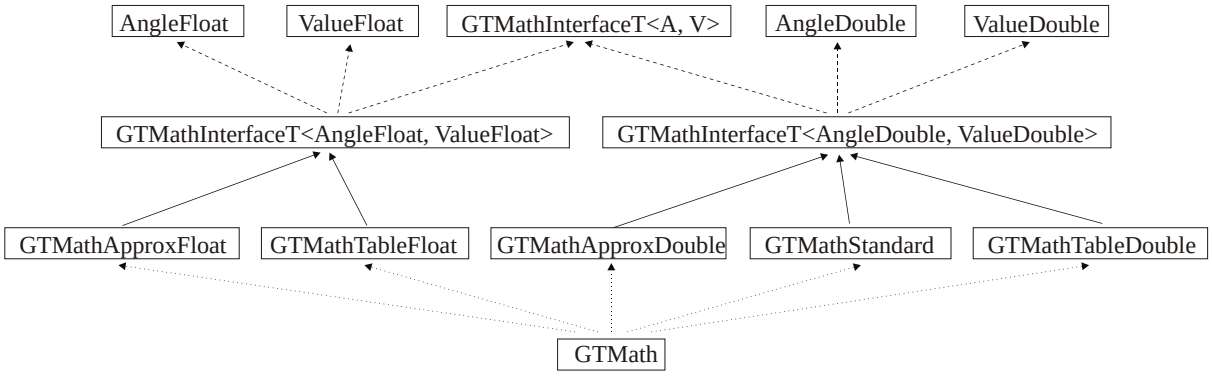


Figure 2.1: Class Structure of GTMath. *AngleDouble*, *ValueDouble*, *AngleFloat* and *ValueFloat* are the fundamental data types. *GTMathInterfaceT<>* defines the required functions. *GTMathApproxFloat*, *GTMathTableFloat*, *GTMathApproxDouble*, *GTMathTableDouble* and *GTMathStandard* are different implementations of the math library and *GTMath* is the class that selects the current implementation and that defines higher data types.

GT2002 project, a modular architecture was chosen. The design was divided in four layers (cf. Fig. 2.1): implementations of the fundamental data types angle and value, an interface which implements all necessary functions virtually, the different implementations of the trigonometric functions using different data types, and the class *GTMath* that simply inherits from one of the implementations to provide a constant interface to all other modules in the code. The math library is configurable by compiler switches that can be set in the configuration file *GTMathConfig.h*[1].

The interface *GTMathInterfaceT<Angle,Value>* is a template class. All functions needed in all implementations are defined in *GTMathInterfaceT<Angle,Value>* virtually. The data types *Angle* and *Value* are included by the template arguments.

Each implementation inherits *GTMathInterfaceT<Angle,Value>* and implements all functions. There are implementations using the standard functions provided by *<math.h>*, implementations using look-up-tables and implementations using polynomial approximations (taken from [10]).

The class *GTMath* inherits one of these implementations. Additionally these class implements all higher data types, which use *Angles* and *Values* but are independent of their implementation. *GTMath* provides vectors for any data types with two or three elements (Vector2<T> and Vector3<T>), 3×3-matrices (*Matrix3x3<T>*, rotation matrices and translation matrices for two and three dimensions (*Pose2D* and *Pose3D*).

[1] All files of the math library can be found in *T:\GT2002\Src\Tools\Mathlib*.

2.1.4.2 Provided Data Types

Angle is the data type for angles. It provides member functions for converting to and from degrees, radians and micro radians as well as the function *normalize* to map the angle to $[-\pi, \pi[$.

Value is the standard data type for all other floating point variables. Value has no special functions. The data type *Value* is a *typedef* to *double* or *float*.

Vector2<T> and Vector3<T> are template classes for vectors with two or three elements, respectively. They provide operators for the inner product and the cross product (^ operator) of two vectors (cross product only for *Vector3<T>*), and functions for the Euclidean length, transposition, normalization, and the angle between the vector and the x-axis (only *Vector2<T>*).

Matrix3x3<T> is a template class for 3×3-matrices. It provides operators to add and multiply two matrices and operators to multiply a matrix and a vector.

RotationMatrix is a matrix especially for rotations. *RotationMatrix* has various functions: functions to rotate the matrix around all axes, functions returning the actual rotation around all axes, and a function to invert the rotation matrix.

Pose2D and Pose3D are transformation matrices in two and three dimensions. They can be multiplied with vectors and *Pose2D* or *Pose3D*, respectively. In addition they can be rotated by angles and translated by vectors.

2.2 Multi-Team Support

The major goal of the architecture presented in this chapter is the ability to support the collaboration between the university-teams in the German national team. Some tasks may be solved only once for the whole team, so any team can use them. Others will be implemented differently by each team, e. g. the behavior control. A specific solution for a certain task is called a *module*. To be able to share modules, interfaces were defined for all tasks required for playing robot soccer in the Sony Legged League. These tasks will be summarized in the next section. To be able to easily compare the performance of different solutions for same task, it is possible to switch between them at runtime. The mechanisms that support this kind of development are described in section 2.2.2 and in appendix E. However, a common software interface cannot hide the fact that some implementations will need more processing time than others. To compensate for these differences, each team can use its own *process layout*, i. e. it can group together modules to processes that are running concurrently (cf. Sect. 2.2.3).

2.2.1 Tasks

Figure 2.2 depicts the tasks that were identified by the GermanTeam for playing soccer in the Sony Legged Robot League. They can be structured into five levels:

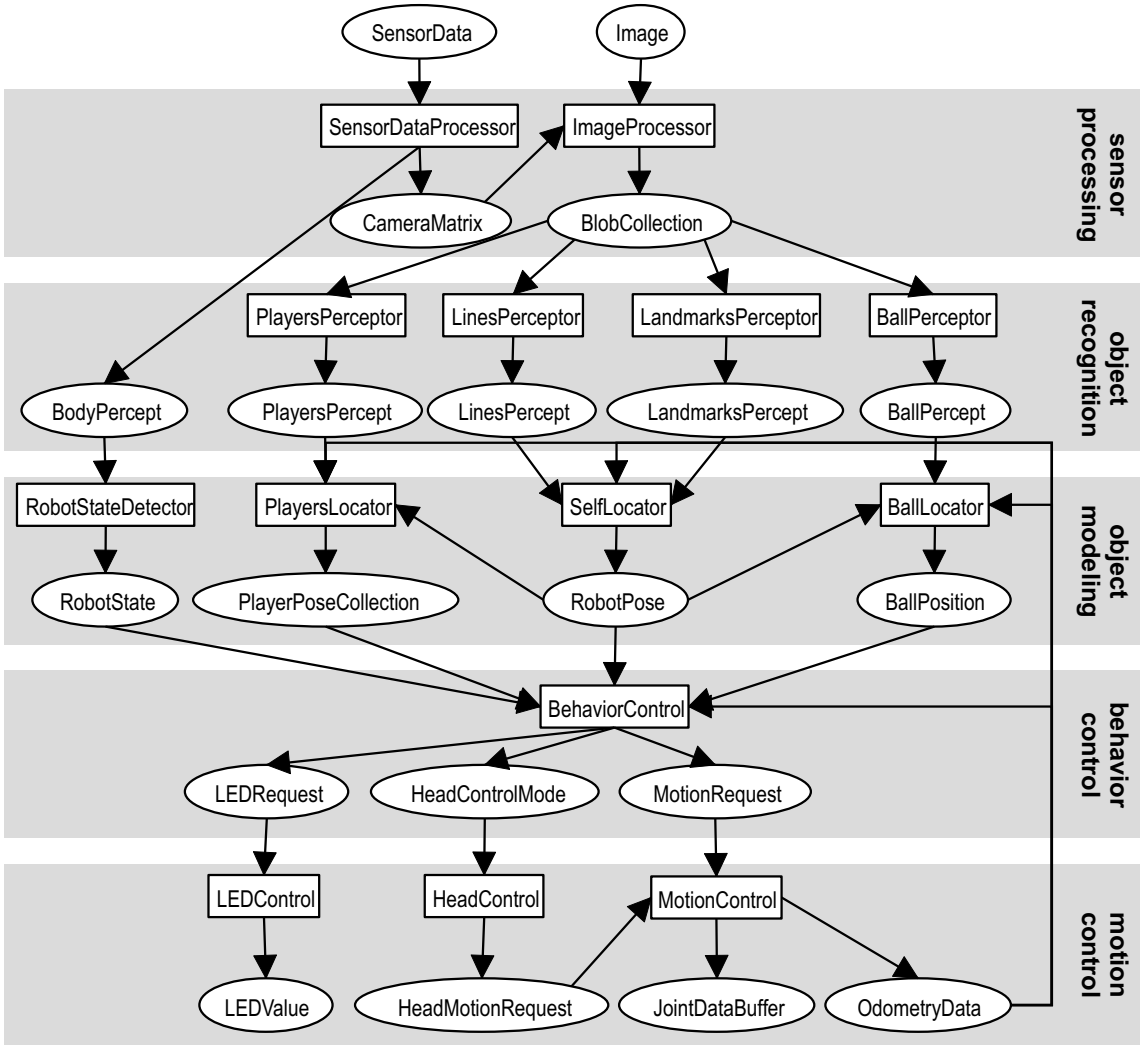


Figure 2.2: The tasks identified by the GermanTeam 2002 for playing soccer.

Sensor Data Processing. On this level, the data received from the sensors is preprocessed. For instance, the image delivered by the camera is segmented, and then it is converted into a set of blobs, i. e. image regions of the same color class. The current states of the joints are analyzed to determine the point the camera is looking at. In addition, further sensors can be employed to determine whether the robot has been picked up, or whether it felt down.

Object Recognition. On this level, the information provided by the previous level is searched to find objects that are known to exist on the field, i. e. landmarks (goals and flags), field lines, other players, and the ball. The sensor readings that were associated to objects are called *percepts*.

Object Modeling. Percepts immediately result from the current sensor readings. However, most objects are not continuously visible, and noise in the sensor readings may even result in

a misrecognition of an object. Therefore, the positions of the dynamic objects on the field have to modeled, i. e. the location of the robot itself, the poses of the other robots, and the position of the ball. The result of this level is the estimated *world state*.

Behavior Control. Based on the world state, the role of the robot, and the current score, the fourth level generates the behavior of the robot. This can either be performed very reactively, or deliberative components may be involved. The behavior level sends requests to the fifth level to perform the selected motions.

Motion Control. The final level performs the motions requested by the behavior level. It distinguishes between motions of the head and of the body (i. e. walking). When walking or standing, the head is controlled autonomously, e. g., to find the ball or to look for landmarks, but when a kick is performed, the movement of the head is part of the whole motion. The motion module also performs dead reckoning and provides this information to many other modules.

This grouping is not strict; it is still possible to implement modules that handle more than a single task. For instance, the grid-based vision module (cf. Sect. 3.2.2) directly determines percepts from the camera image, skipping the step of image preprocessing.

2.2.2 Debugging Support

One of the basic ideas of the architecture is that multiple solutions exist for a single task, and that the developer can switch between them at runtime. In addition, it is also possible to include additional switches into the code that can also be triggered at runtime. The realization is an extension of the debugging techniques already implemented in the code of the GermanTeam 2001 [3]: *debug requests* and *solution requests*. The system manages two sets of information, the current state of all *debug keys*, and the currently active solutions. Debug keys work similar to C++ preprocessor symbols, but they can be toggled at runtime (cf. Sect. E.2). A special infrastructure called *message queues* (cf. Sect. E.1) is employed to transmit requests to all processes on a robot to change this information at runtime, i. e. to activate and to deactivate debug keys and to switch between different solutions. The message queues are also used to transmit other kinds of data between the robot(s) and the debugging tool on the PC (cf. Sect. 5.2). For example, motion requests can directly be sent to the robot, images, text messages, and even drawings (cf. Sect. E.5) can be sent to the PC. This allows visualizing the state of a certain module, textually and even graphically. These techniques work both on the real robots and on the simulated ones (cf. Sect. 5.1).

2.2.3 Process-Layouts

As already mentioned, each team can group its modules together to processes of their own choice. Such an arrangement is called a *process layout*. The GermanTeam 2002 has developed its own model for processes and the communication between them:

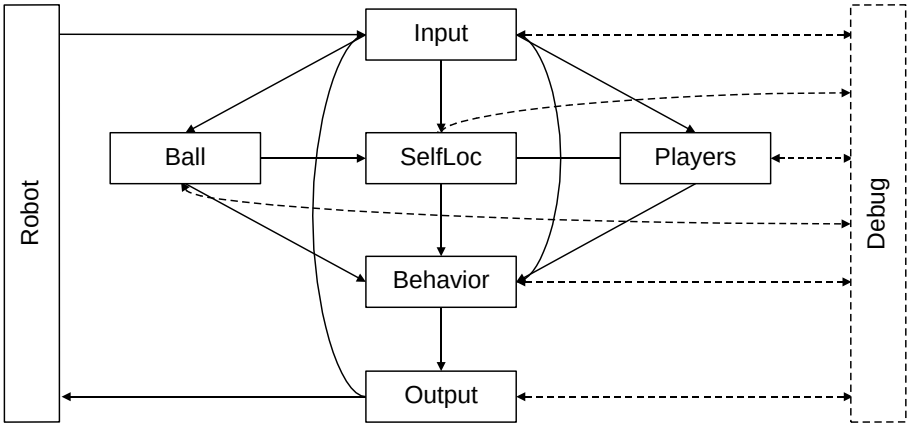


Figure 2.3: The process layout of the Bremen Byters.

2.2.3.1 Communication between Processes

In the robot control program developed by the GermanTeam 2001 for the championship in Seattle, the different processes exchanged their data through a shared memory [3], i. e., a blackboard architecture [12] was employed. This approach lacked of a simple concept how to exchange data in a safe and coordinated way. The locking mechanism employed wasted a lot of computing power and it only guaranteed consistence during a single access, but the entries in the shared memory could still change from one access to another. Therefore, an additional scheme had to be implemented, as, e. g., making copies of all entries in the shared memory at the beginning of a certain calculation step to keep them consistent. In addition, the use of a shared memory is not compatible to the new ability of the Sony AIBO robots to exchange data between processes via a wireless network.

The communication scheme introduced in 2002 addresses these issues. It uses standard operating system mechanisms to communicate between processes, and therefore it also works via the wireless network. In the approach, no difference exists between inter-process communication and exchanging data with the operating system. Only three lines of code are sufficient to establish a communication link. A predefined scheme separates the processing time into two communication phases and a calculation phase.

The inter-object communication is performed by *senders* and *receivers* exchanging *packages*. A sender contains a single instance of a package. After it was instructed to send the package, it will automatically transfer it to all receivers as soon as they have requested the package. Each receiver also contains an instance of a package. The communication scheme is performed by continuously repeating three phases for each process:

1. All receivers of a process receive all packages that are currently available.

2. The process performs its normal calculations, e. g. image processing, planning, etc. During this, packages can already be sent.

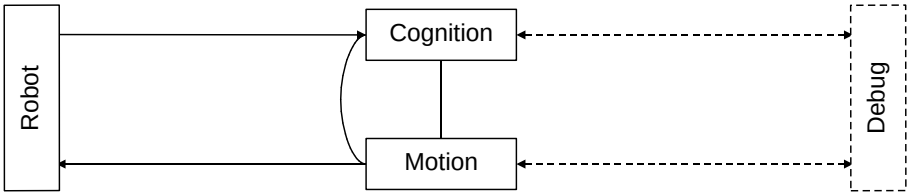


Figure 2.4: The process layout of Humboldt 2002.

3. All senders that were directed to transmit their package and have not done it yet will send it to the corresponding receivers if they are ready to accept it.

Note that the communication does not involve any queuing. A process can miss to receive a certain package if it is too slow, i. e., its computation in phase 2 takes too much time. In this aspect, the communication scheme resembles the shared memory approach. Whenever a process enters phase 2, it is equipped with the most current data available.

Both senders and receivers can either be blocking or non-blocking objects. Blocking objects prevent a process from entering phase 2 until they were able to send or receive their package, respectively. For instance, a process performing image segmentation will have a blocking receiver for images to avoid that it segments the same image several times. On the other hand, a process generating actuator commands will have a blocking sender for these commands, because it is necessary to compute new ones only if they were requested for. In that case, the ability to immediately send packages in phase 2 becomes useful: the process can pre-calculate the next set of actuator commands, and it can send them instantly after they have been asked for, and afterwards it pre-calculates the next ones.

The whole communication is performed automatically; only the connections between senders and receivers have to be specified. In fact, the command to send a package is the only one that has to be called explicitly. This significantly eases the implementation of new processes.

2.2.3.2 Different Layouts

The figures 2.3, 2.4, and 2.5 show three different process layouts. All of them contain a debug process that is connected to all other processes via message queues. Note that message queues are transmitted as normal packages, i. e. a package contains a whole queue. Comparing the three process layouts, it can be recognized that the Bremen Byters try to parallelize as much as possible[2]; while Humboldt 2002 focuses on using only a few processes, i. e. the first four levels of the architecture (cf. Fig. 2.4) are all integrated into the process *Cognition*. The Darmstadt Dribbling Dackels use a compromise between both approaches, integrating the first two levels into the process *Perception*, and the levels three and four into *Cognition* (cf. Fig. 2.5). In the layout of the Bremen Byters, one process is used for each of the levels one, four, and five, and three processes implement parts of the levels two and three, i. e. the recognition and the modeling of individual

[2]In the code release, the process *Ball* has been integrated into the process *Input*, because the ball recognition shall run at the same speed as the image processing.

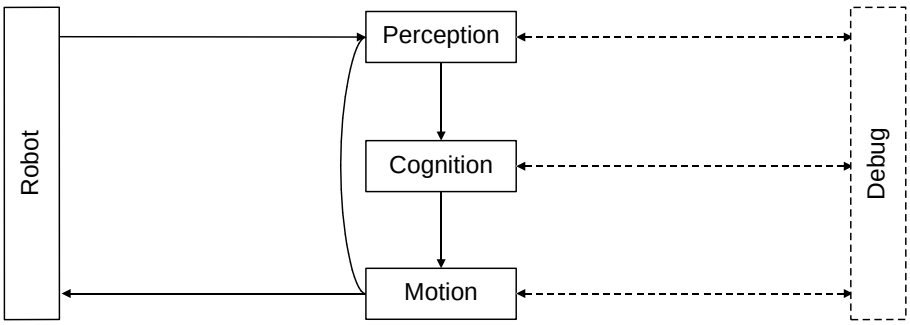


Figure 2.5: The process layout of the Darmstadt Dribbling Dackels.

aspects of the world state are grouped together (cf. Fig. 2.3). Odometry is used to decompose information that is dependent: although both the *Players* process and the *Ball* process require the current pose of the robot, they can run in parallel to the self-localization process, because the odometry can be used to estimate the spatial offset since the last absolute localization. This allows running the ball modeling with a high priority, resulting in a fast update rate, while the self-localization can run as a background process to perform a computationally more expensive method.

In Fukuoka, the GermanTeam used the simple process layout that was introduced by Humboldt 2002, because the more complex layout of the Bremen Byters turned out to have more disadvantages than advantages in timing measurements. The layout of the Darmstadt Dribbling Dackels had no advantages over the Humboldt 2002 layout, because the processes *Perception* and *Cognition* run more or less synchronized, i.e. *Cognition* always waits for *Perception* to deliver percepts, imitating the behavior of a single process.

2.2.4 Make Engine

Using different process layouts requires a quite sophisticated engine to compile the source code. As it is desirable that each process only contains the code that it needs, highly complex dependencies exist between compilation targets and the source files. For the code that is compiled for Microsoft Windows, process layouts can be represented easily by different project configurations. In addition, it is not required to determine the source code relevant for each process, because under Windows, processes are implemented as threads, and these threads are all part of the same program.

However, on the AIBO, each process is a different binary file, and because memory consumption is crucial, processes should be as small as possible, i. e. only the object files required by a process should be linked together.

2.2.4.1 Dependencies

The directory structure of the source code of the GermanTeam does not reflect which source file belongs to which binary, because the source files are grouped based on the selected process

layout. In one layout, e. g., several files share the same process while they are distributed over multiple processes in another layout.

Generating dependencies, creating object files, and linking them together is quite time consuming, especially in huge projects that require permanent modifications, expansions, testing and fine-tuning. Therefore, one major goal of implementing a *make engine* was to execute only the steps absolutely necessary to get a complete build without missing any modifications in source code.

Therefore, a flexible way to generate dependencies between source files and binaries was required. Object dependencies (**.odeps*) are generated by compilers such as the gcc. The object files required for a certain binary can easily be determined if the code follows some usual conventions such as the existence of header files (**.h*) and implementation files (**.cpp*), and if all items declared in a header file are either implemented by an implementation file with the same base name, or by the header file itself.

2.2.4.2 Realization

For each combination of the chosen process layout, build variant and compiler used, the make engine uses a separate build directory (*$PDIR*, located in *T:/GT2002/Build*) to avoid conflicts between different builds as well as the compulsion for a complete rebuild after changing the process layout, build variant, or compiler. Each such *$PDIR* has two subdirectories: *bin* contains the binaries and *obj* contains dependencies and objects in the same subdirectory structure as the source code. All these directories will be (re)generated on the first usage and after cleaning or rebuilding.

The following dependencies are maintained:

File lists (**.list**) containing lists of all **.cpp* files and their direct and indirect includes are generated for each directory to detect all changes possibly influencing the dependencies as fast as possible.

Object dependencies (**.odeps*) are generated for all **.cpp* files in one directory by using a compiler (time consuming). They only have to be (re)generated if the **.list** file in that directory has changed.

Binary dependencies (*.bindeps) only have to be (re)generated if the object dependencies have changed. This is done by recursively adding all **.cpp* files belonging to header files included from **.cpp* already determined as dependency. Binaries only have to be (re)linked if at least one of the objects belonging to them has changed—according to the appropriate **.bindeps* file.

To distinguish between files that changed and files that were just rebuilt without a change, new files are generated with the extension **.new*, and they will only replace the original file if they differ to avoid uselessly executing dependency chains. For the same reason, **.listakt* files conserve the time stamp of the last modification resulting from changing dependencies in that directory and **.listlast* files store complete lists of files and their includes from the last build.

2.2.4.3 Automation and Integration

It is possible to update or completely rebuild a certain process layout (e. g. HU1) in a special build variant (e. g. Debug) with a single command, either from the command line, e. g. with *./GT.bash HU1 Debug*, or from the Microsoft Developer Studio, e. g. by selecting *Rebuild* or *Rebuild All*. For the latter case, the messages generated by the build process are converted to a format that is understood by the Developer Studio. Thus, the list of errors and warnings can be browsed by the usual commands, presenting the source files the messages refer to.

Chapter 3

Modules in GT2002

The GermanTeam has split the robot's information processing onto *modules* (cf. Sect. 2.1). Each module has a specific task and well-defined interfaces. Different exchangeable *solutions* exist for many of the modules. This allows the four universities in the team to test different approaches for each task. In addition, existing and working module solutions can remain in the source code while new solutions can be developed in parallel. Only if the new version is better than the existing ones (which can be tested at runtime), it becomes the *default solution*. Mechanisms for declaring modules and for switching solutions at runtime are described in section E.6.

This chapter describes most of the modules that were implemented. For some solutions only the chosen approach is figured out in detail. The following table lists all modules with all solutions in the code release. The *default solution* is the solution that was used by the GermanTeam for the RoboCup competitions in Fukuoka (marked underlined).

module / task	solutions	from
SensorDataProcessor: Processing of body sensor data.	*DefaultSensorDataProcessor*: Calculates the offset and rotation of the camera relative to the ground and detects pressed switches (cf. Sect. 3.1).	Darmstadt
ImageProcessor: Perceiving visual percepts from images.	*BlobImageProcessor*: Generates Blobs using RLE encoding and a *FloodFill* algorithm. Percepts are calculated from the blobs. It was used by Bremen, Darmstadt, and Dortmund at the German Open 2002 (cf. Sect. 3.2.1).	Bremen
	GridImageProcessor: Scans regions near the horizon for interesting objects using a grid. It was used by Berlin at the German Open 2002 (cf. Sect. 3.2.2).	Berlin
LinesPerceptor: Recognition of field lines in images.	*DefaultLinesPerceptor*: Detects lines by scanning the image along vertical lines. Until now not used during a game (cf. Sect. 3.2.2.6).	Berlin

BallLocator: Modeling of ball position	*DefaultBallLocator*: Used for the soccer games (cf. Sect. 3.4).	Bremen
	JumpingBallLocator: Adds a noise to the ball position for testing the robustness of behaviors.	Berlin
	MultipleBallLocator, SampleMultiBallLocator, StickyBallLocator: Several approaches for the ball challenge.	Bremen
PlayersLocator: Modeling of player positions.	*GT2001PlayersLocator*: Player modelling using a grid from the GermanTeam's 2001 project. Not used in the games (cf. Sect. 3.5).	Bremen
SelfLocator: Estimation of the robot's pose.	*LinesSelfLocator*: Monte-Carlo-based and uses the field lines. Experimental and up to now only tested in the simulator (cf. Sect. 3.3.3).	Bremen
	MonteCarloSelfLocator: Monte-Carlo-based and uses flags and goal posts (cf. Sect. 3.3.2).	Bremen
	SingleLandmarkSelfLocator: Estimating the robot's pose using only single landmarks seen. Used by Darmstadt at the German Open 2002 (cf. Sect. 3.3.1)	Darmstadt
RobotStateDetector: Modeling of body sensor events.	*DefaultRobotStateDetector*: Detects fall downs and measures, for how long switches were pressed.	Darmstadt
BehaviorControl: Decision making. (cf. Sect. 3.6	*BallChallengeBehaviorControl*: Uses potential fields. It was developed by Bremen for the German Open and was applied for the ball challenge (cf. Sect. 4.3).	Bremen
	TUDXMLBehaviorControl: A state machine formalized in XML was developed by Darmstadt for the German Open (cf. Sect. 3.6.1).	Darmstadt
	XabslBehaviorControl: Behavior modules that contain state machines and that are organized in a tree were formalized in the XML language XABSL (cf. Sect. 3.6.2).	Berlin
HeadControl: Control of head movement.	*GT2002HeadControl*: A collection of the most promising head control approaches (cf. Sect. 3.7.3).	All
	SimpleHeadControl: Works together with the *TUDXMLBehaviorControl*.	Darmstadt
LEDControl: Sets the robot's LEDs.	*DefaultLEDControl*: Processes the requests from the behavior control.	Darmstadt
SpecialActions: Kicks and other moves.	*GT2001MotionNetSpecialActions*: Executes special actions that were described in a high level motion language. (cf. Sect. 3.7.2)	Berlin Darmstadt Dortmund

WalkingEngine: Walking	*FixedSequenceWalkingEngine*: Executes precalculated gait patterns. Experimental.	Darmstadt
	FourierWalkingEngine: Uses *Fourier Synthesis* for walking. Experimental.	Berlin
	InvKinWalkingEngine: Uses inverse kinematics. Based on this algorithms Darmstadt and Dortmund developed their walking styles for the German Open 2002. Bremen used the walking from CMU, Berlin from UNSW. Both are not in the code release because of licence incompatibilities (cf. Sect. 3.7.1).	Darmstadt
MotionControl: Setting of the joint values.	*DefaultMotionControl*: Integrates head motions, walk motions, special actions, and LED values (cf. Sect. 3.7).	Darmstadt
	DebugMotionControl: A tool for developing new motions. Works together with the *MofTester Dialog* in the RobotControl application (cf. Sect. J.5.4).	Darmstadt
SoundOutControl: Generation and sending of audio data.	*DefaultSoundOutControl*: Plays wave files for the pattern challenge and for debugging purposes.	Dortmund
SoundInProcessor: Processing of audio data.	*DefaultSoundInProcessor*: Experimental acoustic message detection. Used by Dortmund during the German Open.	Dortmund
SensorDataToMotionRequest: Nice demos.	*Braitenberg*: Implementation of a *Braitenberg Vehicle* on the AIBO. Lets the robot walk to the brightest spot in a room.	Berlin
	WalkDemoSensorDataToMotionRequest: A demo for the walking capabilities of the robot. Walk direction and speed can be set by bending the robot's head.	Berlin

3.1 Body Sensor Processing

The task of the *SensorDataProcessor* is to take the data provided by all sensors except the camera, and to store them, marked with a time stamp, in a buffer. This buffer is used to calculate average sensor values over the last *n* ticks, or to interpolate the values to get estimate sensor values for a given point in time (usually the arrival of a new camera image).

The measurements of the acceleration sensors are used to calculate the tilt and roll of the robot's body. By comparing long term averages and short term averages of these measurements, it is possible to determine whether the robot has been lifted up or whether is has fallen down.

Figure 3.1: From an image to blobs. a) The original image, b) The segmented image, c) The detected blobs.

For every incoming image, the *SensorDataProcessor* calculates a matrix that represents the pose of the camera relative to the robot's body origin. This allows the coordinates of objects detected in camera images to be transformed into the robot's system of coordinates.

3.2 Vision

The vision module works on the images provided by the robot's camera. The output of the vision module fills the data structure *PerceptCollection*. A percept collection contains information about the relative position of the ball, the field lines, the goals, the flags, and the other players. Positions and angles in the percept collection are stored relative to the robot.

Goals and flags are represented by four angles. These describe the bounding rectangle of the landmark (top, bottom, left, and right edge) with respect to the robot. When calculating these angles, the robot's pose (i.e. the position of the camera relative to the body) is taken into account. If a pixel used for the bounding box was on the border of the image, this information is also stored.

Field lines are represented by a sets of points (2-D coordinates) on a line. The ball position and also the other players' positions are represented in 2-D coordinates. The orientations of other robots are not calculated.

Two different approaches for the vision of the robot were implemented, both using the high resolution image (176×144 pixels). The blob-based vision was used by the GermanTeam during the RoboCup 2001 competition in Seattle. This year's code is an improved version of it. The grid-based vision approach was first used at the RoboCup German Open 2002 by the team of the *Humboldt-Universität zu Berlin* and later adopted by the GermanTeam for the RoboCup 2002 competition.

3.2.1 Blob-Based Vision

The blob-based vision module consists of several submodules for special tasks. The *FloodFillRLEBlobDetector* is responsible for segmentation (cf. Fig. 3.1b) and blob recognition (cf.

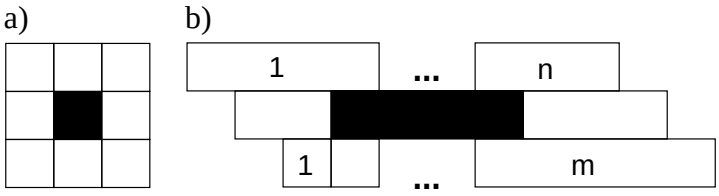


Figure 3.2: a) A pixel with eight neighbors. b) A run with upper and lower neighbors.

Fig. 3.1c) in images. It fills the data structure *BlobCollection* and passes it to the object recognition modules that generate percepts: *LandmarksPerceptor*, *BallPerceptor*, and *PlayersPerceptor*.

The modules for blob-based vision were used by most teams at the RoboCup German Open 2002.

3.2.1.1 Flood-Fill RLE Blob Detector

The *FloodFillRLEBlobDetector* performs blob recognition on run length encoded images.

Segmentation and RLE Compression. The assignment of a pixel's color class is done via a lookup table dividing the YUV color space into cubes of $4 \times 4 \times 4$ units. This table is created off-line by using the tools in *RobotControl*, described in section J.6.

To decrease data and by that way achieve a better performance in finding blobs, every image is run length encoded during segmentation. To use the *FloodFill* algorithm on RLE compressed images instead of pixel images, one needs to know the neighbors of each run. In contrast to a pixel which always has eight neighbors (cf. Fig. 3.2a), a run may have $1 \ldots n$ upper and $1 \ldots m$ lower neighbors ($1 \leq n, m \leq$ number of pixels per row, cf. Fig. 3.2b). Their computation during the algorithm's runtime may become quite expensive. Therefore, all runs are enriched with information about their left upper and lower neighbors during their creation.

Detecting Blobs Using the *FloodFill* Algorithm. The *FloodFill* algorithm is used for blob detection. It is a standard filling algorithm known from computer graphics, e. g. described in [8]. It is a recursive method which detects all runs of the same color belonging to an area. Starting at one arbitrary run it compares every visited run with it's neighbors until there is no unknown run left in the area.

During this process statistics may be made about the shape of this area. This may be any technique from simple bounding boxes to complex splines. As best trade-off between processing time and the information gained about the areas, the shapes of blobs are described by convex octagons. These octagons consist of eight points: the area (labelled by their orientation): the highest (N), lowest (S), most right (E), most left (W), upper right (NE), upper left (NW), lower left (SW), and lower right (SO). Thereby, the last four points are described by their Manhattan distance to the four corners of the image.

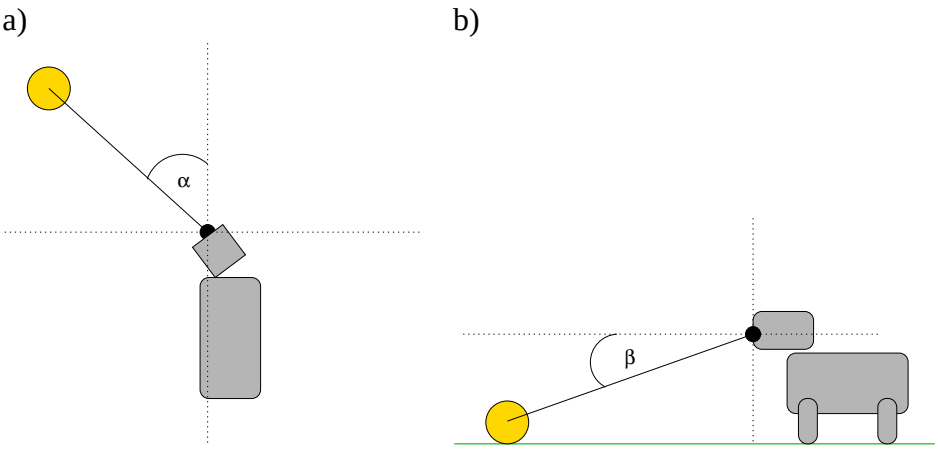


Figure 3.3: Angles to a point relative to the camera. a) Horizonal. b) Vertical.

Conversion to Angles. During area detection, only whole-numbered coordinates of pixels are used. To determine the real positions of objects relative to the robot, position and direction of the camera have to be considered.

To simplify object recognition in the other modules, each octagon is converted to a blob which describes a convex polygon. Every point in such a polygon is represented by two angles α and β relative to the camera position, as shown in figure 3.3.

Competitiveness of this Approach. The *FloodFillRLEBlobDetector* was implemented because the implementation of [15] used a year ago showed a weak performance. A first test in the last years debugging environment showed a 20-90% higher frame rate compared to the previous approach. Running on a Sony robot, the average processing time for a single image is about 23 ms.

3.2.1.2 Landmarks Perceptor

The recognition of the flags and the goals is required to perform the self-localization of the robots on the field. Each flag consists of a unique combination of colors, whereas each goal has only one color. As the goals are located on the green carpet, they can also be detected as an arrangement of two colors. This is important, because the blob collection may contain areas with random colors resulting from objects above the field, or from mixtures between colors of different objects on the field. Grouping two colored areas together provides the possibility to use spatial constraints to filter out most misreadings.

The *LandmarksPerceptor* is based on the approach from the previous year. The new landmarks perceptor works on blobs with vertices in angular coordinates, whereas the previous version performed the conversion from image coordinates to robot coordinates by itself. In addition, the new perceptor uses more constraints to improve the recognition of landmarks.

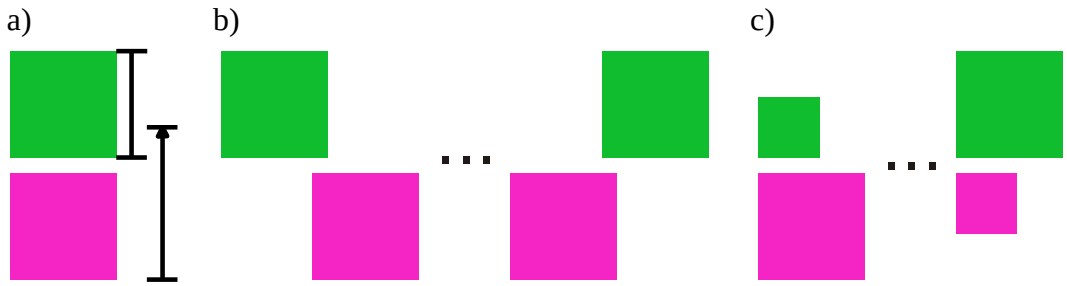


Figure 3.4: Constraints for the recognition of flags. a) Vertical. b) Horizontal. c) Size ratio.

Spatial Constraints. The landmarks perceptor uses the bounding rectangles of the blobs. These rectangles are oriented in parallel to the horizontal and vertical axes. Therefore, vertical and horizontal relations can be handled separately. The perceptor uses the thirteen relations of Allen's [2] calculus to describe the spatial constraints. To encode, e. g., that blob B is immediately above blob A, it can be written:

$$A_y \text{ meets } B_y \tag{3.1}$$

The subscript y denotes that the relation is applied to the vertical ranges of A and B. However, the relations of the calculus are to precise to be used directly, e. g., *meets* means that A exactly ends where B begins, and the image processing method employed cannot deliver this precision. Therefore, one range is enlarged a little bit (cf. Fig. 3.4a), so that "above" can be encoded as

$$\text{enlarged}(A_y) \text{ overlaps } B_y. \tag{3.2}$$

In addition, "above" requires both blobs to overlap in horizontal direction (cf. Fig. 3.4b). In Allen's calculus, this can be written as

$$\neg(A_x < B_x \vee A_x \text{ meets } B_x \vee A_x \text{ metBy } B_x \vee A_x > B_x). \tag{3.3}$$

Combining both expressions, the final definitions of "above" and its complement "below" are

$$\begin{aligned} A \text{ above } B &= \text{enlarged}(A_y) \text{ overlaps } B_y \wedge \\ &\quad \neg(A_x < B_x \vee A_x \text{ meets } B_x \vee A_x \text{ metBy } B_x \vee A_x > B_x) && (3.4) \\ A \text{ below } B &= B \text{ above } A. && (3.5) \end{aligned}$$

Recognition of Flags. To find the flags, the algorithm runs through all pink blobs, sorted by size in descending order, and tries to find an appropriate partner from the yellow, green, and sky blue blobs. Again, larger blobs are found first. A good partner is a blob that is either above or below the pink blob (cf. Fig. 3.4b), and that has nearly the same size (cf. Fig. 3.4c). However, the

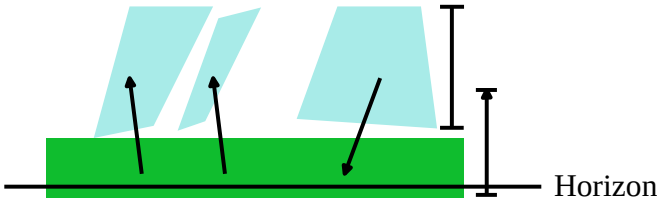


Figure 3.5: Recognition of a goal.

height of the upper blob can be arbitrarily small if it cuts the border of the image, i. e. it cannot completely be seen. This can be the case if the robot is very close to a flag.

For each pink blob, the search stops when the first appropriate counterpart was found, assuming that the largest pair of blobs fulfilling all constraints is always the most likely one. In addition, blobs are never assigned to more than one flag, and a blob that is part of a flag cannot be part of a goal.

Recognition of Goals. The recognition of the goals is a little bit more complicated, because other robots can partially hide them, and therefore split them into more than a single blob. Therefore, the approach for detecting a goal is to search for the largest blob with the goal's color and an area of at least 100 pixels that is not a part of a flag (cf. the upper right blob in Fig. 3.5), then to find the largest green area below that blob (cf. the lower blob in Fig. 3.5) and finally to accumulate all blobs with the color of the goal above that green area (cf. the other upper blobs in Fig. 3.5). Note that the lower boundary of the green blob shall also be below the horizon, i. e. 0° in angular coordinates, because the camera is always above the carpet.

3.2.1.3 Ball Perceptor

The ball is the most significant object in the soccer game. Therefore, it is important to recognize it as often and as precise as possible. It is also essential to exclude wrong ball detections resulting from misreadings in the color image.

The *BallPerceptor* is mainly based on the approach from the previous year. The main difference is that the new ball perceptor—as all perceptors—works on blobs with vertices in angular coordinates instead of blobs with vertices in pixel coordinates. Thus all angular influences of the head position have already been integrated into the blobs.

Blob Filtering. It is assumed that the ball is either located on the carpet, or that it is so close that the carpet cannot be seen anymore. Therefore, the ball perceptor recognizes the ball as the largest orange blob that either extends more than half of the image width or height, or that is neighbored by a green blob.

Center Point and Diameter. To determine the position of the ball, two features must be extracted from the blob collection: the ball's diameter and its center point. The diameter in the

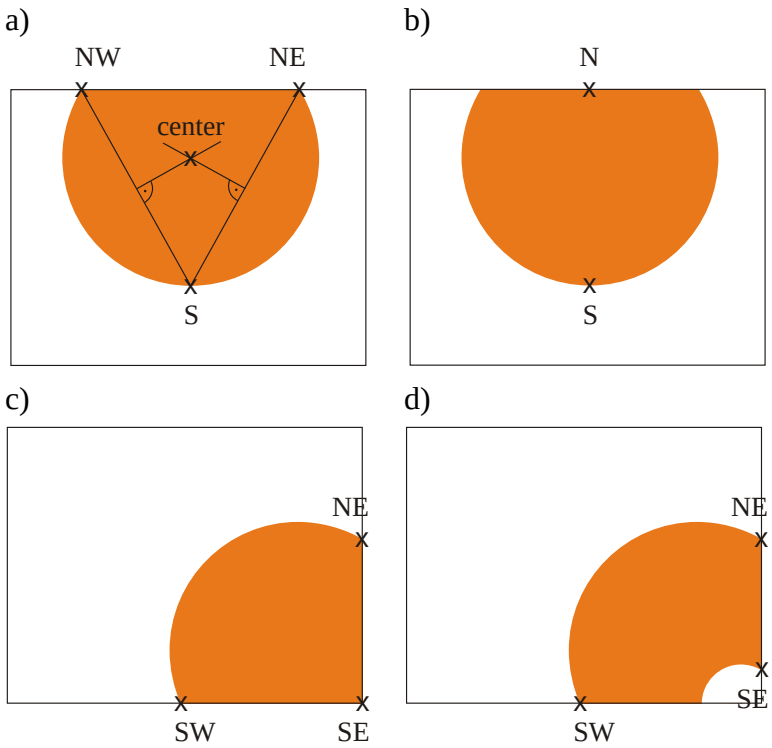


Figure 3.6: Calculating the center point. a) Intersecting the middle perpendiculars. b) *S* can be used for the calculation of the center, *N* cannot. c) *NE* and *SW* lie on the border of the ball, *SE* does not. d) A spot light in the corner. Although not in a corner, *SE* is not located on the border of the ball.

image can be related to the diameter of the ball in reality to determine its distance from the camera. Together with the center point, the ball's relative position according to the camera can be calculated.

