Marco Wallner, BSc

A Robust and Efficient Multi-Robot Framework for Flexible Production

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Priv.-Doz. Dipl.-Ing. Dr.techn. Gerald Steinbauer

Institute of Software Technology

Graz, May 2017

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

Abstract

This thesis addresses the challenges arising with the movement in industry towards flexible and smart production. In such a scenario, fleets of autonomous mobile robots interact with smart machines in a modern production hall. Those robots should perform absolutely autonomous, i.e. they should be able to explore an unknown area without any manual configuration and operate reliable as maintenance should be avoided.

The contribution of this thesis is the analysis and formalization of the task to explore an unknown production hall. With this knowledge, an extensible and robust software architecture is designed and implemented for the special case of the RoboCup Logistics League. For this league, all the needed functionality is implemented to participate and to complete the first phase of that competition. Software was written for this purpose on multiple abstraction levels, e.g. low-level components as way-point navigation, a controller to align in front of a machine, and computer vision tasks for machine detection as well as high-level tasks like an exploration scheduler.

The performance of this scheduler is evaluated and compared to a base-line approach. The results show an improved overall performance due to the scheduling algorithm as well as a reliable overall software stack. This work can therefore be used as a basis for further developments in this area.

Acknowledgements

I would like to express my gratitude to my supervisor Gerald Steinbauer for spending lots of hours on discussing design decisions and algorithms as well as for giving me great feedback for my thesis. He provided great support during my master's thesis and took always the time to answer my questions.

I would also like to thank my team members of our robotics team, Graz Robust and Intelligent Production Systems GRIPS. They supported me with great ideas and intense coding sessions to be able to compete at the RoboCup world championship together.

Furthermore, I want to thank my family and all of my friends for supporting me to achieve my desired goals.

Marco Wallner
Graz, May 2017

Contents

List of Figures

Listings

List of Tables

Acronyms

AMCL **A**daptive **M**onte **C**arlo **L**ocalization. 25

AR **A**ugmented **R**eality. 27, 28, 47, 65, 73

BDI **B**elief **D**esire **I**ntention. 36, 37, 48, 74

BS **B**ase **S**tation. 6, 8, 75

CLIPS **C** **L**anguage **I**ntegrated **P**roduction **S**ystem. 9, 26

CS **C**ap **S**tation. 6, 8, 64, 75

DAO **D**ata **A**ccess **O**bject. 84

DCOP **D**istributed **C**onstraint **O**ptimization **P**roblem. 23

DS **D**elivery **S**tation. 6, 8

DWA **D**ynamic **W**indow **A**pproach. 44

EC **E**valuation **C**riteria. 89

FF-NN **F**eed-**F**orward **N**eural **N**etwork. 44, 70

HOG **H**istogram of **O**riented **G**radients. 44, 70

IA **I**nstantaneous **A**ssignment. 19, 20

IP **I**nternet **P**rotocol. 18

ISO **I**nternational **S**tandardization **O**rganization. 18, 48

LA-DCOP **L**ow-Communication **A**pproximation **D**istributed **C**onstraint **O**ptimization **P**roblem. 23

LiDAR **Li**ght **D**etection **A**nd **R**anging. 25, 27, 35, 44

MPS **M**odular **P**roduction **S**ystem. 4, 6, 8, 35, 64, 68

MR **M**ulti-task **R**obot. 19–21

MRTA **M**ulti **R**obot **T**ask **A**llocation. 19, 23

MSB **M**ost **S**ignificant **B**it. 30

MT **M**ulti-robot **T**ask. 19

NP-hard **N**on-deterministic **P**olynomial-time hard. 23

OSI **O**pen **S**ystems **I**nterconnection. 18, 48

PRS **P**rocedural **R**easoning **S**ystem. xi, 36–41, 49, 74

ROI **R**egion **O**f **I**nterest. 27, 44, 45, 70

ROS **R**obot **O**perating **S**ystem. xii, 25, 28, 41, 43, 47–49, 64, 66, 70, 72–74

RS **R**ing **S**tation. 6, 8, 64

SIFT **S**cale-**I**nvariant **F**eature **T**ransform. 44

SP **S**ynchronization and **P**redicence. 21

SQL **S**tructured **Q**uery **L**anguage. 79, 90

SR **S**ingle-task **R**obot. 19–21

ST **S**ingle-robot **T**ask. 19–21

SVM **S**upport **V**ector **M**achine. 44

TA **T**ime-extended **A**ssignment. 19–21

TCP **T**ransmission **C**ontrol **P**rotocol. 18, 24, 79, 99

TW **T**ime **W**indow. 21

UDP **U**ser **D**atagram **P**rotocol. 24

Chapter 1

Introduction

The industry as we know it is changing dramatically as it has already done before several times as described by Jazdi et al. in [1]. The drastic evolution in technology, the digitalization of the world, as well as the trend to produce customized products, enforced companies to keep their production lines as flexible as possible. Furthermore, optimization of the production chain can be achieved as each and every part gathers information about itself, as well as its environment. It is a huge challenge to handle all this information as well as the dynamics introduced by this new flexibility.

