Institut für Informatik
Lehr- und Forschungseinheit für
Programmierung und Softwaretechnik

Master Thesis
in Software Engineering

An evaluation of LEGO MINDSTORMS for collaborative swarm robotics with regard to the possibility of applying smart agents

Annabelle Teresa Klarl

Aufgabensteller:	Prof. Dr. Martin Wirsing
Betreuer:	Dr. Matthias Hölzl
Abgabedatum:	25.11.2010

I hereby declare that this Master Thesis is my own work and effort and that it has not been submitted anywhere for any award. Where other sources of information have been used, they have been acknowledged.

25.11.2010 _____

Annabelle Teresa Klarl

Abstract

Swarm robotics tries to apply swarm intelligence known from social insects such as ants, termites and bees to robots: while each robot is relatively simple, intelligent behaviour emerges from local interactions among robots without any kind of centralized control. As these simple individuals are not aware of their environment, their goals or even themselves and cannot reason or plan their actions the tasks they can perform are rather limited. To extend the application areas, current research tries to extend swarm robotics with intelligent agents. Intelligent agents should extend simple reflexive robots to smart robots that are able to think and communicate in order to reach solutions depending on knowledge gained over a longer period of time. In this master thesis, we have explored the suitability of LEGO MINDSTORMS for swarm robotics in general and later on for the incorporation of learning, reasoning and planning. We have evaluated the overall performance of LEGO MINDSTORMS in precision of motors and sensors, durability of power supply, computational power and capabilities for communication and physical interaction. Based on this assessment we have defined and implemented some preliminary work for collective transportation such as single and distributed line-following and heading following. For collective transportation a swarm of robots should push an object too heavy or big for a single one according to a specific course, e.g. a black line on the floor. Since collective transportation requires further exploration of areas like distributed intelligence, we have focused on the sub-tasks mentioned above. A single line-follower thereby recognizes a black line a white surface and follows this line. Distributed line-following extends this principle to pushing an object by several robots along this line. In the last application of heading following, some robots try to align their heading according to a heading given by a line-follower. Overall, we appreciate LEGO MINDSTORMS as an inexpensive platform powerful and extendible enough for prototyping new approaches in swarm robotics, even with the possibility for the inclusion of intelligent agents.

Contents

Chapter 1

Introduction

Developing software for physical systems such as robots is a challenging task. Theoretical considerations and simulations verify and test the correctness, suitability and applicability of new ideas and algorithms. However, if applied to a real-life platform many unexpected problems arise that have not been modelled or thought of. Robots, for example, may exhibit an undesired behaviour due to sensor or motor inaccuracies or environment conditions so various that they could not completely be simulated. Of course, simulation makes use of Physics engines and introduces possible noise, interferences and failures, but often real life holds even more difficulties. It therefore is highly advisable to take new algorithms into the real world after their theoretical verification and simulation. Unfortunately, high-resolution custom-built robots are often very expensive and not affordable. That is why many research groups are changing to inexpensive, less powerful off-the-shelf platforms. This master thesis aims to explore the suitability of such a platform: Lego Mindstorms.

1.1 Motivation

Social insects such as ants, termites and bees are fascinating examples for swarm intelligence. Where each individual is relatively simple and limited in its abilities a swarm of insects can collectively achieve intelligent behaviour and carry out tasks that are way beyond the capabilities of its individual members. Swarm robotics tries to apply these principles to robots: each robot in a swarm is relatively incapable of intelligent actions but intelligent behaviour emerges from local interactions among robots without any centralized control. Robot swarms are therefore very robust against loss or malfunction of individual robots and can easily scale to a wide range of sizes. However, the restriction to rather simple individuals leads to some limitations: the robots have limited awareness of themselves, their environment and their goals; they cannot reason or plan to achieve a

distributed solution to the task they are given. An elaborate model for the storage of knowledge is not included. These limitations should be overcome by the aid of intelligent agents. The simple robots should be extended by this advanced individual intelligence to achieve more than simple reflexive robots.

With this thesis, we want to prepare the first step towards swarm robotics with intelligent agents by considering an inexpensive platform for prototyping robots that can exhibit varied and complex individual behaviours while being aware of the influence that their actions have on themselves, the swarm and the environment. This enables autonomous adaptation of individual agents in addition to collective evolution of the whole swarm. We hope that eventually these kinds of swarms lead to systems that are more robust against changes in their environment and easier to adapt to new tasks.

This thesis examines the use of LEGO MINDSTORMS for swarm robotics and the further extension to intelligent agents. By building a swarm of LEGO MINDSTORMS which cooperate to solve the real-life problem of collective transportation, we explore the suitability of this platform for building robot swarms and the compromises that have to be made by using inexpensive off-the-shelf components instead of custom-built hardware. To this end, we evaluate features of an individual LEGO MINDSTORMS robot, such as the precision of the motors and sensors, the durability of the power supply or their computational power, as well as facilities for communication and physical interaction between LEGO MINDSTORMS. Based on this assessment, we have defined and implemented some preliminary work for collective transportation along a line such as single and distributed line-following and heading following. For collective transportation, a swarm of robots should push an object too heavy or big for a single one according to a specific course depicted by a black line on the floor. As for this task especially distributed intelligence has to be explored further, we have focused on the case studies mentioned above. A single line-follower thereby recognizes a black line a white surface and follows this line. Distributed line-following extends this principle to pushing an object by several robots along this line. In the last application of heading following, some robots try to align their heading according to a heading given by a line-follower.

1.2 Outline

The thesis is structured into three main parts. The first gives an overview of the current state of the art and related work in mobile and swarm robotics as well as the field of intelligent agents. Therefore we discuss categories of robots and the main developments for sensing and actuating in the environment. The key features for swarms of robots follow; the current state of the art of intelligent agents and self-awareness ends this part. We also give an outline of the current research

in each of the three fields. The second part describes LEGO MINDSTORMS in full detail. The performance, power and accuracy of the motors, sensors and the main computing unit is discussed and a detailed description of the capabilities of communication and physical interaction is appended. The last chapter focuses on the realization of robots swarms using LEGO MINDSTORMS. We propose an architecture for implementing the controller for a single robot as well as a communicating swarm of robots and conclude this chapter with three exemplary implemented applications.

Chapter 2

Background and Related Work

This first chapter shows the main achievements in mobile and swarm robotics as well as in the field of intelligent agents. We want to explain the key terms in each field and give an overview of the current research.

2.1 Mobile robotics

In accordance with Russell [RN03], robots are physical agents that execute tasks in the physical world. On the one hand, they have sensors to perceive their environment and their own physical effects on the environment. On the other hand, they have actuators to achieve physical effects in order to fulfil their intended purpose.

Robots can be classified by their manipulation and mobility capabilities [RN03]. The first class of robots can only act upon their immediate environment. They only consist of a *manipulator*, for example a robotic arm, that is physically fixed to its working place and cannot be moved away. In contrast, the second class of robots does not have a manipulator, but are rather *mobile*. They can move in their environment, but cannot actually pick something from their environment. Their movement can be restricted to a specific path like with rail tracks or they can move freely. The third class of robots combines these capabilities. The *hybrid* robots have manipulators that are mounted on a mobile chassis. Therefore, the range of the manipulator is extended by the capability of moving the robot's chassis, but the use of the manipulator is complicated as it is no longer fixed to a specific position. In the following, we focus on mobile robots only as we want to solve the application of collective transportation without manipulators.

The first mobile robots have been invented by W. Grey Walter in 1948 [Wal50]. The famous tortoises Elsie and Elmer have been able to move around freely while avoiding obstacles. They could even determine if they were running out

of battery and move to the charging station that was marked by bright light. In 1964, John Hopkins developed the *Beast* with a more complex behaviour. It was the first self-feeding robot which could search for its charging station and plug in. It could either search visually for a black electrical outlet on a white wall or tactually for the slots of an electrical outlet. This behaviour of self-feeding is one of the most desirable in order to provide autonomous robots that can "survive" without human intervention. The first mobile robot with capabilities for artificial intelligence like planning was *Shakey* developed at the Stanford Reasearch Institute in 1966 [Nil84]. With the incorporation of artificial intelligence, truly autonomous robots could be built. In 1994, Ernst Dickmann proposed the first autonomous car called *VaMP* [DB+94]. The longest test drive was approximately 2000 km long (from Munich to Copenhagen and back) while the car drove almost without any human intervention in the normal traffic on a German Autobahn with a speed of at most 180 km/h. The Defense Advanced Research Projects Agency DARPA held 2004, 2005 and 2007 three competitions, the *DARPA Urban Challenge*, where autonomous vehicles have to drive in traffic and perform complex manoeuvres[1]. More recently in October 2010, VisLab could set a new milestone for autonomous mobile robotics in their project *The VisLab Intercontinental Autonomous Challenge* by driving unmanned vehicles from Italy to China without any human intervention[2]. Until today, various mobile robots have been developed for differing application domains from domestic assistants over industry to space exploration[3].

In the next sections, we give a short overview of standard locomotion and perception mechanisms for mobile robots. Afterwards, we want to explain how perception influences navigation. Some classical architectures for implementing mobile robots are demonstrated before we conclude this section about mobile robots with an overview of current research topics.

2.1.1 Locomotion

Locomotion is the crucial feature in mobile robotics. In contrast to manipulators, mobile robots should be able to move unbounded throughout their environment so that their workspace is no longer fixed to the immediate surroundings. To specify the possibility to control their position, heading and inclination the concept of *degrees of freedom DOF* is used. According to Russell [RN03], a DOF describes each independent direction to which a robot can move. Normally six DOFs can be achieved at most: moving forwards or backwards, left or right and up or down as well as turning left or right, tilting forwards or backwards and side to side.

[1] see web page `http://www.darpa.mil/grandchallenge/index.asp`
[2] see web page `http://viac.vislab.it`
[3] see web page `http://www.nasa.gov/mission_pages/mars-pathfinder`

While the concept of DOFs describes each possible position in the robot's workspace regardless how the robot achieved this position, we also need a concept for characterizing the ability of the robot to actually achieve its position. *Controllable degrees of freedom CDOFs* declare in which direction the robot is able to move directly. For example, the workspace of a normal driving car has three DOFs: it is able to move to any position on the ground while its heading can be aligned to any orientation. Nevertheless, the robot achieves this pose by moving forward or backward and steering sideways what refers to only two CDOFs. Mobile robots with a smaller number of CDOFs than DOFs require more time and energy to achieve their desired pose than robots with equivalent CDOFs and DOFs. If both characteristic numbers are equal, the robot is called *holonomic* while all other robots are *nonholonomic*. The naming and exact definition of CDOFs is not consistent in the literature, but in [SN04] the described characteristics are discussed in more detail.

For locomotion, a power source is needed. Most mobile robots are equipped with an electromotor to act upon their environment. Other mechanisms like pneumatics or hydraulics are also applied. Both, pneumatics and hydraulics are gaining power of pressure. While pneumatics uses compressed gas, pressurized liquids are the key components of hydraulics [RN03]. Regardless of the power source, these mechanisms supply the robot with enough power to move its actuators.

The actuators can create motion in various ways. In principle, all kinds of locomotion like walking, jumping, sliding, swimming, flying or rolling are possible. Unfortunately, the natural locomotion mechanisms are very difficult to replicate because nature developed several high sophisticated and specialized means for locomotion like legs. They need excellent coordination and a fast response time. Unfortunately, that far exceeds artificial replications and their control units. The use of fewer legs like the replication of the human legs reduces the need for coordination, but holds the challenge of keeping balance. Mobile robotics applies only a small number of legs as it can be seen in the next section. Additionally, wheeled locomotion is preferably applied. This is a well-understood and well-known technology so that such robots can easily be controlled. The next sections contrast legged, wheeled and tracked locomotion where tracked locomotion is a special form of wheeled locomotion. Other means of locomotion are shortly summarized in the last section.

2.1.1.1 Legs

In nature, legged locomotion can be observed in various ways. The most common leg configuration are four legs for mammals or six legs for insects while two or eight legs are more rare. The biggest advantage of this kind of locomotion is

that the motion is also possible in rough terrain. Legs only need a small patch of ground contact and are moved by lifting. Therefore, they only need small points of adequate ground clearance to place the legs on and can surmount wide ranges of uneven terrain or holes between those points. They also can be used to manipulate the environment similar to robotic arms.

For high adaptability and manoeuvrability, a high degree of freedom per leg is needed to impart forces in a number of directions. For simple motion, at least two DOFs are required because the leg must be lifted, swung to the front and then lowered to the ground. Three DOFs are more common in order to achieve more complex manoeuvres. Principally, each number of legs is possible for a legged robot, but some configurations exhibit useful properties. For example, three legs guarantee static stability. Such a robot is able to balance in place without any correction if the three legs can be arranged to hold the centre of gravity in the middle. For static walking, however, six legs are needed. Static walking means that the robot is able to walk without any balancing correction while some legs are lifted. With six legs, only three legs have to be lifted in order to move while the other three legs can hold the static balance.

Unfortunately, legged locomotion is rather complex. The construction of legs is mechanically difficult and while the control of one leg is already demanding, the coordination of several legs is rather challenging. In addition, legged locomotion needs a lot of energy to lift and swing the legs as well as to hold the balance and sustain the weight of the robot's body. An introduction to legged locomotion can be found in [SN04].

2.1.1.2 Wheels

A simpler locomotion mechanism is wheeled locomotion. Deriving from human invention and not from nature, this kind of locomotion is adapted to motion on flat ground. Wheels are designed to be continuously in ground contact so that they highly depend on the quality of the ground. On soft, loose or uneven ground wheels are inefficient due to rolling friction or skidding. However, on a smooth and flat surface as it can normally be found in laboratory environments wheels are faster than legs since they do not have to be lifted.

Another advantage of wheeled robots is that they do not have to actively balance. Normally, wheeled robots are constructed so that all wheels are in ground contact and support the robot's body all the time. With at least two wheels, static stability can be achieved if the centre of mass is below the wheel axle. However, in the majority of cases more wheels are used. Typical configurations are a trike with two controllable wheels and a third non-controllable ground contact or a four wheeled car (a detailed overview of wheel configurations can be found in [SN04]).

With the use of more than three wheels, the problem of maintaining ground contact in uneven terrains arises. To prevent a robot from tipping, the robot needs some kind of suspension system. In addition to this stability issue, manoeuvrability and controllability must be considered. On the one hand, the robots should be built to reach each desired location easily, but on the other hand we want to be able to exactly control this movement without any deviation or inaccuracy. Beside the wheel configuration, also the wheel design matters for that. In wheeled locomotion, four types of wheels are primarily used: the *standard wheel* is known from the front wheel of a bicycle. It rotates around its wheel axle and the ground contact point by steering. The *castor wheel* resembles a wheel for a shopping cart where the wheel rotates around an offset axis, but is apart from that similar to the *standard wheel*. These two wheels are constrained to a certain direction of rotation. To overcome this limit, the *Swedish wheel* and the *spherical wheel* have been developed. The *Swedish wheel* adds an additional degree of freedom to the *standard wheel* by small rollers around the circumference of the wheel. The *spherical wheel* is the only truly omnidirectional wheel (for a detailed discussion refer to [SN04]).

2.1.1.3 Walking wheels

To combine the adaptability of legs to uneven terrain with the efficiency and stability of wheels, hybrid mobile robots conquer the field and space robotics. These hybrids make use of articulations to overcome obstacles or holes and in the meantime hold the contact to nearly all wheels for guaranteeing static stability. Therefore, these robots can be applied in different outdoor environments. The most famous hybrid robots are the exploration robot *Sojourner* of the NASA Mars Pathfinder Project[4] or the climbing *Shrimp* produced by EPFL [ECM+00].

2.1.1.4 Tracks

An alternative to wheels are tracks which reach a higher efficiency on rough or loose terrain. Tracked robots have several wheels that are connected by tracks striking the floor. With these tracks, the robots have larger ground contact patches which results in better traction on loose terrain compared to wheels. Tracked vehicles come therefore often into action if experiments are done under natural conditions and not in a laboratory. However, tracked locomotion is rather inaccurate. To drive a curve, the tracks have to skid on the ground so that it is hard to predict the actual turn rate of a tracked robot, its centre of rotation and its current heading. It also loses a lot of energy due to skidding in a turn. Compared to a wheeled robot, a tracked robot is therefore rather inefficient on smooth terrain, but improves the manoeuvrability on loose terrain by far.

[4] see web page `http://mars.jpl.nasa.gov/MPF`

2.1.1.5 Others

Beside these main locomotion means of legs, wheels and their modifications, mobile robots in other fields like aerospace or underwater vehicles have to use different mechanisms. Robotic aircraft make use of the thermal lift by incorporating propellers or turbines. The shear force is also the main locomotion mean for underwater vehicles.

2.1.2 Perception

Mobile robots should not move sightless through their environment, but rather perceive their environment in order to avoid obstacles or reach their goals. To acquire knowledge about their environment, robots can be equipped with various sensors which can measure different properties of the environment and the robot itself.

In this section, we want to introduce two classifications for sensors and how the performance of a sensor can be measured. The last subsection shows how the sensory perceptions can be used to improve a robot's behaviour. For more details refer to [SN04].

2.1.2.1 Passive and active sensors

At first, we can classify the sensors according to the way how they take their readings. *Passive* sensors are pure observers of their environment. They only measure the energy or signals of other resources in the environment without emitting energy to the environment. For example, a sound sensor only determines the noise level in the environment.

In contrast to that, an active sensor has to emit energy to the environment to measure its desired environmental property. It then detects the reflection of the emitted energy. For example, a range sensor may emit ultrasonic waves in order to determine the robot's distance to some objects by measuring the difference of time between sending the signal and receiving the reflection. Compared to passive sensors, active sensors can gain more information about the environment, but consume more power and are more sensitive to disturbing sources. The measurements of a sensor may be affected by the outbound energy of a sensor or other devices. For example, several ultrasonic sensors are not able to distinguish between their sent signal or an external reflection.

2.1.2.2 Categories

Another classification divides the sensors into groups of the same class of properties they measure. *Proprioceptive* sensors inform a robot about its internal state. They therefore measure values like the motor speed, joint angles or battery voltage.

On the contrary, *exteroceptive* sensors take their readings in the environment and determine properties of the environment or the relation between the robot and the environment. In the literature [RN03], these exteroceptive sensors are split into different subgroups. The group of range sensors determines the physical closeness to other objects (e.g. touch sensor or ultrasonic sensor) or the localization in a fixed reference frame (e.g. GPS sensor). Image sensors provide vision to the robot. That can be detailed images of the environment with high resolution cameras or a special selection like the colour of the underlying surface. A third group may be heading sensors which measure the orientation of the robot in relation to a fixed reference frame (e.g. compass sensor).

2.1.2.3 Measuring performance

A wide range of sensors is provided by different manufacturers. Some sensors may fit better in the laboratory environment while others support outdoor use. They also vary in their overall performance characteristics so that we need a set of properties how we can quantify the performance of a sensor. [SN04] introduces a number of different properties to measure, but we only focus on a few of them.

The *dynamic range* indicates which values can be detected by the sensor. The lower and upper limit of this dynamic range describe the minimal and maximal value the sensor is able to measure.

The *resolution* is the minimal difference between two values the sensor is able to detect. It denotes which change to the measured value is recognized at minimum. The *sensitivity* directly refers to the resolution. This is the ratio of output change to input change. While we always want a high sensitivity, the sensor should have a *cross-sensitivity* as low as possible. The cross-sensitivity indicates the sensitivity of the sensor to disturbing sources. It measures how environmental signals influence the function of the sensor.

To measure the overall performance of the sensor, the parameter *accuracy* and *precision* are used. The accuracy indicates the conformity of the sensor's measurements to the true value over a longer period of time. That means it depicts how far the values deviate from their environmental inputs. On the other hand, the precision describes the reproducibility of the readings. It recognizes how conform the taken measurements are if the environmental inputs are not changed.

How fast the sensor is able to provide a stream of readings is given by the *bandwidth* or *sampling rate*. It is measured in hertz how many measurements can be realized per second.

2.1.3 Navigation

With the aid of sensors and actuators like wheels or legs as well as of a control unit for planning, mobile robots gain the competence to navigate in their environment. According to [SN04], mobile robots have to *perceive* their environment with sensors and interpret the sensor data to construct an own model of their environment. In order to navigate through this environment, they have to determine their position in it by *localization*. With *cognition*, they plan how to act and move to achieve their goal so that afterwards the actuators can be controlled to the desired *motion*.

Since the last two chapters covered locomotion and perception, the topics left are localization, mapping and path planning where mapping is the construction of a map of the environment and path planning the process of planning how to reach a desired location. All three topics suffer from the problem actuators and sensors cause. Sensors might return faulty or divergent values due to changing environmental conditions. Beside this *sensor noise*, we have to face the problem of *sensor aliasing*. This means that even with a large number of sensors two differing places might trigger the same perception so that the robot's percepts cannot distinguish between these different places. For example, a range sensor returns the same distance reading whether the object in front of the robot is a solid wall or an avoidable obstacle, so this information is not sufficient to decide whether to turn around or to evade. In addition, also the actuators may cause *actuator noise*. For instance, depending on the quality of the underlying surface the robot moves differently due to skidding.

2.1.3.1 Localization

Localization is the determination of the position of an object on a map. This object does not need to be the robot itself, but many applications in mobile robotics are based on that self-localization. The advantage of the map-based approach is that the map can be communicated to a human operator so that the operator can get a picture what the robot *believes* its environment is like and why it chose its navigation path. With the introduction of a map, there comes the risk of false beliefs. Especially, if the map is faulty and diverges from the real world, the robot builds its whole navigation plan on an incorrect basis.

The most common techniques for localization are probabilistic approaches. These approaches assign and update probabilities to every position on the map where the robot might possibly be located. Depending on that, the robot should be able to plan its path to achieve its goal. The most common techniques are Markov localization [Fox98] (or its improvement to Monte Carlo localization [FBDT99]) and Kalman filter localization [SN04, May90]. Some additional techniques are mentioned in [SN04, BEF96].

2.1.3.2 Mapping

All localization techniques rely on a previously known map. However, a robot should be able to navigate through a completely unknown environment because in many applications like rescue services a map cannot be provided beforehand. This problem of navigation with on-the-fly map construction is referred to by *Simultaneous Localization and Mapping Problem SLAM* [RN03]. The robot should be able to be placed in an unknown environment on an unknown position. From there on, it should explore its environment with sensors, "draw" a map and localize itself as well as possible other objects on it. This bears the problem that mapping and localization depend on each other. While mapping the robot needs to know where it is placed and where other objects are located, but while localizing itself it needs some kind of representation of the environment to determine its possible positions.

Mapping is an active research field. The most common class of algorithms for mapping is *EKF-SLAM* which is based on an extended Kalman filter [RN03]. In 2002, it was replaced by the *FastSLAM* algorithm that uses a Rao-Blackwellised particle filter (see [MTKW03]). More techniques for mapping are summarized in [DB06a, DB06b].

2.1.3.3 Path planning

Finally, the robot has to decide where to go in order to achieve its goal. The robot should identify the exact path to follow based on the beliefs about its own location on its map. Thereby, it is constricted by obstacles in the environment and its own mobility (that means that the robot might not reach every desired position due to constraints as degrees of freedom). As mobile robotics was established after the invention of manipulators, path planning for mobile robotics is based on path planning for manipulators. While manipulators have normally several degrees of freedom, mobile robots only need up to three degrees of freedom so that the path planning for mobile robots is way less complex.

The obvious representation of a path planning problem is the *workspace representation* [RN03, SN04]. The robot, the goal position and all obstacles are represented with the standard coordinates of the physical space. For manipulators that is normally given by space coordinates in x-, y- and z-representation. For planning a path from its current position to the goal position, an exact sequence of positions in this workspace representation must be defined. Thereby, the problem is that not all positions in the physical space (also without obstacles) are reachable for the manipulator because the joints cannot be arranged accordingly. Therefore, a planning module has to keep track of all constraints to that physical representation.

Thus, a more common representation is the *configuration representation* [RN03, SN04]. In this representation, the position of a manipulator is given by the configuration of all its joints. In this representation, it is easy to determine which positions are not reachable as various joint angles mutually exclude each other. Also obstacles have to be represented by the configuration the manipulator would take to reach them. A path in such a map of possible configurations is then easily planned because no additional constraints have to be taken into account.

Configuration representation is also used in mobile robotics. The position of a mobile robot is normally given by its x- and y-coordinate on the ground and an angle θ that describes the heading of the robot. For simplification, it is often assumed that the robot is holonomic and that it is a point. For an holonomic mobile robot, we can reduce the configuration space to the two dimensions of x and y and do not have to take steerability problems into account because the robot can move in any direction without shunting any adjusting of its heading. By assuming that the robot is a point, we can neglect that the robot needs some space around the actual path not to collide with obstacles. It is sufficient to plan a path through the environment where all obstacles do not lie on the path. Therefore, the obstacles have to be inflated by the radius of the robot. A detailed discussion of this is given in [SN04].

Basing on the configuration representation, the actual path planning takes place. For that, the classical techniques are road map techniques [RN03, SN04] (based on Voronoi diagrams [Row79] or a visibility graph [LW79]), cell decomposition [BL83] and potential field methods [SN04, GC02]. Road maps reduce the robot's space to a one dimensional representation, the *road map*, in which path planning is simpler than in a multidimensional representation. The idea of cell decomposition is based on the decomposition of the whole search space into simple regions, the *cells*. A path of cells from the initial position of the robot to the desired goal position is computed while within each cell a path from boundary to boundary is computed. The third class of methods is based on the idea that a potential field is spanned across the robot's map. The robot follows this field where the goal is attracting the robot and obstacles are repulsing it.

2.1.4 Architectures

Lastly, we want to have a look at some abstract architectures how mobile robots can be implemented to accomplish their task. A good overview of these different architectures is given in [SK08, RN03, Mat92].

2.1.4.1 Reactive control

The *reactive control* is the simplest control we want to introduce. This technique tightly couples sensors and actuators together. A reactively programmed robot just takes its perceptions and responds directly to them by actuating according to rules. This makes the approach fast because it does not have to fit the robot's perceptions into a world model and reason about them. It therefore does not consume a lot of space and computational power. This technique is often used in changing environments where a world model would have to be adapted frequently. Unfortunately, without that world model a reactive robot cannot learn and does therefore not improve over time.

2.1.4.2 Deliberative control

With *deliberative control*, all actions are considered beforehand according to an elaborate world model. The control system processes every perception to adapt and extend the current world model of the robot. Afterwards, it reasons about its perceptions in its believed environment and plans the further actions of the robot. That gives the robot the chance to adapt to a specific environment and to learn how to act upon that environment. For that, more computational power and space is needed as in the reactive approach and the "reactions" to a specific perception may take a longer time. Therefore, that control is only usable if there is sufficient time to reason about the next actions.

2.1.4.3 Three-Layer architecture

As the reactive and deliberative approaches hold complementary advantages and disadvantages for controlling, the *three-layer architecture* [GBMP97] tries to combine them. This hybrid control has a component to react fast without any reasoning to immediate concerns such as avoiding obstacles. Another component takes the deliberative approach where solutions to complex tasks are planned. To benefit from each other, an additional layer is needed for coordinating the interactions of the reactive and deliberative layer. This layer takes the deliberative plan and transmits it to the reactive layer. On the other hand, it communicates to the deliberative layer whenever the reactive component has to override the decision of the deliberative component to react to immediate situations.

2.1.4.4 Behaviour-based control

The *behaviour-based* approach structures the overall activity of the robot into several atomic behaviours where each represents a specific activity resulting from specific perceptions. Thereby, each behaviour is not limited to simple reactive control, but can have a state and a model of the environment. Based on this, each behaviour decides or plans which action to take. The overall activity of the robot results from the interaction and arbitration of the behaviours. On the one hand, the behaviours can exchange their beliefs and decisions so that each behaviour is able to improve its own model based on the beliefs of the others. On the other hand, behaviours can be organized in a kind of hierarchy known as the subsumption architecture [Bro86]. At the beginning, a standard behaviour is executed which is interrupted by specialized behaviours whenever necessary. For example, a robot normally wanders around, but whenever an object is detected, the behaviour for avoiding takes the control. There is a wide range of literature for behaviour-based programming where [Ark98] and [Jon04] give a good overview.

2.1.5 Open problems and current research

Mobile robotics is a highly active research field. Beginning with robotic hardware like sensors, actuators and their mounting, current research is directed to specialized coordination mechanisms for example for natural walking and outdoor locomotion, exact localization techniques, mapping approaches for dynamic environments, SLAM solutions or accurate and fast navigation. A good overview of the current state of the art in robotics can be found in [SK08] while [SN04] focuses on mobile robotics in special. For more information on special topics refer to the journals *Autonomous Robots* and *Artificial Life and Robotics* published by Springer or visit the conferences of the *IEEE Robotics and Automation Society*[5] or the conference *Robotica*[6].

2.2 Swarm robotics

A single robot is able to carry out various tasks for us so that it facilitates our everyday work and life. Nevertheless, some tasks may be too complex or extensive even for a robot or can be solved faster and easier by a group of collaborating robots. Division of labour, teamwork and cooperation of robots may be the key to even more capable and powerful robotic systems.

[5] see web page `http://www.ieee-ras.org`

[6] see web page `http://robotica2010.ipleiria.pt/robotica2010`

One of the main problems in collaborative robotics is how to coordinate the teamwork of several robots. Swarm robotics is such an approach mainly inspired by the observation of social insects like ants, termites or bees. These insects are relatively limited individuals, but in a swarm they can achieve impressive behaviours like carrying large preys. Thereby, the swarms are normally not coordinated in a centralized manner; their collective behaviour simply arises from local interactions. Swarm robotics tries to apply this principle to large groups of collaborating robots [DS04a]. On the one hand each robot can be simple so that it is at low costs. On the other hand, the swarm is rather flexible and robust against failures due to its huge size and the decentralized control.

The next sections cover the field of swarm robotics in more detail. We exactly define the term of swarm robotics and give some distinguishing characteristics based on [SGBT08, Sah04]. In addition, we outline the differences to swarm intelligence and multiple-robot systems. Afterwards, we give an overview of application domains and achievements in swarm robotics while current research topics will conclude this chapter.

2.2.1 Boundary

When we want to set the boundaries of swarm robotics, we have to start with a definition of the term *swarm*. A swarm is not just a group of individuals. According to Beni who originally defined the terms concerning swarms, swarm intelligence and their applications [Ben05], a swarm is characterized by decentralized control, the lack of synchronicity and members that are simple and identical. In addition, a swarm must have a certain number of units to be counted as a swarm. Beni defined the size between 10^2 and $10^{<<23}$ [Ben05] in order to provide enough members to form a large group, but not too much to be dealt with. This large number of simple and identical individuals holds several advantages for the swarm. On the one hand, the swarm can take advantage from the redundancy of its members. It can adapt to changing conditions by scaling the size of collaborating units and is robust against failures or loss of individuals. Since the individuals are simple and no powerful high-complexity units, a loss of a single individual is rather bearable. Additionally due to the decentralized control, a swarm does not suffer from a single point of failure if the coordinator crashes. However, decentralized control without synchronicity makes the development of coordination mechanisms for the swarm challenging.

The problem-solving behaviour that emerges from such swarms is called *swarm intelligence* [Ben05, JA07]. Swarms solve their tasks by local interactions without any centralized control or global plan. The solution only evolves from exchanging locally available information between adjoining members of the swarm.