Although the calculation of both diameter and center seems to be straightforward, a little bit more effort has to be spent to determine the location of the ball especially in situations when it cannot be seen completely, i. e. only a part of the ball is shown in the camera image. In such cases, the width and the height of a blob cannot be used to determine its diameter, because it is partially hidden, and the center of such a blob is not the center of the ball.

Intersection of Middle Perpendiculars. Therefore, the ball perceptor uses the intersection point of two middle perpendiculars to determine the center point of the ball (cf. Fig. 3.6a). In comparison to the usage of width and height, the advantage of this approach is that three arbitrary points on the border of the ball are enough to calculate its center. As has been described in section 3.2.1.1, each blob consists of eight points. The major question for the approach presented here is, which three among these eight points are used to determine the ball's center. If the ball is completely contained in the image, i. e. no part sticks out to the sides, all eight points lie on the border of the ball. However, if the image does not contain the whole ball, some of the points

lie on the image border rather than on the border of the ball. Therefore, they cannot be used to determine the ball's center point.

Selecting Points. When deciding which points can be used, it must be distinguished between points representing horizontal or vertical extrema and points that stand for diagonal extrema. While horizontal and vertical maxima cannot be used if they are located on "their" border of the image, e. g., if the point *N* is at the upper edge of the image (cf. Fig. 3.6b), diagonal extrema have only to be excluded if they lie in an image corner (cf. Fig. 3.6c).

From the points that satisfy these conditions, the two are selected that are furthest away from each other. Then, a third point is chosen that is farthest away from the two points already selected, and that ensures that at least one of the three points chosen is not located at any image border, which reduces the possibility of calculating a wrong ball position from an orange blob with a white spot light in an image corner (cf. Fig. 3.6d).

After the three points were selected, the center point can be determined by intersecting the middle perpendiculars of lines connecting these points (cf. Fig. 3.6a). The largest distance of the center to one of the points chosen specifies the radius of the ball. Using this information, the direction and distance of the ball relative to the camera are known. As the angular coordinates of the vertices of the blobs already contain the rotations of the head, the direction to the ball is relative to the orientation of the robot, not of the camera. However, as the translational offset of the camera has not been taken into account yet, the relative offset of the ball to the body center is determined as the offset of the camera plus the offset of the ball.

3.2.1.4 Players Perceptor

To avoid collisions with other robots, it is necessary to recognize them. This is the task is of the *PlayersPerceptor*.

Choice and Interpretation of Blobs. As the cotton jerseys of the robots are red and blue, the technique implemented only uses blobs of these colors. As it is possible that several robots of the same team are in a single image, all blobs of one color have to be clustered, whereby each cluster is interpreted as one robot. The assignment of a blob to a cluster is done by simple heuristics, comparing the horizontal distances of blobs. If two blobs overlap or are quite close to each other, they are assumed to belong to the same cluster.

Computation of Distances and Angles. The distance computation for a cluster is based on its size. In a first step, a bounding box is created around the cluster. The vertical angle between the upper and the lower border of this box is compared to the height of a real robot's tricot, including all parts from the legs to the head. Because of the movements of a robot, this height varies, but the resulting miscalculations are small and have to be tolerated. This technique was chosen because of the difficulties in segmenting the grey and black parts of a robot. Single blobs could also not be assigned to the according robot parts.

The angle to a robot is determined as the angle to the center of the according cluster.

Results. Using this approach, it is possible to determine the positions of robots being nearly completely in an image. The results are quite exact, but there are still some remaining problems: Because the number of recognized tricot markers per robot varies in every image due to the robot's direction and the lighting conditions, it is not possible to determine if parts of a robot are masked by another robot or if the robot is at the border of the image. Two robots of the same team standing very close to each other may be recognized as a single robot as well. These problems may lead to large miscalculations during the computation of distances and angles.

Furthermore, it is currently not possible to recognize the orientation of a robot.

3.2.2 Grid-Based Vision

The grid-based vision is a fast image processing module. Images are processed using the high resolution of 176×144 pixels, but looking only at a grid of less pixels.

The idea is that for feature extraction, a high resolution is only needed for small or far away objects. In addition to being smaller, such objects are also closer to the horizon. Thus only regions near the horizon need to be scanned at high resolution, while the rest of the image can be scanning using a relatively wide spaced grid.

When calculating the percepts, the robot's pose, i. e. its body tilt and head rotation at the time the image was acquired, is taken into account.

3.2.2.1 Using a Horizon-Aligned Grid

Calculation of the Horizon. The grid-based vision first calculates the position of the horizon in the image. The robot's lens projects the object from the real world onto the CCD chip. This process can be described as a projection onto a virtual projection plane arranged perpendicular to the optical axis with the center of projection C at the focal point of the lens. As all objects at eye level lie at the horizon, the horizon line in the image is the line of intersection between the projection plane P and a plane H parallel to the ground at height of the camera (cf. Fig. 3.7). The position of the horizon in the image only depends on the rotation of the camera and not on the position of the camera on the field or the camera's height.

For each image the rotation of the robot's camera relative to its body is stored in a rotation matrix. Such a matrix describes how to convert a given vector from the robot's system of coordinates to the one of the camera. Both systems of coordinates share their origin at the center of projection C. The system of coordinates of the robot is described by the x-axis pointing parallel to the ground forward, the y-axis pointing parallel to the ground to the left, and the z-axis pointing perpendicular to the ground upward. The system of coordinates of the camera is described by the x-axis pointing along the optical axis of the lens outward, the y-axis pointing parallel to the horizontal scan lines of the image, and the z-axis pointing parallel to the vertical edges of the image.

To calculate the position of the horizon in the image, it is sufficient to calculate the coordinates of the intersection points h_l and h_r of the horizon and the left and the right edge of the image in the system of coordinates of the camera. Let s be the half of the horizontal resolution

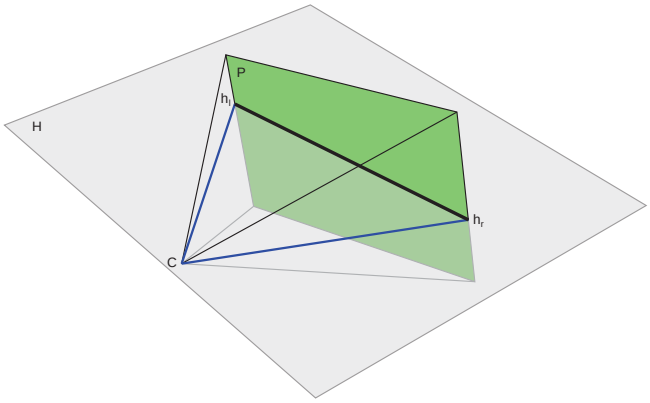


Figure 3.7: Construction of the horizon

of the image, α be the half of the horizontal opening angle of the camera. Then

$$h_l = \begin{pmatrix} \frac{s}{\tan\alpha} \\ s \\ z_l \end{pmatrix}, h_r = \begin{pmatrix} \frac{s}{\tan\alpha} \\ -s \\ z_r \end{pmatrix} \tag{3.6}$$

with only z_l and z_r unknown. Let

$$i = \begin{pmatrix} x \\ y \\ 0 \end{pmatrix} \tag{3.7}$$

be the coordinates of h_l in the system of coordinates of the robot. Solving the equation that describes the transformation between the two systems of coordinates

$$R \cdot i = h_l \tag{3.8}$$

with the rotation matrix

$$R = \begin{pmatrix} r_{11} r_{12} r_{13} \\ r_{21} r_{22} r_{23} \\ r_{31} r_{32} r_{33} \end{pmatrix} \tag{3.9}$$

leads to

$$z_l = -\frac{r_{32} s + r_{31} s \cdot \cot\alpha}{r_{33}}. \tag{3.10}$$

In the same way follows

$$z_r = -\frac{-r_{32} s + r_{31} s \cdot \cot\alpha}{r_{33}}. \tag{3.11}$$

Grid Construction and Scanning. The grid is constructed based on the horizon line, to which grid lines are perpendicular. The area near the horizon has a high density of grid lines, whereas the grid lines are coarser in the rest of the image (cf. Fig. 3.8a).

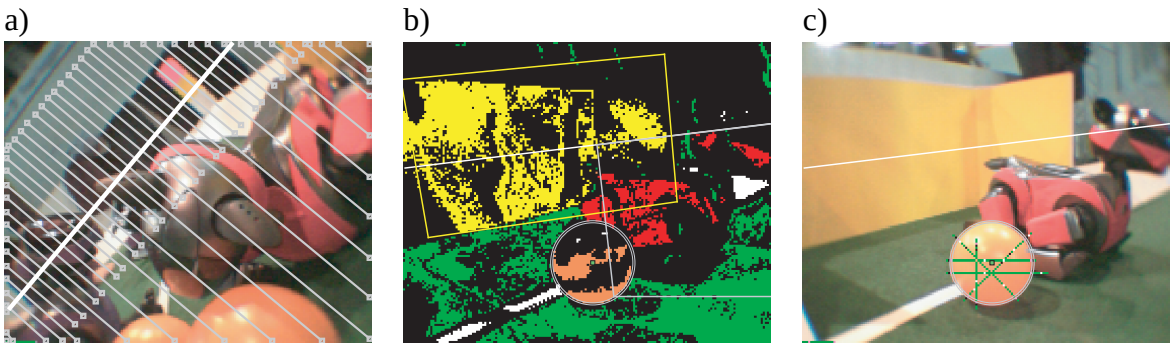


Figure 3.8: Percepts. a) Horizon-aligned grid b) Segmented image. Objects are recognized even with a poor color table. c) Recognition of the ball. Only the green pixels have been *touched* by the ball specialist.

Each grid line is scanned pixel by pixel from top to bottom. During the scan each pixel is classified by color. A characteristic series of colors or a pattern of colors is an indication of an object of interest, e. g. a sequence of two orange pixels is an indication of a ball; a sequence or an interrupted sequence of pink pixels followed by a green, skyblue, yellow, or white pixel is an indication of a flag; an (interrupted) sequence of skyblue or yellow pixels followed by a green pixel is an indication of a goal, and a sequence of red or blue pixels is an indication of a player.

Each time an indication of an object of interest is found, a *specialist* is started that checks whether there is such an object. If this is the case, the specialist finds out where the object is located precisely in the image, and it calculates the position of the object relative to the robot. The specialists are initialized with a *hot spot* that is lying inside the object with a high likelihood. This hot spot is determined from the pattern detected during the search in the grid.

3.2.2.2 Ball Specialist

Scanning the Ball. The ball specialist tries to determine as many pixels as possible at the border of the ball. From the initialization pixel the image is scanned for the border of the ball to the top, right, down, and left leading to a horizontal and a vertical *cut* of the ball. The middle perpendicular of the shorter of the two cuts provides a third cut. The center of this new cut is a good estimation of the center of the ball in most of the cases. From the estimated center a new scan for the border of the ball in the four diagonal directions is started. If one of the two first cuts meets an edge of the image, the ball specialist scans for the border of the ball in both directions along this edge. Each point possibly lying at the border of the ball that was found during this process is stored in a list. (cf. the green pixels in Fig. 3.8c and Fig. 3.9a).

Finding the Border of the Ball. The orange of the ball and the green of the field have completely different values in the u-channel of the yuv-image of the robot's camera. The border of the ball is characterized by a big decrease of the u-value from one pixel in the scan line to the next one. Unfortunately there is also a big decrease of the u-value from one to the next pixel if the scan line runs over a highlight on the surface of the ball caused by a floodlight (cf. Fig. 3.9b). Each decrease and increase of the u-value is marked (cf. the black and white pixels in Fig. 3.8c

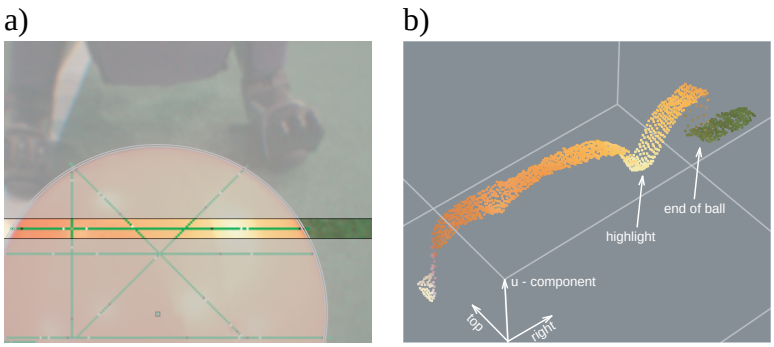


Figure 3.9: Ball specialist. a) Horizontal scan line. b) the u-component of the image near the scan line.

and Fig. 3.9a). A very big decrease or the second of two not directly subsequent decreases in the u-channel (not separated by an increase) are assumed to be the border of the ball.

A second criterion for the border of the ball is a sequence of three pixels of one of the *stop colors*: green, skyblue, yellow, red, and blue. In such a case one of the following pixels is assumed to be on the border of the ball: The last orange pixel, the pixel with the last significant decrease in the u-channel, or the last pixel before the first stop color.

Finding the Best Circle. As the points found on the border of the ball are measured values they will never lie on one and the same circle. First for each triple the resulting circle is calculated. In a second step each of the circles obtains a vote from each pixel having a distance of less than 3 pixels to the circle. The circle with most votes is assumed to be the border of the ball.

Calculating the Position of the Ball. Let v_c be the vector to the center of the ball in the system of coordinates of the camera. This vector is given by the center of the circle in the image, the opening angle of the camera, and the resolution of the image. Let $v_w = R \cdot v_c$ be the same vector in the system of coordinates of the world. Where the 3×3 matrix R describes the transformation between the two systems of coordinates. The intersection of the line described by v_w and a plane parallel to the ground at the level of the center of the ball is the position of the ball.

3.2.2.3 Flag Specialist

Scanning a Flag. The flag specialist measures the height and the width of a flag. From the initialization pixel the image is scanned for the border of the flag to the top, right, down, and left where top/down means perpendicular to the horizon and left/right means parallel to the horizon. This leads to a first approximation of the size of the flag. Two more horizontal lines are scanned in the pink part and if the flag has a yellow or a skyblue part, two more horizontal lines are also scanned there. The width of the green part of the pink/green flags is not used, because it is not always possible to distinguish it from the background. To determine the height of the flag, three additional vertical lines are scanned. The leftmost, rightmost, topmost, and lowest point found by these scans determine the size of the flag. The angles to the four edges of the flag are written into the *PerceptCollection*.

a) b)

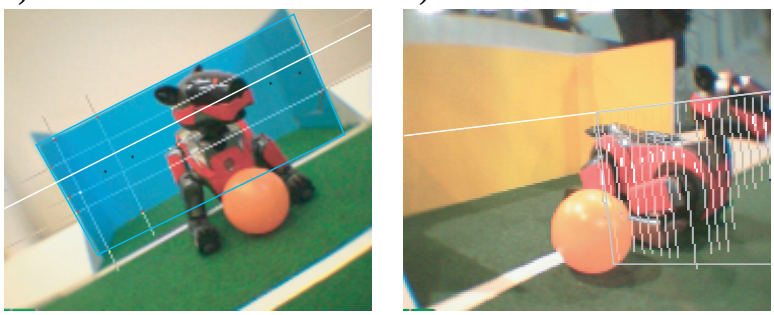


Figure 3.10: a) Recognition of the goal. Only the grey pixels have been *touched* by the goal specialist. b) Recognition of a player.

Finding the Border of a Flag. The flag specialist searches the last pixel having one of the colors of the current flag. Smaller gaps with no color are accepted. This requires the color table to be very accurate for pink, yellow, and skyblue.

3.2.2.4 Goal Specialist

The goal specialist measures the height and the width of a goal. From the initialization pixel the image is scanned for the border of the goal to the top, right, down, and left where again top/down means perpendicular to the horizon and left/right parallel to the horizon. This leads to a first approximation of the size of the goal. Two more horizontal lines and two more vertical lines are scanned in the approximated goal. (cf. the gray pixels in Fig. 3.10a).

To find the border of the goal the specialist searches the last pixel having the color of the goal. Smaller gaps with unclassified color are accepted. The maximal size in each direction determines the size of the goal. The angles to the four edges of the goal and the information whether the end of the goal is outside the image are written to the *PerceptCollection*.

3.2.2.5 Player Specialist

The initialization pixel is assumed to be at the left side of the robot. The player specialist scans lines perpendicular to the horizon beginning at the horizon until a green pixel is found. If there is a sequence of four subsequent vertical scan lines with an occurrence of green without an occurrence of red the process is stopped. (cf. Fig. 3.10b). The center of the lines and the lowest of the first occurrence of green determine the vector to a point at the ground below the robot. The intersection of the line defined by this vector and the ground yields an approximation for the position of the robot.

3.2.2.6 Lines Perceptor

The lines perceptor recognizes characteristic lines in the image. Four types of lines are distinguished: edges between the skyblue goal and the field, edges between the yellow goal and the

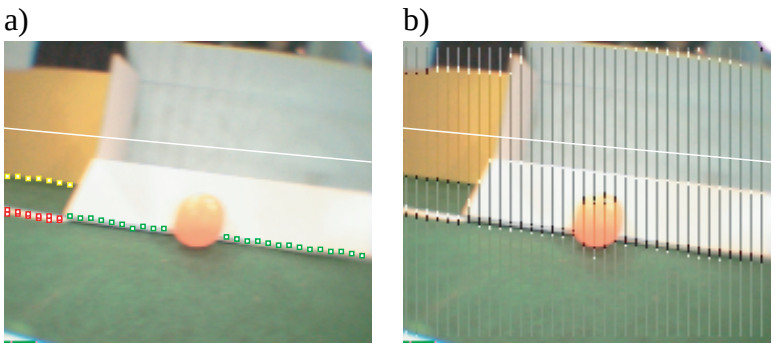


Figure 3.11: Lines perceptor. a) Three types of lines. Yellow: field/goal, green: field/border, red: field line. b) The vertical scan lines are scanned from top to bottom. White pixels: increase in y-channel, black pixels: decrease in y-channel.

field, edges between the border and the field, and edges between the field lines and the field (cf. Fig. 3.11a).

Finding Pixels on Edges. The perceptor scans vertical lines in the image from top to bottom. The lines have a distance of five pixels to one another (cf. Fig. 3.11b). As the green of the field is very dark, all edges are characterized by a big difference of the y-channel of adjacent pixels. An increase in the y-channel followed by a decrease is an indication of an edge.

Classifying Pixels on Edges. If the color above the decrease in the y-channel is skyblue or yellow, the pixel lies on an edge between a goal and the field. The differentiation between a field line and the border is a bit more complicated. In most of the cases the border has a bigger size in the image than a field line. But a far distant border might be smaller than a very close field line. For that reason the pixel where the decrease in the y-channel was found is assumed to lie on the ground. With the known height and rotation of the camera the distance to that point is calculated. The distance leads to expected sizes of the border and the field line in the image. For the classification these sizes are compared to the distance between the increase and the decrease of the y-channel in the image. The projection of the pixels on the field plane is also used to determine their relative position to the robot.

3.3 Self-Localization

The GermanTeam implemented three different methods to solve the problem of self-localization: the *Single Landmark Self-Locator*, the *Monte-Carlo Self-Locator*, and the *Lines Self-Locator*. While the latter is still experimental, the other two approaches were used in actual RoboCup games.

3.3.1 Single Landmark Self-Locator

For the Robocup 2001 in Seattle, the GermanTeam implemented a Monte-Carlo localization method. The approach proved to be of high accuracy in computing the position of the robot on the field. Sadly this accuracy is achieved by using a quite high amount of CPU-time because the position of the robot is modeled as distribution of particles, and numerous operations have to be performed for each particle (cf. Sect. 3.3.2). Besides, the Monte-Carlo self-locator requires the robot to actually look for landmarks from time to time. In a highly dynamic environment it is possible to miss important events like a passing ball while scanning for landmarks. Therefore, we tried to develop a localization method that is faster and that can cope with the landmarks the robot sees in typical game situations without the need of regular scans.

3.3.1.1 Approach

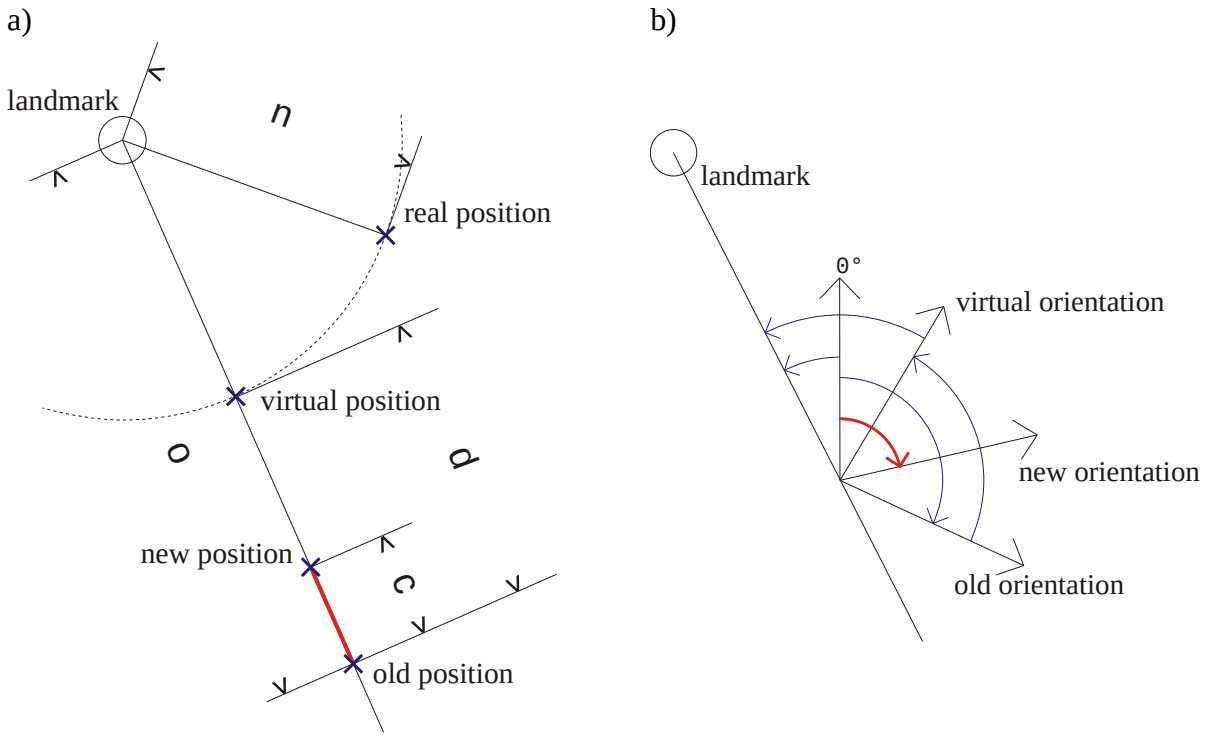
To calculate the exact position of the robot on the field one needs at least two very accurate pairs of angle and distance to different landmarks. If more measurements are available, their accuracy can also be lower. However due to the limitations of the hardware, in most situations it is not possible to obtain such measurements with the necessary accuracy or frequency while maintaining the attention on the game. Normally, a single image from the robot's camera contains only usable measurements of at most two landmarks.

Inspired by the approach of UNSW from 2000, a method for self-localization was developed that uses only the two landmarks recognized best within a single image, and only if they are at least measured with a certain minimum quality. In addition to the approach of UNSW, the quality of the measurements is incorporated in the calculation process.

The single landmarks self-locator uses information on flags, goals, and odometry to determine the location of the robot. The latter comprises the robot's position, its orientation on the field, and the reliability of the localization. The information on flags and goals consists of the distance of the landmark, the reliability of the distance measurement, the bearing of the landmark, and the reliability of that bearing.

Basic Functionality. The self-localization is performed in an iterative process. For each new set of percepts, the previous pair of position and orientation is updated by the odometry data and their reliability values are decreased. If available, the two best landmark percepts and a goal percept are selected. They will be separately used to compute the new position.

In the following, the algorithm will be described by showing how the estimation of the position is improved with a single landmark measurement. At first the new position is estimated and then the orientation of the robot. The newly measured distance to a landmark is used to compute a virtual position. It is set on the straight line from the last estimated position to the landmark at the measured distance from that landmark. According to the relation between the reliability of the last position and the quality of the measurement, i. e. the *distanceCorrectionFactor*, the new position is interpolated between the last position and the virtual position (cf. Fig. 3.12a). The reliability of the new position depends on the quality of the measured distance. The relative orientation to the landmark is computed in a similar fashion (cf. Fig. 3.12b). The orientation



- o distance from old estimated position to landmark
- n measured distance to landmark
- d difference between o and n
- c d multiplied by the *distanceCorrectionFactor*
- α old orientation
- β angle between straight line from landmark to current position and x-axis
- γ measured angle to landmark
- δ difference between old and virtual orientation
- φ new orientation

Figure 3.12: One iteration of the single landmark self-locator. a) Position update. b) Orientation update.

is interpolated between the old orientation and the newly measured orientation relative to the landmark. The interpolation utilizes the quality of the measured angle and the reliability of the last orientation. The reliability of the new orientation depends on the reliability of the measured angle.

Basically the goal is treated as a landmark with its position at the center of the goal line. Since measurements of distances to goals are usually very imprecise, these measurements have a very low quality and therefore play a minor role in the correction of the estimated position. On the other hand the angle to the goal is considered to be of a good quality and of a great importance. The orientation of the robot relative to the goal is among the most important information in RoboCup, and therefore the goal is taken into account after the landmarks. Thus possible false orientations obtained from the landmarks are overruled by the perception of the goal.

At the end of each cycle the computed position, orientation, and reliability values are stored for the next iteration, and the localization data is provided to other modules. This data includes the reliability that is calculated as the product of the reliabilities of orientation and position.

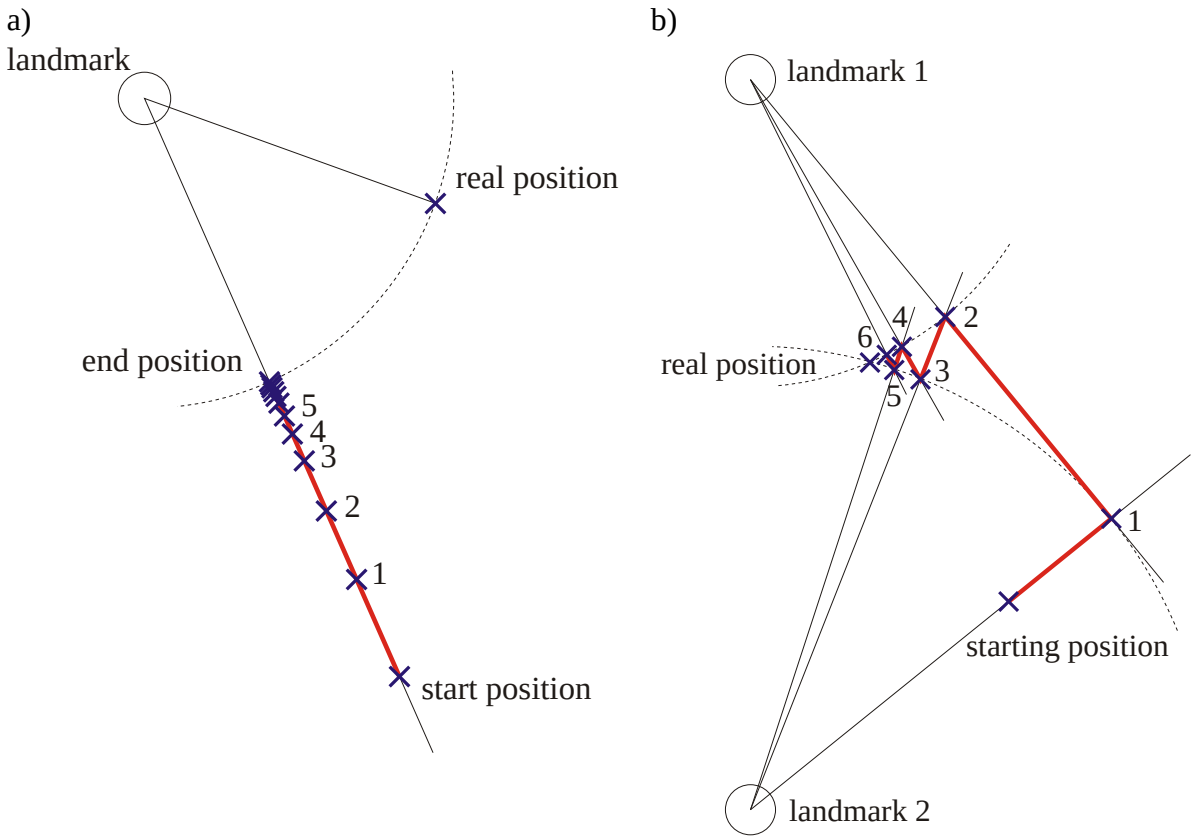


Figure 3.13: a) Problematic situation. The position can only be updated along the straight line to the seen landmark. b) Normal situation. Different landmarks are seen interchangingly and the estimated position settles down near the real position.

Expected Behavior. The algorithm described cannot compute an exact position and even makes quite huge errors when the robot sees only the same landmark over a certain period of time. As a matter of fact in such a situation the position can only be corrected along the straight line from the landmark to the old position (cf. Fig. 3.13a). On the other hand, such situations are rare and if the robot can see different landmarks in a series of images, the position will be corrected along changing straight lines, and it will settle down to an fair estimation of the true position (cf. Fig. 3.13b).

Optimization. To enhance the stability of the localization the parameter *progressiveness* was introduced. It controls the rate by which the estimated position approaches the virtual position. Depending on the value of this parameter the algorithm stalls (zero) or keeps working as before (one). A compromise has to be found between stability and the ability to adapt to changing positions very quickly. With values between 0.10 and 0.15 the localization was more stable and according to the speed of the robot it converged near the true position or was slightly behind if the robot was very fast. The results were further improved by changing the method of decrementing the reliability of the old position at the beginning of each cycle. It now depends on how far the

robot walked according to the odometry data. Thus the reliability of the old position is lower if the robot walked farther and keeps higher if it stands still.

A reliability that is directly determined from the reliabilities of the input data is not really useful to judge the quality of the current estimate of the robot's position. Therefore, the reliability of a new position and a new orientation integrates the difference between the old values and the virtual values for position and orientation. Thus the reliability decreases if there is a big difference between these values. Thus a single set of unfitting data will not significantly diminish the trust in the localization but continuing conflicting measurements lead to a dropping reliability value and thereby a faster re-localization. If the robot detects continuously inconsistent landmarks or no landmarks at all, the reliability value will drop and remain low. Thus, the behavior control can perform the actions required to improve the localization, e. g., to let the head search for landmarks.

3.3.1.2 Results

At the Robocup German Open 2002 in Paderborn the *Darmstadt Dribbling Dackels* used the single landmark self-locator in all games, while the teams from the other universities in the GermanTeam used the Monte-Carlo self-locator (cf. Sect. 3.3.2). The robots of the *Darmstadt Dribbling Dackels* seemed always to be well-localized on the field, and in fact, the team actually won the competition.

In comparison to the Monte-Carlo self-locator that was used in Seattle and Fukuoka, the single landmark self-locator needs less computation time and delivers comparable results under good lighting conditions. However if the color table is bad, the Monte-Carlo approach is better able to model the resulting uncertainties, but that has to be paid by wasting more processor cycles.

A few weeks before the RoboCup in Fukuoka, it had to be decided which localization method has to be used during the contest. Although the GermanTeam is even able to switch alternative implementations at runtime, the modeling of the reliability differed between both approaches. As the behavior control is highly dependent on this modeling, the decision had to be made before the behavior was developed. At that moment, there was no chance to test the situation in Fukuoka. Therefore, the more reliable but slower method was chosen. As the conditions in Fukuoka were perfect, the single landmark self-locator would also have done.

3.3.2 Monte-Carlo Self-Locator

The *Monte-Carlo Self-Locator* implements a Markov-localization method employing the so-called Monte-Carlo approach [9]. It is a probabilistic approach, in which the current location of the robot is modeled as the density of a set of particles (cf. Fig. 3.15a). Each particle can be seen as the hypothesis of the robot being located at this position. Therefore, such particles mainly consist of a robot pose, i. e. a vector representing the robot's x/y-coordinates in millimeters and its rotation θ in radians:

$$pose = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} \tag{3.12}$$

A Markov-localization method requires both an observation model and a motion model. The observation model describes the probability for taking certain measurements at certain locations. The motion model expresses the probability for certain actions to move the robot to certain relative positions.

The localization approach works as follows: first, all particles are moved according to the motion model of the previous action of the robot. Then, the probabilities for all particles are determined on the basis of the observation model for the current sensor readings, i. e. bearings on landmarks calculated from the actual camera image. Based on these probabilities, the so-called *resampling* is performed, i. e. moving more particles to the locations of samples with a high probability. Afterwards, the average of the probability distribution is determined, representing the best estimation of the current robot pose. Finally, the process repeats from the beginning.

3.3.2.1 Motion Model

The motion model determines the odometry offset $\Delta_{odometry}$ since the last localization from the odometry value delivered by the motion module (cf. Sect. 3.7) to represent the effects of the actions on the robot's pose. In addition, a random error Δ_{error} is assumed, according to the following definition:

$$\Delta_{error} = \begin{pmatrix} 0.1d \times \text{random}(-1 \ldots 1) \\ 0.02d \times \text{random}(-1 \ldots 1) \\ (0.002d + 0.2\alpha) \times \text{random}(-1 \ldots 1) \end{pmatrix} \tag{3.13}$$

In equation (3.13), d is the length of the odometry offset, i. e. the distance the robot walked, α is the angle the robot turned.

For each sample, the new pose is determined as

$$pose_{new} = pose_{old} + \Delta_{odometry} + \Delta_{error} \tag{3.14}$$

Note that the operation $+$ involves coordinate transformations based on the rotational components of the poses.

3.3.2.2 Observation Model

The observation model relates real sensor measurements to measurements as they would be taken if the robot were at a certain location. Instead of using the distances and directions to the landmarks in the environment, i. e. the flags and the goals, this localization approach only uses the directions to the vertical edges of the landmarks. The advantage of using the edges for orientation is that one can still use the visible edge of a landmark that is partially hidden by the image border. Therefore, more points of reference can be used per image, which can potentially improve the self-localization.

As the utilized percepts delivered by the landmarks perceptor (cf. Sect. 3.2.1.2) and the flag/goal specialist (cf. Sect. 3.2.2.3 and 3.2.2.4) are bearings on the edges of flags and goals, these have to be related to the assumed bearings from hypothetical positions. To determine the

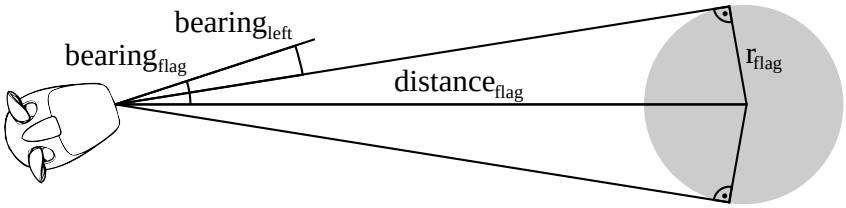


Figure 3.14: Calculating the angle to an edge of a flag.

expected bearings, the camera position has to be determined for each particle first, because the real measurements are not taken from the robot's body position, but from the location of the camera. Note that this is only required for the translational components of the camera pose, because the rotational components were already normalized during earlier processing steps (cf. Sect. 3.2.1.1). From these hypothetical camera locations, the bearings on the edges are calculated. It must be distinguished between the edges of flags and the edges of goals:

Flags. The calculation of the bearing on the center of a flag is straightforward. However, to determine the angle to the left or right edge, the bearing on the center $bearing_{flag}$, the distance between the assumed camera position and the center of the flag $distance_{flag}$, and the radius of the flag r_{flag} are required (cf. Fig. 3.14):

$$bearing_{left/right} = bearing_{flag} \pm \arcsin(r_{flag}/distance_{flag}) \tag{3.15}$$

Goals. The front posts of the goals are used as points of reference. As the goals are colored on the inside, but white on the outside, the left and right edges of a color blob representing a goal even correlate to the posts if the goal is seen from the outside.

Probabilities. The observation model only takes into account the bearings on the edges that are actually seen, i. e., it is ignored if the robot has *not* seen a certain edge that it should have seen according to its hypothetical position and the camera pose. Therefore, the probabilities of the particles are only calculated from the similarities of the measured angles to the expected angles. Each similarity s is determined from the measured angle $\omega_{measured}$ and the expected angle $\omega_{expected}$ for a certain pose by applying a sigmoid function to the difference of both angles:

$$\begin{aligned} s(\omega_{measured}, \omega_{expected}) \quad = \quad & \begin{cases} \mathrm{e}^{-50d^2} & \text{if } d < 1 \\ \mathrm{e}^{-50(2-d)^2} & \text{otherwise} \end{cases} \\ & \text{where } d = \frac{\left|\omega_{measured} - \omega_{expected}\right|}{\pi} \end{aligned} \tag{3.16}$$

The probability p of a certain particle is the product of these similarities:

$$p = \prod_{\omega_{measured}} s(\omega_{measured}, \omega_{expected}) \tag{3.17}$$

3.3.2.3 Resampling

In the resampling step, the samples are moved according to their probabilities. There is a trade-off between quickly reacting to unmodeled movements, e. g., when the referee displaces the robot, and stability against misreadings, resulting either from image processing problems or from the bad synchronization between receiving an image and the corresponding joint angles of the head. Therefore, resampling must be performed carefully. One possibility would be to move only a few samples, but this would require a large number of particles to always have a sufficiently large population of samples at the current position of the robot. The better solution is to limit the change of the probability of each sample to a certain maximum. Thus misreadings will not immediately affect the probability distribution. Instead, several readings are required to lower the probability, resulting in a higher stability of the distribution. However, if the position of the robot was changed externally, the measurements will constantly be inconsistent with the current distribution of the samples, and therefore the probabilities will fall rapidly, and resampling will take place.

The filtered probability p' is calculated as

$$p'_{new} = \begin{cases} p'_{old} + 0.1 & \text{if } p > p'_{old} + 0.1 \\ p'_{old} - 0.05 & \text{if } p < p'_{old} - 0.05 \\ p & \text{otherwise.} \end{cases} \tag{3.18}$$

Resampling is done in three steps:

Importance Resampling. First, the samples are copied from the old distribution to a new distribution. Their frequency in the new distribution depends on the probability p'_i of each sample, so more probable samples are copied more often than less probable ones, and improbable samples are removed.

Inserting Calculated Samples. In a second step, some samples are replaced by calculated samples. This approach follows the idea of Lenser and Veloso [14]: on the RoboCup field, it is often possible to directly determine the position of the robot from sensor measurements, i. e. the percepts. The only problem is that these positions are not always correct, because of misreadings and noise. However, if a calculated position is inserted into the distribution and it is correct, it will get high probabilities during the next observation steps and the distribution will cluster around that position. In contrast, if it is wrong, it will get low probabilities and will be removed very soon. Therefore, calculated positions are only position hypotheses, but they have the potential to speed up the localization of the robot.

Two methods were implemented to calculate possible robot positions. They are used to fill a buffer of *position templates*:

1. The first one uses a short term memory for the bearings on the last three flags seen. Estimated distances to these landmarks and odometry are used to update the bearings on these memorized flags when the robot moves. Bearings on goal posts are not inserted into the buffer, because their distance information is not reliable enough to be used to compensate

for the motion of the robot. However, the calculation of the current position also integrates the goal posts, but only the ones actually seen. So from the buffer and the bearings on goal posts, all combinations of three bearings are used to determine robot positions by triangulation.

2. The second method only employs the current percepts. It uses all combinations of a landmark with a reliable distance information, i. e. a flag, and a bearing on a goal post or a flag to determine the current position. For each combination, one or two possible positions can be calculated.

The samples in the distribution are replaced by positions from the template buffer with a probability of $1 - p'_i$. Each template is only inserted once into the distribution. If more templates are required than have been calculated, random samples are employed.

Probabilistic Search. In a third step that is in fact part of the next motion update, the particles are moved locally according to their probability. The more probable a sample is, the less it is moved. This can be seen as a probabilistic random search for the best position, because the samples that are randomly moved closer to the real position of the robot will be rewarded by better probabilities during the next observation update steps, and they will therefore be more frequent in future distributions.

The samples are moved according to the following equation:

$$pose_{new} = pose_{old} + \begin{pmatrix} 100(1-p') \times \text{random}(-1 \ldots 1) \\ 100(1-p') \times \text{random}(-1 \ldots 1) \\ 0.5(1-p') \times \text{random}(-1 \ldots 1) \end{pmatrix} \tag{3.19}$$

3.3.2.4 Estimating the Pose of the Robot

The pose of the robot is calculated from the sample distribution in two steps: first, the largest cluster is determined, and then the current pose is calculated as the average of all samples belonging to that cluster.

Finding the Largest Cluster. To calculate the largest cluster, all samples are assigned to a grid that discretizes the x-, y-, and θ-space into $10 \times 10 \times 10$ cells. Then, it is searched for the $2 \times 2 \times 2$ sub-cube that contains the maximum number of samples.