1.1 Motivation

One strategy to face the challenges arising with this new requirements, is to use autonomous, mobile robots operating in a factory floor with autonomous machines, capable of performing specific production steps. To keep this approach flexible and reliable, each robot should be able to explore its environment to avoid a manual setup and parametrization. To accomplish this complex task, a robust and efficient software framework is needed to enable these mobile robots to operate autonomously in this dynamic and non-deterministic environment. To provide a testbed for this new technology, and to allow comparison between different approaches, the "Logistics League" was introduced as part of the annual RoboCup world championship (see Chapter 2). Therefore, a software framework (see Chapter 6) was developed to participate in this league, simulating the next industrial revolution and smart production.

1.1.1 Industry 4.0

With the mechanization of production using steam engines in the late 18th century, the first industrial revolution took place. The change to mass production and assembly lines using electricity is known as the second industrial revolution. The utilization of robots for mass production is called the third industrial revolution. Today, the rise of flexible production systems, cyber-physical systems, the internet of things, and cloud computing heralds the start of the fourth

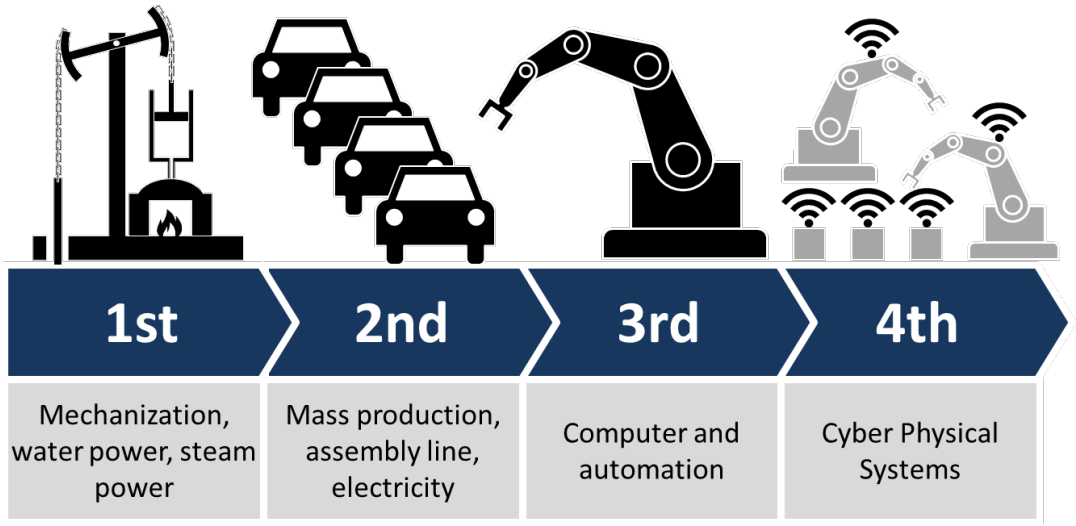


FIGURE 1.1: The four industrial revolutions depicted with their main driving forces.

revolution, also known as "Industrie 4.0" [2]. This steps are summarized in Figure 1.1 by Roser[1]. The German term "Industrie 4.0" (industry 4.0) is used in most of the cases, as the German government initiated and promoted this revolution. This change in the industry is still in progress, and it is necessary to be prepared as Draht et. al. [3] describes this revolution as:

> "However, Industrie 4.0 is a phenomenon that will come inevitably, whether we want it or not."

With this, the aim of this work is to provide a solution for multiple mobile robots, enabling a flexible production in a modern production hall.

To enforce this research area, and to offer a testbed for novel approaches, the RoboCup Logistics League [4] was founded as part of the annual international robotics competition RoboCup [5], proposed and founded in 1997. Competitors from all over the world show here the state of the art in robotics in different disciplines. The league is further described in Chapter 2.

1.2 Goals and Challenges

In the following, the pursued goals of this thesis and the met challenges are described. The goal of this work is to provide a flexible and robust software architecture for robots in the industrial area.

For this, several special aspects needs to be considered. First of all, the solution needs to be flexible as each and every problem arising is different and needs different features. Furthermore, each of the software parts needs to be testable individually. The derived solution should therefore be modular and well structured with clearly defined interfaces.