At first, biologists and engineers teamed up to transfer this kind of intelligence to robotic systems. But while the roboticists were interested in the constructive power of a swarm to form patterns or ordered structures, other researchers were looking at the analytic power of a swarm to recognize the best strategy to solve a task [Ben05]. So swarm intelligence – primarily only applied to robotics – has now a large number of application domains beside robotics reaching from biology over traffic patterns and military applications to optimization [JA07]. For example, swarm optimization mimics the behaviour of swarms of insects which leave traces in the environment other members of the swarm can follow. Two techniques of swarm optimization should be mentioned especially: *Ant Colony Optimization* invented by Dorigo [DS04b] and *Particle Swarm Optimization* developed by Kennedy [KE01]. They are used for academic problems like travelling salesman, but also for real-world problems like vehicle routing, scheduling or medical tremor analysis [DS04b, KE01]. More details on inspiration and techniques in swarm intelligence can be found in [BDT99, KE01].

In this master thesis, we want to concentrate on the application of swarm intelligence to robotics. The term of swarm robotics was first clarified by [Sah04]:

> "Swarm robotics is the study of how a *large number* of relatively *simple physically embodied agents* can be designed such that a desired collective behavior emerges from the *local interactions* among the agents and between the agents and the environment."

In this definition, nearly all distinguishing characteristics of swarm robotics compared to multiple-robot systems are mentioned.

- At first, for swarm robotics we need a *large number* of robots which are relatively *simple*. Here, incapability does not mean that the robots have to be generally simple in their hardware or software. They are rather limited in their capabilities with respect to the task they have to handle. Therefore, the large number of robots should improve the performance of a single robot on carrying out this task.

  Because the robots are not highly specialized, they are disposable. As for swarm robotics the robots are normally *homogeneous*, the swarm can build on high redundancy. A swarm of robots can therefore take more risks in losing single individuals for example in mine sweeping. Firstly the loss of a robot is bearable due to its low costs and secondly it can be replaced by a redundant unit of the swarm.

- The second property concerns *physical embodiment*. All robots should be able to perceive their environment and act upon it. It is not sufficient that they can only sense changes, they should also be able to react to them by

some kind of motion. Şahin explicitly excludes pure sensor networks from swarm robotics [Sah04] as they are not able to physically act upon the environment.

- For acting upon their environment, the swarm robots should also be able to *locally interact* with other robots or directly with the environment. They should be able to exchange information by direct communication or sensing in the environment. Thereby, it is not intended that a global communication channel exists. Such a global channel would restrict the overall communication by its bandwidth and range while local interaction can spread through a whole swarm more easily. Local interaction is also not based on a single communication channel so that coordination is distributed and cannot suffer from the loss of a single individual or resource.

- The last property is inscribed in the use of the word "agent". All robots in the swarm should be *autonomous* agents without a superior coordinator. That directly refers to the requirement of a physical embodiment because for autonomy each robot needs own capabilities for sensory perception and actuation.

All these features distinguish swarm robotics from normal multiple-robot systems. These *intentionally cooperative systems* [SK08] are founded on the knowledge of the presence of other robots. They work together to achieve a global goal and plan their actions with the knowledge of the actions, capabilities and tasks of the other robots in the system. In contrast, in swarm robotics the robots do not know the global goal and do determine their actions depending only on their perceptions and few local interactions to other robots. They only need little communication between team mates to emerge a globally coherent behaviour.

2.2.2 Achievements

Many researchers are attracted to swarm robotics and their applications. The studies cover all domains in the engineering process: the theoretical design and modelling of swarm robotic behaviour and its analysis with regard to performance guarantees as well as its practical implementation in hardware and software. The next sections give an overview of these research topics and the application domains for swarm robotics. A full overview is given in [SGBT08, Sah04].

2.2.2.1 Design

The first and probably the main topic in swarm robotics is how to design robots to gain swarm behaviour. On the one hand, we have to decide with which sensors and actuators the individual robot should be equipped to actuate autonomously upon its environment. We also have to give the robot enough means to interact with its neighbours and the immediate environment – may it be physically or virtually. On the other hand, the designer must implement a behaviour for each robot so that only from local interactions an intelligent collective behaviour emerges. That is the main challenge in swarm robotics because we have to instruct the robots beforehand that they organize themselves without any centralized control. Two approaches are mainly followed: in the *ad-hoc approach*, the designer mainly transfers the desired behaviour of a social insect to a robot by hand. In order to reproduce such a natural behaviour, an exact understanding of the process is needed. In contrast to that, *principled approaches* try to extract a general methodology from natural social behaviour and apply this to arbitrary problems. In principled approaches, we do not design special behaviours, but rather a global procedure how to derive behaviours for swarm members.

2.2.2.2 Modelling and Analysis

As systems for swarm robotics are large and probabilistic, modelling and analysing are quite helpful. Firstly, not all research groups have the means or enough space to test their swarms in real life. Large swarms of robots are yet expensive and often it is not affordable to immediately test risky approaches on a real-life platform. It is also time consuming to set up the experiments in real life so that it is desired to virtually model them first. Additionally, the swarm robotics experiments are probabilistic as they only result from local interactions which can be totally different from one attempt to another. For that, means for analysing the theoretical models can provide insights into the general properties of a specific swarm behaviour.

With modelling and analysis, it can be determined whether a new approach solves a given problem and whether it is applicable to the given platform. We can obtain guarantees for the system performance and determine parameters for the system's behaviour by systematic experiments in virtual that are hardly realizable in real life with full extent. For modelling, there exist three main techniques [SGBT08]: *sensor-based modelling* models each individual robot with its sensing and actuating abilities as well as all interactions and is therefore the basis for most simulation tools; *microscopic modelling*, however, focuses on the states of each individual robot and the transitions between the states; in contrast to both previous approaches, *macroscopic modelling* means modelling at swarm level to obtain the steady state of a model and derive behavioural parameters (a detailed explanation is given in [SGBT08]).

2.2.2.3 Robotic hardware

To take all theoretical considerations into action, one major research direction in swarm robotics is developing the appropriate hardware. For using mobile robots in swarm robotics it is not enough to provide several replications of identical robots. The robot used in a swarm must meet some special demands:

- Each individual in a swarm of robots has to perceive its environment. The use of sensors may affect other sensors either by direct interference like criss-crossing signals or by modifying environmental properties like the ambient light. This interference has to be minimized for the use in swarm robotics.
- A robot should also be able to recognize a team mate. It must be able to determine whether an environmental obstacle, an "enemy" or a member of the swarm is approaching.
- It must have the ability to leave and sense a trace in the environment like ants using pheromones.
- As local communication and interaction is a crucial feature in swarm robotics, the robot needs the possibility for a simple communication channel. This communication must be wireless, but nevertheless fast because the robots are moving in their environment and have to communicate with many neighbours. In addition, wireless communication to a computer allows easier debugging and monitoring of an individual robot.
- For local interaction, the robot also needs the capability for physical interaction.
- To provide a reasonable platform for swarm robotics experiments, the robot must also have long battery life and be small enough to fit in a laboratory setting.
- As each individual of a swarm has to be nearly identical to all others, a programming interface for the whole swarm is desired where one can design the behaviour of an individual and upload it to several members of the swarm.
- For speeding up the development of new algorithms, the developers need tools for simulating each individual robot, its interactions, the swarm as a whole and the environment.
- While maintaining all these features, the robot and all developer tools should be at low costs. If an individual robot is too expensive, this limits the application of a large number of robots and the approach that a swarm may accept the loss of a small number of robots to achieve better performance.

So far, all developed platforms can only satisfy a subset of these features and are only intended to serve as research platforms and not for real-life. However, research is making progress. For some exemplary platforms and their evaluations towards the mentioned properties refer to [SGBT08].

2.2.2.4 Applications

With swarm robotics, we generally want to solve many general problems [SK08]. These special multiple-robot systems should improve the performance of robotic solutions to tasks that are way too complex for a single robot to accomplish. They are desired to solve them faster due to parallelism and to increase the robustness – and therefore reliability and fault tolerance – through redundancy. Due to its large number of members, the swarm is more flexible and scalable to adapt to a changing environment. The amount of communication is reduced because the individuals only interact locally and may be interacting through the environment.

Application domains could be tasks that are too dangerous and/or too big for a single robot. For example in mine sweeping, the swarm is able to cover a wider region and does not suffer from an individual loss because other members can replace the "suicidal" robot. They also provide distributed sensing and are able to survey many differing parts of the environment. In addition, they are able to move to the position of interest and assemble to solve an urgent problem (see [Sah04] for more general application domains).

Current research has focused on some special applications that are mostly simulated or tested in laboratory environments. A transfer to the real world is not yet possible, but is intended. We only want to mention the main problems swarm robotics studies are currently aiming at (for a full discussion and some exemple studies refer to [SGBT08]): *aggregation* and *self-assembly* as means for self-organized *pattern formation*; *dispersion* for spreading over a wide range; *foraging* in order to return a maximum amount of "food"; *connected movement* for moving in a group while maintaining a connection; *cooperative transport* in order to join forces to achieve a specific goal and *self-organized construction* for gathering distributed objects in order to form a specific construction.

2.2.3 Current research

Throughout the whole chapter, we often mentioned current fields of interest for the research because swarm robotics is a quite new research field. We do not yet fully understand how intelligent behaviour emerges from swarms of social insects. We do not have a general methodology how to transfer the principles from swarms of social insects to mobile robots. We need more support for modelling,

simulating and analysing swarm approaches. Robotic hardware is still improvable in the means of accuracy, power consumption and cost and we cannot yet replicate all capabilities of social insects. In solving problems, current research has only focused on some special applications and also these solutions can still only provide laboratory results.

A good overview of current research as well as many links to other research groups is given in [SGBT08]. For an on-going review of the current research in the field of swarm intelligence in general, the Springer journal *Swarm Intelligence* or the magazine of the IEEE Robotics and Automation Society[7] are appropriate. Publications are mainly contributed to the International Conference on Swarm Intelligence ANTS in Brussels[8], the IEEE Swarm Intelligence Symposium[9], the International Conference on Swarm Intelligence ICSI[10] or the SAB Swarm Robotics International Workshop.

At last, we want to mention some important projects on swarm robotics: An older project is *I-SWARM*[11] during from 2004 to 2008. The project of *Swarm-bots*[12] was started in 2001 and continued by the project *Swarmanoid*[13] in 2006 terminating in 2010. A more recent project is the *ASCENS project*[14] launched in October 2010 where the researchers are aiming to enhance the relatively simple members of a swarm with intelligent agent techniques.

2.3 Intelligent agents

In the last section of related work, we want to focus on intelligent agents. As current research tries to improve the performance of robot swarms by introducing intelligent agent technique, we want to explore whether LEGO MINDSTORMS are suitable to be enhanced to intelligent agents. But beforehand, we have to explain what intelligent agents are, how they are implemented and how they achieve some kind of "intelligence". As we want to explore a multiple-robot system, we also have to cover how intelligent agents can work together. Our last focus is the current research in the field of intelligent agents.

[7] see web page `http://www.ieee-ras.org`
[8] see web page `http://iridia.ulb.ac.be/ants`
[9] see web page `http://www.ieee-ssci.org/2011/sis-2011`
[10] see web page `http://www.ic-si.org`
[11] see web page `http://www.i-swarm.org`
[12] see web page `http://www.swarm-bots.org`
[13] see web page `http://www.swarmanoid.org`
[14] see web page `http://www.ascens-ist.eu`

2.3.1 Definition

We want to split the term of *intelligent agents* into two parts to understand what they really are. Even if there is no universally accepted definition of an agent, we want to hold to the definition Wooldridge [Woo09] gives:

> "An agent is a computer system that is *situated* in some *environment*, and that is capable of *autonomous action* in this environment in order to meet its delegated objectives."

In this definition two main objectives have to be highlighted. First of all, the agent has to be *situated in some environment*. This means that an agent does not work for itself, but has to take into account its environment for achieving its goal. It has to perceive its environment, reason about its perceptions and "think" about possible reactions to the given circumstances. It can therefore plan how the environment might influence its actions. Secondly, it has to decide and plan its action rather *autonomously*. Autonomy means thereby that we delegate some goal in a high-level description to the agent and the agent is able to make a plan how to achieve this goal in particular. Therefore, the agent cannot assign new goals to itself, but is able to divide the delegated goal into subgoals to accomplish.

For an intelligent agent, we additionally need some kind of intelligence according to which the agent solves its problems in a smart way. As the term of "intelligence" is not easy to define in general, we follow the approach of Wooldridge [Woo09] and give three main properties an intelligent agent should have:

- We want the agent to solve a given task autonomously. It therefore has to exhibit a goal-directed behaviour. Whatever the agent decides to do should contribute to the solution of the problem so that at the end the desired goal is reached. This property is called *proactiveness*.

- An agent that only exhibits a proactive behaviour is rather inefficient if set into a changing environment or if it cannot observe the environment completely. For example, if an agent has to move from one place to another, a simple goal-directed behaviour would be to compute the direct way to the desired location and follow it in any case. Such an agent would not be able to avoid obstacles on the computed way since it is only able to sense the changes in the environment, but not to alter its way according to it. What an agent needs to adapt to these new circumstances is *reactiveness*. Reactiveness gives the agent the ability to react to unforeseen situations and for example to avoid an obstacle.

Nevertheless, it is not enough to only implement a reactive agent since this agent is only able to react to its perceptions, but cannot pursue a goal-directed strategy. An intelligent agent has to combine reactive and proactive behaviour in a reasonable way. It has to find the right balance between simple reactions and goal-directed actions as avoiding obstacles, but nevertheless keep the direction to the desired location.

- The third property of *social ability* aims towards the active exchange of information with the environment. That does include the communication with humans as well as other intelligent agents. Thereby, an intelligent agent shares its goal to other individuals in order to negotiate or cooperate for collaborative or competitive solutions to its goal. In this context, intelligent agents can either collaborate to enforce the power of each individual or compete for a task for example in auctions.

2.3.2 Abstract architectures

Following this definition of an intelligent agent, there exist two different strategies how to implement intelligent behaviour. They both differ in the basis on which their decisions rely. A *reactive* agent founds its decision only on the present perceptions and not on any previous sensory input or experience. It simply responds to the current environment. Thereby, it is important not to confuse the property of *reactiveness* with the abstract reactive architecture for an intelligent agent. Reactiveness means that the agent is generally able to respond to changes in the environment while a reactive architecture should include reactiveness, proactiveness and social ability. A reactive agent is therefore able to react to changing circumstances and nevertheless to work towards a goal based on its current perceptions.

A more powerful architecture also includes the current *state* of the environment and history of perceptions, actions and effects into the process of decision making. In each step, the intelligent agent adapts this model of the environment to the current perceptions so that it is able to observe the environment over a longer period of time. Based on this model of the environment, the agent can decide what to do where not only the current perceptions, but also previous perceptions are included and interpreted.

In summary, a reactive agents founds its decisions only on the present while an intelligent agent with a state takes also the past into account. In both architectures, it is possible to respond to the current perceptions, but also to work towards a specific goal.

2.3.3 Categories

After we have discussed two rather abstract architectures for intelligent agents, we want to focus on how an agent's behaviour can actually be implemented and where the agent gains its "brain" to plan its actions. In this chapter, we will see how tightly connected the fields of mobile robotics and intelligent agents are as the explained approaches include some of the architectures described for mobile robotics. The next sections introduce the behavioural agent and contrast it with the reasoning agent. Hybrid forms conclude this chapter.

2.3.3.1 Behavioural agent

The first approach is the *behavioural agent*. Such an agent is founded on the same idea like the behaviour-based control in chapter 2.1.4.4 on page 16: the overall behaviour of an agent emerges from the interaction of various simpler or atomic behaviours. Each behaviour accomplishes a specific task whereas the combination and arbitration of all behaviours solves the overall problem. In the original version of this behaviour-based control, the behaviours could not create a model or representation of their world, but current implementations also include a state of the environment in order to decide what actions to take next. As said in chapter 2.1.4.4 on page 16, the most common architecture for such a behaviour-based control is the subsumption architecture [Bro86] that assigns a priority to each behaviour. If several behaviours are runnable at the same time, the coordinator or arbitrator simply takes the behaviour with the highest priority into action. The main problem with this architecture is that it cannot easily be traced how the overall behaviour emerges since it arises from the local interaction and arbitration of various behaviours.

2.3.3.2 Reasoning agent

The main difference between a behavioural agent in its original version and a reasoning agent is the ability to represent an agent's environment. A *reasoning agent* founds its decision on a model of the environment which it is continuously adapted to its perceptions. It therefore has to decide which information about the environment it needs to construct a usable model of the environment. It must also interpret the sensory input depending on its world model and translate the values into the corresponding internal information. After that, the agent has to reason about its next actions based on that representation in reasonable time.

Deductive reasoning agent A *deductive reasoning agent* represents its environment with logical formulae. For that, it has a number of predicates with which it can map the state of interesting objects in the environment. The whole

model of the agent's world is therefore only an accumulation of such predicates. Based on the predicates, we can define some deduction rules what actions the robot should do under which circumstances. The decision-making process of the agent only consists of deriving a valid action from the current state of the environment (modelled by the predicates) and the deduction rules. This means decision-making is a problem of proving and can therefore be solved by a theorem prover. This is also the main disadvantage of this approach since theorem proving is computational complex and the agent may not find a solution in reasonable time.

Practical reasoning agent A *practical reasoning agent* breaks the process of decision-making into two phases: in the first phase, it deliberates *what* it wants to achieve in the next steps while in the second phase it plans *how* this can be achieved. In the *deliberation phase*, the agent determines all options describing which states can be obtained by taking its current beliefs about the environment and its current intentions into account. From these options, it chooses the "best" option according to some filtering function.

If the agent has decided which state should be reached, it can plan how to obtain it in the *means-ends reasoning phase*. Thereby, it takes into account the desired goal or state, the current model of the environment and all possible actions and constructs a plan for reaching the goal. How the plan is derived is not determined and can be implemented freely by the designer.

2.3.3.3 Hybrids

The two approaches of a behavioural or reasoning agent, both bear advantages and disadvantages. While the behavioural approach is faster and reacts in-time to changing conditions, the reasoning approach can reason about the whole history of perceptions and plan the next actions according to it. Obviously, some want to combine these advantages so that new hybrid forms evolve.

The first is mainly the concept of behavioural control in chapter 2.1.4.4 on page 16. Thereby, each behaviour is extended by a world model on which it can base its decisions. The overall behaviour nevertheless emerges from the interaction of all behaviours.

The second hybrid form is mainly the three-layer architecture already explained in chapter 2.1.4.3 on page 15. Here, for each approach – simple reacting and reasoning – an own component is created. They both decide what to do next while a third component glues them together and controls that the reactive layer responds fast to changing conditions and the deliberative layer communicates its plans to the executing reactive layer.

2.3.4 Multiagent systems

Like the definition of a single agent, literature does not give a precise definition of the term of a *multiagent system*. Therefore, we provide the very general definition from Shoham [SL09]:

> "Multiagent systems are those systems that include *multiple autonomous* entities with either *diverging* information or diverging interests, or both."

In this definition, we have to count the autonomous entities as our previously defined intelligent agents. More important is the use of the term "diverging". That shows that in a multiagent system we cannot assume that all entities have the same information, model of the environment, goals and action capabilities. Even the initial "equipment" of the intelligent agents might not be same. Therefore in multiagent systems, we have to face many autonomous individuals where each might be aiming to a different goal. They might either collaborate if they find overlapping interests or compete on the same resource.

Facing this heterogeneity, we need a general approach how to bring these individuals together so that they work on a collective solution. The next section introduces the idea of problem decomposition for collective problem solving. Afterwards, we want to discuss how distributed planning evolves in multiagent systems.

2.3.4.1 Collective problem solving

For collective problem solving, a three-stage activity is recommended [Dur99, Woo09].

Problem decomposition The first stage is a recursive decomposition of the problem. The task is split into several smaller, but nevertheless independent subtasks. That means, the multiagent system identifies single activities of the main problem which can be carried out independently from the others. It further tries to break down these smaller activities into even simpler tasks and so on. When to stop this *problem decomposition* is one of the challenges of this stage. If the problem is decomposed until the subproblems only represent atomic actions, we have to coordinate the combination of a huge number of subresults afterwards. If not fully decomposed, we have to find a limit when to stop the decomposition.

This approach sounds very easy, if we think of a global coordinator of this problem decomposition. However in multiagent systems, we often do not have a single individual that knows the capabilities of all participating agents or is able to break down the problem reasonably due to the lack of expertise. For example in swarm robotics, we also do not want to have such a global coordinator. In swarm robotics, each agent has to decompose the problem by itself or an inherent distribution of the task must emerge.

Task sharing After the problem has been split into manageable subproblems, the multiagent system has to determine which member of the system should solve which subproblem. To identify the adequate agent for a specific task, Wooldridge [Woo09] suggests to initiate a kind of auction. The manager of the decomposition announces the subtask to all available agents in the system. Each agent which has the necessary capabilities for the subtask responds with a bid that describes its exact capabilities regarding the subtask. The manager evaluates all biddings and assigns the subtask to the most appropriate bidding agent.

Subproblem solution and result sharing After this task allocation, each agent has to find a solution to its given subproblem. If an agent does not have enough information or abilities to solve its subtask, it is allowed to ask for help. Other agents can even share their own results to subproblems in order to help another agent to solve its subproblem. By cooperatively exchanging information, several agents might already collaborate in this stage.

Solution synthesis Finally, all "small" solutions have to be gathered and integrated to form the solution for the whole problem. The incorporation of subresults may already take place in the phase of subproblem solution because for solving some subproblems the agent may already need some subresults. However, in this phase of solution synthesis all remaining subresults are merged according to the former problem decomposition.

Similar to the problem decomposition, this step of solution synthesis may be easy if the multiagent system has a global coordinator. If this is not the case, we have to think of a distributed solution to the synthesis where either many agents together execute the whole synthesis or a single agent is determined to take over this last step.

2.3.4.2 Distributed Planning

Planning activities of a group of agents is another obvious problem in multiagent systems. When we want a group of agents to intelligently solve a specific task, the

agents must be able to join forces and to come up with a collaborative solution to the problem. Thereby, the agents have to take into account each agent's activities and that some activities may interfere with other. For that the system needs an overall plan how it should coordinate its individual members. However, we do not want to give a detailed plan to the system, but rather the task itself and all agents in the system should develop a plan together. Of course, distributed planning is just a specialization of collective problem solving, but there are some special techniques for distributed planning. In the following, we do not want to suggest concrete solutions to distributed planning, but introduce three different types of distributed planning [DUR99] which differ in the definition of distribution.

Centralized planning for distributed plans The first kind of distributed planning is aiming to generating a distributable plan which was created in a centralized manner. A special coordinator agent breaks a task into several subtasks that can be executed in parallel and are not dependent on another. These subtasks are then passed to different agents and by solving the subtasks the agents will achieve the goal of the distributed plan (if the plan was correct and nothing unpredictable happened).

Distributed planning for centralized plans If a task is so complex that one agent alone is not able to plan all activities because special knowledge is needed, we are facing another type of distributed planning. In this type the task itself may not be split into several subtasks, but rather the process of planning. A general-purpose planner might divide the overall problem into several subproblems and give these subproblems to specialists. The planning specialists have specialized knowledge in a particular field and generate a plan for subproblems concerning this field. All subplans are given back to the general-purpose planner which synthesizes an overall plan from all subplans. Note that the specialists may not need to execute their subplans; they are only generating a procedure an executing agent should follow.

Distributed planning for distributed plans This last kind of distributed planning for distributed plans is the most challenging process. Here, the generation of the plan as well as the plan itself are distributed. That means that a group of agents has to cooperate to form the plan. Additionally, subtasks have to be formulated in this plan which can be executed parallel, but nevertheless solve the overall problem. The main problem of this distributed planning is that the agents should not conflict with each other, but should rather help each other to achieve their subgoals. Approaches to this kind of planning can be found in [DUR99].

2.3.5 Current research

Current research in the field of intelligent agents, especially regarding to multiagent systems is very active. The main topics are how to achieve cooperation between several agents, how to coordinate the cooperation, how to communicate tasks and results in a multiagent system and how to negotiate between several agents when they have differing goals. In addition, current research tries to transfer intelligent agent techniques like problem solving, planning and learning to a distributed system of agents.

For a good overview of the current state of the art in the field of intelligent agents and multiagent systems refer to [Woo09] and [SL09]. The current research is mainly discussed on conferences like the AAAI Conference on Artificial Intelligence[15], the AAMAS International Conference on Autnomous Agents and Multiagent Systems[16] or the International Joint Conference on Artificial Intelligence IJCAI[17]. In addition, the IEEE Robotics and Automation Society[18] may provide some good articles on intelligent agents.

[15]see web page `http://www.aaai.org/Conferences/AAAI/aaai.php`
[16]see web page `http://www.aamas2011.tw`
[17]see web page `http://www.ijcai.org`
[18]see web page `http://www.ieee-ras.org`

Chapter 3

Lego Mindstorms

If one tries to find an inexpensive, but still powerful platform for experimenting with swarm robotics, one should not be shy to take toys into account. Lego Mindstorms might seem too simple and ordinary to serve as prototypic robots but they provide manifold possibilities to build, program and analyse solutions to real-world problems as the next two chapters illustrate.

Lego itself is primarily a vendor for toys. The carpenter Ole Kirk Kristiansen founded the Lego company as a manufacture for wooden toys in 1932. Its name Lego is derived from the Danish phrase "leg godt" which stands for "play well".

The history of Lego can be divided into four phases. The first phase was characterized by the invention of the interlocking system for bricks in 1958. Such bricks made it easier to build and alter own Lego models what is also one of the advantages of Lego Mindstorms. In the second phase, Lego extended its range of products to mobile vehicles. In 1977, the product line of Lego Technic, on which the Lego Mindstorms are based, followed extending the mobile vehicles with many more technical elements like pneumatic pieces or remote controls and an upgraded interlocking system for rotary connections. Soon afterwards the Lego company tried to push forward their research in technology and learning processes. Therefore they entered a partnership with the Media Laboratory at the Massachusetts Institute of Technology MIT in 1984. This partnership drove the company to the incorporation of intelligence and behaviour into their products. The first result of this symbiosis was the Lego Technic Computer Control in 1986 which is the basis for all of the following computer controlled robotic systems. Unfortunately this computer control did find very little audience so that it was long until this partnership dared to present a new invention. It was not until 1998 that Lego announced their first robotic model, the Lego Mindstorms RCX. Again deriving from the partnership with the MIT it is not only possible to build an own mobile robotic model with this Lego Mindstorms RCX

but also to give a special user-defined behaviour to it. LEGO MINDSTORMS are mainly made up of LEGO TECHNIC pieces, but can be enriched with the normal LEGO SYSTEM bricks. The speciality about LEGO MINDSTORMS is that they have special motors and sensors described in the sections below to perceive their environment and react to it according to their perceptions. LEGO MINDSTORMS are therefore able to explore their environment in an intelligent way by not simply moving around but by sensing their surroundings. To reason about their perception, LEGO MINDSTORMS have a main computing unit that can be fully programmed by the user (see chapter 3.3 on page 46).

Today, the LEGO company sells its products into 160 countries all over the world and has about 7000 employees. In 2009, they had a pretax profit of 2.89 billions DKK which is approximately 390 millions EUR. More about the history and the facts of the LEGO company can be found at their web page [LEG10a].

However, they keep with their company mission "Children are our role models" [LEG10a] stated in 2000. Even the more technical products like LEGO MINDSTORMS are targeted to the audience of children. Children should be inspired to creativeness and systematic consideration. LEGO wants to promote the master builders of tomorrow. While their target group are children between 10 and 16, the owners of LEGO MINDSTORMS are mainly adults with an average age of 35 [BAG07]. Nevertheless, LEGO MINDSTORMS find their way not only into the schools and educational institutes (see for example the German web page for LEGO MINDSTORMS at school in [LPE09]); they are even used in research and fast prototyping for professional real-time applications.

The core of the robotic model in 1998 is called RCX, Robotics Command System. It contained an 8-bit processor at 16 MHz and only 32 kBytes of memory. But in 2006, a newer model was invented with the NXT, the core computing unit seen in figure 3.1 which is used in this thesis.

This LEGO model improves the old RCX model with new motors, which are capable of perceiving their actual movement in the environment, and several complex sensors. The first two sections of this chapter give a detailed view of the motors, sensors and extension kits for the new NXT. As mentioned before, the microcontroller has also been improved to run 10 to 50 times faster and for example been enhanced with I^2C communication as well as Bluetooth and USB instead of infra red communication what is described together with the possible programming languages and interfaces in the ensuing section. Finally, we have a look at how the NXTs can physically and virtually interact before the chapter is concluded by a short summary of the advantages and disadvantages of using LEGO MINDSTORMS in research.

Figure 3.1
The Nxt *with three motors and four different sensors (in the lower part from left to right: touch, sound, light and ultrasonic sensor)*

3.1 Servo motors

Figure 3.2
A servo motor

The first elements of Lego Mindstorms that we want to discuss are their *servo motors* seen in figure 3.2. The Lego Mindstorms Nxt 2.0 Kit contains three of these motors with which the movements of a Lego Mindstorms robot can be controlled precisely. Different Lego parts like wheels or gears can be connected to the motors. A simple "forward" command will start the motor to rotate the orange disk on the left edge of the image and any attached brick. Due to an built-in proprioceptive rotation sensor (see chapter 2.1.2.2 on page 11 on the different categories of sensors) the motor is able to keep track of its covered distance or current velocity. The rotations of the orange disk can be measured in degrees or full rotations which refer to 360 degrees. A precision of one degree is reached while using this rotation sensor [LEG10d][Bom07]. As each rotation is reported to the main computing unit, the current speed of the motor can also be easily computed. So with the aid of this sensor a robot can be controlled to drive a specific distance with a specific speed. While using two motors, the Lego firmware (as well as the other firmwares described in chapter 3.3.2 on page 48) additionally synchronizes the motors so that a "forward" call rotates two attached motors synchronously and the robot drives a straight line [Bom07]. The user therefore does not need to control each motor at low level, but has only to indicate the direction of movement for the whole robot while the firmware is translating and regulating the actual motor activity.

Although the motors can measure their moved distance and speed and the firmware takes care of synchronization of motors, the values returned from the rotation sensor cannot be used to reliably compute the position. Firstly, the angle per rotation is determined with a precision of one degree, but after a longer distance the measurement errors accumulate that one cannot always trust the returned degrees. Secondly, the motors are not able to detect any slipping wheels due to a challenging ground. Therefore, if the user needs reliable information about the localization of the robot, he should use additional techniques for localization and navigation as described in chapter 2.1.3 on page 12. Sometimes a more practical problem is the weight, size and shape of the servo motors. In contrast to the old RCX motors, the motors weigh 81g each which is nearly double of the RCX motors. The motors must therefore be tightened very well to the chassis of the robot. Unfortunately, they have few holes or tubes to connect to. Despite all that these motors can be used to activate wheels, gears, legs, arms or anything else.

3.2 Sensors

The LEGO MINDSTORMS NXT 2.0 Kit includes four sensors: two touch sensors, an ultrasonic sensor and a colour sensor. However, many more compatible sensors are available: LEGO additionally offers a light, sound, compass, RFID and acceleration sensor. All of the mentioned sensors are exteroceptive sensors as described in chapter 2.1.2.2 on page 11 as they acquire information about the robot in its environment. That may be properties of the environment itself (the colour of the surface or the noise level in a room) or even relations between the robot and its environment like the heading of the robot or its acceleration or tilt. More elaborate and complex sensors for LEGO MINDSTORMS are provided by third party manufacturers like HiTechnic[1], mindsensors.com[2] or Vernier[3]. A list of additional sensors can be found at the end of this chapter.