Calculating the Average. All m samples belonging to that sub-cube are used to estimate the current pose of the robot. Whereas the mean x- and y-components can directly be determined, averaging the angles is not straightforward, because of their circularity. Instead, the mean angle θ_{robot} is calculated as:

$$\theta_{robot} = \text{atan2}(\sum_i \sin\theta_i, \sum_i \cos\theta_i) \tag{3.20}$$

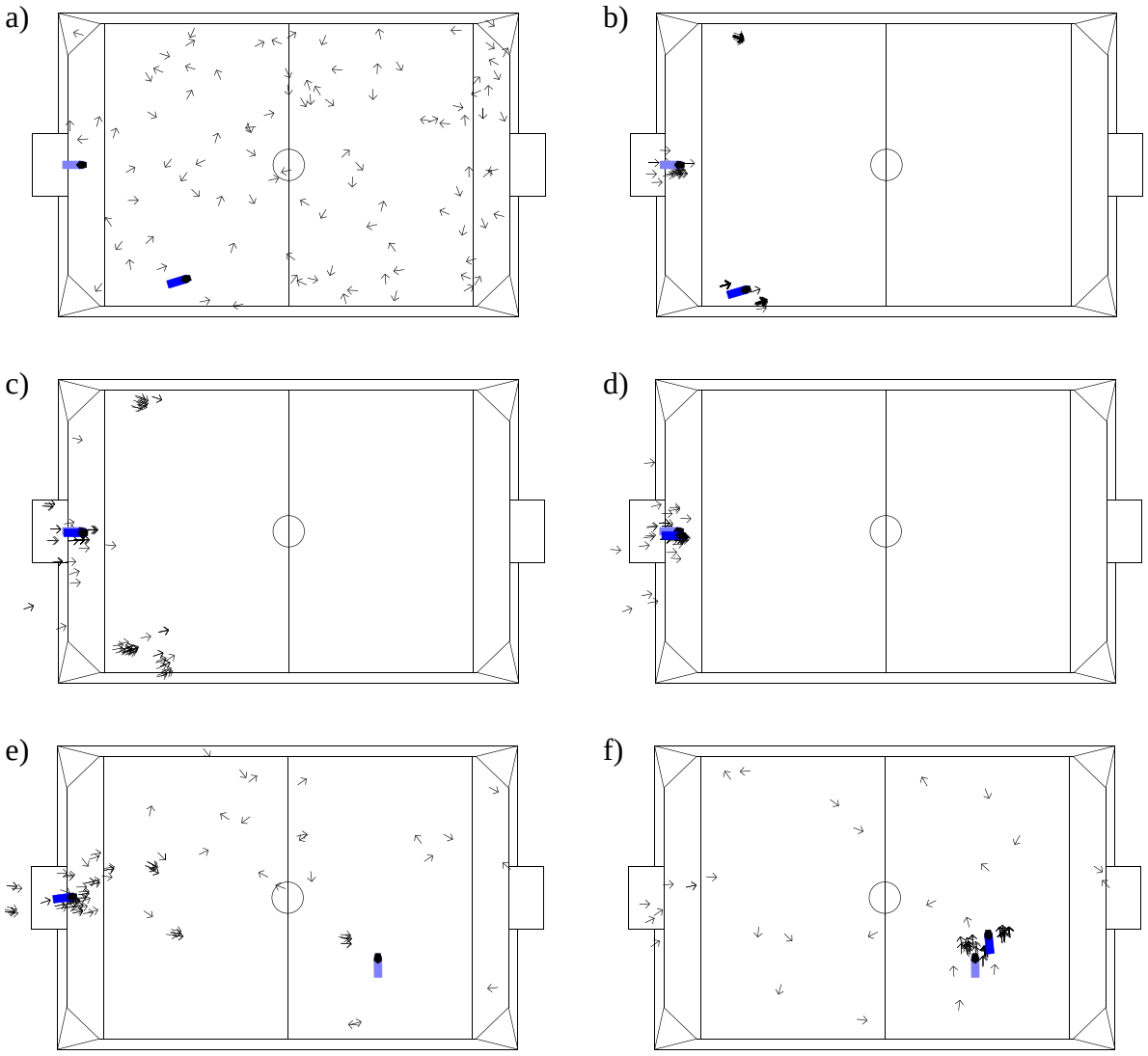


Figure 3.15: Distribution of the samples during the Monte-Carlo localization while turning the head. The bright robot body marks the real position of the robot, the darker body marks the estimated location. a) After the first image processed (40 ms). b) After eight images processed (320 ms). c) After 14 images (560 ms). d) After 40 images (1600 ms). e) Robot manually moved to another position. f) 13 images (520 ms) later.

Certainty. The certainty q of the position estimate is determined by multiplying the ratio between the number of the samples in the winning sub-cube m and the overall number of samples n by the average probability in the winning sub-cube:

$$q = \frac{m}{n} \cdot \frac{1}{m} \sum_i p'_i = \frac{1}{n} \sum_i p'_i \tag{3.21}$$

This value is interpreted by other modules to determine the appropriate behavior, e. g., to look at landmarks to improve the certainty of the position estimate.

3.3.2.5 Results

Figure 3.15 depicts some examples for the performance of the approach using 100 samples. The experiments shown were conducted with SimGT2002 (cf. Sect. 5.1) using the *simulation time mode*, i. e. each image taken by the simulated camera is processed. The results show how fast the approach is able to localize and re-localize the robot. In Fukuoka, the method also proved to work on real robots. The GermanTeam was the only team that supported all features of the RoboCup Game Manager that allows the referee to give instructions to the robots. This includes automatic positioning on the field, e. g. for kickoff. For instance, the robots of the GermanTeam were just started somewhere on the field, and then—while still many people were working on the field—they autonomously walked to their initial positions. In addition, the self-localization worked very well on fields without an outer barrier, e. g. on the practice field.

3.3.3 Lines Self-Locator

The previous two approaches use the colored beacons and the goals for self-localization. However, there are no beacons on a real soccer field, and as it is the goal of the RoboCup initiative to compete with the human world champion in 2050, it seems to be a natural thing to develop techniques for self-localization that do not depend on artificial clues. Therefore, the GermanTeam works on a method to use the field lines to determine the robot's location on the field. This approach is not finished yet, and it was not used during the RoboCup in Fukuoka, but it may be used next year in Padova.

The lines self-locator also follows the Monte-Carlo approach, in fact, most of its code is the same[1]. The main difference between the Monte-Carlo self-locator and the lines self-locator is the observation model. In addition, the lines self-locator is not able to directly calculate positions from perceptions, i. e. it cannot perform a sensor resetting localization [14]. However, this drawback is partially compensated by the fact that field lines are seen more often than beacons.

3.3.3.1 Observation Model

The localization is based on the lines percept as delivered by the *lines perceptor* (cf. Sect. 3.2.2.6). The lines percept consists of relative offsets from the body center to points on the field. Each point is assumed to lie on a line that is belonging to one of four possible classes:

- An edge of a field line,
- an edge between the field and the border,
- an edge between the field and the yellow goal,
- or an edge between the field and the blue goal.

[1] If the lines self-locator leaves the experimental state, there may be a common base class for both beacon-based and lines-based self-localization.

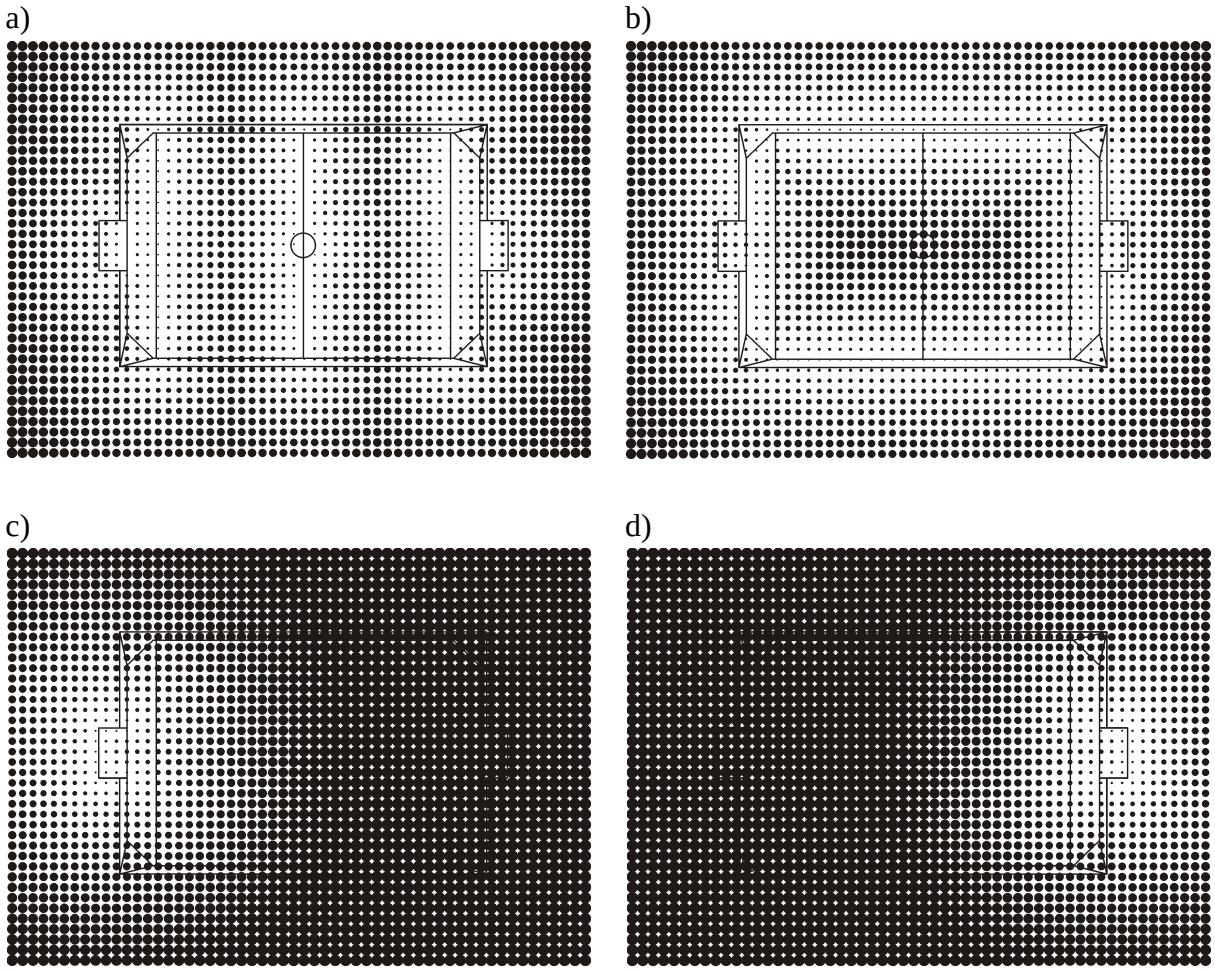


Figure 3.16: Distances from lines. Distance is visualized as thickness of dots. a) Field lines. b) Border. c) One goal. d) The other goal.

Note that the calculation of the offsets to these points is prone to errors because the pose of the camera cannot be determined precisely. In fact, the farther away a point is, the bigger the errors will be.

The points from the lines percept are used to determine the probability of each sample in the Monte-Carlo distribution. As the positions of the samples on the field are known, for each sample it can be determined, where the measured points would be located on the field if the position of the sample would be correct. For each of these points in field coordinates, it can be calculated, how far the closest point on a real field line of the corresponding type (field, border, yellow goal, or blue goal) is away. The smaller the deviation between a real line and the projection of a measured point from a hypothetic position is, the more probable the robot is really located at that position. However, the probability also depends on the distance of the measured point from the robot, because farther points will contain larger measurement errors, i. e. deviations of farther away points should have a smaller impact on the probability than deviations of closer ones.

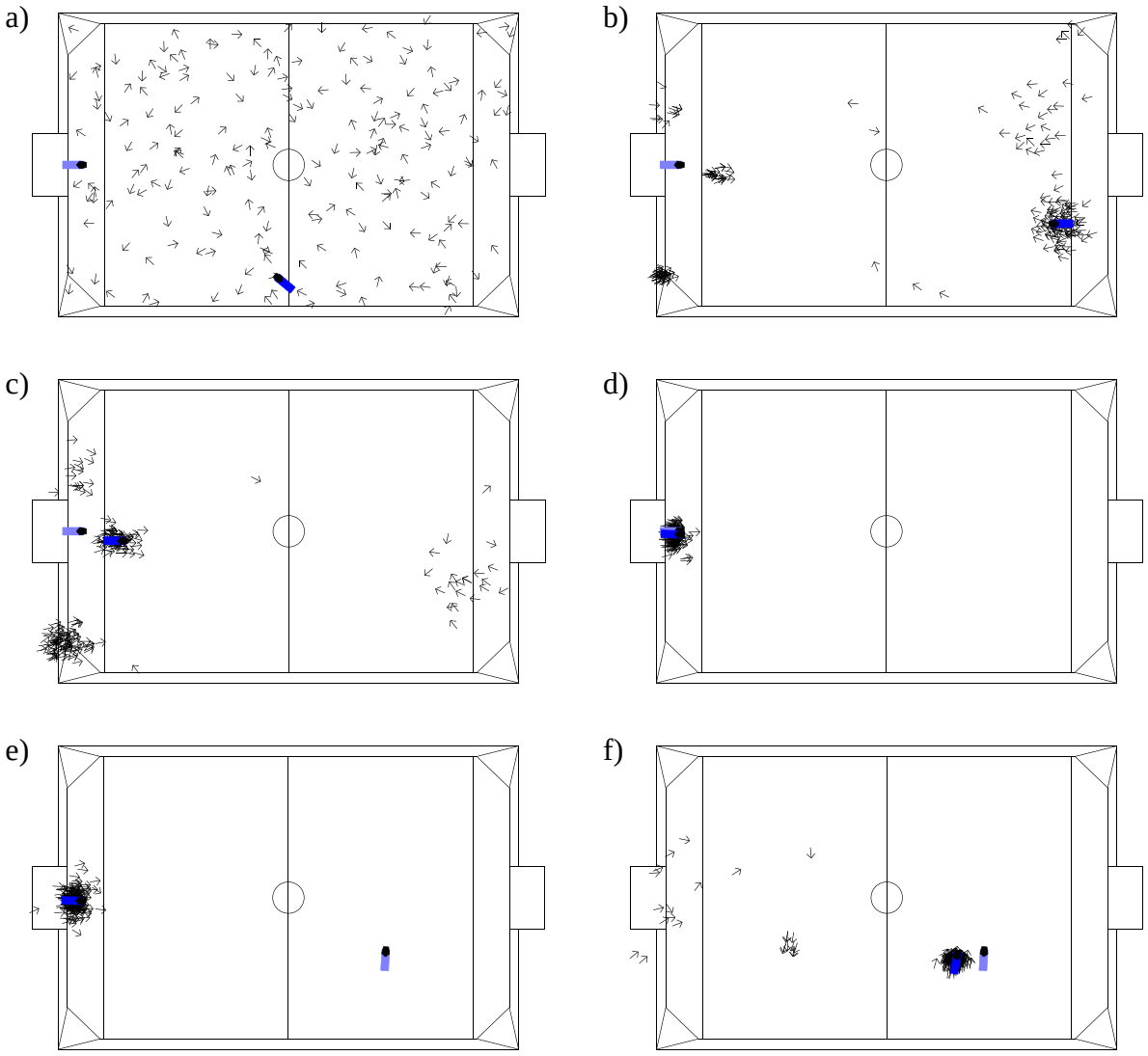


Figure 3.17: Distribution of the samples during the Monte-Carlo localization while turning the head. The bright robot body marks the real position of the robot, the darker body marks the estimated location. a) After the first image processed (40 ms). b) After 16 images processed (640 ms). c) After 30 images (1200 ms). d) After 100 images (4000 ms). e) Robot manually moved to another position. f) 40 images (1600 ms) later.

3.3.3.2 Optimizations

Calculating the probability for all points in a lines percept for all samples in the Monte-Carlo distribution would be a costly operation. Therefore the number of points is fixed, and these points are selected from the lines percept according to the following criteria:

- A fixed number of ten points is chosen by random.

- It is tried to select the same number of points from each line class, because points belonging to border lines and field lines are more frequent, but the points belonging to the goals determine the orientation on the field[2].

- Closer points are chosen with a higher probability than farther away points because their measurements are more reliable.

Calculating the smallest distances of ten points to the field lines is still an expensive operation if it has to be performed for, e. g., 200 samples. Therefore, the distances are precalculated for each line type and stored in two-dimensional lookup tables with a resolution of 2.5 cm (cf. Fig. 3.16). That way, the distance of a point to the closest line of the corresponding type can be determined by a simple table lookup. Therefore, the method is not slower than the beacon-based Monte-Carlo localization.

3.3.3.3 Early Results

Experiments performed in the simulator are quite promising. Figure 3.17 shows an experiment conducted with 200 samples. The method is always able to find the position of the robot, but it takes longer that the beacon-based localization. In addition, the mirror symmetry of the field is a problem. In figure 3.17b, the method first selects the wrong side of the field. However, when a goal comes into view, the position flips over to the correct side (cf.. Fig. 3.17c), and it remains stable. Figures 3.17e and 3.17f demonstrate that the approach is also capable of re-localization after the robot was moved manually.

3.4 Ball Modeling

The ball is the most important object in a soccer game. Therefore it is crucial to keep track of its position on the field. The ball perceptor (cf. Sect. 3.2.1.3) and the ball specialist (cf. Sect. 3.2.2.2) only detect the ball if it is actually seen by the camera of the robot. As the opening angles of the camera are quite small, the ball is missing in many images. Although the head control (cf. Sect. 3.7.3) tries to track the ball in many situations, the head has sometimes to be turned to look at landmarks. It is the task of *ball modeling* to keep track of the location of the ball even if the ball is currently not seen.

In 2002, the GermanTeam still followed a quite simple approach. It is differentiated between two cases, namely whether the ball is currently seen or not:

3.4.1 Ball Is Visible

If the ball is seen, i. e. the percept collection contains a ball percept, the position of the ball is directly determined from the offset stored in the percept and the actual position of the robot.

[2] Please note that the field is mirror symmetric without the goals.

3.4.2 Ball Is Not Visible

In that case, the position is determined, where the ball has been seen last. This is a little bit more complex than one would expect, because it is tried to maintain the relative relationship between the robot and the ball, i. e. if the self-localization of the robot jumps, the ball position will follow. This is especially important if the ball is very close and the robot prepares to perform a kick. However, the locomotion of the robot changes the relative ball position. Therefore, odometry is used to update the relative position of the ball. In addition, the locomotion of the robot can also move the ball if the robot touches the ball. This can be a problem, because if the calculated ball position is inside the body of the robot, the head cannot be rotated to look at the ball. Therefore, the ball position is moved with the robot if the ball enters the "bounding box" of the robot.

Then, it is distinguished between three different cases:

The Camera Is Looking at the Ball, i. e. at its last known position. In this case, the ball seems to have disappeared and its position is unknown. Whether the ball should be contained in the camera image is determined by calculating the vertical and horizontal angles under which the last known ball position is seen. If both angles are smaller than half of the opening angle of the camera, the ball should be seen. However, the approach does not take the fact into account that the ball can still be at its position but is hidden by another robot.

The Camera Is Not Looking at the Ball. It that case, it is assumed that the ball is still at its place. As the head control tries to track the ball in many situations, it will look at the position of the ball very soon. Thus it will still be determined very quickly if the ball has moved from its previous position.

The Ball Position Is Unknown for a Longer Period of Time. If the robot was not able to find the ball for more than seven seconds, it will accept ball positions communicated from other robots. The use of communicated positions is delayed because they are less precise due to the uncertainties in the self-localization of the robot and its communication partners.

3.4.3 Ball Speed

In order to calculate the speed of the ball, previous ball positions and their time stamps are stored in a buffer. The buffer is emptied whenever the ball position is unknown. However, if the buffer is completely filled, the speed of the ball is calculated from the oldest and the current position of the ball. Currently, the buffer has a size of four entries. It turned out that the results are still very noisy, e. g., for a ball that is not moving at all, a speed of up to 5 cm/s is determined.

3.5 Player Modeling

The knowledge of other robots positions is important for avoiding collisions and for tactical planning. The locator for other players performs the calculation of these positions based on players

percepts. In addition, positions of teammates received via the wireless network communication are integrated.

3.5.1 Determining Robot Positions from Distributions

The positions of percepts of other robots are relative to the position of the observing robot. In a first step, they are converted to absolute positions on the field. In a second step, it is tested, whether the absolute positions of the percepts are outside the field. In this case, they are projected to the border along an imaginary line which connects the robot with the absolute position of the player percept. The resulting positions of the percepts are stored in a list for about two seconds.

The soccer field is discretized as a grid. The positions of the percepts are converted into grid points, and distributions in x and y directions are created. Then, the maxima in these distributions are determined. A maximum results from a high density of perceived robots at a certain location in the grid. The maxima are sorted by their distinctiveness in descending order. If a maximum is above a certain threshold, a robot is assumed to be located at the corresponding point. The point in the grid is converted to an absolute position on the soccer field. Finally, this position is added to the *PlayersCollection* that contains the positions of all players recognized.

The process described above is done separately for the opponents and for the teammates.

3.5.2 Integration of Team Messages

This year it was possible to communicate between robots of the own team via a wireless network. So the positions of teammates could be transmitted. In a first approach these positions are also used for the localization of other robots. It is assumed that a position sent by a teammate is often more precise than a position calculated from the percept showing that teammate. Therefore, positions communicated by teammates are used by the players locator.

The positions resulting from percepts are replaced by the transmitted ones. If the robot has not received the positions from all teammates, or if the last position received is too old, the positions calculated from percepts are kept. To avoid representing a teammate twice, a position calculated from percepts must have a minimum distance to all positions received from teammates.

3.6 Behavior Control

The module *BehaviorControl* is responsible for decision making based on the world state, the game control data received from the *RoboCup Game Manager*, the motion request that is currently being executed by the motion modules, and the team messages from other robots. It has no access to lower layers of information processing.

It outputs the following:

- A *motion request* that specifies the next motion of the robot,
- a *head motion request* that specifies the mode how the robot's head is moved,

- an *LED request* that sets the states of the LEDs,
- an *acoustic message* that selects a sound file to be played by the robot's loudspeaker,
- a *team message* that is sent to other players by wireless communication.

For the German Open 2002, each of the four universities of the GermanTeam used a different approach for behavior control, only two of which will be discussed in detail:

The Bremen Byters from the Universität Bremen followed a potential field approach, which was also used for the ball challenge in Fukuoka.

The Ruhrpott Hellhounds from the University of Dortmund developed a Fuzzy Logic based approach for behavior control.

The Darmstadt Dribbling Dackels from the Technische Universität Darmstadt won the German Open 2002 with a state machine approach that was formalized in XML. This solution is described in the next section (cf. Sect. 3.6.1).

Humboldt 2002 from the Humboldt-Universität zu Berlin continued the approach that the GermanTeam used for the RoboCup 2001 in Seattle. In addition, the architecture was formalized in the XML language XABSL. Although *Humboldt 2002* only reached the second place at the German Open 2002, this solution was adopted and used in Fukuoka because it was thought to be the most powerful approach of the four (cf. Sect. 3.6.2)

3.6.1 TUDXMLBehavior

The *TUDXMLBehavior* was developed with the aim to shorten the process of programming a state machine in C++. Modeling behavior in C++ involves two very time consuming problems. On the one hand, for each state described, very much redundant and technical code is necessary. Spontaneous tests require either to write large blocks of code or to copy and paste them from another already existing state as a template. On the other hand, the code can get so complex and confusing, which impedes the development process. During designing a behavior control, it it often necessary to change the original approach. In plain C or C++ it is hard to see the whole structure of a state machine. Therefore, before a modification can be integrated, a lot of effort is required to see all the dependencies between the different states.

Visualizing the state machine can give the behavior designer better access to the information needed. XML/XSL was chosen as the tool to solve this problems. XSL provides the ability to generate C code without programming in C. It transforms XML to C with the help of a stylesheet.

3.6.1.1 Structure of a Behavior Model Document

The state machine is modeled in XML. So every state, the transitions between the states, and some additional information for the generation of the C++ files has to be represented. Each XML description of a behavior has the same structure (cf. Fig. 3.18):

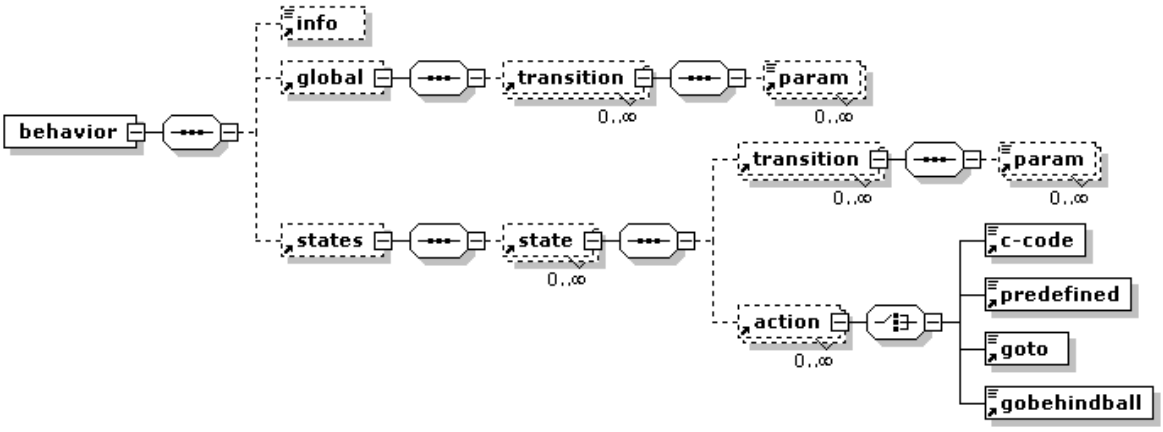


Figure 3.18: Structure of the XML behavior description

```
<?xml version="1.0"?>
<!DOCTYPE behavior SYSTEM "model.dtd">

<behavior name="TUDXMLBehavior">
    <info>(short description)</info>
    <global>
        (...)
    </global>
    <states>
        (...)
    </states>
</behavior>
```

An XML behavior description starts with a header with XML relevant information such as the name of the document type definition. This is followed by the tag “behavior”. The parameter “name” specifies the name for the C++ class to be generated and shall be set to “TUDXMLBehavior” in order to work with the *BehaviorSelector* without any change. With the tag “info”, it is possible to describe the task of a behavior. The final two sections actually represent the state machine. The first one describes the global transitions, whereas the second one defines the states.

In the section “global”, it is possible to define transitions that are valid for all states. This means that global transitions are tested before the local transitions of a state, e. g.

```
<global>
    <transition target="getup">
        <param name="robState" value="crashed"/>
    </transition>
</global>
```

In this example, it is always checked whether the robot felt down. If this is the case, the transition is followed to the state “getup”. Global transitions help to minimize the complexity of the state machine.

Each state consists of transitions and actions. A transition defines the successor of the actual state. An action specifies what should happen if the state is selected, e. g., the state “getup” is defined as

```
<state name="getup" head="lookStraightAhead">
    <transition target="pos1">
        <param name="robState" value="standing"/>
    </transition>
    <action>
        <predefined name="getup"/>
    </action>
</state>
```

It changes to state ”pos1” if the variable “robState” indicates that the robot is “standing”. If this expression is true, the state will terminate and the next time “pos1” will be active. But if the expression evaluates to false, the given action will be performed. In this case the robot will perform a “getup” motion. Multiple “param” terms are linked with a logical *and* operation, i. e. the state will only be changed if all “param” terms are true. If there exists more then one transition, they will be tested in their order of appearance. Together, all transitions are forming a logical *or* operation.

3.6.1.2 Language

This section describes the tags that can be used to describe the behavior of the robot.

action. In an “action” section the action is specified that will be performed. Four different types of actions are available. These are “goto”, “gobehindball”, “c-code”, and “predefined”. “goto” and “gobehindball” are helpers to keep the tags shorter. “predefined” means that an action is already defined in the XLS-Stylesheet, and that it is referred to by its name. “c-code” is a more direct method. In a “c-code” part, it is possible to write C code into the XML-document. This feature has been implemented to allow faster testing and shall only be used for the development of new predefined actions.

```
<action>
    <predefined name="getup"/>
</action>
```

gobehindball is a basic operation to let the robot walk behind the ball heading towards the given point in the given distance.

```
<gobehindball destX="xPosOpponentGroundline" destY="0"
              dist="15.0"/>
```

goto is a basic operation to let the robot walk to a given point oriented towards an absolute direction.

```
<goto x="xPosOwnGroundline" y="yPosCenterGoal"
      dir="absBallDir"/>
```

info offers a way to add a doxygen compatible documentation.

```
<info>(short description)</info>
```

predefined. A "predefined" operation is an operation defined in the XSL stylesheet. The aim of this tag is to have access to a library of abilities only referenced by their names.

```
<predefined name="getup"/>
```

state. A "state" of the state machine consists of a "name" and some optional parameters followed by the transitions and the action to be performed.

```
<state name="getup" head="lookStraightAhead">
    <transition (...) >
        (...)
    </transition>
    <action>
        (...)
    </action>
</state>
```

The optional parameters are:

gamestate. When entering this state, the "gamestate" variable is set to the given value. This is useful, e. g., to remember long term plans over multiple transitions.

head sets a new head control mode.

led sets a new LED request.

clusters. Multiple states can be grouped to one or more clusters. In a transition it is possible to determine how long the current cluster is active. This feature was added during the German Open 2002 to leave cycles after a maximum time.

transition. A transition is the change from one state to another. A transition is an tuple of a condition and a target. If one of the conditions is true the transition is valid and will be used.

```
<transition target="getup">
    <param name="robState" value="crashed"/>
</transition>
```

3.6.1.3 Documentation

Similarly to the generation of C code, the documentation is created.

HTML. The HTML documentation is still under development. At the moment it is only a brief overview over the states generated.

Graph. The graph generated shows the states and the transitions between them (cf. Fig. 3.19).

3.6.1.4 Conclusion

During the German Open 2002 the approach enabled the *Darmstadt Dribbling Dackels* to develop and redesign their behavior in a very fast and comfortable way. The visualization of the graph helped saving a lot of time finding cycles and interpreting the actions of the robots.

3.6.2 XABSL

The *Extensible Agent Behavior Specification Language* (XABSL) is an XML dialect used to specify behaviors of autonomous robots. Although it was developed for the participation in the RoboCup Sony Legged Robot League, it is also intended to be used on other platforms (e. g. the simulation league). In this section the behavior architecture used is introduced; the language elements, a code library called *XabslEngine* and tools are described. A simple goalie behavior as used by *Humboldt 2002* for the German Open 2002 is brought in as an example.

3.6.2.1 The Behavior Architecture

XABSL implements a behavior architecture that developed from the architecture the GermanTeam used for the RoboCup Championship 2001 in Seattle [3]. In addition, thoughts by Burkhard [4] influenced the approach.

The general idea is that there is a hierarchy of behavior modules called “options” that are organized in an option tree with basic behaviors called “skills” as the leaves (cf. Fig. 3.20a). Each option can execute a specific behavior independently from its context in the option tree, using the options and skills that are below it in the tree, e. g., the option *“return-to-own-goal”* is a behavior that lets the robot walk backward to the own goal. The option executes that behavior

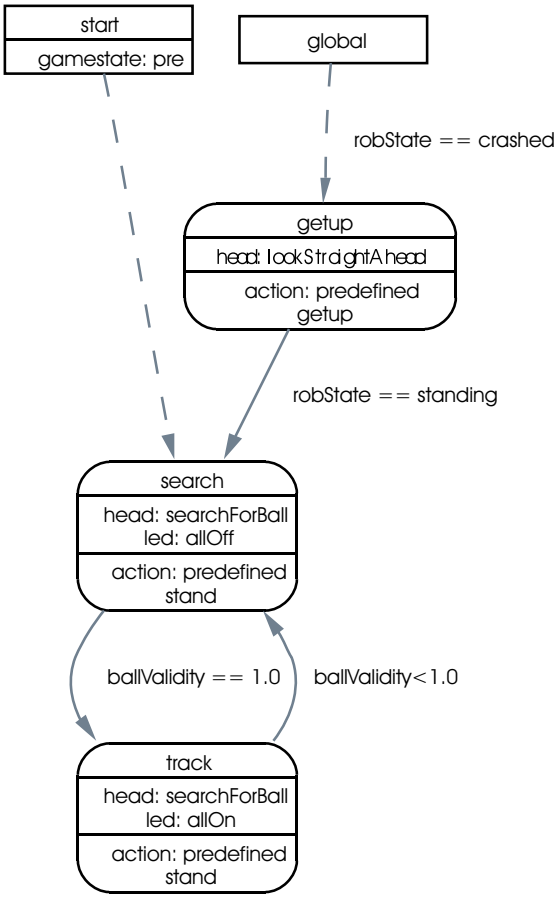


Figure 3.19: Visualization example. The behavior implements searching for the ball and following it with the head.

by using only one skill *"go-to-point"*. The behavior of the option is the same regardless of the context (e. g. whether it is called from *"goalie-ready"* or from *"goalie-play"*). This context-independence allows the creation of more complex behaviors by combining modular options.

The option *"goalie-play"* makes use of that approach. This option represents the complete behavior of a goalie during the entire game. The robot stays inside the own goal if the ball is not seen or if it is far away, using the option *"stay-in-goal"*; it goes for the ball if the ball is nearby using the skill *"go-to-ball"*, kicks the ball away if the robot is at the ball using the skill *"kick"* and returns to the own goal after a successful kick using the option *"return-to-own-goal"*.

This separation of the complete behavior of the agent into different levels in the option tree also allows a separation of different long-term tasks. Behaviors on the lower levels are more short-term and reactive. They are very specialized and usually execute only a very specific task. Changes between these behaviors are not expensive. In contrast higher level behaviors are long-term oriented. They are complex and the costs of switching between these behaviors can be very high, e. g., the decision to wait for a pass is a long-term one, i. e., the robot needs a while for

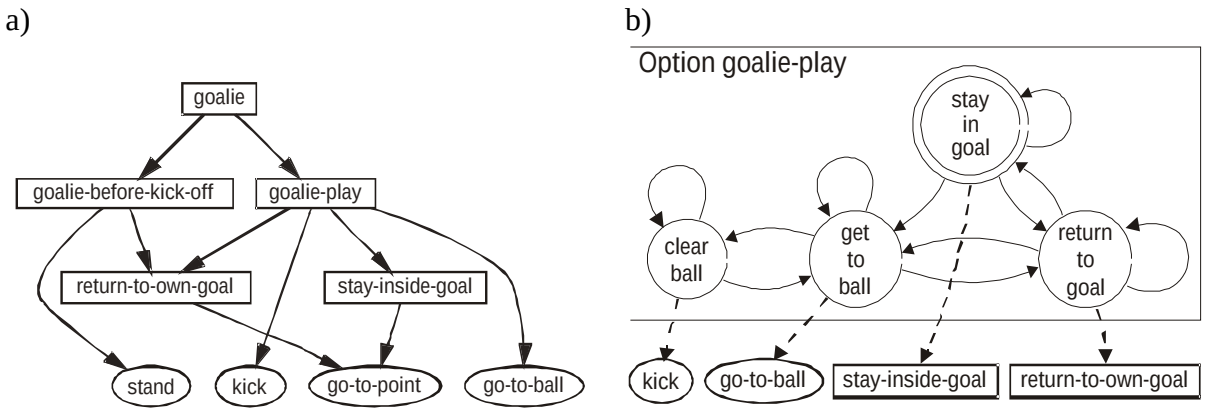


Figure 3.20: a) The option tree of a simplified goalie. Boxes denote “options”, ellipses denote “skills”. b) The internal state machine of the option “goalie-play”. The double circle denotes the initial state.

positioning and should then wait for a pass even if the ball is not seen for a short period of time. Once such a decision is made it should be stuck to in order to avoid having the robot oscillating between different behaviors.

The execution of the option tree always starts from the root option. From there, each option activated determines which suboption will be executed until an option refers to a skill.

The decision which option or skill will be executed next is made using state machines. Each option contains such an internal state machine (cf. Fig. 3.20b). Each state of that state machine stands for a skill or a suboption. Each state has a decision tree that selects between transitions to states (cf. Fig. 3.21).

If an option is executed, the state machine is carried out to determine which state is activated next (which can also be the same option as before). Then, for the active state, the referred suboption or skill is executed and so on. If an option was not activated during the last execution of the option tree, the active state is the initial state.

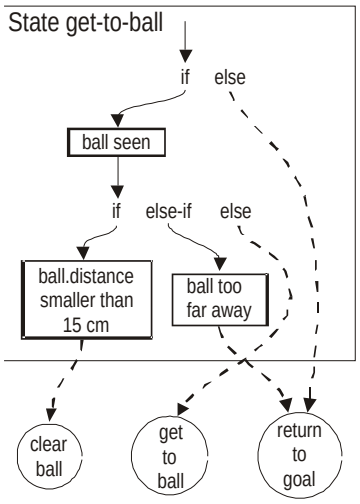
The use of state machines for selecting between suboptions involves the general advantages of state machines. Decisions are not only reactively based on the environment but also on the active state. The decision for a transition to a certain state can vary for different states. It is possible to apply hysteresis between states, e. g. the goalie goes for the ball, when the distance to the ball is smaller than 50 cm. It returns to the goal, when the ball is more than 70 cm away.

To sum up, the architecture introduced has to main features: The combination of modular behaviors (options and skills) in a hierarchical way and the use of state machines for decision making.

3.6.2.2 Formalization of Behavior - XABSL

In the code of the GermanTeam 2001, it turned out that implementing such an architecture in C++ is not very comfortable and error prone. The size of the source files of the complete implementation of the behavior exceeded 500 KB and it was complicated to add new options. Therefore, the *extensible agent behavior specification language* (XABSL) was developed, an XML dialect

a)



b)

```
if (ball.time-since-last-seen < 2000)                //ball seen
{
  if  (ball.distance < 150)  //ball.distance smaller than 15 cm
  {
    transition-to-state (clear-ball);
  }
  else if  (ball.distance > 900)            //ball too far away
  {
    transition-to-state (return-to-goal);
  }
  else
  {
    transition-to-own-state (get-to-ball);
  }
}
else
{
 transition-to-state (return-to-goal);
}
```

Figure 3.21: The decision tree for the state “get-to-ball”. a) The leaves of the tree are transitions to other states. b) Pseudo code of the tree.

specified with XML Schema. The reasons to use XML technologies instead of defining a grammar from scratch were the big variety and quality of existing editing tools, the possibility of easy transformation from and to other languages as well as the general flexibility of data represented in XML languages.

Behaviors using the architecture described in the previous chapter can be completely described in XABSL. There exist language elements for options, their states, and their decision trees. Boolean logic (||, &&, *!*, ==, *!*=, <, <=, >, >=) and simple arithmetic operators (+, -, *, / and %) can be used for conditional expressions.

To be independent of specific software environments and platforms, the formalization of the interaction with the software environment is also included in XABSL by defining symbols. Interaction means access to input functions and variables (e. g. the world state) and to output functions (e. g. to set requests for other parts of the information processing).

The appendix F introduces the language elements and their function in detail.

From a XABSL instance document, an intermediate code is generated. (See section F.13). This was done because on many robotic platforms it would be very hard to deploy an XML parser. In addition, debug symbols, which can be used by debugging environments for visualization and control, e. g. by the *Xabsl Behavior Tester* in the RobotControl application (cf. Sect. J.5.1), and a detailed documentation can be generated from the XML documents.

3.6.2.3 The XabslEngine

For running the formalized behavior on a target platform, the code library *XabslEngine* was developed (described in detail in appendix G). The engine was intended to be platform and application independent and can easily be adapted to other robotic platforms.

The *XabslEngine* parses the intermediate code and links the formalized symbols that were used in the options and states with the variables and functions of the software environment.

In the GT2002 project, the module *XabslBehaviorControl* uses a *XabslEngine* to run XML-formalized behaviors. This is described in detail in appendix H.

3.7 Motion

The module *MotionControl* generates the joint positions sent to the motors and therefore is responsible for controlling the movements of the robot.

It receives a motion request from *BehaviorControl* which is of one of four types (*walk*, *stand*, *perform special action* or *getup*). In addition, if walking is requested it contains a vector describing the speed, the direction, and the type of the walk as there are several different types of walking, such as dribbling the ball, the behavior can choose from. In case of a special action request it contains an identifier defining the requested action.

Furthermore *MotionControl* receives head joint values from the module *HeadControl* which is described below (cf. Sect. 3.7.3). These values are inherited by *MotionControl* but may be overridden if the current motion also requires controlling the head, e. g., for a kick with the head or dribbling the ball while holding it with the head.

Finally *MotionControl* gets current sensor data, because for some motions, sensor input is required, e. g., standing up uses acceleration sensors to detect how to stand up.

From these inputs the module produces a buffer containing joint positions and odometry data, i. e., a vector describing locomotion speed and direction, which, e. g., serves as input for self-localization.

In respect to the system's modular approach, *MotionControl* uses different modules for each of its tasks as well. There is a walking engine module for each possible walking type. Therefore each walking type can be performed by completely different walking engines as well as instances of the same engine with different sets of parameters. How the walking engine works is described below (cf. Sect. 3.7.1). The module executing special actions is described below as well (cf. Sect. 3.7.2). A getup engine module brings the robot to a standing position from everywhere as fast as possible. For standing, the walking engine for the normal walk type is executed with a speed set to zero. Thus changing from standing to walking is possible immediately as the stand position is automatically adjusted to the current walking style.

When the currently used motion module does not reflect the requested motion, the module is changed after it signals that the current motion is finished. Therefore the modules are responsible for correct transitions to other motion types, e. g., a walking engine will signal that a change to a different motion type is only possible after the current step is finished, i. e., all feet are on the ground.

3.7.1 Walking

A walking engine is a module generating joint angles in order to let the robot walk with the speed and the direction requested from behavior control. The implementation described here is

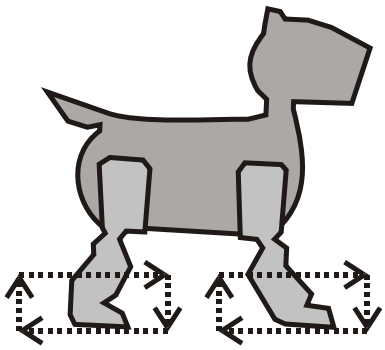


Figure 3.22: Walking by moving feet in rectangles

called *InvKinWalkingEngine* and is a new development from 2002. Although it is mainly a re-implementation applying the same general approach as last year's version, it has some major improvements. A main feature is that the engine and the parameters it uses are separated. The engine offers a huge set of parameters. This allows creating completely different walks with the same engine by having different parameter values. A class containing the set of parameter values is given to the constructor of the engine. Therefore it is possible to have different instances with different parameter sets. It is even possible to transmit new parameters via the wireless network from RobotControl to test them at runtime (cf. Sect. J.7.1).

