Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

MASTERTHESIS

Towards Using ROS in the RoboCup Humanoid Soccer League

vorgelegt von

Marc Bestmann

MIN-Fakultät

Fachbereich Informatik

Technische Aspekte Multimodaler Systeme

Studiengang: Informatik

Matrikelnummer: 6209584

Erstgutachter: Prof. Dr. Jianwei Zhang

Zweitgutachter: Dr. Norman Hendrich

Abstract

Sharing software modules between teams in the RoboCup Humanoid League is difficult since all teams use different frameworks. This leads to reimplementation of software which slows the research process. A common framework for the league would resolve this. Therefore, this thesis proposes a ROS-based architecture which is defined by a set of ROS messages. The teams can decide on the specific implementation of nodes since the messages provide the interfaces. Different tools, especially for visualization, are implemented to be used in conjunction with this architecture. Furthermore, the robot control module of the team Hamburg Bit-Bots is transferred into the new framework to show its usability. The architecture is compared to others and its performance is evaluated.

The presented architecture makes sharing software modules easier and can thereby accelerate the research in the RoboCup Humanoid League. Furthermore, the entry of new teams is simplified, due to the availability of shared modules.

Zusammenfassung

Der Austausch von Softwaremodulen zwischen Teams der RoboCup Humanoid League ist schwierig, da alle Teams unterschiedliche Frameworks benutzen. Dies führt zu Neuimplementation von Software, wodurch der Forschungsprozess verlangsamt wird. Ein gemeinsames Framework für die Liga könnte dieses Problem lösen. Daher stellt diese Masterarbeit eine auf ROS basierte Architektur vor, welche durch einen Satz von ROS Nachrichten definiert ist. Verschiedene Werkzeuge die in Verbindung mit dieser Architektur benutzt werden können, besonders für Visualisierungen, wurden implementiert. Außerdem wurde das "robot control module" der Hamburg Bit-Bots in das neue Framework transferiert um dessen Benutzbarkeit zu zeigen. Die Architektur wurde mit anderen verglichen und ihre Leistungsfähigkeit wurde evaluiert.

Die präsentierte Architektur erleichtert den Austausch von Softwaremodulen und kann dadurch die Forschung in der RoboCup Humanoid League beschleunigen. Außerdem wird der Einstieg für neue Teams in die Liga erleichter, da sie die geteilten Module benutzen können.

Contents

List of Figures

1 Introduction

The RoboCup Foundation was founded to enhance the progress in autonomous robotics by proposing competitions with standard scenarios for research. The most important league is the humanoid soccer league, where robots compete in a modified rule set of human soccer to achieve the goal of winning against the FIFA world cup champions with a team of robots in the year 2050. This goal gives the competing teams a motivation and the game is a basis for comparison of different approaches.

Even though RoboCup is a competition, most of the teams publish their code and hardware to accelerate the research process. But due to the complex software systems running on the robots, the effective transfer of code between teams is almost nonexistent. Therefore this thesis proposes a way to ease code exchange by a transition to the Robot Operating System (ROS), an open source framework for robot programming. Furthermore, standard messages will be proposed for the specific field of robot soccer. To validate the effectiveness of this approach, the current software of the Hamburg Bit-Bots will be transferred to ROS and its performance will be tested.

The structure of the thesis is as following: First, the topic is motivated in section 1.1, related work is discussed in section 1.2 and the concrete goals are stated in section 1.3. Then the fundamentals are explained in section 2. Afterward, the used hardware and software components are explained in section 3. The approach is explained in section 4 and its implementation examined in section 5. An evaluated of this implementation is presented in section 6. Finally, the thesis is concluded in section 7.

1.1 Motivation

While the idea of RoboCup Soccer, with a mixture of competitions and exchange, is a good approach in theory to accelerate the research, some problems still remain. One of the biggest is the sharing of software. Many teams share their code base using services like GitHub, but due to their monolithic architectures, specialized frameworks and hardware specific code, actually using parts of other team's software is difficult. The resulting barriers lead to a waste of time due to reprogramming of existing software. An example is the de facto standard motor, the Robotis Dynamixel, which is used by almost all teams, but every team has its own code for control.

The lack of usable modules generates a high entry threshold for new teams because they have to implement many algorithms from scratch in order to be able to perform the basic tasks needed to participate. Therefore the RoboCup Federation is trying to solve these problems [Gerndt et al., 2015].

One way to simplify code exchange is using a common framework or middleware. The Robot Operating System (ROS) [Quigley et al., 2009] is a good choice because it has a large community, is already widely used in research and supports many different robots [Gerkey, 2015]. It also provides a large set of standard libraries and message types, as well as simulation and debug tools.

In other RoboCup leagues, especially RoboCup Rescue, this is already more widespread (see section 1.2). There are also a few teams in the Humanoid Soccer League which are already using it (cp. 1.2.1). This is a first step in the right direction because the structure is less monolithic due to the split into different ROS-nodes, but the problem of sharing code still persists. Even if two teams have each one node for the same task, e.g. ball detection, the subscribed and published messages are most likely not the same. This is understandable because there exist no standards for these messages. The messages specified by ROS are mostly regarding general sensor input, actuator control, and navigation. In RoboCup Soccer exist different kinds of information that are not commonly communicated in other contexts. Therefore no messages exist which are standardized by ROS. Proposing an additional set of messages can be useful to make nodes substitutable with others. It would also increase the ability to compare the performance of two nodes with each other.

Another point, which will increase its importance in the future, is the collaboration of robots of different teams. The road map of the RoboCup Humanoid League [11] shows that the number of playing robots will increase in the future from currently four to the normal player count for soccer, eleven. An even higher number of robots is needed when counting substitute players. This will force teams to join their robots into a mixed team. Doing this with completely different software frameworks is theoretically possible, but not practical.

The Hamburg Bit-Bots are a team of students which has been participating in the RoboCup Humanoid League since 2011. Until now they did not use ROS or any other framework for their software and based everything on a self-programmed mixture of shared memory inter process communication (IPC) and a modified blackboard system. Transferring their software to an ROS architecture does not only serve as an example and test for the proposed use of ROS in the humanoid league, but also results in a collection of open source packages which can be used by other teams. Last but not least, the architecture change will also ease the work of the Hamburg Bit-Bots, because a modular system is simpler to maintain. Especially in the context of having only students working on the software, who don't have a finished education and who are working on this project for a comparably short time. The modularization enables them to work on one part of the software without having knowledge about the others.

The robots that are currently used by the Hamburg Bit-Bots, shall be used to validate the proposed architecture in terms of usability and performance. The Minibot, created by the Hamburg Bit-Bots, is an upscaled version of the Darwin-OP [Ha et al., 2011]. It was needed because the Darwin-OP will soon no longer be allowed to participate due to an increase of the minimal robot size in the league's rules [11]. The Minibot is a good example platform as it is similar to most robots in the league. To put it in a nutshell, the transfer to a ROS-based architecture shall modularize and parallelize the software, ease the exchange between teams, lower the entry difficulties of new teams and enable the use of standard messages, libraries, and simulators. Together with the open source hardware of the Minibot, the software framework shall provide an easy to access platform.

1.2 Related Work

When looking at the number of publications corresponding to RoboCup and ROS, it is clear to see that ROS is becoming more and more popular. The first paper about ROS was written in 2009 [Quigley et al., 2009], since then its popularity increased enormously [Gerkey, 2015]. While the RoboCup-related publications were stagnating during this time, the number of ROS-related ones already excelled them in 2015 (see fig 1.1).

1.2.1 Publications in RoboCup

The first publication in the RoboCup proceedings concerning ROS, about an external system for finding a ground truth, was in 2011 [Khandelwal and Stone, 2011]. The year after, only one other paper was using ROS and only for comparison with their result [Ruiz et al., 2013]. In 2013, the number rose to 13 including more impacting papers. But until 2016, the number dropped back to zero. This doesn't mean, that the teams stopped using ROS again. It shows that there was a wave when teams started to use ROS in different leagues which resulted in publications about this integration. Afterward, no further papers were written about it, but the usage was still high in 2016 (cp. figure 1.6).

Rescue League

The Rescue League started early to advertise the use of ROS with the goal of an easier exchange between teams. In 2010 and 2011, they organized workshops for the league and in 2012 even a complete summer school [Kohlbrecher et al., 2012]. This resulted in a release of a corresponding ROS meta package [Kohlbrecher et al., 2013] by team Hector. A package which was used and kept up to date. Additionally, in 2013, the ROS support for the "Unified System for Automation and Robot Simulation" (USARSim), was added by a RoboCup team [Kootbally et al., 2013]. In 2014, team Hector stated that using open source software and ROS helped to achieve reliable autonomy in search and rescue [Kohlbrecher et al., 2014].

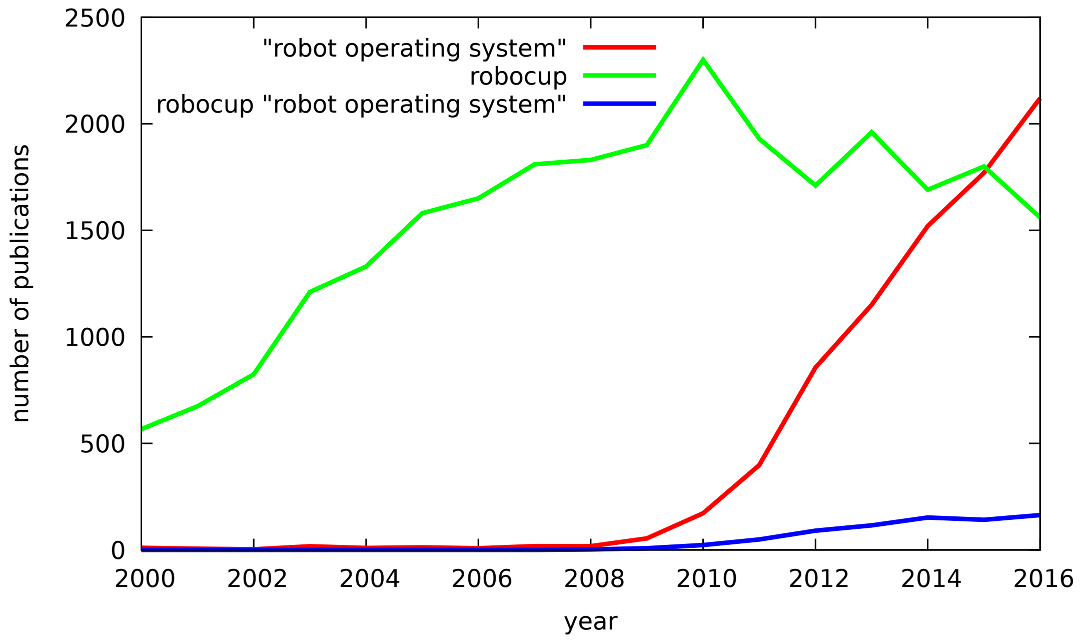


Figure 1.1: Distribution of publications containing the three terms: ”robot operating system”, robocup, robocup ”robot operating system”. The data is based on Google Scholar and was generated by an external tool [1]. The graph shows a steadily increasing number of publications per year using the term ”Robot Operating System”. Publication numbers concerning RoboCup in general were relatively stable over the last ten years but publications concerning ROS and RoboCup have risen. This indicates an increasing interest in ROS inside the RoboCup community.

Standard Platform League (SPL)

The most used framework in the Standard Platform League is based on the code of the B-Human team which is published annually since 2008. This is possible because all teams use the same robot, the NAO [Gouaillier et al., 2008]. B-Humans architecture consists of modules which perform computations based on data which is provided by *representations*, see figure 1.2. These are C++ classes that are stored in a blackboard. Each process has its own blackboard and representations can be shared between processes.

This architecture was compared to ROS in 2013 [Röfer and Laue, 2013]. The authors stated that the parallel architecture of ROS was not useful on the single-core NAO robot. As the custom B-Human framework was already spread in the league, they saw no need for changing towards ROS. In the same year, another paper was published about the transfer towards ROS in the SPL [Forero et al., 2013]. The authors wrote that it is too hard for new teams to adapt the code from B-Human, leading to a reuse of the software stack with almost no own modifications.

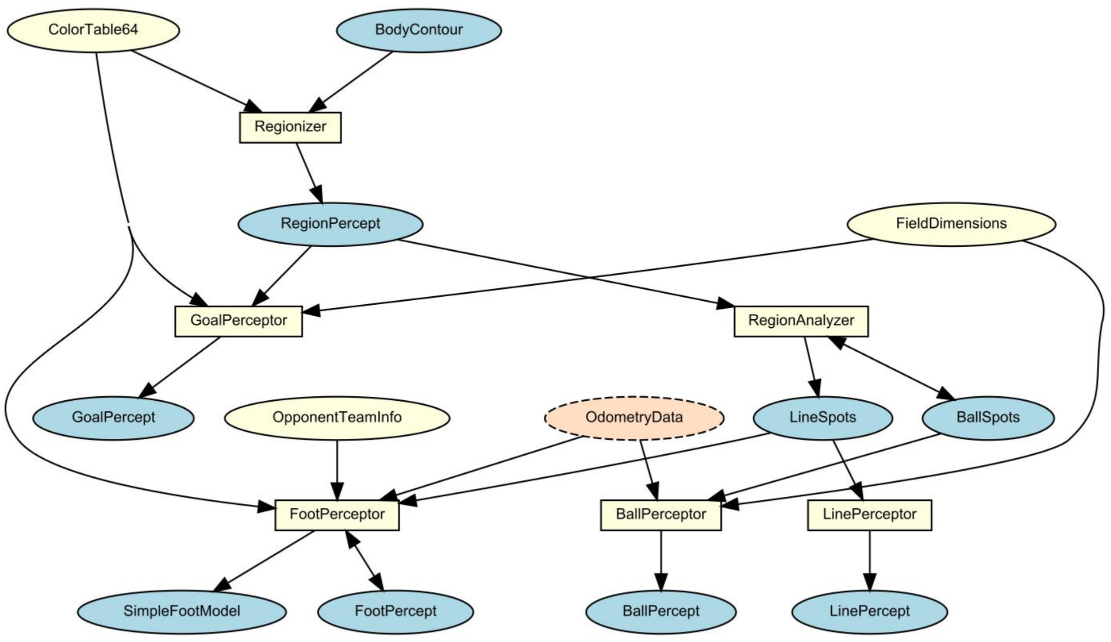


Figure 1.2: Example of an architecture for a robots vision using the B-Human framework [Röfer and Laue, 2013]. The ellipses are representations which provide data and the rectangles are modules which perform computations. The architecture looks similar to ROS but the representations are not handled peer-to-peer but using a blackboard system.

The authors stated that the use of ROS could allow:

- ”to share easily software module between teams,
- to encourage the development of very specialized solutions which can be shared among the teams,
- to facilitate the incorporation of new teams in the league,
- to attract new students and new researchers to the leagues teams,
- to encourage the specialization of some teams in some robot control areas (e.g. motion control or perception), and
- to facilitate the comparison and benchmarking of specific software modules.”[Forero et al., 2013]

Furthermore, the authors defined messages for the migrated motion component of B-Human. Unfortunately, they did not define general messages which would be typically used in RoboCup.

In 2014 a new non-ROS-architecture for the SPL was proposed [Mamantov et al., 2014]. The authors compared it to frameworks used at this point in the SPL, stating that most of the teams are using a modified

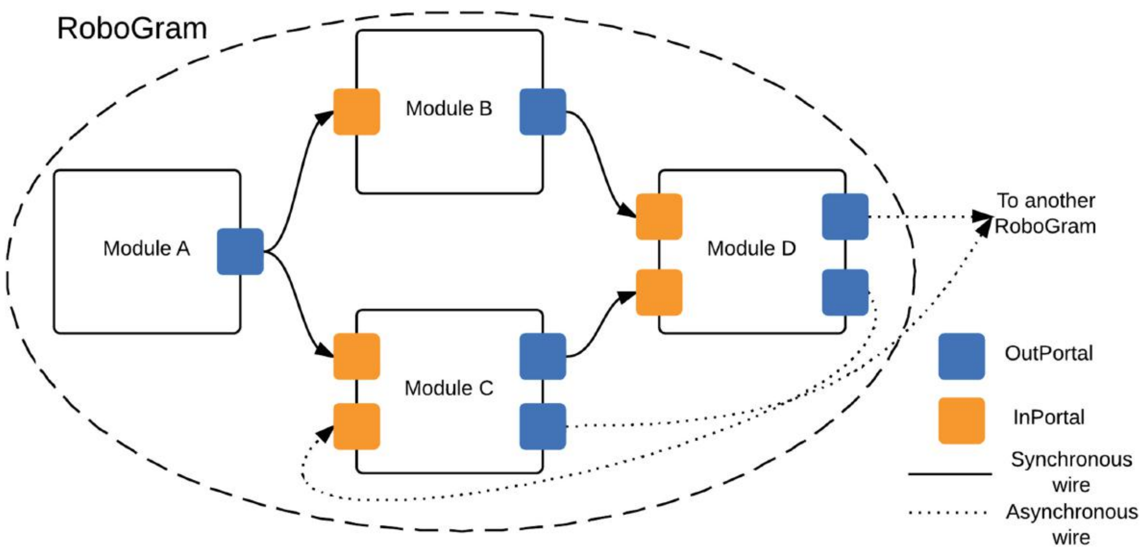


Figure 1.3: Example of a RoboGram with four modules [Mamantov et al., 2014]. Each module can specify `OutPortals` and `InPortals` which can connect to other modules synchronously or asynchronously to transfer data. Each RoboGram is run in one thread of execution and calls its modules successively.

blackboard design. They wanted to change this but not by using ROS due to the complications of running it on the NAO robot. Similar to the mentioned paper by B-Human, the reason for not using ROS was mainly the hardware of the NAO robot. Their proposed architecture was very close to the publisher-subscriber model of ROS. They are using `OutPortals` and `InPortals` which form synchronous or asynchronous connections between modules, as shown in figure 1.3. The transferred data is also specified by messages. Multiple modules form together a RoboGram which has one thread of execution and is sequentially calling a `run()` method on its modules, in an order depending on their connections.

Humanoid League

The humanoid league has no commonly used robot platform like the SPL, but during the last years, many teams in the *Kid-Size* used the Darwin-OP. Therefore a framework for this robot, made by Team DARwIn and UPennalizers, was relatively widespread [McGill et al., 2013]. But as the use of the Darwin-OP decreased and Team DARwIn stopped participating in the league, the framework was not further maintained.

In 2013, the *NimbRo-OP* was released, first with a framework based on the Darwin, but later with full ROS support [Schwarz et al., 2013]. This was the beginning of a development towards bigger robots in the *Kid-Size* and the first step towards using ROS. Unfortunately this robot still had some flaws in hard- and software. While the framework was released on GitHub [12] and further updates were promised [Allgeuer et al., 2013], there were no further commits after one month. Furthermore, the package infrastructure was done badly. The package.xml files are not completed, the packages are not registered in the ROS wiki and all packages are placed in one GitHub repository. This hinders contributions to specific packages. Also, the fact that there were no new commits to the git, is not inviting for contribution.

On the hardware side, the robot was difficult to build by yourself, mainly due to the used carbon fiber which is difficult to manufacture but also due to the lack of documentation. The CAD files were published as `.step` in another GitHub repository [12], but similar to the software git, there were no further commits after one month, no documentation and only a promise that it will be added later. These problems lead to the fact that no other teams used the NimbRo-OP in RoboCup. But it greatly influenced the design of other platforms with a ROS framework. The team WF Wolves used it as a base to build an own robot with another ROS-based architecture [Anders et al.,]. Furthermore, the NimbRo-OP inspired the MU-L8, a robot for RoboCup with a ROS framework [Stroud et al., 2013] [Stroud et al.,]. Unfortunately, neither the CAD files nor the software can be found on the Internet. The corresponding page of the university was not updated since 2014 [13].

In 2015, Robotis released an ROS based framework [14] for the Robotis-OP (formerly known as Darwin-OP). Fortunately, they split the code into different packages, but they didn't put it on the ROS wiki. Furthermore, the release came too late to be used in RoboCup, since teams already started to use other robots.

In 2015, the *igus Humanoid-OP* was released by the constructors of the NimbRo-OP. Based on the same skeleton and mostly the same hardware, it is principally a change in manufacturing method towards 3D printing [Allgeuer et al., 2016]. Unfortunately, the same mistakes already done with the NimbRo-OP were made again. The CAD files are only available as .step without documentation [15], the software is not documented in the ROS wiki and only in one GitHub repository [16]. Like in the NimbRo-OP repository, no active development or contribution can be observed.

All these robots in the humanoid league are using the same motors (Robotis Dynamixel Line), the same motor communication protocol, the same motor controller board and ROS, but still, no common code is used. There exists even an open source ROS package for controlling the Dynamixel motors, which is also not used.

1.2.2 Publications Outside RoboCup

Naturally, there was work done on the subject of humanoid robot control using ROS outside the RoboCup. Still, most publications are rather concerning wheeled robots. While certain problems, e.g. computer vision, remain the same and results from wheeled robots can be transferred, some problems are completely humanoid specific. One example for this is the bipedal gait. There are not many papers related to ROS on this topic since the researchers are trying to find a solution for a certain problem which doesn't need integration into a full architecture and therefore they don't need to use ROS.

There are alternative middlewares to ROS. The most known are Player/Stage project [Gerkey et al., 2003], YARP [Metta et al., 2006] and OROCOS [Bruyninckx, 2001]. All of them are designed for all types of robots but none offers special advantages for humanoid robots. Since they have no advantage over ROS, which is most widespread, there is no gain in using one of them.

Three general paradigms are currently present for robot control architectures: hierarchical, reactive, and hybrid deliberative/reactive [Murphy, 2000]. These are presented in figure 1.5. They are all based on three commonly accepted primitives of robotics: *Sense*, *Plan*, *Act*. The primitives can be defined based on their input and output, cp. figure 1.4.

The hierarchical paradigm is the oldest one. It strictly follows the three steps of sensing, planning and acting sequentially in a loop. It often collects all information in a global world model which is then used to make decisions in the planning step. The advantage of this paradigm is that it can use elaborated plans, but it can be difficult aggregate a good model, depending on the environment.

A more biologically inspired paradigm is the reactive one. It has no planning step but directly maps sensor information to an action. Multiple of these reactive behaviors are run in parallel to achieve an overall behavior of the robot. It has a fast execution time, but, due to the missing planning phase, the complexity of achievable behaviors are limited.

The third paradigm, hybrid deliberative/reactive, tries to solve this problem by adding another kind of planning to the reactive paradigm. The robot planning consists of a decomposition of the task into subtasks. These subtasks are then executed by reactive behaviors. To be able to do this, the sensed data has to be available to the planner, as well as to the behaviors.

Robot Primitives	Input	Output
Sense	Sensor data	Sensed information
Plan	Information (sensed and/or cognitive)	Directives
Act	Sensed information or directives	Actuator commands

Figure 1.4: The three commonly accepted primitives in robotics with their in- and output [Murphy, 2000]. *Sense* extracts information from the sensor data. *Plan* decides on directives based on the available information. *Act* moves the actuators based on sensed information or directives.

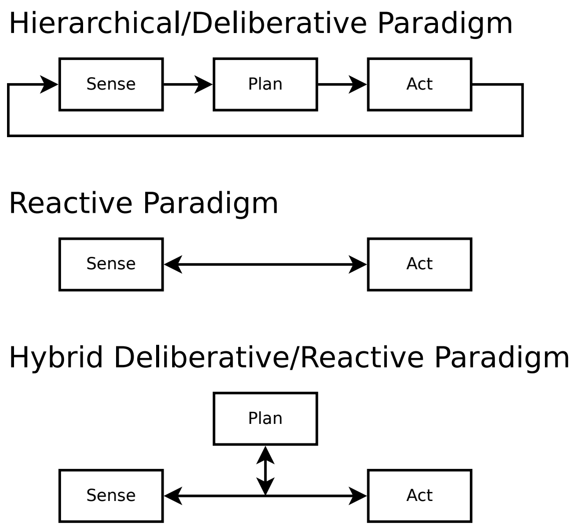

Hierarchical/Deliberative Paradigm
Sense
Plan
Act
Reactive Paradigm
Sense
Act
Hybrid Deliberative/Reactive Paradigm
Plan
Sense
Act

Figure 1.5: The three general paradigms in robotics [Murphy, 2000]. The hierarchical paradigm uses a loop of sense-plan-act which often includes a global model. The reactive paradigm has multiple parallel behaviors without planning phase. In the hybrid paradigm, the planning is done by decomposition of the task into subtasks which are executed by reactive behaviors.

1.2.3 ROS Usage in RoboCup

Every RoboCup team has to write a technical report about the teams' hard- and software in order to participate in the championships. These are called team description paper (TDP). It is thereby possible to get a good estimation of the number of teams which were using ROS in the last year by looking at these papers. It is often difficult to see how far the team is using ROS since the description of the architecture is often rather short. Furthermore, it can happen that a team is using ROS but not mentioning it in the TDP. Thereby the statistic in figure 1.6 is a lower boundary for teams that use ROS at least in some part.

The distribution of ROS in the different leagues is very heterogeneous. For some leagues, it is clear that using ROS would not be possible due to limited hardware, e.g. Standard Platform League. In others, the tasks are not controlling an actual robot, e.g. Rescue Agent Simulation, and therefore it is not useful to use ROS. ROS is used the most outside of RoboCup Soccer. This is due to the fact that these robots have wheels and have to do navigation using SLAM. Therefore the standard ROS navigation stack is often used. In the soccer leagues, the area is clearly defined and simple enough to be abstracted to a 2D plane. Creating maps is not needed and localization can be done using simple transforms to recognized field features.