Before we have a look at each sensor in particular, we want to discuss the differences between analog and digital sensors of LEGO MINDSTORMS. Analog sensors use PIN 1 of the input ports of the NXT (see chapter 3.3 on page 46) which limits the sampling rate to a maximum for all analog sensors as explained in the following: every active analog sensor emits energy to the environment. It must therefore be supplied with energy by an external source, here the main computing unit. In LEGO MINDSTORMS, an active sensor gets 3 ms of power supply from its main computing unit, the NXT. Afterwards the value of this sensor is measured in 0.1 ms. That means that another read of an active sensor can be started each

[1] see web page `http://www.hitechnic.com`
[2] see web page `http://www.mindsensors.com`
[3] see web page `http://www.vernier.com`

3.1 ms which approximately refers to a sampling rate of 333 Hz. One could think that passive analog sensors would have a higher sampling rate because they don't need time to emit energy to the environment. Unfortunately, the sampling rate for passive sensors is still at most 333 Hz. That is because active and passive analog sensors both utilize the same analog-to-digital converter which is clocked to 333 Hz. Therefore, each passive analog sensor has to wait the timespan for energy supply of active sensors before its value can be converted (for more technical details, see the LEGO MINDSTORMS NXT Hardware Developer Kit [LEG06b]). If not mentioned otherwise, all sensors described in the next sections are analog sensors.

In contrast to that digital sensors use PIN 5 and 6 for connecting to the NXT. Digital sensors use the Inter-Integrated Circuit I^2C protocol to communicate with the NXT. I^2C communication is an industrial communication standard invented by Philips [LEG06b] to communicate with peripheral in embedded devices[4] (the current specification can be found in [NXP07]). The digital sensors also include an external microcontroller for handling the sampling of the physical environment. Therefore, they are not limited by the use of an analog-to-digital converter and can sample as fast as possible. With the digital interface of I^2C communication, these sensors are able to perform their functions independently from any controlling unit such as the NXT. They only send their results to the NXT or get new tasks from the NXT while analog sensors are completely controlled by the NXT itself. Digital sensors are also able to store specific device names and individual parameters. The names are normally used for distinguishing between different sensors while the parameters can hold device specific data such as calibration values or start-up times.

In the next sections we have a look at each standard sensor that includes a short description, its range, precision, possible disturbing sources and application domains. Afterwards a short overview of third party sensors is given. For further details refer to the manufacturers themselves [LEG10c, HiT09, MIN10, VER10, COD07] or see [BAG07, LPE09, BOM07]. The reference [AFF07] is especially mentioned where not only the sensors are described in detail, but also some applications for each sensors are explained.

[4] see web page `http://www.i2cbus.com`

3.2.1 Touch sensor

The *touch sensor* has the typical shape of a LEGO sensor as it can be seen in figure 3.3. At the front there is an orange button that reacts to pressing. The button has a hole in the middle so that other parts can be connected to the touch sensor. The normal status of this sensor is not pressed which refers to the value of zero. If the button is pressed, an electric circuit is closed and therefore changes the value to one. Because this sensor only reacts to external pressure on the button and does not need to emit any energy to the environment, it is a passive sensor.

Figure 3.3
The LEGO *touch sensor*

There are two main difficulties with this type of sensor. Firstly, the sensor is only able to detect a touch contact if the button is pressed hard enough. Secondly, the programmer must take care that even a short press is registered since this sensor does only return the value of one if the button is currently pressed.

Most applications use this sensor to detect an obstacle into which the robot is bumping or to grab something that is touching a robot's hand equipped with a touch sensor.

3.2.2 Ultrasonic sensor

The *ultrasonic sensor* is able to measure the distance to an object in front of it. It is made to resemble two eyes as shown in figure 3.4 as it provides a kind of vision to a LEGO MINDSTORMS robot, but its function is rather based on "hearing". The sensor sends ultrasonic waves for 0.3 ms that are reflected by objects in reach. The sensor then waits for a reflection of its sent signal. It can detect the reflection of up to eight objects. The time between the send of a signal and the return of its reflection is used to calculate the distance to the object which reflected the ultrasonic waves. Therefore, the functionality of this sensor can be compared to the way bats measure distances.

Figure 3.4
The LEGO *ultrasonic sensor*

As this sensor must actively send ultrasonic waves to the environment, it is active. It is also one of the digital sensors for LEGO MINDSTORMS which means it uses the I^2C protocol to communicate with the computing unit and is therefore relatively independent from the main computing unit. With the ultrasonic sensor, distances between 1 and 255 cm can be measured while it reaches a precision of 3 cm. The closer an object, the more precise is the measured distance because the signal becomes weaker if the distance is bigger.

But not only the distance of an object influences the accuracy of this sensor, also the physical properties of the object may cause inaccuracies. For instance, if the object does not have a smooth surface, the ultrasonic waves are not reflected directly to the sensor but might criss-cross the distance to the sensor. Other examples are very thin, small or soft objects that do not reflect the signal in full strength or absorb the signal partially. Several ultrasonic sensors operating in the same room might also be a problem. If ultrasonic waves emitted from different sensors are criss-crossing the room, a sensor cannot distinguish which reflection is its own reflection or another signal. However, this sensor is used in many applications for object detection or obstacle avoidance.

3.2.3 Colour sensor

In addition to the ultrasonic sensor described above, the *colour sensor* in figure 3.5 provides visual input to a LEGO MINDSTORMS robot, too. Its first function of three is to detect six different colours namely black, white, red, green, blue and yellow. By using another firmware than the original LEGO firmware (e.g. the leJOS NXJ firmware), the sensor is even able to read any colour value of the RGB scale that is shown to the sensor. A value consisting of three components (for red, green and blue) between 0 and 1023 is returned.

Figure 3.5
The LEGO *color sensor*

For detecting colours, the colour sensor is an active sensor equipped with a tri-colour LED which emits red, green or blue light, and a colour detector. The third black pin visible in figure 3.5 is only a dummy that might be replaced in future versions. While in colour detection mode, all three colours have to be turned on to receive an accurate RGB value. Unfortunately, this might be a problem for example in the leJOS NXJ firmware (see chapter 3.3.2.5 on page 51) as not all three colours can be switched on simultaneously. The sensor turns on every colour separately, reads the current colour value and combines the three values afterwards. Constantly turning on and off the lights results in flickering of the lamp. Sometimes the sensor may even be overcharged so that the robot's behaviour gets stuck while reading colour values. The sensor may read colour values that are not reproducible or cannot read any colour values at all since the LED is not correctly working any more and only flickering. That is why the colour detection mode should be used very cautiously, especially in the leJOS NXJ firmware. The colour sensor should not be read as fast as possible since this slows down the whole execution of the robot's behaviour. It rather should be used to detect colours under specific circumstances, for example if the touch sensor is pressed, so that the sensor does not operate at full capacity. If a continuous

colour detection is needed, it is either recommended to read the colours without any lamps turned on if possible or to switch on only one colour of the LED and to derive the desired value from this reading.

Applications for this function of colour detection are widely spread: the behaviour of a robot can be controlled by showing different coloured objects to it, so maybe a green ball causes the robot to dance while a red one stops its behaviour. Sorting different coloured objects is another well known application for this sensor.

In addition to colour detection, the colour sensor also provides the function of measuring the light intensity in a room or of surfaces. There are two modes for measuring the light intensity: *ambient light mode* or *reflected light mode*. In ambient light mode the sensor does not illuminate the underlying surface. The sensor rather reads the light intensity passively so that this value strongly depends on the ambient light sources that illuminate the environment. Since in ambient light mode the sensor is very sensitive to varying light incidence and may return different readings of the same surface depending on the ambient light, there is the possibility to use this sensor with the LED turned on (which means as an active sensor). In this reflected light mode, the sensor illuminates the surface with one colour of its LED and reads the amount of reflected light. Thereby, it is not important which colour is turned on; the sensor is able to read the light intensity in any case. The advantage of this mode is that the reading is no longer depending on the ambient light because the sensor enlightens the surface by itself. The read values should therefore always be the same. Overall whether used in ambient or reflected light mode, the colour sensor returns a value between 0 and 100 referring to the light intensity n a room or of the surface underneath the sensor.

Of course, the sensor is very sensitive to any changes in the light sources, so the returned values may oscillate around some light intensity. Nevertheless, this light intensity mode is frequently used for example in line-following applications where the robot must follow a line by measuring the contrast between a black line and the surrounding white space. An application for detecting the ambient light intensity is a burglar alarm system on the basis of LEGO MINDSTORMS. Changing the ambient light by switching on the lights in a room could activate the system by detecting the new light source with the colour sensor.

The colour sensor even has a third function. If mounted not to face the ground but the surroundings, the three colours of the LED can function as colour lamps. Each colour can be switched on separately to emit coloured light or all can be turned on together to show white light. The colour sensor as a light source enriches a robot with the ability to communicate with its environment. A red beam for example could mean that the robot has detected something approaching

very close while a green beam means that the robot does not expect danger from an intruder. Also for localization, the colour sensor can be very useful because a light source is well detectable for cameras.

3.2.4 Light sensor

The *light sensor* in figure 3.6 is the old version of the colour sensor described in the last section. This sensor was formerly used to distinguish between light and dark and to detect light intensity. Similar to the colour sensor it can be used passively or actively referring to either ambient or reflect light mode. The only difference is that this sensor illuminates a surface with a red LED lamp instead of a tri-colour LED. Unfortunately, this sensor can neither detect colours (it is only able to read light intensities) nor illuminate its surroundings without reading the light intensity. So it is recommended to use the new colour sensor instead.

Figure 3.6
The LEGO *light sensor*

3.2.5 Compass sensor

For localization or accurate navigation, it might be useful to determine a robot's heading. With *the compass sensor* shown in figure 3.7, the earth's magnet field can be measured. This sensor is provided by HiTechnic, but LEGO offers it in its official LEGO store. The sensor returns a value between 0 and 360 degrees representing the current heading of the robot according to the earth's magnet field. Unfortunately, it is not easy to find any information how this sensor calculates the heading; the documentation reveals only that a precision of one degree can be expected. In contrast to the sensors described above, this sensor has a sampling rate of 100 Hz which means that the heading is read each 10 ms. Like the ultrasonic sensor, the compass sensor is a digital sensor (see chapter 3.2 on page 36).

Figure 3.7
The LEGO *compass sensor*

The compass sensor is very sensitive to disturbing sources. Any magnetic objects like metal items, motors, batteries or wires may cause interference and the sensor returns a heading differing from the actual heading by several degrees. This so called compass deviation can be reduced by calibrating the sensor. In calibration mode the robot turns at least two complete rotations very slowly

while it calculates a correction. After the calibration the deviation of the heading should be minimized. Nevertheless, the compass sensor has to be mounted at least 15 cm away from the motors and 10 cm away from the intelligent brick, the NXT, because these devices disturb the sensor continuously. In addition to that it is essential that the compass sensor is mounted correctly. It must be attached horizontally to the robot's chassis so that it does not vibrate while the robot is moving.

This kind of sensor is often used for localization or accurate navigation of mobile robots. If the robot is instructed to turn a specific angle, it can be verified with aid of the heading information derived from the compass sensor whether the robot did actually turn the angle or whether any drift occurred. So the robot may adjust its turn rate by itself to reach the desired heading.

3.2.6 Sound sensor

Figure 3.8
The LEGO *sound sensor*

An appreciated feature of a robot is the ability to recognize spoken commands and to react accordingly. LEGO has released the *sound sensor* seen in figure 3.8 that upgrades the skills of a LEGO MINDSTORMS robot to simple "hearing" skills. The analog sound sensor measures noise levels in both dB and dBA. In detecting adjusted decibels dBA the sensor measures the full range of sound waves, but emphasizes frequencies between 3 kHz and 6 kHz which corresponds to the range where the human ear is most sensitive. In dB mode the sensor detects the full range of sound waves without any amplification. As output, this sensor returns the sound pressure level in a range between 55 dB and 90 dB where the upper limit corresponds approximately to the level of a lawnmower. As the different dB values are not easy to understand, the LEGO firmware converts the decibels to readings in percent. Therefore, LEGO specifies on its website [LEG10d] how to read these values: 4-5 % refers to a silent living room where 5-10 % is the noise level of a person talking in some distance. 10-30 % may be a normal conversation not far away or a music played at normal level. The highest percentage of 30-100 % is reached by people shouting or music played at high volume.

However, with this sensor a robot can not only distinguish noise levels, it is also able to detect different tone pitches or patterns of sound. This opens a wide range of applications: Again imagine a burglar system that reacts to any noise in a quiet room. For this the sound sensor only needs to detect an increment in the ambient noise level, for example if a burglar cracks the door. Another robot

could for instance mime a dancer. It analyses music according to tone pitches and moves depending on them. After the dance the robotic dancer even reacts to the following applause. According to the applause the dancer would simply bow and stop or start a little extra performance.

Finally, the recognition of patterns must be mentioned. A robot with a sound sensor is not only able to detect noise levels or tone pitches, it is also able to sample sound over a longer period for example the clapping of hands. This enables a robot to react to specific patterns of clapping, for example to two claps to turn on or off its lights. Of course, while using the sound sensor each noise in the room is crucial. Unintentional claps or a casual conversation could disturb the recognition and cause unintended reactions.

3.2.7 RFID sensor

Figure 3.9
The LEGO *RFID sensor*

The *RFID sensor* shown in figure 3.9 is originally designed by CODATEX[5] and adds an interesting feature to a LEGO MINDSTORMS robot. The abbreviation RFID stands for "radio-frequency-identification". This technology allows to uniquely identify and localize especially equipped objects without any direct contact. For that, electromagnetic waves are exchanged between the RFID reader and the RFID transponder on the object. Principally, there are different ways how to implement this kind of identification but we only focus on the operating mode of the RFID sensor. For further remarks see the web pages `http://rfid.net` or `http://www.rfid-basis.de` on RFID technology. The RFID reader is able to identify special RFID tags by sending a radio frequency signal which is received and modulated by a tag according to the desired response. Thereby, the modulation encodes the unique serial number of the RFID tag or other stored data which the RFID reader itself receives and interprets.

The speciality about this technology is that there is no need for a line of sight between the reader and the receiving tag so neither objects between the two devices nor dirt upon a tag disturbs the transmission. In the case of the RFID sensor the transmitted signal has a frequency of 125 kHz. The tag, the so called transponder, must be of type EM4102. Normally an RFID reader is able to read any data sent from the stimulated tag but this RFID sensor only reads the id of the transponder which is 5 bytes long. No further data can be included into the transmitted data nor is the RFID sensor able to change any settings or values on the tag. This must be done by a different manipulation device not included in the LEGO MINDSTORMS product line. The typical reach of an RFID reader depends

[5]see web page `http://www.codatex.com`

on its application and might be between 1 cm for devices which must be directly connected to the tag and 10 m for systems used in warehousing. As the RFID sensor of LEGO MINDSTORMS is not as elaborate as needed in warehousing, the distance between the sensor and the tag may only be 3 cm to guarantee the identification mechanism.

The RFID sensor operates in three different modes: in single read mode, the sensor reads an available transponder exactly one time. Afterwards, it switches to sleep mode so that the RFID sensor needs no energy. The sampling rate in this mode is only two to three reads per second (2-3 Hz) which relates to a read each 300 to 500 ms. If a higher sampling rate is needed, the transponder should be read in continuous read mode. In this mode the sensor sends its radio frequency signal up to ten times per second (10 Hz) and does not switch to sleep between two reads. So the sensor is able to read a transponder more continuously than in single read mode and therefore samples the data in 100 ms. The third operating mode is called stop and simply lets the sensor go to sleep.

It should also be mentioned that the sensor is an active one because it actively must send a radio frequency signal to detect a transponder. This sensor again uses the I^2C protocol to communicate with the computing unit and it is therefore called digital.

Like the compass sensor, the RFID sensor is very sensitive to any disturbances. Other electromagnetic signals like Bluetooth, WLAN or signals deriving from monitors, motors or fluorescent lamps may interfere the transmission of signals between sensor and tag. Even two different RFID sensors in the same room may disturb each other. For correct identification the tag should always be directly in front of the RFID sensor. Moving the tag too far away or too high or low above might place it out of range of the RFID sensor antenna. The sensor has an LED lamp that is turned on if the sensor can read a tag so that it is possible to identify whether the tag is correctly placed.

The obvious application of an RFID sensor is the detection of special objects. The robot can be controlled by "being shown" different tags so that different behaviours are initiated depending on the identified tag. Imagine a candy vending machine. If a consumer wants to purchase chewing gum, he presents a tag holding the id for chewing gum to the machine. The candy vending machine identifies the tag, charges the price for the goods and after receiving the money it dispenses the candy. But the RFID mechanism could also be used for localization issues. For that purpose we need a map on which different tags are placed. Each tag stands for a specific position so that when the robot is moving across the map, it can easily determine where it is located at the moment by reading the tag underneath

it. Although the range of the RFID sensor seems small, for localization purposes it is highly useful because every 6 cm a different tag can be placed without the RFID sensor detecting two tags at one time.

3.2.8 Acceleration or tilt sensor

The last sensor that is offered by the official LEGO shop is the *acceleration* or *tilt sensor* that looks the same as the compass sensor (see figure 3.10). With this sensor the robot knows whether it tilts side to side, left or right or up or down according to reference [LEG10c]. In addition to the tilt the acceleration can be measured in three axis. The unit of the acceleration is the gravitational acceleration g. The limit of acceleration the sensor is able to measure is -2g respectively +2g in which the precision is 200 counts per g. The sensor can be read each 10 ms which refers to a sampling rate of 100 Hz. As all complex sensors, the acceleration sensor is a digital sensor (see chapter 3.2 on page 36).

Figure 3.10
The LEGO *acceleration or tilt sensor*

The most popular application for such a sensor is a self-levelling robot. This is a robot that is not stable due to its wheel or leg configuration, but must balance to stand upright. An example is an upright robot with two wheels side to side – a segway – which obviously always threatens to tilt and must therefore move forward and backward to hold its balance.

3.2.9 Third party sensors

Beside the sensors offered by the official LEGO shop, there are some third party manufacturers who provide even more complex sensors. A well known provider is HiTechnic which supplies LEGO with some sensors for the official shop (for example the compass and acceleration sensor). Other advanced sensors purchased by HiTechnic are for instance an angle sensor which measures axle rotation position and rotation speed, a gyro sensor which is another sensor for detecting rotation, an enhanced colour sensor which is able to distinguish more colour nuances or infra red linker, receiver or seeker sensors for remote control via infrared signals[6]. Even more elaborate sensors are offered by mindsensors.com. Their sensors are mostly adapted for special applications. For example, mindsensors.com sells a camera that tracks up to eight coloured objects, a line sensor array which makes it easier to follow a line with a PID controller (the PID controller is described in chapter 4.2.1 on page 85) or high precision distance sensors which measure

[6]see web page `http://www.hitechnic.com`

the distance more precisely by using infra red light[7]. Last, the company Vernier should be mentioned[8]. This company has a wide range of sensors on offer which are not fabricated for LEGO MINDSTORMS, but can nevertheless be used by them.

These third party manufacturers do not only offer many additional sensors to LEGO MINDSTORMS. Experimenter kits are also available to build your own sensors. Additionally, if the connection ports for sensors are running out, sensor multiplexers help to increase the number of connection ports.

3.3 Microcontroller: NXT

The most important part of a LEGO MINDSTORMS robot is its NXT brick distinguishing the product line of LEGO MINDSTORMS from all other LEGO products. By using this NXT, the normal LEGO TECHNIC bricks gain a controller that can be programmed by the user to execute any behaviour. The possibilities of moving are no longer restrained to some predefined movements or remote control. A robot with an NXT controller can perceive its environment with its sensors and execute a user-defined behaviour depending on the perceptions. That means the user can define accurately how to interpret the sensor perceptions and how to react to these interpretations. In the following, we have a closer look at the technical details and the different programming interfaces of this "intelligent" brick. The section gives only a short overview which features the NXT provides, refer to the LEGO MINDSTORMS NXT Hardware Developer Kit [LEG06b] for more details.

Figure 3.11
The NXT

3.3.1 Technical details

The NXT includes two processors which are responsible for different functions. The main processor is an Atmel 32-bit ARM processor running on 48 MHz with 256 kByte flash and 64 kByte RAM memory where the co-processor is an Atmel 8-bit AVR processor running on only 8 MHz with 4 kByte flash and 512 Byte RAM memory (refer to [LEG06b] and [BOM07] to gain more details). The 256 kByte flash memory of the ARM processor is the main storage of the NXT. It holds the firmware – the operating system of an embedded system – as well as all user-defined files or programs. As the firmware for example of leJOS NXJ(see chapter 3.3.2.5 on page 51) already needs nearly half of the flash storage, the programs should be compressed as much as possible to gain enough space for additional

[7] see web page `http://www.mindsensors.com`
[8] see web page `http://www.vernier.com`

files as sound or log files. The main processor controls the display – a black and white graphical LCD with a resolution of 100 x 64 pixels –, the USB port, the Bluetooth controller and the sound amplifier chip. It is also responsible for all input and output ports except the communication with the servo motors and their rotation sensors. The motor activity and the rotation sensors must constantly be monitored to keep track of the covered distance and of the rotations the robot has done. This monitoring and the control of the four buttons are the purpose of the co-processor. With these four buttons the user is able to interact with the NXT. For example, he can navigate through the menu of the NXT.

For the power supply of the NXT, either six AA batteries which provide 9 volts, six AA rechargeable batteries which provide between 7.2 and 7.5 volts or an optional rechargeable lithium battery-pack which provides at least 7.4 volts can be used [BAG07]. For intensive use it is recommended to purchase the rechargeable battery because the NXT needs a lot of power to supply its processors, motors and sensors with and therefore the battery must often be charged. Normal rechargeable batteries have a much lower durability than the lithium battery from LEGO. In a line-following experiment with two motors running at full speed and a colour sensor continuously reading the light intensity the robot runs about four hours with the rechargeable battery. To recharge the battery-pack it takes again approximately four hours. While charging, the NXT is fully functional and only limited by the length of the cable to the charger.

To perceive its environment and act upon it the NXT provides three output ports and four input ports. The servo motors can be connected to the output ports whereas the sensors are controlled and read via the input ports. For connecting devices to the ports, 6-wired cables with an RJ12 plug are being used. The input ports have two distinctive features: One is that the four physical input ports can handle far more than four sensors by using an extender or a multiplexer as mentioned in chapter 3.2.9 on page 45. The other speciality is that port 4 can be used as a high speed communication port for digital sensors which transmits data at a speed of 921.6 kBits per second. A normal input port for digital sensors is only running at 9.6 kBits per second. While the normal communication of digital sensors uses the Inter-Integrated Circuit I^2C protocol, this high speed port is connected to an RS485 controller after the normal input circuit. Currently there are no devices developed by LEGO which take advantage of this high speed communication port but a use for digital cameras, sound recording devices or memory expansions which need a faster communication [BAG07] is conceivable. For more technical details on the ports refer to the LEGO MINDSTORMS NXT Hardware Developer Kit [LEG06b].

The NXT can also act upon its environment by playing sound. For that the NXT is equipped with a loudspeaker to play sounds with an 8-bit resolution. With the original LEGO firmware, the NXT is only able play .RSO sound files which are generated from .WAV files by using a tool called *WAVtoRSO*. By using

other firmware like the leJOS NXJ firmware (see chapter 3.3.2.5 on page 51), the loudspeaker is also able to play an 8-bit .WAV sample file without any conversion.

The last features to mention are the NXT communication capabilities. The NXT supports standard USB 2.0 communication with a speed of 12 MBits per second. This port is normally used for uploading new firmware or transferring files or programs from or to a computer. The use of the USB port makes the NXT to a plug-and-play device but it is not possible to work with another USB peripheral device (like a USB stick) attached to the NXT at the same time. Beside the USB port, there is a Bluetooth chip available. For more details on Bluetooth communication see chapter 3.4.2 on page 55.

3.3.2 Programming languages and interfaces

An NXT can be programmed to adopt a specific behaviour. With the LEGO MINDSTORMS NXT 2.0 Kit a LEGO software called NXT-G is provided that is suitable for getting started with programming an NXT. As NXT-G is aimed at beginners and tedious to use for complex programs, some open source projects provide more powerful and feature-rich programming interfaces. There are also some efforts to provide powerful but still easy-to-program solutions for educational purposes. The next few sections give a short overview of the currently most popular programming interfaces for NXTs. A good summary of all mentioned languages can be found in [AFF07] while the web pages of each project listed in the corresponding section go into more detail.

3.3.2.1 NXT-G

NXT Graphical, or short *NXT-G*, is the original LEGO software shipped with the LEGO MINDSTORMS NXT 2.0 Kit. The official web page can be found on `http://www.ni.com/academic/mindstorms`. The software is developed by National Instruments based on LabVIEW which is a professional graphical programming software used to design, control and test products like MP3 players, cell phones, vehicle air bag safety systems or even autonomous vehicles and industrial robots [NAT10]. However, this graphical programming interface is targeted to children (mainly young children between 8 and 14 [AFF07]) and adults without any programming experience.

Programming with NXT-G is like creating a flowchart. There is a palette of predefined behaviour blocks visualized as graphical icons. Each icon represents a specific function as moving the motors, measuring a distance with the ultrasonic sensor or detecting colours with the colour sensor. The user can easily drag such a block from the palette to the programming screen while he is asked to change the standard parameters of this block as desired. By dragging several blocks in

sequence or adding blocks for special control loops, the user creates a sequential and sometimes branching and/or repeating behaviour that can be uploaded to the NXT afterwards. Figure 3.12 shows an example of the user interface.

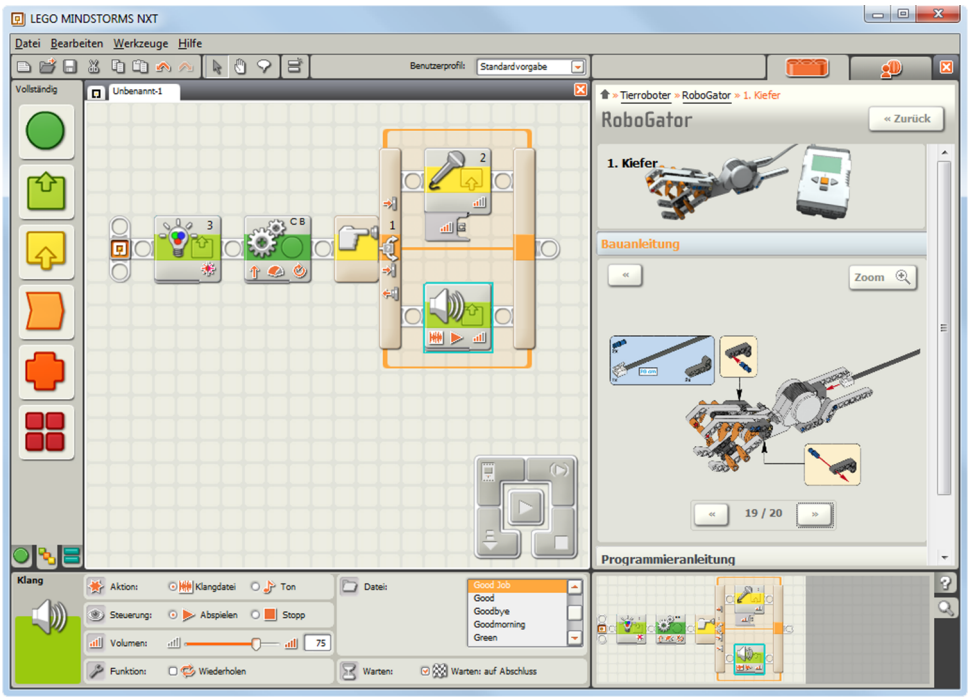

Figure 3.12

The graphical user interface of NXT-G: On the left, the standard palette of blocks can be seen. In the middle, there is the programming screen while at the right the RoboCenter with step-by-step instructions for a special LEGO *model is placed.*

NXT-G has two special features worth to be mentioned: Firstly, with the original LEGO firmware the user has the possibility to program on the NXT itself. The brick menu contains a feature where the user may choose blocks similar to the blocks in the graphical user interface and combine them to simple executable behaviours. Of course, this on-brick interface does not provide the full range of programming features of NXT-G, but it is a quick and easy way to get a first glance. Besides that the NXT-G programming interface has a built-in resource centre called RoboCenter that shows step-by-step instructions for building and programming some standard LEGO MINDSTORMS models. The RoboCenter together with the intuitive interface of dragging and dropping predefined blocks helps novices to get a first grip at programming NXTs.

On the other hand, this principle of drag-and-drop is tedious for experts. If one gets used to how to arrange the blocks or has programming experiences beforehand, too many keystrokes and mouse clicks have to be used to achieve simple

behaviours. Creating larger programs or more complex behaviours is also not recommended because even relatively short programs do not fit on the screen any more. In addition, programmers are very limited in their programming possibilities because they can only use predefined blocks. For example, there is no access to arrays, floating-point numbers or trigonometric functions which would be helpful for navigation or localization problems. The use of user-defined variables is also awkward for an experienced programmer. Another big disadvantage is that there is no debugging tool available. If the programmed behaviour results in an error, there is no other way than trial and error to fix the problem.

3.3.2.2 Microsoft Robotics Developer Studio

There are mainly two alternatives to NXT-G which are also using the original LEGO firmware. One is the *Microsoft Robotics Developer Studio*, or short *RDS*, described in this section while the other, *NBC* or *NXC*, is looked at in the following section. RDS is available at no cost from its web page `http://www.microsoft.com/robotics`. This platform was not developed to support LEGO MINDSTORMS alone but to provide a wide range of support for robotic applications. Like NXT-G it provides a visual programming language tool where the user creates its robotic applications by drag-and-drop. This tool is able to generate C#-code from the graphical representation which then can be manipulated by an experienced programmer. The tools is also enriched by a built-in visual simulation environment. In this environment the user can simulate its robotic model equipped with different sensors as compass, GPS, brightness or sonar sensor in different 3D-worlds. The 3D-simulator has therefore an elaborate Physics Engine which simulates the behaviour of the robot in various situations very realistically. However, for inexperienced users this tool might be too advanced. The tool provides a wide range of developer tools for different robotic platforms so the user has to choose and customize RDS according to his needs.

3.3.2.3 NBC and NXC with BricxCC

The second alternative to NXT-G is *NBC* or *NXC* with *BricxCC* as development environment. These two programming interfaces are recommended by the LEGO company beside NXT-G and RDS because they are also operating on the original firmware. Next Byte Codes (or short NBC) and Not eXactly C (or short NXC) are available at the web page `http://bricxcc.sourceforge.net/nbc`. Both languages are based on the very common programming language C and compile the source code to NXT-G byte code. While NBC is a very low-level assembler-like language, NXC is a high-level text-based programming language that is built on-top of the NBC Compiler. Both languages are open source, but both have the same problems like NXT-G due to compiling to NXT-G byte code. For example,

they lack floating-point numbers and arrays. Nevertheless, for experienced programmers these languages are a good alternative to NXT-G because text-based programming is possible, the source code is much smaller and the compilers generate compressed byte code that is quickly uploaded to the brick. With BricxCC a comfortable integrated development environment is also available.

3.3.2.4 RobotC

In the next three sections, we introduce three more sophisticated languages to program an NXT. Each language is based on a different high-level programming language. The first to look at is *RobotC* which is a C-based programming language for robotics. It is distributed by Robotics Academy at Carnegie Mellon University on their web page `http://www.robotc.net`. As Robotics Academy developed this language for LEGO Education it is only available for free for 30 days, afterwards a commercial license is required.. By buying the RobotC kit the user receives a optimized and fast firmware for the NXT together with the RobotC API and a well equipped development environment for programming the robot's behaviour in a text-based way. In addition, the C-based source code editor assists the user by providing various text blocks that implement some standard robotic control capabilities or behaviours or by displaying suggestions (known as autocompletion) and syntax highlighting. Thereby, the editor can be used in basic or expert mode so that programming is neither too difficult for novices nor too tedious for experts. The main advantage of RobotC is its interactive debugger. It is the only execution environment which supports automatic tracing of program execution with user-defined breakpoints. Custom programs can be debugged remotely from a computer in real-time, the programs can be suspended and each variable watched.