[1] Christoph Roser at AllAboutLean.com - Own work, CC BY-SA 4.0

To prove the practicality of the developed architecture, a concrete implementation needs to be shown. This software has to provide all the functionality to allow a fleet of autonomous robots to explore an unknown factory hall and to be extensible enough to allow easy implementation of further functionality, such as flexible production. This comes up with several challenges, such as an incomplete knowledge about the world, wrong or inaccurate observations, and therefore conflicts within the knowledge-base.

1.3 Contribution

In the context of this thesis, the problems given in the RoboCup Logistic League (see Chapter 2) are analyzed and classified. Furthermore, the problem to explore an unknown factory hall with randomly distributed machines is formalized.

A complete software stack is provided, enabling multiple mobile robots to work together in a modern factory. Therefore, a solution to the problems arising is given, i.e. from scheduling of different task, to low-level issues as the detection of machine lights, and way-point navigation. The necessary prerequisites used are described in Chapter 5 and details of the software architecture can be found in Chapter 6. With this, all the features needed to solve the exploration problem of the RoboCup Logistic League (described in Section 2.5.1) are provided. Furthermore, the architecture is kept modular, making it easy to add further functionality such as an additional scheduler for the production phase (described in Section 2.5.2).

Parts of the developed solution, e.g. an overall architecture and a solution to derive an exploration schedule, was presented at the Austrian Robotics Workshop 2017 and published in [6].

1.4 Outline

This work is structured as follows: The RoboCup Logistic League is presented as a testbed and benchmark for the developed software framework and algorithms in Chapter 2. The problem to explore an unknown space is formalized in Chapter 3, research about similar problems and their solutions is presented in Chapter 4, and the prerequisites in Chapter 5. The developed three-layer software architecture is introduced and described in Chapter 6. Results from experiments using the developed software can be found in Chapter 7. The overall work is concluded in Chapter 8 and a perspective for future developments is given in Chapter 9.

Chapter 2

RoboCup Logistics League

In this chapter, the RoboCup Logistic League is introduces as the benchmark and test-bed for the developed framework. The history of the league and the chosen abstraction level are shown as well as the defined rules.

2.1 Aim of the League

The RoboCup Logistic League was founded to act as an unified benchmark for novel algorithms in the context of flexible production. To do so, a modern factory hall is mocked up in the philosophy of industry 4.0. This means, that teams of autonomous, mobile robots uses specialized, fixed manufacturing stations to produce customized goods. Randomly arriving orders define which commodity needs to be delivered within which time-window, i.e. no information is given in advance (simulating customized orders from an online shop).

This concept is put into action in this league using multiple **M**odular **P**roduction **S**ystem (MPS) (see Section 2.3) distributed in a factory (the game-field, described in Section 2.4), each capable of performing specific refinement tasks, i.e. process a semi-finished product to a more complex semi-finished product, or a final product.

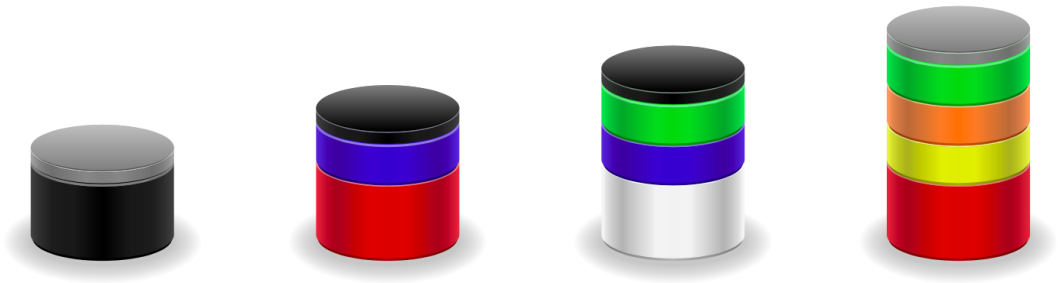


FIGURE 2.1: Products in the RoboCup Logistic League simulating customized products.

2.2 Products

As the refinement steps are not the main concern of this league and to avoid the need of expensive manufacturing units (e.g. lathes, milling machines), the products (can be seen in Figure 2.1 from the official rulebook[1]) and the needed of raw-material is mocked up too.

2.2.1 Raw-Materials

- *Base:* The base is the lowermost element and is available in the three colors: black, red, and silver. The base can be combined with rings and a cap.
- *Ring:* To allow complex products, intermediate rings are introduced. A product can have zero, one, two or three rings. The rings are available in the colors blue, green, yellow, and orange.
- *Cap:* Every product is finished with a cap. These caps are available in the colors gray and black. Caps can be mounted onto bases or rings.