In the Humanoid League, ROS is more used in the taller sub-leagues. This is probably connected to the fact, that the Kid-Size League was mostly using Darwin-

OPs and similar small robots which had not enough computing power or multiple cores to run ROS effectively. In the last years, a growth in robot size and thereby also a growth in computational performance was observable, removing the limitations of using ROS for many teams.

1.2.4 Summary

The RoboCup Foundation tries to encourage the sharing of code and therefore the use of a suitable software architecture. The most common candidate for a middleware is ROS. Other leagues are already widely using it, leading to an acceleration of progress inside RoboCup but also to an impact in general research. Only the first steps have been made in the humanoid league for now, but the possibilities look promising.

1.3 Thesis Goals

The main goal of this thesis is building a general software framework for different humanoid robots. It should provide an architecture adapted to the RoboCup context, but also usable in general research. While it will be based on the existing code of the Hamburg Bit-Bots, the ROS paradigms and mechanisms should be used as far as possible. The architecture should be modular enough to enable exchange and comparison of nodes with other teams. This includes also using the standard ROS libraries, e.g. “tf”, for expanding compatibility with other ROS nodes. To ease the exchange further, a set of standard messages which apply to the RoboCup Soccer context shall be defined. Also, a set of nodes for the basic abilities, e.g. communicating between team member robots, shall be released as packages and shared between teams.

The parallelism of the code shall be increased from using currently two cores to using at least eight cores, which are currently available on the Minibot. Other teams should be included in this process to ensure, that the resulting architecture actually works for other teams. After finishing, the architecture shall be tested on the Minibot of the Hamburg Bit-Bots in terms of performance and compared to the existing architecture.

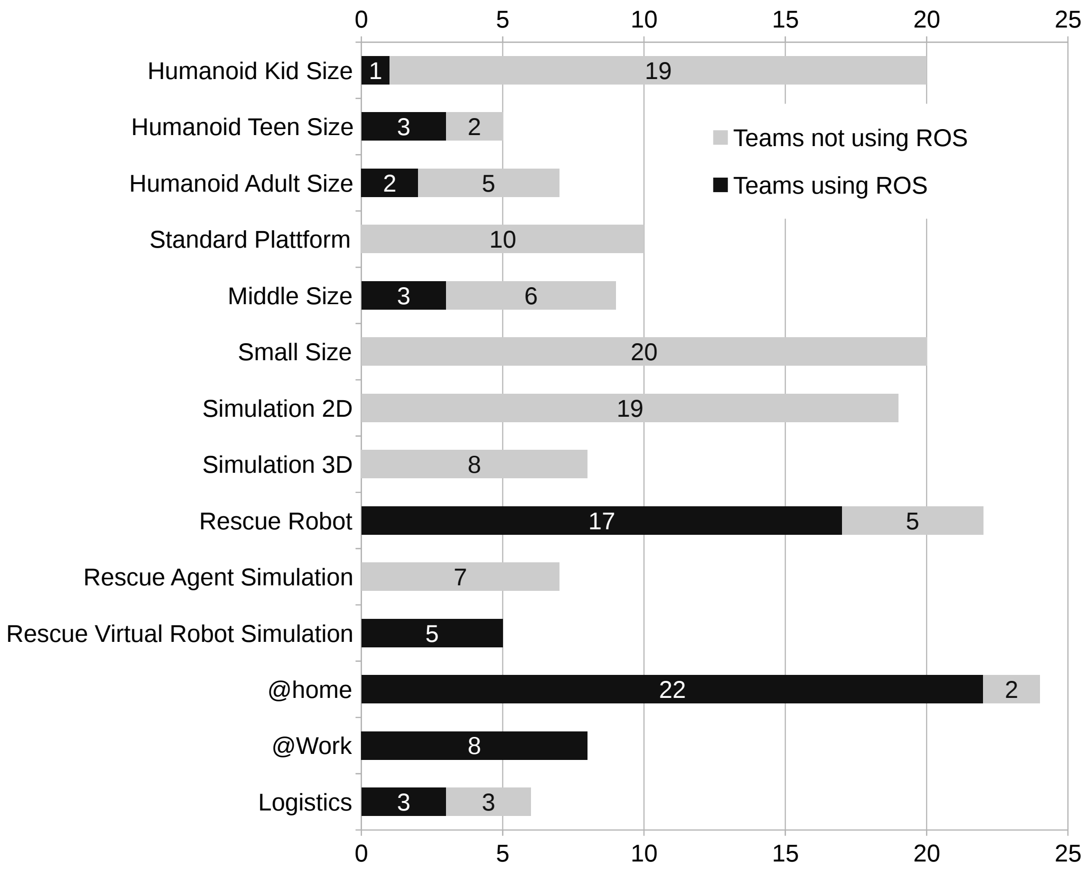


Figure 1.6: Lower boundary of teams using ROS at least in a part of their software in the championship 2016. The data was collected by examining the team description papers [17]. Therefore only teams are counted which wrote that they are using ROS. The chart shows that the overall use of ROS is high in RoboCup, especially if the simulated leagues are not counted. The percentage of usage in the leagues with humanoid robots (first four) is less than in the rest, where wheeled robots are used. It is possible that this is connected to the available computing power, which was significantly lower on humanoid robots in the last years. Since using ROS results in a performance overhead, most teams with humanoid robots wrote their own performance oriented frameworks.

2 Fundamentals

In this chapter, the fundamentals for understanding the thesis' topic will be explained. First, the RoboCup environment will be presented in 2.1. Then humanoid robots will be described in section 2.2. In the end, the Robot Operating System (ROS) will be explained in section 2.3.

2.1 RoboCup

The RoboCup is an initiative to make robotic research more comparable and to increase its visibility to the public. It was founded in 1992 with the official goal to win against the human soccer world champion with a team of robots by the year 2050 [18]. To achieve this, every year a world championship is held, where teams of researchers from all over the world let their robots compete against each other. Over the years also different local events, e.g. German Open and Iran Open, have evolved to give more chances for comparison. The RoboCup consists of different leagues, which do not all play soccer. There are also leagues for industrial operation, home robotics, and rescue robots, see figure 2.1. As this thesis is focused on humanoid robots, both concerning leagues are presented in the following.

RoboCupSoccer	Humanoid SPL Middle Size Small Size Simulation
RoboCupRescue	Robot Simulation
RoboCup@Home	Open Platform Domestic SP Social SP
RoboCupIndustial	@Work Logistics
RoboCupJunior	Soccer OnStage Rescue

Figure 2.1: The RoboCup Leagues and their categories for 2017. SP means standard platform.

2.1.1 Standard Platform League

The Standard Platform League (SPL) was introduced with the goal to let the researchers only focus on the software. Therefore all teams use the same robot, a NAO from the company Aldebaran [Gouaillier et al., 2008]. This does not only make the software more comparable but also more exchangeable due to the same hardware platform. As the developers can focus on the software, the speed of the development progress is high and games in this league are already quite fluent.

2.1.2 Humanoid League

The Humanoid League (HL) gives every team the freedom to buy or build their own humanoid robot, which only has to fit in a set of measurements to ensure that it's humanoid enough. All non human sensors, e.g. depth cameras or laser-range finders, are forbidden. The robots are divided in three size classes: Kid-, Teen- and Adult-Size. Although some de facto standards for the basic layout of the robots have evolved, most teams have very different robots. This makes exchanging code more difficult since it has to be abstracted from the hardware. Therefore, the progress of the HL is not as far as the SPL, where no abstraction from the hardware is needed. Most teams in the HL have still problems with basic control of their robots, e.g. walking.

2.1.3 The Game

Both leagues play with almost the same rules and in almost the same environment. It consists of a 9m x 6m field made of carpet (SPL) or artificial grass (HL), two goals and soccer line markings. Robots can communicate with each other via WiFi and get the current state of the game from an external computer also via WiFi. All robots have to act autonomously and can therefore only use the data of their sensors and the communicated data of their teammates for decision making. The most important information are the position of the ball relative to the robot and the robot's position on the field which is mostly determined by the goals and line markings. Additional data are the positions of team mates and opponents. Based on this the robots try to kick the ball in direction of the opponent goal to eventually score a goal. Passes and more complex team behavior are rarely used for the moment, especially in the HL.

The referee team consists of two normal referees and one referee sitting at a computer next to the field. This computer runs a software called *game controller* which is transmitting the current status of the game, e.g. goals and remaining time, to the robots. It also tells robots when they get a penalty. These robots have to listen to this command and are not allowed to move while having a penalty.

2.2 Humanoid Robots

"A humanoid robot is a robot that has a human-like shape." [Kajita et al., 2014]
Humanoid robots should be able to interact with the world like humans. This is necessary to prevent efforts for making the world robot friendly. Robots which strongly resemble humans in appearance and behavior are called androids.

The development of humanoid robots follows two main goals. First, for achieving an artificial intelligence similar to the human one, a learning process is needed which is also similar to the human one. Therefore the robot has to have comparable capabilities in sensing and actuation. The second goal is related to the integration

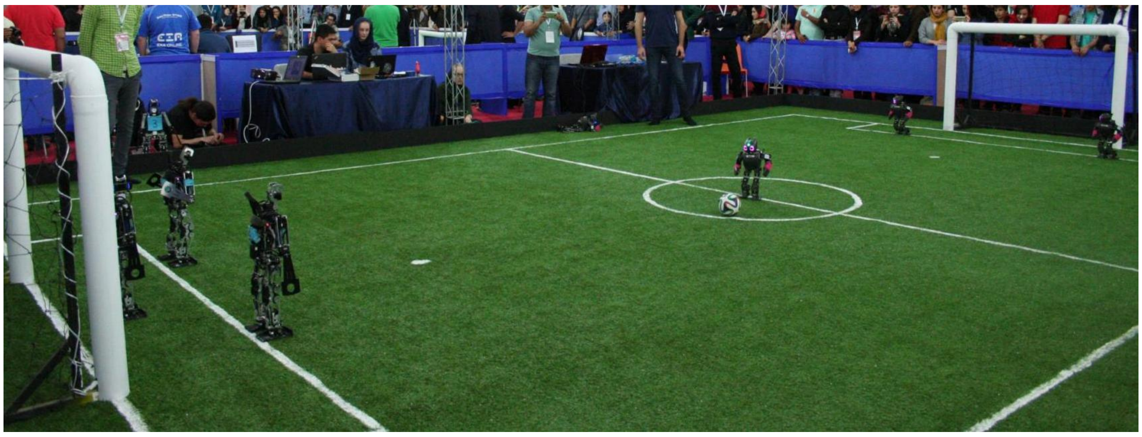


Figure 2.2: A Humanoid Soccer League field which is 9m x 6m in size and consists of an artificial turf. While the field is clearly specified, the outsides are not, which can be challenging for vision algorithms. The picture was taken during the Iran Open 2016.

of robots in our environment. Since everything humans use is adapted to them, a robot which should work in this environment, e.g. helping old people at home, must be able to interact with this environment. Otherwise, the environment would have to be adapted to the robot, for example by replacing stairs with ramps for wheeled robots, which would be more cost intensive.

In the following, humanoid robots are further described in the three typical sections sensing, computing and actuating.

2.2.1 Sensing

Sensing on humanoid robots is more difficult than on other types of robots since the movement of the robot leads to noise in the sensors. For example, camera images get blurry if the shutter speed is too slow or if it is a rolling shutter sensor. Also, odometry data is less certain than on wheeled robots. Furthermore, the position of sensors in relation to the world is not stable. On a wheeled robot, the camera is usually a fixed hight above the ground. A humanoid robot has to do forward kinematics from its support feet to the camera to get the position and orientation. This also requires knowing which leg is the supporting leg and if the robot is standing at all. For this an inertia measurement unit (IMU) and position encoders in the joints are crucial. These are described in the following.

The IMU is a combination of a gyroscope and an accelerometer. It measures angular velocities and linear accelerations. It is normally installed inside the torso near the center of mass. The sensor values can be used to detect falls and to identify the side on which the robot landed after a fall.

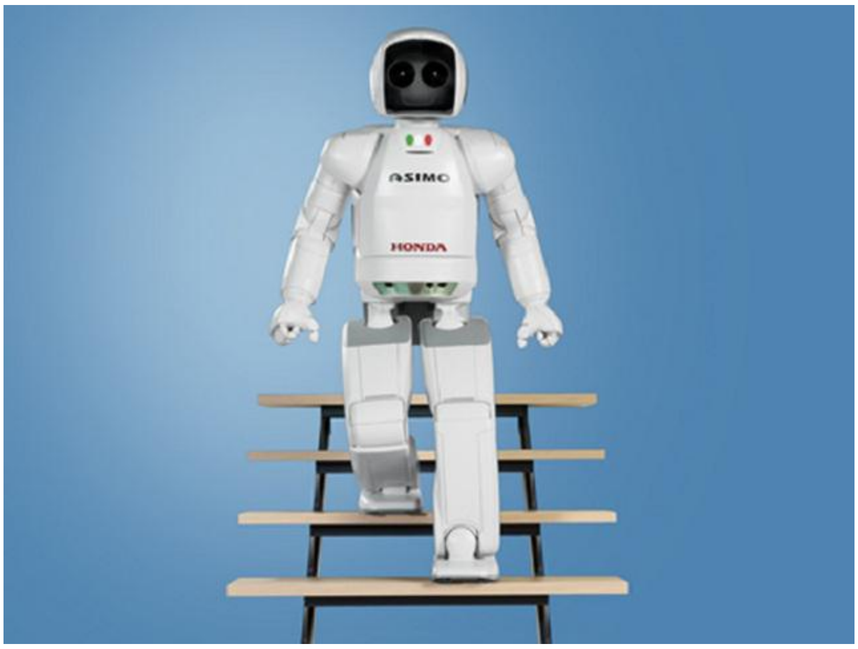


Figure 2.3: The Asimo robot walking on a flight of stairs [2]. The algorithm was adapted to this specific stair size, therefore it is not applicable in the real world. This transfer from the lab environment to the real world is tried to be achieved in RoboCup by enforcing realistic testing conditions in which teams have to prove their performance.

Furthermore, the positions of the robot's joints have to be measured. This is usually realized by a combination of a magnet on the end of the outermost gear and a hall sensor [Ramsden, 2011]. When the gear rotates, the magnet field rotates accordingly. The hall sensor measures this and a microcontroller can compute the position of the servo based on this data.

2.2.2 Computing

In the early days of humanoid robots, having enough computational power was difficult because the space and mass for the computational unit are very limited. Today, single-board computers, e.g. Raspberry Pi [Upton and Halfacree, 2014], and Intel NUCs [Perico et al., 2014] are mostly used, depending on the size of the robot. All these boards provide GPUs and multiple CPU cores. Therefore the interest in parallelizing software and outsourcing computation to the GPU, especially for neural networks and computer vision, rose in the last years. Sometimes multiple cheap single-board computers are installed and connected using an Ethernet network, to get a high performance for low cost. This comes with the disadvantage that the software has to be able to run in parallel on a distributed system.

2.2.3 Actuating

A humanoid robot is typically composed of multiple joints. Each joint has to be actuated which can be done in two different ways. Either the joint can be moved by tendons, similar to the human movement, or a motor can be placed directly into the joint axis. The later approach is used more often in smaller robots, since it needs less space. Using tendons makes exact positioning of the joint more difficult, but also enables to use bigger and linear motors, since they don't have to be placed inside the joint.

Actuators need a high strength in relation to their weight, since they have to carry themselves. Multiple actuators are typically needed for one limb, resulting in a high number of cables, which have to be laid over multiple joints. Therefore a bus system is often used to limit the number of cables. The bus requires the motors to have a micro controller, to receive goal positions and to send the current position. Micro controller, motor and a gearbox are usually grouped together in a case and called *servo*. For reaching the goal position and for holding it, a controller is required in the firmware of the servo. This is usually a PID controller [Wescott, 2000].

2.3 Robot Operating System

The Robot Operating System (ROS) was developed by Willow Garage, originally for the PR2 robot in 2007 [Quigley et al., 2009]. It is an open source framework for developing software in robotics with a focus on the ability to run parallel on distributed computer systems. It can be run on different operating systems, but only Ubuntu and Debian are officially supported. Its main advantage is a big library of available software modules for common robotics tasks. These are developed, maintained and documented by the ROS community and adding further modules is easy. Using ROS decreases the time for developing software as most of the parts can be taken from the library. In the following section, a short overview of the concepts of ROS is provided. A deeper insight can be found in the online documentation [3].

2.3.1 Nodes

A *node* is a process in the ROS system. It can communicate with other ROS nodes via topics or services. In doing so, all nodes form a computational graph. Each node has typically a clearly defined subtask. For example one node gets the camera image, a second node detects balls on this image, and a third node computes the ball positions. Nodes can be easily reused in different tasks, for example the node which gets the camera image can be reused in a different task that detects goals.

Open source implementations of nodes for most standard subtasks, especially hardware controlling, already exist. Thereby the effort in implementing a new task is drastically decreased. The most important packages used for this thesis are mentioned in section 2.3.7.

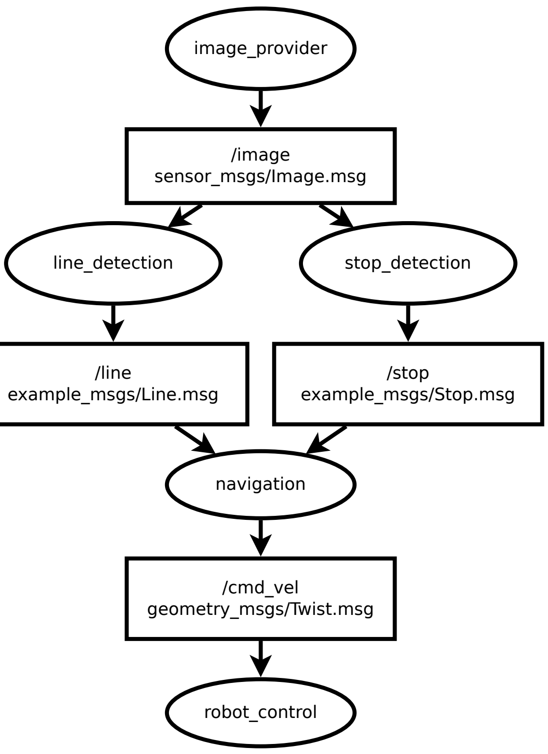


Figure 2.4: Example ROS architecture for a simple wheeled robot with the task to follow a line until it finds a stop marker. The nodes are displayed as ellipses with a name and the topics are shown as rectangles with name and message type. First, an image is provided from the camera. Lines and stop markers are detected on this image. Based on this information, a navigation node computes the necessary movement of the robot and publishes it. The robot_control node is then controlling the motors of the robot accordingly.

Twist.msg		Stop.msg	
geometry_msgs/Vector3	linear	uint8	RED = 0
geometry_msgs/Vector3	angular	uint8	GREEN = 1
		uint8	color
		float32	distance

Figure 2.5: Two example messages. The `Twist` message (left) is already defined in ROS. The `Stop` message (right) is newly defined for the special application case of figure 2.4.

2.3.2 Topics and Messages

Messages are the main communication method between ROS nodes. Messages are always published on a topic, which is identified by a name and can only transmit one type of message. A node can subscribe to a topic and will get the messages other nodes publish to it. Each node can subscribe to and publish on any number of topics and will not know with whom it communicates.

Communication can happen between nodes running on the same computer as well as nodes distributed over different computers, as long as they are connected with an TCP/IP network. This is not only useful for parallelization but also eases the visualization of the robots state on a separate desktop computer. Each message has a type which is either predefined in the ROS system or defined by a user package. The interface of a node is mainly defined by the types of the messages it subscribes and publishes. These types can be either predefined in ROS or be newly created by a developer, cp. figure 2.5.

2.3.3 roscore

The `roscore` is the most central part of a running ROS system. It consists of the `rosmaster`, the ROS parameter server and the `rosout` logging node. The ROS master is registering which topics are published by the nodes and on which topics nodes want to subscribe. If one node wants to subscribe to a topic which another node is publishing, a peer-to-peer connection is established from one node to the other. Thereby the master does not become a bottleneck since it only connects the nodes but does not need to handle the messages. To do this, a subscribing node asks the master for a list of nodes which publish on this named topic. The master holds a table of all publishers and sends their names to the subscribing nodes. It also remembers which nodes are subscribing to this topic, if a new publisher is started later. This process is shown in figure 2.6.

The ROS parameter server is a global key-value store. It is mainly designed to be used for static configuration parameters. Transmission of data between nodes should be done via topics to prevent making the parameter server a bottleneck. All parameters are globally visible. If a parameter should be able to be changed during runtime, dynamic reconfigure can be used. It provides the possibility to state on compile time which parameter values should be changeable and provides an `rqt` plug-in for it, see section 2.3.9. The `rosout` logging node subscribes to the */rosout* topic which is the standard topic for logging. ROS has built in methods to send data to this topic which are displayed on runtime and written in a log file.

2.3.4 Services

Services can be seen as remote procedure calls (RPC). In contrast to messages which are an unidirectional stream of data, service calls are blocking and waiting for a response. They have defined types which consist of a request and a response

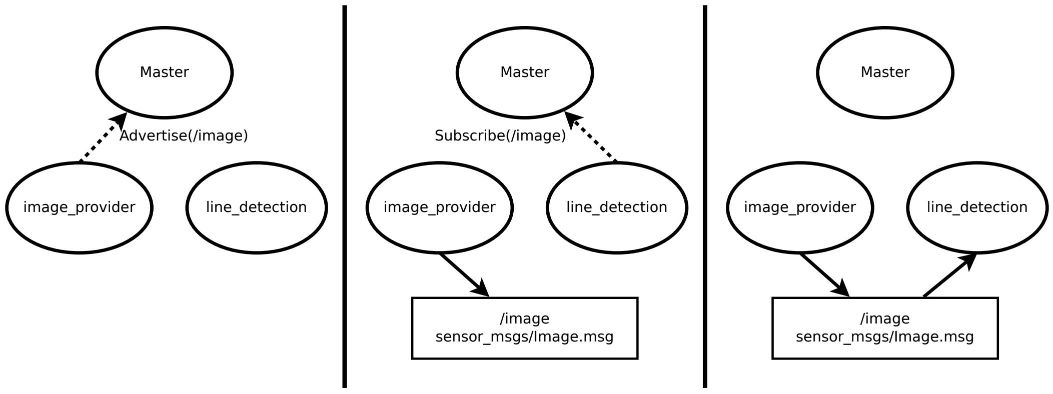


Figure 2.6: Procedure for establishing a peer-to-peer connection via the ROS master. First, one node advertises the topic it wants to publish. The master remembers by whom this topic is published. When another node tries to subscribe to this topic, the master informs both nodes about each other, so that they can start a communication.

message. The node providing the service is called server and the node calling the service is called client. Services are useful for fast tasks, but should not be used when getting the result can take a long time, because the calling node is blocked until completion. For longer tasks, actions should be used (section 2.3.5). A possible service for the example described in figure 2.4 would be a manual stop service. It would be advertised by the navigation node and would stop the robot even without markers. As this would take not a lot of time, basically just alternating the state of the navigation node, this would be possible to do with a blocking service call.

2.3.5 Actions

Actions are used when a task which takes a long time should be called asynchronously, in contrast to the synchronous service calls. Each action has a specified type which consists of three messages: goal, feedback, and result. A node providing the action, the action server, is called by another node, the action client, by sending a goal. The action server will now try to achieve this goal, while the action client is not blocked and can perform other tasks. The server will constantly send feedback messages to the client to inform it about the status of the process towards the goal. The server will send a result message when the goal was reached or if the action was interrupted. Actions can be interrupted by sending new goals which the server considers more important or by request of the action client, for example, if the sent goal is not useful anymore. A possible action for the example described in 2.4 would be driving a certain distance on the line. This needs some time because the robot has to move. Therefore a blocking call is not feasible. The action runs asynchronously and can also be interrupted, for example, if the line ends before reaching the desired distance.

```xml
<package>
  <name>example_package</name>
  <version>1.0.0</version>
  <description>Short example for a package.xml.</description>

  <maintainer email="ex@ample.org">Jane Doe</maintainer>

  <license>BSD</license>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>example_2</build_depend>
  <run_depend>std_msgs</run_depend>
</package>
```

Figure 2.7: Example of a package.xml. Different tags can be used to specify metadata of the package. These are necessary to state the owner of this package and its license. The listed dependencies are used by catkin for compiling.

2.3.6 Code Organization

The smallest and main unit for organizing software for ROS is a package. A package has a *package.xml* (cp. figure 2.7) which describes different metadata of this package, e.g. the package name, the author, the license and dependencies on other packages. A package can build on its own if all dependencies are met. The content can, among other things, be ROS nodes, visualization plug ins, libraries or datasets. It can be distinguished between dependencies on build and runtime. Different packages can be grouped in meta-packages, which hold no content of their own but only collect packages which belong together.

2.3.7 Code Distribution

The idea of reusing nodes can only be utilized if the nodes are shared. Therefore a wiki is provided with documentation to the packages. Usually the packages are hosted in GitHub repositories which are linked in the wiki. A description of the package and its documentation is provided in the wiki. Since the package dependencies are clearly specified in the `package.xml`, it is easy to install them. Further development, e.g. bug fixes, of packages is usually done in the GitHub repository by creating pull requests or issues.