3.3.2.5 leJOS NXJ

The second elaborate high-level programming language for an NXT is *leJOS NXJ* (pronounced as the Spanish word "lejos" for "far") which is used in this master thesis. leJOS NXJ can be obtained from the project's web page at `http://lejos.sourceforge.net`. leJOS stands for "the Java Operating System" because "le" is an article in Spanish or French while "JOS" stands for the abbreviation of Java Operating System. NXJ symbolizes the package of the underlying Java API that provides functions specifically to the NXT brick. leJOS NXJ is developed by about 15 developers and is open source. Due to that it has a big community and currently around 200 thousand downloads.

More technically spoken, leJOS NXJ provides a Java Virtual Machine to run Java programs on the NXT. The JVM derives from the TinyVM originally developed by Jose Solorzano for the LEGO MINDSTORMS RCX and replaces the LEGO

firmware on the NXT. This JVM provides a subset of the complete Java language that is small enough to be stored and executed on the NXT. leJOS NXJ also contains a library of Java classes to get access to the specific NXT features like communicating to and controlling motors, sensors and the built-in display.

For implementing programs for a computer which communicates with a NXT, a Java library called PC API is available. With the incorporated Java classes it is possible to communicate with any leJOS NXJ program running on a NXT using the LEGO Communications Protocol LCP or Java streams over Bluetooth or USB. An NXT can even be remote controlled from a computer by using this library. Note that on the computer this library is only an add-on to the normal JVM so there are no restrictions in the normal use of Java on a computer.

Some helpful software is provided with which the user can for example flash the firmware to the NXT, upload and download arbitrary files to and from the NXT, remotely browse the files on the NXT or remotely monitor an executed program.

Being a tiny JVM leJOS NXJ is platform independent which means programs can be developed on all platforms, Windows, Linux and Mac OS X. Any Java editor can be used for developing. Preferentially, NetBeans or Eclipse with a special leJOS NXJ plug-in are being used as development environments. leJOS NXJ holds all advantages of implementing in Java. It is currently the only known programming language for NXTs beside RDS (see chapter 3.3.2.2 on page 50) that is object-oriented. It also supports pre-emptive multi-threading, synchronization of threads and thread-safe manipulation of data. This is very useful for managing different behaviours, parallel communication with other robots or data logging. For data logging purposes it is not only possible to log remotely to a computer but also to the local file system. leJOS NXJ provides a basic file system which can be accessed from any running program on the NXT, from the NXT menu system or remotely from a computer. While implementing the programmer certainly notes the limitations of the Java language to a tiny JVM but the restrictions hardly count. The leJOS NXJ JVM provides nearly all standard classes, e.g. arrays, floating-point numbers or Strings, but not Maps or Sets. Sometimes the classes have a reduced functionality which is compensated by an extensive API for robot specific functions. For example a special behaviour model, accurate motor control, access to all LEGO and many third party sensors, Monte Carlo Localization or Kalman filters are included. In addition, copious leJOS NXJ sample are provided by the current release. In spite of the reduced size, the JVM comes with the familiar exception model and garbage collection known from Java.

The familiar exception model raises the problem of displaying thrown exceptions. In the current release of leJOS NXJexception debugging is not very comfortable. On the built-in display only an identifying number of the thrown exception and a number of the method which threw the exception are displayed. The user has to search these numbers in the compile output to get the name of the corresponding exception and method. Additionally, it is not shown where the exception was

exactly thrown, so debugging large method bodies is tedious. But in the new version of leJOS NXJwhich is currently still under development but expected soon, there is a better way of stack tracing which is similar to the normal Java Stacktrace.

Another problem is remote debugging of executing programs. A remote debugging tool that the NXT display and special user-defined log messages in a remote console on a computer does exist. However, debugging via Bluetooth or even USB slows down the execution of the robot's behaviour so much that many problems cannot be reproduced while debugging. It is also not possible to suspend the execution or to watch any variables. It is therefore advisable to write a separate logging module that continuously logs the program either to a file stored on the NXT or to a computer. With this logging data it is easier for the programmer to reconstruct the circumstances of an unexpected termination more easily.

Most of the problems with leJOS NXJ can be prevented by special treatment or are already eliminated in the developer snapshot so that they do not influence the use of leJOS NXJ any further. Therefore, the advantages of this object-oriented firmware by far outweigh its disadvantages. The advantages mentioned above together with the free availability of the sources, a big active community at the official leJOS NXJ forum and even a book fully dedicated to leJOS NXJ written by one of the main developers [BAG07] helped us decide to make use of leJOS NXJ in this master thesis. Since the group "Lehr- und Forschungseinheit Programmierung und Softwaretechnik" also commonly uses Java and many frameworks are written in Java, this adds another advantage to using leJOS NXJ for this master thesis. Furthermore, a challenging outlook is that a community supporting nxtOSEK building on the shoulders of leJOS NXJ is aiming to develop a real-time operating system for NXTs [CHI09].

3.3.2.6 pbLua

The last language introduced is *pbLua* which stands for "programmable brick Lua". pbLua is developed by Ralph Hempel (see his web page at `http://www.hempeldesigngroup.com/lego/pblua`) as a replacement firmware for the NXT based on Lua 5.1[9]. Lua itself is an embedded scripting language with a C-API frequently used in game programming. It is normally not used to write standalone programs, but rather to glue existing high-level components for example written in C together [LAB10]. Ralph Hempel ported this principle to pbLua. A speciality of this language is that source files are not compiled before uploading to the brick. The NXT must be flashed with the pbLua firmware which contains a pbLua interpreter responsible for compiling and executing pbLua programs. Nevertheless, pbLua does not have a big community[10], but can be found in the academic area.

[9] see web page `http://www.lua.org`

[10] see web page `http://pblua.org`

3.4 Interaction between LEGO MINDSTORMS

A new trend in robotics is to make robots collaborate and cooperate to achieve a task by working together. For that, a robot does not only need to reason and act for itself alone in an intelligent way. It rather has to communicate with other robots to discuss and coordinate a common solution to a given problem. On the one hand the robots must exchange messages to coordinate their collective work and on the other hand they have to physically interact to profit from their collective power. In the next two sections, we therefore discuss what types of interactions - physical and virtual - are possible with the platform of LEGO MINDSTORMS.

3.4.1 Physical interaction

Firstly, we have a look at physical interactions between LEGO MINDSTORMS. The easiest solution for physical interaction is to connect two LEGO MINDSTORMS together permanently. In doing this, there is no difference to building a single robot's chassis so that this connection is very simple to be built up. How to coordinate a collective movement of such permanently connected robots is more problematic. The movements of both these robots have to be well coordinated in order to not expose too much mechanical stress to the common chassis. Driving one robot in one direction while the other robot moves in another could result in either damage of the chassis or the wheels or other means of transportation due to attrition and unexpected movements of the connected robots.

Therefore, a better solution is to connect two robots so that each robot has a certain degree of freedom to move in a direction without affecting the other robot in its movement. LEGO provides some hinges to connect LEGO parts with a fixed connection that is nevertheless flexible to some degree. If two robots are connected that way, the angle between the two robots is no longer fixed. The robots may move in different directions to reach a better accuracy in their collective movement. The coordination is not as crucial as in the example before because they have a certain degree of freedom. However, they are permanently connected, so no extra effort must be spent to hold them together.

A third possibility of connecting two robots are on-the-fly connections. That means that the two robots are not attached to each other by their constructor beforehand, but connect themselves during execution time. For that, an elaborate mechanism for detecting the other robot, negotiating about the connection and realizing as well as upholding the connection is needed. To give an example, imagine two robots with one hand each that should take each others hand and move together without loosing the grip. The first problem is to find the other robot and to recognize what part of the robot is the hand. Before a robot should take the other robot's hand, it must negotiate with the other robot how

they are going to connect. For instance, one robot could hold the other one's hand or they could both try to grab. They also have to agree to which level they lift their hands to reach each other, otherwise they will miss the contact. While fulfilling this negotiated connection, the robots need to know whether the connection was effectively established and is uphold. As one can see, there are many considerations to be made for this free connection, but this connection gives more freedom to the possible movements of the robots. Each one can move to its desired direction and if necessary the connection can be released without any damage.

All connections mentioned are direct connections between robots. In addition, two robots can physically interact by interacting with the same object. For example, two robots could push an object too heavy or big for each of them. In doing this, they also have to coordinate their movements to push the object in the desired direction and not to loose contact with the object. Indirect interaction can also be achieved in different ways. First of all, the robots can be permanently connected with the object to push, but this directly refers to the first paragraph of this chapter as they all share a common chassis. Hinged connections between the robots and the object have also to be handled equally to the hinged connection described before. So the third possibility are on-the-fly connections. Here, all kinds of pushing, dragging or lifting are interesting. Again all these interactions do need excellent coordination between robots. Coordinated movement and collective transportation are topics in current research in mobile or rather swarm robotics as the reader can see in chapter 2.2.2.4 on page 22.

3.4.2 Communication

As described in the previous chapter, robots must frequently communicate to coordinate collective solutions. Therefore, it is necessary to have some kind of virtual communication channel between two robots. For this purpose, LEGO MINDSTORMS offer wireless communication via Bluetooth.

With this chip, the NXT is able to communicate with all devices which use the LEGO MINDSTORMS NXT Communication Protocol LCP (details on that protocol can be found in [LEG06a]) and support the Serial Port Profile SPP which is one of the standard Bluetooth profiles. For instance, uploading and downloading files from the NXT to a computer (and vice versa) is very comfortable via Bluetooth because the user does not need to put the robot out of the experimental environment for plugging it to a computer. Other Bluetooth-enabled devices can be connected to the NXT, too. Cell phones, for example, can be used to remote control a robot. For remote control, LEGO provides some direct commands that can be sent to the NXT and are then interpreted and translated to specific functions by the NXT. However, the user does not need to use these commands

because arbitrary content can be sent to the NXT while using the LEGO MINDSTORMS NXT Communication Protocol as mentioned above. Communication is also possible the other way round, from the NXT to a remote device. An application for that could be a camera which the NXT uses to take pictures of its environment whenever it detects something interesting.

In the original LEGO firmware, Bluetooth can only be used in *master-slave-mode*; other firmwares facilitate the Bluetooth communication as it is described below. In master-slave-mode, one device always functions as a master that controls the communication. The master is the only device that is able to send data without any previous request. All slave devices have to wait for a request from the master. That means that the slaves can only communicate to their master and not directly to other slaves. Thereby, the master is able to communicate only to one device at the same time and cannot send several requests to different slaves in parallel. That principle guarantees that all data sent by a slave is read by the master and does not get lost while the master is listening to another slave. However, that causes two problems because the master might not fast enough request for data. Firstly, data may still get lost because the slave tries to hold its messages up-to-date. For example, if the slave informs the master whenever its touch sensor is pressed, but the master does not regularly ask the slave for the current state, a message about a touch contact may be overridden by the next state of the sensor. So the master never comes to know that the touch sensor has actually been pressed. If the slave does not hold its messages up-to-date, this might cause the second problem. The slave stores all messages in a message queue where it always pops the first message whenever the master requests data. If the master requests too slowly, the number of messages increases and might exceed the size of the queue. Additionally, the master always receives the first message in the queue that might be old.

In connections between an NXT and a non-NXT remote device, the NXT always functions as a slave, so the remote device is able to transmit data at its own instigation and does not have to wait for a request from the NXT. Otherwise remote control could be difficult because the remote commands may not reach the NXT in time. Connections between several NXTs have always one master NXT and several slave NXTs. Thereby, an NXT cannot be both, master and slave. Only the master NXT is able to connect to up to three slave NXTs. So the master NXT communicates to each of his three slave NXTs. However, the slave NXTs can only answer a request from this master NXT and are not able to connect to any other device because they cannot function as a master for another connection (as previously mentioned) and they have only one communication channel for sending data to a master device. More details on Bluetooth with the original LEGO firmware can be found in the LEGO MINDSTORMS NXT Bluetooth Developer Kit [LEG06a].

As the master-slave-mode for Bluetooth is relatively uncomfortable, other firmwares like the leJOS NXJ firmware try to lift some of these restrictions. Unfortunately, the properties of Bluetooth communication in leJOS NXJ are not well documented. In the leJOS NXJ firmware the master-slave-mode does not exist. The Bluetooth communication is rather a stream-based communication as known from Java where each device can send data without any previous request. Unfortunately, the leJOS NXJ firmware could not dispose the limit of four communicating NXTs due to the hardware restrictions of only four communication channels per NXT. Additionally, it is possible to wait only for one incoming connection and to connect actively to three other devices by name. Furthermore, LEGO MINDSTORMS programmed with leJOS NXJ have the problem that switching between two different connections needs at least 300 ms what is very long when transferring time dependent data as sensor data.

In swarm robotics where many more robots have to communicate we need the possibility to connect to more than four robots and the exchange rate must be higher than 3-4 messages per second. That is why a message routing protocol was developed in this master thesis that controls the communication between several NXTs via a routing computer. By using a computer as a router, each NXT does only need to connect to the computer (and not directly to another NXT) which saves the time needed to switch active connections. The computer routes any message to the addressed NXT so that the lag of time on the NXT for switching between connections is eliminated (only the connection to the computer is used) and the restrictions of four connections per NXT is lifted. For details how this routing protocol is implemented and should be used, see chapter 4.1.2.2 on page 72.

Overall, Bluetooth adds an interesting communication channel to LEGO MINDSTORMS. As this communication mechanism does not need a line of sight to transmit data, it is a great improvement compared to the old LEGO MINDSTORMS RCX which only offered communication via infra red signals. For Bluetooth communication, the NXT described in chapter 3.3 on page 46 has a built-in CSR BlueCore 4 version 2 chip. It transmits data at 460.8 kBits per second and has only a range of 10 m while more elaborate Bluetooth devices can reach a range up to 100 m. The reduction was due to saving power the Bluetooth chip consumes.

Concluding, the Bluetooth capabilities of the LEGO MINDSTORMS NXT improve the old communication via infra red signals in speed, power consumption, standardization and the need for a line of sight. Of course, it would be appreciated to extend the communication to the faster and long ranging Wi-Fi communication with a speed up to 200 MBits per second. Unfortunately, LEGO decided to use the slower Bluetooth option because it is constructed for the communication between two devices and not in between a whole network and has a power consumption that is way below the one of Wi-Fi. If there is no need to transmit data in real-time, Bluetooth is very interesting for applications that need more

computational power than the built-in processors provide or more space than the built-in flash memory. For instance with Bluetooth, an NXT is able to run processing-intensive code for artificial intelligence on a more powerful processor or to store its perceptions to a computer.

3.5 LEGO MINDSTORMS as a research platform

In this last section about LEGO MINDSTORMS, we want to discuss the advantages and disadvantages LEGO MINDSTORMS hold for the use in scientific research. For that, we compare its possibilities to the *marXbot* constructed by the École Polytechnique Fédérale de Lausanne[BL+10]. At first many restrictions may catch the eye. LEGO MINDSTORMS contain only a 48 MHz main processor that does not have enough power to execute scientific artificial intelligence code in reasonable time. Besides that, the RAM memory is limited to only 64 kByte what permits only restricted computations. The only solution to that problem is to outsource resource-intensive computing to a device with a higher computational power. To overcome this limit, the marXbot comes with a 533 MHz ARM 11 processor that has 128 MByte RAM. The operating system is Linux-based while on LEGO MINDSTORMS only specific firmwares like the original LEGO software or leJOS NXJ are runnable. By the use of Linux nearly all types of applications executable on Linux are usable so the programmer is not restricted to some special API.

A problem that is well known in embedded system is the restricted flash memory. 256 kByte of storage can only hold a small amount of data. Therefore, the engineer has to plan and design the software very accurately while even the implementation must be optimized in time and space. Also logging must be considered carefully not to run out of space. In contrast, the marXbot solves this problem by using an SD memory card with unlimited size of space.

A problem more specifically to the use of LEGO MINDSTORMS is the fixed number of input and output ports. Only three motors can be attached to the NXT which lowers the mobility of a LEGO MINDSTORMS robot. Normally, two motors are needed for driving the robot leaving just one motor to control any attached manipulator as a robotic arm or hand. Similarly, only four sensors can be used with LEGO MINDSTORMS. However, some third party manufacturers provide sensor multiplexers by now which allow to connect several sensors to one port at the NXT. Of course, the marXbot is equipped with many more sensors then LEGO MINDSTORMS. It currently uses up to 100 high-resolution sensors for distance measurement, obstacle detection, pressure or acceleration determination or RFID sensing and especially an omnidirectional camera with 3 megapixels. As LEGO MINDSTORMS are designed to serve as toys and not as scientific robots, the sensors are not high-resolution sensors. But as it can be seen in chapter 3.2

on page 36, the resolution is normally sufficient for prototyping. To study simple problems and first prototypes, LEGO MINDSTORMS provide fair sensing. For more accurate sensing and actuating a different platform must be used.

Additional to that, the use of LEGO MINDSTORMS for swarm robotics is also restricted to simple applications. Communication is only possible via Bluetooth which provides a transmission rate of 460.8 kBits per second and has a range of 10 m. For indoor case studies, the range is normally sufficient enough, but the transmission rate rises a challenge if time dependent sensor data should be transmitted. The marXbot has more elaborate possibilities for wireless connection. In addition to Bluetooth, Wi-Fi is available which offers a higher transmission rate and a wider range if necessary. For direct interactions between several marXbots, it has a built-in attachment module with which two robots can assemble easily and very strongly. They can hold on to each other even if not connected correctly or if one robot loses its grip on the ground due to a hole. Thereby, the connection is nevertheless flexible so the two robots do not have to be aligned in a predefined way and must not hold their first alignment during collective movement. That self-assembly is of course far beyond the possibilities of LEGO MINDSTORMS as described in chapter 3.4.1 on page 54. Sure enough, the marXbot holds many more advantages as listed above if the reader refers to its specification[BL+10], but they are not discussed here in full detail.

On the opposite, fast prototyping is one of the biggest advantages of LEGO MINDSTORMS. Due to the well known LEGO coupling system, robotic prototypes can be built very easily. In most other robotic platforms, a robot is either built beforehand and therefore targeted to a specific application or must be brazed and soldered by the user himself. Once such a robot is built, it cannot easily be altered or extended (for a good overview of programmable robots see the web page `http://www.active-robots.com/products/robots`). LEGO MINDSTORMS in contrast are aiming for a frequent change of construction so that different models can be built. That helps a lot in prototyping as normally it is not known from the beginning which features must be provided or which model does fit the conditions best. The marXbot is a robot that can only be altered by professionals. It is constructed to provide many possibilities to the user by offering a number of sensors, a fast main computing unit and a robust chassis. However, if additional hardware or another model of a robot is needed, the marXbot cannot easily be adapted.

In addition, LEGO MINDSTORMS are not very expensive compared to other robotic platforms aiming to solve different problems. As the marXbot is not purchasable, we compare LEGO MINDSTORMS to the robot "Johnny 5" of Lynxmotion[11] which is a tracked vehicle with two separately controllable arms. This torso and a tri-track chassis is sold for approximately $1000 (corresponding to

[11] see web page `http://www.lynxmotion.com/c-103-johnny-5.aspx`

about 750 EUR) while LEGO MINDSTORMS only cost approximately 300 EUR. Additionally, "Johnny 5" has to be equipped with sensors for perceiving its environment because the base kit does only contain the motors for moving the arms and the tracks. That adds another $15 to $30 per sensor. Of course, some sensors are more accurate than the LEGO MINDSTORMS sensors and a more powerful controller can be purchased, but constructing a robot is not as easy as with LEGO MINDSTORMS since there is no easy coupling system.

Although the LEGO MINDSTORMS sensors may not be extremely accurate, there exists a wide range of various sensors for manifold applications. If any sensor is missing, third party manufacturers provide extension kits to build own well suited sensors. With sensor multiplexers, the number of input ports for sensors can be increased. In theory, each port can handle 128 digital sensors at one time according to [AFF07].

An inexpensive platform does also make an experimental set-up reproducible for other research groups. For reproducibility, LEGO also offers a tool named LEGO DIGITAL DESIGNER [LEG10b] where one can design its own LEGO model and provide step-by-step build instructions to other users (as an example, the robotic models of this master thesis can be found at the appendix chapter B on page 117 or on the enclosed CD). Reproducibility may even extend the research community. This assumption is confirmed by the big scientific community that is already using LEGO MINDSTORMS. They are distributed into many different communities each using a different programming interface as described in the sections above. These programming languages are mainly high-level languages while other robotic platforms must often be implemented in assembler. However, due to many active communities the development of these high-level languages advances more scientific topics as localization or special filters.

Chapter 4

Swarm robotics with Lego Mindstorms

After introducing Lego Mindstorms and their components, we want to discuss the use of them for swarm robotics. While other research groups as the group around Prof. Dr. Dorgio in Brussels[1] are experimenting with custom-build robots for example in the EU-project Swarm-bots[2] (lasting from October 2001 to March 2005) or the EU-project Swarmanoid[3] (lasting from October 2006 to September 2010), we want to explore whether Lego Mindstorms are suitable enough to serve as prototypes for swarm robotics with intelligent agents. In the previous chapter, the technical details of Lego Mindstorms have been presented. Now, we want to put them into action and test their performance in real life. Thereby, the capacity of one operating Lego Mindstorms robot as well as the cooperation of several Lego Mindstorms are fields of interest. This chapter should reveal the limits of the hardware in real-life operation as well as the possibilities and performance of software for Lego Mindstorms. For this master thesis, there are five Lego Mindstorms Nxt 2.0 Kits available with rechargeable Lego batteries for each and four additional compass sensors. For programming the Nxts, the language leJOS NXJ has been used due to the advantages mentioned in chapter 3.3.2.5 on page 51.

4.1 Architecture

Before implementing some exemplary applications for Lego Mindstorms, it is advisable to think of a software architecture all applications can build upon. Thereby, it must be thought of programming a single robot and its probably many

[1] see web page `http://www.ulb.ac.be/rech/inventaire/unites/ULB675.html`
[2] see web page `http://www.swarm-bots.org`
[3] see web page `http://www.swarmanoid.org`

alternating behaviours as well as giving a guideline how to control a network of robots and their communication. The next two sections at first suggest an architecture for a single robot's behaviour and then demonstrate two possibilities of communication between LEGO MINDSTORMS.

4.1.1 Single NXT architecture

We want to introduce an architecture for a single robot that is able to sense its environment, reason about its perceptions and actions and react upon its environment. In addition, it has the capability to communicate with other robots or computers. Before the actual architecture for such a robot is discussed and an implementation is suggested, a special model for implementing a robot's overall behaviour is presented, the *behaviour model*.

4.1.1.1 Behaviour model

Depending on its perceptions, a robot normally executes various reactions. At first, a programmer might be tempted to implement all reactions in one program flow that picks the right reaction after a long series of conditional statements. This programming style results in long flows that become confusing very fast so that it is often unclear which reaction occurs due to which condition. Therefore, in robotics another approach should be used.

The main principle of *behaviour-based programming* is to identify closed units of reactions that have to be executed under designated conditions (see chapter 2.1.4.4 on page 16 for a detailed explanation and literature on behaviour programming). The overall behaviour of a robot is broken down into such units where each unit describes one atomic *behaviour* of the robot. A behaviour consists of three components: when to come into action, how to react when running and what to do if stopped. As a running example, we explain the breakdown of the overall behaviour of a simple line-following robot which recognizes a black line on a white surface. The overall behaviour of this robot is to follow the black line and if the line gets lost, to search for the line. Therefore two atomic behaviours can easily be derived. One behaviour, we call it *drive straight*, comes into action if the ground underneath the robot is black and simply drives straight ahead. If this behaviour is stopped, the robot stops its driving. If the ground underneath becomes white, the second behaviour, the *find line* behaviour, takes action. The robot searches for the line until the ground underneath is black. Searching for the line is a more complex behaviour that consists of rotating up to 360 degrees, travelling some distance to the front and starting the rotation again. This behaviour is stopped as soon as the ground underneath becomes black.

In addition to the atomic behaviours, we also need a controlling instance that monitors the execution of the behaviours. On the one hand, this instance is responsible for the execution of the current behaviour, but on the other hand it has to check regularly whether the current behaviour must be stopped and another behaviour should be executed. In the line-following example, this instance has to evaluate whether the ground underneath is black or white and according to this, choose the *drive straight* or *find line* behaviour.

There are two things the programmer must think of while defining the atomic behaviours: what to do if no behaviour or two behaviours in parallel are ready to run. Normally, a robot should have a standard behaviour that describes its normal behaviour. In many cases, it is meaningless to have a mobile robot stay motionless and wait for changes in the environment. More often, the robot moves around to meet some special conditions and then to fulfil a special task. In our example, this problem that no behaviour is runnable occurs if the ground underneath is neither black nor white. It would be useless to stop the robot moving under this condition because the colour of the ground will not change further and so the robot will not follow the line. In this case, the drive straight behaviour becomes the normal or basic behaviour of the robot. That means the condition under which this behaviour comes into action is altered so that it is always runnable. But then, a second problem is met. If the line is white, two behaviours are executable. While using a standard behaviour, this is a common problem. However, this can be solved intuitively by introducing priorities for each behaviour. A standard behaviour gets a low priority while a special task is uprated. With the aid of priorities, the controlling instance is now able to choose the highest ranking behaviour that is executable. In the line-following example, we rank the special task of finding the line higher than driving straight so that the drive straight behaviour is constantly executed until the ground underneath becomes white and the higher ranking behaviour of finding the line comes into action.

In conclusion, in mobile robotics it is recommendable to make use of the currently described behaviour model. It achieves a clear separation of behaviours and a considerable structure for executing behaviours. Furthermore, it is very easy to add a special task to the overall behaviour of a robot by extending the behaviour pool of the robot with this new behaviour.[4]

4.1.1.2 Four layered architecture

To bring the control of sensors and actuators and the behaviour model together, we introduce a four layered architecture as seen in figure 4.1.

[4]Some practical advice on using the behaviour model can be found in the leJOS NXJ main tutorial at `http://lejos.sourceforge.net/nxt/nxj/tutorial/Behaviors/BehaviorProgramming.htm`.

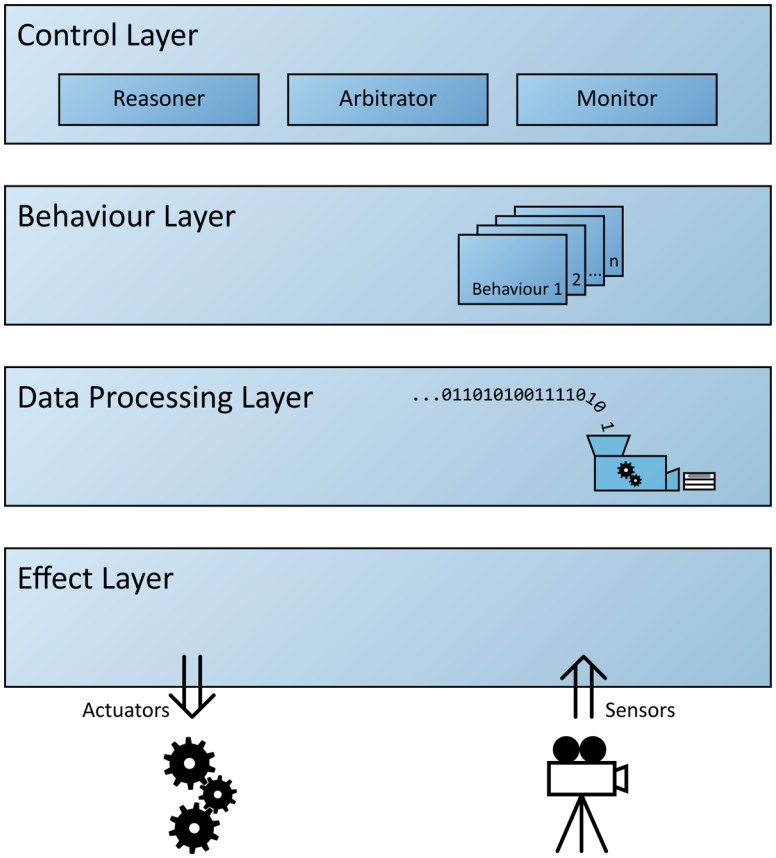

Figure 4.1
A four layered architecture for the implementation of a single robot.

One control cycle of the robot starts at the bottom of these four layers, run up to the topmost layer and give some controlling commands back to the bottom. So we start with the lowermost layer called *Effect Layer*. To interact with its environment a robot has various sensors with which it can perceive its environment and some actuators – normally motors, but also lamps or loudspeakers – with which it can react to different environmental conditions. The goal of the Effect Layer is to control these sensors and motors. On the one hand, the Effect Layer takes care that the sensors record the desired data whenever necessary. On the other hand it activates the actuators whenever a movement or reaction is needed.

Often, the Effect Layer receives only raw data from the sensors and can only turn the actuators on or off. So a second layer is needed to translate the raw data to more readable and meaningful information and vice versa. This layer is called *Data Processing Layer*. This layer is for example responsible that the light intensities read by a colour sensor is translated to the corresponding value in a colour space like RGB. Otherwise the Data Processing Layer takes also the command to drive the robot at a circle with a specific radius and steers the different wheels at the corresponding speed to gain this circle.

The next important layer is the *Behaviour Layer* which brings perception and reaction together. In this layer, it is defined what kind of effects should be caused by specific perceptions. At first, each behaviour determines whether it is runnable due to the perceptions from the Data Processing Layer. Then it finds the right reaction to the perceptions if its execution conditions are met. For example, a start-and-stop behaviour might take different colours as an input. According to the perceived colour, it might send the command to move or to stop the actuators. Another behaviour might turn the robot to the left or the right depending on the colour.

As a robot may also have various behaviours that come to action in different situations, there must be another layer to control the switch between the behaviours as mentioned in the previous section. This layer is called *Control Layer*. Three components form the main tasks of this layer. Firstly, it must be monitored which behaviour is ready to run or to react to the current perception. This is carried out by the *Monitor* component. The *Arbitrator* component controls the execution of the runnable behaviour with the highest priority. The last component, the *Reasoner*, gives the robot the capability of reasoning and planning. All layers and components except the Reasoner model a simple reactive robot that only takes a perception and reacts according to some predefined rules to this perception. With the Reasoner, however, the robot gains a "brain" that is able to learn how to react best. This component can be implemented to incorporate all state of the art techniques of intelligent agent for reasoning, planning or learning.

After explaining each layer separately, we give now an insight how these layers practically work together. Again, we discuss the line-following robot introduced in the previous chapter. To determine the brightness of the underlying surface, this robot has a colour sensor. The Effect Layer at first takes care that this sensor regularly measures the light intensity of the underlying surface. The received raw data is prepared by the Data Processing Layer to a value according to the grey scale. This value is then used to determine which behaviour of the line-following robot must be executed. For example, the behaviour to search for the line – find line – comes to action if the underlying surface is white. However, the line-following behaviour – drive straight – is always ready to run as we defined this behaviour as the standard behaviour for this robot. For the moment, we assume that the underlying surface is white so that both behaviours are runnable. This information is transmitted from the Behaviour Layer to the Control Layer where the Monitor registers each runnable behaviour according to its priority for execution at the Arbitrator. At this step, the Reasoner may interfere to change the ordering of the registered behaviours. When receiving an ordering of executable behaviours, the Arbitrator starts the first behaviour. As the drive straight behaviour is only the standard behaviour and the find line behaviour has a higher priority, the find line behaviour is executed now. The behaviour computes its reaction according to the perception. For the line finding behaviour, this could

mean to rotate ten degrees to the left. At this step, the Reasoner may again interfere to alter the decision of the behaviour with regard to some knowledge gained previously. For example, the Reasoner would not suggest to turn to the left but to the right because in all previous runs the line was found on the right of the robot. The rotation command is transmitted from the Behaviour Layer to the Data Processing Layer where the right turn of ten degrees is translated to the corresponding backward rotation of the right wheel and forward rotation of the left wheel. This information is forwarded to the Effect Layer which simply activates the motors of the two wheels.