3.7.1.1 Approach

The general idea is to calculate the position of the feet relative to the body while they move in rectangles around the center position (cf. Fig. 3.22). The joint angles needed to reach the foot position are then calculated with inverse kinematics.

For the direction of walking the four legs are more or less treated as wheels. Seen from above the rectangles are rotated to the desired walking direction (cf. Fig. 3.23a, b). Turning is done by moving each leg in a different diagonal direction (cf. Fig. 3.23c). Walking and turning can also be combined resulting in curved walking. This is done by simple vector addition for each leg (cf. Fig. 3.23d).

The walking speed is defined by the size of the rectangles. The time for one step is constant but when walking faster the step length is greater.

The position and size of these rectangles is defined by the current parameter set. One thing that is fixed for all parameter sets is the walking gait. The engine uses a trot gait, i. e., two diagonally opposite legs perform the same movement, while the other two legs move with a half gait phase offset.

3.7.1.2 Parameters

As mentioned before the actual walking style the engine generates is mainly defined by the set of parameters applied. The parameters are the following:

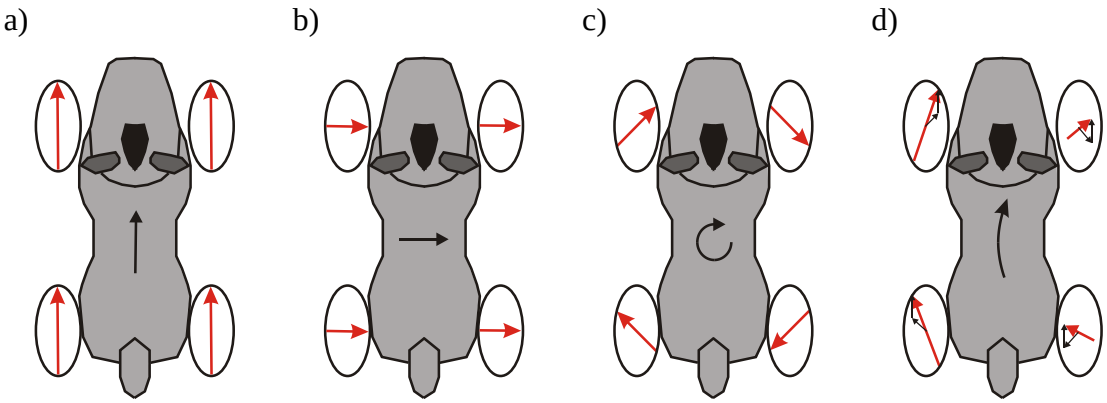


Figure 3.23: Principle of treating legs as wheels. Walking a) forwards, b) sideways. c) Turning. d) Turning while walking forward.

footMode. This parameter selects how the feet will be moved while in the air. Besides the rectangular shape mentioned above it is also possible to have the feet move in a half circle like it was the case in our last year‘s walking engine. This parameter was not used much since the rectangles seemed to provide a better general performance. It was mainly included for a greater flexibility.

foreHeight, foreWidth, foreCenterX. These values describe the center foot position of the forelegs relative to the body of the robot.

hindHeight, hindWidth, hindCenterX. The same values for the hind legs describing center foot positions.

foreFootTilt, hindFootTilt. The foot rectangles are rotated by these angles to compensate for different fore and hind walking heights.

foreFootLift, hindFootLift define the feet lifting, i. e. the height of the rectangles.

legSpeedFactorX, legSpeedFactorY, legSpeedFactorR. These values are factors between the speed of the fore and the hind legs. With these parameters it is possible to have different speeds for the fore and the hind legs. This can be useful when, e. g., the forelegs are limited to very small steps due to their position but the hind legs may still do greater steps.

maxStepSizeX, maxStepSizeY. These are the maximum step sizes that are applied when walking with full speed. They are the radii of the ellipses shown in figures 3.24 and 3.23. The rectangles are clipped to these ellipses.

maxSpeedXChange, maxSpeedYChange, maxRotationChange. By these values the acceleration of the robot is limited. A rapid change of the walking request from behavior control is applied gradually according to these limits. This prevents stumbling or even falling over when the request is changed.

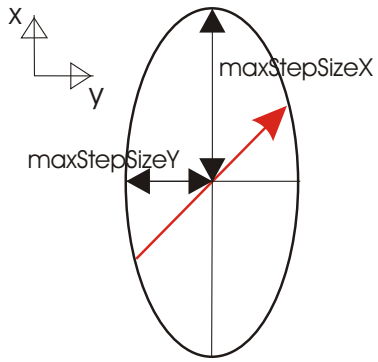


Figure 3.24: Ellipse describing possible foot target positions seen from above

counterRotation. When walking sideways the robot sometimes tends to walk a circle instead due to different contact situations or different step lengths of fore and hind legs. With this value a rotation is generated while walking sideways that compensates this unwanted effect.

stepLen. This is the time for one complete step cycle (cf. Fig. 3.25).

groundPhase, liftPhase. These values define the timing of the step cycle (cf. Fig. 3.25). *groundPhase* defines how much time of the step cycle the foot will be on the ground. *liftPhase* defines how fast the foot will be lifted or lowered.

headTilt, headPan, headRoll, mouth. If these values are given they are used as angles for the head joints and the mouth. Setting these values disables the normal head motion control but can be useful, e. g. for a special walking type that holds the ball with the head.

3.7.1.3 Odometry correction values

Due to slippage the effective speed a walk produces differs from the calculated speed the feet have on ground and it depends quite heavily on the current underground. Therefore the maximum speed of a walk has to be measured manually. The measured speeds are stored in a file on the robot's memory stick and, they are used to correct the leg speeds so that the resulting speed of a certain gait matches the motion request.

3.7.1.4 Inverse Kinematics

After the desired leg position is calculated, it is necessary to calculate the required leg joint angles to reach that position. First, the knee joint angle is calculated as it is determined by the distance from shoulder joint to the paw. Next, the angle of the shoulder joint that rotates the leg to the side is calculated from the side distance between the paw and the shoulder. Finally, the distance from the shoulder to the paw in robot body direction defines the other shoulder joint angle.

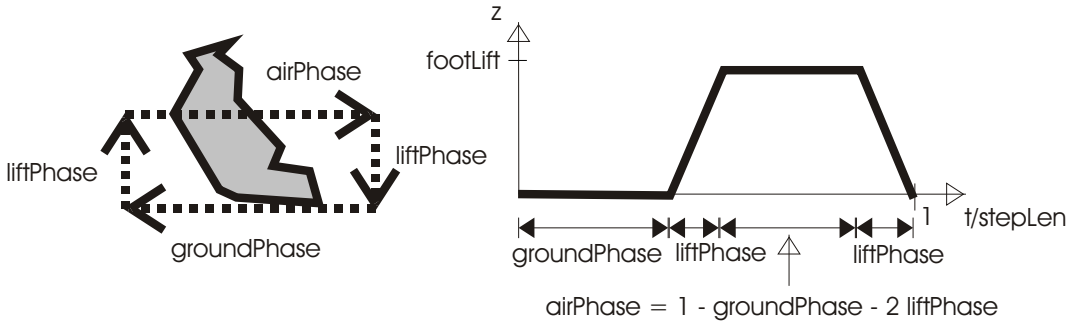


Figure 3.25: Timing of one step cycle

3.7.2 Special Actions

Special actions are all motions of the robot that are not generated by their own algorithms but merely consist of a sequence of fixed joint positions. Currently this includes a wide variety of kicks with which it is possible to play the ball from different positions relative to the robot to various directions. The behavior is responsible for choosing the correct kick according to the position of the ball and the game situation.

The module *SpecialActions* is responsible for performing these motions. It receives the currently requested motion and produces joint angles as well as the odometry vector of the resulting movement.

The module implements a chain of nodes which is traversed every time the module is executed. These nodes either contain joint data, transitions, or jump labels.

Joint data nodes contain angles for all joints which are sent to the robot as well as timing information that state for how long these values will be sent.

Transition nodes contain a destination node and an identifier for the target special action. If the currently requested motion matches the target, the transition is followed. By this mechanism the nodes will be traversed. This ensures that the requested special action as well as the transitions from the current motion get executed. With transitions it is possible to define that another action has to be executed before the requested action, e. g. grabbing the ball before kicking.

The nodes for each special action are specified in a special description language which is compiled into C code with its own compiler which is described in section 5.4. The generated code is part of the special action module. For each special action there is one file in the description language which contains all the necessary joint data and transition statements.

In addition, there is one special file called *extern* which serves as entry point to the module. It contains transitions to all special actions of which the correct one will be executed when the module is entered from other motion types. *extern* also serves as special transition target for leaving the special action module. If another motion type is requested, the special action module continues until a transition to *extern* is reached. By this the current special action will always be finished, avoiding, e. g., starting to walk while standing on the head.

The odometry data is calculated from the current movement speed and direction that are taken from a table containing values for all special actions. This table can contain information about the result of completely executing a special action once, e. g. that the bicycle kick turns the robot by 180 degrees. In addition the table may contain entries giving a constant speed for a special action.

3.7.3 Head Motion Control

The module *HeadControl* determines where the robot is looking at. It receives *head control modes* from the behavior control and generates the required *head motion requests* which contain the angles of the three head joints. These requests are sent to the module *MotionControl* that forwards them directly to the motors (cf. Sect. 3.7). Furthermore, *HeadControl* receives sensor data and the internal world model.

3.7.3.1 Head Control Modes

Since the robot can only see a small portion of its environment, it is necessary to have its head (and thus its camera) point in certain directions depending on the situation the robot is facing. A number of such situations have been identified and suitable head motions where developed, called *head control modes*. Some modes are more elemental, such as the *look-at-point* mode, whereas others utilize the basic modes (or the concepts of these modes) to achieve more complex solutions such as tracking the ball. The most interesting modes are:

Look at point. If this mode is selected, the robot will look at a certain position in 3-D space specified in its own system of coordinates. It compares the current rotation angles of the head with the desired angles (calculated from the line pointing from the camera to the point in 3-D space). The difference between the two is used to turn the head.

Search for Landmarks. In this mode the robot searches for landmarks, i. e. flags and goals. The head moves in an ellipse motion scanning the area in front of the robot. No knowledge of the world is used to guide the scanning motion.

Track the ball. This mode was developed to allow the robot to track the ball successfully while maintaining its bearing on the field. To achieve this, a robust ball tracking had to be established that allows the robot to look up at landmarks occasionally. To minimize the time needed for the latter, world model information is used to predict where landmarks should be. The robot looks up in the direction of the landmarks every two seconds and then quickly returns its gaze to the ball (this motion takes less than a second). The visual information gained is used to improve the localization. For obvious reasons this method does not work well if the robot's localization is very bad. However, looking up and away from the ball in most cases yields information about the robot's whereabouts even if the intended landmark cannot be seen. Only those landmarks that can actually be looked at are used. Landmarks that are close to the robot are preferred. To improve ball tracking, the ball's speed is used

to predict where the ball will appear next. The quality of the localization is not taken into account.

3.7.3.2 Speed Adjustment

It is possible to adjust the speed of all head motions. A low speed leads to smooth head motions which prevent fuzzy pictures.

3.7.3.3 Joint Protection

After destination joint angles and the speed to reach them have been calculated the actual joint angle values are determined. This is done by setting positions between the starting and destination position. This position is calculated not only respecting the selected speed to reach the destination but also by limiting maximum speed and acceleration. Therefore, the head performs very smooth motions, accelerating and decelerating softly. Extreme accelerations occurring on rapidly changing destination angles are prevented. As a result, the wear and tear of the joints is reduced.

Chapter 4

Challenges

In addition to the normal robot soccer contest, there is a so-called challenge in the Sony Legged Robot League comprising three different tasks to be solved by the teams. In 2002, there were the *pattern analysis challenge*, the *collaboration challenge*, and the *ball challenge*. Although nothing worked as expected, the GermanTeam still reached the sixth place during this contest.

4.1 Pattern Analysis Challenge

The task of the pattern analysis challenge was to recognize three different patterns chosen from a set of five. The patterns were placed over two flags and a goal and could be rotated arbitrarily around their surface normal.

The robots behavior for this challenge was written in XABSL. A special perceptor is started when the robot stands in front of a pattern. The perceptor marks the centers of black or white rectangles. This is done by finding large differences in the y-channel of the image while scanning horizontally and vertically (cf. Fig. 4.1).

Then, all center points of the black or white rectangles are clustered, connecting all points that are close together (cf. the thin vertical and horizontal lines in Fig. 4.1). It is assumed that the pattern is in the center of the image. Therefore, the cluster closest to center is selected.

For shape recognition, an approach was implemented that calculates the convex hull of the points in the selected cluster. Then, it is checked whether there are holes in the shape of the pattern, i. e., whether points on the convex hull exist, that are far away from any point in the pattern. For this, the center points of the lines of the convex hull are used, not the vertices. Then, the following rules are applied:

- If there is more than one hole in the shape of the pattern, it is assumed to be a “T”.
- If there is exactly one hole, the pattern is an “L”.
- If there are no holes, the pattern has a convex shape, i. e., it is either a triangle, a square. or a rectangle.

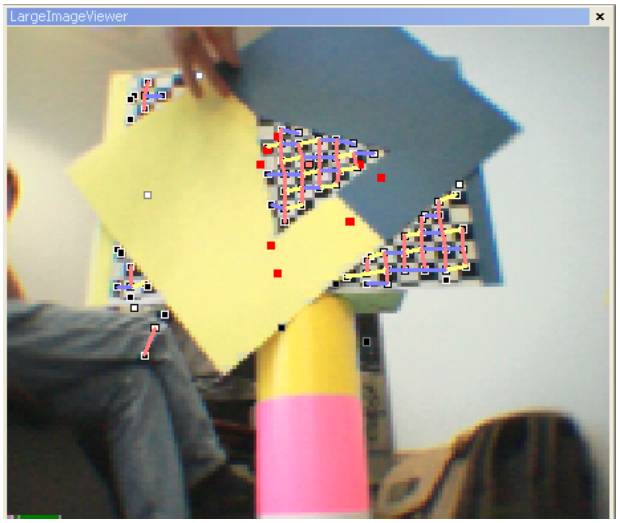


Figure 4.1: Recognized centers of checkers

In the latter case, a quadrilateral is calculated containing the leftmost, rightmost, upmost, and lowest point of the convex hull.

- If two of these points are close together, the pattern is a “triangle”.
- If the distances between the four vertices differ more than a certain threshold, the pattern is a “rectangle”. Note that the width to height ratio of the image is corrected before based on the tilt of the head.
- Otherwise, the pattern is a “square”.

During the competition the robot said *T* for all three patterns. It seems that the pattern was merged with the background, resulting in a shape with several holes in it. However, as one of the patterns really was a “T”, one answer was correct.

4.2 Collaboration Challenge

The task of the second challenge was to rotate a bar 180 degrees and to carry it from the middle of the field to the penalty area. The bar was 90 centimeters long and had blue and red markers at the ends. As the bar was to large to be moved by one robot alone the task had to be accomplished by two robots in collaboration.

The original approach was to write a special bar perceptor to recognize the markers as well as the white in the middle of the bar and thereby calculating position and direction of the bar. Although the perceptor showed some very good results, there was not enough time to achieve the necessary reliability and stability in localizing the bar.

Therefore a simpler approach was implemented that made use of the fact that the starting position of the bar was known. Both robots walked to the ends oriented along the bar only relying

on normal localization. A simplified bar perceptor was implemented that only detected the white ends of the bar and used them to correct the relative position to the bar as the self-localization was not precise enough to position exactly in front of the bar. The next step was to walk with both robots onto the bar. A special walking style was used for this task as the normal walking was too low to walk on the bar. Once on the bar the robots could very hardly lose the bar. It was therefore easy rotating it until localization indicated a 180 degrees turn and then pushing it to the penalty area. Synchronization of both robots could easily be acquired using wireless communication. After each step the robot waited until the other signaled it has finished that step, too.

While this approach worked quite good in several tests unfortunately in the competition the robots failed to get onto the bar correctly and therefore did not succeed in rotating the bar. Regrettably there was not enough time to test the simplified perceptor especially under the lighting conditions on the competition field.

4.3 Ball Challenge

The strategy for this challenge was kept as simple as possible: Find a ball, walk to an appropriate kicking position and finally kick the ball into the goal. Meanwhile try to avoid obstacles.

4.3.1 Architecture

This challenge has been used as a testbed for a different approach of behavior control. Instead of modeling the complete behavior as a large state machine, only a small state machine with eight different states was implemented. It simply consists of a few states representing essential tasks like *Search ball*, *Kick forward* or *Go to Ball*. These states use a potential field to navigate on the field.

4.3.2 Modeling of the Potential Field

The entire potential field consists of different single potential fields. There are several objects with individual influences. The potential influence is similar to the electrical charge in an electrical field. The potential fields of all objects are added and the value at a specified position indicates the influence at this point. The combined potential field is used to determine the next position where the robot should move to. For the potential field approach in RoboCup see also [18].

4.3.2.1 The Different Objects

The potential field of an object is characterized by three basic parameters. These are the influence radius for the area of potential influence, the maximum potential and the minimum potential. For some objects, e. g. for the robots, an additional parameter is needed.

The parameters are read from a configuration file. This makes it possible to test different sets of parameters without recompiling the code.

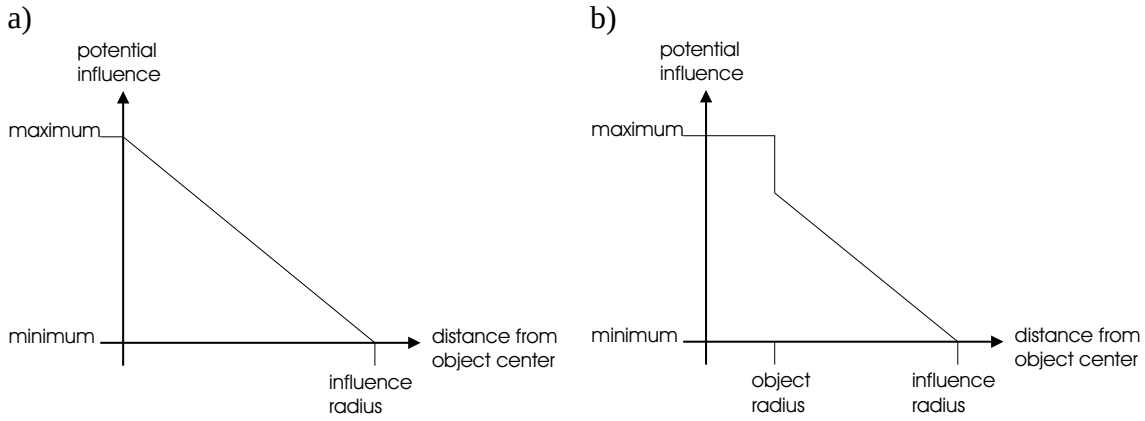


Figure 4.2: Graphs of potential field object functions. a) Ball b) Robot

For each object the value of the potential can be computed at a given position. To reduce calculation time, a linear function is used for this. The values are depending on the distance from the center of the object and the maximum and minimum value of their potential influence (cf. Fig. 4.2).

To get the value of the potential at a particular position the influences of all objects have to be added. The following part describes the potential field and additional parameters of all objects used.

Ball. For the ball object no additional parameters are needed. There is no peculiarity in calculation.

Robots. An additional parameter for the radius of the object is used. As there is no direction calculated by the perception of other robots, half of the robots length is taken. If the position, where the potential influence should be calculated for, lies inside the object radius, the maximum value is returned. Otherwise the value is calculated by the linear function.

Border. For the border of the game field the width of the border is needed. If the requested position lies on the border, the maximum value for this object is returned, otherwise it is calculated. Additionally it is tested if the destination is in the goal area. In this case, the goal is also interpreted as a border.

Game field. For the potential field of this object a target position is needed. This position is the point of interest, where the robot is heading for. So the potential field descends to this point. The influence radius must be big enough to cover the whole game field.

4.3.3 Visualization of the Potential Field

In RobotControl it is possible to activate a visualization of the potential field for the field view. The game field is divided into rectangles. For the middle of each rectangle, the potential value is calculated and interpreted as a gray scale value. Values under 0 or above 255 are cut off. Areas of

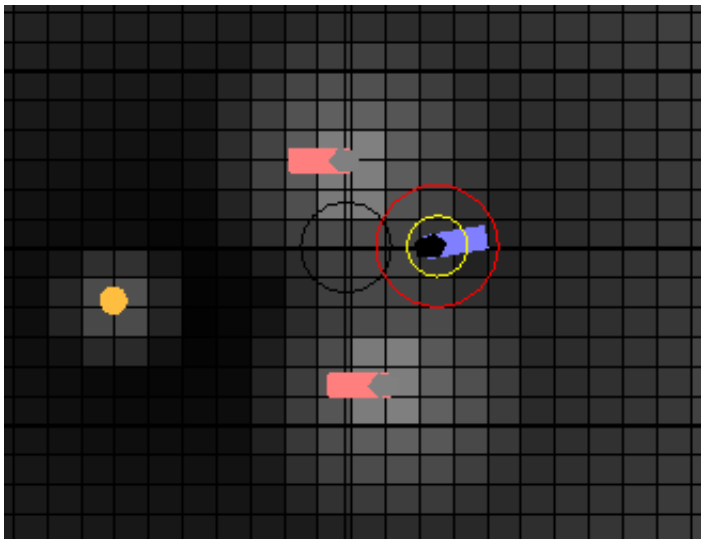


Figure 4.3: Visualization of the potential field. A robot getting stuck in a local minimum.

high potential values are displayed from light gray to white and areas with low potential values from dark gray to black (cf. Fig. 4.3).

4.3.4 Path Planning

Having modeled the field, an algorithm is needed to find the next position to walk to.

4.3.4.1 Gradient Descent

To move the robot to a target position having the lowest potential on the field, the gradient descent approach tries to find a position in the robot's near environment, which has a lower potential than the current position. This is done by computing a number of positions on a circumference around the robot. For each position the potential is calculated. After selecting the position with the lowest potential, a motion request leading to this position is generated. This process is iterated until the robot is near the target.

4.3.4.2 Local Minima

One problem using the gradient descent approach are local minima. As only the near environment is considered, there are several possible constellations, e. g. two robots standing close to each other, causing the robot to choose a position which is suboptimal and does not lead to the target. One example is shown in Fig. 4.3.

To avoid such situations, the A* algorithm [11] may be used to find a path to the target. An implementation has begun but due to the lack of time it was not finished in time.

4.3.5 Results

After a lot of successful tests with at least six goals in three minutes, the robots failed during competition. Unfortunately they obstructed each other and did some kicks to the goalposts. After three minutes only two goals had been scored, worse than during any test.

In spite of this disappointment, the potential field approach seemed to work well. It was capable of making the robot move to a target with adequate speed and precision and without any major mistake.

Chapter 5

Tools

The GermanTeam spent a lot of time on programming the tools that do not run on the AIBO platform but that helped very much in the development of the soccer software.

In section 5.1 and 5.2 two very similar programs are described: SimGT2002 and RobotControl. They both have in common:

- The complete source code that was developed for the robot is also compiled and linked into these applications. That allows algorithms to be tested and debugged very easily. New source code can be tested with the tools before compiling it for the robot and testing it on the field.
- As the interfaces of the source code to the physical robot are very narrow, the robot could be easily replaced by a simulator.
- They provide a lot of debugging and visualization tools.

Section 5.3 describes a router software for the wireless network that dispatches messages between the robot and SimGT2002/RobotControl.

The *Motion Configuration Tool* (cf. Sect. 5.4) was used by the GermanTeam for the easy development of special actions, e. g. the kicks.

5.1 SimGT2002

SimRobot is a kinematic robotics simulator that was developed at the Universität Bremen [16]. It is written in C++ and is distributed as public domain [1]. It consists of a portable simulation kernel and platform specific graphical user interfaces. Implementations exist for the *X Window System*, *Microsoft Windows 3.1/95/98/ME/NT/2000/XP*, and *IBM OS/2*. Currently, only the development for the 32 bit versions of Microsoft Windows is continued, and a Java version is under development.

SimRobot consists of three parts: the *simulation kernel*, the *graphical user interface*, and a *controller* that is provided by the user. The GermanTeam 2002 has implemented the whole simulation of up to eight robots including the inter-process communication described in appendix C

as such a controller, providing the same environment to robot control programs as they will find on the real robots. In addition, an object called *the oracle* provides information to the robot control programs that is not available on the real robots, i. e. the robots' own location on the field, the poses of the teammates and the opponents, and the position of the ball. On the one hand, this allows implementing functionality that relies on such information before the corresponding modules that determine it are completely implemented. On the other hand, it can be used by the implementators of such modules to compare their results with the correct ones.

SimRobot, linked with the special controller that provides the interface to the robots and linked with the robot code is called *SimGT2002*. The following sections will give a brief overview over SimRobot, and how it is used to simulate a team of robots.

5.1.1 Simulation Kernel

The kernel of SimRobot models the environment, simulates sensor readings, and executes commands given by the controller. A simulation scene is described textually as a hierarchy of objects. Objects are bodies, emitters, sensors, and actuators. Some objects can contain other objects, e. g. the base joint of a robot arm contains the objects that make up the arm.

Emitters. SimRobot uses a very abstract model of measurable quantities. Instead of defining objects as lamps or color cameras that emit and measure light, it uses objects that emit intensities of particular *radiation classes* (emitters) and objects that measure these intensities (sensors). Hence, it is up to the user to define some of these abstract classes to represent real phenomena. In case of the Sony AIBO robots, the radiation classes 0, 1, and 2 represent the three channels of the YUV color model.

There are only two types of emitters in SimRobot: *radial emitters* send their radiation to all directions, whereas *spot emitters* have a certain opening cone. In addition, an ambient intensity can be specified for each radiation class that defines the base intensity for all surfaces in a simulation scene. Hence, the surfaces that are not reached by any of the emitters in a scene still have a sensible radiation signature. In the RoboCup simulation, only a high degree of ambient white light is used and no emitters.

Bodies. Currently, bodies can only be modeled as a collection of polygons. Each polygon has a radiation vector that defines its appearance—together with the radiation of the emitters that reaches the surface. The GermanTeam uses color tables to map the colors measured by the robot's camera onto *color classes*. To avoid to have two different color tables, one for the real robot and one for the simulation, the simulation scene is automatically colored according to the actual color table. This is also the reason why no additional emitters are employed for illumination. Their influence may have changed the colors of the surfaces, resulting in a wrong mapping from colors to color classes in the image processing modules of the simulated robots.

Actuators allow the user or the *controller* to actively influence the simulation. They can be used, e. g., to move a robot or to open doors. Each actuator can contain other objects, i. e. the objects

that it moves. SimRobot provides four types of actuators: rotational joints, translational joints, objects moving in space in six degrees of freedom, and vehicles with typical car kinematics, i. e. with a driving axle and a steering axle. SimRobot is only a kinematic simulator; thus it cannot directly simulate walking machines. Therefore, the motion of the simulated AIBOs is generated by a trick: the GermanTeam 2002 robot control program has its own model of which kind of walk will generate a certain motion of the robot. This model is also employed for the simulation. Thus, the simulated robots will always behave as expected by their control programs—in contrast to the real robots, of course. In addition, the body tilt is simulated. This is performed on the assumption that the body roll is always zero. In each simulation step, the distance of the four feet to the ground is determined. Then the robot body is moved and rotated around the tilt axis in a way that at least one foreleg and one hind leg touches the ground. This approach only fails if the feet are not the lowest parts of the robot's body, e. g. when it performs the "getup" action.

Sensors. SimRobot provides a wide variety of sensors. However, only three types of information can be sensed:

Intensities of Radiation. There are two types of cameras that allow measuring two-dimensional arrays of intensities of radiation. The *camera* object imitates normal pinhole cameras, and is used to simulate AIBO's color camera. The *facette* simulates cameras with a spherical geometry, i. e. the angle between all adjacent pixels is constant. The sensor readings can be calculated using *flat shading*, i. e. each surface has a single combination of intensities, or with different intensity signatures for each pixel. In addition, it is possible to determine shadows.

Distances. There are several sensors that measure distances. A *whisker* can imitate the behavior of an infrared sensor. On the one hand, it is used to simulate the PSD sensor in AIBO's head. On the other hand, whiskers could be employed to implement the ground contact sensors in the feet of the robots. As these sensors are not used by the GermanTeam, this has not been implemented yet.

Collision detection. For every actuator, it can be detected whether a collision-free execution of the last command was possible. This information is not available in reality, but it is required for the simulator to suppress motions that result in collisions. However, as each robot has 20 degrees of freedom, this costly calculation is even too slow for a single robot, but it surely is for eight. Therefore, the current simulation does without a collision detection.

Apart from the latter, all sensor readings can be disturbed by a selectable amount of white noise.

5.1.2 User Interface

The user interface of SimRobot includes an editor for writing the required scene definition files (cf. Fig. 5.1, upper left window). If such a file has been written and has been compiled error-free,

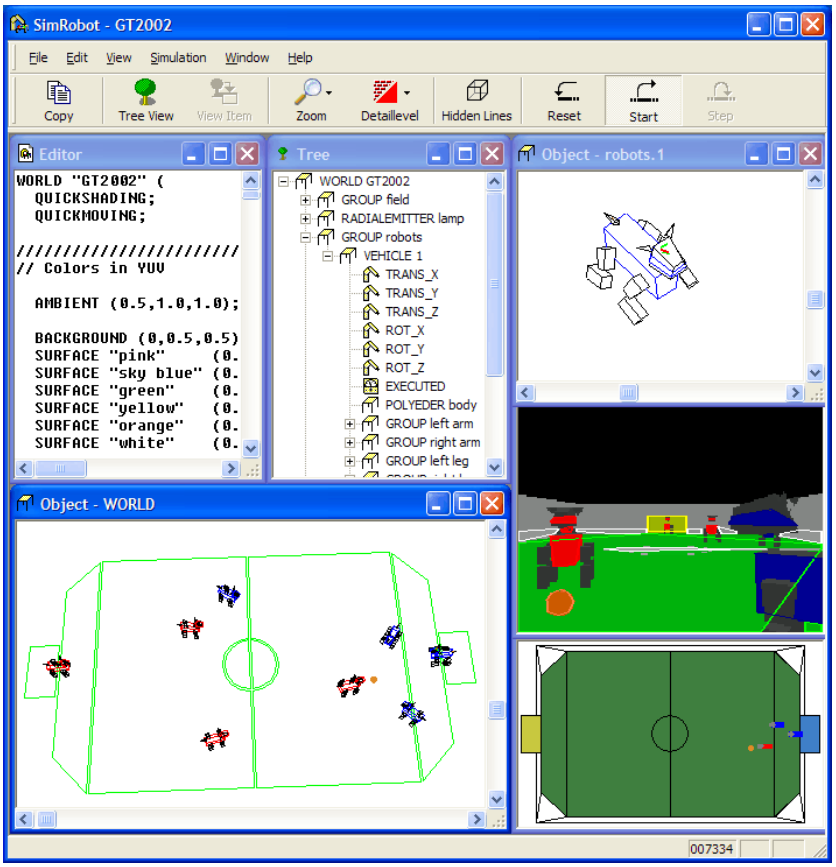


Figure 5.1: SimRobot simulating the GermanTeam 2002.

the scene can be displayed as a tree of objects (cf. Fig. 5.1, upper middle window). This tree is the starting point for opening further views. SimRobot can visualize any object and the readings of any sensor that are defined in a scene. Objects are displayed as wire-frames with or without hidden line removal (cf. Fig. 5.1, lower left and upper right window) and parts of the scene can be hidden. In case of the lower left window, the flags, the goals, and the field border are not displayed.

Sensor data can be depicted as line graphs, column graphs, monochrome images, and color images (cf. Fig. 5.1, middle right window). In addition, depth images can be visualized as single image random dots stereograms. Any of these views and a numerical representation of the sensory data can be copied to the system's clipboard for further processing, e. g., in a spreadsheet application or a word processor. The whole window layout is stored when a scene is closed and restored when SimRobot is started again with the same scene.

SimRobot also has a console window that can be used to enter text and to print some data on the screen. SimGT2002 uses this window to print text messages sent by the robot processes, and it allows the user to enter a large variety of commands. These are documented in appendix I.

5.1.3 Controller

The controller implements the sense-think-act cycle; it reads the available sensors, plans the next action, and sets the actuators to the desired states. Then, SimRobot performs a simulation step and calls the controller again. Controllers are C++ classes derived from a predefined class *CONTROLLER*. Only a single function must be defined in such a controller class that is called before each simulation step. In addition, the controller can recognize keyboard and mouse events. Thereby, the simulation supports to move around the robots and the ball.

A very powerful function is the ability to insert *views* into the scene. These are similar to sensors but in contrast to them, their value is not determined by the simulation but instead by the controller. This allows the controller to visualize, e. g., intermediate data. In fact, the middle right window in figure 5.1 is a view that contains a camera image overlaid by the so-called blob collection, i. e., colored octagonal areas detected by the robot control program's image processor. The lower right window is completely drawn by the controller: a field with the visualization of the estimations of a robot's own pose (in this case the right goalie), the locations of some other robots, and the position of the ball.

The whole environment that the processes of a robot control program will find on a real robot has been resembled as such a controller. It supports multiple robots, each robot can run multiple processes, these processes can communicate with each other, and also the communication between different robots is supported. Thus the code of a whole team of four communicating robots runs in the simulator.

5.2 RobotControl

In contrast to SimGT2002 that evolved from a pure simulator, RobotControl (cf. Fig. 5.2) was initially intended to be a general support tool that should help to increase the speed and comfort of the software development process.

First, it functions as a debugging interface to the robot. Via the wireless network or a memory stick, messages can be exchanged with the robot. Almost all internal representations of the robot (images, body sensor data, percepts, world states, sent joint data) and even internal states of modules can be visualized.

In the other direction, many intermediate representations of the robot can be set from RobotControl. For instance, one can send motion requests that are normally set by the behavior control module of the robot to test the motion modules separately.

Second, as in SimGT2002, the complete source code for the robots is compiled into RobotControl and encapsulated in "simulated robots". The debugging interfaces of RobotControl function both for the simulated and the physical robots. So it is possible to test source code without switching to a robot. The virtual robots can receive their data from a simulator (which was adapted from SimGT2002), a real robot, or a log file. The GermanTeam could develop its vision modules long before they had a wireless network connection to the robot by testing the algorithms on log files.

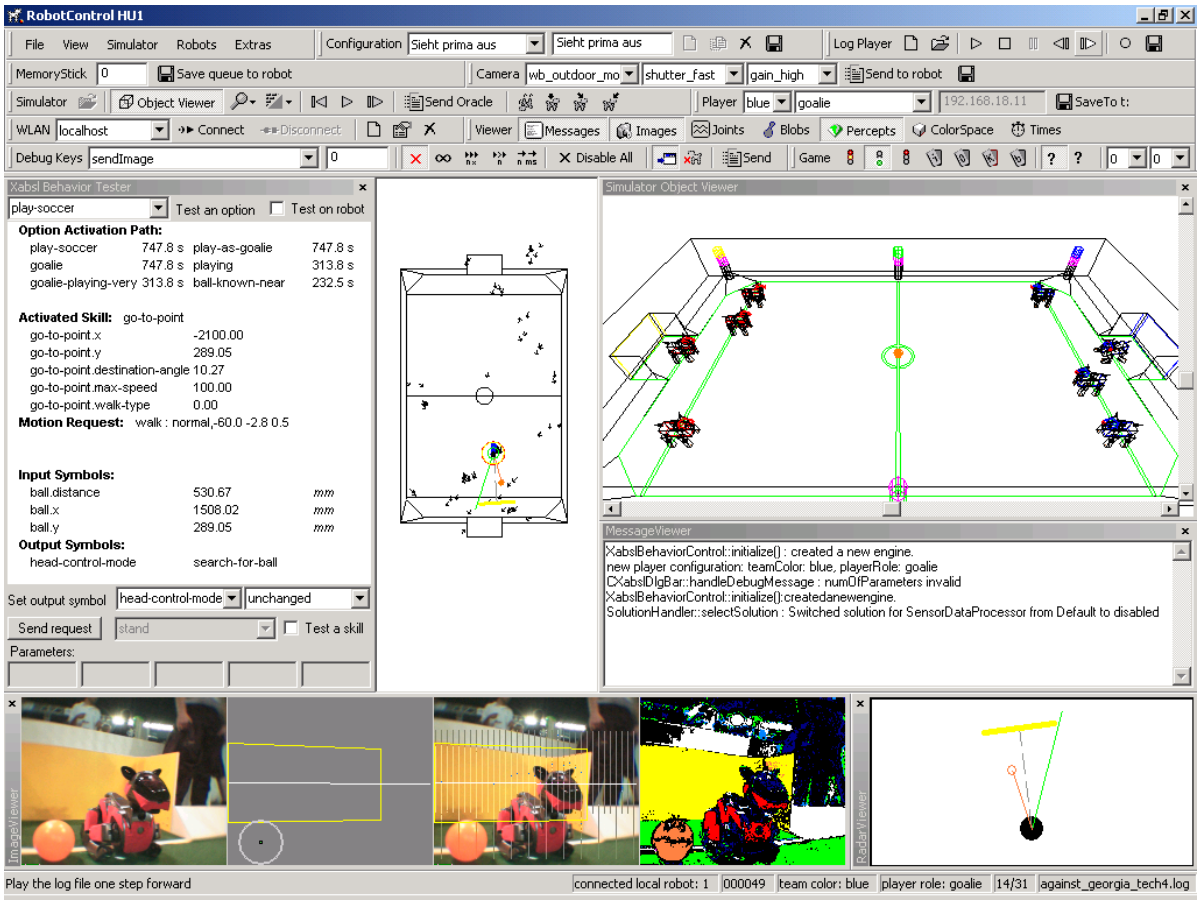


Figure 5.2: The RobotControl application

In addition, a variety of other helper tools is integrated into the application, e. g. for color calibration or for copying data to the memory sticks.

Almost all of RobotControl's functionality was programmed into toolbars and dialogs. There are simple interfaces to create and embed them in the application, so that many team members could easily program graphical user interfaces for their debugging needs. Appendix J describes that in detail.

This is also one of the two main differences between RobotControl and SimGT2002: In SimGT2002 most of the interaction with the program is done using a text console whereas in RobotControl many graphical user interfaces exist. As many tasks require a graphical user interface, e. g. creating color tables, SimGT2002 provides only a small portion of the functionality of RobotControl.

The second difference is that RobotControl can only communicate with exactly one simulated or physical robot at the same time. Although it can simulate up to 4 robots simultaneously, only one of them can be "connected" to the application.

RobotControl has a very modular structure (cf. Fig. 5.3): it almost only consists of the simulated robots, a simulator, the tool bars, the dialogs, an interface to the Wireless network, and

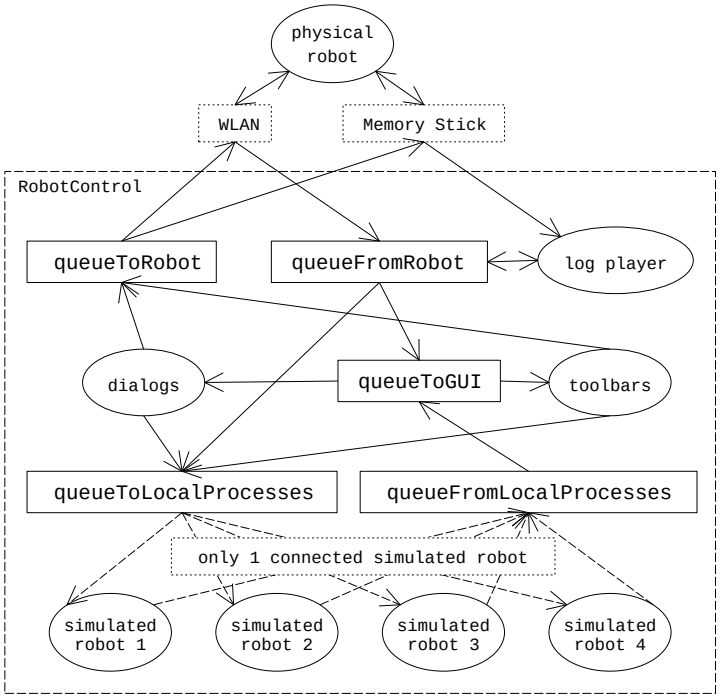


Figure 5.3: Data flow in the RobotControl application

message queues (cf. Sect. E.1) between these units. Dialogs and toolbars only communicate via the queues.

To send data to the robot or to the simulated robot, messages are put into the *queueToRobot*/*queueToLocalProcesses*. The *queueToRobot* can be sent to the robot via the wireless network or by writing the queue onto a memory stick. The *queueToLocalProcesses* is sent to the simulated robot that is currently connected to the application. One can change, which of the 4 robots shall be connected. Messages from the robot can be received over the wireless network or from a log file that was written from the robot onto the memory stick. They arrive in the *queueFromRobot*. From there, some of the messages are first sent to the simulated robot *queueToLocalProcesses*, some directly to the *queueToGUI*. All messages from the currently connected simulated robot arrive in the *queueFromLocalProccesses* and later in the *queueToGUI*. At last, the messages in the *queueToGUI* are distributed among the dialogs and toolbars. The *log player* records messages from the *queueFromRobot* and stores them there again when playing a log file.

5.3 Router

It is desirable to exchange data between the tools running on the PC and the robots. This year, a wireless network was introduced into the Sony Legged Robot League. On the side of the PC, only Linux and CygWin are supported platforms for the wireless communication. However, the

GermanTeam uses tools running natively under Microsoft Windows. Therefore, they have no direct access to the new wireless communication capabilities of the robots.

That is the point where the *Router* comes into play. It functions as a mediator between the physical robots and the native Windows tools. On the one hand, it communicates with up to eight robots using the *tcpGateway*. On the other hand, it exchanges data with the Windows tools via a different IP-port for each robot. The data transferred are *message queues*, one from the PC to each robot, and a second back from each robot to the PC.

That way, RobotControl can send a message queue to the robot by sending it to the appropriate IP-port (by convention the least significant byte of the robot's IP-address plus 15000). The Router will receive the queue and forward it to the *tcpGateway* on the PC. Then the gateway will send the queue to the *tcpGateway* on the robot via the wireless network. The latter will again forward the message queue to the *Debug* process on the robot. The other way round, the robot can send a message queue to RobotControl.

start.bash. The router needs a couple of configuration files to work properly, namely *connect.cfg*, *object.cfg*, and *port.cfg*. These files are small and quiet easy to edit manually as long as one only wants to communicate with one robot using the *tcpGateway*. But they are hard to read and maintain for multiple robots, e. g. for two robot teams of four players each, complete point-to-point connections between all robots of the same team, *roboCupGameManager* control for these eight robots, and debug connections to the PC.