2.3.8 ROS Packages Used in This Thesis

In the following a short description of the ROS standard packages which were used in this thesis is provided.

rosbag

`rosbag` is a package which can record the messages of a ROS system during runtime and save them into a file. This file, also called `rosbag`, can later be further investigated or replayed. This allows easier debugging and testing of nodes without the need to bring up the whole ROS graph. It is also particularly helpful to compare performance of different algorithms on the same dataset.

tf2

`tf2` keeps track of the different coordinate transforms and provides information of relations between them. One example would be the position of the foot tip in relation to the camera. The `robot_state_publisher` package subscribes to `JointState` messages to get the current joint positions and provides these information to `tf2`. `tf2` can be called by any node to get coordinate transforms, not only at the current time but also for past positions.

URDF

The Unified Robot Description Format (URDF) is an XML format for describing robots. The `urdf` ROS package provides a parser which can read these files, enabling other packages, e.g. `tf2`, to use this to abstract from the robots structure. When creating a new robot, only creating a corresponding URDF is needed to make ROS software run on it. The URDF is therefore crucial for hardware abstraction.

xacro

`xacro` provides macros to `xml` documents. This is especially useful in the ROS context to make URDF files, since they are often symmetric. By using `xacro`, it is possible to define, for example, an arm just once and then use a macro to generate a left and a right arm out of it. The `xacro` files are expanded to `xml` files by running the `xacro` ROS package.

2.3.9 Visualization

One of ROS' core strengths is providing different tools for visualization. Due to its publisher-subscriber architecture, it is very easy to establish a data stream from the program to the visualization. Either the visualization can subscribe on topics that are already being in use or additional topics can be provided from the software to deliver more information to the visualization. Messages in ROS are only published if there is a subscriber on the topic, therefore publishing additional topics for visualization reasons does not cost performance when the visualization is not running. ROS provides two main tools for visualization which can be extended by plug-ins. It is also possible to implement own tools which are independent from these.

rqt

rqt is a QT based interface with ROS connection (figure 2.8). Plug-ins can be launched and provide a `QtWidget`. Multiple widgets can be displayed at the same time. They can be resized and positioned by drag and drop. ROS provides a set of plug-ins but writing an own plug-in is possible too. These plug-ins can be used to visualize data in 2D but also to provide a controlling interface. In the following, some examples of important plug-ins are shown.

The `node_graph` plug-in shows all currently running nodes and topics. Published and subscribed topics are connected to their nodes and thereby it is easy to see the flow of data. This plug-in is especially helpful to get an overview of the running software and to find misconnected nodes.

The `topic_monitor` lists all current topics. It is possible to subscribe to them and to get the current transmitted values. Further on, statistics about the publishing rate and the used bandwidth can be shown for each topic.

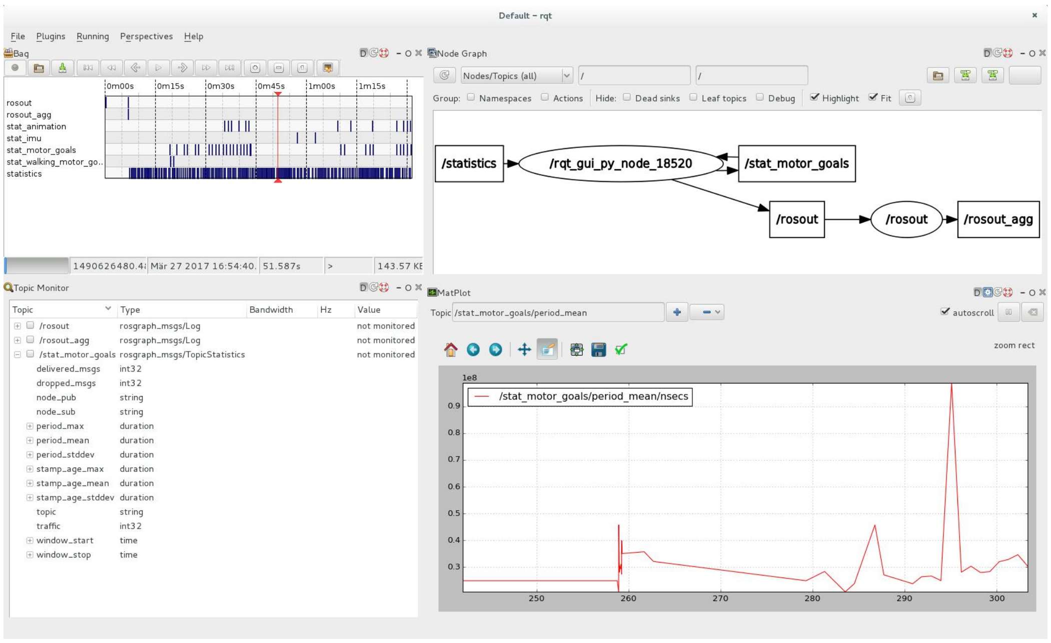
The `rqt_plot` plug-in enables live plotting of data, using `matplotlib`. The `dynamic_reconfigure` plug-in provides an interface to change parameter values previously defined to be reconfigurable. This is useful when tuning parameter values, because changing it is possible on the fly and does not require a restart of the software.

RViz

RViz provides a 3D visualization of the robots state and its environment. The standardized URDF format is used to get a visual robot model, which is then used to show the current positions of the robots joints. Furthermore, sensor data can be displayed using marker messages. These messages can be published by any node and define three dimensional states which are displayed in RViz. This is for example helpful to get a visualization of recognized objects. Furthermore, a lot of different standard ROS messages can directly be visualized in RViz, e.g. camera images, depth clouds, laser scans and point clouds. Thereby RViz provides visualization without additional effort, if the standard messages are used. It is especially used for localization and mapping, because it is possible to see the current sensor inputs of the robot as well as its map in the same window (c.p. figure 2.9).

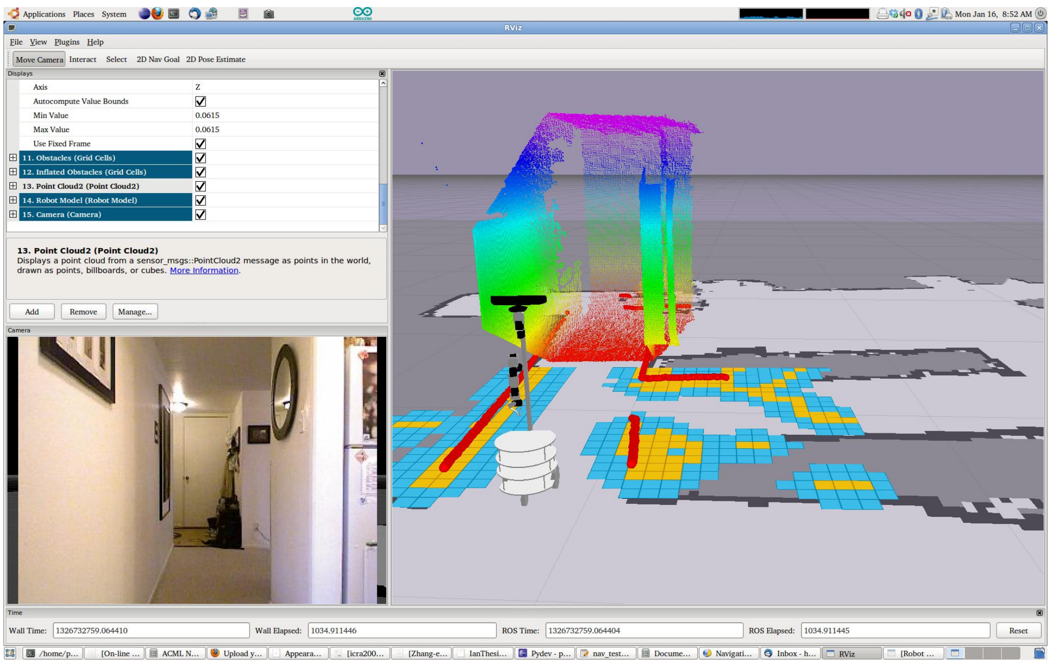
2.3.10 Simulation

Simulation is a crucial part when developing robot software, since it gives the developer the possibility to run his software without using hardware. This can prevent hardware damages because bugs can be found before running it on the robot and it can be used to accelerate development, e.g. by testing in faster than real time. While ROS can generally use any simulator, Gazebo is normally preferred, since it has a good ROS integration and was also originally developed for ROS. In order to use Gazebo, an URDF of the robot is required.



Topic	Type	Bandwidth	Hz	Value
/rosout	rosgraph_msgs/Log			not monitored
/rosout_agg	rosgraph_msgs/Log			not monitored
/stat_motor_goals	rosgraph_msgs/TopicStatistics			not monitored
delivered_msgs	int32			
dropped_msgs	int32			
node_pub	string			
node_sub	string			
period_max	duration			
period_mean	duration			
period_stddev	duration			
stamp_age_max	duration			
stamp_age_mean	duration			
stamp_age_stddev	duration			
topic	string			
traffic	int32			
window_start	time			
window_stop	time			

Figure 2.8: The rqt interface with different plug-ins started. `rosbag` is opened (top left), it publishes values, which is visible by the edges in the `node_graph` (top right). The data is plotted using `rqt_plot` (bottom right). The `topic_monitor` lists the current topics (bottom left).



Axis	Z
Autocompute Value Bounds	✓
Min Value	0.0615
Max Value	0.0615
Use Fixed Frame	✓
11. Obstacles (Grid Cells)	✓
12. Inflated Obstacles (Grid Cells)	✓
13. Point Cloud2 (Point Cloud2)	✓
14. Robot Model (Robot Model)	✓
15. Camera (Camera)	✓

13. Point Cloud2 (Point Cloud2)
Displays a point cloud from a sensor_msgs::PointCloud2 message as points in the world, drawn as points, billboards, or cubes. More Information.

Wall Time: 1326732759.064410 Wall Elapsed: 1034.911446 ROS Time: 1326732759.064404 ROS Elapsed: 1034.911445

Figure 2.9: Example visualization with RViz. In this case a wheeled robot is mapping its surroundings using depth data from a Kinect [4].

This URDF is used to display the robot in the simulator and to compute its collisions with itself and the environment. The simulator can provide sensor data in the corresponding standard messages. To actuate the servos of the robot, different controllers are available which work with the corresponding messages, e.g. `JointTrajectory`.

3 Hardware and Software

In this chapter, the used hard- and software are explained in detail. First the used Minibot robot platform is described in section 3.1. Afterward, the previous software architecture is investigated in section 3.2.

3.1 Minibot

The Minibot was designed by the Hamburg Bit-Bots because their previous robot platform, the Darwin-OP, became too small to effectively participate against the other robots in the league. Furthermore, the computational power of the Darwin-OP was low and newer boards could not be installed easily due to the restricted space. Therefore an upscaled platform was constructed with the same composition of joints. The computational unit was replaced by an Odroid XU3 with eight cores [10].

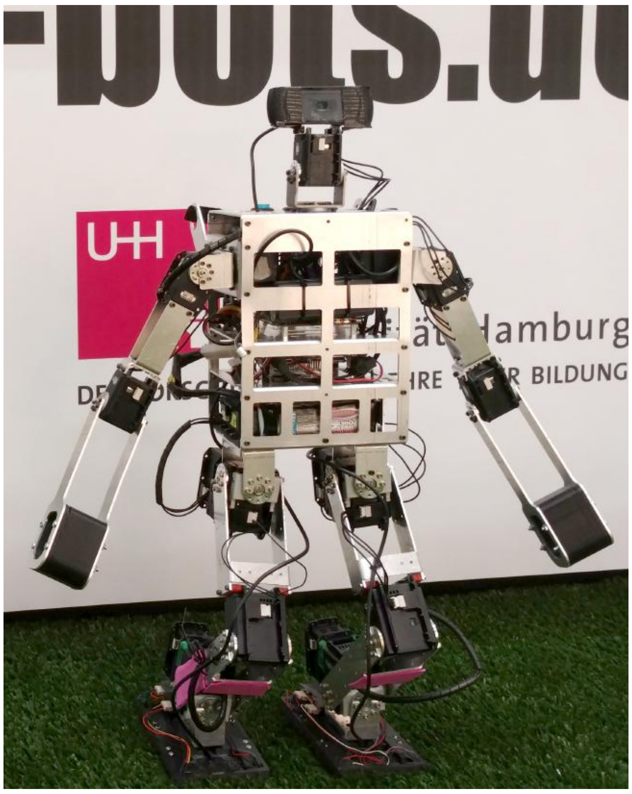


Figure 3.1: The Minibot in its walking position.

The robot has two degrees of freedom (DOF) in the head, two in each shoulder, one in each elbow, three in each hip, one in each knee and two in each ankle. This leads to overall 20 joints. These joints are connected with bended sheet aluminum parts. The size of the robot is 0,7 m and the weight 5 kg. Since most sensor types are forbidden in the HL, it only has a camera, an IMU and the position encoders of the servos. Lately, an experimental set of pressure sensors was installed under the feet, to stabilize the robots walking.

In the following, the used servo motors and the motor controller board are further investigated.

3.1.1 Dynamixel Servos

Dynamixel is a series of servo motors by the South Korean company Robotis [5] which are designed for the use in robotics. They are widespread in humanoid robotics and especially in the RoboCup Humanoid League. Different models are available but all share the property that they are daisy-chainable and communicate using a bus protocol. In this thesis, three different models from the M series were used, the MX-28, MX-64 and MX-106, see figure 3.2. They all use the TTL protocol and differ only in size and torque. The TTL bus is used with 1 *megabaud*.

The M series can communicate using single read and write packages as well as bulk read and sync write, which are used to access all motors at the same time. Since accessing all motors at once requires less packages on the TTL bus and therefore the update rate can be increased, it is commonly used in RoboCup. Regardless which of these methods is used, only a single goal position can be transfered. Since it would often be more interesting to send joint trajectories an effort was made to implement an alternative framework for these servos [Fabre et al., 2016]. For the moment, almost all teams are still using the standard firmware, therefore being able to accept only one goal position.

Figure 3.2: The MX-28T, MX-64T and MX-106T servos [6]. All three are controlled by the same TTL bus protocol. The only difference is the maximal torque.

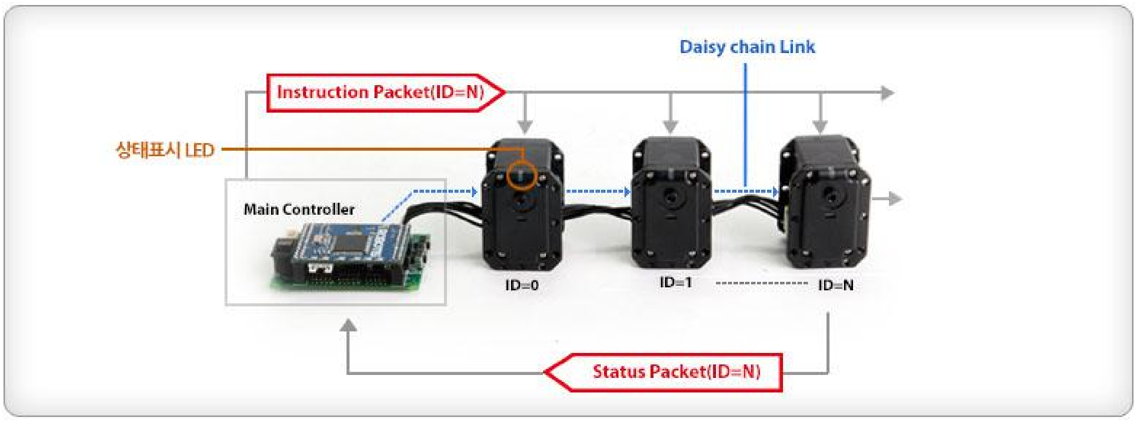


Figure 3.3: Communication between controller board and motors via a daisy chained TTL bus system [7].

3.1.2 CM730

The CM730 is a motor controller board also produced by Robotis. It was originally used in the Darwin-OP [Ha et al., 2011] but is now widely used in the Humanoid League. It is connected to a computer via USB2.0. It provides five connected TTL sockets for five chains of Dynamixel motors. Therefore it is easy to make a chain for both arms, both legs, and the head. Unfortunately, all chains are on the same TTL bus. Therefore all communication with the motors has to be done via one bus. This limits the communication rate to each motor by the count of the connected motors. Furthermore, the CM730 provides an on board accelerometer and gyroscope, as well as two general purpose buttons and a reset button.

3.2 Bit-Bots Framework

The old architecture of the Hamburg Bit-Bots was designed to run in two processes because the originally used Darwin-OP robot had a single core processor with two hyper-threads. Therefore the software was split into two processes. In the first one, all software closely related to the hardware (motion, walking, animation) was integrated. The vision processing and the behavior were included in the second one. Those two processes use a shared memory IPC to communicate with each other.

3.2.1 Module Architecture

The main part of the software is divided in modules. Each of them is providing an update method. All modules are executed together in a loop by calling their update methods. The main work of a module should be done in this method, but a module can also define a pre- or post-method, which is called before or after the loop.

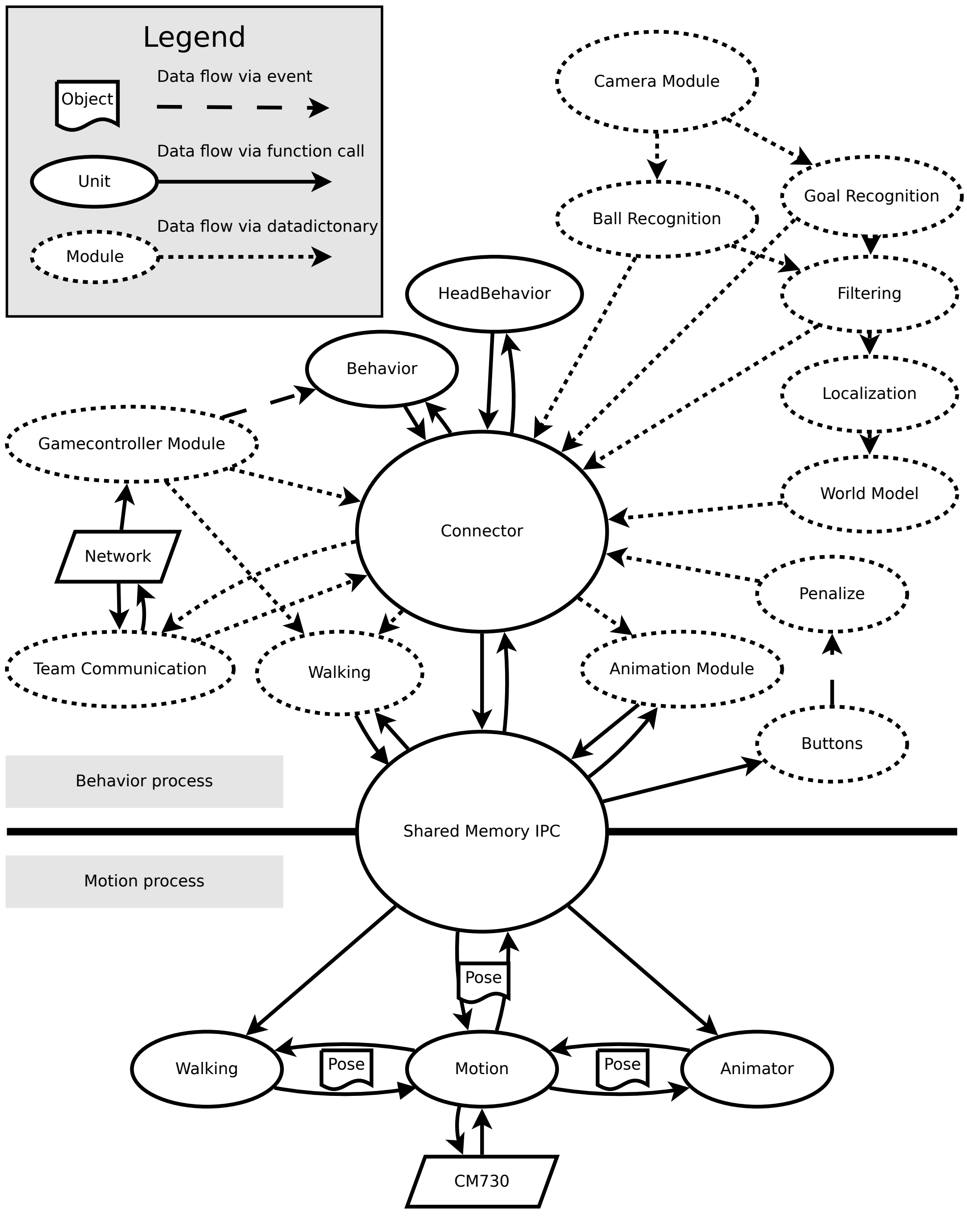


Figure 3.4: Overview of the original Hamburg Bit-Bots architecture. The top half shows the behavior process, which is connected via the shared memory IPC to the motion process. The modules exchange information via the data dictionary and the connector collects all of them to provide it to the behavior. Additionally, events are used to inform software parts about pressed buttons and changes of the game controller. In the motion process, a `pose` object is exchanged via function calls. It includes current and goal positions for all motors.

Module	Requires	Provides
Camera		RAW_IMAGE
Ball Recognition	RAW_IMAGE	BALL_INFO
Goal Recognition	RAW_IMAGE	GOAL_INFO
Vision Visualization	BALL_INFO, GOAL_INFO	

Figure 3.5: Example of the module architecture with two possible orders of modules. The modules would be run either in the displayed order or with ball and goal recognition switched.

The order of the method calls is defined by the input and output data of a module. Each module defines provided and required data by named strings. The architecture makes sure that a module is only executed after the required data was provided in this cycle. If a module is started which requires data that no other started module is providing, an error is given directly at the start.

The data is shared between the modules by a shared dictionary which is passed as an argument to each update method. This architecture can be seen as a blackboard architecture, in which the data dictionary is the blackboard, the modules are the knowledge sources and the control is done by providing a fixed running order due to the required and provided data [Nii, 1986]. Threaded modules are possible, but they are only useful for I/O-operations because Python can not actually run them in parallel. An advantage of this architecture is, that it makes sure that all modules are called sequentially, which simplifies some algorithm. For example, the behavior module gets all aggregated recognition information from the vision for each image. Disadvantages are that it is not generally parallel, very adapted to a specific system and the framework is not used outside of the team.

3.2.2 Motion

The motion is responsible for all tasks regarding the control of the robot. It is handling the communication to the motors and the IMU, provides the current pose to the IPC and gets motor goals via the IPC, for example from the animation module. The motion also takes care of falling and getting up. Therefore it plays also animation by itself by using the animator class. Furthermore, the motion holds a walking object for computing the next walking pose, if necessary. There are different parts of the software that want to set the motor positions of the robot: animations from the behavior, head positions from the head behavior, walking, falling and standing up. Therefore the motion has to decide which part of the software gets control over the motors. In summary, the motion acts like a state machine, without having its structure.

3.2.3 Visualization

To show the status of all robots during a game, a graphical user interface was implemented for GTK+ called `debug_ui`. It receives data from the robots via UDP, since the robots are not allowed to use any TCP during the game. In the source code, first a scope was defined, e.g. ball detection. Then data could be transmitted for this scope, e.g. the balls position. This gave a structure to the data which was used by the `debug_ui` to order it. Different widgets could be started to display data. There were implementations for a tree-like view of the raw data, an image viewer with markings for detected objects, an overview of the motors temperatures, a visualization of relative object positions to the robot and an overview of the field. The program was not used frequently because it had to be compiled additionally and it was not stable.

4 Architecture

Different objectives for the software architecture were listed in section 1.3. First and most of all, it should be usable on different robots, as well as by different teams. Therefore an architecture is presented which tries to ensure a maximum of flexibility.

The structure of this chapter is as following: First, the basic approach is presented in section 4.1. Afterward, an overview of the architecture and the used names is provided (section 4.2). Finally, the chapter is concluded with the definition of the ROS messages in section 4.3.

4.1 Basic Concept

The goal of the architecture is to be flexible, so that teams can adapt it to their implementation. Still, it has to allow an easy exchange of modules. To achieve this, only ROS messages and their topic names were defined. It is also not necessary to publish all defined topics. This leads to the possibility of different implementations of nodes that are still exchangeable and comparable, cp. figure 4.1. Thereby, semantic parts, but not necessarily single nodes, are exchangeable.

It's crucial to give the teams freedom in design, since not all their architectures can be exactly the same. As RoboCup is a research competition, methods and algorithms change often. New concepts have to be compared to the old ones to see if there is an improvement. Comparing is also simpler using this common architecture since input and output of different implementations stay the same.

4.2 Overview

There are different tasks that a robot architecture has to perform. The most basic is to communicate with the robots hardware, e.g. via the TTL bus to the servos. In the proposed architecture this is called *hardware communication.* It also abstracts from the concrete hardware by providing a ROS interface to get its sensor data and to control it.

Above that, there is a part of software that controls the robot's hardware by doing meaningful actions. In the case of a humanoid robot this is *walking* for generating motor positions in order to achieve a gait and *animation* to play predefined sequences of positions. Since there are potentially multiple users of the robot's joints, there has to be a managing unit which controls the access, the *hardware control manager (HCM).* It acts similar to the biological cerebellum which handles motor functions in the human brain.