2.2.2 Composite Products

Having these basic elements, products of different complexity can be assembled, classified as C_0, C_1, C_2 and C_3. The simplest one, C_0 consists of a base with a cap only. The next level, C_1 already has an intermediate ring with a defined color. This is the same for C_2 with two intermediate rings and C_3 with three intermediate rings.

2.2.3 Additional Bases

To be able to simulate the need for additional raw material, some production steps require to feed in bases before the actual refinement step can be executed. This complexity is implemented for the mounting of rings, i.e. to mount a ring a defined number of additional bases needs to be provided in advance. How many of these bases for which color is defined per game in a random fashion by the referee box described in Section 2.5.3.

These additional production steps are then referenced as CC_0, CC_1, CC_2 with no extra base, one and two additional bases needed for a ring to be mounted, respectively.

[1] RoboCup Logistic rulebook at robocup-logistic.org/rules

2.3 Modular Production Systems (MPS)

To assemble the products, the mobile robots need to deliver the basic elements and semi-finished products to the so-called MPS. There are four different types of stations capable of doing different refinement steps:

- **B**ase **S**tation (BS): One station per team providing bases in ordered color.
- **R**ing **S**tation (RS): There are two stations per team, each responsible for mounting rings of two distinct colors.
- **C**ap **S**tation (CS): There are also two stations per team. The stations host a shelf with bases with caps. To mount a cap onto a provided base, first one of the bases of the shelf needs to be fed in. The station removes the cap and stores it; the empty base needs to be discarded. After this, the actual base can be entered, and the cap is mounted.
- **D**elivery **S**tation (DS): One station with several slots per team where finished products must be delivered to.

2.4 Gamefield

The production hall is structured in 24 zones, named "Zone #1" to "Zone #24" as it can be seen in Figure 2.2. Each of these fields has a width of 2 m and a height of 1.5 m. Therefore, the whole game field without insertion area is 12 m×6 m in size. The game field is surrounded by a border (drawn in blue) with several openings. In every zone there can be one production system at most. They are distributed randomly with some restrictions. In general, the area is separated into two halves, one per team, where the corresponding machines are located. To add the need to consider the robots of the other team too, i.e. to do collision avoidance and dynamic path planning, two of the machines (chosen randomly) are located at the space of the opponent team. To guarantee equal conditions for both teams, the machines are positioned in a mirrored fashion. In the example drawn in Figure 2.2, the first ring station (RS1) and the first cap station (CS1) are swapped.

2.5 Rules and Gameplay

In every game, two teams compete at the same time on a shared field. Each team gets six production machines to work with. The gameplay is mainly separated into two phases, the exploration, and the production phase.

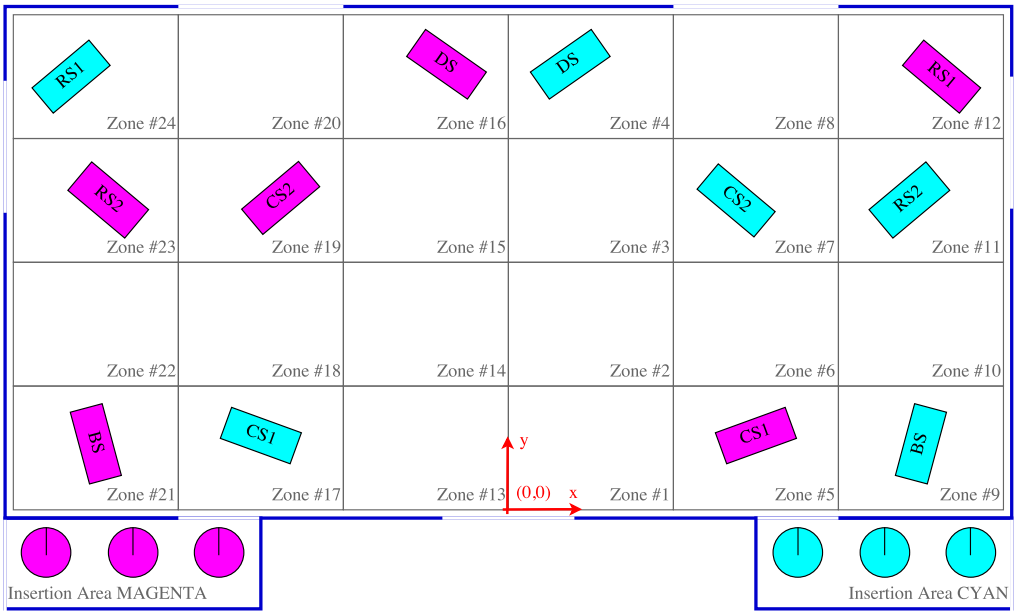


FIGURE 2.2: Gamefield in the RoboCup Logistics League with randomly distributed production machines, insertion area for both teams, zone naming and origin of the used coordination system.