4.1.1.3 Implementation

As this four layered architecture is very high-level we have a closer look at how this is realized in this thesis for LEGO MINDSTORMS. Figure 4.2 gives an overview of the implementation.

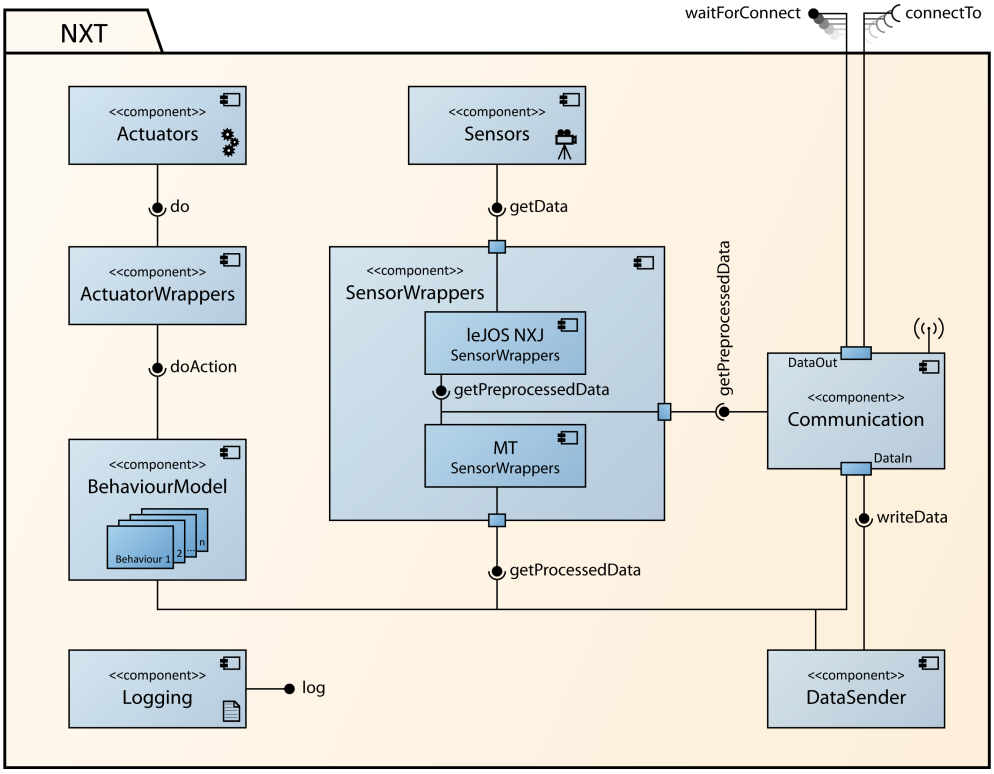

Figure 4.2

The component diagram for an implementation of the four layered architecture.

This component diagram exactly follows the architecture just described, but extends it by some additional modules. At first, let us consider the similarities. The Effect Layer is realized by the components *Actuators* and *Sensors* which directly control the underlying hardware. In the components *ActuatorWrappers*

and *SensorWrappers*, the high-level conversion of the raw data takes place as a task of the Data Processing Layer. The reaction to the perception is computed in the *BehaviourModel* which combines all tasks of the Behaviour and Control Layer.

However, the architecture of figure 4.2 does not directly support cooperation between several robots. For that purpose, the *Communication* component was added which fits in the presented four layered architecture. Until now, the Data Processing Layer received its sensor data only via the Effect Layer from its own built-in sensors. For a swarm of robots, it might also be interesting what kind of perceptions other robots take. Therefore the *SensorWrappers* component should not only represent the own built-in sensors, but also the sensors of other robots (we call them *remote sensors*). Consequently, the *SensorWrappers* component might not only get raw data from its own sensors, but rather receive processed data from the remote sensors via the *Communication* component. For that, the other robot must send this processed sensor data to the asking robot. This is done by the *Communication* component. If there is no time for polling the data, the remote robot can also send its sensor data actively via the *DataSender* component. However, the *DataSender* and *Communication* components can not only be used to transmit or receive sensor data to or from a remote device, but also to send arbitrary data such as files or log messages. The last component, the *Logging* component, is a vertical function to all other components. It is used to log the current program execution for further analysis.

For a better understanding of this component diagram, the modules and their interconnections are now described in more detail.

Sensors component Firstly, several classes is needed to address and control the sensors as said in the Effect Layer of the four layered architecture in figure 4.1. In this master thesis, the framework of leJOS NXJ is used which provides all the necessary classes packed in the package `lejos.nxt`. leJOS NXJ takes care of activating the right sensors and reading the correct pins in order to receive the desired values. For requesting the raw data from a sensor, leJOS NXJ provides the `getData`-interface. Only a status code whether the request was successful or not and the read data in bytes are returned.

SensorWrappers component To convert the raw data to a readable value, the *SensorWrappers* component which refers to the Data Processing Layer comes into action. This component is divided into a subcomponent called *leJOS NXJ SensorWrappers* which is provided by leJOS NXJ in the package `lejos.nxt` and a subcomponent called *MT SensorWrappers* which was implemented in this master thesis. The leJOS NXJ SensorWrappers subcomponent handles raw sensor data and requires therefore the `getData`-interface that is currently provided by the Sensors component. It converts the raw data to some high-level information. For

example, this subcomponent computes the colour of the underlying surface given the stream of bytes provided by a colour sensor. This high-level information is then provided by the `getPreprocessedData`-interface. The MT SensorWrappers subcomponent requires the `getPreprocessedData`-interface to filter noise reads or for example to convert a colour value to a value according to the grey scale. This abstraction is provided to other components by the `getProcessedData`-interface afterwards.

An interesting feature of the *SensorWrappers* component is that this component does not only wrap its own built-in sensors, but also remote sensors. Therefore, the `getPreprocessedData`-interface might not only be provided by the leJOS NXJ *SensorWrappers* subcomponent itself. The preprocessed data can also be received from external components which provide the data of external sensors. This brings the advantage that its own as well as the remote sensor data is handled equally. It is also thinkable to use this interface to mock a sensor if an environment should be simulated.

DataSender component The *DataSender* component consumes the processed sensor data. The function of this module is only to pass any data to external resources – this may be other robots or a supporting computer. For that, it requires on the one hand the processed data to transmit and on the other hand a communication channel to write the data to. The data is easily provided by the currently described `getProcessedData`-interface of the *SensorWrappers* component. The transmission to the addressed device is handled by a `writeData`-interface of the Communication component which is described in the next paragraph.

Communication component The *Communication* module provides the remote communication. To communicate with remote devices, this module has the `DataOut`-port. This port is able to actively connect to other external resources by its `connectTo`-interface. Beside this active connection, it is also possible to wait for an incoming connection by the `waitForConnect`-interface. However, this interface should be used with caution because any remote device might connect to this open communication channel or if no remote device connects, this channel may be open for a long time. Depending on the implementation of the *Communication* component, an NXT can connect to up to three remote devices by name and wait for one incoming connection or it can connect actively or passively to any number of external devices. For details on that see chapter 4.1.2 on page 70.

The *Communication* module provides useful interfaces that can be used once a connection to a remote device is established. The first is the `getPreprocessedData`-interface mentioned above in the paragraph about the *SensorWrappers* component. This interface supplies for example the *SensorWrappers* component with preprocessed data received from remote devices. This data can derive from other robots' sensors, from a computer or even from a simulation environment which

mocks environment conditions. With this interface, the NXT gains the advantage to rely in its behavioural decisions not only on its own sensor data, but also on foreign perceptions.

To receive sensor data from a remote sensor, there must also be a possibility to send sensor data from one NXT to another. The *Communication* module is able to handle this transmission. The data to transmit must either actively or passively arrive at the `DataIn`-port of the Communication component. In this case actively means that the *Communication* component asks for data to send by its `getProcessedData`-interface. At the moment, this is satisfied by the corresponding provided interface of the *SensorWrappers* component. Otherwise the `DataIn`-port does also provide a `writeData`-interface where other components can submit their data to be sent as seen with the *DataSender* module.

BehaviourModel component The main component of this architecture is the *BehaviourModel* component. It is split into two parts like the *SensorWrappers* component. leJOS NXJ offers a number of classes for monitoring and executing behaviours as described in chapter 4.1.1.1 on page 62. In this master thesis, several exemplary behaviours have been added to this component. These exemplary behaviours require sensor data input. For that, the *BehaviourModel* component consumes the provided `getProcessedData`-interface of the *SensorWrappers* module. After computing the actual reaction to this perception in the currently executing behaviour, the *BehaviourModel* component requires another interface to act upon the robot's environment. Via the `doAction`-interface, the *BehaviourModel* component is able to control its actuator at a high level. For example, it might give the command to drive in a circle or to switch on white light to this interface. Only the Reasoner is currently missing which means that the behaviours can only be executed as implemented. It is not possible to let the robot learn its behaviour or to reason about its decisions and reactions. Nevertheless, as the *BehaviourModel* is well split into a monitoring and execution part and the behaviours itself, it is easy to add a reasoning instance to match the architecture.

ActuatorWrappers component Because the actuators can normally only be turned on or off or be regulated in their speed, a component is needed that translates high-level commands into the correct set of these low-level commands. The framework of leJOS NXJ also supplies this conversion in analogy to the sensors in the *SensorWrappers* component. In our component diagram, we model this component provided by leJOS NXJ in the package `robotics.navigation` as the *ActuatorWrappers* component. This module at first receives high-level commands via its provided `doAction`-interface. For the example mentioned above, the command to drive in a circle is then translated into low-level commands where one wheel is rotated faster than the other wheel. Or switching on white light results in turning on all available lamps. These potentially multiple commands are given to the required `do`-interface which represents the low-level interface to the actuators.

Actuators component Lastly, the *Actuators* component which is also part of the leJOS NXJ framework in the package `robotics.navigation` is the controller of all actuators. Any command how to control the actuators is given to this module via the provided `do`-interface. The appropriate low-level command is then given directly to the actuators for execution. This component refers therefore to the Effect Layer as seen in figure 4.1.

Logging component As a vertical layer, the *Logging* component is included in this component diagram. Because this module is used in all other described modules, only its provided `log`-interface is shown. With this interface any message can be logged during execution. The *Logging* module stores the messages together with a time stamp either in a local file on the NXT or transfers it to a remote device via Bluetooth.

4.1.2 Communication between LEGO MINDSTORMS

With the architecture described above, a single robot that is sensing and acting upon its environment can be implemented. The architecture also incorporates a module to communicate with other devices. This communication is realized by Bluetooth as LEGO MINDSTORMS are already offering this communication capability. Since the original Bluetooth communication is very restricted (see chapter 3.4.2 on page 55) and leJOS NXJ could not overcome all limitations (see chapter 3.3.2.5 on page 51), this master thesis has a close look at the possibilities of communication between several NXTs when using leJOS NXJ. In the first section, we discuss peer-to-peer (P2P) communication using the leJOS NXJ firmware. In contrast to that, the second section covers virtual peer-to-peer communication via a routing computer so that direct connections between NXTs are no longer needed. Lastly, we want to highlight the model for data exchange between two peers developed in this master thesis as well as its implementation.

4.1.2.1 P2P communication

The most convenient way of exchanging data between two peers – may it be only NXTs or an NXT and a computer – is the direct connection between the communicating devices. This situation is shown in figure 4.3.

This figure shows several LEGO MINDSTORMS robots illustrated by LEGO MINDSTORMS NXTs and a different external device like a computer. Using the leJOS NXJ library, each NXT can only connect to three other devices by name and can wait for one incoming connection. While connecting to a computer, the NXT must always wait for a connection request from the computer. In figure 4.3, the outgoing requests are pictured by required interfaces while the incoming connection is depicted by a provided interface (or lollipop).

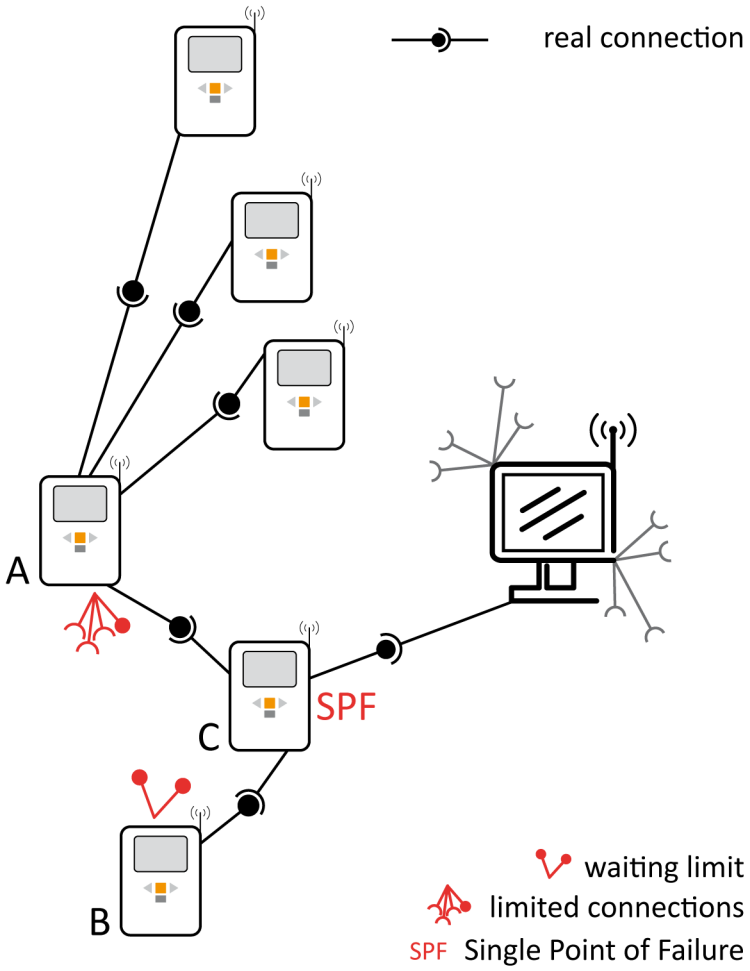

Figure 4.3
Peer-to-peer communication using leJOS NXJ

With this network architecture, several problems come to mind. The main restriction of the communication interface is that an NXT can only wait for one incoming connection and connect to three devices by name. If all NXTs in the network connect to a computer (for example for logging purposes), they have to use the incoming channel because leJOS NXJ does only support connections from a computer to the NXT and not the other way round. Consequently, other connections in the network are not possible any more because no NXT can wait for an incoming request from another NXT any more. Logging to a local file and therefore eliminating the computer connection by that only shifts this problem. As seen in figure 4.3, it is now possible to connect several NXTs, but the NXT A cannot connect to another device any more because the NXT A does not have any free outgoing or incoming connections. We call this the *limited connections problem.*

The problem of just one incoming connection itself is referred to by *waiting limit*. Due to this waiting limit, it is not possible to wait constantly for incoming connections. As soon as one device has connected to the open port for incoming connections, the NXT cannot wait for other connections any more. This can be seen at NXT B in figure 4.3 However, it would be quite useful to continuously wait for incoming requests as the NXT itself may not know which other devices are interested in its data. In addition, if we want to introduce a connection to a computer, the connected NXT cannot communicate to another NXT any more as described before.

Another obvious problem is that for example NXT C exhibits a *single point of failure SPF*. This device is the only device forming and holding the network between NXT A, B and the computer. An error or crash of this bonding device will destroy this subnetwork as no other device in the network is able to connect all three devices.

A last problem derives from the firmware regardless of any provider. The switch between different connections on an NXT requires at minimum 200 to 300 ms. This *switch delay* is due to the clean-up of message buffers for the communication on the chip and cannot be reduced. Unfortunately, this timespan prohibits a frequent sending of time dependent data to multiple devices.

4.1.2.2 Virtual P2P communication

To overcome the four problems mentioned in the previous section, another network architecture was analysed in this master thesis. All NXTs have been connected to a routing computer that simply forwards all messages to the correct NXT. An overview can be seen in figure 4.4.

In this network, all NXTs use their incoming connection to wait for a connection from a computer. This computer functions as the main router of all messages in this network. Since it connects to all NXTs by name, it knows each partner by his unique name and can thereby route any data to this recepient. Via the routing computer, each NXT is able to connect virtually to other NXTs. The NXT can either wait for a specific connection from an NXT or connect actively to another NXT by name. The NXTs do not communicate directly but via the routing computer which controls the connections and the data exchange protocol (more details on this protocol are given in the next section). Beside NXT to NXT communication, an NXT is also able to communicate with the routing computer itself. For that, the NXT also has to initialize a virtual connection to the computer using the real connection to the computer only as a routing gateway.

Although this virtual P2P communication causes more traffic in the network, it solves the four problems mentioned before. The limited connections problem does not occur any more. As each NXT can now wait passively or connect actively to an arbitrary number of other devices, NXT A can now connect to NXT B.

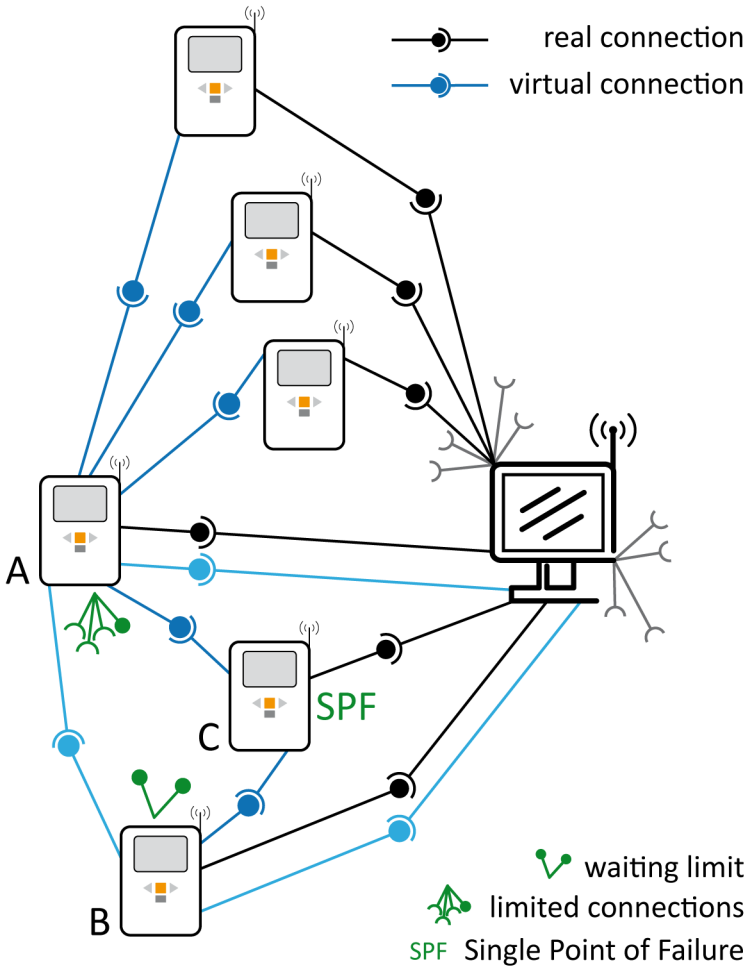

Figure 4.4
Virtual peer-to-peer communication using leJOS NXJ.
Black connections depict real connections between NXT *and computer while blue connections symbolize possible virtual connections realized by routing via the computer.*

In the same manner the waiting limit is lifted so that NXT B is now able to wait for a connection from NXT A in addition to the connection to NXT C. While an NXT can only wait for one real connection, it is now able to wait virtually for any number of incoming connections. The only real incoming connection is used for connecting to the computer while this connection is then used to forward any previous connection requests.

The NXT C also is not a single point of failure any more since NXT A, B and the computer can now directly connect. However in figure 4.4, it still seems that the routing computer is a single point of failure because this is the only device that is really connected to the NXTs and routes all messages in this network. But of course, for this routing computer all means of mirroring, crash detection and prevention can be applied.

Unfortunately, this virtual P2P communication cannot completely overcome the transmission delay. The NXTs do not need to switch between different real connections any more so that the switch delay is fully eliminated. But on the other hand, they always transmit their data to the computer which forwards the data to the addressed receiver. The initialization and maintenance of such a virtual network needs additional data exchange that causes a transmission delay: a connection request has to be sent to the computer which forwards the request to the addressed device, and the response takes two additional steps. For exchanging data between two virtually connected devices additional data has to be sent that describes from which device to which other device the routing computer should forward the message. That means that compared to the direct P2P communication all messages need two steps to arrive at the destined location, all messages are sent twice due to these two steps and additional routing information is needed. This increases the transmission time between two devices to approximately 100 ms so that although the switch delay of 200 to 300 ms is fully eliminated, this protocol causes some additional delay.

Depending on the application, different communication styles are appropriate: if the NXTs do not often switch between connections and do not need many connections to other NXTs, the direct P2P communication is more efficient because there is no transmission delay due to routing. Otherwise the virtual P2P communication provides faster transmission.

4.1.2.3 Communication model

In this master thesis, both types of communication networks have been implemented. They both are based on the communication model shown in figure 4.5.

As it can be seen, this communication model is structured similar to the ISO-OSI communication model. Each layer has its specific responsibilities for transmitting the actual message (these specifications have to be realized by any implementing communication protocol while the concrete implementation is not given by this communication model). For sending a message between two devices A and B, all three layers shown have to be traversed; first for packing a message on device A, then for unpacking the received message on device B.

The bottom layer, the *Physical Layer*, is responsible for administration of the physical connection between two devices. It takes care of correct initialization while it also closes a physical connection correctly. Compared to the two types of network described in the previous section, this layer manages all connections that are established directly between two devices.

The middle layer, the *Routing Layer*, administrates virtual connections. Therefore, it builds upon the Physical Layer and sends management data over the physical connection. It manages the correct initialization of a virtual connection

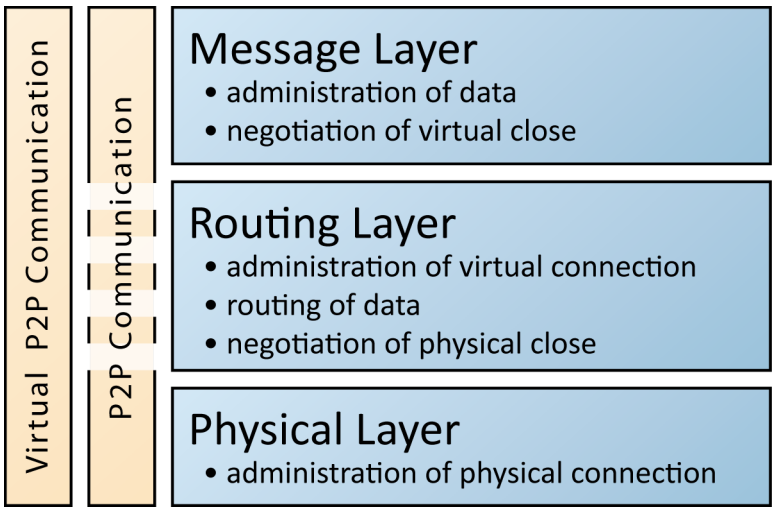

Figure 4.5
The communication model for data exchange between several NXT*s following the ISO-OSI communication model*

as well as its complete clean-up. This layer is also responsible for the correct routing of the actual message. On the one hand, it has to take care of adding the right routing data to the original message, such as the name of the sender and the name of the receiver. On the other hand, it reads this routing information in order to forward the data to the correct device or read the message itself if it is directed to this device. In addition to that, the Routing Layer negotiates the closing of the underlying physical connection. As the Physical Layer is only responsible for the correct clean-up – for example that all buffers are emptied and the connection streams are closed –, the Routing Layer manages the negotiation whether the physical connection has to be closed and whether both sides agree on that decision. In contrast to that, the Routing Layer does not negotiate about the initialization of a physical connection as there is no communication channel through which the negotiation could take place. This can only be done in the Physical Layer. However, it is important to have the negotiation about closing in this Routing Layer because only this layer is able to decide whether the physical connection is still needed for any virtual connection due to its task of administrating the virtual connection.

Because the tasks of the Routing Layer only concern the virtual connection except the negotiation of correctly closing the physical connection, this layer is not necessarily needed in the normal P2P communication (the box for the normal P2P communication is not solid in its middle to depict this in figure 4.5). While communicating directly without a router, there is no need for an additional virtual connection as all the data can be sent directly over the physical connection. In direct connections, routing information is not necessary either since the message always reaches the desired location immediately. The only task left is the

negotiation of closing, but this task is given to the topmost layer in concern of P2P communication so that the Routing Layer can then be fully omitted for the normal P2P communication.

The topmost layer, the *Message Layer*, is the only layer which is actually able to interpret the sent message. Therefore, its most important responsibility is the packing and unpacking of data. It has to distinguish the different message types, to evaluate the message correctly and to forward the information to the correct application or internal receiver or to answer it correctly. This layer must be able to read and answer requests for certain data such as light intensity values, compass data or specific files. On the other hand, it must also be able to interpret a received message or response and pack data, i.e. light intensity, compass data or specific files. In reading or sending the actual message, it is not important for this layer whether the message was transmitted via a real or virtual connection. The other layers take care that the communication channel stays transparent for the Message Layer. In addition to the interpretation of the actual message, the Message Layer also has to negotiate the correct closing of a virtual connection in analogy to the Routing Layer. Here, the Message Layer (and not the Routing Layer) has to negotiate the closing of the virtual connection because it knows whether there is any data left to be transmitted. Otherwise, only the Routing Layer is able to initialize a virtual connection as in the Message Layer no communication channel is available beforehand through which to negotiate the initialization of a virtual connection. Since for direct connections there is no need for an additional virtual connection, in a direct P2P communication network the Message Layer can negotiate the closing of the physical connection instead of the virtual connection. In this case, the Routing Layer can be omitted as described before.

4.1.2.4 Communication protocol

The currently described communication model has been implemented for this master thesis. The next section explains how the model is currently translated into a communication protocol. For illustrating the communication protocol, we use figure 4.6.

In this figure, the transmission of a message from device `A` to device `B` is shown while the device `Router` functions as a routing instance for this message forwarding it from device `A` to device `B`. At the moment, only uni-cast is supported in this communication protocol so that it is only possible to send a message to one device. However, the protocol can easily be extended to implement multi-cast, too. The possibility to send the same message to several devices simultaneously reduces the traffic in a P2P network since only one message has to be sent to the routing instance, but several devices will receive the message from the routing instance. In the detailed explanation of the protocol, we will highlight what has to be changed to support multi-cast.

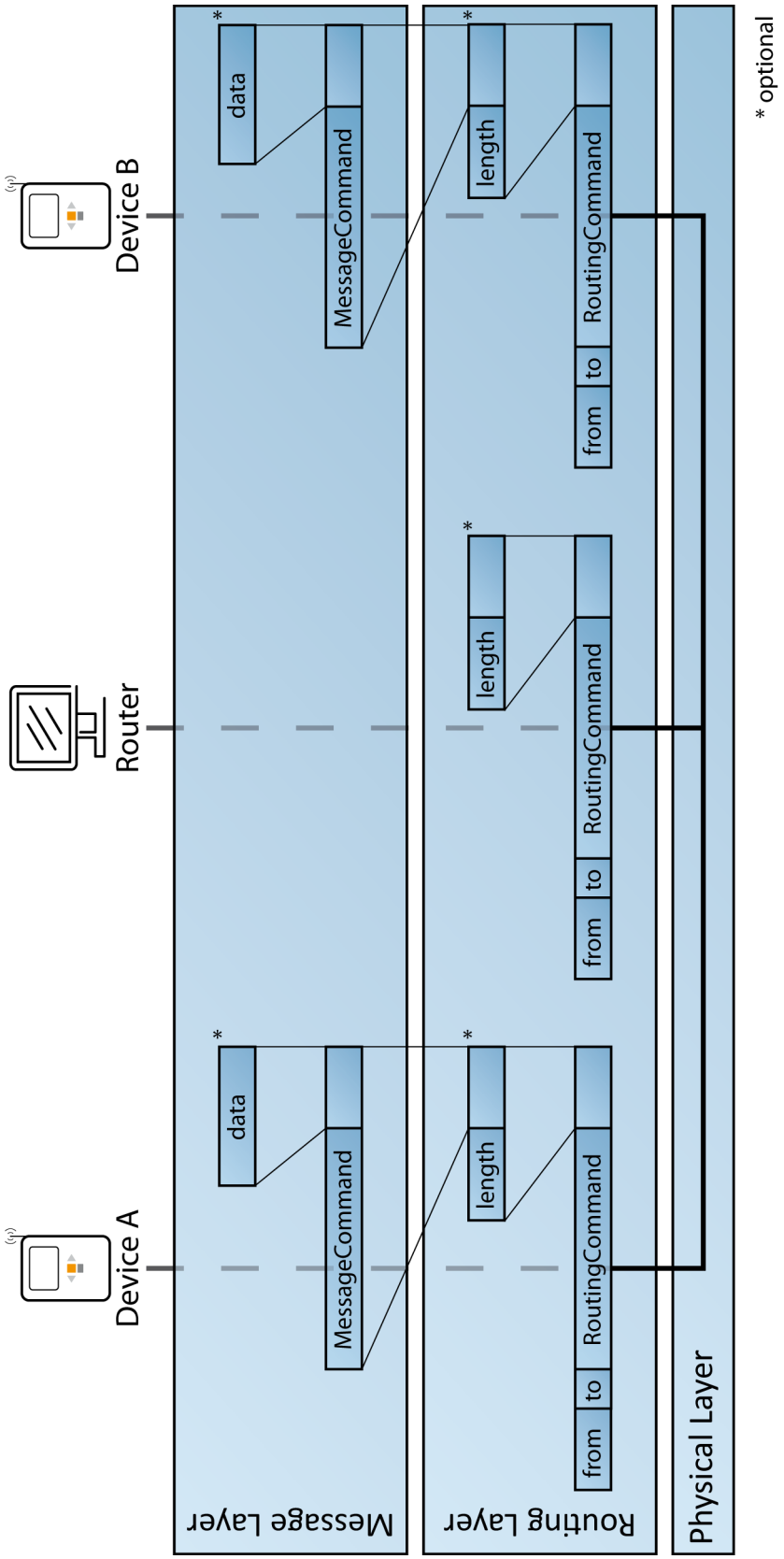

Figure 4.6

A communication protocol implementing the communication model shown in figure 4.5.

In figure 4.6, the three layers of the underlying communication model are depicted as horizontal boxes. At first, each layer and its headers are described. The detailed explanation will be followed by a table summarizing all headers and fields used in this communication protocol. Lastly, an overview is given how a message is routed through this protocol.

We start our explanation at the topmost layer because the original message is packaged and unpacked in this Message Layer. There are three different types of messages to be distinguished in this layer. The first are commands concerning the negotiation of closing the virtual connection, the second requests for specific data and the third responds or sends without any previous request. This information is enclosed in a particular header called *MessageCommand* that prefixes the actual message. The actual message called *data* in figure 4.6 may either be empty or contain an arbitrary number of bytes.

In the *MessageCommand*, the current *type of the message* – namely `COMMAND`, `REQUEST` or `SEND` according to the previous description – and the *type of the content* is stored. For the message type `COMMAND`, there are three different content types: `CLOSE_VIRTUAL` for requesting to close the virtual connection, `CLOSE_VIRTUAL_ACK` for acknowledging a close request for the virtual connection or `CLOSE_VIRTUAL_DECLINE` for declining a close request. The actual message after the *MessageCommand* `COMMAND` is always empty because no additional data must be sent to negotiate about closing a connection except the three given content types (a sequence diagram explaining the actual close protocol is given in the appendix A.2 on page 112).