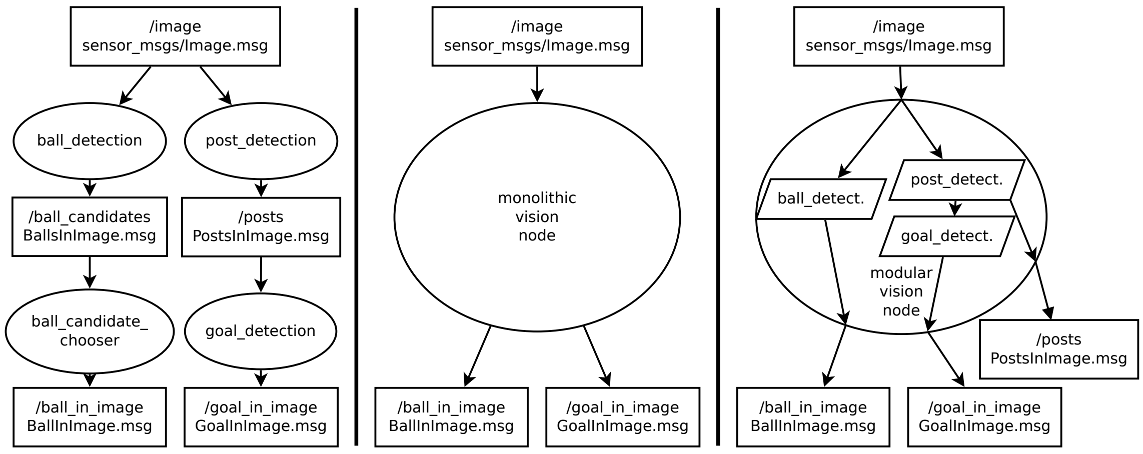


Figure 4.1: This figure presents the basic concept of the architecture. Three different possible implementations for the vision are shown. On the left, the ball and goal detection are implemented in different steps. First, the candidates for balls and goal posts are chosen and then the actual result is computed in a second step. In the center, everything is done in one node, e.g. because the code was transferred from an old framework and was not modular. On the right, one node is used again, but it has an internal modular structure, thus allowing to publish additional data for visualization. All three have implementations have different nodes but can be exchanged easily and their results can be compared.

The HCM detects falling, interrupts animations or walking if needed and plays standing up animations when the robot fell down. In the special case of RoboCup Soccer, it makes sure that the robot completely stops to move if it is penalized, by using the information provided trough the *gamecontroller client*.

Above this HCM, there is a higher decision taking part, the *behavior*. It controls the robot on a more abstract level, deciding for example to kick and where to go. This can be split into a body and a head behavior, where the head is only controlled by sending goals to look for, e.g. tracking the ball. Finding a path to the navigation goal is done by the *path planning*. It decides on a path and generates the necessary walking commands.

In order to know where to go, the robot needs information about its surroundings. Since only humanoid sensors are allowed in the HL, the robot has to rely on its camera. The *image acquisition* gets the image from the camera and removes distortion from it. The *vision* uses this image to detect features of the field and to get visual odometry. The positions of the detected objects in relation to the robot have to be computed from the position on the image, also known as transferring from image space to *relative* space, which is done in the *pixel to position transformation (PPT)*. The *absolute* position of the robot on the field is computed by the *localization* by using odometry and field features.

All vision information are aggregated to a *personal model (PM)* which can filter them over time. The robot can exchange information with other team players, using the *team communication*. The personal model can be updated to a *team updated model (TUM)* by using these information.

Overall, this architecture is mostly similar to the hierarchical paradigm, cp. section 1.2.2. Mainly, the camera is used for sensing. The resulting information are aggregated into a global model which is used by the behavior to plan the actions, e.g. walking. The main difference to the classical hierarchical paradigm are the HCM and the head behavior. The HCM acts more like a reactive behavior, since it transfers directly from sensing, i.e. the IMU, to actions, i.e. animations. It takes no directives from the behavior to do so, which would be the case in a hybrid deliberative/reactive paradigm. But it would be possible to add a directive to tell the HCM if it should stand up or not. This was not done, since it is obligatory to directly stand up in RoboCup Soccer. The head behavior is a more clear definition of a reactive behavior. It takes a directive from the body behavior, specifying the searched object, and controls the motors afterwards by itself.

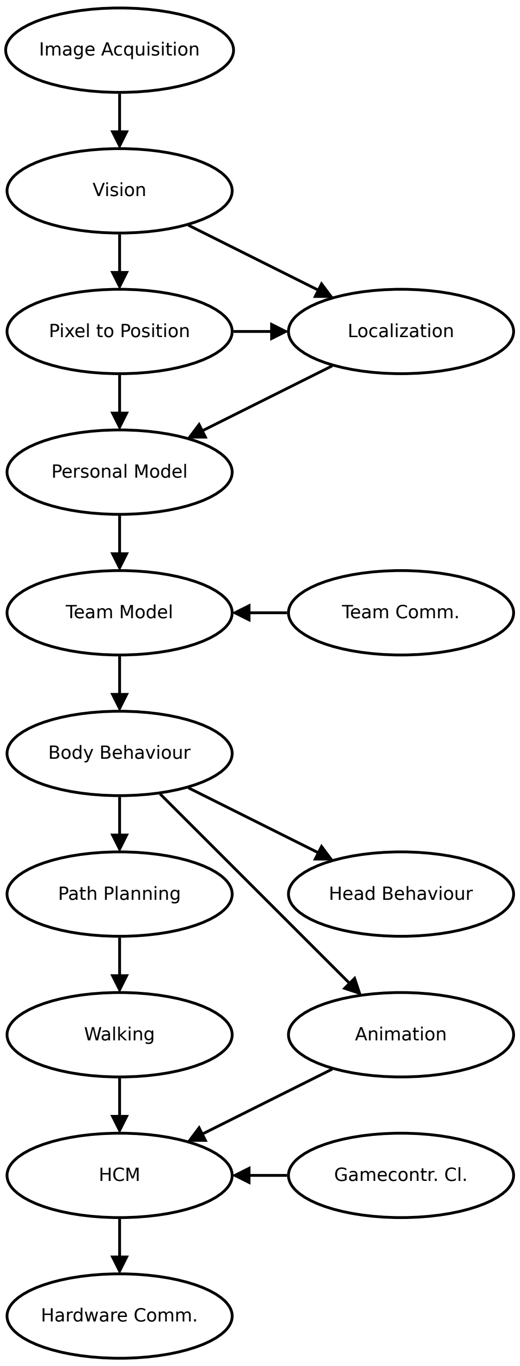


Figure 4.2: Very simplified representation of the different task and the information flow in the proposed architecture.

4.3 Message Definitions

Following the basic concept, described in section 4.1, a set of messages were defined for all tasks, described in section 4.2. These messages are shown in their data flow context in figure 4.3. Since the nodes are not defined, there are multiple ways to implement an actual architecture based on these messages. Two examples are shown in figure 4.4 and 4.5, with different degrees of granulation. The exact definition of all messages is included in the appendix (section 8).

The definition for the necessary topics and messages should not be done by only one person or one team, as it would be too specific and "over fits" on their current architecture. At the same time, developing the architecture with all teams in the Humanoid League together is also not practical. Therefore the development was done in three steps. First developing a prototype only inside the Hamburg Bit-Bots team. This resulted in the basic principle of defining only messages and topics. In a second step, this prototype was introduced to two other teams (WF Wolves and Rhoban FC) of which one was using ROS already and one wasn't. The prototype was changed based on their feedback and was already very near to the end result. During the actual implementation of nodes, new arising changes were discussed between the teams. In a final step, the architecture will be presented at the RoboCup Symposium 2017 to the other teams. In the next subsections, the different parts of the architecture are further investigated.

4.3.1 Image Acquisition

The image acquisition uses only standard ROS messages (`Image.msg` and `CameraInfo.msg`), as it is already well defined in ROS. Cameras are the only allowed optical sensor in the league, therefore we don't have to take care about depth cameras or laser-range-finders. Stereo cameras are allowed but currently not used by any team, as they often decalibrate during falling. Therefore they were not taken into account by the architecture for the moment.

4.3.2 Vision

The image can be used to extract visual odometry for the localization. No message has to be specialized for this since ROS already provides an odometry message (`Odometry.msg`). A less sophisticated approach for getting localization information is the visual compass. It is often used in RoboCup since the symmetry of the field is a problem for the localization. As compass sensors are not humanoid, this distinction can only be done visually. One common approach is to look for flamboyant markers outside of the field. Another approach is looking at the ceiling and comparing it with images that were recorded before the game. In any case, this results in an angle of orientation on the field. A simple message with an angle and a confidence are specified to transfer this data to the localization (`VisualCompassRotation.msg`).

Furthermore, there are four defined classes of objects to be recognized in RoboCup Soccer: ball, goals, obstacles and lines. Therefore it is simple to define four different pipelines, one for each of the classes. While theoretically, one message would be enough for each one of these, some extra messages for intermediate results have been defined, based on the current recognition algorithms in use.

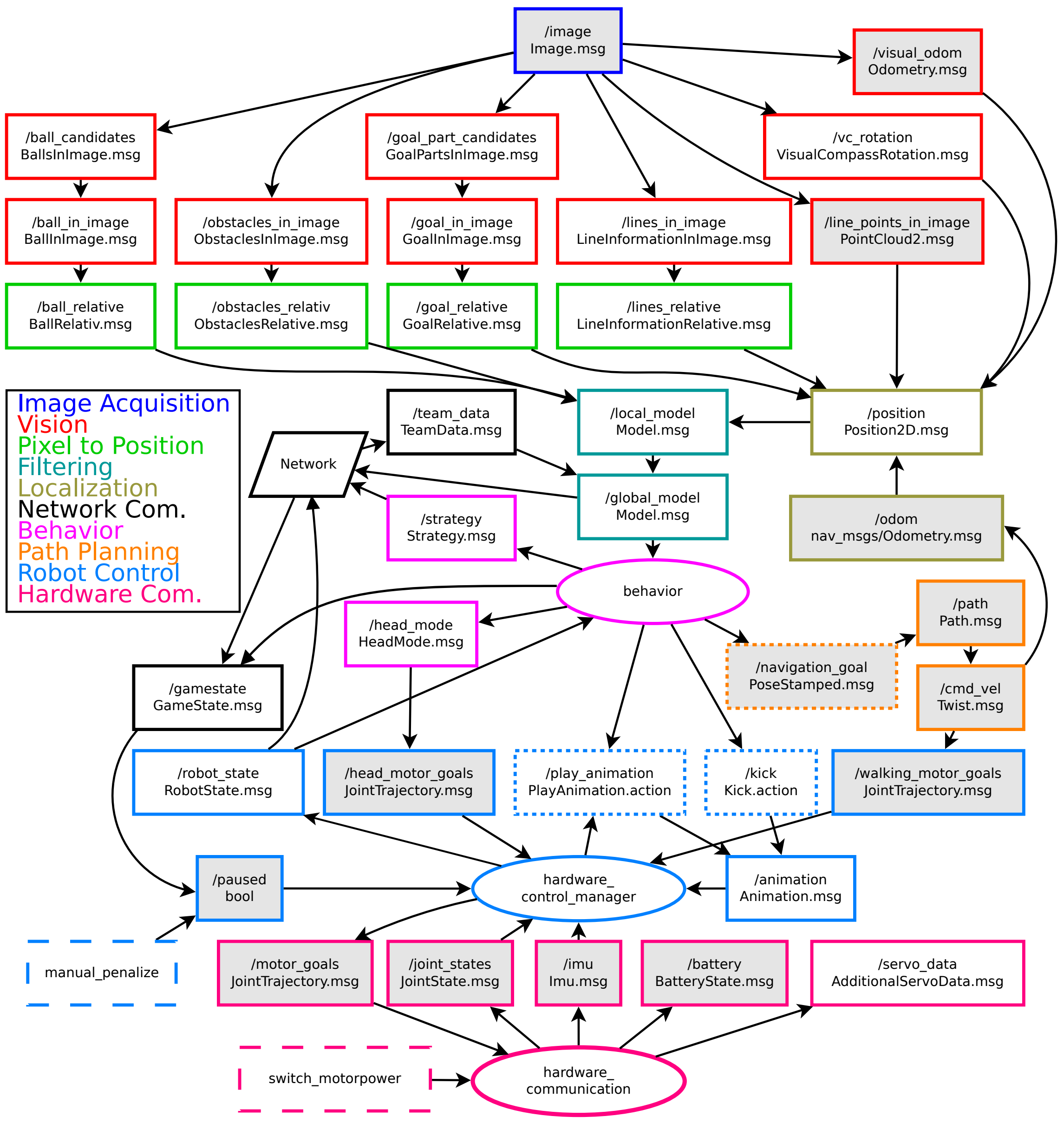


Figure 4.3: Proposed and ROS standard messages (rectangles) form together the data flow. The nodes (ellipses) are only shown for clarification and don't have to be implemented in this way. Actions are dotted and services dashed. Grey colored messages are ROS standard messages. The messages are grouped into different categories (colors). The vision pipeline (first three sections) transfers sensor data to information (*Sense*). These are provided to the localization and the world model. The model is used by the behavior to decide on the directives (*Plan*), which are handled by the robot control (*Act*). The hardware communication abstracts from the used robot platform.

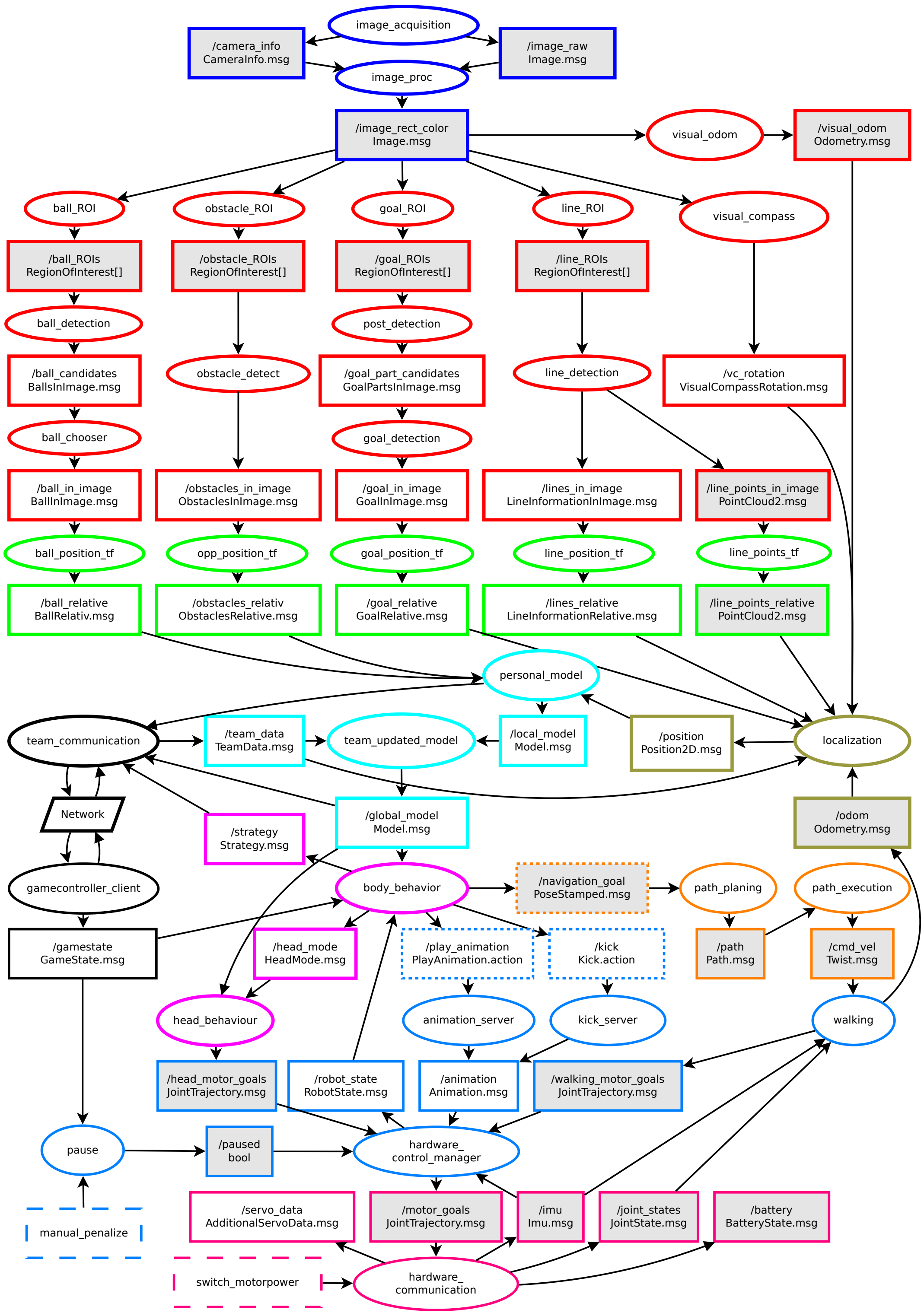

image_acquisition
/camera_info CameraInfo.msg
/image_raw Image.msg
image_proc
/image_rect_color Image.msg
visual_odom
/visual_odom Odometry.msg
ball_ROI
obstacle_ROI
goal_ROI
line_ROI
visual_compass
/ball_ROIs RegionOfInterest[]
/obstacle_ROIs RegionOfInterest[]
/goal_ROIs RegionOfInterest[]
/line_ROIs RegionOfInterest[]
ball_detection
post_detection
/ball_candidates BallsInImage.msg
obstacle_detect
/goal_part_candidates GoalPartsInImage.msg
line_detection
/vc_rotation VisualCompassRotation.msg
ball_chooser
goal_detection
/ball_in_image BallInImage.msg
/obstacles_in_image ObstaclesInImage.msg
/goal_in_image GoalInImage.msg
/lines_in_image LineInformationInImage.msg
/line_points_in_image PointCloud2.msg
ball_position_tf
opp_position_tf
goal_position_tf
line_position_tf
line_points_tf
/ball_relative BallRelativ.msg
/obstacles_relativ ObstaclesRelative.msg
/goal_relative GoalRelative.msg
/lines_relative LineInformationRelative.msg
/line_points_relative PointCloud2.msg
personal_model
team_communication
/team_data TeamData.msg
team_updated_model
/local_model Model.msg
/position Position2D.msg
localization
Network
/strategy Strategy.msg
/global_model Model.msg
/odom Odometry.msg
gamecontroller_client
body_behavior
/navigation_goal PoseStamped.msg
path_planing
path_execution
/gamestate GameState.msg
/head_mode HeadMode.msg
/play_animation PlayAnimation.action
/kick Kick.action
/path Path.msg
/cmd_vel Twist.msg
head_behaviour
animation_server
kick_server
walking
/head_motor_goals JointTrajectory.msg
/robot_state RobotState.msg
/animation Animation.msg
/walking_motor_goals JointTrajectory.msg
pause
/paused bool
hardware_ control_manager
manual_penalize
/servo_data AdditionalServoData.msg
/motor_goals JointTrajectory.msg
/imu Imu.msg
/joint_states JointState.msg
/battery BatteryState.msg
switch_motorpower
hardware_ communication

Figure 4.4: In this example, all messages are used. While this would allow an exchange of very small parts, it is probably not going to be implemented like this in reality. The real number of used nodes is lower, e.g. because some algorithms can not provide an intermediate result or a recognition of goals is not necessary for some localizations.

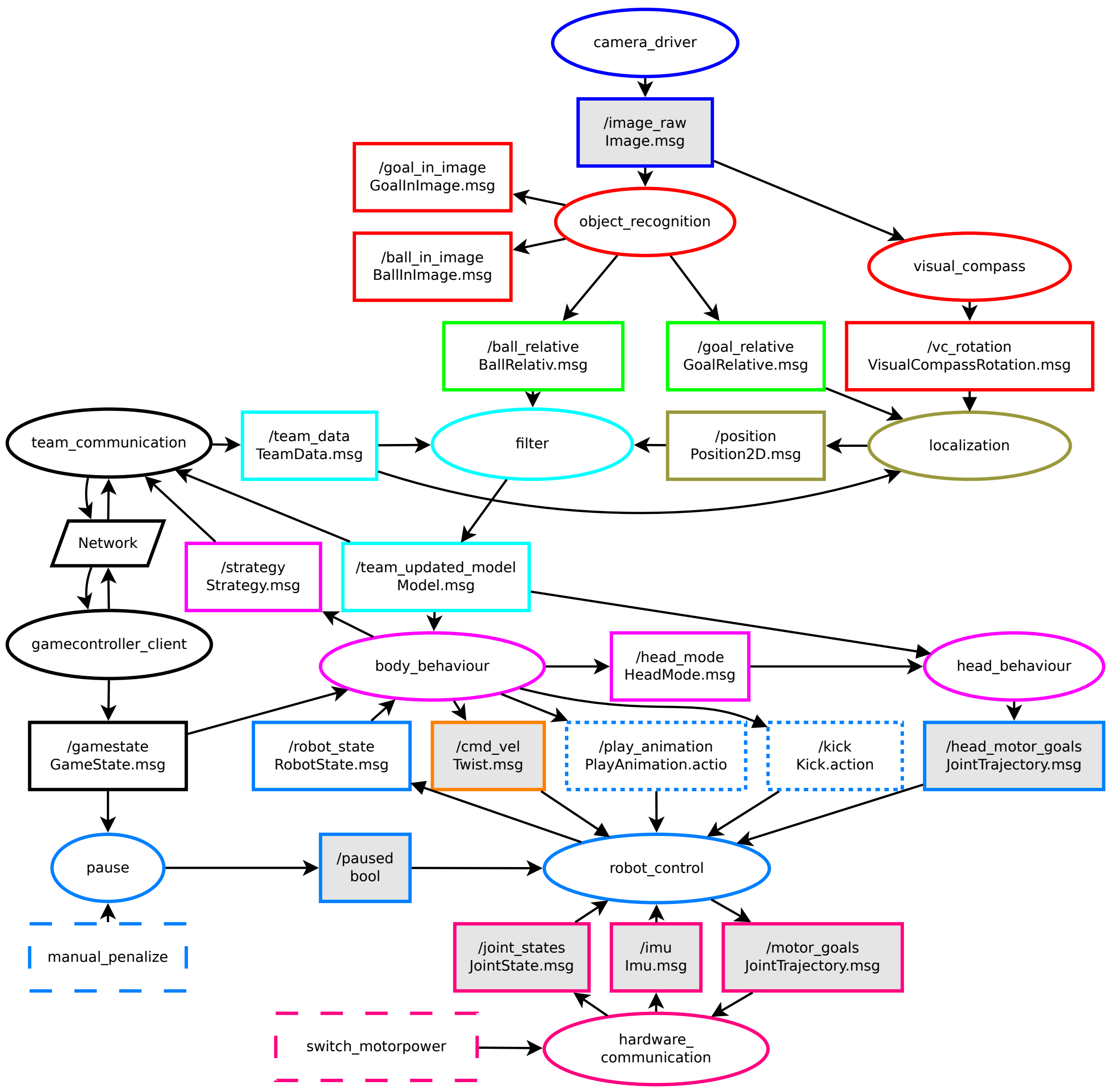


Figure 4.5: This is a more typical example of how the architecture would look when it is used by a team. The vision can't detect obstacles or lines. Furthermore, it directly outputs the relative positions. The positions in image space are only published for visualization. The localization is only based on detected goals and the visual compass. Filtering is done in only one node. The path planning is integrated into the behavior. Animations and kicks are handled by the same node.

Most of the teams have multiple steps in their vision algorithm. First regions of interest (ROI) are defined, for example the ball is usually expected below the horizon, as it stays usually on the ground in RoboCup Humanoid Soccer. Therefore computation time can be saved and false positives can be reduced by using ROIs, which is already a defined message in ROS (`RegionOfInterest.msg`).

Ball

For the ball, a message was defined specifying the center position, the size on the image and a confidence value (`BallInImage.msg`). If wanted, an intermediate result can be provided by publishing an array of ball candidates (`BallsInImage.msg`).

Obstacles

The obstacles can be of different kinds. Therefore the message (`ObstaclesInImage.msg`) includes an enumeration to tell if it's a team robot, an opponent robot, a human leg or something not recognized. Furthermore, the position on the image is defined with a rectangle. As in all specified vision messages, a confidence value is included.

Goal

In order to detect goals, parts of them have to be identified, this can be goal posts or bars. A first message (`GoalPartsInImage.msg`) was defined for publishing information about these goal parts. The result of the goal detection can be published using a second message (`GoalInImage.msg`), which can also be used to transmit information about not completely discovered goals. When viewing only one goal post, it is not clear on which side the center of the goal lies. Sometimes, educated guesses can be done, for example if the post was near the boarder of the frame, implicating that the other one lies probably outside the image. To enable this, a goal center position was added to the message.

Lines

There are two different ways to detect field lines. One is to detect only line points and to generate a point cloud out of it, for which a ROS message already exists (`PointCloud2.msg`). The other one is to identify crossings, line segments and the center circle as features. To enable this, a message (`LineInformationInImage.msg`) was defined.

4.3.3 Pixel to Position Transformation

To get the position of the objects in relation to the robot, a transformation from the image space to the relative space has to be done. To publish the results, messages for all objects were defined (`BallRelative.msg`, `ObstaclesRelative.msg`, `GoalRelative.msg` and `LineInformationRelative.msg`). They define the objects positions in relative space and a confidence value. The position is usually computed in ROS by using `tf`, but there are different other approaches in RoboCup, e.g. using the diameter of the ball on the image.

4.3.4 Localization

The localization uses the information about the detected goals and lines, as well as the visual compass and the odometry to compute the position. ROS already specifies a position message with covariance matrix, but still a new one was created (`Position2D.msg`). It consists of a position without covariance, a confidence value and a description of the standard for displaying absolute positions. Thereby teams that can not provide the complex covariance matrix, can use the easier confidence value. Furthermore, the standard for the absolute position ensures, that all teams use the center point of the field as a reference with the same directions for positive and negative values.