Because these configuration files have to be changed to adapt to different conditions (smaller number of robots, only one team, no *roboCupGameManager*), the script *T:\GT2002\Bin\start.bash* automates that work. The following message is generated by the script after its first execution, i. e. when *connect.cfg*, *object.cfg*, and *port.cfg* do not exist. It explains the usage of the script:

```
usage: start.bash [-gm] [ IP | ( subnet [ auto |
         ( A1 [A2 [A3 [A4 [B1 [B2 [B3 [B4]]]]]] ) ] ) ]
  where -gm starts the RoboCup Game Manager
        subnet is equal for all robots followed by .Ai or .Bi
        A1 is the robot to be controlled by RobotControl
        Ai are own robot IP adresses
        Bi are opponent robot IP adresses
example1: start.bash       # uses old *.cfg
example2: start.bash -gm 10.0.1.100
example2: start.bash 10.0.1  100 101
example3: start.bash 10.0.1  auto
example4: start.bash 10.0.1  100 101 102 103   110 101 102 103
```

If the script is parameterized correctly, it performs all tasks required to start the router:

- Stop previous instances of the *ipc-daemon* and the *oobjectManager* with *stop.bash*,
- start a new instance of the *ipc-daemon*,

- start the *oobjectManager* which will start the router, and, if desired, the *roboCupGameManager*,
- and kill the *ipc-daemon* after the termination of the *oobjectManager*.

stop.bash. As already mentioned above, there is second script called *stop.bash* in the same directory that removes all processes started by *start.bash*, including the *ipc-daemon* and its temporary files.

5.4 Motion Configuration Tool

The *Motion Configuration Tool* was developed by the team of the Humboldt-Universität zu Berlin for RoboCup 2000, and it was modified to fit into the new modular architecture in 2002. It compiles a set of motion specifications described in a special language into C code with a Flex/Bison-based parser. The parser checks the motion set defined for consistency, i. e., it checks for missing transitions from one motion to another and for transitions to unknown motions.

With this tool it is possible to generate motions that consist only of fixed sequences of joint positions quickly and easily. This is the case for all kicks implemented by the GermanTeam as well as some other motions including, e. g., head stand and ball holding. The resulting C code is integrated into the *SpecialActions* module (cf. Sect. 3.7.2).

For interactively testing motions and their transitions, RobotControl provides a dialog to request specific motions. The requested motion and the transition from the current one will be executed immediately (cf. Sect. J.5.2). Furthermore, a second dialog is provided to transmit new motion descriptions to the robot and execute them without the need to recompile anything (cf. Sect. J.5.4).

Motion Description Language. The specification for a single motion consists of the description of the desired action and the definition of a set of transitions to all other motions. This is simplified by using groups of motions, e. g., it is possible to define that the transition from motion X to any other motion always goes via the motion Y.

As most simple motions (such as kicking or standing up) can be defined by sequences of joint data vectors, a special motion description language was developed, in which all motions are defined. Programs in this language consist of transition definitions, jump labels, and lines defining motor data. A typical data line looks like this:

```
~~~  ~~~  -350 -190 1750  -350 -190 1750  -1840 -40 2500  -1840 -40 2500  1 25
```

The first three values represent the three head joint angles, the next three values describe the mouth and the tail angles, followed by the twelve leg joint angles, three for each leg, all angles given in milliradians. The last but one value decides whether specified joint angles will either be repeated or interpolated from the current joints angles to the given angles. The last value defines how often the values will be repeated or over how many frames the values will be interpolated,

respectively. The tilde character in the first six columns means that no specific value is given, i. e. “don’t care”. This has special importance for the head joint angles as it allows head motion requests to be executed.

Chapter 6

Conclusions and Outlook

The GermanTeam was founded a few months before the RoboCup 2001 in Seattle. The new team members had only little time to get into the source code previously developed by the team from the Humboldt-Universität zu Berlin. This lead to a certain dissatisfaction with the achieved results in 2001.

But in this year, as there were was almost 10 months of time, the team had the chance to rewrite the whole programs nearly from scratch. The software architecture, debugging mechanisms, and support tools were completely renewed. Although this consumed a lot of time, we think that this paid out.

Despite all the problems that arise when software is developed by a group of persons distributed over different towns, we recommend to build up national teams as the GermanTeam is one. Having enough participating team members, different solutions for single tasks can be employed and compared to each other. The different scientific backgrounds of the members from different universities enriched the project very much. At last, the rivalry between the single teams results in better solutions for single tasks.

Altogether we were quite satisfied with the results we achieved, and we are continuing that work. We hope to reach even better results in the competitions next year in Padova.

6.1 The Competitions in Fukuoka

6.1.1 Results

When the GermanTeam travelled to Fukuoka, the code (especially the behavior) was far from being finished. But from game to game a lot of improvements were made. All in all, we finished the round robin being second in the group (cf. Tab. 6.1). In the quarter final, we had to compete against last year's world champion UNSW and lost.

Although we only reached the quarter final, we were satisfied with the results. There were only two other teams that scored a goal against CMU, this year's world champion, i. e. Tokyo in the round robin and UNSW in the final. Only one other team could score against UNSW, i. e. CMU in the final.

Round Robin	
GermanTeam – Rome	5:0
GermanTeam – Tokyo	4:0
GermanTeam – CMU	1:3
GermanTeam – GeorgiaTech	4:1
Quarter Final	
GermanTeam – UNSW	1:6

Table 6.1: The results of the GermanTeam in Fukuoka

6.1.2 Experiences

Kicking the Ball. Except from being at the border of the field, the robots of the GermanTeam never tried to get behind the ball to reach a good kick position. Instead of this they always walked directly to the ball, and only when they reached it they chose one out of ten possible kicks dependent on the desired kick direction. To get the ball behind them, they chose the *bicycle kick*. It proved to be a very good strategy because they never lost time for getting behind the ball. Other robots that also tried to get for the ball did not prevent them from kicking.

Communication. Another important issue is that it is mostly fatal when two robots of the own team try to kick the ball at the same time. Therefore a big variety of approaches to prevent the robots from getting stuck together was implemented and tested. Using the wireless network, two robots that were close to the ball tried to negotiate which of them will kick the ball. The other robot then followed the ball in a safe distance. Although it sometimes happened that the robot closer to the ball recoiled from it, in general it was a very successful approach. Finally, the robots of the GermanTeam played better with wireless communication than without.

Position-Dependent Behavior. A good localization allows implementing completely new behaviors. As the GermanTeam had one this year, the position of the robots could be used for very many decisions. Each robot had its own “area of responsibility”, which prevented the robots from being all in the proximity of the ball. Besides that some “emergent” passes were possible. Very rarely, our robots walked into their own penalty area.

The RoboCup Game Manager by Sony makes the games look much more natural for the audience. As the robots of the GermanTeam walked on their own to their kickoff positions, it was almost never needed to get on the field for carrying the robots around. The current score that is transmitted by the program allows performing some dances after the game depending on if the game was lost or won. Next year, the robots may even change their behavior depending on the score.

Speed and Precision. Although the basic behaviors of the GermanTeam improved very much from the last year, they are still too slow and too imprecise. To win a RoboCup competition in

the Sony Four Legged League, one has to have the fastest and most precise motions, which we had not.

6.2 Future Work

The GermanTeam now owns a powerful code basis for the next year's work. So less time needs to be spent on the software architecture and more new approaches can be examined. For the RoboCup German Open in April 2003, each of the four universities will set up its own team based on the shared code basis with own solutions for different tasks. From their different research interests, the teams will also focus on different topics next year.

6.2.1 Humboldt-Universität zu Berlin

An important aspect of our future work will be the application of case based reasoning to robot control architectures and machine learning. The efforts will be pursued not only in the Sony Legged League but also in the Simulation League.

A behavior architecture called the "Double Pass Architecture" [5] has already been implemented in the Simulation League and will be applied to the Sony Legged League. It provides for long term "deliberator" planning and short time "executor" reactions. The executor allows quick reactions even for the options on the higher levels in the option hierarchy. This is made possible by using the reduced search space defined prior by the deliberator. It implements a kind of bounded rationality. Therefore, the state machine concept has to be extended for the two separate passes of the deliberator and the executor (the name "Double Pass Architecture" refers to these two passes). Many useful behaviors have been developed. Selecting the appropriate one becomes an increasingly difficult task. It becomes even more difficult if behaviors are combined to more complex ones, such as they can be described in the option hierarchy. The *Extensible Agent Behavior Specification Language* (XABSL) will be extended and adopted to that.

Another prerequisite of useful decisions is a reliable world model. In case of the Sony AIBO, knowledge about the environment is exclusively derived from the camera image. With this limited field of view, information gathering has to be optimized. First steps in this direction have been taken by actively scanning for landmarks using world model information (i. e. pointing the camera in a direction where a landmark should be according to the world model). A tighter coupling of information gathering and information processing turned out to be desirable rather then having the two run as separate processes. Active vision and attention based vision approaches will be examined. World and object modeling will be extended to make use of negative information (e. g. the ball was not seen) and to actively search for information that is needed (e. g. have the robot look for a specific landmark that is needed to clarify the robot's position on the field). Having both, complex behavior and reliable world model, the correspondence of situations and most appropriate actions have to be resolved. This will be done by methods of case based reasoning. Cases describe typical behavior in typical situations (e. g. standard situations). The recent situation is matched against the case base, and the most similar cases are analyzed for proposals of behaviors. The behaviors are adapted according to the recent situations. Problems to

be solved in the next steps include description of cases, definition of useful similarity measures and adaptation methods.

6.2.2 Technische Universität Darmstadt

The team in Darmstadt will continue their efforts towards a complete, efficient, and validated simulation of the four-legged robot's dynamics and its use for dynamic off-line optimization, on-line stabilization, and control of dynamic walking and running gaits. For the behavior control of cooperating robots as well as for object recognition it is planned to investigate alternative approaches to the already existing ones.

In 2001, the GermanTeam was able to measure the true positions of the robots on the field with a camera mounted at the ceiling. We propose to reintegrate such a mechanism in GT2003 at least for one half of the field. By this different localization methods can be compared on an empiric level. Besides with the exact data not only the self-localization but also a number of other algorithms can be tested and even automatically learned. This includes localization of other robots, the ball, odometry, and others.

6.2.3 Universität Bremen

First of all, the work on the *LinesSelfLocator* (cf. Sect. 3.3.3) will continue, enabling the robots to self-localize without the colored beacons.

As the use of the Markov-localization was quite successful in the GermanTeam 2002, probabilistic approaches will also be introduced to model the location of the ball and of the opponents. In contrast to the field of self-localization, in which the sensor resetting approach by [14] can only be used if an estimation of a global pose can directly be derived from the sensor readings, the sensor resetting method seems to be a promising approach for the probabilistic modeling of the locations of the opponents and the ball in a robot-centric system of coordinates. In addition, work that was done on tracking people [17] can be integrated in such an approach. The modeling of the world state will also exploit the ability of the robots to communicate.

After such a probabilistic world model has been realized, it will be investigated, how the uncertainties can influence the behavior, or whether even a probabilistic behavior control can be implemented. The German Open 2003 will be the testbed for such an approach.

6.2.4 Universität Dortmund

We will continue in developing a robust and detailed distributed world model. We will implement methods that consider the reliability of the particular percepts and classifications. For that, we will implement and evaluate a novel classification method that implies confidence information for every classification. Furthermore we are planning to develop a world-model interchange protocol (WIP), that allows including and excluding single robots from the world-model interchange transparently.

Chapter 7

Acknowledgements

The GermanTeam and its members from Berlin, Bremen, Darmstadt, and Dortmund gratefully acknowledge the continuous support given by the Sony Corporation and its Open-R Support Team. The GermanTeam thanks the organizers of RoboCup 2002 for travel support. The team members from Berlin and Bremen thank the Deutsche Forschungsgemeinschaft (DFG) for funding parts of their respective projects. In addition, the team members from Bremen thank the Deutscher Akademischer Austauschdienst (DAAD), the American Association for Artificial Intelligence (AAAI), and the Fachbereich 3 of the Universität Bremen for travel support. The team members from Dortmund thank the Deutsche Arbeitsschutz Ausstellung (DASA) and the Thyssen Krupp AG for their effective cooperation. Further, we thank the Deutscher Akademischer Austauschdienst (DAAD), Lachmann & Rink GmbH, the Dortmund Project, the Freundegesellschaft der Universität Dortmund e.V., and the Faculty of Information Science for travel support.

The GermanTeam uses a variety of code libraries and tools and also likes to thank the authors of them:

- This product includes software developed by the Apache Software Foundation (http://www.apache.org/).
- A code library called “Sizing Control Bars” from Cristi Posea. (http://www.datamekanix.com) is used for the dialogs in RobotControl.
- A code library for “Internet Explorer-like toolbars” from Nikolay Denisov (nick@actor.ru) is used.
- The code library “Grid Control” from Chris Maunder (cmaunder@mail.com) is used for the “Settings” dialog.
- The “dot” tool from the GraphViz collection (http://www.graphviz.org) is used for behavior documentation purposes.

Appendix A

Installation

The GermanTeam uses Microsoft Windows as development platform. The package provided was used under Windows 2000 and Windows XP. It may also run under Windows 98/ME, but that has not been tested. The center of the development process is Microsoft Visual C++; all parts of the system are edited and built with this software.

A.1 Required Software

As there are quite a large number of developers in the GermanTeam, the software installation must be flexible, because they use computers with different configurations. The GermanTeam uses two virtual drive letters for the installation, i. e. drive letters created with the shell command *subst*:

U: contains most of the software packages that are required for the development.

T: contains the source code of the GermanTeam.

The two drives have to be assigned before the installation of the software, and the assignment has to be repeated after each reboot. So this will normally take place in a batch file that is called during start-up, e. g.

```
subst t: "C:\Documents and Settings\%USERNAME%\
            My Documents\RoboCup"
subst u: "C:\Program Files\RoboCup"
```

Of course, the two folders have to be created before the virtual drives can be assigned to them.

There are two possible configurations: on the one hand, the GreenHills C++ compiler can be used for the development, on the other hand, the gcc can be employed together with the new Open-R SDK. There are tools that have to be installed for both possibilities and others that are

only required for one case. The following three sections list the software packages together with their necessary installation paths. In addition, the version numbers of the tools used are specified. That does not mean that other versions will not work, but only the given versions were tested.

A.1.1 Common Requirements

- Microsoft Windows 2000/XP
- Microsoft Visual C++ 6.0 SP5 (can be installed anywhere)
- CygWin 1.3.10 (*u:\cygwin*)
- CygIPC 1.10-1 (unpack it to CygWin-Path /). Do not install it as a service!
- gtk+ 1.3 (unpack it to CygWin-Path /)
- Open-R for CygWin 1.1.27 (as *u:\OPEN_R_WIN*)
- Doxygen (http://www.doxygen.org). The three binaries *doxygen.exe*, *doxysearch.exe*, and *doxytag.exe* have to be copied to the directory *t:\GT2002\Tools\Doxygen*. This has to be done after the source code of the GermanTeam has been installed.

A.1.2 GreenHills-Based Development

- Aperios 1.3.2 (*u:\aperios*)
- GreenHills C++ (*u:\green*)
- Open-R for Aperios R008 (as *u:\OPEN_R_SYS*), only works with RoboCup memory sticks

A.1.3 gcc-Based Development

- Open-R SDK and MIPS developer tools (CygWin-Path */usr/local/OPEN_R_SDK*)

A.2 Source Code

The source code has to be unpacked to *T:* generating a directory tree starting with *T:\GT2002*. There are several subdirectories under this root:

Bin contains the binaries of the programs running on the PC after they have been compiled.

Build contains all intermediate files during compiling.

Config contains the configuration files. Most of them will also be copied to *OPEN-R/APP/CONF* on the memory stick.

Doc contains this document[1] and the documentation generated by *Doxygen* from the source files.

Make contains makefiles, batch files, and Visual C++ project files. The *GT2002.dsw* is located here, i. e. the file that has to be launched to open Visual C++. In addition, there are some registry files (*.reg*) that can be added to the registry by starting them. For instance, the file *SimGT2002.reg* contains the window layouts of several SimRobot scenes in the *T:\GT2002\Config\Scenes* path.

Src contains all source files of the GermanTeam.

Tools contains additional tools, e. g. *Doxygen*.

The directory *Src* contains subdirectories that can be grouped in two categories: on the one hand, some directories contain code that runs on the robots, on the other hand, other subdirectories hold the code of the tools running on the PC.

A.2.1 Robot Code

DataTypes contains source files implementing classes the main purpose of which is to store information rather than to process it. Objects of classes defined here are often communicated between different processes or even different robots.

Modules contains source files implementing the modules of the robot control program. For each module there exist an abstract base class, one or more different implementations, and, at least if there are multiple implementations, a module selector that allows switching between the different implementations.

Platform contains the platform dependent part of the robot code. There exist three subdirectories containing the platform specific implementations for *Aperios*, *Win32*, and *Linux* (CygWin). A fourth directory *Win32Linux* contains implementations that are shared between CygWin and Windows. *Platform* itself contains some header files that automatically include the code for the right platform.

Processes contains one subdirectory for each process layout, and each of these subdirectories contains one directory for each process, the *object.cfg*, the *connect.cfg*, and the *.mcf*/*.ocf*-file required for the process layout.

[1] Or at least it is a good idea to place it there.

Tools contains all that does not fit into the other categories. The mathematical library (cf. Sect. 2.1.4) can be found here, the implementation of streams (cf. App. D) and message queues. However, the code of the Windows tools such as RobotControl cannot be found here.

A.2.2 Tools Code

MotionConfigToolConsole contains the code of the tool (cf. Sect. 5.4) to create new motions, i. e. special actions (cf. Sect. 3.7.2).

RobotControl contains the code of the tools with the same name (cf. Sect. 5.2).

Router contains the code of the mediator between the robots and the Windows-based tools such as RobotControl (cf. Sect. 5.3).

SimRob95 contains SimRobot, a kinematic robotics simulator. It is the framework for SimGT2002 (cf. Sect. 5.1) and the simulation integrated into RobotControl (cf. Sect. 5.2).

A.3 The Developer Studio Workspace GT2002.dsw

The Developer Studio Workspace GT2002.dsw contains several projects, most of them in different configurations:

Documentation. This project creates the documentation of the code using *Doxygen*. The project the documentation of which will be generated can be selected between *GT2002*, *RobotControl*, and *SimGT2002*. Please note that legacy code such as *SimRobot* does not support *Doxygen* and generates no proper help files. In addition, the description of the robot behavior modeled with *XABSL* can be created.

GT2002 creates the code for the robots. It can be selected between *Release* (no debugging information and support), *Debug* (debugging information and support, but not via the wireless network), and *DebugWLan* (debugging information and support via the wireless network). In addition, one of the four process layouts *HU1*, *HU2*, *TUD1*, and *UB1* can be selected. When GT2002 is built, the batch file *GT.bat* is called. It selects whether *GT.bash* (employing the GreenHills compiler) or *GT.bash.gcc* (using the gcc) is executed. Therefore *GT.bat* has to be edited to switch between the two compilers.

The result of the build process can be copied to a memory stick by calling *copyfiles.bash*. Please note that this file assumes that the memory stick can be found in drive *E:*. If this is not the case, the shell script must be adjusted appropriately. Try *copyfiles.bash - -help* to see all options.

MotionConfigToolConsole creates the tool with the same name (cf. Sect. 5.4). It can only be selected between a release version with optimized code and without debugging information and a debug version. The executable will be copied to *T:\GT2002\Bin*.

RobotControl can be built for the four different process layouts. The executable will be copied to *T:\GT2002\Bin*.

Router. The router (cf. Sect. 5.3) can only be built as a debug or a release version. The result will be copied to *T:\GT2002\Bin*, together with several executables from the Open-R for Linux/CygWin release. Note that only the debug version checks for errors during the initialization of the program.

SimGT2002 can be built for the four different process layouts (as RobotControl). The executable, together with the help file, will be copied to *T:\GT2002\Bin*.

The other projects generate libraries required by one of the projects named above. There is no need to build them directly because they will be built on demand.

Appendix B

Getting Started

If you have installed the required software and the source code we suggest you to follow the introduction to GermanTeam's code given in this section. Step by step it is explained how to use the debug tool on the PC, how to play with the robots and how to let the robots play.

B.1 First Steps with RobotControl

The debug tool of the GermanTeam provides a lot of possibilities. In this section the most useful features and some impressive effects are presented. For a more detailed description see appendix J.

B.1.1 Looking at Images

Image and Segmented image. Open the *Image Viewer Dialog*, the *Color Table Dialog* and the *Color Space Dialog*. In the *Log Player Toolbar*, open a logfile, e. g. *Config\logfiles\against_cmu_1.log* and click the *Step Forward* button several times. In the bottom right corner of RobotControl the name of the current log file and the current number of the message are displayed.

All images contained in the logfile are displayed in the *Image Viewer* and in the *Color Table Dialog*. Both dialogs also show the segmented image. The *Color Table Dialog* and the *Color Space Dialog* show the colors of the YUV-cube used by the current image.

With clicks with the left and the right mouse buttons into one of the images in the *Color Table Dialog*, you can modify the color table. Clicking and dragging with the left mouse button in the *Color Space Dialog* changes the point of view. With the context menu you can select to show height maps of single channels of the image. The image viewer has two more areas to display an image. In the context menu of these areas you can choose the images to be shown there, e. g. the raw image and the segmented image again.

Results of the Image Processing. All images from the logfiles are not only displayed by the dialogs, but also put into the image processor, too. Each image contains position and rotation of

the camera relative to the body. This information was calculated by the *SensorDataProcessor* on the robot. Therefore the *SensorDataProcessor* on the PC has to be switched off for the image processor to work properly: in the *Settings Dialog* change to the setting *images from robot*. If that setting does not exist, create a new one by pressing the new button while the default setting is selected. Then change the local solution for the *SensorDataProcessor* from "Default" to "disabled".

Several debug drawings illustrate how the image processor works. With the context menu the drawings can be selected. Select "perceptCollection", "ball" and "horizon" to see the horizon line in the image and how the flags, the goals, and the ball are recognized. The color table *T:\GT2002\Config\fukuoka.c64* is adapted to the logfiles from Fukuoka. If you change something in the color table, you can see the effect on the percepts immediately. To see recognized lines, select the debug drawing "lines" and change the local solution for the *LinesPerceptor* from "disabled" to "Default".

B.1.2 Discover the Simulator

The simulator provides the same data as a real robot would do. To see this, open the *Simulator Toolbar*, the *Object Viewer* (i. e. press the button on the *Simulator Toolbar*) and the *Large Image Viewer*. In the *Simulator Toolbar* press the *start* button. In the *Image Viewer* you can see the images the simulator provides.

With the *Joint viewer Dialog*, you can visualize the joint and sensor data provided by the simulated robot. In the debug keys toolbar select *Edit table for local processes*. Select the debug key "sendJointData" and choose *Always*. Then select the debug key "sendSensorData" and choose *Always*. With the context menu in the joint viewer select "Sensors: head" and "Joints: Head". The line graphs show the values of the joints and the sensors while the robot looks to the left and to the right alternately. Then select "Actorics: legFR" and deselect the values for the head. In the *Simulator Toolbar* press the *back switch* of the robot two times for the duration of a second to start the robot. Now you can see the joint data of the front right leg while the robot moves.

B.2 Playing Soccer with the GermanTeam

B.2.1 Preparing Memory Sticks

First of all you need a complete build of GT2002, a gcc build for the use of the *Programming Memory Stick* (PMS) or a GreenHills build for the use of the *RoboCup Memory Stick*. You can change between these two possibilities by editing *T:\GT2002\Make\GT.bat*. The result of the build process will be in *T:\GT2002\Build\MS*. This directory contains all binaries and system configuration files, but not our own configuration files.

Then you may want to change the player role, the team color or the IP address, e. g. to prepare memory sticks for two different robots. You can use *RobotControl* for that: first configure you wireless network with the *WLan Toolbar* (cf. Sect. J.2.5), even if you do not plan to use it. After

that use the *Players Toolbar* (cf. Sect. J.2.6) to change the player role and the team color, and save these settings to *T:*.

Finally you will have to copy all binaries and configuration files to a memory stick. We wrote *T:\GT2002\Make\copyfiles.bash* for that reason. You may have to edit that file, because drive *E:* is assumed to be the memory stick by default[1]. If it is adapted to your needs you may call it by pressing the *copyfiles* button in *Players Toolbar* (cf. Sect. J.2.6). Then you get information about the configuration on the stick (team color, player role, WLan settings) to check whether writing was successful.

B.2.2 Establishing a WLAN Connection

After inserting a WLAN card into an AIBO, creating a memory stick as explained in section B.2.1, inserting it into an AIBO and starting that robot, you should be able to ping the robot. Its IP address was shown as the result of executing *t:/GT2002/Make/copyfiles.bash*. The IP addresses of the robot and the computer trying to ping it have to be in the same subnet, the ESSID has to be the same as well as the (usage of) encryption. Furthermore you should use APMode 0 for adhoc connections and APMode 2 when using an access point or a computer with Windows XP for pinging the robot.

If ping works for all robots you want to use, you should start the router (cf. Sect. 5.3) by executing *T:\GT2002\Bin\start.bash* with the IP addresses of the robots as parameters. This will enable you to establish a debug connection to one robot as well as the opportunity for all robots of the same team to communicate with each other. The robots use point-to-point connections for that. Each connection is visualized by a green or yellow LED, so if you want to start a complete team of four robots, each should have 3 yellow and green LEDs switched on.

In some cases the Linux inter process communication (IPC) for Cygwin is not very stable, so if you recognize strange error messages when trying to start the router, abort it. Executing *start.bash* again might help. Due to restrictions of Cygwin you will not be able to establish all connections for two complete teams including intra team communication, *RoboCup Game Manager* control and debug channels, but one and a half team should work fine.

B.2.3 Operate the Robots

Once all the robots are started, they are in the state “initial”. The red LEDs show the role of the player:

1 red LED: Goalie

2 red LEDs: Defender

3 red LEDs: Striker 1

4 red LEDs: Striker 2

[1] If you use the option *–force* you should be **absolutely sure** that *copyfiles.bash* will use the proper drive, because it will format it without any question!

Now you can start the router on the remote computer. If all the green LEDs are on, then the robot has connections to all other robots. If not, then there are problems with the wireless network.

Then you can use the *RoboCup Game Manager* by Sony or touch the back switches of the robots to set them into the "ready" state. Both top LEDs of the robot blink alternating. Note that the robots walk to their start positions on their own. If they walk around without arriving there, you can press one of the pressure sensors on the head to stop them. If the robots arrived at their start position, both top red LEDs blink synchronously. For the *striker1*, you can set whether the own team does the kickoff by using the *RoboCup Game Manager* or pressing one of the head switches for more than one second.

The game can be started with the *RoboCup Game Manager* or by touching the back switches of the robots. Both red top LEDs are on during the whole game.

The robots can be stopped again with the *RoboCup Game Manager* or by pressing their back switch for more than 1 second. They are in the "ready" state again then.

B.3 Explore the Possibilities of the Robot

In this section some nice "experiments" are described that demonstrate the possibilities of the robot. To have a more relaxed robot for these tests the we do not use the "play soccer" behavior but a more leisurely one.

First compile the robot's source code (project *GT2002*) with configuration *HU1 DebugWLan*. Then create the behavior with *make-with-another-temporary-initial-option.bat* in the folder *T:\GT2002\Src\Modules\BehaviorControl\XabslBehaviorControl*. Select the option "head-control-mode-switcher" as initial option. Use *T:\GT2002\Make\copyfiles.bash* to copy the result of the build process to a memory stick. You may want to establish a WLAN connection to the robot. Refer to section B.2.2 for this issue.

B.3.1 Send Images from the Robot and Create a Color Table

In the *Debug Keys Toolbar* type "1" in the edit box and select the button *n times*. Each time you press *Send* an image is sent from the robot. Now you can do with "real" images everything described in section B.1.1 with images from log files.

You can create a color table based on these images. Have a look at the segmented images from the logfiles from Fukuoka if *fukuoka.c64* is used. The segmented images from your field should look the same way with the color table for your field for the image processor to work correctly.

B.3.2 Try Different Head Control Modes

In the initial behavior pressing the back button of the robot switches between different head control modes:

Follow tail. If the tail is moved the head follows its motion. The joint angles of the tail are used to set the tilt and pan joints of the head. The roll joint of the head can be rotated and will stay as forced.

Stay as forced. The head can be rotated to any position and will stay as forced. The tail follows the motion of the head.

Look parallel to ground. The tilt and roll joint of the head are set such that the head always looks parallel to ground, even if the robot's body is rotated.

Search for landmarks. The robot looks to the left and to the right alternately. The tilt joint corrects a possible inclination of the body of the robot.

Search for ball. The robot scans for the ball at the top, right, bottom and left. If the ball is found it will be tracked.

B.3.2.1 Search for ball with different color tables

The robot can receive a new color table during runtime. This can be tested in the following way: Switch to the mode "search for ball" so that the robot tracks the ball. In the *Color Table Dialog*, clear the orange channel and use the *send* button to send the new color table to the robot. Then the robot can not see the ball anymore and starts looking around to find it. Press *undo* and *send* to send the old color table—the robot finds the ball again and keeps looking at it.

B.3.2.2 Pay Attention to the Horizon when the Robot is Lifted from Ground and Rotated

The position of the horizon in the image depends on the rotation of the robot and the rotation of the head. Request images from the robot every 500 ms with the *Debug Keys Toolbar* to see this. The images can be shown with the *Large Image Viewer Dialog*. Select the debug drawing "horizon" in the context menu. In the *Settings Dialog*, change the local solution of the *Sensor-DataProcessor* from "Default" to "disabled".

Now look at the images and the horizon when rotating the robot and the head. Attention! When you rotate the robot it might execute a getup motion. To avoid this you can disable the *BehaviorControl* on the robot with the *Settings Dialog*. You can not switch between different head control modes until the solution for the behavior is switched to "XabslBehaviorControl" again.

B.3.3 Watch Different Walking Styles

With the Settings dialog you can switch between different walking engines during runtime. To compare the walking styles you can produce motion requests with the *Motion Tester Dialog* or with the *Joystick Motion Tester Dialog*. As the BehaviorControl also produces motion requests you have to disable it, i. e. select the setting "motion tester".

You can also create motion requests using the head of the robot as an input device. If you select the setting "walk demo" the tilt angle of the robot's head controls the forward/backward

speed, the roll angle the left/right speed and the pan angle the rotation speed. If you press the back button, the robot looks straight ahead and stands still.

B.3.4 Create Own Kicks

To create your own kicks use the *Mof Tester Dialog* and the "create kicks" setting which activates the "Debug" solution of the *MotionControl*. Now you can move the legs of the robot to a desired position and they will stay as forced. If you press *Read* the sensor data of each joint is read and transmitted to the dialog. Now move the legs to the next position and press *Read* again. In this way you can record the whole kick or an other motion step by step. With *Execute* you can execute the motion. For more details study section J.5.4.

B.3.5 Test simple behaviors

To test some simple behaviors, all modules have to work with the default solution. If you did some experiments before, change to the "default" setting in the *Settings Dialog*.

Open the *Xabsl Behavior Tester Dialog* and select "Test on robot". Now you can see the option activation path, the current skill with its parameters, the current motion request and the current values of the selected input symbols. If you move the ball in front of the robot you can see how the value for "ball.distance" changes.

B.3.5.1 Test Skills

Skills are the basic components of the behavior, here is a closer look at two of them. With the *Xabsl Behavior Tester Dialog*, you can test a skill with the desired parameters. Select the check box "Test a skill" in the lower part of the dialog. In the edit boxes you can type in the parameters for the skill.

To test the skill that goes to the ball select "go-to-ball" in the combo box. In the edit box for the first parameter type for example "300" and press the button *Send request*. If the robot can see the ball it goes there and stops at a distance of 300 mm (center of ball to point beneath the pan joint of the head). Now try different distances and see how the robot reacts. You can also try to change the third parameter of this skill - the maximal speed when going to the ball (mm/s).

With the skill "go-to-point" you can see if the self-localization works. The first to parameters specify the x and the y position of the robot on the field. The unit of measurement is millimeter, the origin of the system of coordinates is the center of the field. The x-axis points to the opponent goal. The third parameter specifies the desired angle of the robot at the destination point. With the other combo boxes in the *Xabsl Behavior Tester Dialog* you can set the output symbol for the "head-control-mode" to different values, for example "search-for-landmarks", search for ball or "look-at-visible-landmarks" and study the effect to the accuracy of the localization.

B.3.5.2 Test Options

Options represent the higher level of the behavior. They do not have parameters. You can select different options with the combo box at the top of the *Xabsl Behavior Tester Dialog*. Be sure that the check box “Test a skill” is not selected.

The option “go-to-ball-and-kick” is one of the most important options in the play-soccer behavior. If you select this option you can perform the following tests. Place the ball behind the robot. The robot will turn until it sees the ball then go there and kick to the direction of the opponent goal. Put the ball somewhere on the field. The robot will go to the ball. When it has almost reached the ball hold a hind leg of the robot to simulate a blocking robot. After a few seconds the robot will rotate its legs to free from the obstacle. It is your choice if you release the robot as a consequence of this motion.

If you select the option “striker2-search-for-ball” the robot will visit a left and a right point at the center line alternately. This behavior is used during a game by the second striker to find the ball.

Appendix C

Processes, Senders, and Receivers

C.1 Motivation

In GT2001, there exist two kinds of communication between processes: on the one hand, Aperios queues are used to communicate with the operating system, on the other hand, a shared memory is employed to exchange data between the processes of the control program. In addition, Aperios messages are used to distribute the address of the shared memory. All processes use a structure that is predefined by Sony's stub generator. This approach lacks of a simple concept how to exchange data in a safe and coordinated way. The resulting code is confusing and much more complicated then it should be.

However, the internal communication using a shared memory also has its drawbacks. First of all, it is not compatible with the new ability of Aperios to exchange data between processes via a wireless network. In addition, the locking mechanism employed may waste a lot of computing power. However, the locking approach only guarantees consistence during a single access, the entries in the shared memory can change from one access to another. Therefore, an additional scheme has to be implemented, as, e. g., making copies of all entries in the shared memory at the beginning of a certain calculation step to keep them consistent.

The communication scheme introduced in GT2002 addresses these issues. It uses Aperios queues to communicate between processes, and therefore it also works via the wireless network. In the approach, no difference exists between inter-process communication and exchanging data with the operating system. Three lines of code are sufficient to establish a communication link. A predefined scheme separates the processing time into two communication phases and a calculation phase.

C.2 Creating a Process

Any new process has to be part of a special *process layout*. Process layouts group together different processes that make up a robot control program, and they are stored in subdirectories under *T:\GT2002\Src\Processes*. Currently, process layouts are named after the universities who used them, e. g. *HU1* for the first layout of the Humboldt-Universität zu Berlin. An Aperios process

is allowed to have a name with a maximum length of eight characters. To create a new process, one has to think such a name up and

- insert a new line into *T:\GT2002\Src\Processes*ProcessLayout*\object.cfg*, following the format */MS/OPEN-R/APP/OBJS/*name.*bin*,
- for the GreenHills environment, create new *object* and *download* lines in *T:\GT2002\Src\Processes*ProcessLayout\ProcessLayout.*mcf*,
- for the Open-R SDK environment, create a new *object* line in *T:\GT2002\Src\Processes*ProcessLayout\ProcessLayout.*ocf*,
- create a new subdirectory under *T:\GT2002\Src\Processes*ProcessLayout,
- create a *.cpp* file in this new directory with the same name,
- and, although not required, create two new folders in the GT2002 project both under *GT2002\Processes*ProcessLayout and *RobotControl\SharedCode\Processes*ProcessLayout in the Microsoft Developer Studio, inserting the new source file there. Note that all folders must be named differently in the Developer Studio, so you may have to extend the process layout name at the end of your new folder name.

The new source file must include “Tools/Process.h”, derive a new class from *class Process*, implement at least the function *Process::main()*, and must instantiate the new class with the macro *MAKE_PROCESS*. As an example, look at this little process[1]:

```
#include "Tools/Process.h"

class Example : public Process
{
  public:
    virtual int main()
    {
      printf("Hello World!\n");
      return 0;
    }
};

MAKE_PROCESS(Example);
```

The process will print “Hello World” once. If the function *main()* should be recalled after a certain period of time, it must return the number of milliseconds to wait, e. g.

[1]Note that the examples given here will not compile, because the debugging support required by *class Process* is missing. One can derive from *class PlatformProcess* instead, naming the *main*-function *processMain*.

```
return 500;
```

to restart *main()* after 500 ms. However, if *main()* itself requires 100 ms of processing time, and then pauses for 500 ms before it is recalled, it will in fact be called every 600 ms. If this is not desired, *main()* must return a negative number. For instance,

```
return -500;
```

will ensure a cycle time of 500 ms, as long as *main()* itself does not require more than this amount of time.

Note that if main returns 0, it will only be recalled if there is at least one blocking receiver or at least one active blocking receiver (cf. next section). Otherwise, the process will be inactive until the robot will be rebooted.

C.3 Communication

The inter-object communication is performed by *sender*s and *receiver*s exchanging *packages*. Packages are normal C++ classes that must be *streamable* (cf. the technical note on streams in appendix D). A sender contains one instance of a package and will automatically transfer it to a receiver after the receiver requested it and the sender's member function *send()* was called. The receiver also contains an instance of a package. Each data exchange will be performed after the function *main()* of a process has terminated, or immediately when the function *send()* is called. A receiver obtains a package before the function *main()* starts and will request for the next package after *main()* was finished. Both senders and receivers can either be blocking or non-blocking objects. The function *main()* will wait for all blocking objects before it starts, i. e. it waits for blocking receivers to acquire new packages, and for blocking senders to be asked to send new packages.[2] *main()* will not wait for non-blocking objects, so it is possible that a receiver contains the same package for more then one call of *main()*.

C.3.1 Packages

A package is an instance of a class that is streamable, i. e. that implements the $<<$ and $>>$ operators for the classes *Out* and *In*, respectively. So, an example of a package is

```
class NumberPackage
{
  public:
    int number;
    NumberPackage() {number = 0;}
};
```

[2] Note that under RobotControl, a process will be continued when a single blocking event occurs. This is currently required to support debug queues.

```
Out& operator<<(Out& stream,const NumberPackage& package)
{
  return stream << package.number;
}

In& operator>>(In& stream, NumberPackage& package)
{
  return stream >> package.number;
}
```

Note also that it is a good idea to provide a public default constructor.
A special case of packages are Open-R packages:

- Packages that are received from the operating system must provide a streaming operator that reads exactly the format as provided by Open-R. The packages are all defined in *<OPENR/ODataFormats.h>*. However, the data types provided there do not reflect the real size of the objects, they are only headers. Therefore, new types must be declared that have the real size of the Open-R packages. This size can be determined from their *vectorInfo.totalSize* member variable. The size is constant for each type, but it may vary between different versions of Open-R. Such data types are only required to implement the streaming operators, they are not needed elsewhere.
- Packages that are sent to the operating system require special allocation operators. Therefore, special senders (cf. next section) were implemented that allocate memory using the appropriate methods, and then use these memory blocks for the communication with the operating system.

C.3.2 Senders

Senders send packages to other processes. A process containing a sender for *NumberPackage* could look like this:

```
#include "Tools/Process.h"

class Example1 : public Process
{
  private:
    SENDER(NumberPackage);

  public:
    Example1() :
      INIT_SENDER(NumberPackage,false) {}

    virtual int main()
    {
```

```
      ++theNumberPackageSender.number;
      theNumberPackageSender.send();
      return 100;
    }
};

MAKE_PROCESS(Example1);
```

The macro *SENDER* defines a sender for a package of type *NumberPackage*. As the second argument is false, it is a non-blocking sender. Macros as *SENDER* and *RECEIVER* will always create a variable that is derived from the provided type (in this case *NumberPackage*) and that has a name of the form *the*Type*Sender* or *the*Type*Receiver*, respectively (e. g. *theNumberPackageSender*).

Packages must always explicitly be sent by calling the member function *send()*. *send()* marks the package as to be sent and will immediately send it to all receivers that have requested a package. However, each time the function *main()* has terminated, the package will be sent to all receivers that have requested it later and have not got it yet. Note that the package that will be sent has not necessarily the state it had when calling *send()*. As packages are not buffered, always the actual content of a package will be transmitted, even if it changed since the last call to *send()*.

As the communication follows a real-time approach, it is possible that a receiver misses a package if a new package is sent before the receiver has requested the previous one. The approach follows the idea that all receivers usually want to receive the most actual packages. The only possibility to ensure that a receiver will get a package is to only send it, when it already has been requested. This can be realized by either using a blocking sender, or by checking whether the sender has been requested to send a new package: *theNumberPackageSender.requestedNew()* provides this information. Note: a sender can provide a package to more than one receiver. *requestedNew()* returns true if at least one receiver requested a new package. This is different from a blocking sender: a blocking sender will wait for *all* receivers to request a new package!

C.3.3 Receivers

Receivers receive packages sent by senders. A process that reads the package provided by *Example1* could look like this:

```
#include "Tools/Process.h"

class Example2 : public Process
{
  private:
    RECEIVER(NumberPackage);

  public:
    Example2() :
      INIT_RECEIVER(NumberPackage,true) {}
```

```
    virtual int main()
    {
      printf("Number %d\n",theNumberPackageReceiver.number);
      return 0;
    }
};

MAKE_PROCESS(Example2);
```

Here, the function *main()* will wait for the *RECEIVER* (i. e. the second parameter is *true*), so it will always print out a new number.

However, one thing is missing: Aperios has to know which process wants to transfer packages to which other process. Therefore, the file *connect.cfg* has to be extended by the following line:

```
Example1.Sender.NumberPackage.S Example2.Receiver.NumberPackage.O
```

If more than one receiver is used in a process, the non-blocking receivers shall be defined first. Otherwise, the packages of the non-blocking receivers may be older than the packages of the blocking receivers. To determine whether a non-blocking receiver got a new package, call its member function *receivedNew()*.

Appendix D

Streams

D.1 Motivation

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

There are two reasons not to use the C++ iostreams library for this purpose: on the one hand, it does not guarantee that the data is streamed in a way that it can be read back without any special handling, especially when streaming into and from text files. On the other hand, the iostreams library is not fully implemented on all platforms, namely not on Aperios.

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

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

D.2 The Classes Provided

Currently, the following classes are implemented:

PhysicalOutStream. Abstract class

OutFile. Writing into files

OutMemory. Writing into memory

OutSize. Determine memory size for storage

StreamWriter. Abstract class

OutBinary. Formats data binary

OutText. Formats data as text

Out. Abstract class

OutStream<PhysicalOutStream,StreamWriter>. Abstract template class

OutBinaryFile. Writing into binary files
OutTextFile. Writing into text files
OutBinaryMemory. Writing binary into memory
OutTextMemory. Writing into memory as text
OutBinarySize. Determine memory size for binary storage
OutTextSize. Determine memory size for text storage

PhysicalInStream. Abstract class

InFile. Reading from files

InMemory. Reading from memory

StreamReader. Abstract class

InBinary. Binary reading

InText. Reading data as text

InConfig. Reading configuration file data from streams

In. Abstract class

InStream<PhysicalInStream,StreamReader>. Abstract class template

InBinaryFile. Reading from binary files
InTextFile. Reading from text files
InConfigFile. Reading from configuration files
InBinaryMemory. Reading binary data from memory
InTextMemory. Reading text data from memory
InConfigMemory. Reading config-file-style text data from memory

D.3 Streaming Data

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

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

The file will be written into the configuration directory, e. g. *T:\GT2002\Config\MyFile.txt* on the PC. It will look like this:

```
1 3.14000 Hello\ Dolly
42
```

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

```
#include "Tools/Streams/InStreams.h"
// ...
InTextFile stream("MyFile.txt");
int a,d;
double b;
char c[12];
stream >> a >> b >> c >> d;
```

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

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

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

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

void read(In& stream)
{
  int a,d;
  double b;
  char c[12];
```

```
  stream >> a >> b >> c >> d;
}
// ...
OutTextFile stream("MyFile.txt");
write(stream);
// ...
InTextFile stream("MyFile.txt");
read(stream);
```

D.4 Making Classes Streamable

Streaming is only useful if as many classes as possible are streamable, i. e. they implement the streaming operators $<<$ and $>>$. The purpose of these operators is to write the current state of an object into a stream, or to reconstruct an object from a stream. As the current state of an object is stored in its member variables, these have to be written and restored, respectively. This task is simple if the member variables themselves are already streamable.