The second message type `REQUEST` can be combined with three content types so far: `LIGHT` for requesting the current light intensity read by a remote sensor, `DEGREE` for getting the current compass data of a remote sensor or `FILE` for retrieving a specific file from the remote device. Likewise for the negotiation of closing, no additional data has to be sent for the content types `LIGHT` and `DEGREE` because they simply ask for the corresponding sensor data. In contrast to that, while sending a request for a file the *data* field cannot be empty. The header with content type `FILE` must always be followed by data that describes the name of the file to retrieve. Here, it must be ensured that a file name with arbitrary length can be understood by the receiving device. Requests for other data can simply be added by introducing a new content type. The *data* field can be used to transmit any additional data that is necessary to fully specify the desired data. For example, to request the current distance of the remote device to any object in front of it the content type `RANGE` could be used while no additional data has to be sent.

The last message type `SEND` is similarly structured to the content type `REQUEST`: `LIGHT` specifies the transmission of a light intensity value, `DEGREE` the transmission of compass data and `FILE_EMPTY`, `FILE_WHOLE` as well as `FILE_PART` the transmission of a file. After the header with content type `LIGHT` or `DEGREE`, the *data* part contains

the corresponding numerical value. While sending a file, we chose to distinguish between the upper three content types mentioned above. After each content type depicting the transmission of a file, the name of the file follows except for `EMPTY_FILE`. Afterwards, the *data* part is empty for the content type `FILE_EMPTY` meaning that the requested file was empty or does not exist on the remote device (for future versions, it would be useful to distinguish between files that are empty or not available). In contrast, the content type `FILE_WHOLE` transmits a whole file while the content type `FILE_PART` sends the requested file in parts because the file is too large to be sent as a whole. Similarly to the request of further data, a new content type of the message type `SEND` can be introduced to signalize the sending of other data. For example, after the new content type `RANGE` the field *data* contains a numerical value corresponding to the current distance of the remote device to any object in front of it. The protocol for requesting or sending data can be seen in appendix A.2 on page 112.

Concluding, in the Message Layer the different types of messages for negotiation of closing, requesting or sending data are distinguished by the *MessageCommand* that is enhanced by the content type according to the current information to be transmitted. Depending on the message and content type, data in arbitrary length is sent afterwards or it is left blank.

After the data has been packaged so that the receiver can easily determine how to interpret the message, routing as well as administration information has to be added. According to the communication model of chapter 4.1.2.3 on page 74, the underlying Routing Layer takes care of the administration of the virtual connection, of the routing the current message and the negotiation of closing the physical connection. To distinguish these three responsibilities, the Routing Layer adds an additional header called *RoutingCommand* to the packed message. Another enclosing header contains the routing information *from* which device *to* which device this message is directed. If multi-cast should be supported, we only have to extend the single-valued field *to* to a multi-valued field so that a message can be addressed to several devices. The routing device reads the entire field *to* and forwards the message to every device mentioned in the field *to* seperately.

While the routing information must always be present and can only differ in the particular names, the *RoutingCommand* distinguishes three *routing types*: `VIRTUAL_ADMIN` for any command concerning the administration of the virtual connection, `DATA` for signalizing that the following message contains data to be interpreted in the Message Layer or `COMMAND` for negotiating to close the underlying physical connection. As the administration of the virtual connection or the negotiation of closing do not need any enclosed message, the routing type `DATA` is the only case when data emitting from the Message Layer is transmitted, too. This data can be any type of message described in the previous paragraph about the Message Layer (for a detailed description of the data transmission protocol see the appendix A.1 on page 107).

In both other cases, no data is sent after the *RoutingCommand*. In the case of VIRTUAL_ADMIN, the connection initialization is negotiated with the *administration types* CONNECT_REQUEST, CONNECT_ACK or CONNECT_DECLINE. Otherwise closing a virtual connection takes place without any further communication to the remote device as the negotiation of closing takes place in the Message Layer as described in the previous paragraph. For the administration of virtual connections, there is one more administration type called RECEIVER_NOT_KNOWN. With this type it can be described that a request or message previously sent did not reach its designation because it was not known at the routing instance. Therefore, this virtual connection should be disposed of.

The negotiation of closing the physical connection is similar to the previously described protocol of closing the virtual connection and only needs the administration types CLOSE_PHYSICAL, CLOSE_PHYSICAL_ACK and CLOSE_PHYSICAL_DECLINE so refer to this section for a more detailed description.

The implementation of the Physical Layer is not broken down as the administration of the physical connection is implemented by the leJOS NXJ firmware itself. The provided interfaces of leJOS NXJ can be used to either connect directly to another device by name, to wait for an incoming connection or to close and clean-up an existing physical connection. As mentioned before, the negotiation of closing the physical connection takes place in the Routing Layer.

All headers and fields are summarized in the following table. In addition, it is shown whether the data part after the header is empty or not. A short description of the command is also given.

MessageCommand

message type	content type	data	description
COMMAND	CLOSE_VIRTUAL	☐	Requests the close of the virtual connection
	CLOSE_VIRTUAL_ACK	☐	Acknowledges the close request for the virtual connection
	CLOSE_VIRTUAL_DECLINE	☐	Declines the close request for the virtual connection
REQUEST	LIGHT	☐	Requests the current light intensity of a remote sensor
	DEGREE	☐	Requests the current compass data of a remote sensor
	FILE	☑	Requests a file with the file name given in the data part

MessageCommand

message type	content type	data	description
SEND	LIGHT	☑	Transmits the current light intensity of a remote sensor
	DEGREE	☑	Transmits the current compass data of a remote sensor
	FILE_EMPTY	☐	Indicates that the requested file was empty (so no data is sent)
	FILE_WHOLE	☑	Transmits a file as a whole, the data part contains the name of the file, its size and content
	FILE_PART	☑	Transmits a file in parts due to a large file size. The data part contains the number of parts into which the file was split, the name of this part of the file, its size and content

RoutingCommand

routing type	administration type	data	description
DATA	–	☑	Contains data to be interpreted by the Message Layer (the data part contains the length of the sent message and the message itself packed as described in the Message Layer)
VIRTUAL_ADMIN	CONNECT_REQUEST	☐	Requests the initialization of a virtual connection
	CONNECT_ACK	☐	Acknowledges the initialization of a virtual connection
	CONNECT_DECLINE	☐	Declines the initialization of a virtual connection
	RECEIVER_NOT_KNOWN	☐	Indicates that the previously sent message could not be delivered because the receiver was not known at the routing instance

RoutingCommand

routing type	administration type	data	description
COMMAND	CLOSE_PHYSICAL	□	Requests to close the physical connection
	CLOSE_PHYSICAL_ACK	□	Acknowledges the close request for the physical connection
	CLOSE_PHYSICAL_DECLINE	□	Declines the close request for the physical connection

After all headers and fields have been described in detail, we have a look at the entire routing protocol of figure 4.6. At first, we discuss the routing of a message that contains a request for compass data to be transmitted from device A to device B. In the Message Layer, the compass data itself is enclosed in the *data* field and is prefixed with the *MessageCommand* SEND. The Routing Layer adds the *length* in bytes of the current data field and the *MessageCommand* together as a new header. This header is needed that the unpacking Routing Layer knows how many following bytes to forward to the Message Layer. It also prefixes the whole packed message with routing information such as the name of the device A the message originates from in the field *from*, the name of the device B the message is addressed to in the field *to* and the *RoutingCommand*. In this case, the *RoutingCommand* determines that data is sent so its routing type must be DATA. The whole package is given to the Physical Layer where leJOS NXJ additionally adds information to route the message to the right device (this is not further described as this is part of the leJOS NXJ library). The Physical Layer physically transmits the whole message with all headers to the Router.

The Router has to unpack the message until it reaches the routing information. Depending on the field *to* and the *RoutingCommand* DATA, the Router knows that this message is not addressed to itself, but to the device B and contains data to be forwarded. So it decides to forward the contained data to device B. Thereby, it does not decipher what message was actually sent, but only reads the field *length* and forwards as many following bytes as the size of *length* allows. In the receiving device B, the message is also unpacked until the routing information is reached. As the field *to* contains the name of this device B, the message is unpacked further to decipher the contained data.

For routing the negotiation of closing a virtual connection initiated by device A to device B, the routing is handled the same way, but the optional *data* field does not contain any data. The *MessageCommand* is composed of the message type COMMAND and one of the content types CLOSE_VIRTUAL, CLOSE_VIRTUAL_ACK or CLOSE_VIRTUAL_DECLINE and is packed in the Routing and Physical Layer as men-

tioned above. The whole message is then transmitted to the Router. The Router does only unpack the message until the Routing Layer to know where to the message is addressed. Again it does not need to decipher the actual message as it is not necessary for the Router to know whether any negotiation is going on. It simply forwards the message to device B as written before. Only device B unpacks the whole message as the message is addressed to it.

For all other tasks of the Routing Layer like initialization of a virtual connection and negotiation of closing, the Message Layer is not needed. For these tasks, only the *RoutingCommand* and the routing information from which device to which device to send are needed. The fields *MessageCommand*, *data* and *length* are unnecessary in these cases and are therefore marked as optional in figure 4.6. For the request of a virtual connection from device A to device B, the *RoutingCommand* would be composed of the routing type `VIRTUAL_ADMIN` and the administration type `CONNECT_REQUEST` while the the field *from* contains A and the field *to* B. This message is transmitted to the Router which unpacks the routing information and forwards the connection request to device B. Device B has to evaluate whether any application is waiting for a connection request and therefore to accept it or no application is waiting and therefore to decline it.

As mentioned in the description of the communication model in chapter 4.1.2.3 on page 74, a routing device is only needed if the network is based on virtual P2P communication. So for the direct P2P communication, the complete Routing Layer is not needed and therefore also the routing device can be omitted. The transmission is handled similarly to the description in the previous paragraphs, but all headers of the Routing Layer can be omitted as the sending device directly transmits its message to the correct device without any routing.

In summary, with the architecture described in the first part of this chapter, we were able to build robots that perceive their environment and react according to their perceptions. The reactions are encapsulated into behaviours so that the overall behaviour of a robot can easily be extended and clearly be represented. Additionally, robots can communicate with other robots to coordinate a collective task. For that purpose, we recommend to use the virtual P2P communication with a routing instance for controlling the communication as direct connections result in a delay for switching between connections which hinders time-critical multi-endpoint communication. Nevertheless, robots programmed in leJOS NXJ using this architecture are also able to communicate directly using the normal P2P communication.

4.2 Applications

In this section, we want to introduce the evaluation of the suitability of LEGO MINDSTORMS for swarm robotics by developing practical applications. Some exemple implementations should reveal whether several benchmark problems for swarm robotics can be realized using LEGO MINDSTORMS. Therefore, we tested different models of LEGO MINDSTORMS robots, implemented various behaviours using the architecture described in the last section, made use of normal and virtual P2P communication for collective or coordinated tasks and checked the possibilities of LEGO MINDSTORMS in computational power, precision of sensors and motors, physical and virtual interaction and durability of power supply. For each task we evaluated the overall performance of our implementation and identified achievements and limits of LEGO MINDSTORMS.

For testing the feasibility of LEGO MINDSTORMS solutions to real-life problems, we formulated the task of collective transportation along a line. The idea is to build a swarm of LEGO MINDSTORMS robots that pushes an object too heavy or too big for a single robot along a black line on a white surface. The line may thereby take an arbitrary course of straight lines or curves which the swarm of robots must detect. More specifically, we thought of one robot reading the line on the floor and therefore gaining the knowledge of the actual course for pushing the object. This robot must not function as the coordinator of the whole task by giving commands to the other robot, but rather as a base of knowledge where the other robots can get the information required for their individual job. Two other robots are meant to actually push the object in front of the line reading robot. They both have to acquire their information about the actual course from the line reading robot and to individually determine their next actions depending on that. While pushing the object, the two pushers must keep the object directly in front of the line reader and above the line.

Unfortunately, we had to discover that robots that are aware of their location or a global goal are far away from current research in swarm robotics. Swarm robotics currently focuses on solving problems in swarms of robots which consist of relatively simple individuals. It is a current research topic to integrate intelligent agents into swarm robotics. So for this master thesis, we did some preliminary work for testing LEGO MINDSTORMS in the tasks of line-following, heading following and distributed line-following as it is described in the next sections. Building on that foundation, especially localization and distributed intelligence have to be further explored for solving the formulated collective transportation problem. At the end of this section, we suggest some further topics worth exploring to solve our formulated problem.

In figure 4.7, an exemplary course is shown on which all line-following applications are running. It consists of single line segments spanning a DIN A4 sized sheet of paper – straight or curved – that can build an arbitrarily shaped course. The line itself is 4 cm wide. All applications have been tested on different courses with sharp or smooth turns running clockwise or counter-clockwise to eliminate accidental effects.

Figure 4.7
An exemplary course for testing a line-following robot.

In the following, we describe each of the three applications mentioned above, line-following, heading following and distributed line-following. The used robotic models as well as the software is discussed and concluded with a summary of achievements and limits of the current implementation.

4.2.1 Single line-follower

The first project covers the development of a line-following robot. This robot should be able to recognize a black line on a white surface and to detect the curvature of this line in order to drive along the line. This application is challenging in different ways. At first, the line-follower should be able to follow any line with an arbitrary curvature. The robot does not know the course beforehand. It has

to recognize the course on-the-fly. Thereby, the line is composed of straight or curved segments while a curve may arch either to the left or to the right and has an arbitrary radius. The line-follower should drive this course smoothly. It is meant to jerk intensively left or right, but rather to adapt its steering gently. In addition to that, we want the robot find the line by itself. The robot should be able to be placed anywhere on the white surface and nevertheless be able to find the black line. Another challenge is the execution under differing light conditions. Depending on the ambient light, the contrast of black and white may vary. Therefore, different light values may stand for the line or the loss of the line. To meet these challenges, we have built two different robotic models that can be equipped with two different behaviours each. The following sections introduce and compare them.

4.2.1.1 Robotic models

The first robotic model tested is a tracked vehicle shown in figure 4.8. It has two tracks twined around two wheels each. The front wheels are motorised and can be steered forwards or backwards. To drive a curve, one wheel has to be steered faster than the other. The main computing unit, the NXT, is mounted in the middle of the chassis above the two tracks. At the front of the chassis, a colour sensor is attached to face toward the floor. With this colour sensor, the contrast of the surface underneath is measured. To be somewhat independent from the ambient light, the LED lamp emits blue light during execution. This does not guarantee complete independence from differing ambient light, but decreases the influence of shades or direct illumination. To achieve good readings, the sensor is mounted only 0.5 cm above the floor. So the emitted light reaches the floor very focused and the sensor is concentrated on a particular point of the surface underneath.

This robotic model is very stable in general. Due to its tracks, it touches the ground on long ground patches. With the large contact surface, a tracked vehicle is able to move heavy-weight load while wheels may skid soon. The tracks and the wide distance between the tracks contribute also to the static balance of this robotic model. As the tracks can be independently rotated forwards or backwards, this robot is able to spin in place. Thereby, the centre of rotation is between the two tracks. Depending on the centre of gravity of the model, the centre of rotation may move to the front or the back, but never reaches the front wheels.

Unfortunately, the actual rotation or steering is difficult if the robot is meant to drive a curve. For driving a curve or spinning in place, a large portion of the tracks has to skid against the surface. Depending on the nature of the ground, the actual turn rate may differ significantly. Also the shift of the centre of gravity to one side or end of the robot's chassis has a large effect on the skidding of the tracks. To compensate this uncertainty, the turn rate is adjusted by using a

Figure 4.8
A tracked LEGO MINDSTORMS *model (the compass sensor mounted above the robot's chassis is used for the application in chapter 4.2.2 on page 94).*

compass sensor. The current heading is continuously compared to the intended heading so that the turn rate can be modulated to match the desired rotation. However, skidding is very inefficient on smooth terrain. While tracks keep a robot manoeuvrable on loose terrain, they lose very much of their energy in skidding while driving on even ground.

For applications in the laboratory the disadvantages of the tracked robot overshadow its advantages. Therefore, we have built another robotic model that has a good and precise steering behaviour. This robot resembles a trike in having three independent wheels (see figure 4.9). Two bigger standard wheels are attached at the front of the robot's chassis. They are motorised and can only be steered forwards or backwards in similarity to the tracks. The third wheel is much smaller and is hinged to rotate freely. In out model, this so called castor wheel is not motorised. Similarly to the tracked vehicle, this robot has a colour sensor mounted to face the floor in the same manner as mentioned above. The NXT is attached in the middle between the three wheels to guarantee a good distribution of weight.

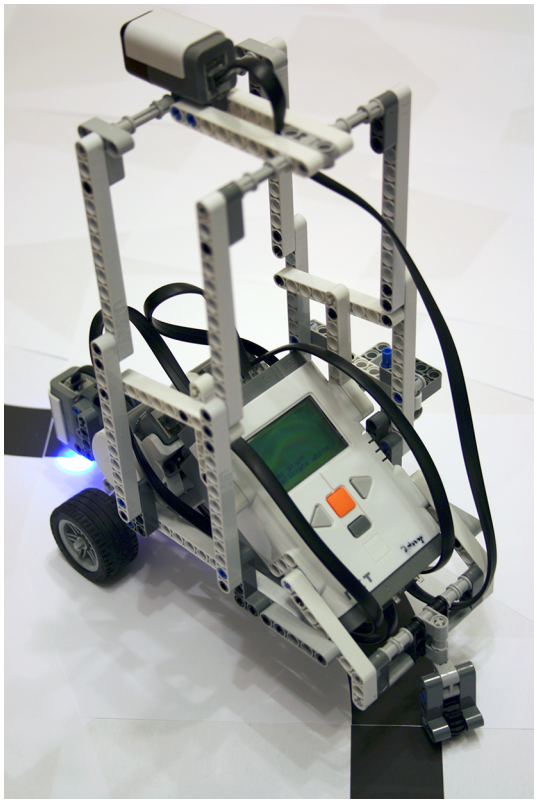

Figure 4.9
A LEGO MINDSTORMS *trike (the compass sensor mounted above the robot's chassis is used for the application in chapter 4.2.2 on page 94).*

With the three wheels, static balance is guaranteed for this robot which means that no mechanism of self-balancing is needed to prevent this robot from tumbling. Though this model is not as stable as the tracked vehicle because of the single rear wheel. Compared to the tracked vehicle, the trike is controlled more accurately. Small movements are carried out in more precision than with tracks because the LEGO MINDSTORMS tracks are skidding on the rims of the wheels. As the wheels barely skid on the floor, the trike is also faster while being supplied with the same power as a tracked robot.

However, having a short ground contact patch with only three wheels reduces the grip of the robot on the ground. Heavy objects cause the wheels to slip while pushing the object. In addition, the centre of a rotation is no longer in the centre of gravity, but moves to the centre between the two steerable wheels. Therefore, the robot wiggles at the back when changing direction and needs it additional space to sheer out.

4.2.1.2 Software

Both introduced robots can be executed with two different compositions of behaviours for line-following. They both are implemented according to the architecture for a single robot described in chapter 4.1.1 on page 62. The components for sensing and actuating are implemented to guarantee correct control of motors and sensors and can be used in any application. The Communication and DataSender is not needed for this application because the line-follower does not have to communicate to other robots. However, the logging file can be transmitted to a computer if the Communication component is included. The BehaviourModel is the only component that includes application specific code. In this component, several behaviours have been implemented to suit the line-following application. For implementing, leJOS NXJ was used as programming platform. As the official release in the current version 0.8.5 lacks some features like the correct support for interfaces or for the new colour sensor, the implementation is based on the current developer snapshot which will soon be released.

The first composition of behaviours implements a *basic line-follower*. This robot is not able to drive a smooth course. Additionally, it is not possible to calibrate the contrast of white and black so that this robot is dependent on the correct ambient light. The composition consists of two behaviours – one for searching the black line called *basic find line* and one for following the black line called *basic following*. Following the line is the standard behaviour of this composition. The robot thereby always drives a straight line without any curve. Strictly speaking, this behaviour does therefore not make the robot follow the line. Only together with the second one the desired overall behaviour is achieved. The second behaviour of searching the black line comes into action if the ground underneath the colour sensor is not black. If the colour sensor detects another colour than black, the behaviour makes that the robot searches for the line by rotating in place until the colour underneath is black again. If the line was not found until the robot has rotated 360 degrees, the robot is steered some distance to the front and the spinning in place is repeated. In general, the line-following behaviour is gained because the robot at first drives straight until it reaches the border of the black line. There, the second behaviour comes into action because the read colour is not black any more. The robot rotates in place until it is above the line and again starts to move straight.

Obviously, this introduced composition of behaviours is still rather unintelligent. The second composition of behaviours for an *advanced line-follower* tries to improve the *basic line-follower* of the last paragraph. The previous standard behaviour *basic following* made the robot drive a straight line regardless of the black-white contrast on the floor. The new standard behaviour called *advanced following* takes this contrast into account and adjusts its course according to that. A certain grey value depicts the range of the line which the robot should

always try to follow. For example, a value of 20 on the grey scale is read if the sensor is directly above the border between the black line and the white surface. If the robot has to keep the left border underneath its sensor, simple rules can be derived. If the read light value is smaller than 20, the robot is moving away from the border and reaching the range of the black line. If the robot tries to follow the left border of the line (so the black line is on the robot's right), it has to adjust its course to drive a little to the left. Otherwise, if the read light value is bigger than 20, the robot is moving to the white surroundings and has to steer to the right. While continuously adjusting the course to the current light value, the line-follower does not drive straight the whole time, but steers curves and straight lines whenever needed.

This principle of adjusting output values according to the difference between an actual value and its desired value is called *feedback control* according to [ÅM10]. It is well known from biological systems, where for example the level of glucose in the bloodstream is regulated by the production of insulin, and it is applied in many engineering fields like aerospace and robotics. The simplest control is the *on-off control*. In our line-following example, this controller would only determine whether the actual light value is bigger or smaller than the desired light value and apply a fixed correction. Therefore, the controller is very sensitive to small changes or sensor noise. For instance, a current light value of 100 would result in the same corrective action as a light value of 21. That overreaction causes wide oscillations of the robot.

Therefore, we use a more elaborate controller, the *PID control*. This controller uses three terms to compute the correction: **P**roprotional, **I**ntegral and **D**erivative term. The first is the *proportional control* which minimizes the overreaction of the on-off control. In this step, the difference between the actual value and the desired value is computed and the correction is derived depending on that difference. Only if a certain upper or lower limit is reached, a fixed maximal correction is applied to avoid a stimulation of the actuators beyond their capabilities. So for the line-following example, a light value of 21 would result in a small correction of the course to the right while the light value of 100 would achieve a sharp turn to the right. Unfortunately, with proportional control only the system does not become steady at the reference value (see [ÅM10]).

A second controller, the *integral control*, avoids this steady-state errors by accumulating the error (the difference between the actual and the desired value) over time which is the integral of the error over time in general. The goal of this controller is to keep the accumulated error close to zero. For example, if the error of the line-follower stays positive over a longer period of time, this integral increases and causes the robot to steer a sharper turn to the right as it would be the case with the proportional control only. That means that the integral control takes the past into account and tries to derive the correction from the knowledge thereof.

This consideration leads to the idea of predicting the prospective error in addition. Therefore, the third *derivative control* is introduced. This control interpolates the prospective error by differentiating the error with respect to the time. A common implementation of the derivative component is taking the difference between the actual error and the previous error. If the two errors do not differ, for instance the line-follower has been too far to the left in two preceding steps, this derivative control predicts that the line-follower will stay too far to the left if the same correction is applied and will increase the turn to the right.

The overall control action of a PID control is the sum of these three controls while the **P**roportional control is based on the present error, the **I**ntegral control on the past error and the **D**erivative on the prospective error. Thereby, a constant factor has to be determined for each term how much the term should influence the corrective action. The assignment of these factors is very complex and depends strongly on the system that is controlled. Beside experimental tuning, there exist some tuning rules like the Ziegler-Nichols-Tuning-Rules. In this master thesis, Ziegler-Nichols-Tuning and experimental tuning have been tested. Unfortunately, Ziegler-Nichols-Tuning caused high oscillation of the robot while we had better results by using experimental tuning. However, the tuning is not examined any longer as we are hoping to include learning techniques for deriving the parameters (for a detailed explanation of PID control and tuning refer to the [ÅM10, ÅH05, ZN42]).

The implementation of PID control for the line-follower is given in figure 4.10.

```
1 float lightValue   = colourSensor.getLightValue();
2 float currentError = lightValue - reference;
3 integral           = integral + currentError;
4 derivative         = currentError - lastError;
5 lastError          = currentError;
6
7 float turnRate = kp * currentError +
8                  ki * integral +
9                  kd * derivative;
```

Figure 4.10
Implemenation of a line-following PID controller

At the beginning of the PID control the variables for `integral`, `derivative` and `lastError` are set to the value of 0. Then the computation of the present, past and future term cycles and returns a turn rate how to steer the robot at the current situation of readings. For that, the current light value of the underlying surface is determined by `colourSensor.getLightValue()`. For the present term, the difference between the current light value and the reference light value is computed in line 2. For the past term, the current error is simply accumulated in the variable `integral` as suggested in the last paragraph. However, the `derivative` term for predicting

the future error is estimated by the difference between the current error and the current error in line 4. Weighted with a constant factor for each, these three terms determine the turn rate for the PID controlled robot.

This PID control is the main part of the *advanced following* behaviour. The *basic find line* behaviour for searching the line complements the overall behaviour for the line-following robot. At first, the robot follows the line by computing its turn rate as described in the previous section about PID control. Whenever the surface under the colour sensor is completely white, it starts to search for the line as described for the composition of behaviours for the *basic line-follower*. This composition for an *advanced line-follower* holds two advantages: Firstly, the robot does not turn sharply or oscillate if the parameters are correctly tuned. It rather follows the line smoothly. Secondly, the behaviours are not based on the discrete colours black and white any longer, but rather on grey values. At the beginning, the robot calibrates the current light values that depict white and black and computes the contrast of the border of the line in respect to these calibration values. Therefore, the robot is able to run under different light conditions as the current values are calibrated at the beginning. During execution, the robot also does not rely on discrete colour values. However, it reads the current light value in order to steer with the aid of PID control and switches the behaviour for following the line to searching for the line whenever a certain light value (computed after the calibration) is reached.

4.2.1.3 Achievements and limits

After we have discussed the different robotic models and implementations of line-following, we want to conclude with a short comparison. We start with the behaviours because the PID controlled behaviour for the *advanced line-follower* outperforms the *basic line-follower* by far whatever robotic model is used. The *basic line-follower* is very basic. First of all, the overall behaviour is designed for special ambient light conditions and does not adapt to different light sources. Of course, a calibration phase could be added to overcome this limitation, but the robot would nevertheless follow the line poorly. Since it does only drive straight while on the line and spin in place while off the line, the robot cannot steer any curve smoothly. It rather jerks at any curvature. In contrast, the PID controlled behaviour of the *advanced line-follower* causes smooth actions if the parameters are tuned correctly. Hence, this tuning is the big challenge with a PID controller. Additionally, this behaviour is based on light values instead of discrete colour values so that a calibration phase at the beginning is easily added. These readings of light values should always be done with only one colour of the LED switched on. The use of white light results (primarily in leJOS NXJ) in constantly turning on and off the different colours which consumes a lot of time and many resources. However, reading light values is as sufficient with coloured

light as it is with white light. The only handicap both behaviours exhibit is a restricted search for the line by the use of the *basic find line* behaviour. At the moment, the robot only spins in place to find the line and travels some distance to the front if the line was not found after a rotation of 360 degrees. If the line is to be found behind the robot out of range of one complete rotation, the robot will therefore never find the line. However, this behaviour can easily be adapted to cover a bigger range.

Thus, the PID controlled behaviour was used to compare the two robotic models. For both models, the parameters for PID control have been tuned independently to reach the best performance possible. The main difference between both models is the speed on the course. As mentioned above, the tracked vehicle is skidding at the whole length of the contact patch of the tracks to steer a curve while the trike skids less and has a short contact surface on each wheel. Therefore, the tracked vehicle loses a bigger part of the velocity due to skidding than the trike. The trike circles the course in approximately three quarters of the time the tracked vehicle requires. Thereby, the tracked vehicle is waggling even more than the trike. An exemplary execution can be found on th enclosed CD. Experimental tuning and Ziegler-Nichols-Tuning could not overcome this difference that can be explained by the difficult controllability of tracked vehicles.

Beside these differences, both robotic models performed well. They both could be driven at a speed of 20 wheel diameters per second around the course without losing the line (depending on the battery charge, the maximum speed of a LEGO MINDSTORMS motor is about 40 wheel diameters per second). This high speed could be achieved as the colour sensor could be read every 10 to 15 ms. After a complete loss of the line or when placed rectangularly on the middle of the line, the robot could follow the line after approximately 10 seconds. If fully charged, a line-follower is running for about four hours without any undesired behaviour or exception until the power supply is empty. The curvature of a segment the robot is able to follow is restricted by the choice of the PID parameters to a radius of 7 cm. On sharper turns, the robot loses the line, but the behaviour for line finding sets the robot back on the line. Edges in the course also cause the robot to lose the line as they cannot be driven smoothly. One drawback is the logging of the execution. When logging to the local file system, the motors are not controlled during the real writing process. Fortunately, this does not happen very often and does not last long so that the line-following is barely affected. Similarly, logging to a remote computer consumes so much computation time that the reading of the colour sensor is slowed down and the line-following is a little inaccurate.

Concluding, both robotic models achieve a good performance in line-following when using the PID control. The wheeled vehicle outperforms the tracked vehicle, especially in speed, due to better controllability.

4.2.2 Heading followers

The second preliminary work extends the line-follower previously discussed. A line-following robot circles an arbitrary course while several other robots try to align their heading according to this line-follower. The line-follower should be able to determine its own heading, for example with a compass sensor. The other robots can either request the heading information from the line-follower or register at it for continuous transmission. With this information, they should be able to adjust their own heading according to it. Thereby, the transmission latency should be as small as possible that the heading followers are not lagging behind. The network should also be robust against crashes or failures. That means that a faulty heading follower does not influence the the line-follower itself. Nor does a crash of the line-follower prevent the heading follower from aligning themselves to the last clean heading information. The next sections introduce the robotic models as well as the software for this application and discuss the achievements and limits of the described approach.

4.2.2.1 Robotic models

The robotic models for line-following and heading following are the same as they have been presented for line-following only. Either the more stable tracked vehicle or the trike model that is better controllable can be used for either follower. The only extension is the compass sensor that is mounted high above the NXT. As mentioned in chapter 3.2 on page 36, the compass sensor is influenced by any metal or current-carrying items, especially motors or the NXT. So the compass sensor has to be attached high above the robot's chassis. However, the current attachment of the sensor sometimes causes incorrect readings if the robot abruptly changes its moving direction and therefore induces some rocking of the sensor.

The properties of tracked or wheeled vehicles as described in the previous section are also relevant for this application. Here again, we want to point out that in our experimental setting wheeled vehicles are easier to control than tracked vehicles. Nevertheless, for the line-follower or the heading followers either model can be applied. Even a mixed set of robots (for example a wheeled line-follower, a wheeled heading follower and a tracked heading follower) is possible for this application.

4.2.2.2 Software

To take this application into action, two behaviours based on the architecture of chapter 4.1.1 on page 62 have to be implemented: a line-follower that continuously supplies the network with heading information and heading followers that continuously read the heading information and align according to it. For

line-following, the PID controlled behaviour of the *advanced line-follower* of the last section is applied. However, the line-following robot needs additional threads that take care of the transmission of the heading information. The threads are not another behaviour because the transmission takes place parallel to any executed behaviour. That means that the robot follows the line and in the meantime transmits the heading information without any interruption of the behaviour. There are two possibilities of transmission: the line-follower continuously sends the compass data to any interested device or only responses the compass data upon request. Thus, the line-follower can either store the current compass data in order to answer a request which is done in the Communication module or send it to registered devices of its own accord by the DataSender component.