4.3.5 Filtering

The absolute position of the robot and the relative positions of the ball and obstacles are forming the personal model. This model is used to filter the obstacle and ball data over time, preventing the behavior to make decisions based on a single falsely detected objects. This model can be updated by the information from the team communication and published on a different topic. One message is defined (`Model.msg`) which consists of a `BallRelative.msg`, a `ObstaclesRelative.msg` and a `Position2d.msg`. This eases the comparison between filtered and non-filtered data. Furthermore, relative positions were chosen for the case where the robot does not know its absolute position on the field.

4.3.6 Behavior

The behavior consists often of two parts in RoboCup, a main behavior for the body and a subbehavior for the head. This partitioning lowers the complexity of each behavior. Usually, the body behavior decides where the robot should go and starts animations and kicks. It also provides its current action and role to the team communication, so that the other robots can be informed. Therefore a message was defined to publish the current strategic information (`Strategy.msg`). The head behavior only controls the position of the head due to its current mode which is set by the body behavior using a newly defined message (`HeadMode.msg`). The modes can be searching for ball, goal, localization features on and off the field, looking in a direction and don't move. Teams are not forced to split their behavior in two parts, but since this is a de facto standard, the `HeadMode` message was include.

4.3.7 Path Planning

ROS provides messages for path planning. The navigation goal is set by an action (using `PoseStamped.msg`). Other messages specify the path (`Path.msg`) and the resulting movement commands (`Twist.msg`). These messages were originally designed to be used by wheeled robots but are applicable in this context too, because it is possible to abstract from the movement of the motor joints to an abstract planar movement using a walking algorithm. This has not only the advantages that no further messages have to be defined, but also that it is possible to use a variety of ROS tools without further work. For example setting of navigation goals and visualization of the computed path can be done in RViz. Furthermore, already available navigation algorithms can be used.

4.3.8 Network Communication

The network is used for communication with the game controller and other robots in the team. The software tells the game controller that the robot is online and receives the data from it, which is published using a defined message (`GameState.msg`). It can be used by the behavior for strategy but also by the HCM to stop the robot during penalty. Furthermore, data from vision and behavior of other robots is shared using another message (`TeamData.msg`). This data can be used in the filtering for updating the model.

4.3.9 Robot Control

The robot control provides all necessary parts for controlling the joints of the robot. Communication in this part is mainly done by using the ROS standard message, to be able to use ROS functionalities, e.g. tf2 and RViz. An additional message (`RobotState.msg`) was defined to inform the rest of the software about the state of the HCM. This is for example important for the localization, as it needs to know if the robot fell down, in order to adapt its model.

For a humanoid robot there are different canonical and artificial states which can be published by a HCM. Each state is named with an identifier which is also used in the definition of the `RobotControlState` message, cp. section 8. The most basic states are related to the position of the robot. It can stand correctly (`CONTROLABLE`), it can be walking (`WALKING`), it can be on the way to fall (`FALLING`), it can be lying on the floor (`FALLEN`) and it can be on the way to get back up (`GETTING_UP`). Furthermore, there are two states related to starting and stopping of the robots software. The robot can just be started and still occupied on initialization (`STARTUP`) or on the way to shut down (`SHUTDOWN`). The last send status of the HCM before turning off is `HCM_OFF`. There are two RoboCup specific states, namely being penalized by the game controller (`PENALIZED`) or getting into and out of the penalty position (`PENALTY_ANIMATION`).

The robot can also play animations (`ANIMATION_RUNNING`) or record animations (`RECORD`). Another state, which is utile outside the game is when all the motors are turned of, for example when only the camera is used (`MOTOR_OFF`).

Furthermore, a message for playing animations has been defined (`Animation.msg`). It provides the motor positions and information if the animation is finished or if a new one is started. To start animations, two actions were defined. One to start statical animations (`PlayAnimation.action`) and one explicitly for the kick, where a dynamic goal can be defined. The kick uses a new message (`Kick.action`) which consists of two `Vector3.msg` for the ball position and the target.

The requested speeds, which are transmitted to the walking, are a `Twist` message. This is the normal ROS standard, originally used for wheeled robots, but it can also be applied here. Using this standard enables the use of other ROS packages, e.g. the `joy` package for remote controlling the robot. The walking can also listen to the robot control state in order to know when it has to stop walking, because the robot fell down.

4.3.10 Hardware Communication

A good abstraction from the used hardware is necessary to make the proposed architecture run on any robot. At this point it is crucial to use the ROS defined standards to enable the use of tools like RViz and MoveIt!. Therefore the node that is controlling the hardware has to publish the current motor values in a `JointState` message.

For accepting new motor goals, two message types would have been possible. Either a `JointState` message, which would allow one set of position, velocity and effort values, or a `JointTrajectory` message which describes a complete trajectory based on multiple points. Each point consists of position, velocity and effort. The Dynamixel servos, which are commonly used, are not able to accept a trajectory but only one goal point. However, there is an alternative firmware which can do it, cp. section 3.1.1. If the servo can accept only one goal position it is also possible to send the different points of the trajectory message one after another from the hardware node to the motor. The `JointTrajectory` message is the more powerful interface and was therefore chosen.

The Dynamixel servos are providing information about their temperature and voltage, which cannot be transmitted via the `JointState` message. Since this motor is widespread in the league, an additional message (`AdditionalServoData`) was defined to publish these values. The hardware communication also has to publish all other sensor data, except the camera image. The possible sensor types are very restricted due to the rules. Therefore it can only be an IMU. There is already a well defined ROS message (`Imu.msg`) for the IMU which is also used. Further information on the robot are more specific and can be implemented by every team itself, for example, information about pressed buttons.

5 Implementation

The parts of the architecture which are expected to be used by all teams, were implemented as packages, so that other teams switching to ROS would not have to reimplement these. They can take these packages and will get all necessary basic capabilities, on top of which they can implement their own vision and behavior algorithms. Obviously it is also possible to use only parts of these packages and reimplement others, for example the walking. The nodes were written in *rospy* as far as possible, since it has a good readability [Sanner et al., 1999]. The intention is to rather sacrifice performance than having code which can not be easily understood or reused by others. Furthermore, ROS makes it very simple to distribute the code on different computers. For example, by installing an additional single-board computer. Thereby, performance is less an issue.

The packages for the whole HL are presented first (section 5.1). Afterwards the packages which were implemented especially for the Hamburg Bit-Bots are examined in section 5.2.

5.1 Humanoid League Packages

All package that are presented in this section were implemented to be usable by any team in the Humanoid League. They are all fully compatible to the defined architecture.

5.1.1 Messages Package

All the defined messages were put in one package. Thereby this package has very few dependencies, only to `message_generation` and ROS defined message packages. This allows easy use of this package, even for teams who don't want to completely follow the proposed architecture. The complete list of all message definitions can be found in chapter 8.

5.1.2 Game State Receiver

For the implementation of the game_state_receiver, the already existing package from the WF Wolves was used. Only small adaptions were needed. The message was changed to the definition in the humanoid_league_msgs package and the team and robot IDs were changed to parameters.

5.1.3 Team Communication

During the last years, the team FUmanoids established a communication standard in the league called Mixed Team Communication protocol (Mitecom) [8]. This is important because robots need the ability to communicate with robots from other teams. Beginning this year, this ability is mandatory to participate in a drop-in challenge, where robots from different teams play together [9]. Furthermore, the player count will increase in the future up to eleven [11], forcing teams which can not effort so many robots to build joint teams. Therefore having a standard for communication is important. The existing Mitecom was expanded to provide additional information about positions of other robots on the field and to add the ability to share simple strategies between different robots.

5.1.4 Speaker

The speaker package provides an easy way to use the Linux text-to-speech program *espeak* in ROS. Vocal output is needed for two reasons: First, as a way to provide information to the robot user. Either for telling the robot's next action, or to get a humans attention to a problem, for example, an emergency shut down. Secondly, the RoboCup wants to use natural team communication in the future. While this package is not sophisticated enough for team communication, it can still be used for first tests in this area. The implementation is done by a simple node which has three queues with different priorities. Messages can be send to the node, including a text and a priority. The node executes *espeak* with the texts, in an order depending on their priority. Different options, e.g. volume, can be adjusted using dynamic reconfigure.

5.1.5 RoboCup Visualization

ROS already provides two powerful tools for visualization, e.g. RViz. Still, there are special use cases in RoboCup which are not provided directly by ROS. For some of them, plug-ins were implemented during this thesis, to provide the most important functions from the start and to motivate others to provide additional plug-ins in the future. The implemented plug-ins are described in the following.

rviz_marker

On of the most important features is to visualize the beliefs of a robot related to its position on the field and the recognized objects, like ball and lines. These can be displayed in RViz via markers. To do this, messages of type `Marker` have to be published. A node was implemented which subscribes on the relative positions of objects and publishes the corresponding marker messages. Thereby, only this node have to be started and the position of ball, goal posts and obstacles are visualized in RViz (figure 5.1).

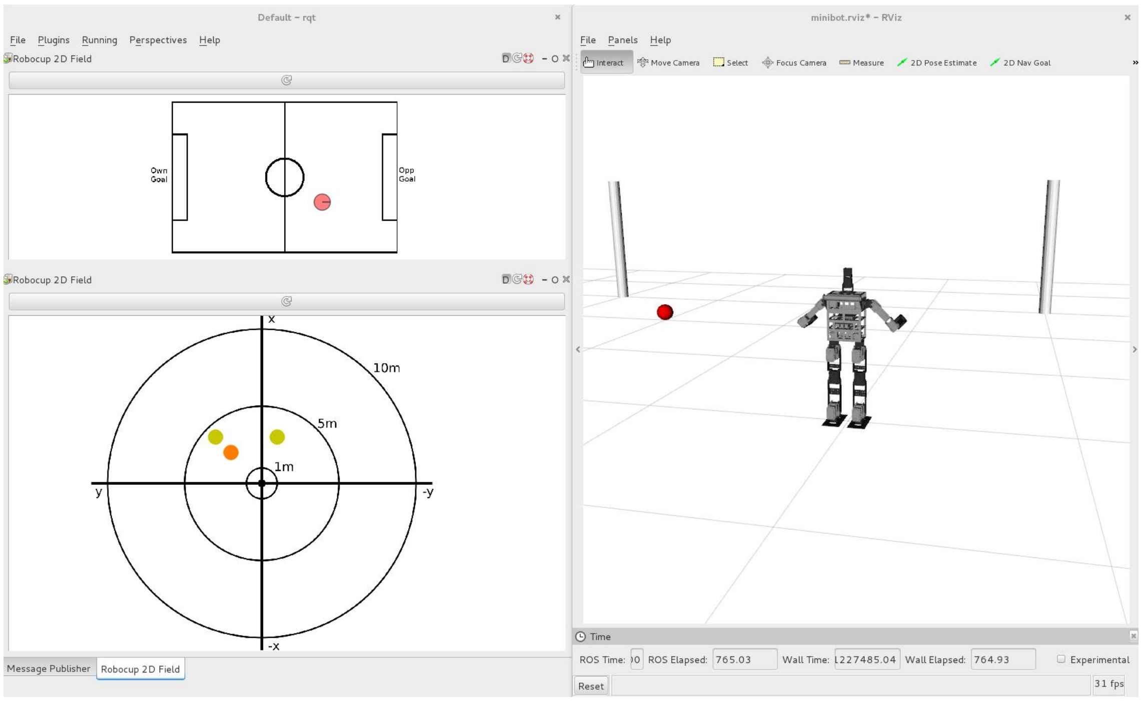


Figure 5.1: The screenshot shows the different visualization tools compared to each other. The `field_rqt` plug-in (top left) shows the absolute position of the robot on the field. Relative positions of objects are visualized in the `relative_rqt` plug-in (bottom left) and in RViz (right), using markers generated by `rviz_marker`.

field_rqt

While using Rviz to visualize the recognized objects provides a nice interface, often a more simplistic interface is better to get a fast overview. Therefore a rqt plug-in was written, which displays a 2D image of the field (figure 5.1). On this, shapes are painted, which show the beliefs of the positions of the robots itself, the ball and other robots. It can be displayed together with other rqt widgets at the same time and needs only a small display space.

relative_rqt

This rqt plug-in provides the same data as the `rviz_marker` package but in a 2D top down view (figure 5.1). It has the advantage of a clearer interface and it takes less display space when used.

image_marker

While the first two plug-ins, described above, are useful to visualize the beliefs which result from the object recognition, this plug-in is for the object recognition itself. Displaying the camera image is already possible in RViz and therefore it would be a feasible solution to provide an image to RViz where all recognized objects are already marked. Still, the use case is often to display only certain information of the object recognition. This can be achieved without implementing an additional plug-in by using dynamic reconfigure to activate and deactivate markings. Thereby only one node is needed which subscribes to the camera image, draws shapes depending on the current parameters and provides this image on another topic which is then displayed in the standard image viewer plugin.

5.1.6 Simulator

ROS already has a well connected simulator: Gazebo. The communication works directly with the standard messages, therefore no additional work has to be done to interface the simulator. Still, a Gazebo world with a field, goals and a ball is needed in order to simulate a RoboCup Soccer game. Team NimbRo already created such a world for their ROS architecture and released it open source. Unfortunately, the package included some NimbRo specific files and had dependencies. Therefore the world and object files were extracted and put together in a new package. Every team using the architecture needs only to provide an URDF for their robot platform and will get complete simulator support (figure 5.2).

5.2 Hamburg Bit-Bots Packages

In this section, the packages are presented which were especially written for the Hamburg Bit-Bots. Nevertheless, these packages can be used by other teams. A graph of the implemented nodes is shown in figure 5.3.

5.2.1 Robot Control

The tasks for controlling the basic robot abilities was described in section 4.2. These tasks were implemented into three different nodes for HCM, walking and animation. Their realization is further investigate in the following.

Hardware Control Manager

The main task of the HCM is to evaluate the current state of the robot and to decide what actions should be taken. A set of general useful states were already defined in the `RobotControlState` message, section 4.3.9. A common approach to implement them is using a state machine. Given that the number of states is low, a hierarchical state machine is not needed.

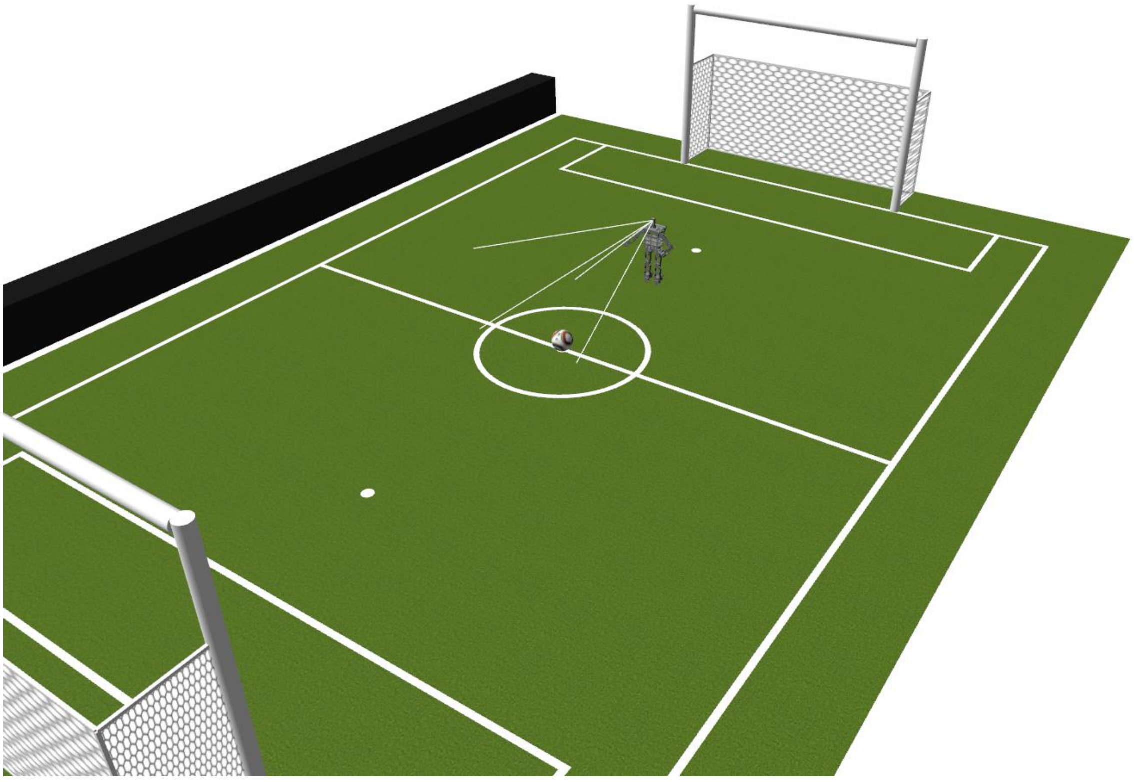


Figure 5.2: The Humanoid League environment which was extracted from the NimbRo code base. The Minibot robot was integrated using the modeled URDF, cp section 5.2.3. The lines in front of its head visualize the robot's field of view.

The resulting state machine is shown in figure 5.4. In the following, names of states in the message are written in upper case, e.g. `GETTING_UP`, and concrete implementations are written in camel case, e.g. `GettingUp`. Methods are lower case, for example `entry()`.

We create the following additional states to make the state machine more understandable and closer to the previous functionalities of the Bit-Bots motion: `PENALTY_ANIMANTION` is divided into `PenaltyAnimationIn` and `PenaltyAnimationOut`, to easier differentiate between them. `GETTING_UP` is divided into `GettingUp` and `GettingUpSecond`, because the getting up procedures of the Bit-Bots consists of two parts. `WALKING` is divided into `Walking` and `WalkingStopping` to show the fact that the walking tries to stop, for example to play an animation. The walking has always to do a few steps to come to a safe stop. `SHUTDOWN` is divided into `ShutDown` and `ShutDownAnimation`, which is showing that the motion is on the way to shut down by playing a sit down animation.

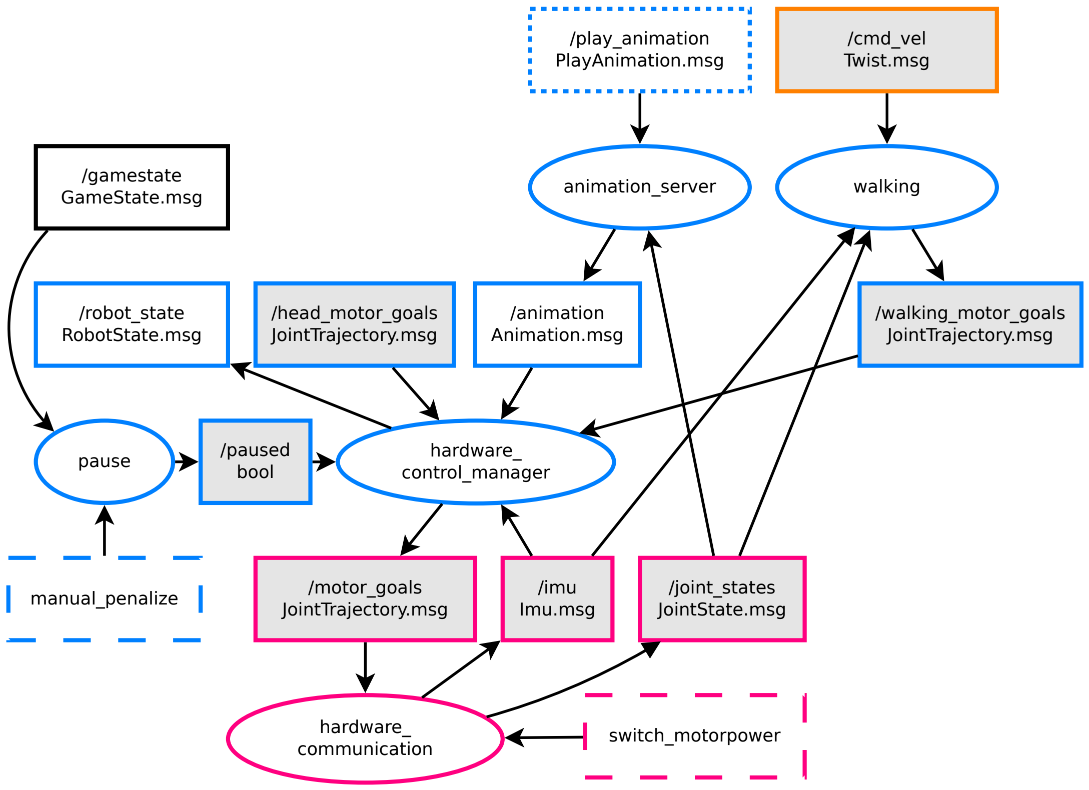


Figure 5.3: The implemented nodes for the robot control and hardware communication of the Hamburg Bit-Bots. All functionalities from the previous motion process were transfered but restructured in single independent packages.

To implement this state machine, first the classes `AbstractStateMachine` and `AbstractState` were defined. The AbstractState class defines three main methods: `entry()`, `evaluate()`, `exit()`. The `entry()` method is called once when the state machine transitions to this state, to instantiate this state. The `evaluate()` method checks if the state should change and returns the state to which the state machine should transition. Finally, the `exit()` method is called on leaving this state.

The `AbstractStateMachine` defines the two methods `evaluate()` and `set_state(state)`. When evaluate is called, it first checks if it should shut down and acts accordingly by going to state `ShutDownAnimation`. If this is not the case, the current active state is evaluated by calling the state's `evaluate()` method. This returns either a state to which the state machine should switch or `None` if it should stay in this state. If the state machine should switch, the `set_state(state)` method is called. In `set_state(state)`, first the `exit()` method of the current state is called, the current state is changed to the new state, the current state is published and the entry method of the current state is called. This process is visualized in figure 5.5.

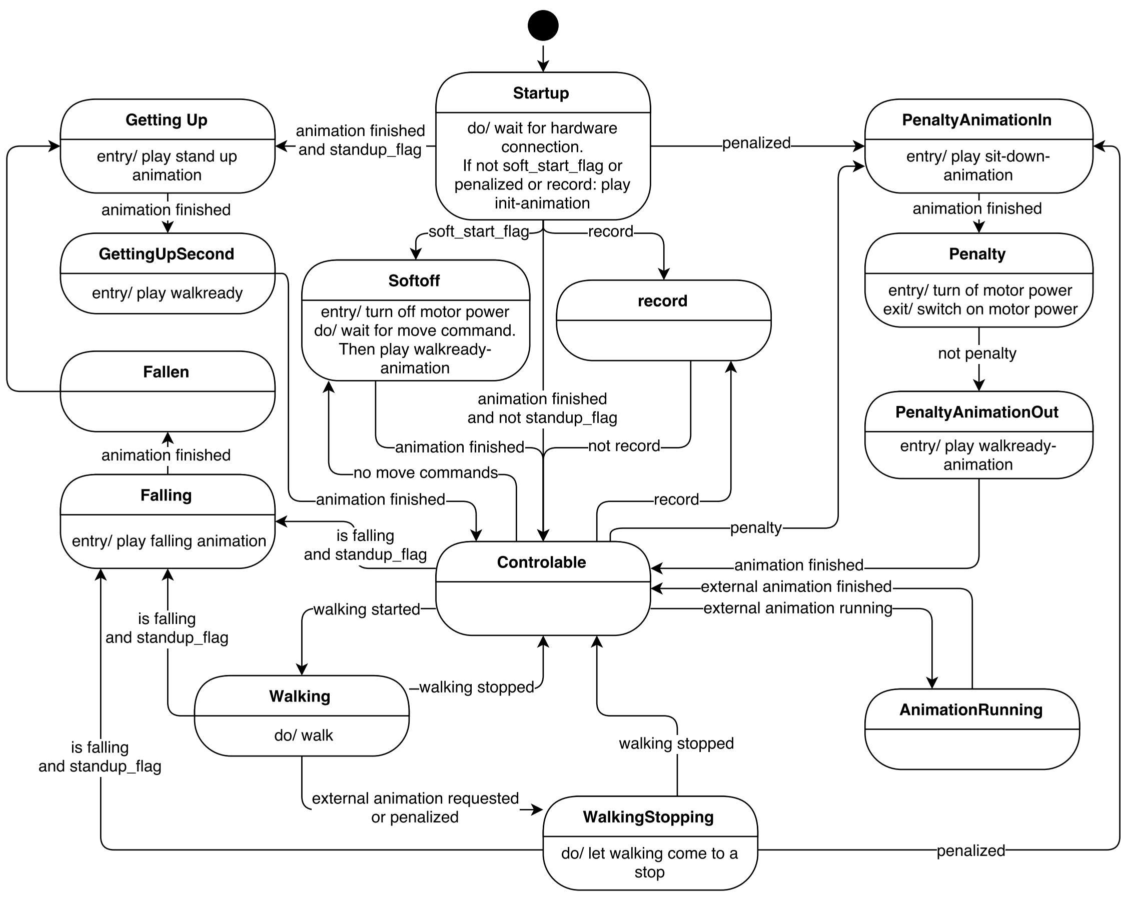


Figure 5.4: UML state diagram of the HCM state machine, without shutdown connections for better readability. Conditions for state transitions are written on the arrows. Each state is defined by a name (top) and its actions (bottom) which can be done in the `entry`, `exit` or `evalutation` (tagged with *do*) method.

The `AbstractStateMachine` is implemented by a `HcmStateMachine` class and the `AbstractState` class is implemented by a set of classes representing the states of the state machine. To share information between these classes a singleton is introduced which holds all the necessary data, e.g. the gyro and accel values, and has some utility methods. Furthermore, it holds an object of type `FallChecker`, which provides methods to know if the robot is fallen or falling.