D.4.1 Streaming Operators

As the operators $<<$ and $>>$ cannot be members of the class that shall be streamed (because their first parameter must be a stream), it must be distinguished between two different cases: In the first case, all relevant member variables of the class are public. Then, implementing the streaming operators is straightforward:

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

class Sample
{
  public:
    int a,b,c,d;
};

Out& operator<<(Out& stream,const Sample& sample)
{
  return stream << sample.a << sample.b
                << sample.c << sample.d;
}

In& operator>>(In& stream,Sample& sample)
{
  return stream >> sample.a >> sample.b
                >> sample.c >> sample.d;
}
```

However, if the member variables are private, the streaming operators must be friends of the class. This can be a little bit complicated, because some compilers require the function prototypes to be already declared when they parse the *friend* declarations:

```
class Sample;
Out& operator<<(Out&,const Sample&);
In& operator>>(In&,Sample&);

class Sample
{
  private:
    int a,b,c,d;
  friend Out& operator<<(Out&,const Sample&);
  friend In& operator>>(In&,Sample&);
};
// ...
```

Another possibility to avoid these additional declarations would be to define public member functions that perform the streaming and that are called from the streaming operators. However, this would not be shorter.

If dynamic data should be streamed, the implementation of the operator $>>$ requires a little bit more attention, because it always has to replace the data already stored in an object, and thus if this is dynamic, it has to be freed to avoid memory leaks.

```
class Sample
{
  public:
    char* string;
    Sample() {string = 0;}
};

Out& operator<<(Out& stream,const Sample& sample)
{
  if(sample.string)
    return stream << strlen(sample.string) << sample.string;
  else
    return stream << 0;
}

In& operator>>(In& stream,Sample& sample)
{
  if(sample.string)
    delete[] sample.string;
  int len;
```

```
  stream >> len;
  if(len)
  {
    sample.string = new char[len+1];
    return stream >> sample.string;
  }
  else
  {
    sample.string = 0;
    return stream;
  }
}
```

D.4.2 Streaming using *read()* and *write()*

There also is a second possibility to stream an object, i. e. using the functions Out::write() and In::read() that write a memory block into, or extract one from a stream, respectively:

```
class Sample
{
  public:
    int a,b,c,d;
};

Out& operator<<(Out& stream,const Sample& sample)
{
  stream.write(sample,sizeof(Sample));
  return stream;
}

In& operator>>(In& stream,Sample& sample)
{
  stream.read(sample,sizeof(Sample));
  return stream;
}
```

This approach has its pros and cons. On the one hand, the implementations of the streaming operators need not to be changed if member variables in the streamed class are added or removed. On the other hand, this approach does not work for dynamic members. It will corrupt pointers to virtual method tables if classes or their base classes contain virtual functions. Last but not least, the structure of an object is lost (not the data) when it is streamed to a text file, because in the file, it will look like a memory dump, which is not well readable for humans.

D.5 Implementing New Streams

Implementing a new stream is simple. If needed, a new medium can be defined by deriving new classes from *PhysicalInStream* and *PhysicalOutStream*. A new format can be introduced by deriving from *StreamWriter* and *StreamReader*. Streams that store data must be derived from class *OutStream*, giving a *PhysicalOutStream* and a *StreamWriter* derivate as template parameters, reading streams have to be derived from class *InStream*, giving a *PhysicalInStream* and a *StreamReader* derivate as template parameters.

As a simple example, the implementation of *OutBinarySize* is given here. The purpose of this stream is to determine the number of bytes that would be necessary to store the data inserted in binary format, instead of actually writing the data somewhere. For the sake of shortness, the comments are removed here.

```
class OutSize : public PhysicalOutStream
{
  private:
    unsigned size;
  public:
    void reset() {size = 0;}
    OutSize() {reset();}
    unsigned getSize() const {return size;}
  protected:
    virtual void writeToStream(const void*,int s) {size += s;}
};

class OutBinary : public StreamWriter
{
  protected:
    virtual void writeChar(char d,PhysicalOutStream& stream)
      {stream.writeToStream(&d,sizeof(d));}

    virtual void writeUChar(unsigned char d,
                            PhysicalOutStream& stream)
      {stream.writeToStream(&d,sizeof(d));}

    virtual void writeShort(short d,PhysicalOutStream& stream)
      {stream.writeToStream(&d,sizeof(d));}

    virtual void writeUShort(unsigned short d,
                             PhysicalOutStream& stream)
      {stream.writeToStream(&d,sizeof(d));}

    virtual void writeInt(int d,PhysicalOutStream& stream)
      {stream.writeToStream(&d,sizeof(d));}
```

```
    virtual void writeUInt(unsigned int d,
                           PhysicalOutStream& stream)
      {stream.writeToStream(&d,sizeof(d));}

    virtual void writeLong(long d,PhysicalOutStream& stream)
      {stream.writeToStream(&d,sizeof(d));}

    virtual void writeULong(unsigned long d,
                            PhysicalOutStream& stream)
      {stream.writeToStream(&d,sizeof(d));}

    virtual void writeFloat(float d,PhysicalOutStream& stream)
      {stream.writeToStream(&d,sizeof(d));}

    virtual void writeDouble(double d,
                             PhysicalOutStream& stream)
      {stream.writeToStream(&d,sizeof(d));}

    virtual void writeString(const char *d,
                             PhysicalOutStream& stream)
    {
      int size = strlen(d);
      stream.writeToStream(&size,sizeof(size));
      stream.writeToStream(d,size);
    }

    virtual void writeEndL(PhysicalOutStream& stream) {};

    virtual void writeData(const void* p,int size,
                           PhysicalOutStream& stream)
      {stream.writeToStream(p,size);}
};

class OutBinarySize : public OutStream<OutSize,OutBinary>
{
  public:
    OutBinarySize() {}
};
```

Appendix E

Debugging Mechanisms

The software architecture of the programs and tools developed contains a rich variety of debugging mechanisms, which are described in detail in the following sections.

E.1 Message Queues

Besides the package-oriented inter-object communication with senders and receivers (cf. Sect. C.3), *message queues* are used for the transport of debug messages between processes, platforms, and applications. They consist of a list of *messages*, which are stored and read using streams (cf. App. D).

Writing data to Message Queues. As almost all data types have streaming operators, it is very easy to store them in message queues. When a message is written to a queue, a *message id* is added to identify the type of the data. In addition, a time stamp is stored for every new message. A typical piece of code looks like this:

```
#include "Tools/MessageQueue/MessageQueue.h"
// ...
Image myImage;
myMessageQueue.out.bin << myImage;
myMessageQueue.out.finishMessage(idImage);

int numOfBalls = 3;
myMessageQueue.out.text << "found " << numOfBalls << " balls."
MyMessageQueue.out.finishMessage(idText);

int a, b, c, d;

myMessageQueue.out.bin << a << b;
myMessageQueue.out.bin << c << d;
myMessageQueue.out.bin.finishMessage(id4FunnyNumbers);
```

First an image is written in binary format to the queue. The type of the message is *idImage*. Then the text *"found 3 balls"* is written in text format to the queue. The id *idText* marks the message as unstructured text. At last, four integer values are written to the queue as a message of the type *id4FunnyNumbers*.

Transmitting Message Queues. Message queues are exchanged between processes such as all other packages. They are also used for the transmission of debug messages via the wireless network. They can be written to and read from logfiles. Messages can be moved to other queues using

```
myQueue.copyAllMessages(otherQueue);
myQueue.moveAllMessages(otherQueue);

message >> otherQueue;
```

copyAllMessages copies and *moveAllMessages* moves all messages to another queue. The third statement shows how a single message can be copied.

Distribution of Debug Messages. As all messages can be written in any order into a queue, a special mechanism for distributing the message is needed. A *message handler* does this job, e. g.:

```
class MyMessageHandler : public MessageHandler
{
  virtual bool handleMessage(InMessage& message);
};
// ...
bool MyMessageHandler::handleMessage(InMessage& message)
{
  switch (message.getMessageID()
  {
  case idImage:
    message.bin >> myImage;
    return true;
  case idText:
    message >> otherQueue;
    return true;
  case id4FunnyNumbers;
    message.bin >> a >> b >> c >> d;
    return true;
  default:
    return false;
  }
```

```
}
// ...
MyMessageHandler handler;
myQueue.handleAllMessages(handler);
```

The class *MyMessageHandler* is derived from *MessageHandler*. In the implementation, for every *message id* the data is read differently from a queue. Whereas the image is just streamed to a local member variable, the text message is copied to another queue.

Instantiating Message Queues. The class *MessageQueue* has two different mechanisms to store the data into memory. On Windows and Linux platforms, a dynamic list is used. But on the Open-R platform, dynamic memory allocations are expensive. Therefore on that platform the messages are stored in a static memory buffer.

If the code containing a *MessageQueue* shall run on the Open-R platform, the size of the queue in bytes has to be set using

```
MessageQueue queue;
queue.setSize(1000000); // ignored on Win32 and Linux
```

Message Queues and Processes. The class *Process*, which is the base class for all processes, already has the members *debugIn* for incoming and *debugOut* for outgoing debug messages. In addition, *Process* is derived from *MessageHandler* so that every process can distribute debug messages.

E.2 Debug Keys

The tools RobotControl and SimGT2002 (cf. Sect. 5) can process a wide range of messages from physical or simulated robots. As it is not possible to send all the messages at once, *debug keys* are used to toggle the output of these messages. They are also transmitted to the robot using message queues. In *T:\GT2002\Src\Tools\Debugging\DebugKeyTable.h* a variety of keys is declared.

Each key can have one of these states:

Disabled. No output is sent for the key.

Send always. The output is sent always.

Send n times. The output is sent n times.

Send every n times. The output is sent every n times.

Send every n ms. The output is sent every n milliseconds.

The class *DebugKeyTable* has a member *isDebugKeyActive()* that determines if the message shall be sent dependent on the state of a given key, e. g.:

```
if (myDebugKeyTable.isDebugKeyActive(sendImages))
{
  myQueue.out.bin << image;
  myQueue.out.finishMessage(idImage);
}
```

E.3 Debug Macros

To simplify the access to outgoing message queues and to the right debug key table, in *T:\GT2002\Src\Tools\Debugging\Debugging.h* two macros are defined:

OUTPUT(type, format, data) stores *data* with *format* and message type *type* in the outgoing queue of the process.

INFO(key, type, format, data) works similar to *OUTPUT*, but only, when the debug key *key* is active.

Example:

```
OUTPUT(idText,text,"Hello World");
OUTPUT(idText,text,"Found " << numberOfBalls << " balls.");
OUTPUT(idSensorData,bin,mySensorData);
INFO(sendImage,idImage,bin,myImage);
```

Both macros are not expanded in *Release* configurations to save processing time.

E.4 Stopwatch

To track down waste of time in the code, in *T:\GT2002\Src\Tools\Stopwatch.h* two macros are defined:

STOP_TIME(expression) measures the system time before and after the execution of *expression* and outputs the difference as a text.

STOP_TIME_ON_REQUEST(eventID, expression). If the time keeping is requested for the *eventID*, the time is measured before and after the execution of *expression* and sent as an debug message with the id *idStopwatch*. With the time diagram dialog (cf. Sect. J.3.4) the requests can be generated and the results of the measurements are displayed.

Example:

```
STOP_TIME_ON_REQUEST(imageProcessor,
  pImageProcessor->execute(theImageReceiver, blobCollection);
);

STOP_TIME(
  for(int i = 0; i < 1000000; i++)
  {
    double x = sqrt(i);
    x *= x;
  }
);
```

E.5 Debug Drawings

At each place in the code a debug drawing can be drawn and sent. There exist two types of drawings: *imageDrawings* are in pixel coordinates and will be displayed in the image viewer (cf. Sect. J.3.1), whereas *fieldDrawings* are in the system of coordinates of the field and will be shown in the field view and the radar viewer (cf. Sect. J.3.2). To generate a *DebugDrawing* the following has to be done:

- *Tools/Debugging/DebugDrawing.h* has to be included in the file from which it will be drawn.
- In *T:\GT2002\Src\Tools\Debugging\DebugDrawing.h* a new drawingID has to be added to one of the enumeration types *FieldDrawing* or *ImageDrawing*. In the method *getDrawingName()*, a string representation for the new drawingID has to be given. In the method *getDebugKeyID()*, a debug key for requesting the drawing has to be added. This debug key has to be defined in *T:\GT2002\Src\Tools\Debugging\DebugKeyTable.h*.
- In the file that should draw, the following has to be added, e. g. to create a drawing called “sketch”:

```
// create the drawing
CREATE_DEBUGDRAWING(sketch);

// ...

// paint to the drawing
PAINT_TO_DEBUGDRAWING(sketch,
  sketchDrawing.line(
    (int)low.x, (int)low.y,
    (int)high.x, (int)high.y,
    DebugDrawing::ps_solid, 0,
```

```
      DebugDrawing::Color(127,127,127)
    );

    sketchDrawing.dot((int)low.x, (int)low.y,
      DebugDrawing::Color(255,255,255),
      DebugDrawing::Color(0,0,0) );
      sketchDrawing.dot((int)high.x, (int)high.y,
      DebugDrawing::Color(0,0,0),
      DebugDrawing::Color(255,255,255)
    );
  );

  // ...

  // send the debug drawing
  SEND_DEBUGDRAWING(sketch);
```

If the code runs on the PC, all debug drawings are created automatically. If the code runs on the robot, a debug drawing is only painted if the corresponding request is sent.

E.6 Modules and Solutions

To obtain solutions for the modules that can be exchanged during runtime, there is a base class for each module, e. g. the class *ImageProcessor*. All solutions for this module (e. g. *BlobImageProcessor* and *GridImageProcessor*) are derived from this base class. Each module has its own *ModuleSelector* class, e. g. the *ImageProcessorSelector*. An instance of the selector class is created in the process the module is part of. The selector class creates instances of all solutions of the module during construction, but executes only one of the solutions during runtime.

The data structure *SolutionRequest* stores what the current solution for each module is. This data structure can be modified on the PC and sent to the robot (cf. Sect. J.2.3). Each process contains a *SolutionHandler* which receives the solution requests. Each module selector class has a reference to the solution handler, and thus it can decide, which of the solution has to be executed.

To add a new module, the base class, the selector class based on the template *TModuleSelector*, and the different classes for the solutions that derive from the base class have to be created. In the selector class all solutions have to be added. In *T:\GT2002\Src\Tools\SolutionRequest.h* the new module and the solutions have to be added to the enum data types and to the functions providing names for the modules and solutions. In one of the available processes, an instance of the selector class has to be instantiated, and its *execute()* method has to be called.

Appendix F

XABSL Language Reference

This chapter describes the syntax and semantics of the *extensible agent behavior specification language* (XABSL). For an introduction to XABSL see section 3.6.2, in which a simplified goalie behavior is used that was developed by *Humboldt 2002* for the German Open 2002.

F.1 General Structure of an XABSL Document

As already mentioned, the syntax of the XABSL Language is specified in XML schema (In a the code release of the GermanTeam file */Src/Modules/XabslBehaviorControl/xabsl/xabsl-1.0.xsd* contains this definition.). The Schema exports the namespace *xabsl* (*http://www.ki.informatik.hu-berlin.de/XABSL*). As the existing XSL stylesheets are not independent of the name space used in instance documents up to now, no name space prefix should be used for elements in instance documents. The root element of a document is *agent*. Documents have the following code structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<xabsl:agent xmlns ="http://www.ki.informatik.hu-berlin.de/XABSL"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="xabsl/xabsl-1.0.xsd">
  <description>
    ...
  </description>
  <environment>
    ...
  </environment>
  <options initial-option="...">
    ...
  </options>
  <skills>
    ...
  </skills>
</agent>
```

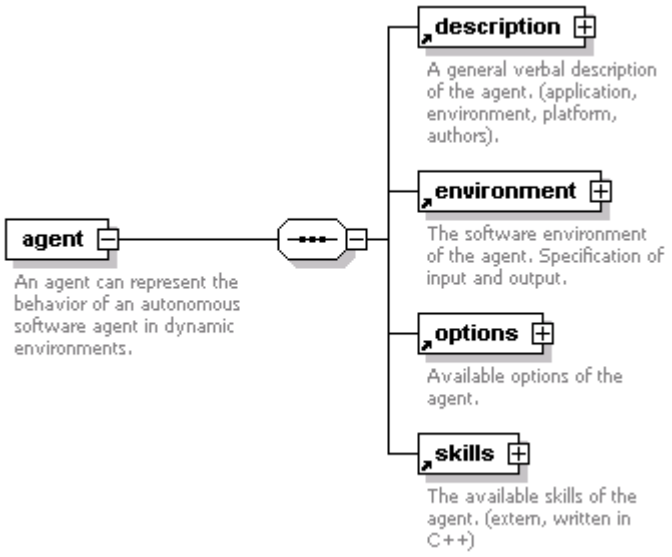


Figure F.1: The Syntax of the Root Element *agent*.

The root element *agent* has four required elements: *description*, *environment*, *options*, and *skills*. (cf. Fig. F.1). They will be described in the following sections.

Note that not all attributes and identity constraints will be mentioned (please refer to the schema file for additional information).

F.2 The Element *description*

That element *description* contains very general information about the agent (cf. Fig. F.2). It is only used for the documentation that can be generated from a XABSL document.

title contains a general project name. This is used for the headline of the generated documentation.

application is a verbal description of the application that the agent was designed for.

platform describes the robot or the computer system the agent runs on.

software-environment contains a description or name of the software system, project or environment, in that the formalized behavior is embedded.

author names one author of the agent. Several author elements can be used.

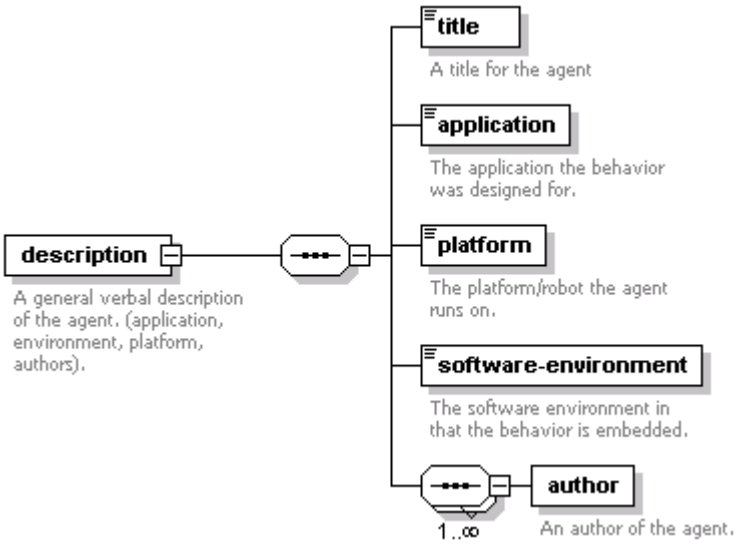


Figure F.2: The element *description.*

F.3 The Element *environment*

To be independent of specific software environments and platforms, the formalization of the interaction with the software environment is also included in XABSL by defining symbols. Interaction means access to input functions and variables (as, e. g., of the world state), to output functions (e.g. to set requests for other parts of the information processing).

The decisions that are made in the decision trees of the state machines depend almost exclusively on *input symbols*. For every single part of represented knowledge about the world, every advanced analysis function, and every single state of the agent that is used for decision making, an input symbol has to be defined. From the robot soccer example:

```
<decimal-input-symbol name="ball.x"
  description="absolute x position of the ball" measure="mm"/>
<decimal-input-symbol name="ball.distance"
  description="distance to ball" measure="mm"/>
<boolean-input-symbol name="dribble-is-useful"
  description="Determines if the dribble skill can be applied."/>
```

The symbol "*ball.x*" stands for a variable that contains the absolute x coordinate of the ball position in the world state. "*ball.distance*" stands for a simple function that calculates the distance to the ball from the absolute ball coordinates. "*dribble-is-useful*" stands for a complex analysis function that determines if a single skill can be applied.

Although all these symbols stand for very different structural entities, they are all treated alike. In XABSL it is not specified whether a symbol stands for a function or a simple variable.

There are three different types of input symbols:

decimal-input-symbol. Symbols for floating point or integer input functions and variables.

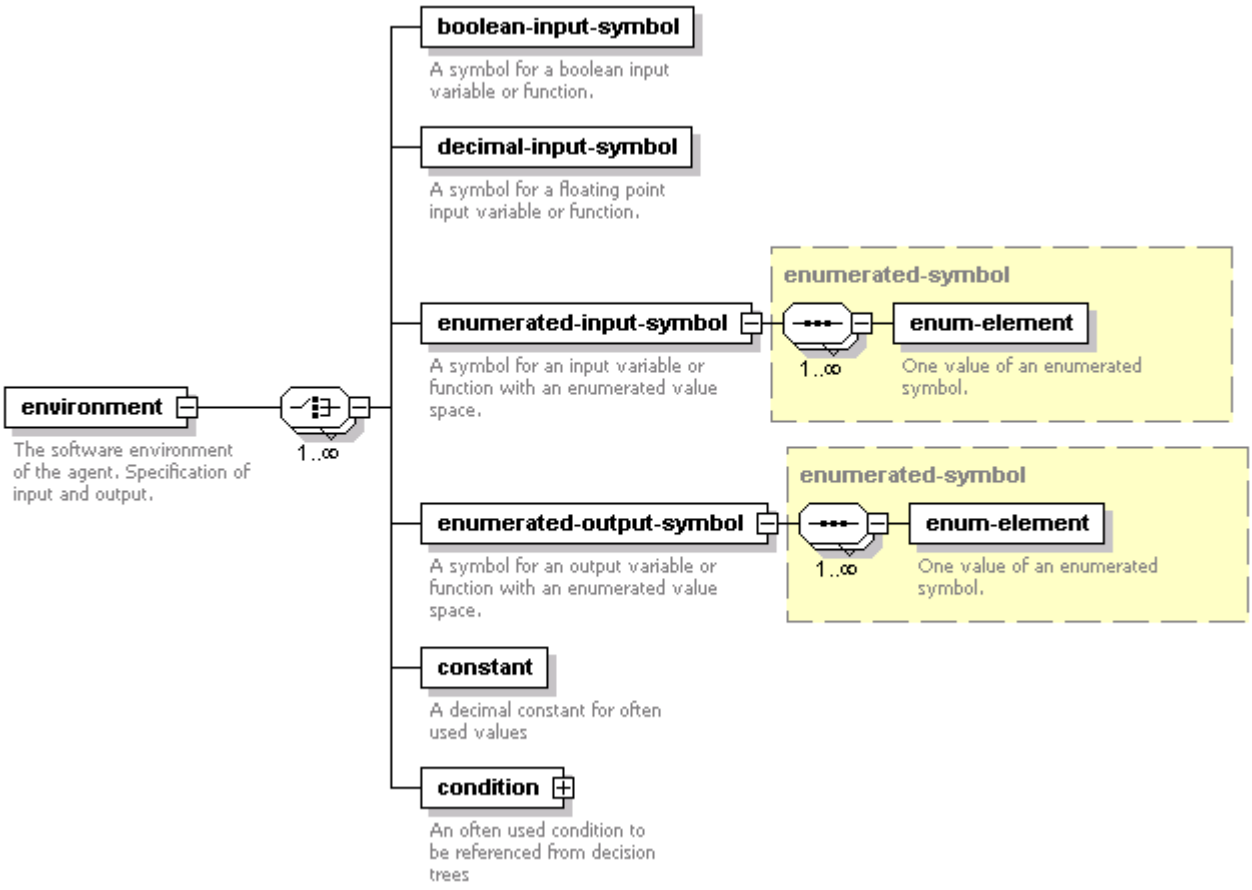


Figure F.3: The element *environment*

boolean-input-symbol. Symbols for boolean input functions and variables.

enumerated-input-symbol. Symbols for enumerated input functions and variables (as enums in C++).

All the input symbols have the required attribute "*name*" as a name of the symbol and "*description*" that contains a verbal description of the symbol. Decimal input symbols have the required attributes "*range*" for the range that values can reach and "*measure*" for the measure of the symbol. Enumerated input symbols contain "*enum-element*" elements for every enumeration element.

As already mentioned, there are also output symbols. These symbols are not used for decision making but for setting states in the remaining information processing. Output symbols are set from states inside options. Strictly speaking, if a state sets an output symbol to a specific value, then the symbol is always set to the value when the state is active.

Up to now, there is only an enumerated version of output symbols: "*enumerated-output-symbol*" stands for an enumerated output function or variable. It has the same attributes as "*enumerated-input-symbol*". From the robot soccer example:

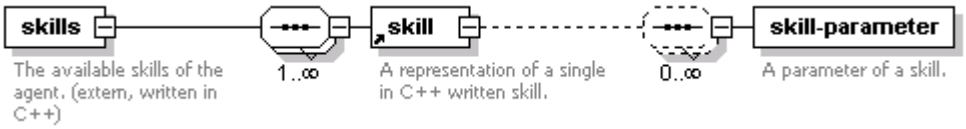


Figure F.4: The element skills

```
<enumerated-output-symbol name="head-control-mode"
   description="a mode for the head control."
   required="true" allow-overwrite="true">
  <enum-element name="search-for-ball"/>
  <enum-element name="search-for-landmarks"/>
  <enum-element name="look-left"/>
  <enum-element name="look-right"/>
</enumerated-output-symbol>
```

The symbol “*head-control-mode*” stands for a request, where the Sony robot shall direct the head (that contains the camera of the robot) to. In options/states, in which it is necessary only to look at the ball, the symbol is set to “*search-for-ball*”. When good localization is necessary, “*search-for-landmarks*” is set. The attributes “*required*” and “*allow-overwrite*” are ignored up to now.

”*constant*” defines a constant decimal value. This is useful for often used parameters of conditions or skills. Again in the robot soccer example the constant

```
<constant name="goalie.home-position.x"
  description="The x home position of the goalie"
  measure="mm" value="-2000"/>
```

is used as the x position of the center of the own goal. If such values are defined only once and referenced from other parts of the document, these values can be changed very easily.

At last, in “*environment*”, “*condition*” elements can be inserted. These are predefined conditions that can be referenced from decision trees inside states. It is useful to define conditions used often here to keep the document short. For a detailed description see section F.8.

All the elements of “*environment*” can appear in any number and order.

F.4 The Element *skills*

Skills are the basic abilities of the agent. Programmers may want to use advanced algorithms or programming language concepts so that skills are not specified in XABSL but written in C++. Therefore, there an interface has to be defined to be used by the options in XABSL. That is done by the element “*skills*” (cf. Fig. F.4)

“*skills*” can contain any number of the element “*skill*”, which defines a representation of a skill. “*skill*” has the required attributes “*name*” for the name and “*description*” as an verbal

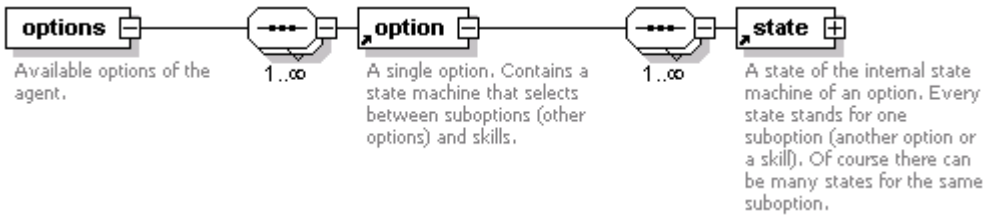


Figure F.5: The element *options*

description of what the skill does. In addition, any number of "*skill-parameter*" child elements can be inserted for the numeric parameters the skill has. They require the same attributes as "*decimal-input-symbol*".

F.5 The Element *options*

After having defined the interfaces to the software environment, the behavior itself can be specified. "*options*" (cf. Fig. F.5) contains all the options of the agent. The attribute "*initial-option*" indicates which option is the root option, i. e. the execution of the option tree starts there.

As the previously introduced elements, each option has the required attributes "*name*" and "*description*". "*initial-state*" refers to the initial state of the state machine. Then for each state of the state machine a child element "*state*" is inserted.

F.6 The Element *state*

The element "*state*" represents a single state of an option's internal state machine. "*name*" is a required attribute. Then each state must have a "*following-option*" or a "*following-skill*" child element. Both have the required attribute "*ref*" that must be the name of an existing option or skill. If the state is active, the referred option or skill is executed after that option.

If a skill shall be executed when the state is active ("*following-skill*"), "*set-skill-parameter*" child elements can be inserted. The attribute "*name*" has to be the name of an existing parameter of the skill. The value of the parameter itself is a decimal expression (cf. Sect. F.11).

With the optional child element "*set-output-symbol*" output symbols can be set. The attribute "*ref*" has to be the name of an existing enumerated output symbol. "*value*" is the value the symbol is set to. If the state is active, then the value of the function or variable that the symbol is referring to is set to the given value.

Then each state must have the element "*decision-tree*". That is a recursive tree with *if* / *else-if* / *else* nodes and transitions to other states as leaves. When an option is executed, the active state's decision tree is carried out to determine the next active state. "*decision-tree*" is a "*statement*", which is a recursive data type, too (cf. next section).

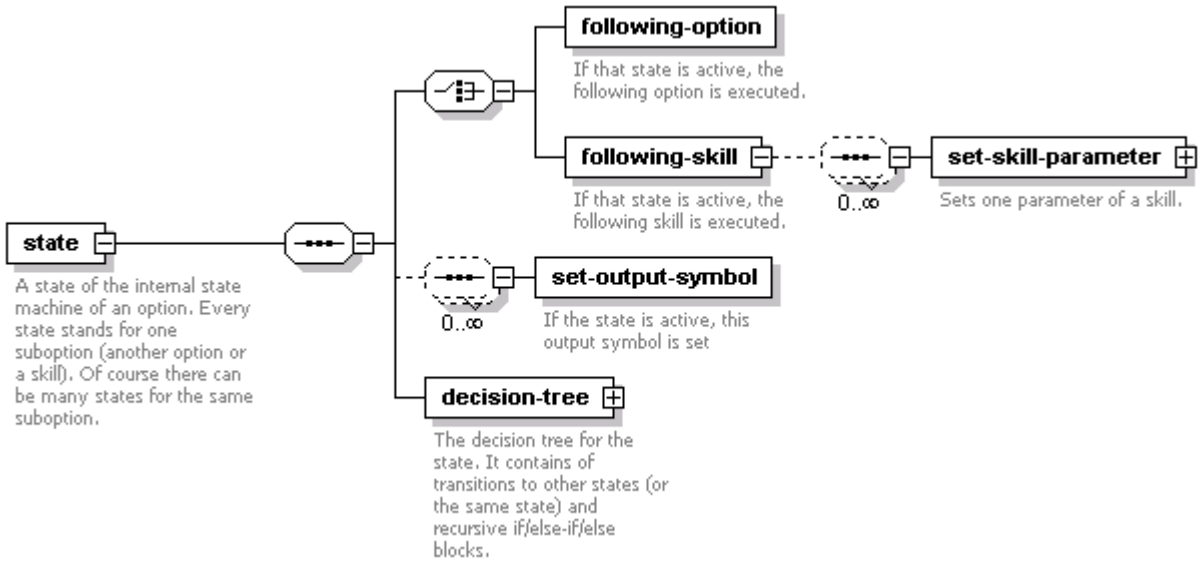


Figure F.6: The element *state*

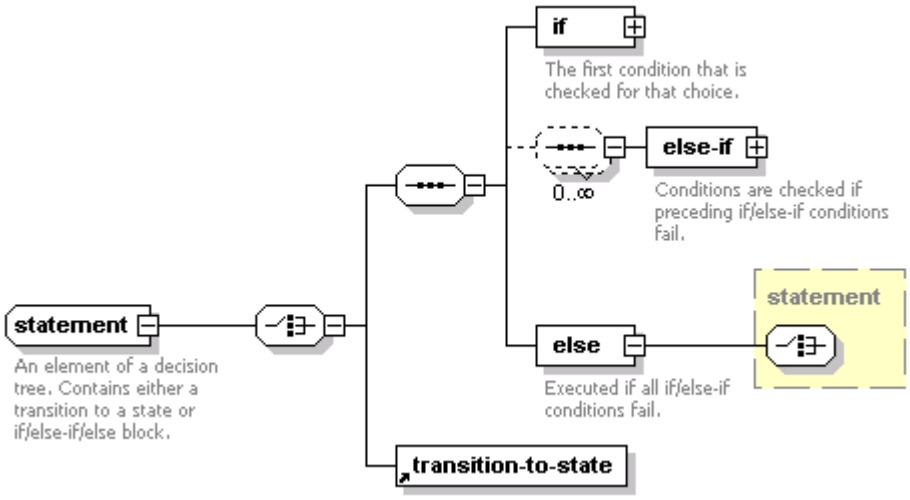


Figure F.7: The type *statement*

F.7 The Type *statement*

Statements are nodes of a decision tree. Either they are a leaf of the tree (“*transition-to-state*”) or an *if* / *else-if* / *else* block. The “*transition-to-state*” element has the required attribute “*ref*” that has to be the name of an existing state in the option.

If the statement is not a transition to a state, than it has to contain exactly one “*if*”, any number of “*else-if*” and exactly one “*else*” elements. The “*else*” element is of the type “*statement*” again. “*if*” and “*else-if*” are of the type “*conditional-statement*” (cf. next section).

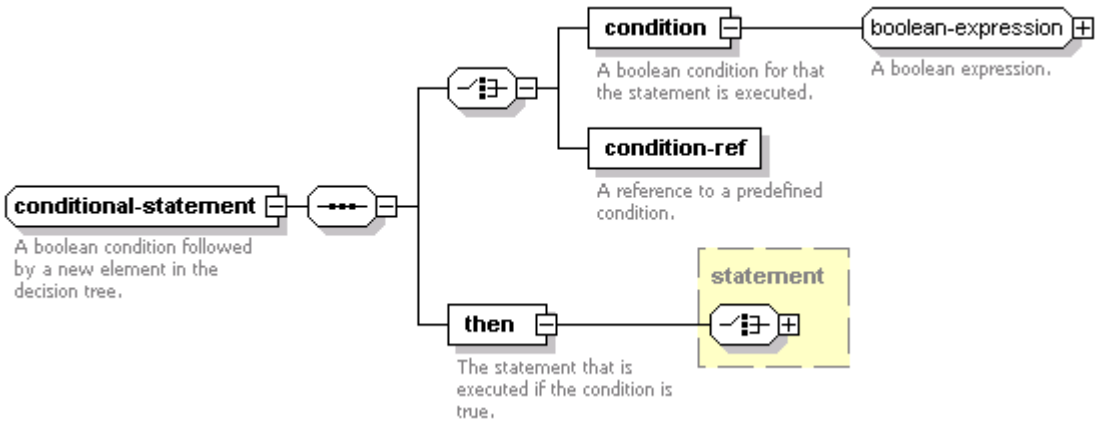


Figure F.8: The type *conditional-statement*

The execution of the statement starts with the "*if*" case of the statement. If the condition for the "*if*" case fails, all "*else-if*" conditions are checked. If no condition becomes true, the statement in the "*else*" statement will be executed.

F.8 The Type *conditional-statement*

Both "*if*" and "*else-if*" are of type "*conditional-statement*" (cf. Fig. F.8). Such an element contains a condition and a statement that is executed if the condition becomes true. The condition is either a "*condition*" element that contains the group "*boolean-expression*" (cf. next section) or a "*condition-ref*" element that refers to a predefined condition that was declared in the "*environment*" element. "*condition-ref*" has the required attribute "*ref*" that has to be the name of an existing predefined condition. There is no difference in the execution of the conditional statement whether "*condition*" or "*condition-ref*" is used. The predefined conditions were only introduced to increase the editing comfort. The conditional statement has to contain one "*then*" element that is of type "*statement*" again. That statement is executed if the condition or the referred condition becomes true.

F.9 The Group *boolean-expression*

"*condition*" elements refer to the group "*boolean-expression*" (cf. Fig. F.9). "*boolean-expression*" has to contain one of the following child elements:

boolean-input-symbol-ref. A reference to a boolean input symbol. The expression results in true if the variable that the symbol stands for is true or if the function the symbol stands for returns true. The required attribute "*ref*" has to be the name of an existing boolean input symbol.

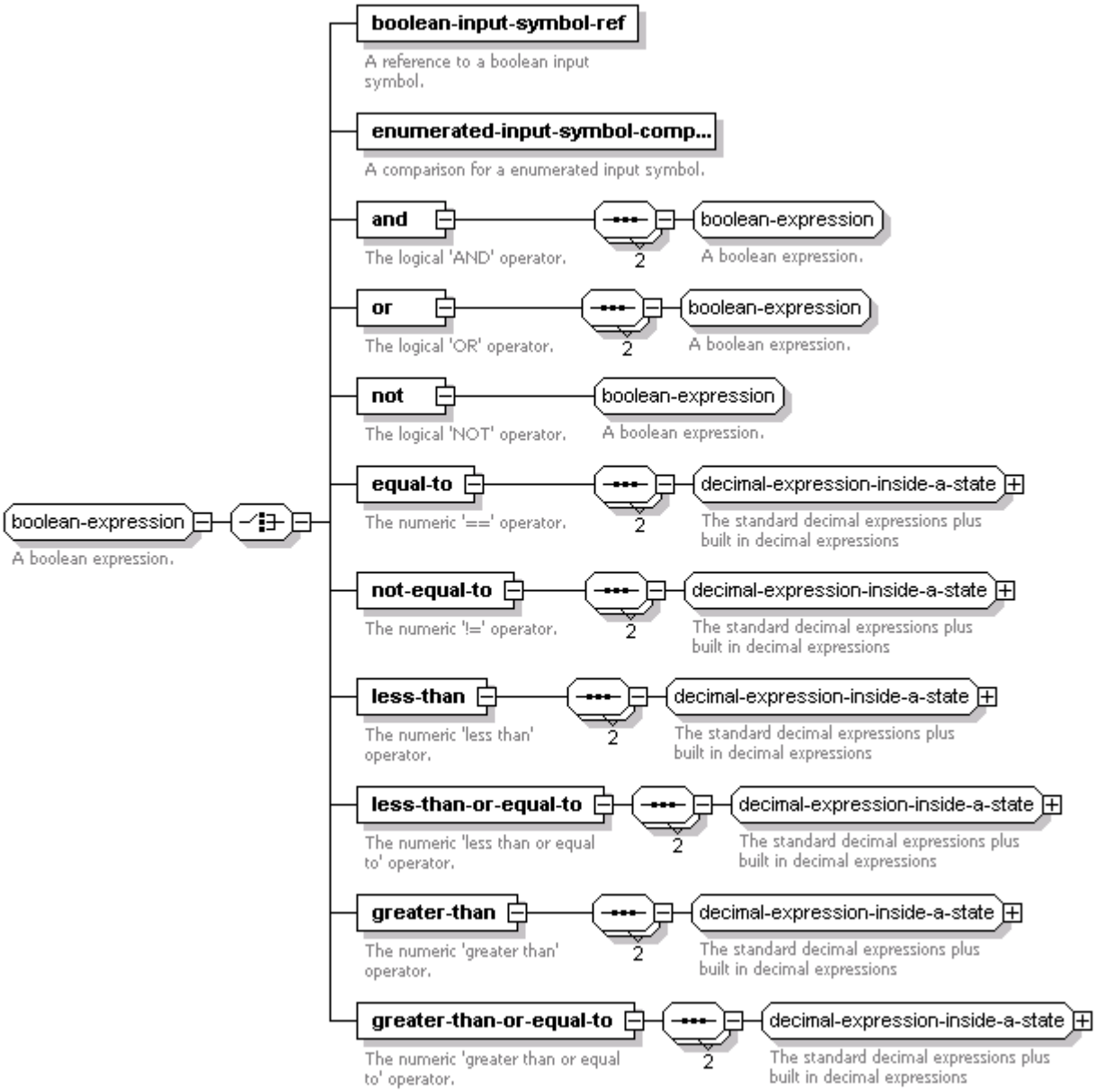


Figure F.9: The group *boolean-expression*

enumerated-input-symbol-comparison. Returns true if an enumerated input symbol has a given value. An example from robot soccer:

```
<enumerated-input-symbol-comparison
  ref="self.switches" enum-element="back-pressed"/>
```

That expression becomes true, if the symbol "*self.switches*" has the value "*back-pressed*". In the example that means that the back switch of the Sony robot was pressed. The required

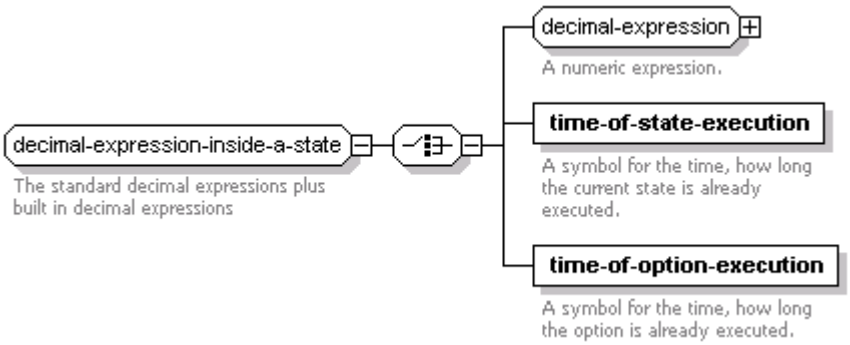


Figure F.10: The group *decimal-expression-inside-a-state*

attribute "*ref*" has to be the name of an existing enumerated input symbol, "*enum-element*" has to be the name of an enumeration element of that symbol.

and / or. The logical && / || operator. Both have two "*boolean-expression*" child elements. If the expression is executed, first the two boolean operand expressions are evaluated. After that, the operator is applied and the result will be returned.

not. The logical ! operator. It has a single "*boolean-expression*" child element.

equal-to / not-equal-to / less-than / less-than-or-equal-to / greater-than / greater-than-or-equal-to. The == / != / < / <= / > / >= operator. All of them have two "*decimal-expression-inside-a-state*" (cf. next section) child elements. If these expressions are executed, the decimal values of the operands are determined. Then the operator is applied and the result will be returned.