The robots that align their own heading according to the heading of the line-follower are called heading followers. They have two tasks in general: to request or register for the heading information at the line-follower and to align according to it. Similar to the line-follower, the reception of the compass data is swapped to an independent thread that is executed parallel to the actual behaviour of the heading follower. This thread either requests new compass data continuously or it waits for the compass data without any former request. In the architecture of chapter 4.1.1 on page 62, this thread can be found in the Communication module. The received data is then provided to any behaviour by a wrapper for a remote compass sensor located in the SensorWrappers component. For heading followers, we implemented a simple behaviour that only consists of one part called *follow heading* behaviour. This behaviour takes the remote compass data (via the remote compass sensor currently mentioned) and its own compass data as input. It compares both heading information and tries to rotate in place to gain the desired remote heading. Because new heading information normally arrives fast, the rotation is often interrupted by new instructions. That means that the degrees to rotate are updated whenever new data arrives even if the ongoing rotation is not yet finished. That results in a good accuracy of aligning since the robot tries always to align to the most recent heading information and does not try first to head to the old orientation and then to the new.

Lastly, we want to have a look at the network configuration for the heading following application. Each heading follower must be connected to the line-follower as this robot provides the interesting heading data. Thereby, the connection can be direct or indirect meaning that the NXTs are either connected directly in a P2P network (see chapter 4.1.2.1 on page 70) or to a router that transmits the requests or data to the correct receiver (see chapter 4.1.2.2 on page 72). Both network configurations are realized in this master thesis and can be used for this application. Regardless of the network configuration, both ways of data transmission – by request or send without request – can be realized.

4.2.2.3 Achievements and limits

This extension of the line-following application performs also well. The PID controlled line-follower can follow the line with the same speed and accuracy as before although the NXT has to serve additional threads for data transmission and reception of requests. Thereby, some time limits described in the next paragraphs have to be kept that a new tuning is not needed.

In general, the compass sensor can be read every 10 to 20 ms without any influence on the line-following behaviour. Theoretically, a transmission to a remote device via direct P2P communication can be started every 20 to 30 ms. This overall transmission delay is composed of the time for packing and unpacking the data according to the protocol described in chapter 4.1.2.4 on page 76 and the time for the transmission itself. Unfortunately, if the line-follower should transmit its compass data directly to one remote device, this transmission delay has even to be increased to 80 to 90 ms so that the line-follower is still able to follow the line without any new tuning.

As soon as the line-follower has to handle more than one physical direct connection, this delay is drastically increased. The implementation of the Bluetooth stack in leJOS NXJ as well as in the original LEGO software needs approximately 300 ms for cleaning up the queues for one connection and switching to another. If the line-follower transmits its compass data continuously to different devices via direct connections, the normal delay is increased by this switch delay whenever two preceding transmissions are directed to different devices. If the line-follower must send its compass data to more than two devices, this switch delay is added for every connection that has to be cleaned-up. That means that in a network of four devices, one device receives data approximately each 900 ms because its own connection has to be cleaned-up and then two other connections have to be served before the first device can receive data again. To overcome this restriction, the use of a routing computer is recommended as introduced in chapter 4.1.2.2 on page 72. In this network configuration, each NXT is only connected to the router while all other connections are emulated on this routing connection. Thus, the normal delay for a transmission doubles because of the additional device in the middle, but the switch delay is eliminated. Theoretically, the sending device is able start a new transmission every 20 to 30 ms as said before. The router needs 20 to 30 ms in addition so that the message reaches its receiver after 40 to 60 ms. If more than one device is to be served, one device has to wait for its message until all other device have to be served so that for four remote devices each device receives the data after 60 to 90 ms because two devices are served before the third. With the extension to multi-cast, this delay could further be reduced since then the sending device would only need to send one message to the router and the router would forward it to all other devices. So whenever all devices request the same data (as here the heading information), the sending device sends the message simultaneously to all devices and is able to start another sending to all

devices after 20 to 30 ms. Of course, the line-follower cannot transmit at full speed and without retuning the line-follower, so we introduced an artificial delay of 500 ms. That means in our implementation the heading followers receive their compass data after 500 ms what is sufficient enough for this purpose.

With both network configurations – P2P communication or virtual P2P communication – an arbitrary number of heading followers can be handled. We could only test this application for up to three heading followers, but there is no restriction to employ more robots. Likewise, each robotic model can be applied as line-follower or heading follower. In a mixed set of robotic models, the difference between tracked and wheeled vehicles can be seen again. Because the wheeled vehicles are better controllable and are not skidding, they reach their desired heading faster and with fewer subsequent adjustings. Overall, the network is very robust against any crashes or failures. If one of the heading followers is crashing, the line-follower and all remaining heading followers are working as before. If the line-follower stops working, the heading followers use the last clean compass data and align according to it. The only occurring interference is caused by metal or current-carrying items. For example, if one of the robots is placed above a current line, its heading might permanently deviate from the desired heading.

Although these heading followers are working well, we could think of some improvements. Firstly, the line-follower is a bottleneck in the distribution of the heading information. All devices have to receive their data from this device only. An improvement could be to provide other devices that request the heading information from the line-follower, but supply further devices with this information. Thus, fewer devices are requesting directly from the line-follower. Secondly, the use of a Kalman filter could be advantageous. With a Kalman filter [Kal60], noise reads of sensors can be eliminated. For example, if a single value does not fit the sequence of received compass data, this value is detected and corrected by this filter as an exact correction to this noise read drives the robot in a totally wrong direction.

4.2.3 Distributed line-follower

In order to approach the overall goal of collective transportation along a line, we want three robots to follow a line together. Thereby one robot is not motorised so that it has to be pushed by the others. However, this robot is the only one that can read the line underneath so that the other robots have to rely on that information to steer the course. Similar to the single line-follower, this small swarm of robots should be able to follow any course by on-the-fly recognition. The steering should be as smooth as possible. Additionally, the application should be executable under different light conditions. However, for distributed line-following even more challenges have to be faced. Firstly, the pushing by the motorised robots should

be synchronized such that the non-motorised cart does not lose the line. Even if the line is lost, the swarm should be able to search for the line together by pushing the cart in a coordinated way to find the line. The pushing itself is a second challenge. We have to find a way how the pushers can match their actions to move in the desired direction and how they can determine where each robot is pushing in order to compute their next pushing action.

4.2.3.1 Robotic models

For this application, we need two different robotic models: one that serves as the object to be pushed and one that functions as pushers. The object to be pushed is the cart in figure 4.11. It is three times as wide as a trike and consists only of the NXT mounted on a rectangular chassis and six free hinged castor wheels of different sizes. The wheels on the side are mainly holding the cart on course while the middle wheels only support the weight of the NXT on the chassis. None of the wheels is motorised. Thus, they are as free to move as possible. At the front of chassis, a colour sensor is attached facing the floor. As it is used to measure the contrast on the ground, its distance to the ground is only 0.5 cm like for the single line-follower. The cart can be either pushed at the back of the chassis or pulled at each side. The pushers are connected to the side of the cart.

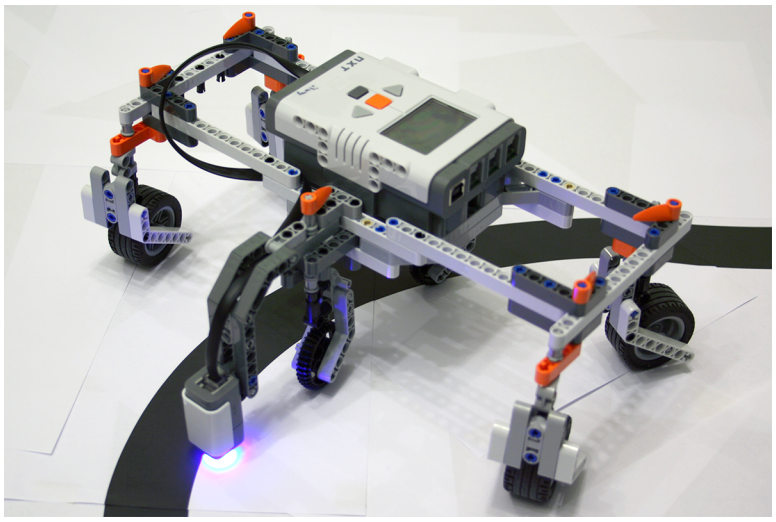

Figure 4.11
A non-motorised LEGO MINDSTORMS *cart which functions as the object being pushed by other robots.*

Since we observed that wheeled vehicles are better controllable and more accurate, we decided to design the pushers like the line-following trike. The model is exactly the same as for line-following purposes, but its moving direction is reversed. This is because if it is the other way round, the light sensor would be in the centre of

the rotation which is a handicap for the *find line* behaviour. For simplification, we connected the pushers to the cart with a fixed bonding at either side of the cart.

4.2.3.2 Software

The implementation of the cart is rather simple because the only thing to do for the cart is to provide the current contrast of the ground to other devices. Therefore, no behaviour is implemented for the cart. The light value is continuously read and provided by the Communication and DataSender module in their own threads. These modules wait for any request or continuously transmit the read colour sensor data in analogy to the previously described line-follower that transmits its compass data.

However, the pushers have to receive this sensor data. They do therefore also implement the Communication component in order to request or wait for the light values. Much like the heading followers, this reception is working parallel to the push behaviour and does not interrupt the behaviour. The push behaviour itself is very similar to the PID controlled behaviour of a single line-follower. Instead of using its own colour sensor for determining the current light value, each pusher receives the light value from the remote cart. Afterwards, each pusher computes the turn rate for the cart individually. Thereby, each pusher uses the PID behaviour explained in chapter 4.2.1 on page 85 to compute the current turn rate that the pushed cart has to steer. As each pusher is fixed to the cart, it knows its distance to the middle of the cart. Hence, each pusher can infer which turn rate it has to follow how fast in order to push the cart the desired turn rate at the desired overall speed. Both pushers compute these turn rates and velocities individually without any synchronization after computation.

4.2.3.3 Achievements and limits

In general, the distributed line-following is working like a single line-follower. At the beginning, we have a calibration phase where the non-motorised cart calibrates the light values for black and white. Due to this calibration, the line-following is almost independent from the ambient light conditions. In addition, the colour sensor illuminates the ground underneath with the blue coloured LED in order to guarantee steady contrast readings.

The challenge that most restricts this application in comparison to the single line-follower is the delay for transmission of light values from the cart to the two pushers. While connected directly to n other devices, we have to struggle with a delay of up to n times 300 ms for cleaning-up connections for a switch. Even with a virtual P2P communication network, we have to face a delay of 40 to 60 ms as explained in the last section about heading following. Because each pusher

has to calculate its PID controlled turn rate in each step, the transmission delay even rises to 80 to 90 ms due to high utilization of the main computing unit. Comparing to the single line-follower, that is long where no new sensor data is available for adapting the course. Due to that delay, the pushers have to drive much slower to hold the cart on line. Thus, the cart itself is pushed to drive at a speed of 2 wheel diameters per second (comparing to 20 wheel diameters per second for a single line-follower). The two pushers deduce their velocity according to the course they have to drive. This speed is currently the maximum for fulfilling the application. If the cart is driven faster, it would be able to detect the line in one step, but might have lost it until the next step where the pushers are able to receive new sensor data. However, the cart "waggles" much around the course due to the described delay. If the cart is directly above the line in the first step, the pushers will steer so far to the left that in the second step the cart is at the very edge of the line. For an observer, this might look as if in each step just one pusher is moving, but actually both are moving. Because the colour sensor mainly detects the very edge of the line or the line itself and not any contrast in between, the turn rate for the cart is always relatively high so that one pusher has to move fast with a big radius and the other slow with a small radius. Of course, the sharp turn can be omitted by adequate, in this case lower values for the PID parameters. However, this would restrict the minimal radius the cart is able to follow. With the current parameters, the cart is just able to follow a curvature with a radius bigger than 13 cm at minimum.

Compared to the single line-follower, we had to change the conditions when the distributed line-follower is to be thought to have lost the line. As the cart jerks from the middle of the line to the very left border of the line, it often happens that the light value signalizing completely white and therefore the loss of the line is detected. In the majority of cases, this does not mean that the line has really been lost, but rather that the pushers move too far to the left and a correction to the right would solve the problem. So we altered the condition for the detection of the loss of the line to "completely white has been read for at least three times in a row". With it, the pushers are given the chance to compensate the overreaction to the left by a correction to the right before the behaviour for searching the line is started. With that change, the cart almost never loses the line on our arbitrary course with curves with at least 13 cm.

The current implementation of distributed line-following holds some problems worth studying in the future. Due to the transmission delay, the overall speed of the distributed line-following is very slow. In order to keep this delay as small as possible, the pushers do not request the light values from the cart, but wait for it without any previous request. Further improvements should therefore include the reduction of this transmission delay in order to drive the cart a little faster and the grant of the possibility for requesting the light value. At the moment, we do not know how to avoid the switch delay of the NXT so that we have to use

the more complex protocol via a routing device. Further improvements of this protocol with respect to the evaluation time would be interesting. Additionally, a synchronization phase between the two pushers is needed. Momentarily, they are both receiving the light values and computing their actions independently without any negotiation or coordination. This is working sufficiently enough for the moment, but a protocol for exchanging and negotiating the collective movement would improve the coordination of action.

4.2.4 Future work

As mentioned at the beginning of this section, these applications are preliminary studies in order to achieve our formulated collective transportation problem. Later on, we want to solve the problem of pushing an object too big and heavy for one robot freely without any fixed connection between the pushers and the pushed object. Research recently starts to consider directed collective transportation within a cluttered environment [FBBD10] while only few other achievements in collective transportation were made [CNB+06, GD09, TD06]. However, we need a very accurate way of collective movement so that the pushed objects are held on a line that is recognized on-the-fly either by the object itself or the swarm.

Furthermore, the spreading of information about the course to follow must be coordinated in a decentralized manner. At the moment, there is no coordination at all, but we want to provide the information globally so that all robots do not have to request their information at a single device, but at several different. We want the robots also to negotiate and coordinate their movement in common.

Besides coordination, localization is an important topic if the robots are pushing the object without any fixed connection and especially if the object itself does no longer read the line, but another robot. Each robot needs therefore accurate localization information about its whereabouts, its relative location to the object and the location of the object with reference to the robot with the colour sensor (and therefore the line). Hence, the robot could estimate how the object is situated, how far the course of the pushed object deviates from the line and how the robot itself has to move in order to hold the object on course. For localization, the current hardware of LEGO MINDSTORMS is not very well equipped. The only sensors provided by LEGO are a compass sensor and distance sensors. To solve this problem many third party sensors such as cameras, RFID sensors or accurate distance sensors should be taken into account. A camera mounted above all robots may also help to identify each robot's position in a global reference.

In swarms of robots, we also have to take care of failure detection. We need a simple possibility to detect a failure or crash of one of the robots. The other robots should be able to eliminate or repair the faulty individual and to compensate the malfunction or loss of this robot.

In real life, a swarm of robots normally has to discover its surroundings before it can start its task. For the task of collective transportation, we want to have a swarm of equally equipped robots – that means each is capable of pushing an object or reading a line. Each robot does not have a special job beforehand. At the beginning, they all should swarm to find the object to move and the line along which to push. If one robot has found one of these, it should inform all other robots and they should negotiate which robot takes over which job. They should reach a common decision and start their overall task together. For that, we need a simple mechanism for swarming and detecting the desired objects, for example they could be marked with special coloured LEDs or RFID tags. Afterwards, the robots must have a precise negotiation protocol in order to find a common decision. For the execution of their task, a coordination protocol as mentioned before is indispensable.

To further extend a single robot to an intelligent agent beside the negotiation process, a first approach would include a learning algorithm for learning how to follow a line. This would for instance solve the problem of tuning a PID controlled line-follower. The first challenge is to develop a cost-benefit function for a single robot to learn how a line is followed. Thereby, it is important to penalize searching for the line and jerking between the two borders of the line while moving forwards. Keeping a certain contrast underneath is rewarded high enough to prevent the robot from pausing above the line without any movement. This learning algorithm must then be extended to hold for a distributed line-follower. Thereby, the difficulty is that there are many robots that have to learn a collective strategy of line-following. The action of one robot always influences the other robots and has to be considered in the learning process. Especially distributed learning is an open field of current research.

Chapter 5

Conclusion and further prospects

In this master thesis, we had a detailed look at the capabilities of LEGO MINDSTORMS for swarm robotics and intelligent agents. We found them as an inexpensive off-the-shelf platform with a wide range of sensors. Even if some sensors may be missing, third party providers support experimental kits to develop customized sensors. Due to multiplexers each LEGO MINDSTORMS robot can be equipped with up to 100 sensors to perceive its environment in full range. With these sensors and the LEGO motors, the robots are gaining a precision sufficient enough for scientific prototyping and getting a first grip at solutions in swarm robotics. That is why several research groups appreciate LEGO MINDSTORMS as prototyping platforms and the active community – may they be non-professionals, professionals or researchers – is growing.

In addition, the leJOS NXJ library provides a comfortable programming interface for LEGO MINDSTORMS. It is one of two languages for LEGO MINDSTORMS NXTs that is object-oriented and therefore supports the current trend of object orientation in software engineering. With this library, we could implement different applications using an elaborate architecture for controlling a robot's behaviour. Due to the support of multi-threading, we were even able to execute different tasks as communication, logging and steering in parallel. Even if the library is not fully evolved, leJOS NXJ is by widely used now and has a very active community. The developers themselves explain the use of leJOS NXJ features and are very open for comments and suggestions.

Nevertheless, LEGO MINDSTORMS have some limitations that restrain their use to prototypic robots only. Beside the mentioned low accuracy of the sensors and motors, the power of an NXT is restricted. It is powerful enough to control different behaviours, but processor-intensive artificial intelligence computations should not be done with the main computing unit. Fortunately, the NXT provides Bluetooth for the communication to other more powerful devices. Such computations can be swapped to another device while the simple execution is left for the NXT itself. However, the use of Bluetooth induces some more problems. Firstly, it is

not as fast as Wi-Fi which is the current state of art in wireless communication. Secondly, the direct communication of an NXT is limited to four connected devices, but we could introduce a network configuration where an NXT is connected virtually to an arbitrary number of remote devices by a routing instance. Thirdly, the Bluetooth protocol for NXTs is stream-based so that a broadcast of messages is not possible. However, these limitations are tolerable for prototyping and may be replaced by a custom-built platform if necessary for more precise tests.

For the future use of LEGO MINDSTORMS, three extensions of the current platform and programming interface would be valuable. On the one hand, LEGO MINDSTORMS robots need the possibility of self-localization. With the LEGO sensors, only the current heading and distances to plane surfaces can be measured. For accurate localization, more elaborate techniques such as GPS or vision via cameras must be applied. On the other hand, the simulation of LEGO MINDSTORMS robots would save time and resources. New ideas and algorithms could be tested theoretically beforehand and many problems could be eliminated before a real life test. A last interesting extension for LEGO MINDSTORMS is the programming interface of nxtOSEK. They try to provide a real-time operating system and could therefore help to improve the response time by far.

In summary, LEGO MINDSTORMS are not a high resolution platform, but provide many possibilities for scientific use. Advantages are the various available programming platforms with different levels of abstractions and focuses, the wide range of inexpensive sensors, basic support for wireless communication and the possibility to evolve different robotic models. With some extensions, we hope to solve real life applications as collective transportation on the platform of LEGO MINDSTORMS.

APPENDICES

Appendix A

Communication protocol

This appendix gives an overview how the communication protocol introduced in chapter 4.1.2.4 on page 76 is implemented in this master thesis. Several sequence diagrams show the program flow while receiving routed messages using virtual P2P communication (see chapter 4.1.2.2 on page 72). Sending a message is not illustrated since this is in general just the reversion of the shown figures.

A.1 Routing Layer

The main entry point for receiving a routed message is the `run`-method in the `RoutingReader` (for the normal P2P communication we implemented the `read`-method in the `MessageReader` as the main entry point because the whole Routing Layer is omitted in this implementation). This method loops to continuously read any incoming messages until the communication channel is closed as it can be seen in figure A.1. In receiving a message, this method deciphers each header described in the communication protocol in chapter 4.1.2.4 on page 76. At first, the name of the sending device, the name of the receiving device and at last the *RoutingCommand* called *code* in figure A.1 is read. This *RoutingCommand* is then used to switch between the three tasks of the Routing Layer. If the input stream for this connection was closed from the remote side, the received code has the value of -1 and the `RoutingReader` has to close this connection as there is nothing more to read from this closed stream. Otherwise the code equals the command for the routing types `DATA`, `VIRTUAL_ADMIN` or `COMMAND` and the corresponding method for evaluating the following message is called (their behaviour is discussed in the next paragraphs). For space purposes, the information about the routing type is only stored in the upper four bits of the one-byte long code which is meant by the use of *upperCode* in figure A.1. If the received code is anything other than these commands, the `RoutingReader` does not know how to interpret it and skips the current evaluation.

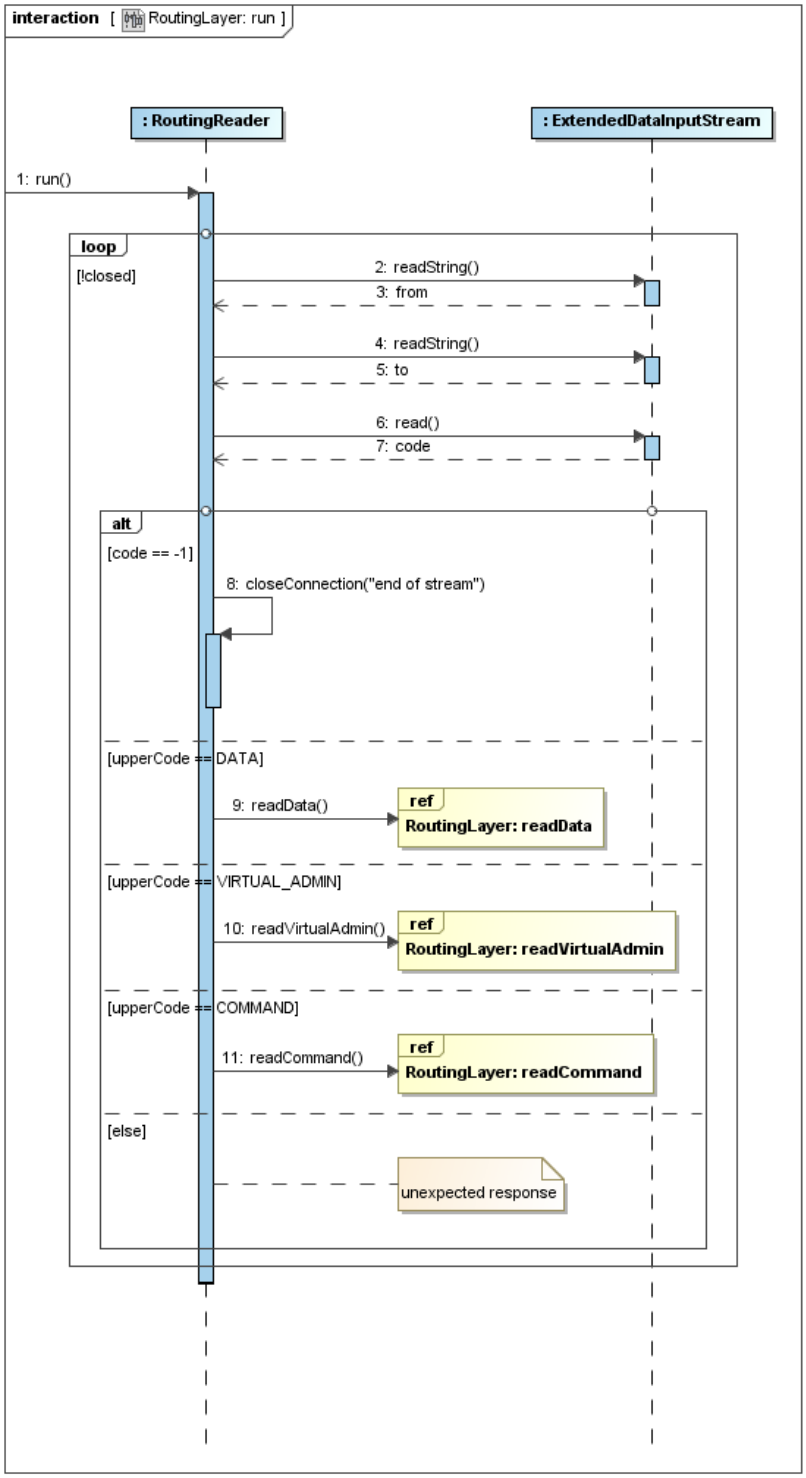

Figure A.1

The sequence diagram for the main entry point of receiving a routed message.

The first task illustrated in figure A.2 is forwarding any message that holds data to the Message Layer (for example sensor data or the negotiation of close of the virtual connection). As the Routing Layer does not know how long the current message for the Message Layer actually is, an integer is sent describing the length of the following message. After reading this integer and storing it to the local variable `length`, the `RoutingReader` reads as many bytes from the stream as the size of *length*. If the message is addressed to this device (so the field *to* equals the name of this device), these bytes are forwarded via the `NXTConnectionManager` that manages all virtual connections on the NXT to the facade for the virtual connection `BTCommManaged`. From there on, the bytes are given to the `MessageReader` which further unpacks the message (refer to chapter A.2 on page 112).

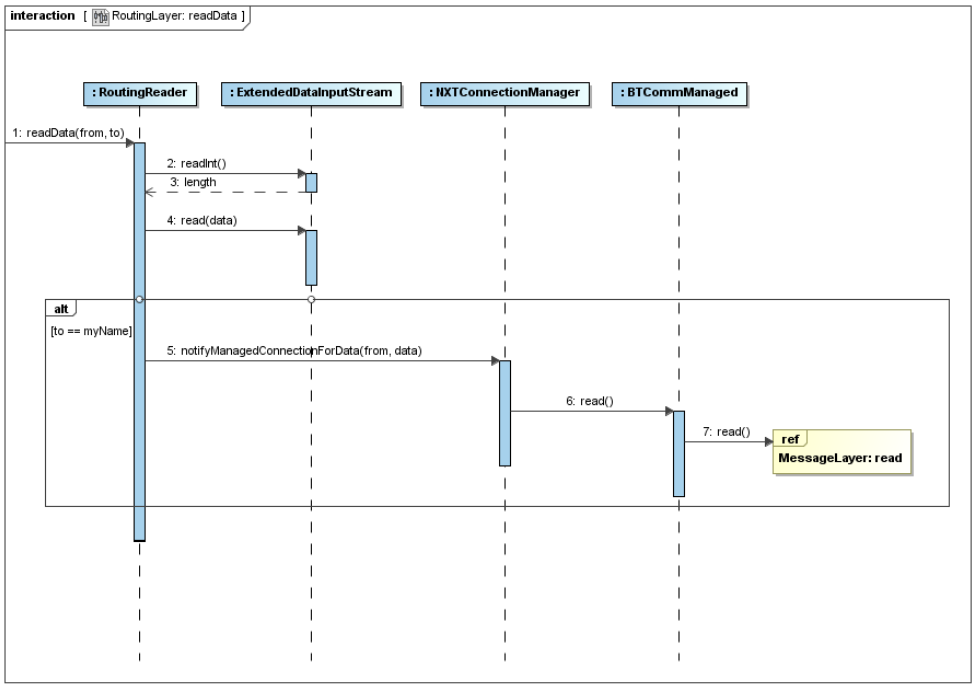

Figure A.2
Forwarding a message from the Routing to the Message Layer.

Figure A.3 describes the program flow for the administration of the virtual connection. Thereby, the `RoutingReader` has to switch between the administration types for a connection request, a positive or negative answer to a connection request and for an unknown receiver. These administration types are encrypted in the last four bits of the one byte long *RoutingCommand code* what is meant by the use of *lowerCode* in figure A.3. In each case, the `RoutingReader` has also to check whether the received command is addressed to this device by comparing the field *to* and its own name. In the case of an connection request, the `RoutingReader` additionally checks whether this device is currently waiting for a connection request from the device the connection request originates from. If so, the `RoutingReader`

forwards to the `NXTConnectionManager` that the connection request can be acknowledged. Otherwise it forwards to decline the connection request. In the case of receiving an acknowledgement or decline, the `RoutingReader` has only to forward this response to the `NXTConnectionManager` which informs any waiting application that the connection could be established or not. Similarly in the case of receiving the administration type `RECEIVER_NOT_KNOWN`: the `RoutingReader` only functions as a forwarder since the `NXTConnectionManager` can only inform each application that the desired receiver is not available. As in the `run`-method, if any other command is received, the `RoutingReader` simply skips the evaluation of this command and proceeds with the next message.

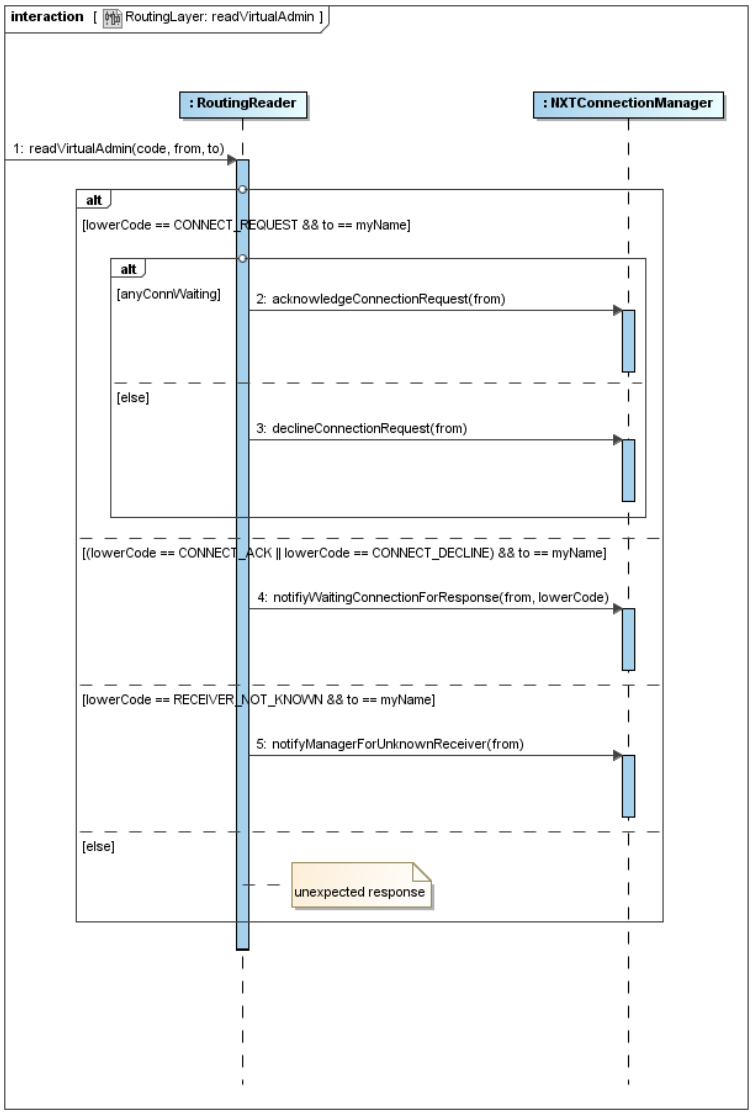

Figure A.3
Administration of the virtual connection.