This completes the state machine, but a connection to the other ROS nodes is necessary to get information and to publish the current HCM state. Therefore the class `HardwareControlManager` initiates a ROS node. It creates an object of type `HcmStateMachine` and calls its evaluate method periodically. To get the current servo positions and IMU values, it subscribes to the related topics.

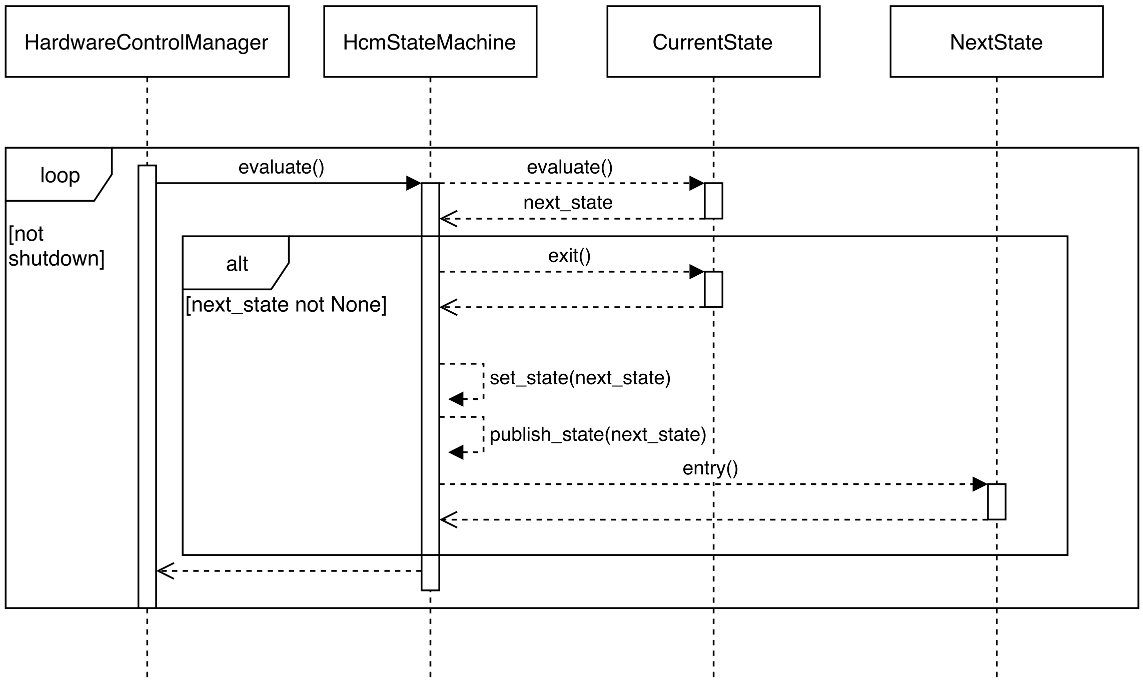


Figure 5.5: UML sequence diagram of the HCM main thread. `CurrentState` and `NextState` represent any unequal subclasses of the `AbstractState`. The `evaluate()` method of the current state returns the next state or `None` if the state should not be changed. If a new state is returned, the current `exit()` method is called. Afterwards the `entry()` method of the new state is called.

In the callback methods, the information is written into the VALUES object and is thereby accessible by the states of the state machine. When the node receives motor goals from the walking, animation or the head, it decides directly inside the callback function if these goals can be applied in the current state. Furthermore, the values in VALUES are updated correspondingly, e.g. to tell the state machine that an animation is playing or the walking is active.

hcm_visualization

In some cases it is difficult to recognize in which state the HCM state machine is at the moment, especially if the states are changing often. An additional plug-in provides a visualization of the states, the state transitions and a history of the previous states (figure 5.7). It is loosely based on the node graph plug-in which provides a similar GUI. The `InteractiveGraphicsView` class of QT is used to have a sort of canvas on which geometrical shapes can be drawn. To easily draw the state machine, a similar approach as in the node graph plug-in is used. First, the dot code which represents this state machine is generated and then built in methods to

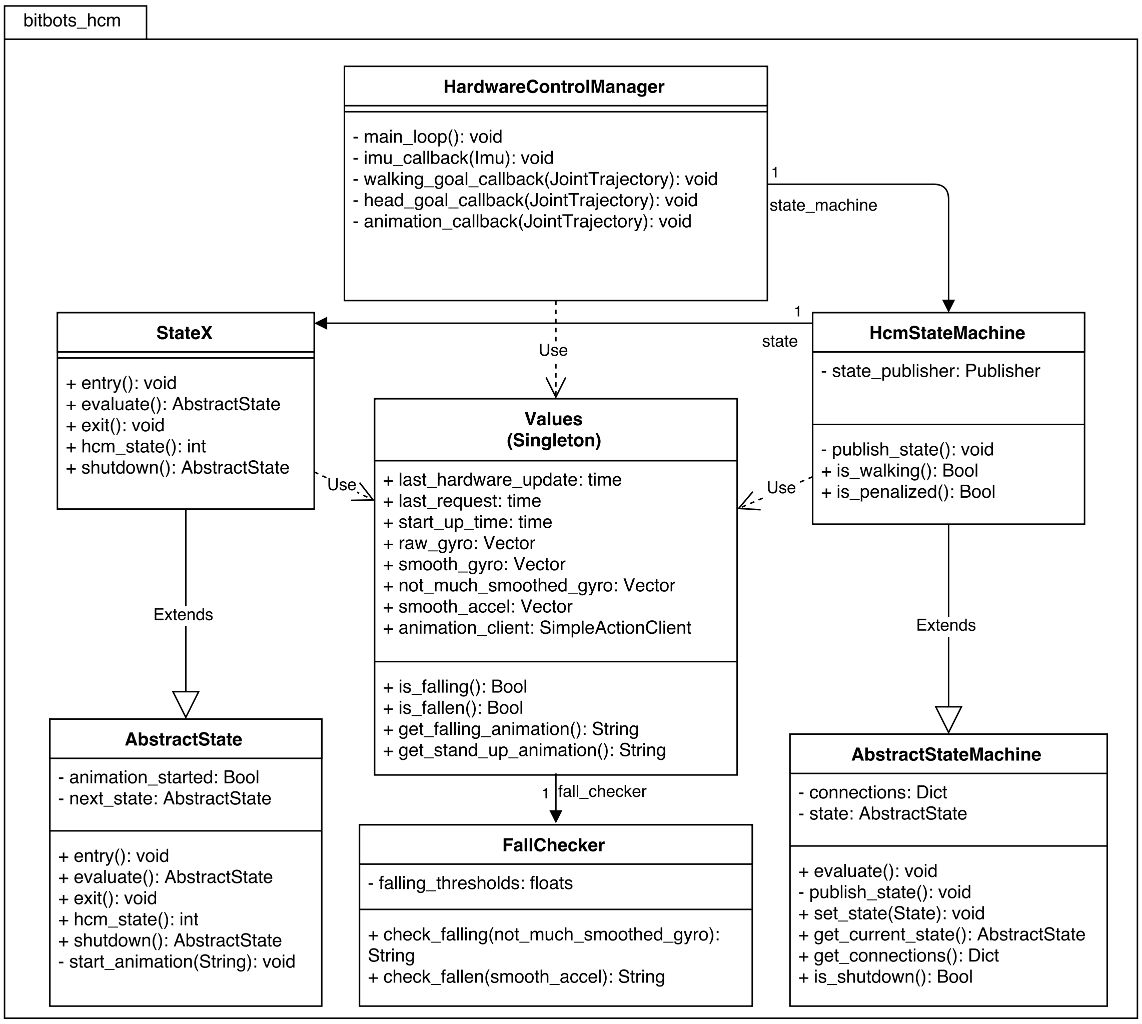

bitbots_hcm
HardwareControlManager
- main_loop(): void
- imu_callback(Imu): void
- walking_goal_callback(JointTrajectory): void
- head_goal_callback(JointTrajectory): void
- animation_callback(JointTrajectory): void
1
state_machine
1
state
Use
StateX
+ entry(): void
+ evaluate(): AbstractState
+ exit(): void
+ hcm_state(): int
+ shutdown(): AbstractState
Use
Values
(Singleton)
+ last_hardware_update: time
+ last_request: time
+ start_up_time: time
+ raw_gyro: Vector
+ smooth_gyro: Vector
+ not_much_smoothed_gyro: Vector
+ smooth_accel: Vector
+ animation_client: SimpleActionClient
+ is_falling(): Bool
+ is_fallen(): Bool
+ get_falling_animation(): String
+ get_stand_up_animation(): String
Use
HcmStateMachine
- state_publisher: Publisher
- publish_state(): void
+ is_walking(): Bool
+ is_penalized(): Bool
Extends
Extends
AbstractState
- animation_started: Bool
- next_state: AbstractState
+ entry(): void
+ evaluate(): AbstractState
+ exit(): void
+ hcm_state(): int
+ shutdown(): AbstractState
- start_animation(String): void
1
fall_checker
FallChecker
- falling_thresholds: floats
+ check_falling(not_much_smoothed_gyro): String
+ check_fallen(smooth_accel): String
AbstractStateMachine
- connections: Dict
- state: AbstractState
+ evaluate(): void
- publish_state(): void
+ set_state(State): void
+ get_current_state(): AbstractState
+ get_connections(): Dict
+ is_shutdown(): Bool

Figure 5.6: Simplified UML class diagram of the bitbots_hcm package. The `HardwareControlManager` implements the node and provides callback methods. It has an object of the type `HcmStateMachine`. This state machine consists of different states which all have a class which extends `AbstractState`. All states, the state machine, and the ROS node are using a singleton called `Values` to exchange information. This singleton also has an object of type `FallChecker`, to examine if the robot is falling or fallen.

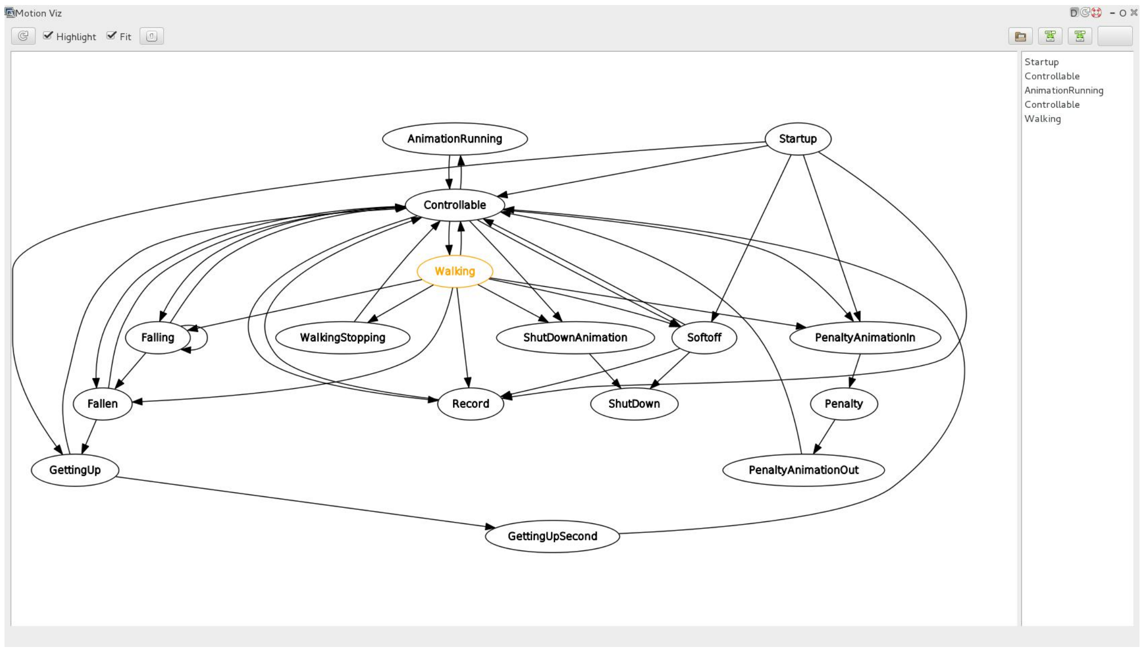


Figure 5.7: The graph shows the different states of the state machine and their transitions. The current state is colored and a history of the active states is provided on the right.

draw this dot code are used. The current state is colored in orange and it is possible to hover over a state with the mouse to color the states which have transition to or from this state. Additionally a list is displayed on the right side which lists the last active states. This is very useful since states might change fast.

Walking

The walking was already capsuled in the old framework. It is provided by the `ZMPWalkingEngine` which allows to set the walking velocities, start and stop the walking and to get the next goal pose for the robot with the `process()` method. Around this, a ROS wrapper has been written which holds an object of this type and calls its `process()` method periodically. The resulting motor goals are published afterwards. The current motor positions and the IMU values are provided in class variables from the call back function of their subscribers. The walking engine itself is based on the ZMP walking [Vukobratović and Borovac, 2004]. Its core is written in C++ and it provides a Python interface via Cython. The functionalities of the walking engine itself were not changed.

Animation

The animation is done in a simple action server which can cancel running animations if needed. Therefore the animation node has an object of type SimpleActionServer. A method `execute_cb()` is defined, which is called in a different thread by the `SimpleActionServer` whenever a new goal is received. In this method, the corresponding motor goal values for the animation are generated in a loop. At each generation point, it is checked if a new goal is available on the action server. If this is the case, the thread decides if the current goal should be canceled or not.

The generation of motor goals for a given animation is done by an interpolation over saved keyframes. To do this, first an interpolation method is chosen, e.g. linear or cubic hermite [De Boor et al., 1978]. Then the keyframes are used as splines and the curve for each motor is interpolated. This is strongly based on the pre-existing code. The animations are saved in *yaml* files, which are recorded either by an external tool or are written by hand. Such a file consists of a header and a finite number of keyframes. Each keyframe has a duration, pause, and motor goal positions for a set of motors. These files can be parsed into objects of the class `Animation`. To get smooth trajectories for the motor goals, the `Animator` class provides a `update()` function. This function has to be called periodically until the animation is completed.

Pause

The pause node subscribes to the information of the game controller client and provides a service to penalize the robot manually. These two sources of penalties are merged into one and published on the `/paused` topic. The manual penalize can overwrite the penalty from the game controller, but not the other way around. This is useful to reset penalty states if the wifi connection is distorted during a game.

5.2.2 Hardware Communication

The most common board for controlling the motors in the Humanoid League is the CM-730 (cp. section 3.1.2). It is also used in the Minibot and thereby it is a good choice to implement the hardware abstraction for this board, as it can be used both for this thesis and by other teams. In the old framework, code for accessing the board was already present but mixed with code for animation and hardware control. The related parts of the code were extracted and put together in one package. A Cython class for the CM-730 was written which provides an interface for the different classes which are used for the low-level communication.

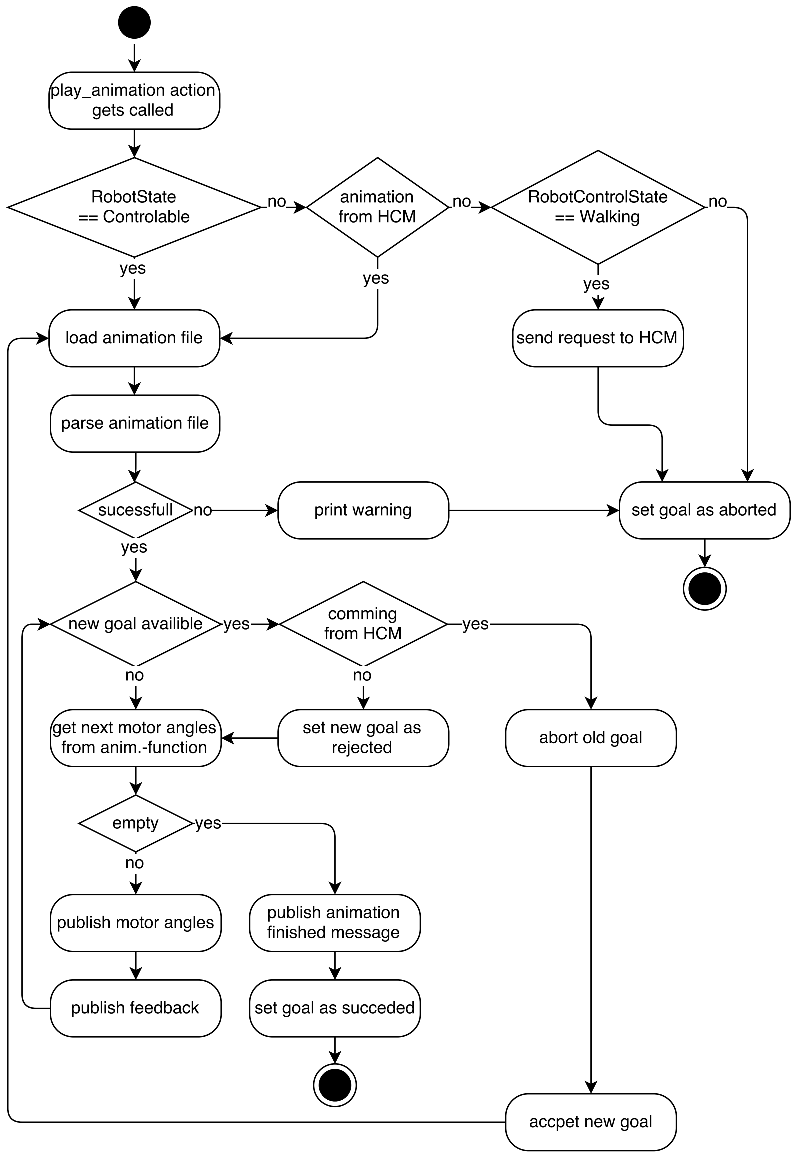


Figure 5.8: UML activity diagram showing how a call of the `PlayAnimation` action is handled. First, the current robot state is evaluated, to see if an animation can be run. If this is the case, the animation is loaded and parsed. Then the animation function is called in a loop to get the next motor goals, which are then published. This loop can be intersected by a new goal that comes from the HCM. If this is the case, the old animation is aborted and the new one is started.

5.2.3 Minibot URDF

Many ROS built in features like RViz or tf require an URDF model of the robot. The robot that was used in this thesis, Minibot, was constructed by the Hamburg Bit-Bots themselves. Therefore an URDF had to be made (figure 5.9). The construction of the robot was done in the CAD program Autodesk Inventor. This program allows assembling multiple parts together inside the program. Thereby it was easy to export a single mesh for each joint, which consists of multiple real parts. This makes the URDF simpler, as only one visual file per link is needed. Two versions of meshes were provided to the URDF per link. One high-resolution file for the visual representation and a second low-quality version for the collision model. The second version was modified using Meshlab to reduce the number of polygons and to replace parts of the object with bounding boxes. This increases the performance of the model during simulation. Additionally to the tf links for the movable parts of the robot, some fake links were provided. These point to the position of the camera, the end of the arms and the tips of the feet. This increases the usability of this model.

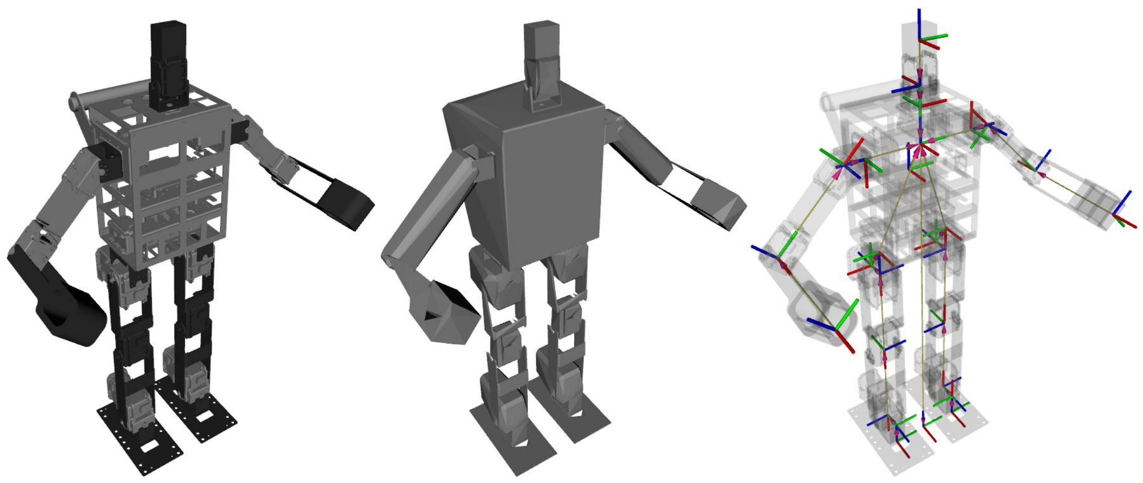


Figure 5.9: Three different visualizations of the Minibot URDF model. The visual representation (left) is used for displaying the robot in RViz or Gazebo since it resembles it the most. The collision model (center) is a down-sampled version of the visual model. It is used by Gazebo for the collision detection since it has fewer vertexes which accelerates the detection of collisions. The `tf` structure (right) shows the coordinates systems of all joints and the parent-child relationship between them.

6 Evaluation

In this chapter, the previously presented approach and its implementation are evaluated to investigate if it is feasible. First, the architecture is compared to others in section 6.1. Then, the transfer process is investigated in section 6.2 and the performance is tested in section 6.3. Afterwards, the building of a community around this architecture is investigated in section 6.4. Finally, the influence on the league is examined in section 6.5.

6.1 Architecture Comparison

The two other frameworks from the HL, which are open source and based on ROS, are chosen, for comparison. These frameworks are from the teams NimbRo and WF Wolves, which were already mentioned in section 1.2. First, the architecture of NimbRo is examined and compared. Afterward, the one of the WF Wolves. Finally, another comparison with the old architecture of the Hamburg Bit-Bots is done in order to investigate if an improvement can be observed.

6.1.1 NimbRo

The NimbRo / igus robot and its software architecture stayed basically the same between both robots [Allgeuer et al., 2013] [Schwarz et al., 2013] [Allgeuer et al., 2016]. Therefore only one comparison was made with the newest version. An overview of the architecture can be seen in figure 6.1.

While this figure gives a nice overview of the present software parts, it does not actually show the present nodes and topics. Therefore a second diagram, figure 6.2, was created based on the available code. This shows that the number of actual ROS nodes is lower than one would expect after seeing figure 6.1. This is due to the fact that the motion modules are part of the robot control node, the localization is actually part of the vision node, and tool nodes are not displayed in the second figure.

This low granulation leads to the problem that only large pieces of the NimbRo architecture could be exchanged, e.g. the complete vision, not only the ball recognition. The same problem persists with the motion modules which are representing the different direct controls of the robot, e.g. walking and fall protection. These are modularized but use a particular plug-in system with a shared robot model object which is also following no standard. Thereby these parts are modular but can only be used in this specialized context and not with arbitrary ROS software. This is a disadvantage compared to the proposed architecture, which has a high granulation

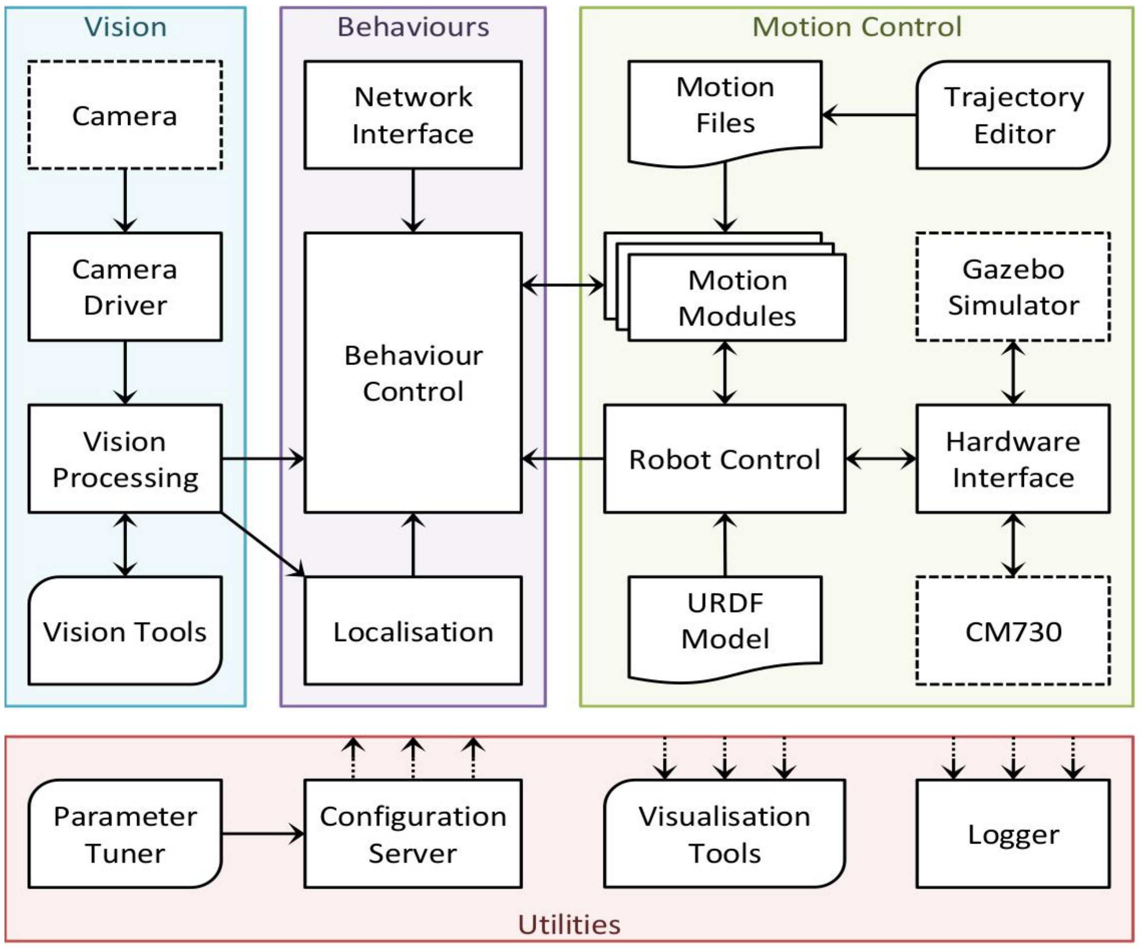


Figure 6.1: The architecture of the NimbRo/igus robot as described in [Allgeuer et al., 2016]. The software is split into different modules which are grouped in categories. Unfortunately, these modules are not one to one represented by a ROS node, cp. figure 6.2

of nodes which can be used with arbitrary ROS code. A potential advantage of this software is the performance, since the number of message transfers is far lower than in the proposed architecture, especially in the motion section.