That possibly looks quite complicated, but the example in section F.12 might help to understand how to notate such expressions.

F.10 The Group *decimal-expression-inside-a-state*

In contrast to boolean expressions decimal expressions return a decimal value when they are executed. The group "*decimal-expression-inside-a-state*" is either of the group "*decimal-expression*" (cf. next section) or of one of the following two elements:

time-of-state-execution is a built-in decimal symbol that determines how long the current state is already active (in milliseconds).

time-of-option-execution is a built-in decimal symbol that determines how long the current option is already activated (in milliseconds).

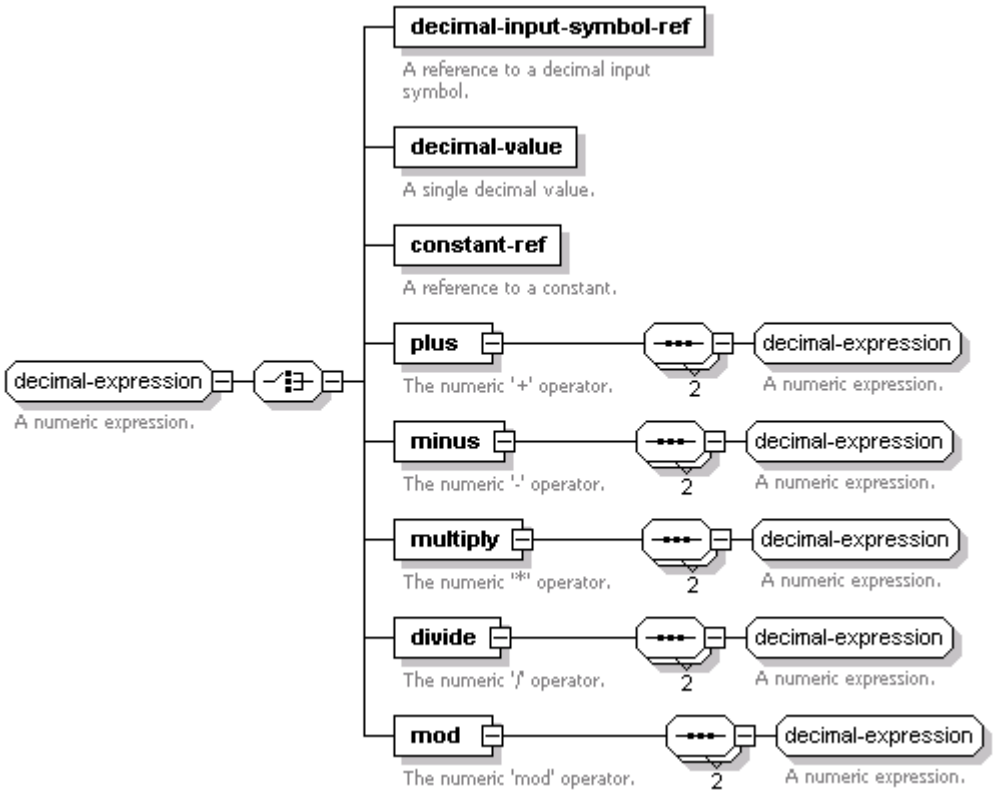


Figure F.11: The group *decimal-expression*

F.11 The Group *decimal-expression*

The group “decimal-expression” (cf. Fig. F.11) has one of the following child elements:

decimal-input-symbol-ref. A reference to a decimal input symbol. If that expression is executed, the value of the variable that the symbol stands for is returned, if the symbol stands for a function, then the function is executed its result will be returned. An example:

```
<decimal-input-symbol-ref ref="ball.distance"/>
```

That expression returns the value of the distance to the ball.

decimal-value. A decimal value. Example:

```
<decimal-value value="150"/>
```

That expression returns the value *150.0*.

constant-ref. A reference to a decimal constant. Example:

```
<constant-ref ref="goalie.home-position.x"/>
```

If, e. g., the constant was declared as *-2000*, the expression returns *-2000*.

plus / minus / multiply / divide / mod. The numeric + / - / * / / / % operators. Each of these elements contains two "*decimal-expression*" child elements. If the expression is executed, first both operand expressions are executed and then the operator is applied and the result will be returned.

F.12 Example

From the robot soccer example, a state "*get-to-ball*" of the option "*goalie-play*" (cf. Fig. 3.21) is shown below:

```
<option name="goalie-play" initial-state="stay-in-goal"
        description="Goalie when not waiting for a kick off">
    ...
 <state name="get-to-ball">
  <following-skill ref="go-to-ball"/>
  <set-output-symbol ref="head-control-mode"
                     value="search-for-ball"/>
  <decision-tree>
   <if>
    <condition description="ball seen">
     <less-than>
      <decimal-input-symbol-ref ref="ball.time-since-last-seen"/>
      <decimal-value value="2000"/>
     </less-than>
    </condition>
    <then>
     <if>
      <condition description="ball.distance smaller than 15 cm">
       <less-than>
        <decimal-input-symbol-ref ref="ball.distance"/>
        <decimal-value value="150"/>
       </less-than>
      </condition>
      <then>
       <transition-to-state state="clear-ball"/>
      </then>
     </if>
     <else-if>
      <condition description="ball too far away">
       <greater-than>
```

```
          <decimal-input-symbol-ref ref="ball.distance"/>
          <decimal-value value="900">
         </greater-than>
        </condition>
        <then>
         <transition-to-state state="return-to-goal"/>
        </then>
       </else-if>
       <else>
        <transition-to-state state="get-to-ball"/>
       </else>
      </then>
     </if>
     <else>
      <transition-to-state state="return-to-goal"/>
     </else>
    </decision-tree>
   </state>
   ..
  </option>
```

If the state “*get-to-ball*” of the option “*goalie-play*” is active, the skill “*go-to-ball*” is executed after that option. If that state is active, the output symbol “*head-control-mode*” is set to the value “*search-for-ball*”.

F.13 Tools

From an XABSL instance file, three different documents can be generated using XSL style sheets:

Intermediate Code. As it is often hard to deploy XML software on robotic platforms, the XABSL files are not parsed directly on the robot. Instead an intermediate code is generated and later parsed on the agent’s system by the *XabslEngine* (cf. App. G) The code generated has a very easy context free grammar consisting only of numbers and a few strings for symbols.

Debugging symbols. The engine internally references all symbols, options, or states only by numbers. For debugging tools and mechanisms, the document provides a mapping of the numbers the engine uses to names of the entities.

Documentation. A detailed documentation can be generated from the formalized behavior (the images in Fig. 3.20 and Fig. 3.21 are taken from that documentation). After making changes in the behavior it is recommended to have a look at the documentation generated first before testing the behavior. Many possible errors can be seen there very quickly.

A XABSL distribution contains amongst other things the following six files:

```
xabsl/generate-debugging-symbols.xsl
xabsl/generate-documentation.xsl
xabsl/generate-intermediate-code.xsl

xabsl/make-debugging-symbols.bash
xabsl/make-documentation.bash
xabsl/make-intermediate-code.bash
```

Although the three style sheets do the transformations, it is useful to invoke them by using the three *bash* scripts.

They require a path to an executable XSTL processor (the GermanTeam uses Xalan). The scripts provided require a *bash* shell. In addition, the generation of the documentation requires an installed DOT tool (included in Graphviz) for the image generation.

Appendix G

The XabslEngine Class Library

The *XabslEngine* is a C++ code library that is needed to run a behavior that is formalized in XABSL on the desired platform. The engine was intended to be platform and application independent. This results in a variety of abstract helper classes that have to be adapted to the current software environment. The following sections describe how to embed the engine into a software project.

G.1 Files

The following files are needed for the Engine:

```
XabslEngine/XabslDefinitions.h
XabslEngine/XabslEngine.cpp
XabslEngine/XabslEngine.h
XabslEngine/XabslOption.cpp
XabslEngine/XabslOption.h
XabslEngine/XabslSkill.h
XabslEngine/XabslSymbols.cpp
XabslEngine/XabslSymbols.h
XabslEngine/XabslTools.h
```

The dependencies between these files are shown in figure G.1. To use the engine "*XabslEngine/XabslEngine.h*" needs to be included.

G.2 XABSL Definitions

In "*XabslEngine/XabslDefinitions.h*" the data type of decimal symbols and skill parameters has to be defined with a *typedef*. This can be any floating point type that has the <, <=, >, >=, ==, *!=*, +, -, *, /, and % operators. For instance

```
typedef double DECIMAL;
```

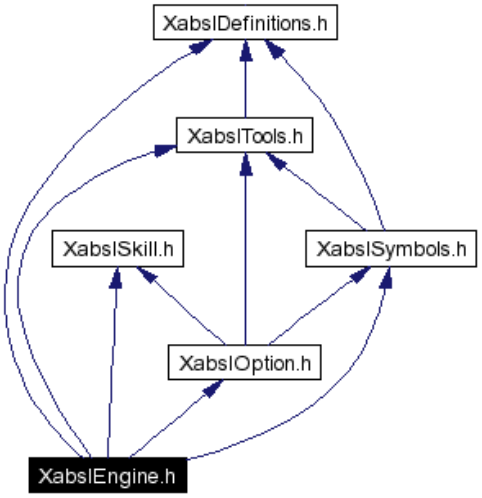


Figure G.1: Include dependencies of the file *XabslEngine.h*

sets the type of decimal entities to *double*.

If the preprocessor directive *XABSL_DEBUG_INIT* is defined, a lot of debug messages will be displayed during the initialization of the engine.

G.3 System Functions

In "*XabslEngine/XabslTools.h*" two abstract helper classes that establish a link to the target platform are defined.

XabslErrorHandler is used by the engine for displaying errors and debug messages:

```
class XabslErrorHandler
{
public:
  XabslErrorHandler() : errorsOccured(false) {};
  virtual void printError(const char* text) = 0;
  virtual void printMessage(const char* text) = 0;
  bool errorsOccured;
  ...
};
```

One has to derive a class (e. g. "*MyErrorHandler*") from that class, and the *printError(..)*, and the *printMessage(..)* functions have to be implemented.

XabslInputFile gives the engine access to a file containing the intermediate code:

```
class XabslInputFile
{
public:
  virtual bool open() = 0;
  virtual void close() = 0;
  virtual DECIMAL readValue() = 0;
  virtual bool readString(char*destination, int maxLength)=0;
};
```

Also from that class a custom class (e. g. "*MyInputFile*") has to be derived, and the pure virtual functions have to be implemented:

open() opens the file containing the intermediate code. Note that the code does not have to be read from a file. It is also possible to read it from a memory region or any other type of stream.

close() is called by the engine after having read the data.

readValue() reads a numeric value from the file.

readString() reads a string from the file.

Please note that the file contains comments (*//...*) that have to be skipped by the read functions:

```
// divide (7)
7
// multiply (6)
6
// decimal value (0): 52.5
0 52.5
// reference to decimal symbol (1) ball.y:
1 13
```

The comments have to be treated as in C++ files, i. e., a comment ends with the end of a line. In the example only *7 6 0 52.5 1 13* should be read from the file.

G.4 Skills

In "*XabslEngine/XabslSkill.h*" a base class for skills is defined. All skills have to be derived from that class:

```
class XabslSkill
{
public:
  virtual void execute(XabslSkillParameters& parameters) = 0;
```

```
};
```

To give the engine access to these custom skills, a class, e. g. "*MySkillCreator*", has to be derived from *XabslSkillCreator*:

```
class XabslSkillCreator
{
public:
  virtual XabslSkill* createSkill(const char* name) = 0;
};
```

Then, the function *createSkill()* has to create a skill, and it has to return a pointer to the skill for the given name of the skill. Example:

```
XabslSkill* MySkillCreator::createSkill(const char* name)
{
  if (strcmp(name,"skill-foo")==0)
    return new SkillFoo();
  else
    return 0;
};
```

G.5 Symbols

As the formalization of the behavior uses symbols that stand for the entities in the software environment, the mapping from these symbols to real variables and functions has to be done. For that one has to derive a class (e. g. "*MySymbolProvider*") from *XabslSymbolProvider* ("*XabslEngine/XabslSymbols.h*"):

```
class XabslSymbolProvider
{
public:
  virtual void getDecimalInputSymbol(const char* stringID,
    XabslDecimalInputSymbol& symbol) = 0;

  virtual void getBooleanInputSymbol(const char* stringID,
    XabslBooleanInputSymbol& symbol) = 0;

  virtual void getEnumeratedInputSymbol(const char* stringID,
    XabslEnumeratedInputSymbol& symbol) = 0;

  virtual void getEnumeratedOutputSymbol(
    const char* stringID,
```

```
    XabslEnumeratedOutputSymbol& symbol) = 0;

  virtual bool getValueOfEnumElement(const char* stringID,
    const char* enumElement,
    int& value) = 0;
};
```

On initialization the engine asks the symbol provider to return the address of a variable or a function for every symbol. An example from the GT2002 project:

```
void MySymbolProvider::getDecimalInputSymbol(
  const char* stringID,
  XabslDecimalInputSymbol& symbol)
{
  if (strcmp(stringID,"ball.x")==0)
    symbol.pVariable =
      (DECIMAL*)&(worldState.pBallPosition->getPosition().x);

  else if (strcmp(stringID,"ball.angle")==0)
    symbol.pFunction = &getBallAngle;
}
```

When the engine finds the decimal input symbol "*ball.x*" in the intermediate code, it executes the *getDecimalInputSymbol()* function that sets the symbol to a pointer to the variable for the x position in the world state. For the symbol "*ball.angle*" the symbol is set to a pointer to the function "*getBallAngle()*".

Refer to the source code in "*XabslEngine/XabslSymbols.h*" for more details.

G.6 Initializing an Engine

Once all the helper classes have been derived, the engine can be initialized. The constructor of the engine takes the following parameters:

```
XabslEngine(XabslInputFile& inputFile,
            XabslErrorHandler& errorHandler,
            XabslSkillCreator& skillCreator,
            XabslSymbolProvider& symbolProvider,
            TimeFunction timeFunction);
```

Besides references to instances of *XabslInputFile*, *XabslErrorHandler*, *XabslSkillCreator*, and *XabslSymbolProvider* a *TimeFunction* has to be given as a parameter. "*TimeFunction*" is a pointer to a function that returns the current system time in milliseconds, and that is defined as follows:

```
typedef unsigned long (*TimeFunction)();
```

The intermediate code is already parsed in the constructor, and the behavior, all symbols, options, and states with their decision trees are created. When the initialization fails, the engine displays detailed error messages using the *errorHandler*.

The programmer can check whether the creation of the engine was successful by testing the variable *errorsOccured* inside the *errorHandler*.

G.7 Executing the Engine

Only if the engine was created successfully, it can be executed. The execution is done with the parameterless function *execute*:

```
pMyEngine->execute()
```

G.8 Debugging Interfaces

The *XabslEngine* provides a variety of debugging functions:

```
void executeFromOption(int option);
```

executes the option tree not from the specified root option but from the option given as parameter.

The debugging environment should use the generated debug symbols for the mapping from option names to numbers. This is the same with all the other debug interface functions.

```
void setSkillParameter(int skill, int param, DECIMAL value);
```

sets a parameter of a skill.

```
DECIMAL getSkillParameter(int skill, int param) const;
```

returns the value of a parameter of a skill.

```
int getNumberOfSkillParameters(int skill) const;
```

returns the number of parameters of a skill.

```
void executeSkill(int skill);
```

prevents the execution of the option tree and executes only a specified skill.

```
int getSelectedSkill();
```

returns the selected skill.

```
DECIMAL getDecimalInputSymbol(int symbol) const;
```

returns the value of a decimal input symbol.

```
bool getBooleanInputSymbol(int symbol) const;
```

returns the value of a boolean input symbol.

```
int getEnumeratedInputSymbol(int symbol) const;
```

returns the value of an enumerated input symbol.

```
int getEnumeratedOutputSymbol(int symbol) const;
```

returns the value of an enumerated output symbol.

```
int setEnumeratedOutputSymbol(int symbol, int value);
```

sets the value of an enumerated output symbol.

```
int getInitialOption() const;
```

returns the initial option.

```
int getFollowingSubOption(int option) const;
```

returns the option that is the current successor of an option.

```
int getActiveState(int option) const;
```

returns the active state of an option.

```
unsigned long getTimeSinceOptionStart(int option) const;
```

returns how long an option has already been activated.

```
unsigned long getTimeSinceActiveStateStart(int option) const;
```

returns how long a state of an option has already been activated.

Appendix H

XABSL in GT2002

In GT2002 a variety of solutions for behavior control modules exist. One of them is the *XabslBehaviorControl* which makes use of a formalization of robot behavior in an XML language called XABSL. This chapter describes how XABSL is integrated in the GT2002 project and how to develop behaviors using it.

H.1 XabslBehaviorControl

The class *XabslBehaviorControl* is a *BehaviorControl* solution that embeds the *XabslEngine* into the GT2002 environment. It consists of the following C++ files that are, as nearly all files mentioned in this appendix, located in *T:\GT2002\Src\Modules\BehaviorControl\XabslBehaviorControl*.

```
XabslBehaviorControl.cpp
XabslBehaviorControl.h
XabslGT2002SymbolProvider.cpp
XabslGT2002SymbolProvider.h
XabslSkills.cpp
XabslSkills.h
```

To develop behaviors only the files *XabslGT2002SymbolProvider.h*, *XabslGT2002SymbolProvider.cpp* (cf. Sect. H.2.5), *XabslSkills.h*, and *XabslSkills.cpp* (cf. Sect. H.2.6) need to be edited.

XabslBehaviorControl derives from three classes (cf. H.1): *BehaviorControl* is the base class of all behavior control modules. *XabslErrorHandler* and *XabslSkillCreator* are helper classes for the *XabslEngine*.

The *XabslBehaviorControl* has a member *pEngine*, which is a pointer to an *XabslEngine*. The creation of the engine on the robot takes a few seconds because of the slow file access to the intermediate code.

If the creation of the engine fails, an error messages is generated and *XabslBehaviorControl* sends the motion request "*scratchHead*" that causes the robot to scratch its head and do nothing

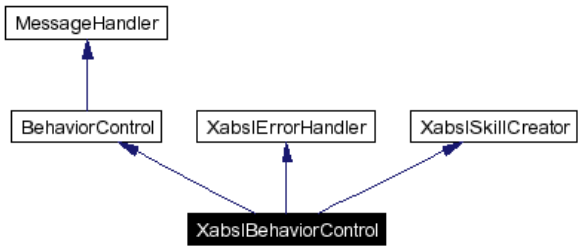


Figure H.1: Inheritance graph of class *XabslBehaviorControl*

else. In most cases, the creation of the engine fails due to a missing skill or symbol that was added to the XML file but not to the *XabslBehaviorControl*.

H.2 Editing Behaviors

H.2.1 Files

The entire formalized behavior is contained in the file *Behavior-for-Fukuoka.xml*. Although it can be edited using any text editor, it is highly recommended to use *XML Spy* (*http://www.xmlspy.com*, version 4.3 or better). Up to now this is the only XML editor known to the authors that can completely validate identity constraints in XML schemas.

When using *XML Spy*, it is recommended to open the workspace file *xmlspy-workspace.spp* instead of opening the XML file directly. This workspace contains all the XABSL related files including *Behavior-for-Fukuoka.xml*.

H.2.2 Coding Conventions

The standard XML coding conventions should be applied (*XML Spy* does a lot of this automatically). Attributes should be of the style "*funny-nice-behavior*" and "*very-important-symbol*" (not "*funnyNiceBehavior*", "*VeryImportantSymbol*", "*funny_nice_behavior*", "*very_important_symbol*". The "*description*" attributes should carefully be set to help other developers to understand what the behavior does.

H.2.3 Validation

It is required to validate the XML file with "*xabsl/xabsl-1.0.xsd*" before applying the style sheets if the XSLT processor does not already do this (e. g. xalan, which is used in GT2002, produces only cryptic error messages and does not check all constraints correctly.). Invalid XML files produce unpredictable results since the *XabslEngine* doesn't perform a validity check.

XML Spy automatically validates the file while saving. If an identity constraint is not met, *XML Spy* shows the name of the constraint, which often tells what to do. When using another validation method, make sure that the program can validate identity constraints.

H.2.4 Generating Files

There are four *bash* scripts in the *XabslBehaviorControl* directory:

make-code.bash generates the intermediate code (*T:\GT2002\Config\xabsl-intermediate-code.dat*) and the debugging symbols (*T:\GT2002\Config\xabsl-debugging-symbols.dat*). As the biggest part of *xabsl-intermediate-code.dat* consists of whitespace and comments, a cleaned version is written to *T:\GT2002\Config\xabsl-ic.dat* from that file that will later be copied to the memory stick.

make-documentation generates a detailed HTML documentation of the behavior (*T:\GT2002\Doc\XabslAgentBehaviorDocumentation\index.html*).

make-all.bash executes both of the previous two scripts.

make-all-without-documentation calls only *make-code.bash* and marks the previously generated documentation files as obsolete so that no misunderstandings about differences between the robot's behavior and the documentation can arise.

In the GT2002 projects *GT2002*, *RobotControl* and *SimGT2002 make-all-without-documentation.bash* is automatically executed during a build when the XML file has changed. The documentation needs to be generated manually: By building the *Documentation* project with the configuration *Xabsl Behavior* or by running *make-documentation.bash* directly.

H.2.5 Adding Symbols

After adding a new input or output symbol to the XML file, the file *XabslGT2002SymbolProvider* has to be updated accordingly. Depending on the type of the symbol that was added, the *getDecimalInputSymbol*, *getBooleanInputSymbol*, *getEnumeratedInputSymbol* or *getEnumeratedOutputSymbol* function must be extended.

The name of the symbol is always given with the parameter *stringID*. The result has to be filled into the variable *symbol* that is always the second parameter. If the symbol stands for a variable in the software environment, *symbol.pVariable* has to be set to the address of that variable. If the symbol stands for a function, *symbol.pFunction* has to be set to the address of that function. Add the function to the *XabslGT2002SymbolProvider*. Example:

```
void XabslGT2002SymbolProvider::getDecimalInputSymbol(
  const char* stringID,
  XabslDecimalInputSymbol& symbol)
{
  if (strcmp(stringID,"ball.x")==0)
    symbol.pVariable =
      (DECIMAL*)&(worldState.pBallPosition->getPosition().x);

  else if (strcmp(stringID,"ball.angle")==0)
```

```
    symbol.pFunction = &getBallAngle;
         ...
}
```

For the symbol "*ball.x*" a pointer to the variable in the world state is returned. For "*ball.angle*" a pointer to the member function *getBallAngle()* is returned.

Note that the function referenced has to be defined as static. If the functions are members of *XabslGT2002SymbolProvider*, a work-around is used to give the static functions access to the member variables of the class: *getInstance()* returns a pointer to the instance of *XabslGT2002SymbolProvider*. Example (definition and implementation):

```
static DECIMAL getBallAngle();

DECIMAL XabslGT2002SymbolProvider::getBallAngle() {
  return (Geometry::angleTo(
    getInstance()->worldState.pRobotPose->getPose(),
    getInstance()->worldState.pBallPosition
       ->getPosition())).toDegrees();
}
```

H.2.6 Adding Skills

After adding a new skill to the XML file a class has to be added to *XabslSkills.h* and *XabslSkills.cpp*, e. g.:

```
class XabslSkillStand : public XabslGT2002Skill
{
public:
  XabslSkillStand(const WorldState& worldState,
    const XabslGT2002SymbolProvider& symbolProvider,
    MotionRequest& motionRequest)
    :  XabslGT2002Skill(worldState, symbolProvider,motionRequest) {};
  virtual void execute(XabslSkillParameters& parameters);
};
```

and

```
void XabslSkillStand::execute(
  XabslSkillParameters& parameters)
{
  motionRequest.motionType = MotionRequest::stand;
}
```

At last, the string id of the skill as used in XML has to be connected to the class, e. g., *XabslBehaviorControl::createSkill* has to extended like this:

```
XabslSkill* XabslBehaviorControl::createSkill(
  const char* name)
{
  if (strcmp(name,"stand")==0)
    return new XabslSkillStand(
      *worldState, *pSymbolProvider, *motionRequest);
  else if (strcmp(name,"turn")==0)
    return new XabslSkillTurn(
      *worldState, *pSymbolProvider, *motionRequest);
          ...
```

H.3 Testing and Debugging

An extensive debugging interface is included into *XabslBehaviorControl*. Almost all internal states of the engine can be monitored and changed. The data type *XabslDebugRequest* can be sent to the module. It contains a request which state parameters and states to be monitored and allows to influence the execution of the engine. The module sends *XabslDebugMessage*s.

In the RobotControl application the *Xabsl Behavior Tester Dialog* (cf. Sect. J.5.1) provides a user interface to access these debugging mechanisms.

Appendix I

SimGT2002 Usage

SimGT2002 is based on SimRobot [1], a kinematic robotics simulator. In fact, only a so-called controller has been added to SimRobot, that provides the same environment to robot control code that it will also find on the real robots. Therefore, SimGT2002 shares the user interface with SimRobot. This user interface is documented in the online help file that comes with SimRobot. In addition, the scene description language that is used to model the simulation scenes is explained in the help file. Hence, these descriptions are not repeated here.

I.1 Getting Started

SimGT2002 can either be started directly from the Windows Explorer (from *T:\GT2002\Bin*), from Microsoft Developer Studio, or by starting a scene description file[1]. In the first case, a scene description file has to be opened manually, whereas it will already be loaded in the latter two cases. When a simulation is started for the first time and no layout has been patched into the Windows registry, only the editor window will show up in the main window. Select *Simulation|Start* to run the simulation. The *Tree View* will appear. A *Scene View* showing the soccer field can be opened by double-clicking *WORLD GT2002*. However, the scale of the display will not be appropriate. After selecting *View|Zoom|4x* and *View|Perspective Distortion|Level 1* the field will fit into the window. In addition, the presentation can be simplified by reducing the *View|Detail Level*. Please note that there also exist keyboard shortcuts and toolbar buttons for most commands.

After starting a simulation, a script file may automatically be executed, setting up the robots as desired. The name of the script file is part of the scene description file. Together with the ability of SimRobot to store the window layout, the software can be configured to always start with a setup suitable for a certain task.

Although any object in the *Tree View* can be opened, only displaying certain entries in the object tree makes sense, namely the *WORLD*, the objects in the group “robots”, and all *VALUE*s of objects starting with *VIEW*.

[1] This will only work if SimGT2002 was started at least once before.

I.2 Scene View

The *Scene View* appears if the *WORLD* is opened from the *Tree View*. As stated above, a 4x-zoom and a level 1 perspective distortion are ideal for displaying the field. The view can be rotated around two axes, and it supports several mouse operations:

- If a robot is clicked between its forelegs (on the field plane), it can be dragged to another position.
- If a robot is clicked between its hind legs, it can be rotated around its body center, i. e. the middle between its forelegs.
- If an active robot (see below) is double-clicked, it is the currently selected robot, i. e. the robot console commands are sent to.
- The ball can be dragged around. Note that its "click position" is on the field plane.
- If the bar from the corresponding challenge is available, it can be dragged around by clicking on its center.
- The bar can also be rotated by dragging one of its ends.

I.3 Robot View

A *Scene View* containing a single robot can be opened by double-clicking a *VEHICLE* in the sub-tree *robots* of the *Tree View*. In such a view, the robot is displayed centered, and it can be zoomed to fill the entire window. This allows seeing more details of the robot, e. g. the state of its LEDs. The view supports four different mouse actions:

- If the back of the robot is clicked, this simulates an activated back switch of the robot. A second click will deactivate the switch.
- If the back area of the top surface of the head is clicked, this simulates an activated back head switch. Again, a second click will deactivate the switch.
- If the front area of the top surface of the head is clicked, this simulates an activated front head switch. It is deactivated by a second click.
- A double-click in the window throws the robot on its side. Note that this can only be seen in the global *Scene View*. In the robot view, the fact that the robot felt down is visualized by hiding its tricot. A second double-click will bring the robot back on its feet.

I.4 Scene Description Files

The language of scene description files is documented in the online help file of SimRobot. However, there are some facts that are special in SimGT2002:

- At the top of a scene description, just below *WORLD*, the instruction *REMARK* can be used to specify the name of the script that will be executed when the simulator is started. A script file contains commands as specified below, one command per line. The default location for scripts is *T:\GT2002\Config\Scripts*, their default extension is *.con*. If no file is specified, SimGT2002 will use *T:\GT2002\Config\Scripts\console.con* if it exists.
- Near the end of a scene description file, there is a group called "robots". It contains all *active* robots, i. e. robots for which processes will be created.
- Below the group "robots", there is the group "extras". It contains *passive* robots, i. e. robots which just stand around, but which are not controlled by a program. Passive robots can be activated by moving their definition to the group "robots".
- Below that, there is the group "balls". It contains the balls, i. e. normally a single ball, but several ones for the ball challenge.
- There can also be a macro "bar", which was used for the corresponding challenge.

A lot of scene description files can be found in *T:\GT2002\Config\Scenes*.

I.5 Console Commands

Console commands can either be directly typed into the console window or they can be executed from a script file. There exist two different kinds of commands. On the one hand, *global commands* change the state of the whole simulation, or they are always sent to all robots. On the other hand, *robot commands* only have an impact on the set of currently *selected robots*.

I.5.1 Global Commands

call <**file**>**.** Executes a script file. A script file contains commands as specified here, one command per line. The default location for scripts is *T:\GT2002\Config\Scripts*, their default extension is *.con*.

cls. Clears the console window.

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

gc reset | **ready** | **playing** | **final** | **kickOff (blue** | **red)** [<**blueScore**> <**redScore**>]**.** Game control. The command is sent to all robots. The *kickOff*-command is interpreted according to the team color of each robot. *gc reset* resets the score counters.

help | **?.** Displays a help text.

robot ? | **all** | <**name**> {<**name**>}**.** Connects the console window to a set of *selected robots*. All commands in next section are only sent to the selected robots. The command *robot ?* displays a list of all robot names. To select a single simulated robot, it can also be double-clicked in the *Scene View*. To select them all, type *robot all*.

sc [**gameManager**] **(** <**a.b.c.d**> | <**a.b.c**> <**d**> {<**d**>} **).** Starts a wireless connection to real robots. The syntax is very similar to the one of the start-script of the router (cf. Sect. 5.3). The command will start the router in the background and will display its messages in the console window. It should only be used once and only in the script that is executed when the simulation is started. It will add new robots to the list of available robots (named by the least significant byte of their IP-addresses), and for each of these robots, a full set of views is added to the *Tree View*. Please note that physical robots only send debug drawings on demand, so the views will remain empty until the drawings are requested by the appropriate debug keys. When the simulation is reset or SimGT2002 is exited, the router will be terminated.

st off | **on.** Switches the simulation of time on or off. Without the simulation of time, all calls to *SystemCall::getCurrentSystemTime()* will return the real time of the Windows PC. However, as the simulator runs slower than real-time, the simulated robots will receive less sensor readings than the real ones. If the simulation of time is switched on, each step of the simulator will advance the time by 8 ms. Thus, *SimGT2002* simulates real-time, but it is running slower. By default this option is switched off.

<**text**>**.** Comment. Useful in script files.

I.5.2 Robot Commands

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

dk ? | **(** <**key**> **off** | **on** | <**number**> | **every** <**number**> [**ms**] **).** Sets a debug key. The GermanTeam uses so-called debug keys to switch several options on or off at runtime. Type *dk ?* to get a list of all available debug keys. Debug keys can be activated permanently, for a certain number of times, or with a certain frequency, either on a counter basis or on time (cf. App. E.2). All debug keys are switched off by default.

hcm ? | <**mode**>**.** Sets the head control mode (cf. Sect. 3.7.3). Type *hcm ?* to get a list of all available head control modes.

hmr <**tilt**> <**pan**> <**roll**>**.** Sends a head motion request, i. e. it sets the joint angles of the three axes of the head. This will only work if the actual head control mode is *none*. The angles have to be specified in degrees.

mr ? | <**type**> [<**x**> <**y**> <**r**>]**.** Sends a motion request. This will only work if no *behavior control* is active. Type *mr ?* to get a list of all available motion requests. Walk motions also have to be parameterized by the motion speeds in forward/backward, left/right, and clockwise/counterclockwise directions. Translational speeds are specified in millimeters per second; the rotational speed has to be given in degrees per second.

msg off | **on.** Switches the output of text messages on or off. All processes can send text messages via their debug queues to the console window. As this can disturb entering text into the console window, it can be switched off. However, by default text messages are printed.

pr continue | **illegalDefender** | **obstruction** | **keeperCharged** | **ballHolding.** Penalize robot. The command sends one of the four penalties to all selected robots, or it signals them to continue with the game after a penalty.

prc ? | <**role**> **(blue** | **red).** Sets the player role and the team color. Type *prc ?* to see all player roles available.

so off | **on.** Switch sending of *oracled world states* on or off. *Oracled world states* are normally sent to all processes. This allows the modules calculating the world state to be switched off without a failure of the robot. However, the option can produce confusing results if parts of the world state are only sometimes calculated by the robot. Then, the world state sometimes results from the robot's own calculations and sometimes from the simulator. Therefore, sending oracled world states to the robots can be switched off. By default, it is switched on. Note that this command only has an effect on simulated robots.

sr ? | <**task**> **(?** | <**solution**> **).** Sends a solution request. This command allows switching the solutions for a certain task. Type *sr ?* to get a list of all tasks. To get the solutions for a certain task, type *sr* <*task*> *?*.

I.6 Adding Views

In SimGT2002, *views* are used to display debug drawings (cf. App. E.5). These are generated by the robot control program, and they are sent to SimGT2002 via *message queues* (cf. App.E.1). In SimGT2002, the views are defined in the source code. They are instantiated separately for each robot. All views in the current code are defined in *T:\GT2002\Src\Platform\Win32\SimRobot\RobotConsole.cpp*. There are two kinds of views: *image views* and *field views*.

Image Views. An image view displays information in the system of coordinates of a camera image. It is defined by giving it a name and by listing the debug drawings that will be part of the view, e. g. the *imagePerceptCollection* view is defined as:

```
IMAGE_VIEW(imagePerceptCollection)
  image, perceptCollection, lines
END_VIEW(imagePerceptCollection)
```

Field Views. A field view displays information in the system of coordinates of the soccer field. It is defined similar to image views. For instance, the *worldState* view is defined as:

```
FIELD_VIEW(worldState)
  fieldPolygons, fieldLines, worldState
END_VIEW(worldState)
```

Appendix J

RobotControl Usage

This chapter describes how to use the RobotControl application. As RobotControl is a very complex system, not all features will be described. But it will help to get an overview about the capabilities of the program.

J.1 Starting RobotControl

Requirements. RobotControl needs at least version 4.0 of the Microsoft Internet Explorer installed on your system to work properly. In addition, it is important that RobotControl can access the GT2002 directory tree on drive *T:*.

After the First Start the application looks a little bit strange. No child windows appear and all tool bars are pushed together. But the toolbars can be moved with the mouse to be distributed over more rows. To switch toolbars on or off, right-click on the tool bar area and select the visible tool bars from the popup menu. Dialogs can be opened by using the “View” menu or, for some of them, by using the “Views” toolbar. Note that the window layout will not be restored during the next start of the application unless it is saved using the “configuration” toolbar (cf. Sect. J.2.2).

J.2 Application Framework

The following toolbars and dialogs form the framework of the application.

J.2.1 The Debug Keys Toolbar

Figure J.1: The Debug Keys Toolbar

The *Debug Keys Toolbar* (cf. Fig. J.1) is used to switch debug keys on or off (cf. App. E.2). Each debug key can be parameterized in four different ways describing how often and in which frequency it will be enabled.

The combo box contains all available debug keys. To edit the properties of a debug key, select the key from the list and use on of these buttons:

Disabled. The debug key is disabled.

Always. The debug key is always enabled.

n **times.** The debug key is enabled for n times, i. e., it will return *true* the next n times it is asked, and *false* afterwards. n has to be entered into the edit control before the button is pressed.

Every n **times.** The debug key is enabled every n-th time, i. e., it will return *true* every n-th call, and *false* in between.

Every n **ms.** The debug key is enabled every n milliseconds, i. e., it will return *false* until at least n milliseconds passed since the last time it returned *true*.

Disable All. Disables all debug keys.

There are two debug key tables in RobotControl, one for a physical robot connected via the wireless network, and one for the selected simulated one. With the buttons *Edit table for robot* and *Edit table for local processes* one can select which of them is edited. *Send* sends both debug key tables to the proper destinations by putting them into message queues, i. e. nothing will change as long as *Send* has not been pressed.

J.2.2 The Configuration Toolbar

Figure J.2: The Configuration Toolbar

The *Configuration Toolbar* (cf. Fig. J.2) is intended to manage different configurations of RobotControl. Up to now, it only works for window layouts.

As it is sometimes very difficult to arrange the dialogs and toolbars in the main window for efficient use, the *Configuration Toolbar* allows saving window layouts to the Windows registry. With the *New* button, new layouts can be created from the current one, *Rename* renames a layout, and *Delete* deletes a layout. Note that changes in the layout are not automatically saved. Instead, this has to be done manually by pressing the *Save* button. One can select between different layouts by selecting a different entry in the combo box.

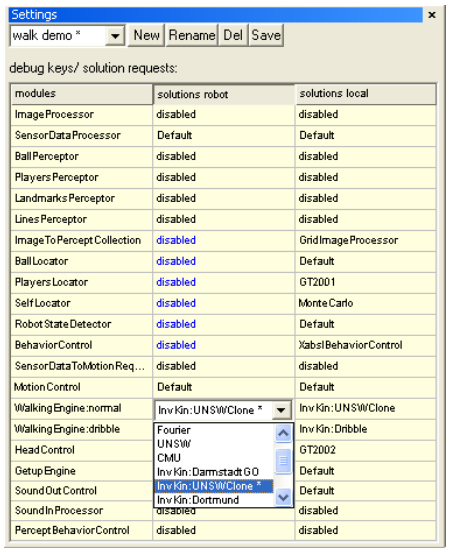


Figure J.3: The Settings Dialog: RobotControl's most often used dialog

J.2.3 The Settings Dialog

With the *Settings Dialog* (cf. Fig. J.3), solutions for modules (cf. Sect. 3) running on the robot or on the PC can be switched. A certain combination of solutions is called a setting and can be stored. All settings are stored in *T:\GT2002\Config\Settings*. The default solution for each module is marked with an asterisk.

J.2.4 The Log Player Toolbar

Figure J.4: The Log Player Toolbar

In RobotControl, logfiles can store a set of messages of any kind used to communicate between modules or between dialogs or toolbars and modules, e. g. it is possible to record pictures sent by a robot in a logfile, and then play that logfile several times to test different kinds of image processing with exactly the same input data.

The *Log Player Toolbar* (cf. Fig. J.4) is used to record, play, and modify such logfiles. Its buttons should be known from other players, e. g. CD-players. *Play*ing a logfile sends all messages in it to all running modules in RobotControl as well as to all dialogs that can handle that type of message. *Step forward* and *Step backward* do the same with single messages.

*Record*ing appends all messages sent from a robot via the wireless network to the actual logfile (in memory) that can be *save*d afterwards.

J.2.5 WLan Toolbar

Figure J.5: The WLan Toolbar

The *WLan Toolbar* (cf. Fig. J.5) is used to create, edit, and switch between different wireless network configurations. It also allows connecting to (and of course disconnecting from) one of the robots in the current wireless network configuration. The *Players Toolbar* decides to which robot a connection is established.

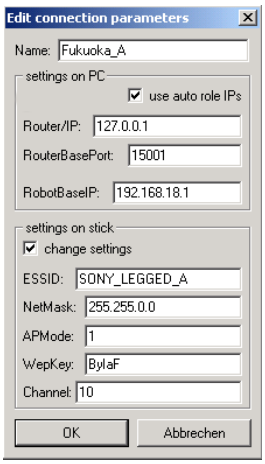


Figure J.6: The WLan New Dialog

Creating new and modifying existing wireless network configurations opens the dialog shown in figure J.6. Connections to robots are established via the router (cf. Sect. 5.3). The dialog allows all relevant parameters to be specified: the (base) IP address of a robot as well as the IP address of the router and its base port that is used to map between port numbers on the router and robot IP addresses to be routed to.

The *AutoIP*-checkbox is used to decide whether the robot IP address is fixed or a base address, to which the team color and the player role are added to to get the real IP address.

Furthermore settings can be edited such as the wireless network *ESSID*, the *Netmask*, the *APmode*, the *WepKey*, and the *Channel*. In addition, it can be selected to save these settings with the *Players Toolbar* described next.