The last task of negotiating a close of the virtual connection is shown in figure A.4. Here again, the `RoutingReader` has to switch between the three possible administration types of `CLOSE_PHYSICAL`, `CLOSE_PHYSCIAL_ACK` as well as `CLOSE_PHYSCIAL_DECLINE` (again stored in the lower four bits of the code) if the message is addressed to this device. The protocol declares that at first a close request from device B must be received at device A. That request is acknowledged by device A to device B while device A closes the physical connection immediately after this response by forwarding this decision to the Physical Layer. If device B received an acknowledgement, it also closes immediately. If it receives a decline, the connection cannot be closed. So in the current implementation device B informs the `NXTConnectionManager` and simply proceeds with reading the stream. Any other command cannot be understood and is therefore skipped.

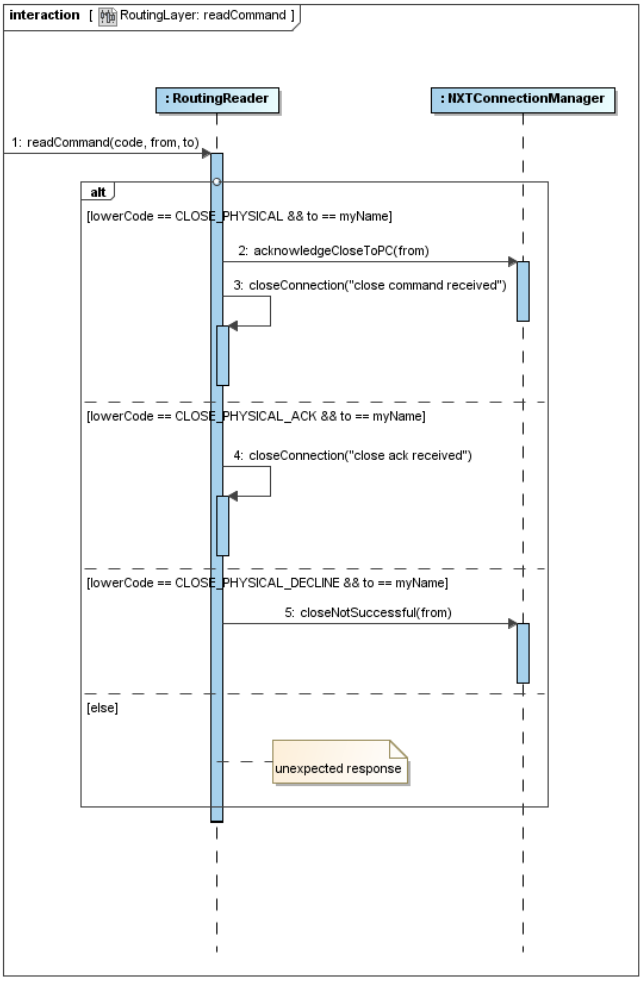

Figure A.4
Protocol for negotiating a close of the physical connection.

A.2 Message Layer

The `read`-method of the `MessageReader` is the main entry point for the Message Layer. All messages that must be unpacked in the Message Layer are firstly given to this method. It reads the *MessageCommand* as described in the communication model in chapter 4.1.2.3 on page 74 and distinguishes its three tasks depending on the read message type. The *MessageCommand*, called *code* in figure A.5, may be of the value -1. This means that the underlying input stream has been closed so the connection has to be closed as there is no more data to read available due to the closed input stream. If the code equals one of the message types for `COMMAND`, `REQUEST` or `SEND` (similarly to the `RoutingReader` this information is stored in the upper four bits of the one byte long code depicted by *upperCode*), the according method to evaluate the following message is called. If the code is anything other than that, the `MessageReader` skips the evaluation of this code in analogy to the `RoutingReader`.

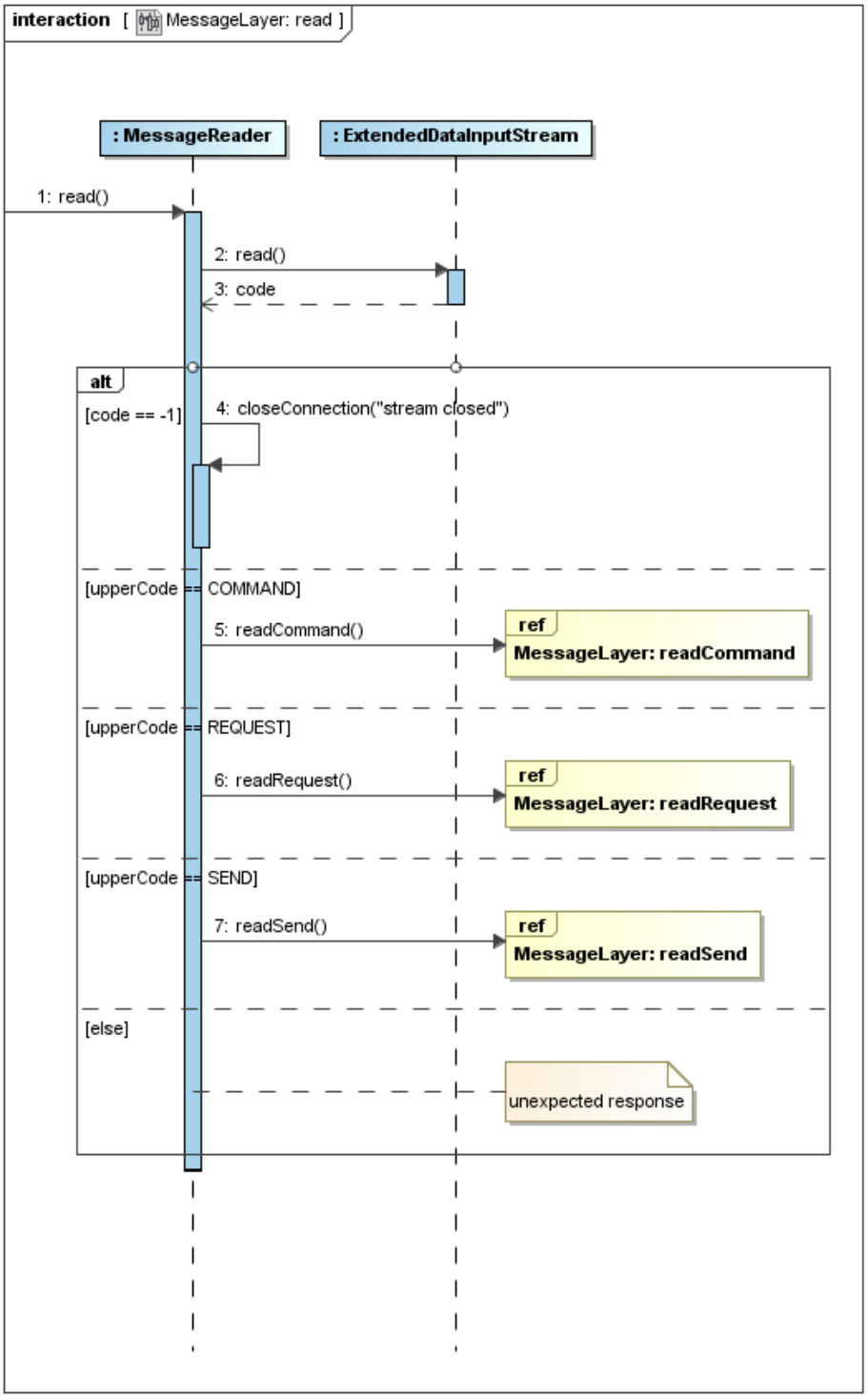

Figure A.5
The main entry point of the Message Layer.

The negotiation of closing the virtual connection in figure A.6 is almost the same like for the physical connection. The only difference is that on the Message Layer a case exists where to decline a request for close. That is if the device transmits a file in parts, but not all parts are transmitted when the close request arrives.

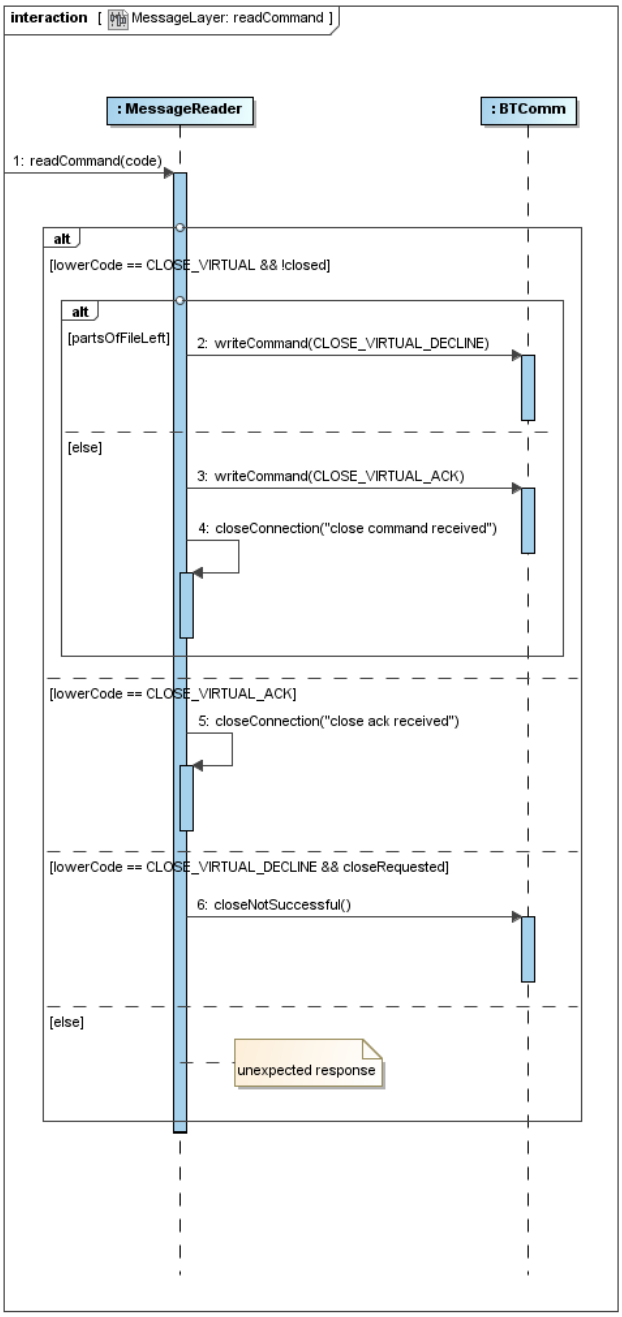

Figure A.6
Protocol for negotiating a close of the physical connection

The decryption of a request is straight forward as can be seen in figure A.7. The information about the content type is stored in the four lower bits of the one byte long code (depicted by *lowerCode*). Depending on the value of the lowerCode the `MessageReader` writes a light value, compass data or a file in response to the request. Only while receiving a request for a file, the `MessageReader` has to read the following bytes to get the name of the requested file. If the lowerCode does not equal one of the three content types, the `MessageReader` skips the evaluation.

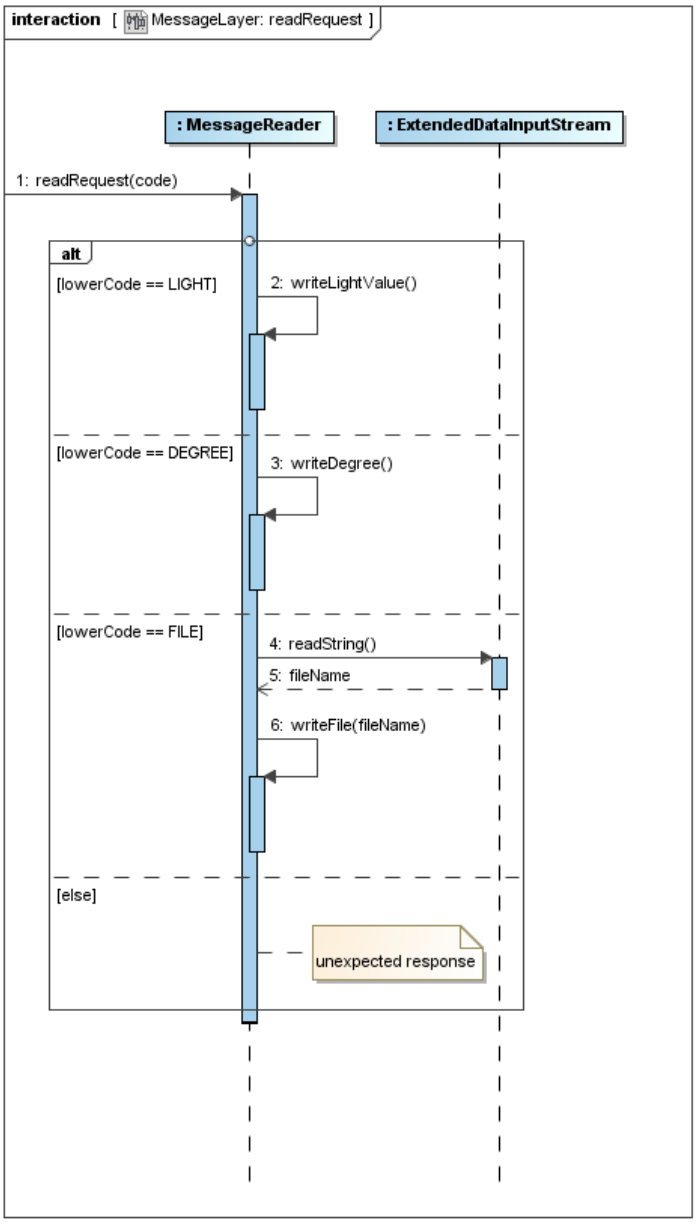

Figure A.7
Evaluation and response to a request.

As illustrated in figure A.8, a response to a previous request is evaluated in the same way as the request itself. According to the lowerCode, the `MessageReader` switches between the reception of a light value, compass data or a file. After receiving a light value or compass data, the `MessageReader` immediately informs all observers that there is new data available. However, the transmission of a file is handled special because the file might be transmitted in parts. The `MessageReader` reads a file in the method `readFile`. There it is distinguished whether an empty file, the whole file or only a part of a file was sent. For an empty file or the whole file, the observers are immediately notified about the reception. While receiving a file in parts, the `MessageReader` waits for the notification until all parts have been gained.

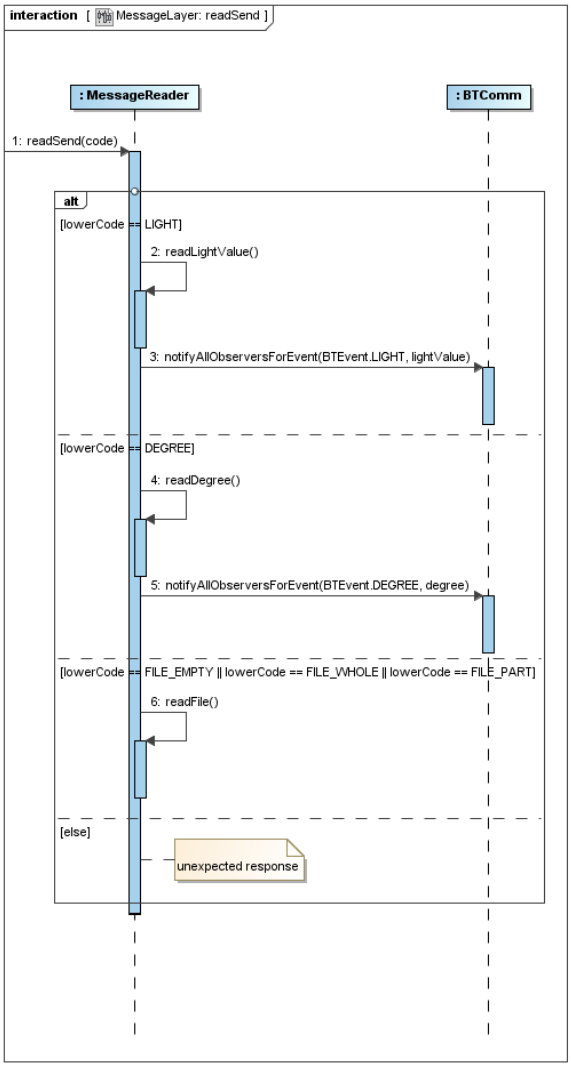

Figure A.8
Reception of data.

Appendix B

Robotic models in Lego Digital Designer

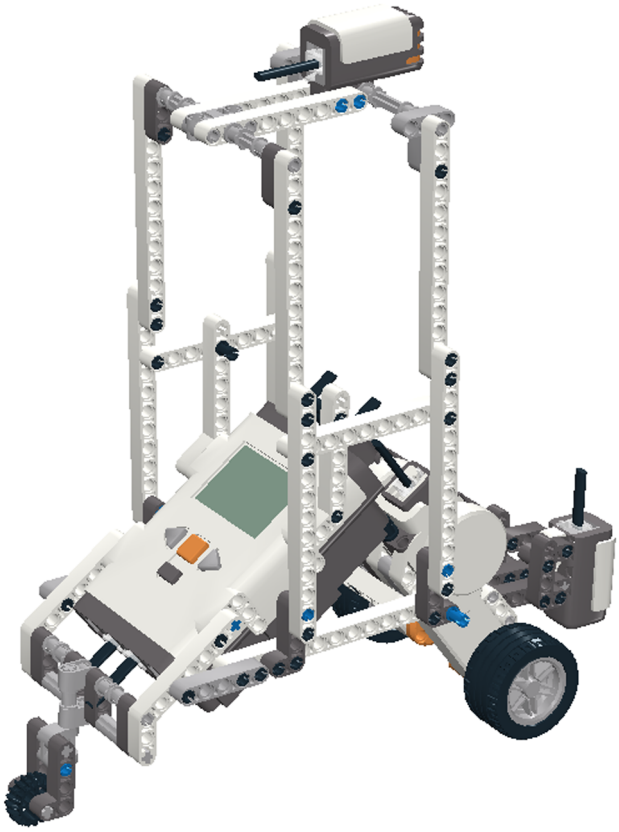

Figure B.1
Trike model designed in Lego Digital Designer

Figure B.2
Distributed line follower designed in LEGO DIGITAL DESIGNER

Bibliography

[ÅH05] Åström, Karl J., Hägglund, Tore: *Advanced PID Control*. ISA - The Instrumentation, Systems, and Automation Society, 2005

[ÅM10] Åström, Karl J., Murray, Richard M.: *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2010. – http://www.cds.caltech.edu/~murray/books/AM08/pdf/am08-complete_04Mar10.pdf[23.11.2010]

[AFF07] Astolfo, Dave, Ferrari, Mario, Ferrari, Giulio: *Building Robots with LEGO MINDSTORMS NXT*. Syngress Publishing, Inc., 2007. – http://issuu.com/mestika/docs/lego_mindstorms_nxt[23.11.2010]

[Ark98] Arkin, Ronald C.: *Behavior-based Robotics*. 1st. Cambridge, Massachusetts, USA : MIT Press, 1998

[Bag07] Bagnall, Brian: *MAXIMUM LEGO NXT, Building Robots with Java Brains*. Winnipeg, Manitoba, Canada : VARIANT PRESS, Winnipeg, Manitoba, 2007

[BDT99] Bonabeau, Eric, Dorigo, Marco, Theraulaz, Guy: *Swarm Intelligence: From Natural to Artificial Systems*. New York, USA : Oxford University Press, 1999

[BEF96] Borenstein, J., Everett, H.R., Feng, L.: *"Where am I?": Sensors and Methods for Mobile Robot Positioning*. University of Michigan, Michigan, 1996. – http://www-personal.umich.edu/~johannb/Papers/pos96rep.pdf[23.11.2010]

[Ben05] Beni, Gerardo: From Swarm Intelligence to Swarm Robotics. In: Şahin, Erol (Hrsg.), Spears, William M. (Hrsg.): *Swarm Robotics* Bd. 3342. Springer Berlin / Heidelberg, 2005, p. 1–9

[BL83] Brooks, Rodney A., Lozano-Pérez, Tomás: A subdivision algorithm in configuration space for findpath with rotation. In: *Proceedings of the Eighth international joint conference on Artificial intel-*

ligence - Volume 2. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1983, p. 799–806

[BL+10] Bonani, M., Longchamp, V. et al.: The marXbot, a Miniature Mobile Robot Opening new Perspectives for the Collective-robotic Research. In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2010)*, IEEE Press, 2010. – http://infoscience.epfl.ch/record/149974/files/marxbot-iros2010.pdf[23.11.2010]

[Bom07] Bombös, Lars: *LEGO MINDSTORMS NXT, Ein Überblick*, Hochschule Karlsruhe Technik und Wirtschaft, Semesterarbeit, 2007. – http://ls12-www.cs.tu-dortmund.de/teaching/courses/ws0708/es/uebungen/download/Mindstorms_Bericht.pdf[23.11.2010]

[Bro86] Brooks, R.A.: A Robust Layered Control System for a Mobile Robot. In: *IEEE Journal of Robotics and Automation* RA-2 (1986), Nr. 1, p. 14–23. – http://people.csail.mit.edu/brooks/papers/AIM-864.pdf[23.11.2010]

[Chi09] Chikamasa, Takashi: *nxtOSEK*. Website, 2009. – http://lejos-osek.sourceforge.net[23.11.2010]

[CNB+06] Campo, Alexandre, Nouyan, Shervin, Birattari, Mauro, Gross, Roderich, Dorigo, Marco: Negotiation of Goal Direction for Cooperative Transport. In: *Ant Colony Optimization and Swarm Intelligence: 5th International Workshop, ANTS 2006* Bd. 4150. Berlin, Germany : Springer-Verlag, 2006, p. 191–202

[COD07] CODATEX Hainzlmaier GmbH & Co.KG: *RF ID Sensor for LEGO MINDSTORMS NXT*. First Edition. Salzburg, 2007. – http://www.codatex.com/picture/upload/image/Short_Manual_Web.pdf[23.11.2010]

[DB06a] Durrant-Whyte, Hugh, Bailey, Tim: Simultaneous Localisation and Mapping (SLAM): Part I The Essential Algorithms. In: *IEEE Robotics and Automation Magazine* 2 (2006). – http://www-personal.acfr.usyd.edu.au/tbailey/papers/slamtute1.pdf[23.11.2010]

[DB06b] Durrant-Whyte, Hugh, Bailey, Tim: Simultaneous Localisation and Mapping (SLAM): Part II State of the Art. In: *IEEE Robotics and Automation Magazine* 2 (2006). – http://www-personal.acfr.usyd.edu.au/tbailey/papers/slamtute2original.pdf[23.11.2010]

[DB+94] Dickmanns, Ernst D., Behringer, R. et al.: The Seeing Passenger Car 'Va-MoRs-P'. In: *Intelligent Vehicles Symposium '94* (1994)

[DS04a] Dorigo, Marco, Sahin, Erol: Guest Editorial. In: *Autonomous Robots* 17 (2004), p. 111–113

[DS04b] Dorigo, Marco, Stützle, Thomas: *Ant Colony Optimization*. Cambridge, Massachusetts : The MIT Press, 2004

[Dur99] *Chapter* Distributed problem solving and planning. In: Durfee, Edmund H.: *Multiagent systems*. Cambridge, MA, USA : MIT Press, 1999, p. 121–164

[ECM+00] Estier, T., Crausaz, Y., Merminod, B., Lauria, M., Piguet, R., Siegwart, R.: An innovative Space Rover with Extended Climbing Abilities. In: *In Space and Robotics*, 2000, p. 333–339

[FBBD10] Ferrante, Eliseo, Brambilla, Manuele, Birattari, Mauro, Dorigo, Marco: "Look out!": Socially-Mediated Obstacle Avoidance in Collective Transport. In: *Swarm Intelligence: 7th International Conference, ANTS 2010* Bd. 6234. Berlin, Germany : Springer-Verlag, 2010, p. 572–573

[FBDT99] Fox, D., Burgard, W., Dellaert, F., Thrun, S.: Monte Carlo Localization: Efficient Position Estimation for Mobile Robots. In: *Proc. of the National Conference on Artificial Intelligence*, 1999, p. 343–349. – `http://www.cs.washington.edu/ai/Mobile_Robotics/abstracts/sampling-aaai-99.abstract.html`[23.11.2010]

[Fox98] Fox, Dieter: *Markov Localization: A Probabilistic Framework for Mobile Robot Localization and Navigation*, University of Bonn, Germany, PhD-Thesis., 1998. – `http://www.cs.washington.edu/homes/fox/postscripts/fox-thesis.pdf`[23.11.2010]

[GBMP97] Gat, Erann, Bonnasso, R. P., Murphy, Robin, Press, Aaai: On Three-Layer Architectures. In: *Artificial Intelligence and Mobile Robots*, AAAI Press, 1997, p. 195–210

[GC02] Ge, S.S., Cui, Y.J.: Dynamic Motion Planning for Mobile Robots Using Potential Field Method. In: *Autonomous Robots* 13 (2002)

[GD09] Gross, Roderich, Dorigo, Marco: Towards Group Transport by Swarms of Robots. In: *International Journal of Bio-Inspired Computation* 1 (2009), Nr. 1–2, p. 1–13

[HiT09] HiTechnic Division: *HiTechnic, the 1st choice in LEGO certified robotic sensors*. Website, 2009. – `http://www.hitechnic.com`[23.11.2010]

[JA07] Jetic, Aleksandar, Andina de la Fuente, Diego: Swarm intelligence and its applications in swarm robotics. In: *Proceedings of the WSEAS international conferences : 6th WSEAS International Conference on Computational Intelligence, Man-Machine Systems and Cybernetics*, WSEAS, 2007

[Jon04] Jones, Joseph L.: *Robot Programming: A Practical Guide to Behavior-Based Robotics*. New York, USA : McGraw-Hill Companies, Inc., 2004

[Kal60] Kalman, R. E.: A New Approach to Linear Filtering and Prediction Problems. In: *Transactions of ASME – Journal of Basic Engineering* 82 (Series D) (1960)

[KE01] Kennedy, James, Eberhart, Russell C.: *Swarm Intelligence*. San Francisco, USA : Morgan Kaufmann Publishers, 2001

[Lab10] Lablua: *The Programming Language Lua*. Website, 2010. – `http://www.lua.org`[23.11.2010]

[LEG06a] LEGO group: Lego Mindstorms *NXT Bluetooth Developer Kit*. Version 1.00. Billund, 2006. – `http://mindstorms.lego.com/en-us/support/files/default.aspx`[23.11.2010]

[LEG06b] LEGO group: Lego Mindstorms *NXT Hardware Developer Kit*. Version 1.00. Billund, 2006. – `http://mindstorms.lego.com/en-us/support/files/default.aspx`[23.11.2010]

[LEG10a] LEGO group: *About us - Corporate Information*. Website, 2010. – `http://aboutus.lego.com/en-US/default.aspx`[23.11.2010]

[LEG10b] LEGO group: *LEGO Digital Designer: Virtual Building Software*. Website, 2010. – `http://ldd.lego.com/default.aspx`[23.11.2010]

[LEG10c] LEGO group: *LEGO MINDSTORMS: Products*. Website, 2010. – `http://mindstorms.lego.com/en-us/products/default.aspx`[23.11.2010]

[LEG10d] LEGO group: Lego Mindstorms*: What is NXT?* Website, 2010. – `http://mindstorms.lego.com/en-us/whatisnxt/default.aspx`[23.11.2010]

[LPE09] LPE Technische Medien GmbH: Lego Mindstorms *education*. Website, 2009. – `http://www.nxt-in-der-schule.de/lego-mindstorms-education-nxt-system`[23.11.2010]

[LW79] Lozano-Pérez, Tomás, Wesley, Michael A.: An algorithm for planning collision-free paths among polyhedral obstacles. In: *Commun. ACM* 22 (1979)

[Mat92] Mataric, Maja: Behavior-Based Control: Main Properties and Implications. In: *Proceedings, IEEE International Conference on Robotics and Automation, Workshop on Architectures for Intelligent Control Systems*, 1992, p. 46–54

[May90] *Chapter* The Kalman filter: an Introduction to Concepts. In: Maybeck, P.S.: *Autonomous robot vehicles*. New York, USA : Springer-Verlag New York, Inc., 1990

[Min10] Mindsensors.com: *mindsensors.com*. Website, 2010. – `http://www.mindsensors.com`[23.11.2010]

[MTKW03] Montemerlo, Michael, Thrun, Sebastian, Koller, Daphne, Wegbreit, Ben: FastSLAM 2.0: An Improved Particle Filtering Algorithm for Simultaneous Localization and Mapping that Provably Converges. In: *Proceedings of the International Conference on Artificial Intelligence (IJCAI*, 2003, p. 1151–1156. – `http://ai.stanford.edu/~koller/Papers/Montemerlo+al:IJCAI03.pdf`[23.11.2010]

[Nat10] National Instruments Corporation: Lego Mindstorms *NXT - Powered by NI LabVIEW*. Website, 2010. – `http://www.ni.com/academic/mindstorms`[23.11.2010]

[Nil84] Nilsson, Nils J.: Shakey The Robot / AI Center, SRI International. 1984 (323). – Technical report. – `http://www.ai.sri.com/pubs/files/629.pdf`[23.11.2010]

[NXP07] NXP: *UM10204 I^2C-bus specification and user manual*. Rev. 03, 2007. – `http://www.nxp.com/documents/user_manual/UM10204.pdf`[23.11.2010]

[RN03] *Chapter* Robotik. In: Russell, Stuart, Norvig, Peter: *Künstliche Intelligenz: Ein moderner Ansatz*. 2nd edition. Prentice-Hall, Englewood Cliffs, NJ, 2003, p. 1093–1149

[Row79] Rowat, Peter F.: *Representing spatial experience and solving spatial problems in a simulated robot environment*, University of British Columbia, Vancouver, Canada, PhD-Thesis., 1979

[Sah04] Sahin, Erol: Swarm Robotics: From Sources of Inspiration to Domains of Application. In: *Swarm Robotics*, 2004, p. 10–20

[SGBT08] Sahin, Erol, Girgin, Sertan, Bayindir, Levent, Turgut, Ali E.: Swarm Robotics. In: Blum, Christian (Hrsg.), Merkle, Daniel (Hrsg.): *Swarm Intelligence*. Springer Berlin Heidelberg, 2008 (Natural Computing Series), p. 87–100

[SK08] Siciliano, Bruno (Hrsg.), Khatib, Oussama (Hrsg.): *Springer Handbook of Robotics*. Berlin Heidelberg, Germany : Springer, 2008

[SL09] Shoham, Yoav, Leyton-Brown, Kevin: *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. New York, USA : Cambridge University Press, 2009

[SN04] Siegwart, Roland, Nourbakhsh, Illah R.: *Introduction to autonomous mobile robots*. Cambridge, Massachusetts, USA : The MIT Press, 2004

[TD06] Trianni, Vito, Dorigo, Marco: Self-Organisation and Communication in Groups of Simulated and Physical Robots. In: *Biological Cybernetics* 95 (2006), p. 213–231

[Ver10] Vernier Software & Technoloy: *Vernier Software & Technology - Probeware for Science, Technology, Engineering and Math Education*. Website, 2010. – `http://www.vernier.com`[23.11.2010]

[Wal50] Walter, W. G.: AN ELECTRO-MECHANICAL "ANIMAL". In: *Dialectica* 4 (1950)

[Woo09] Wooldridge, Michael: *An Introduction to MultiAgent Systems*. 2nd. Chichester, United Kingdom : John Wiley & Sons Ltd, 2009

[ZN42] Ziegler, J. G., Nichols, N. B.: Optimum settings for automatic controllers. In: *Transactions of ASME* (1942)

List of Figures

Contents of CD

The enclosed CD contains the following content:

- this master thesis in PDF format
- videos of the applications single line-follower, heading follower and distributed line-follower
- the source code for the applications single line-follower, heading follower and distributed line-follower in the Eclipse project *NXTProject*, the source code for the communication with a computer in the Eclipse project *PC-Communications* and the source code of leJOS NXJ in the Eclipse project *classes_current* (developer snapshot from 06.08.2010)