The low number of nodes also leads to a low parallelism, since every node is a process. Still, a node can implement multiple threads to increase the parallelism. This was not done for the robot control node, since all plug-ins get called one after another in a loop.

The hardware abstraction of the NimbRo architecture is on a high level. This means, parts like behavior and vision could run on different robot but lower parts, like walking or head control doesn't. This results also from the use of the motion plug-in system with the robot model. In the proposed architecture, only the lowest node needs to be exchanged and all other still work if they are implemented properly.

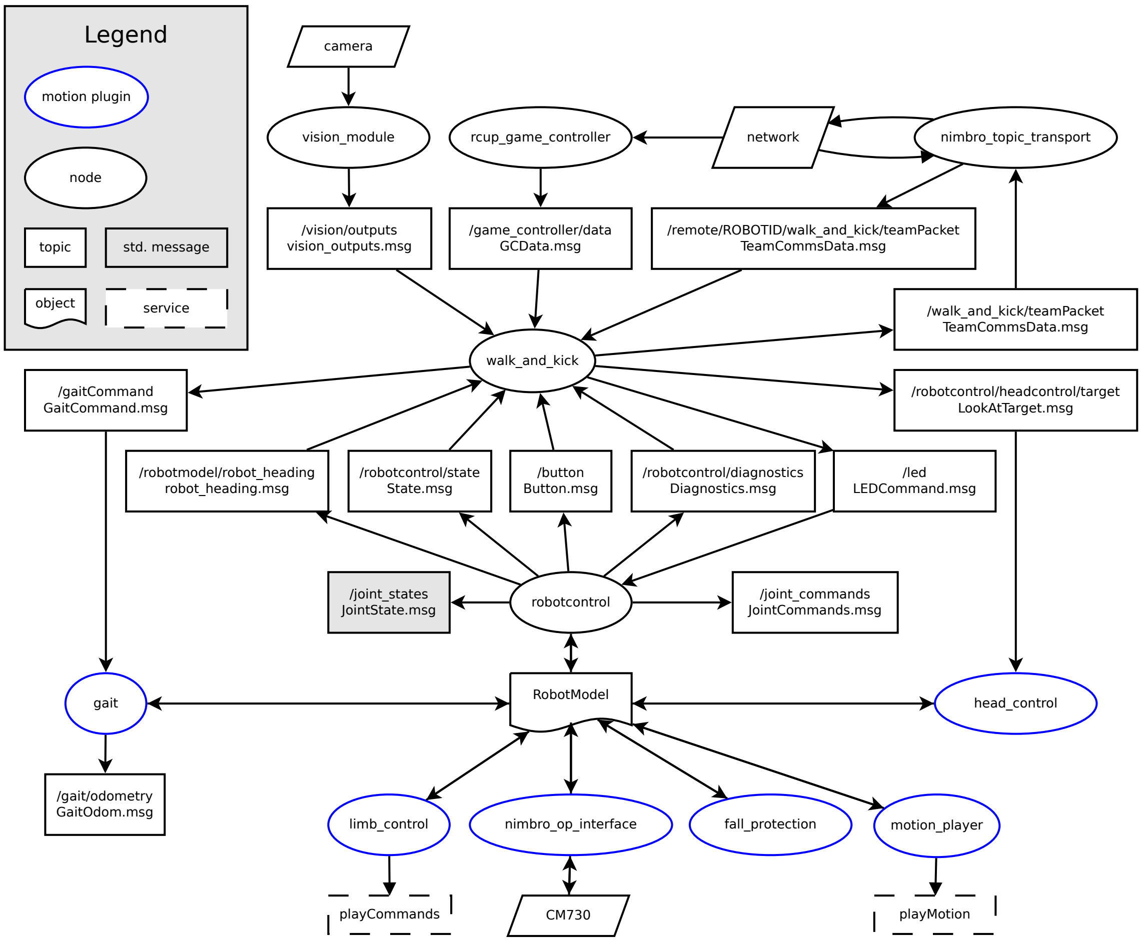


Figure 6.2: The ROS nodes and topics of the NimbRo architecture based on the code, without displaying tools. Three main nodes are present: `vision_module`, `walk_and_kick` and `robotcontrol`. The `vision_module` node does the object recognition and the localization. Decisions are taken by the `walk_and_kick` node. The `robotcontrol` node provides all methods to control the robots joints. It is split up into different *motion plugins* that share a common *RobotModel*. The team communication is done using `nimbro_topic_transport` which transports the local `TeamCommsData` message to the team mates and provides the received data to `walk_and_kick`.

NimbRo		ROS	
GaitOdom.msg		Odometry.msg	
Header	header	Header	header
uint32	ID	string	child_frame_id
Pose	odom	PoseWithCovariance	pose
Pose2D	odom2D	TwistWithCovariance	twist
GaitCommand.msg		Twist.msg	
float32	gcvX	Vector3	linear
float32	gcvY	Vector3	angular
float32	gcvZ		
bool	walk		
uint8	motion		

Figure 6.3: Comparison of messages defined and used in the NimbRo software and the standard message defined by ROS which should be used in this case. `Odometry` is used to provide information about the walked distance from the walking algorithm to the localization and `Twist` is normally used to provide a directive of velocities to the walking.

The used messages are almost completely not ROS standard. This can be very reasonable in some cases but messages were implemented, for which standards already exist, see figure 6.1.1. This makes modules not compatible with other ROS packages. In the presented example, the walking uses different messages for accepting commands and providing odometry information. Thereby the normal ROS navigation stack would not be usable. In the proposed architecture, ROS standards were used as far as possible and messages were only defined for RoboCup Soccer specific information. Thereby this architecture would also be able to use for example the ROS navigation stack.

In addition to the specialized messages, some of the ROS internal structures were reimplemented. Normally, parameters are handled in ROS using the parameter server and dynamic reconfigure, see section 2.3.3. Team NimbRo implemented their own parameter server because the `dynamic_reconfigure` package "was insufficient for the task as it did not allow the reconfigurable parameters to be shared globally between the various nodes of the system."[Allgeuer et al., 2013]

They did not state any examples for these parameters and therefore it remains unclear which parameters have to be configurable at runtime and are used by more than one of their nodes. Even if this feature is necessary, it should have been added to the `dynamic_reconfigure` package rather than implementing a complete new package for this. The resulting problem is that all NimbRo packages have a dependency on this specialized parameter server. Using one of their nodes would also mean to use the parameter node and ending up with two different parameter systems. This is a high barrier in using their code, since it would result in having two different systems for handling parameters. Furthermore, a user needs to learn how to use the NimbRo parameter server. In the proposed architecture this problem does not arise as only the standard parameter handling methods are used. All reconfigurable parameters are also node specific. For visualization RViz and rqt were used, in relation with programmed plug-ins. This is good since the standard tools are used and their functionalities are expanded.

In summary, the main advantage of the proposed architecture in comparison to the NimbRo's is that using single packages of it is easier as there are fewer interdependencies and ROS standards are used. The NimbRo architecture could have a higher performance since there are fewer messages transferred and it is completely written in C++. On the other hand, it is less parallel and therefore only has an advantage on CPUs with a low core number.

6.1.2 WF Wolves

The WF Wolves developed their software completely independently from the NimbRo architecture, even though they are using a modified version of their robot platform. No papers were released about this platform, thereby all following analysis is only based upon their open source code. An overview of their nodes and topics is given in figure 6.4.

The number of nodes is higher in comparison to the NimbRo code but less than in the proposed architecture. The level of hardware abstraction is basically the same as in the NimbRo architecture. The `moveit_humanoid` node handles all direct control of the robot, e.g. walking. Thereby changing the robot would also mean that all these functionalities would have to be changed. Some of these algorithms are outsourced to the *body board* which makes it hard to reuse them. Furthermore, the node's name is confusing, since *MoveIt!* is a software for robotic manipulation, which is also integrated in ROS, but the node does not have anything to do with it.

The used messages are ROS standard where it is applicable, e.g. `Twist.msg` for walking commands. The new defined messages are all grouped in one package, making it easy to use them, since only one additional dependency on a package without further dependencies is necessary.

The packages of this architecture are easier to use than the NimbRo ones since they have less special dependencies or messages. The main disadvantage is the use of the *body board* which makes transferring of robot control algorithms, e.g. walking, difficult.

6.1.3 Hamburg Bit-Bots

The previous architecture of the Hamburg Bit-Bots was star-shaped due to the use of the shared memory IPC and the connector, cp. figure 3.4. Star-shaped architectures are prone to have a bottleneck. The new architecture is more homogeneous, since all data is transfered peer-to-peer.

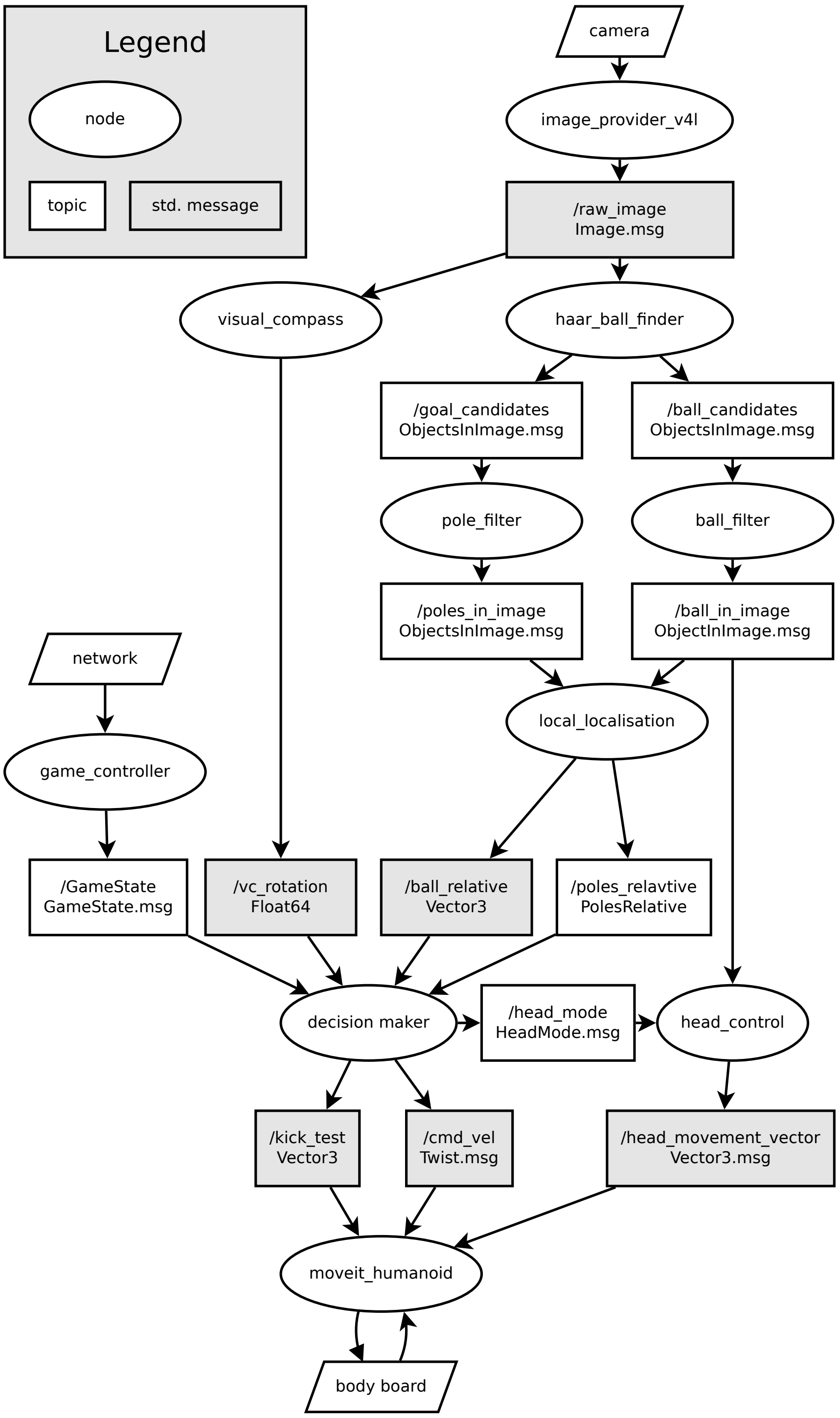


Figure 6.4: The ROS nodes and topics of the team WF Wolves based on the code, without displaying tools. The vision process is distributed over multiple nodes. The behavior consists of two nodes (`decision_maker` and `head_control`). The robots joints are controlled by an external *body board* which is accessed via the `moveit_humanoid` node.

Furthermore, the new architecture is far more parallel than the old one. This is especially important because the new used robot platform has at least eight CPU cores. In the old architecture only two processes were used, one for the motion and one for all higher level computing. This was appropriate at that time, since the used robot had only two hyper threads. The old module structure was not able to run in parallel, except for IO, because it used Python which has a global interpreter lock. This means all threads of one process can not be run in parallel, when using Python. The new architecture still uses Python, but each node is a single process. Thereby the global interpreter lock only affects threads in the scope of one node.

The old architecture had a custom made interface for displaying information about the status of the robot. While it was theoretically possible to implement different visualizations, it had some problems, see section 3.2.3, which lead to a low usage and little effort on implementing plug-ins. The new architecture can profit from the available ROS tools. Especially useful was the new possibility to plot values using `rqt_plot`.

6.2 Transfer process to ROS

After finishing the definition of the architecture and the messages, it was easy to develop the different parts of the software with multiple developers, due to the clear definition of interfaces and low dependencies between the nodes.

ROS also enables to easily test a package with mockup data, thereby not requiring a complete implementation of all packages to run basic tests. This is especially important for the behavior, since it has the most dependencies on data from other packages. In the past, testing the behavior before the competitions was normally not possible due to problems with other modules.

The needed time for transferring parts of the software to the new framework depended highly on the previous degree of modularization. The walking port, which was already modular in the old framework, took around three hours with testing. Only a small class had to be written which accepts the goal velocities, calls the computation method and publishes the results on a ROS topic. The implementation of the HCM and CM730 nodes took much longer since they were not modular and a large part of them had to be reimplemented to get clean modules.

6.3 Performance

All ROS nodes were written using rospy with the intention of making it easy to understand for users. It would have been possible to achieve a higher performance using roscpp. Still, an effort was made to ensure a usable performance of all nodes. During programming, it became clear that three major factors played a role in increasing the performance of a rospy node: First, message objects should be only generated once and then be reused. This prevents time consuming object generation.

	Standing	Animation	Walking
cm730	58%	58%	58%
joint_state_publisher	4%	4%	4%
hcm	10%	20%	20%
walking	12%	12%	40%
animation	7%	46%	7%
pause	0%	0%	0%
buttons	0%	0%	0%
Sum of all nodes	91%	140%	129%
previous motion	45%	50%	60%

Figure 6.5: CPU loads of different nodes and the previous motion process (left) in three cases of robot action (top), measured on the Minibot. The highest load is generated by the `cm730` node which has to communicate with the servos at all time. Therefore the load generated by it does not change. The loads of `animation` and `walking` are only high when they have to act, but they have a ground load which comes from their callback methods. This is about 6 to 7% per subscription on a 100Hz topic, i.e. */imu*, */joint_states* and */motor_goals*.

Second, parameters should completely be read from the parameter server at the start and saved locally. Communication with the distant parameter server is not only time consuming, the server can also become a bottle neck, since it is a centralized facility. Third, using `rospy.Time.now()` for the generation of message stamps is more expensive than `time.time()`, since it synchronizes the clock via the network. This is not necessary in this case, since all nodes are currently run on the same machine.

The CPU load is shown in figure 6.5 and compared to the previous motion process. All values for the imu, the current joint positions, and the joint goals are being published at around 100Hz. More is not possible due to the limitations of the TTL bus which was the same in the previous motion process.

Due to the split into different nodes, a new inter process communication was introduced by the publisher-subscriber relation using ROS messages. Thereby latencies are introduced into the software. These have to be as low as possible to achieve a high reactivity of the robot. The resulting latencies are shown in figure 6.6. They are higher than expected, therefore a simple test publisher and subscriber was implemented and tested on the Minibot as well as on a desktop computer (i5-2400 CPU @ 3.10GHz). For this, an empty imu message with a time stamp was transferred at 100Hz, cp. figure 6.7. It is observable that the ARM board of the Minibot produces more and higher peaks. It remains uncertain why this is the case, but it is possible that the Linux kernel version (3.10) on the Odroid is linked to this. Unfortunately, there are no newer kernels available for the board and therefore it was not possible

From	To	Message Type	Latency
cm730	hcm	Imu.msg	7.45 ms
animation	hcm	Animation.msg	17.74 ms
walking	hcm	JointTrajectory.msg	16.90 ms
hcm	cm730	JointTrajectory.msg	32.51 ms

Figure 6.6: Latencies between sending and receiving of messages. The `Imu.msg` has a lower latency since its size is smaller. The `JointTrajectory.msg` which are received by the `cm730` node have a doubled latency because they are passed through the `hcm` node. The stamp is done by the creating node, i.e. `walking` or `animation`, therefore two transmissions are counted.

to validate this. Still, achieving lower latencies would be possible by using other hardware and maybe newer kernel versions.

The highest latency is observed in the `cm730` node on the motor goal topic, because these message have passed by the `hcm` node. Thereby their latency is doubled, since they are transfered two times. It would also be possible to pass the messages directly to the `cm730` node, but this would make it more complex to ensure that only one node is writing motor goals at the same time.

When running this software on the Minibot, no difference to the previous architecture could be observed optically, e.g. while walking. This is a very subjective perception, but no other objective method for comparing the performances could be found in the scope of this thesis.

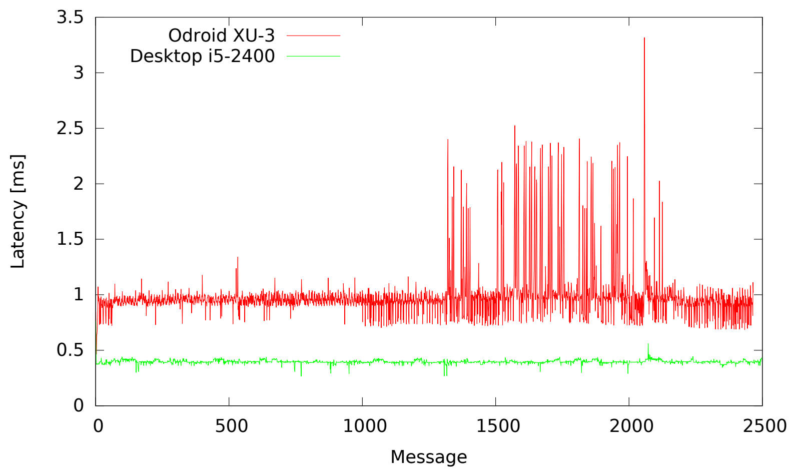


Figure 6.7: Comparison of latencies between the used Odroid XU-3 and the desktop computer. An empty but stamped `Imu.msg` was sent and received at 100Hz. The Odroid shows a general higher latency and higher spikes. This is possibly related to its outdated kernel version.

6.4 Community Building

Bringing others to use and further develop the implemented packages is crucial for having an impact on the league and thereby on the research. In order to make this happen, there has to be a motivation for others to do so. This is normally achieved by providing a gain which is higher than the necessary effort when using this architecture. The gain was already described in the sections 1.1, 6.1 and 6.3. The effort will be investigated later in this section. Furthermore, others have to know about the existence of these packages and a community has to be formed around them.

First the code has to be easily accessible for everybody. By providing it on GitHub, it is not only easy to get the code but also to contribute, since different ways, e.g. making pull request or stating issues, are available on the platform.

The packages have to be included in the existing communities which are the ROS and the RoboCup community. Pages in the ROS wiki were created, which describe the packages and provide links to the repository and further documentation. For RoboCup, it is more necessary to search a direct contact to the teams, since it is a smaller community. Some teams were directly involved in the development process. On releasing the first version, a mail to the leagues mailing list will be send, providing information and inviting other teams to use it. A paper was submitted to the RoboCup symposium and the architecture will be presented there. Furthermore, the software will be listed in the open source section of the HL website after its release.

The code has to be understandable so that others can use and further develop it. Therefore most of the parts were written in Python, as it provides a good readability, even if this costs performance. The modularization and the compliance of ROS standards helps understanding the code. Another important role is played by the documentation. Therefore the code was commented in English and an overview documentation was provided for every package in form of a *readme* file. Furthermore, general documentation can be found in the submitted paper and this thesis.

6.5 Influence on the League

The influence of the league is not clear yet, since the proposed architecture will be presented in the future. Still, some indicators look promising. First, the RoboCup Foundation supported this framework financially. This shows a general interest in it. Furthermore, two other teams were involved in the development process and they will change to this architecture until the next year. This means that for the first time an architecture will actually be shared by multiple teams in the Humanoid League.

7 Conclusion

This chapter concludes the thesis in section 7.1 and provides a perspective for further work on this topic in 7.2.

7.1 Conclusion

In this thesis, a ROS based architecture was proposed with the goal of sharing software modules between teams in the RoboCup Humanoid League. A flexible approach of defining only messages was chosen for this architecture. Furthermore, additional packages with visualization tools and utilities were provided, so that they can be used by all participating teams. Other teams were integrated in the process of defining these messages.

The resulting architecture was compared to two previously existing ROS based architectures in the league. It was shown that the proposed architecture is more flexible, easier to adapt by different teams and closer to the ROS standards. Furthermore, a comparison was made to the previous architecture of the Hamburg Bit-Bots and it was shown that the proposed architecture is easier to use and can make better use of the new robot *Minibot*.

This new robot was modeled for RViz and Gazebo and integrated into a simulated RoboCup environment. ROS nodes for the communication with the hardware and the control of the joints, i.e. walking and keyframe animations, were written for the Minibot. These nodes were successfully tested and the robot is now usable with the proposed architecture.

While the impact on the league can only be observed in the future, it was already shown that the approach is feasible by exchanging nodes between the Hamburg Bit-Bots and the WF Wolves. The usage of ROS and this architecture offers the possibility for a better transfer of software between RoboCup and general research.

7.2 Further Work

The 2017 season of RoboCup will show the first results of the proposed architecture, since the Hamburg Bit-Bots and the WF Wolves will each participate in three competitions with it. The experiences that will be made during them can then be used for further refining of messages or for adding additional ones. After the RoboCup world championship, the architecture will be presented to the league.

Further work will be mostly necessary on the visualization tools. Their features can still be improved and further packages can be added. One example would be showing the status of the team communication which is necessary because the wireless connections are very instable during competition due to a high amount of networks. Another important aspect for the visualization is to enable data transfer via UDP, for example by *ROSUDP*. Only a unidirectional UDP connection is allowed during matches to prevent human control. Therefore the implemented visualization tools are currently not allowed during competition.

The performance of the implemented robot control for the team Hamburg Bit-Bots could be further increased, especially regarding the latencies. It should also be tested on other robot platforms to ensure its general usability.