J.2.6 Players Toolbar

The *Players Toolbar* (cf. Fig. J.7) allows choosing the team color and the player role of the actual robot, either a simulated one or a robot connected to via the wireless network. If the current wireless network configuration (cf. Sect. J.2.5) uses *AutoIP*, the IP address automatically changes when changing team color or player role. Team color and player role change messages are sent

Figure J.7: The Players Toolbar

to all processes running on the PC and even to the robot connected via the wireless network. This way team color and player role can be changed at runtime. All these settings can be saved to *T:\GT2002\Config\player.cfg* that is used to determine team color and player role on the PC and on robots at startup. According to the current wireless network configuration (cf. Sect J.2.5) the file *T:\GT2002\Config\wlanconf.txt* is also changed after pressing the *Save*-button. That allows easy creation of memory sticks for a complete team, supported by the *copyfiles*-button.

J.2.7 Game Toolbar

Figure J.8: The Game Toolbar

With the *Game Toolbar* (cf. Fig. J.8) the game control data can be generated and sent. So for tests with a single robot, Sony's RoboCup Game Manager is not needed.

J.3 Visualization

J.3.1 Image Viewer and Large Image Viewer

The two image viewer dialogs (cf. Fig. J.9) display images and debug drawings from the *queueToGUI* (cf. Fig. 5.3) so images from the robot, the log player, or the simulator are displayed. The *Image Viewer* has space for four images with fixed resolution. The *Large Image Viewer* shows only one image and is sizeable. With the context menu different types of images and different debug drawings can be selected. The context menu also contains *Copy drawings* and *Copy image and drawings* which copies only the debug drawings as a vector graphic or the image with the drawings as a bitmap to the clipboard. Important: to see a debug drawing created in a special solution of some module (cf. Sect. 3), this solution has to be selected (cf. Sect. J.2.3). For example, to see the debug drawing *line*, in the settings dialog the solution for the *Lines Perceptor* has to be changed from *disabled* to *Default*.

J.3.2 Field View and Radar Viewer

The *Field View* and the *Radar Viewer* (cf. Fig. J.11) both display percept drawings. The *Radar Viewer* display the percepts relative to the robot. The *Field View* draws the percepts based on the robot's localization on the field. In both dialogs the drawings can be selected with the context menu. *Copy drawings* copies the drawings as a vector graphic to the clipboard.

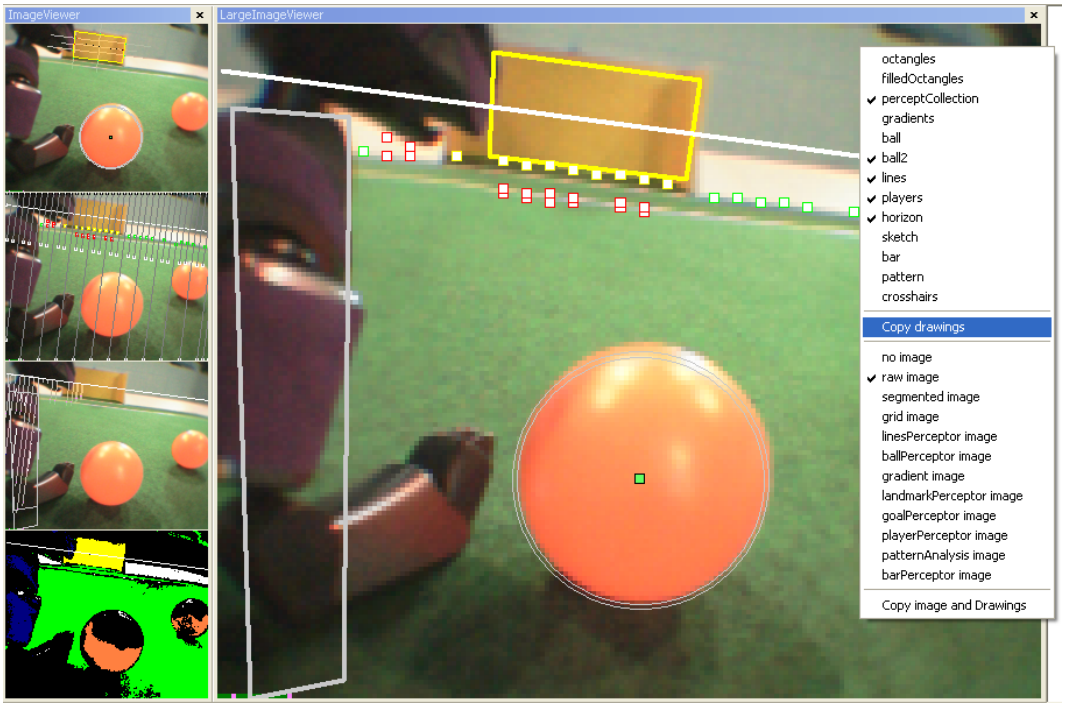


Figure J.9: Image Viewer and Large Image Viewer Dialog

J.3.3 Color Space Dialog

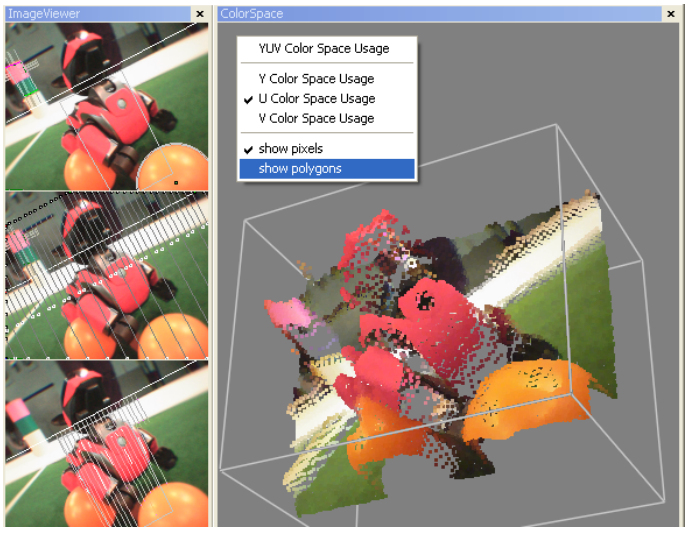


Figure J.10: The Color Space Dialog showing the u-channel of an image as a height map.

The *Color Space Dialog* (cf. Fig. J.10) visualizes how an image uses the YUV color space, and displays the y, u, and v channel as a height map. By dragging with the left mouse button the 3-D scene can be rotated. With the context menu the type of view can be selected.

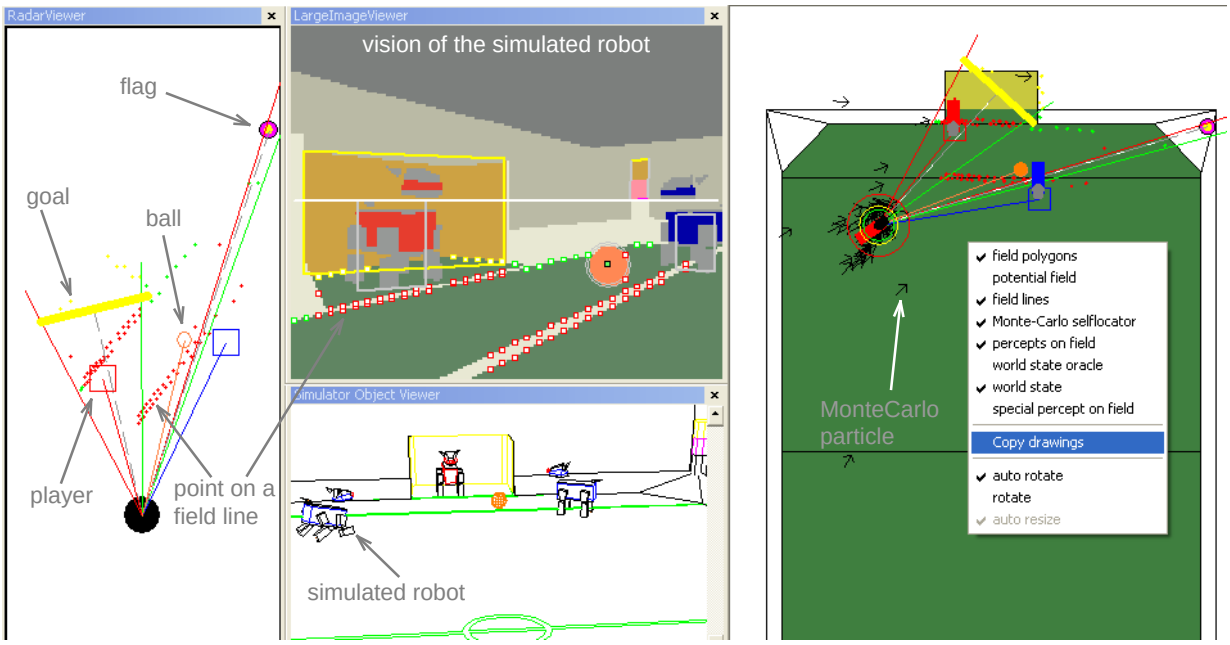


Figure J.11: RobotControl's Field View and the Radar Viewer Dialog

J.3.4 Time Diagram Dialog

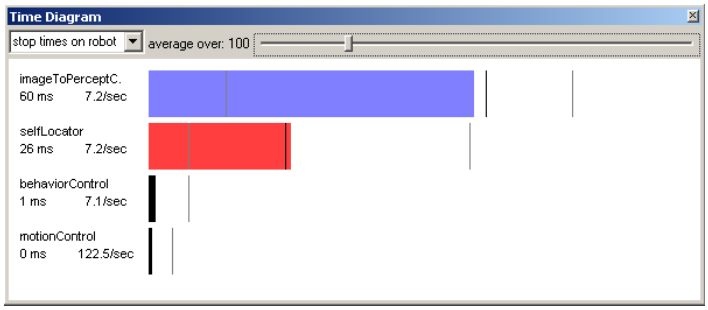


Figure J.12: The Time Diagram Dialog

The *Time Diagram Dialog* (cf. Fig. J.12) visualizes the times which different modules need for their execution in terms of bars. The values next to each bar show the measured time in milliseconds and the frequency (in times per second).

Times can be measured on the robot by selecting *stop times on robot*. If you are using the simulator (cf. Sect. J.4), the times can be measured on the computer by selecting *stop times local*. The option *view log files* displays the measured times of recorded logfiles. Since the times for the execution of the modules can vary very much from one measurement to the next one, the motion of the time-indicators can be smoothed by using average values. The average can be chosen between 2 and 500 measurements. Clicking the right mouse button in the dialog opens a dialog

in which the modules of interest can be selected. This dialog also offers the option to export the values to a file in a comma-separated format.

The design of the dialog varies, depending on its size and the number of the selected modules.

J.4 The Simulator

The simulator is a very powerful extension for RobotControl. Like SimGT2002 (cf. Sect. 5.1), it is based on *SimRobot*. The simulator offers a lot of possibilities to develop, test and debug new algorithms or alternative solutions for modules without using a robot.

Figure J.13: Toolbar of the Simulator

As shown in Figure J.14, all relevant objects for robot-soccer are included in the simulation: the field (including landmarks, goals, lines, etc.), players and the ball. Other objects (e. g. for challenges) can be added with ease. The created image depends on the position of the robot and also on the current angles of head and leg joints.

a) b)

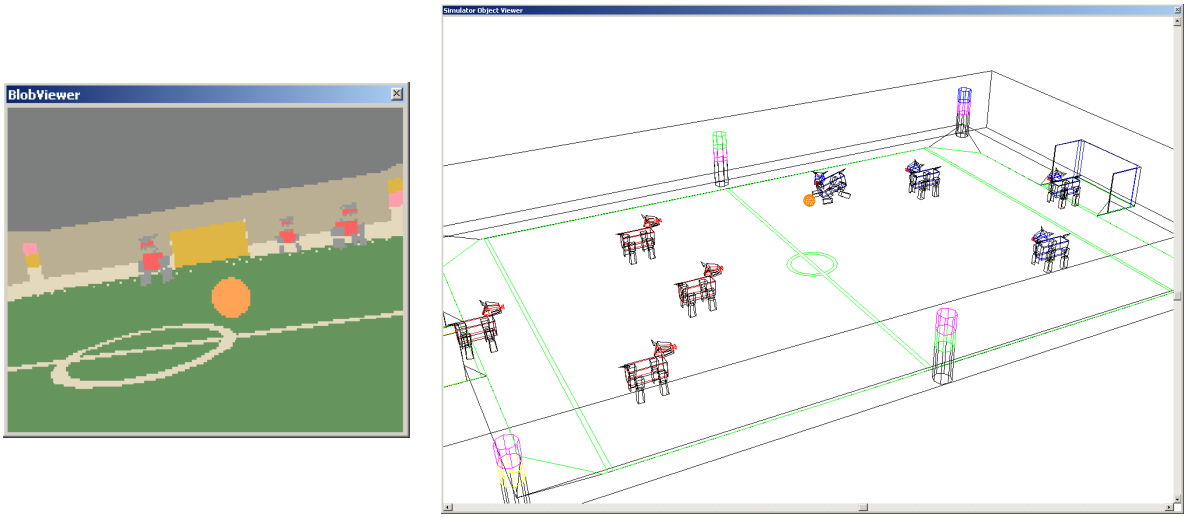


Figure J.14: a) Created image and b) Object Viewer of the simulator

For developing modules which result in movement of the robots, e. g. behavior, the object viewer (cf. Fig. J.14) shows the complete field with all simulated objects. It can be activated by the button *Object Viewer* of the simulator toolbar (cf. Fig. J.13). The vantage point of the observer is variable and can be changed by moving the bars under and beneath the displayed scene. The zooming level, detail level, and the perspective distortion can be adjusted by using the appropriate buttons in the toolbar. Besides these options, the toolbar contains buttons to start

and reset the simulation, and to force a step-by-step mode. The touch sensors at the back and the head of the robot can be “virtually pressed” by the buttons marked with an arrow at the according position. One of those buttons pretends the robot to be fallen aside.

A very helpful feature of the simulator is the oracle. It lets the robot know everything of its environment exactly. This can help to develop modules without being dependent on other modules. For example a behavior can be implemented and tested without a self-locator. The button *send oracle* activates this function.

Up to four robots of one team can be simulated at the same time. To send commands or receive information from a robot, connect to it by choosing it out of the list at *Robots* in the menu bar. This menu also includes the option to generate images for the connected robot or all simulated robots. Be aware, that generating images for all robots needs a lot of computing power. An entry in the status line of RobotControl shows the currently connected robot.

J.5 Debug Interfaces for Modules

For some of the Modules specific debug interfaces were developed.

J.5.1 Xabsl Behavior Tester

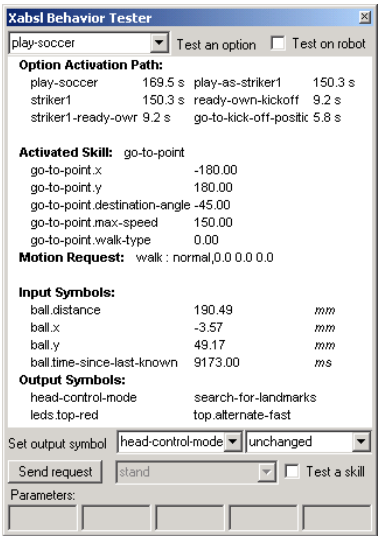


Figure J.15: The Xabsl Behavior Tester Dialog

The *Xabsl Behavior Tester* is a debugging interface to the *XabslBehaviorControl* module (cf. Sect. 3.6.2). One can view almost all internal states of the engine. *Options* and *Skills* and *Output Symbols* can be selected manually for separate testing.

With the checkbox *test on robot* in the right upper corner one can set if the behavior shall be tested on the robot or in the simulator.

In the combo box *test an option* an option can be selected to be the root option. The execution of the option tree starts then from that option. This allows testing single options separately.

At the bottom of the dialog with the checkbox *test a skill* the execution of the options can be switched off completely. Instead the skill that is selected in the combo box at the left bottom is executed with the parameters that are entered into the edit fields below.

At the top of the white area the *Option Activation Path* is displayed. The first column shows all the activated options. The second column shows the time how long these options are already activated. Then in the third column the active state of each option, and in the fourth column the time how long the state is already active, are displayed.

Then the *Activated Skill* shows, which skill is currently activated and its parameters. The *motion request* shows the motion request that resulted from the execution of the skills.

The *Input Symbols* section shows the current values of the selected input symbols. The selection, which symbols shall be displayed, can be done with the context menu of the dialog. Same with *Output Symbols*.

The dialog reads the file *xabsl-debugging-symbols.dat*. It is important that this file was generated from the same XML file as the intermediate code. Otherwise it may possibly cause program crashes.

At last, in the context menu exists an entry *Reload Files*. The dialog rereads the debugging symbols and sends the intermediate code to the robot and to the local processes, where the engine is newly created. That allows the testing of changes in the behavior without rebooting the robot or restarting *RobotControl*.

J.5.2 Motion Tester Dialog

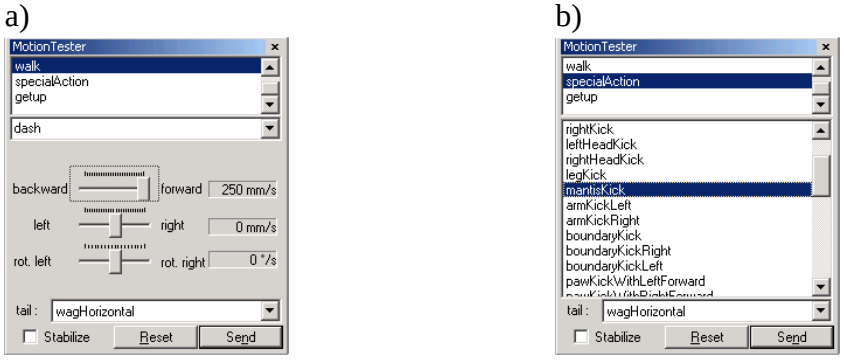


Figure J.16: The Motion Tester Dialog with a) walking and b) special actions

With the *Motion Tester Dialog* (cf. Fig. J.16) it is possible to send *MotionRequests* from *RobotControl* to the robot.

In the upper area of the dialog the different modes stand, getup, walk, or special action can be chosen. In walk mode the velocities in x and y direction and the rotation speed can easily be set by sliders. In special action mode the different special action (i. e. kicks or joy dance) can be chosen from an select box. In the lower area of the dialog the movement of the tail can be set.

Possible settings are, e. g., wag horizontal or parallel to ground (based on the evaluation of the gravity sensor).

To send the *MotionRequest* to the robot, you have to push the *send* button.

For another way to control the robots motion, the *Joystick Motion Tester* (cf. Sect. J.5.5) is available.

J.5.3 Head Motion Tester Dialog

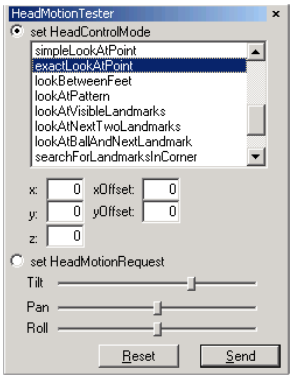


Figure J.17: The Head Motion Tester Dialog

The *Head Motion Tester Dialog* (cf. Fig. J.17) is handy to test the head control module, it is possible to send *HeadMotionRequests* or to set the *HeadControlMode* from *RobotControl*.

The desired *HeadControlMode* is selectable from a list of all available modes. Some modes require additional parameters (i. e. the coordinates of a point to look at). These can be set in the appearing input elements.

In *HeadMotionRequest* mode, the desired joint values can be set by sliders.

J.5.4 Mof Tester Dialog

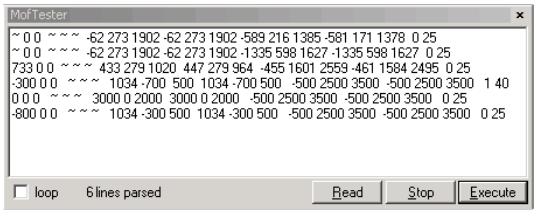


Figure J.18: The Mof Tester Dialog

The *Mof Tester Dialog* (cf. Fig. J.18) is used to write and test new motions. Motions are specified in a description language (cf. Sect. 5.4). Joint data lines from these descriptions may be entered into the input field of the dialog and can be sent to the robot via the wireless network at runtime.

For this dialog to work it is necessary that the module *DebugMotionControl* is running on the robot instead of the default motion control module. The debug module will not execute normal motion requests from behavior control but rather wait for debug messages sent from the *MofTester* dialog. It is activated on the robot by switching module solutions with the *Settings Dialog* (cf. Sect. J.2.3).

The *execute* button parses the input field for lines containing joint data information and sends the sequence to the robot. If the *loop* checkbox is activated when pressing *execute* the sequence will be executed repeatedly.

The *stop* button stops any sequence currently being executed.

The *read* button provides a very handy tool when creating new motions. It reads the robot's current joint angles from sensor data and puts them into a new line in the input field. This is extremely useful in combination with the stay-as-forced motion mode the *DebugMotionControl* module provides while not executing joint data sequences. In stay-as-forced mode all motors are controlled with feedback from sensor input, and therefore they always maintain their current position. In this mode joints may be moved manually and the resulting joint angles can be read into the *Mof Tester Dialog*.

J.5.5 Joystick Motion Tester Dialog

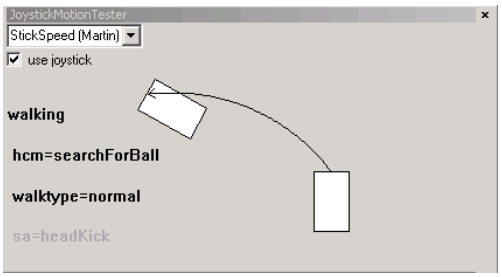


Figure J.19: The Joystick Motion Tester Dialog

The *Joystick Motion Tester Dialog* (cf. Fig. J.19) is used to control a robot that is connected via the wireless network by joystick. Using such an analog input device is, e. g., useful for testing new walking engines or parameters or just walking stability. The primary aim is not a complete remote control, but only moving the robot around (and optional moving its head) while all other modules keep running autonomously.

This dialog is only tested with an MS Sidewinder Precision 2 USB joystick, but should work with any joystick providing three axes, an accelerator, and eight buttons. After the joystick has been attached, the *use joystick* box has to be checked to start using this dialog. If no joystick seems to be connected, the dialog will refuse to work.

The joystick controls the walking engine of a robot connected with RobotControl via the wireless network. The direction the robot should move to according to the state of the joystick is visualized in the dialog. Red text in *Joystick Motion Tester Dialog* signals that current changes

have not been executed yet, black text shows the actual states or commands, and grey text visualizes states or commands that were sent last but are inactive at the moment.

The buttons 1 to 4 can be used for different kicks, button 7 to start the *getup* motion and button 5 to switch from walking control to head control. As long as button 5 is pressed, the joystick will control the head instead of the walking. That will be visualized by the dialog, too.

To ease the use of the dialog for different people, different schemes for joystick control were implemented. One is called *StickSpeed*, in which the walking speed is completely controlled by the three axes of the joystick, the accelerator is used to switch between head control modes, the walking type can be changed with button 8, and all special actions can be generated by holding button 6 pressed and moving the accelerator.

Another scheme for *Joystick Motion Tester Dialog* is called *AcceleratorSpeed*: the accelerator is used to control the forward walking speed, the axes are only used for sideward speed and direction, and walk types can be changed with button 6.

Commands will only be sent via the wireless network if they differ from the previous command and at most every 300 ms, because the router throughput and especially its response times do not allow much more. So whirling around the joystick will definitely not encourage the robot to do the same.

J.6 Color Calibration

J.6.1 The Color Table Dialog

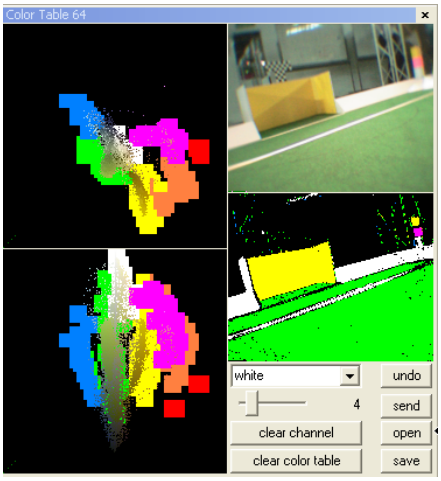


Figure J.20: The Color Table Dialog

The *Color Table Dialog* (cf. Fig. J.20) is used to create color tables for image processing. The current image from the simulator, a logfile or the robot's camera (via WLAN) is shown top right (cf. Fig. J.20). Beneath the same image can be seen, segmented with the current color table. Choosing a color class and clicking with the left button on a pixel in one of the two images will

assign the color class to the $4 \times 4 \times 4$ cube in YUV color space according to the color value of the pixel. The result takes effect immediately and the segmented image will be updated. Clicking with the right button on a pixel will remove the according color class assignment. There are also functions to undo the last assignment, to clear all assignments for a whole color class and to reset the complete table.

On the left side of the window, two sights of the current color table in YUV color space are shown. They have no additional functions.

After having assigned all needed colors, the table can be saved to a file. The GT2002 vision modules need a file named *coltable.c64*. It is also possible to load an existing file and to modify it.

J.6.2 HSI Tool Dialog

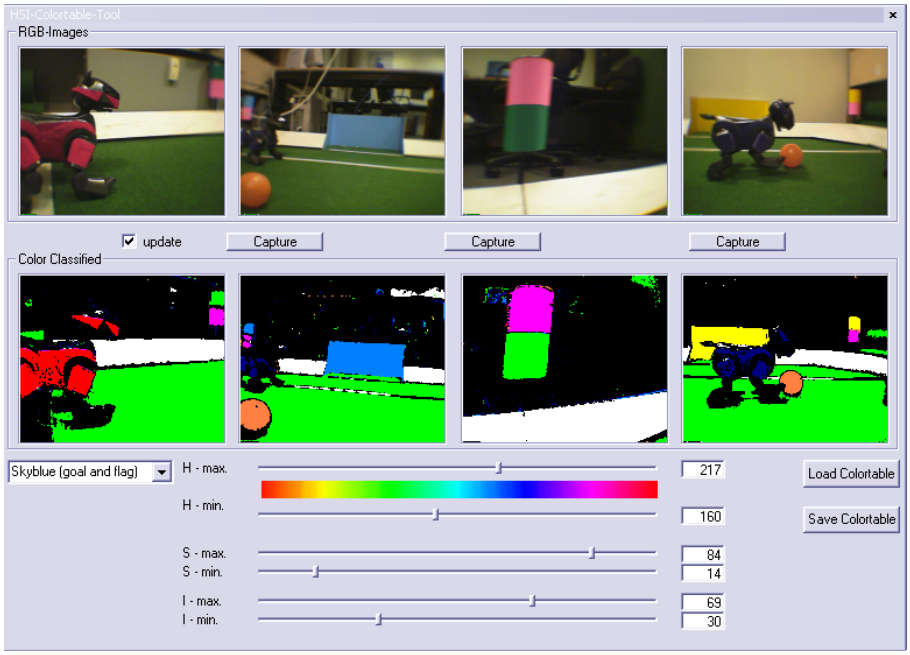


Figure J.21: The HSI Tool Dialog

This tool uses the HSI color space to create a color table. It allows the user to specify the minimum and maximum values for hue, saturation and intensity for each color class using sliders and gives an instant feedback by real time color classification of four images at the same time. These images are directly taken from the memory of *RobotControl* and can be held as long as needed. It is also possible to update an image simultaneously with the one in the memory of *RobotControl*. The tool saves the color table both in HSI and YUV format. By selecting an image there is another dialog with a zoomed view (cf. Fig. J.22) of it and the belonging color classified image. In this dialog the color class can simply be edited by selecting single pixels. There is also an undo function for several steps.

This HSI approach leads to good results very quickly by defining big sectors in the color space and it is more tolerant against changing light conditions. The tool can be combined with the other color table tool using YUV color space. First, the HSI tool is used to quickly generate a color table and the YUV tool can be employed for fine tuning if required.

Dialog structure. The main dialog (cf. Fig. J.21) consists of two parts. In the upper half are spaces for four RGB images and below them the corresponding color classified images will be displayed. Below each space there are buttons for capturing an image and at the left is a check box for an automatic update of the image above it.

In the lower half of the dialog are most of the controls. There is a combo box with entries for all color classes used by the image processing module, a button for loading HSI color tables, a button for saving the HSI and the corresponding YUV color table and six sliders for determining the range of the selected color class in the HSI color system. An HSI color class consist of a minimum and a maximum value for hue (H), saturation (S), and intensity (I).

Main dialog. With the *capture* buttons in the upper half of the main dialog (cf. Fig. J.21) the actual image that *RobotControl* has in memory can be saved. It can be done for four images. When the box before *update* below the left place is checked, this image is updated automatically, when *RobotControl* gets a new image from the robot or a log file. The color classified image will be updated automatically when you change the range of a color class.

With the combo box at the lower left of the dialog the color class to be edited can be selected. The sliders show the minimum and maximum values of this color class. The ranges for H, S, and I can be modified by changing the positions of the sliders. While moving a slider the color classified image is permanently updated.

The range of values for the hue of a color class goes from 0 to 360, and for saturation and for intensity from 0 to 100. Because the value for hue in HSI color system lies on a circle, it is possible that the maximum value for this range is smaller than the minimum value. For a red color, e. g., the minimum of the hue range could be 350 and the maximum could be 15. For saturation and intensity the minimum value should be below the maximum value.

The color table can be saved by pressing the *Save* button. It appears a file dialog where the destination and the name of the color table can be selected. The suffix *.hsi* will be added automatically. A YUV color table converted from the HSI color table will also be saved with the same name and the suffix *.c64*.

An HSI color table can be load by pressing the *Load* button. It can be selected in the appearing dialog. It is only possible to load HSI color tables. YUV color tables are not supported yet by the HSI tool.

Zoom dialog. By performing a click with the left mouse button on one of the RGB images, another dialog window with a zoomed view (cf. Fig. J.22) of the selected image and the belonging color classified image will appear. In this dialog it is possible to improve the precision of the ranges for color classes by selecting single pixels. At the lower left the combo box with the color classes to edit is located. There is also an *Undo* button for the last six changes.

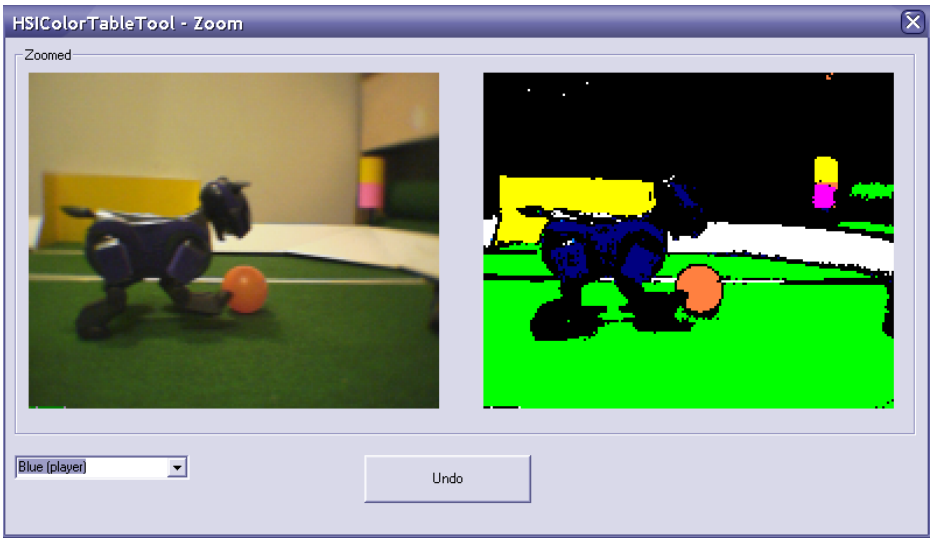


Figure J.22: The HSI Tool Zoom Dialog

By pressing the left or the right mouse button on a pixel in one of the zoomed images the selected color class will be modified. If the left mouse button has been pressed and the color of the selected pixel lies outside the color class, the range will be enlarged until the values for hue, saturation, and intensity are inside this range. If the right mouse button has been pressed and the selected pixel lies inside the range of the color class, the range will be reduced until the values for hue, saturation, and intensity are outside of it.

By selecting single pixels, it is possible to determine which color should belong or not belong to the chosen color class. Thus precision of the class can be improved.

J.6.3 Camera Toolbar

Figure J.23: The Camera Toolbar

The *Camera Toolbar* (cf. Fig. J.23) is used to set the parameters that are provided by the robot's camera. These parameters are set with the combo boxes. White balance may be set to indoor, outdoor, or FL mode. Shutter speed may be selected from slow, medium, or fast. Finally low, medium, or high camera gain can be chosen.

The *send to robot* button sends the selected parameters via the wireless network to the robot. The new settings are applied immediately. When viewing the camera pictures with the image viewer (cf. Sect. J.3.1) the effects of different camera settings can be observed.

The *save* button writes the settings to a file named *camera.cfg*. This file is loaded at the start of RobotControl to initialize the *Camera Toolbar* with its contents. But more important this file

is copied to the memory stick and loaded when booting the robot. The settings from this file are used on the robot unless different parameters are sent with the toolbar.

J.7 Other Tools

J.7.1 Debug Message Generator Dialog

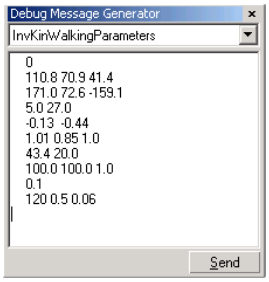


Figure J.24: The Debug Message Generator Dialog

The *Debug Message Generator Dialog* (cf. Fig. J.24) is used to generate less commonly used debug messages for which no special dialog exists. The combo box is used to select what type of debug message is generated. When pressing the *send* button a message will be generated by parsing the input in the text field. The dialog may be extended easily by adding the code to parse the text input into a debug message. Therefore it allows generating new debug messages with *RobotControl* without having to create a new dialog. It is used, e. g., to send a new set of parameters to the walking engine. By this it is possible to change the walking style at runtime and therefore develop new walks very quickly. Another application is to test playing acoustic messages on the robot by sending the corresponding debug message.

J.7.2 Test Data Generator Dialog

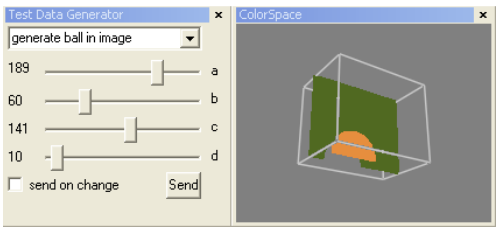


Figure J.25: The Test Data Generator Dialog generated an image which is shown by the Color Space Dialog

The *Test Data Generator Dialog* (cf. Fig. J.25) can generate any type of data and put it into a message queue. At the moment two different *schemes* exist. One generates an image with a green

background and an orange circle. The size and position of the orange circle can be modified with the sliders. This was very helpful for testing the ball preceptor. The other scheme generates an image that demonstrates the YUV color space. The first slider modifies the Y value. The image shows all U and V values.

J.7.3 Evolver Dialog

With the *Evolver Dialog*, new gait patterns for the robot can be developed by applying an Evolutionary Algorithm (EA). This class of optimization algorithms uses populations of individuals, in which each individual (in this context) represents a different gait pattern. In the course of the algorithm, the fitness of each individual is tested on the robot by executing tournaments (i.e. foot-races). The better ones are chosen to constitute the next population, the other patterns are discriminated. A good introduction to EA is given by [13]. The *Evolver Dialog* allows specifying the parameters for the algorithm as well as executing the evolution process.

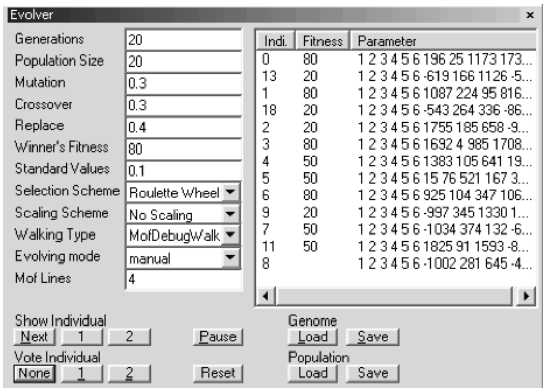


Figure J.26: The Evolver Dialog

The dialog window, as illustrated in figure J.26, is arranged in two columns. The left one shows the parameters for the algorithm, while the right part illustrates parameter sets of actual individuals for debug purposes. The set of buttons at the bottom of the window can be used to run and evaluate tournaments.

The parameters *Generations* and *Population Size* are used to define the number of generations and the number of individuals in each generation. In practice, $0.5 \cdot$ *Generation* $\cdot$ *Population Size* tournaments have to be run to finish a complete evolution process. The experiments conducted show, that a population size of 10 to 20 seems to be a reasonable choice [6, 7]. The *Mutation* parameter is a floating point variable the value of which can be chosen among numbers ranging from 0 to 1. Exemplary, by setting *Mutation* to 0.3, about 30 % of the parameters of each individual will be mutated. The *Crossover* parameter affects the generation of a new population. If *Crossover* is set to 0.3, then 30 percent of the entire population consists out of individuals that are generated by the crossover function. The rest (70%) are selected survivors of the last generation. The *Replace* parameter defines the probability that an individual is replaced by a randomly generated or predefined one when a new generation is built up. If *Replace* is set to 0.4, then 60% of the

new generation is generated out of individuals of the former population. The value *Winners Fitness* is assigned to the winner of a tournament. The looser's fitness is set to $100-$*Winners Fitness*. If no winner can be determined, both individuals get a fitness value of 50. A value more than 50 and up to 100 is a reasonable choice for the *Winners Fitness*. *Standard Values* determines, how many individuals of a new population are taken from *T:\GT2002\Config\genome.gen*. There, a standard individual can be defined. When *Standard Values* is set to zero, no individual is initialized from predefined value. If *Standard Values* is set to 0.1 and *Replace* is set to 0.4, then $10\% \cdot 40\% = 4\%$ of all individual are initialized by standard values[1].

Selection and Scaling Scheme defines, how the fitness value is scaled and used to discriminate between the different individuals. The different methods are explained in [13]. *Walking Type* enables to choose between different walking engines implemented in GT2002. The actually implemented walking engines and their parameters are explained in section 3.7.1. The *Evolving Mode* parameter can be used to select between different kinds of tournaments. Actually, automatic and semi-automatic fitness estimation modules are developed [6, 7]. Nevertheless, this methods do not perform with full functionality under all side-constraints. Therefore, *Evolving Mode* can only be set to "manual" in this release. The last parameter *MofLines* is only useful when *WalkingType* is set to *MofDebugWalk*. It determines the number of parameters (genomes) each individual consists of (cf. Sect. 3.7.1 for details.)

The buttons at the bottom of the window provide the following functionality: *Next*, *1*, and *2* of the *Show Individual* row send specific gait patterns to the robot, such that the user can evaluate their fitness. Usually, only the *Next*-Button is used to show the following walking pattern. In some cases, the user might be unsure, then he or she can recall the first *1* or the second *2* gait explicitly. A double-click on an individual from the list (right column) starts the implicated gait pattern automatically, too. The *Pause* button can be used to stop an actually executed walk immediately.

The quality of the individual can be evaluated by pressing one of the buttons *None*, *1*, or *2* in the *Vote Individual* row. If the first shown pattern is better, *1* has to be pressed, if the tournament results in a tie, the *None* button must be pressed. *Reset* starts and restarts the whole evolution process. To save intermediate results, the *Load* and *Save* buttons can be used to save a selected *Genome* or an entire *Population*, respectively.

[1] For the first generation, *Standard Values* defines, how many individuals of the whole *Population Size* are initialized by a predefined individuum.

References

[1] Simrobot homepage. http://www.tzi.de/simrobot.

[2] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:832–843, 1983.

[3] R. Brunn, U. Düffert, M. Jüngel, T. Laue, M. Lötzsch, S. Petters, M. Risler, T. Röfer, K. Spiess, and A. Sztybryc. Germanteam 2001. In *RoboCup 2001*, number 2377 in Lecture Notes in Artificial Intelligence, pages 705–708. Springer, 2002.

[4] H.-D. Burkhard. Mental models for robot control. In *Proceeding on Dagstuhl Seminar "Planbased Control of Robotic Agents"*. Springer LNAI-Series, 2001.

[5] H.D. Burkhard, J. Bach, R. Berger, B. Brunswieck, and M. Gollin. Mental models for robot control. In M. Beetz et al., editor, *Plan Based Control of Robotic Agents*, number 2466 in Lecture Notes in Artificial Intelligence. Springer, 2002. To appear.

[6] Ingo Dahm and Jens Ziegler. Adaptive methods to improve self-localization in robot soccer. In *RoboCup Symposium Fukuoka*, 2002.

[7] Ingo Dahm and Jens Ziegler. Using artificial neural networks to construct a meta-model for the evolution of gait patterns of four-legged walking robots. In *to appear in International Conference on Climbing And Walking Robots (CLAWAR)*, November 2002.

[8] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, 1997.

[9] D. Fox, W. Burgard, F. Dellaert, and S. Thrun. Monte Carlo localization: Efficient position estimation for mobile robots. In *Proc. of the National Conference on Artificial Intelligence*, 1999.

[10] John F. Hart et al. *Computer Approximations*. SIAM Series in Applied Mathematics. Wiley, 1968.

[11] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. In *IEEE Transactions on Systems, Man, and Cybernetics*, volume SSC-4:2, pages 100–107. Institute of Electrical and Electronics Engineers, Inc., 1968.

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

[13] J. R. Koza. *Genetic Programming*. MIT Press, Cambridge, MA, 1992.

[14] S. Lenser and M. Veloso. Sensor resetting localization for poorly modeled mobile robots. In *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, 2000.

[15] F. K. H. Quek. An algorithm for the rapid computation of boundaries of run length encoded regions. *Pattern Recognition Journal*, 33:1637–1649, 2000.

[16] Th. Röfer. Strategies for using a simulation in the development of the Bremen Autonomous Wheelchair. In R. Zobel and D. Moeller, editors, *Simulation-Past, Present and Future*, pages 460–464. Society for Computer Simulation International, 1998.

[17] D. Schulz, W. Burgard, D. Fox, and A.B. Cremers. Tracking multiple moving targets with a mobile robot using particle filters and statistical data association. In *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, 2001.

[18] T. Naruse T. Takahashi Y. Nagasaka, K. Murakami and Y. Mori. Potential field approach to short term action planning in robocup f180 league. In *RoboCup 2000*, Lecture Notes in Artificial Intelligence. Springer, 2001.