8 Appendix

AdditionalServoData.msg

```
# This message provides additional data from the servos, which is not
   included in JointState.msg
# Should mainly used for monitoring and debug purposes
# Setting the value to -1 means there is no data from this motor
float32[] voltage
sensor_msgs/Temperature[] temperature
```

Animation.msg

```
Header header
# This is a request to make HCM controllable, e.g. stop walking
bool request
# First message of this animation
bool first
# Last message of this animation
bool last
# Is this animation comming from the hardware control manager
bool hcm
# Joint goals
trajectory_msgs/JointTrajectory position
```

BallInImage.msg

```
# Providing a (possible) ball in the image
# The header is included to get the time stamp for later use in tf
std_msgs/Header header
# Center point of the ball, the z-axis should be ignored (in pixel)
geometry_msgs/Point center
# Diameter of the ball (in pixel)
float64 diameter
# A certainty rating between 0 and 1, where 1 is the surest.
# 0 means no ball was found
float32 confidence
```

BallRelative.msg

```
# Provides the relative position of the ball
# The header is included to get the time stamp
std_msgs/Header header
# x to front
# y to left
# z height (should normally be 0, if ball was not high kicked)
# Everything is measured in meters
```

```
geometry_msgs/Point ball_relative
# A certainty rating between 0 and 1, where 1 is the surest.
# 0 means no ball was found
float32 confidence
```

BallsInImage.msg

```
# Contains multiple balls on an image. Should be mainly used to provide
    ball candidates (for example round shapes)
# in the first step of the vision pipeline.
# The header is included to get the time stamp for later use in tf
std_msgs/Header header
# An empty array means no balls have been found.
humanoid_league_msgs/BallInImage[] candidates
```

BarInImage.msg

```
# A (possible) goal bar in the image. It is defined by the two end
   points and a width.
# The header is included to get the time stamp for later use in tf
std_msgs/Header header
# Two points defining the significant axis of the post
geometry_msgs/Point left_point
geometry_msgs/Point right_point
# Orthogonal to significant vector (in pixel)
float32 width
# A certainty rating between 0 and 1, where 1 is the surest.
float32 confidence
```

GameState.msg

```
# This message provides all information from the game controller
# for additional information see documentation of the game controller
# https://github.com/bhuman/GameController
std_msgs/Header header
uint8 GAMESTATE_INITAL=0
uint8 GAMESTATE_READY=1
uint8 GAMESTATE_SET=2
uint8 GAMESTATE_PLAYING=3
uint8 GAMESTATE_FINISHED=4
uint8 gameState
uint8 STATE_NORMAL = 0
uint8 STATE_PENALTYSHOOT = 1
uint8 STATE_OVERTIME = 2
uint8 STATE_TIMEOUT = 3
uint8 secondaryState
bool firstHalf
uint8 ownScore
uint8 rivalScore
# Seconds remaining for the game half
int16 secondsRemaining
# Seconds remaining for things like kickoff
```

```
uint16 secondary_seconds_remaining
bool hasKickOff
bool penalized
uint16 secondsTillUnpenalized
# Allowed to move is different from penalized.
# You can for example be not allowed to move due to the current state
    of the game
bool allowedToMove
bool dropInTeam
uint16 dropInTime
uint8 penaltyShot
uint16 singleShots
string coach_message
```

GoalInImage.msg

```
# A goal on the image. Should be extracted from the seen posts and bars
# The header is included to get the time stamp for later use in tf
std_msgs/Header header
# Left post (or the only seen one)
humanoid_league_msgs/PostInImage left_post
# Right post, or null if only one post of the goal is seen
humanoid_league_msgs/PostInImage right_post
# Vector pointing to the (probable) center of the goal.
# Should only be used if only one goal post is visible. If both are
    visible this should be none.
# This is normally an educated guess, using the goal bar or the
    position of the post on the image
# The point can also be outside of the image
geometry_msgs/Point center_direction
# A certainty rating between 0 and 1, where 1 is the surest.
# 0 means no goal was found.
float32 confidence
```

GoalPartsInImage.msg

```
# The header is included to get the time stamp for later use in tf
std_msgs/Header header
PostInImage[] posts
BarInImage[] bars
```

GoalRelative.msg

```
# Relative position to a goal
# The header is included to get the time stamp for later use in tf
std_msgs/Header header
# Position of the left goal post feet (in meter)
geometry_msgs/Point left_post
# Position of the right post, null if only one post was seen
geometry_msgs/Point right_post
# Vector pointing to the (probable) center of the goal (in meters).
# Should only be used if only one goal post is visible. If both are
    visible this should be none.
```

```
# This is normally an educated guess, using the goal bar or the
    position of the post on the image
geometry_msgs/Point center_direction
# A certainty rating between 0 and 1, where 1 is the surest.
# 0 means no goal was found
float32 confidence
```

HeadMode.msg

```
# This message is used for communicating between the body behaviour and
    the head behaviour
# The body tells the head by this message what it shall do
# Search for Ball and track it if found
uint8 BALL_MODE=0
# Search for goal posts, mainly to locate the robot on the field
uint8 POST_MODE=1
# Track ball and goal by constantly switching between both
uint8 BALL_GOAL_TRACKING=2
# Look generally for all features on the field (ball, goals, corners,
    center point)
uint8 FIELD_FEATURES=3
# Look for features outside of the field (perimeter advertising, walls,
    etc).
# Can be used for localization using features on the ceiling.
uint8 NON_FIELD_FEATURES=4
# Simply look down to its feet.
uint8 LOOK_DOWN=5
# Simply look directly forward
uint8 LOOK_FORWARD=7
#Don't move the head
uint8 DONT_MOVE=8
# Look to the ceiling, for example for visual compas
uint8 LOOK_UP=9
uint8 headMode
```

Kick.action

```
geometry_msgs/Vector3 ball_pos
geometry_msgs/Vector3 target
```

LineCircleInImage.msg

```
# Defines a line circle in image space, i.e. the center circle
std_msgs/Header header
# The circle is defined by an arc with left and right end points and a
    point in the middle for getting the radius
geometry_msgs/Point left
geometry_msgs/Point middle
geometry_msgs/Point right
```

LineCircleRelative.msg

```
# Defines a line circle in relative space, i.e. the center circle
```

```
std_msgs/Header header
# The circle is defined by an arc with left and right end points and a
    point in the middle for getting the radius
geometry_msgs/Point left
geometry_msgs/Point middle
geometry_msgs/Point right
```

LineInformationInImage.msg

```
# Contains all line related information on the image itself
# The header is included to get the time stamp for later use in tf
std_msgs/Header header
LineIntersectionInImage[] intersections
LineSegmentInImage[] segments
LineCircleInImage[] circles
```

LineInformationRelative.msg

```
# Contains all relative information about line features on the field
# The header is included to get the time stamp for later use in tf
std_msgs/Header header
LineSegmentRelative[] segments
LineIntersectionRelative[] markings
LineCircleRelative[] circles
```

LineIntersectionInImage.msg

```
# A line intersection feature in the image
std_msgs/Header header
# The type defines which kind of intersection is present
uint8 UNDEFINED=0
uint8 L=1
uint8 T=2
uint8 X=3
uint8 type
# The line segments related to this crossing
humanoid_league_msgs/LineSegmentInImage segments
# A certainty rating between 0 and 1, where 1 is the surest.
float32 confidence
```

LineIntersectionRelative.msg

```
# Information about a special line feature on the field
# The header is included to get the time stamp for later use in tf
std_msgs/Header header
# The type defines which kind of intersection is present
uint8 UNDEFINED=0
uint8 L=1
uint8 T=2
uint8 X=3
uint8 type
# The line segments related to this crossing
```

```
humanoid_league_msgs/LineSegmentRelative segments
# A certainty rating between 0 and 1, where 1 is the surest.
float32 confidence
```

LineSegmentInImage.msg

```
# A normal line segment in the image
# The header is included to get the time stamp for later use in tf
std_msgs/Header header
# Two points defining the vector of the line. The center is
    orthogonally in the middle of the line
geometry_msgs/Point start
geometry_msgs/Point end
# Orthogonal to the significant vector
float32 start_width
float32 end_with
# A certainty rating between 0 and 1, where 1 is the surest.
float32 confidence
```

LineSegmentRelative.msg

```
# A line segment relative to the robot
std_msgs/Header header
# Start and end position of the line
# x in front of the robot
# y to the left
# z should be 0
geometry_msgs/Point start
geometry_msgs/Point end
# A certainty rating between 0 and 1, where 1 is the surest.
float32 confidence
```

Model.msg

```
# The model message contains all information from the object
    recognition after filtering
BallRelative ball
ObstaclesRelative obstacles
geometry_msgs/PoseWithCovarianceStamped position
```

ObstacleInImage.msg

```
# An obstacle in the image, which can be a robot, a human or something
    else
# The header is included to get the time stamp for later use in tf
std_msgs/Header header
# Main color of the obstacle, to differentiate between robots and other
     things like human legs
# Something we cant classify
uint8 UNDEFINED = 0
# Robot without known color
uint8 ROBOT_UNDEFINED = 1
```

```
uint8 ROBOT_MAGENTA = 2
uint8 ROBOT_CYAN = 3
# A human legs , e.g. from the referee
uint8 HUMAN = 4
# Black poles which are normally used for technical challenges
uint8 POLE = 5
uint8 color
# The number of the robot , if it is a robot and if it can be read. Put
    in −1 if not known
uint8 playerNumber
# The corresponding section in the image
geometry_msgs/Point top_left
uint8 height
uint8 width
# A certainty rating between 0 and 1, where 1 is the surest.
float32 confidence
```

ObstacleRelative.msg

```
# An obstacle relative to the robot
# The header is included to get the time stamp for later use in tf
std_msgs/Header header
# Main color of the obstacle , to differentiate between robots and other
     things like human legs
# Something we cant classify
uint8 UNDEFINED = 0
# robot without known color
uint8 ROBOT_UNDEFINED = 1
uint8 ROBOT_MAGENTA = 2
uint8 ROBOT_CYAN = 3
# A human legs , e.g. from the referee
uint8 HUMAN = 4
# Black poles which are normally used for technical challenges
uint8 POLE = 5
uint8 color
# The number of the robot , if it is a robot and if it can be read. Put
    in −1 if not known
uint8 playerNumber
# Position (in meters)
geometry_msgs/Point position
# Educated guess of the width (in meters)
float32 width
# A certainty rating between 0 and 1, where 1 is the surest.
float32 confidence
```

ObstaclesInImage.msg

```
# The header is included to get the time stamp for later use in tf
std_msgs/Header header
ObstacleInImage[] obstacles
```

ObstaclesRelative.msg

```
# The header is included to get the time stamp for later use in tf
std_msgs/Header header
ObstacleRelative[] obstacles
```

PlayAnimation.action

```
# goal definition
# name of the animation
string animation
# if the animation commes from the hardware control manager, it should
    be played directly, interrupting other animations and walking
# it's propably falling or standing up
bool hcm
---
#result definition
bool successful
---
#feedback
uint8 percent_done
```

Position2D.msg

```
# The position system is the same as mitecom. The following part is
    taken from the mitecom documentation:
# https://github.com/fumanoids/mitecom
# The origin of the absolute coordinate center is the center of the
    middle
# circle (center of field). The x axis points towards the opponent goal
    , the
# y axis to the left.
#
#      y
#      ^       ______________________
#      |    M  |          |          |  O
#      |    Y  |_ -x, y   |   x, y  _|  P
#      |    G  | |        |         | |  P
# 0    +    O  | |       ( )        | |  G
#      |    A  |_|        |         |_|  O
#      |    L  |  -x,-y   |   x,-y   |  A
#      |       |__________|__________|  L
#      |
#      +------------------+--------------> x
#                         0
#
# The 0 value of the orientation is pointing to the opponent side (
    right side in the image).
# The value increases counter clockwise
# Everything in meters (because it is the ROS standard)
# The header is included to get the time stamp for later use in tf
std_msgs/Header header
geometry_msgs/Pose2D pose
```

```
# A certainty rating between 0 and 1, where 1 is the surest.
float32 confidence
```

PostInImage.msg

```
# A (possible) goal post in the image. It is defined by two end points
    and a width.
# The header is included to get the time stamp for later use in tf
std_msgs/Header header
# Two points defining the significant axis of the post
geometry_msgs/Point foot_point
geometry_msgs/Point top_point
# Orthogonal to significant vector (in pixel)
float32 width
# A certainty rating between 0 and 1, where 1 is the surest.
float32 confidence
```

RobotControlState.msg

```
# This message provides the current state of the hardware control
    manager (HCM), which is handling falling, standing up and the
    decision
# between playing animations and walking
# Robot can be controlled from a higher level
uint8 CONTROLABLE=0
# Robot is currently falling
# it can not be controlled and should go to a position that minimizes
    the damage during a fall
uint8 FALLING=1
# Robot is lying on the floor
# maybe reset your world model, as the state should be unsure now
uint8 FALLEN=2
# Robot is currently trying to get up again
uint8 GETTING_UP=3
# An animation is running
# no walking or further animations possible
# Falling detection is deactivated
uint8 ANIMATION_RUNNING=4
# The hardware control manager is booting
uint8 STARTUP=5
# The hardware control manager is shutting down
uint8 SHUTDOWN=6
# The robot is in penalty position
# It can not be controlled
uint8 PENALTY=7
# The robot is getting in or out of penalty position
uint8 PENALTY_ANIMANTION=8
# The robot is used for recording animations
# Reserved all controling to a recording process
# No falling detection is processed and no stand ups will be done
uint8 RECORD=9
# The robot is walking
```

```
uint8 WALKING=10
# A state where the motors are turned off, but the hardware control
    manager is still waiting for commandos and turns the motors on,
# if a move commando comes
uint8 MOTOR_OFF=11
# Last status send by the hardware control manager after shutting down
uint8 HCM_OFF=12
uint8 state
```

Speak.msg

```
# This message is used to activate the audio output of the robot
# This can be used for debug proposed but also for natural language
    team communication
# The text will only be outputed if "filename" is empty
string text
uint8 LOW_PRIORITY=0
uint8 MID_PRIORITY=1
uint8 HIGH_PRIORITY=2
uint8 priority
# If a file should be read, the path has to be specified here.
    Otherwise this string should be null
string filename
```

Strategy.msg

```
# This message provides information about the current strategy of the
    robot to the team communication so that it can be
# shared with other team robots
# Which role the robot has currently
uint8 ROLE_IDLING=0
uint8 ROLE_OTHER=1
uint8 ROLE_STRIKER=2
uint8 ROLE_SUPPORTER=3
uint8 ROLE_DEFENDER=4
uint8 ROLE_GOALIE=5
uint8 role
# The current action of the robot
uint8 ACTION_UNDEFINED=0
uint8 ACTION_POSITIONING=1
uint8 ACTION_GOING_TO_BALL=2
uint8 ACTION_TRYING_TO_SCORE=3
uint8 ACTION_WAITING=4
uint8 action
# Offensive strategy
uint8 SIDE_LEFT = 0
uint8 SIDE_MIDDLE = 1
uint8 SIDE_RIGHT = 2
uint8 offensive_side
```

TeamData.msg

```
# This message contains all information provided by the mitecom
   standard for team communication.
# Everything is in meters (ROS standard) not to be confused with
   millimeters (mitecom standard)!
# Set belief values to 0 if object was not recognized.
# More information here: https://github.com/fumanoids/mitecom
std_msgs/Header header
# Every value is an array because we can have multiple robots
   communicating with us.
# The values match with the robot ids
uint8[] robot_ids
uint8 ROLE_IDLING=0
uint8 ROLE_OTHER=1
uint8 ROLE_STRIKER=2
uint8 ROLE_SUPPORTER=3
uint8 ROLE_DEFENDER=4
uint8 ROLE_GOALIE=5
uint8[] role
uint8 ACTION_UNDEFINED=0
uint8 ACTION_POSITIONING=1
uint8 ACTION_GOING_TO_BALL=2
uint8 ACTION_TRYING_TO_SCORE=3
uint8 ACTION_WAITING=4
uint8[] action
uint8 STATE_INACTIVE=0
uint8 STATE_ACTIVE=1
uint8 STATE_PENALIZED=2
uint8[] state
# Absolute position values
geometry_msgs/Pose2D[] robot_positions
# Relative ball position, theta of Pose2D is not used
Position2D[] ball_relative
# Relative position of the opponent goal, theta of Pose2D is not used
# This is helpful if the robot has no global position, but sees the
   goal
Position2D[] oppgoal_relative
# Positions of opponent robots, if they are recognized
# The letter of the robot is arbitrary as the sending robot does not
   know the id of a seen robot
Position2D[] opponent_robot_a
Position2D[] opponent_robot_b
Position2D[] opponent_robot_c
Position2D[] opponent_robot_d
# Positions of team robots, if they are recognized
# The letter of the robot is arbitrary as the sending robot does not
   know the id of a seen robot
Position2D[] team_robot_a
Position2D[] team_robot_b
Position2D[] team_robot_c
float32[] avg_walking_speed
```

```
float32[] time_to_position_at_ball
float32[] max_kicking_distance
# Strategy over which side the team tries to attack
# Especially useful during a kickoff
uint8 UNSPECIFIED=0
uint8 LEFT=1
uint8 RIGHT=2
uint8 CENTER=3
uint8[] offensive_side
```

VisualCompassRotation.msg

```
# This message is used to specify the orientation of the visual compass
    in relation to a RoboCup Soccer field
# 0 points to the opponent goal line, 3.14 to the own goal line
float32 orientation
# A certainty rating between 0 and 1, where 1 is the surest.
float32 confidence
```

Bibliography

[Allgeuer et al., 2016] Allgeuer, P., Farazi, H., Ficht, G., Schreiber, M., and Behnke, S. (2016). The igus Humanoid Open Platform. *KI-Künstliche Intelligenz*, pages 1–5.

[Allgeuer et al., 2013] Allgeuer, P., Schwarz, M., Pastrana, J., Schueller, S., Missura, M., and Behnke, S. (2013). A ROS-based software framework for the NimbRo-OP humanoid open platform. In *Proceedings of 8th Workshop on Humanoid Soccer Robots, IEEE-RAS Int. Conference on Humanoid Robots, Atlanta, USA*.

[Anders et al.,] Anders, B., Stiddien, F., Krebs, O., Gerndt, R., Bolze, T., Lorenz, T., Chen, X., Londero, F. T., et al. WF Wolves & Taura Bots–Humanoid Teen Size Team Description for RoboCup 2016.

[Bruyninckx, 2001] Bruyninckx, H. (2001). Open robot control software: the OROCOS project. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 3, pages 2523–2528. IEEE.

[De Boor et al., 1978] De Boor, C., De Boor, C., Mathématicien, E.-U., De Boor, C., and De Boor, C. (1978). *A practical guide to splines*, volume 27. Springer-Verlag New York.

[Fabre et al., 2016] Fabre, R., Rouxel, Q., Passault, G., N'Guyen, S., and Ly, O. (2016). Dynaban, an Open-Source Alternative Firmware for Dynamixel Servo-Motors. In *Symposium RoboCup*.

[Forero et al., 2013] Forero, L. L., Yánez, J. M., and Ruiz-del Solar, J. (2013). Integration of the ROS framework in soccer robotics: the NAO case. In *Robot Soccer World Cup*, pages 664–671. Springer.

[Gerkey, 2015] Gerkey, B. (2015). ROS, the Robot Operating System, Is Growing Faster Than Ever, Celebrates 8 Years. *Retrieved from IEEE Spectrum: http://spectrum.ieee.org/automaton/robotics/roboticssoftware/ros-robot-operating-system-celebrates-8-years*.

[Gerkey et al., 2003] Gerkey, B., Vaughan, R. T., and Howard, A. (2003). The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th international conference on advanced robotics*, volume 1, pages 317–323.

[Gerndt et al., 2015] Gerndt, R., Seifert, D., Baltes, J. H., Sadeghnejad, S., and Behnke, S. (2015). Humanoid robots in soccer: Robots versus humans in RoboCup 2050. *IEEE Robotics & Automation Magazine*, 22(3):147–154.

[Gouaillier et al., 2008] Gouaillier, D., Hugel, V., Blazevic, P., Kilner, C., Monceaux, J., Lafourcade, P., Marnier, B., Serre, J., and Maisonnier, B. (2008). The nao humanoid: a combination of performance and affordability. *CoRR abs/0807.3223*.

[Ha et al., 2011] Ha, I., Tamura, Y., Asama, H., Han, J., and Hong, D. W. (2011). Development of open humanoid platform DARwIn-OP. In *SICE Annual Conference (SICE), 2011 Proceedings of*, pages 2178–2181. IEEE.

[Kajita et al., 2014] Kajita, S., Hirukawa, H., Harada, K., and Yokoi, K. (2014). *Introduction to humanoid robotics*, volume 101. Springer.

[Khandelwal and Stone, 2011] Khandelwal, P. and Stone, P. (2011). A low cost ground truth detection system for RoboCup using the Kinect. In *Robot Soccer World Cup*, pages 515–527. Springer.

[Kohlbrecher et al., 2014] Kohlbrecher, S., Kunz, F., Koert, D., Rose, C., Manns, P., Daun, K., Schubert, J., Stumpf, A., and von Stryk, O. (2014). Towards Highly Reliable Autonomy for Urban Search and Rescue Robots. In *Robot Soccer World Cup*, pages 118–129. Springer.

[Kohlbrecher et al., 2013] Kohlbrecher, S., Meyer, J., Graber, T., Petersen, K., Klingauf, U., and von Stryk, O. (2013). Hector open source modules for autonomous mapping and navigation with rescue robots. In *Robot Soccer World Cup*, pages 624–631. Springer.

[Kohlbrecher et al., 2012] Kohlbrecher, S., Petersen, K., Steinbauer, G., Maurer, J., Lepej, P., Uran, S., Ventura, R., Dornhege, C., Hertle, A., Sheh, R., et al. (2012). Community-driven development of standard software modules for search and rescue robots. In *SSRR*, pages 1–2.

[Kootbally et al., 2013] Kootbally, Z., Balakirsky, S., and Visser, A. (2013). Enabling codesharing in rescue simulation with usarsim/ros. In *Robot Soccer World Cup*, pages 592–599. Springer.

[Mamantov et al., 2014] Mamantov, E., Silver, W., Dawson, W., and Chown, E. (2014). Robograms: A lightweight message passing architecture for robocup soccer. In *Robot Soccer World Cup*, pages 306–317. Springer.

[McGill et al., 2013] McGill, S. G., Yi, S.-J., Zhang, Y., and Lee, D. D. (2013). Extensions of a robocup soccer software framework. In *Robot Soccer World Cup*, pages 608–615. Springer.

[Metta et al., 2006] Metta, G., Fitzpatrick, P., and Natale, L. (2006). Yarp: Yet another robot platform. *International Journal of Advanced Robotic Systems*, 3(1):8.

[Murphy, 2000] Murphy, R. (2000). *Introduction to AI robotics*. MIT press.

[Nii, 1986] Nii, H. P. (1986). Blackboard application systems, blackboard systems and a knowledge engineering perspective. *AI magazine*, 7(3):82.

[Perico et al., 2014] Perico, D. H., Silva, I. J., Vilão, C. O., Homem, T. P., Destro, R. C., Tonidandel, F., and Bianchi, R. A. (2014). Hardware and software aspects of the design and assembly of a new humanoid robot for robocup soccer. In *Robotics: SBR-LARS Robotics Symposium and Robocontrol (SBR LARS Robocontrol), 2014 Joint Conference on*, pages 73–78. IEEE.

[Quigley et al., 2009] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan.

[Ramsden, 2011] Ramsden, E. (2011). *Hall-effect sensors: theory and application*. Newnes.

[Röfer and Laue, 2013] Röfer, T. and Laue, T. (2013). On B-human's code releases in the standard platform league–software architecture and impact. In *Robot Soccer World Cup*, pages 648–655. Springer.

[Ruiz et al., 2013] Ruiz, J. A. Á., Plöger, P., and Kraetzschmar, G. K. (2013). Active scene text recognition for a domestic service robot. In *RoboCup 2012: Robot Soccer World Cup XVI*, pages 249–260. Springer.

[Sanner et al., 1999] Sanner, M. F. et al. (1999). Python: a programming language for software integration and development. *J Mol Graph Model*, 17(1):57–61.

[Schwarz et al., 2013] Schwarz, M., Pastrana, J., Allgeuer, P., Schreiber, M., Schueller, S., Missura, M., and Behnke, S. (2013). Humanoid teensize open platform nimbro-op. In *Robot Soccer World Cup*, pages 568–575. Springer.

[Stroud et al.,] Stroud, A., Carey, K., Chinang, R., Gibson, N., Panka, J., Ali, W., Brucato, M., Procak, C., and Morris, M. Team MU-L8 Humanoid League–TeenSize Team Description Paper 2014.

[Stroud et al., 2013] Stroud, A. B., Morris, M., Carey, K., Williams, J. C., Randolph, C., and Williams, A. B. (2013). MU-L8: The Design Architecture and 3D Printing of a Teen-Sized Humanoid Soccer Robot. In *8th Workshop on Humanoid Soccer Robots, IEEE-RAS International Conference on Humanoid Robots, Atlanta, GA*.

[Upton and Halfacree, 2014] Upton, E. and Halfacree, G. (2014). *Raspberry Pi user guide*. John Wiley & Sons.

[Vukobratović and Borovac, 2004] Vukobratović, M. and Borovac, B. (2004). Zero-moment point—thirty five years of its life. *International Journal of Humanoid Robotics*, 1(01):157–173.

[Wescott, 2000] Wescott, T. (2000). Pid without a phd. *Embedded Systems Programming*, 13(11):1–7.

Internet Sources

[1] https://github.com/Pold87/academic-keyword-occurrence.

[2] http://www.b92.net/zivot/nauka.php?yyyy=2012&mm=09&dd=07&nav_id=641200.

[3] wiki.ros.org.

[4] http://www.pirobot.org/blog/0025/.

[5] http://en.robotis.com.

[6] http://support.robotis.com/en/techsupport_eng.htm.

[7] http://en.robotis.com/index/product.php?cate_code=101010.

[8] https://github.com/fumanoids/mitecom.

[9] https://www.robocuphumanoid.org/materials/rules/.

[10] http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127.

[11] https://www.robocuphumanoid.org/wp-content/uploads/\HumanoidLeagueProposedRoadmap.pdf.

[12] https://github.com/NimbRo/.

[13] http://www.eng.mu.edu/abwilliams/heirlab/index.html.

[14] https://github.com/ROBOTIS-OP.

[15] https://github.com/igusGmbH/HumanoidOpenPlatform.

[16] https://github.com/AIS-Bonn/humanoid_op_ros.

[17] http://www.robocup2016.org/de/symposium/team-description-papers/.

[18] http://www.robocup.org/about-robocup/a-brief-history-of-robocup.

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Hamburg, den 06.04.2017

Marc Bestmann

Veröffentlichung

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Hamburg, den 06.04.2017

Marc Bestmann