2.5.1 Exploration Phase

In the first phase of the competition, the robots have no information about the factory floor. This emphasizes the idea of fully flexible production, i.e. to put a robot into a new hall and the robot should start to explore this hall, find the machines, and start producing products fully autonomously. This is simulated in this league with the first, so-called, "Exploration Phase". The robots spread out and try to find the machines. To reward only correct detected machines, every machine lights up an unique random pattern. This pattern as well as the zone the machine is in, has to be reported to get points. Wrong reports are penalized. A detailed scoring scheme corresponding to the official rulebook for the RoboCup Logistics League 2016 can be seen in Table 2.1.

Reported	**Correct**	**Points**
Zone	yes	+3
	no	-2
Light Pattern	yes	+5
	no	-4
Total Points	At most 48 points (reporting all 6 machines correctly). At least 0 points (no negative points for exploration phase).	

TABLE 2.1: Rewarded points during the exploration phase.

2.5.2 Production Phase

In the second phase of the game, the robots get orders to complete. These orders have a dedicated earliest delivery time and a deadline, i.e. the products need to be delivered in a specified time-window. The products to produce are of different complexity, and therefore a different number of points can be gained for producing them. A detailed scoring scheme corresponding to the official rulebook for the RoboCup Logistics League 2016 can be seen in Table 2.2. As it can be seen in the scoring, manufacturing a product consists of several steps.

As an example, the production of a C_1 product with a ring demanding one additional base is listed:

1. Prepare the BS to output an arbitrary base.
2. Take the base and put it into the slide of the RS as an additional base.
3. Prepare the BS to output a base in the demanded color.
4. Prepare the RS to mount a cap in the demanded color onto a base.
5. Take the base from the BS and input it into the RS.
6. Prepare the CS to mount a cap onto a base.
7. Take the base with a cap of correct color from the shelf at the BS.
8. Input the base with the cap into the CS.
9. Remove and dispose the base without cap from the CS.
10. Take the base with ring from the RS.
11. Input the base with the ring in CS.
12. Take the base with the ring and cap from the CS.
13. Prepare the DS with the demanded delivery-slot.
14. Input the product in the DS.

As it can be seen, even the manufacturing of a simple C_1 product is rather complex and involves several steps. This complexity is further increased as the MPS goes out of order if it is used in a wrong way (i.e. wrong preparation). Additionally, two of the machines go out of order at random time-points for a random amount of time during the game.

2.5.3 Referee Box

The referee box controls the overall game, illustrates the game state, monitors the feedback from the robots and generates the random orders. A screenshot of the referee box durring the setup phase can be seen in Figure 2.3.

This automated control system for robotic competitions is described in [7]. The goal of this software solution is to have no or as little as possible need of human

Completed Sub-Task	**Description**	**Points**
Additional base	Additional base was fed into the ring station.	+2
Finish CC_0 step	Prepare the ring station for a color requiring no additional bases.	+5
Finish CC_1 step	Prepare the ring station for a color requiring one additional base.	+10
Finish CC_2 step	Prepare the ring station for a color requiring two additional bases.	+20
Finish C_1 pre-cap	Last ring of a C_1 product mounted.	+10
Finish C_2 pre-cap	Last ring of a C_2 product mounted.	+30
Finish C_3 pre-cap	Last ring of a C_3 product mounted.	+80
Mount cap	Mount the cap on a ordered product.	+10
Delayed delivery	Delivering an order within 10 seconds after the deadline leads to an reduced score. Defining the end of the delivering time-slot as T_e and the actual delivery time t, the score S is given as $S = \lfloor 15 - \lfloor t - T_e \rfloor \cdot 1.5 + 5 \rfloor$	up to +20
Late delivery	Delivering an order more than 10 seconds after the deadline	+5
Wrong delivery	Deliver a final product to the designated loading zone out of the requested time range or after all products requested in the period have already been delivered.	+1
False delivery	Deliver an intermediate product.	0
Obstruction penalty	Deliver a workpiece to a machine of the other team.	-20

TABLE 2.2: Rewarded points during the production phase.

interaction in such automated runs. The rules of the game, domain specific facts, and the scoring scheme are defined as **C L**anguage **I**ntegrated **P**roduction **S**ystem (CLIPS) rule set allowing easy adoption for new game rules and even other leagues.

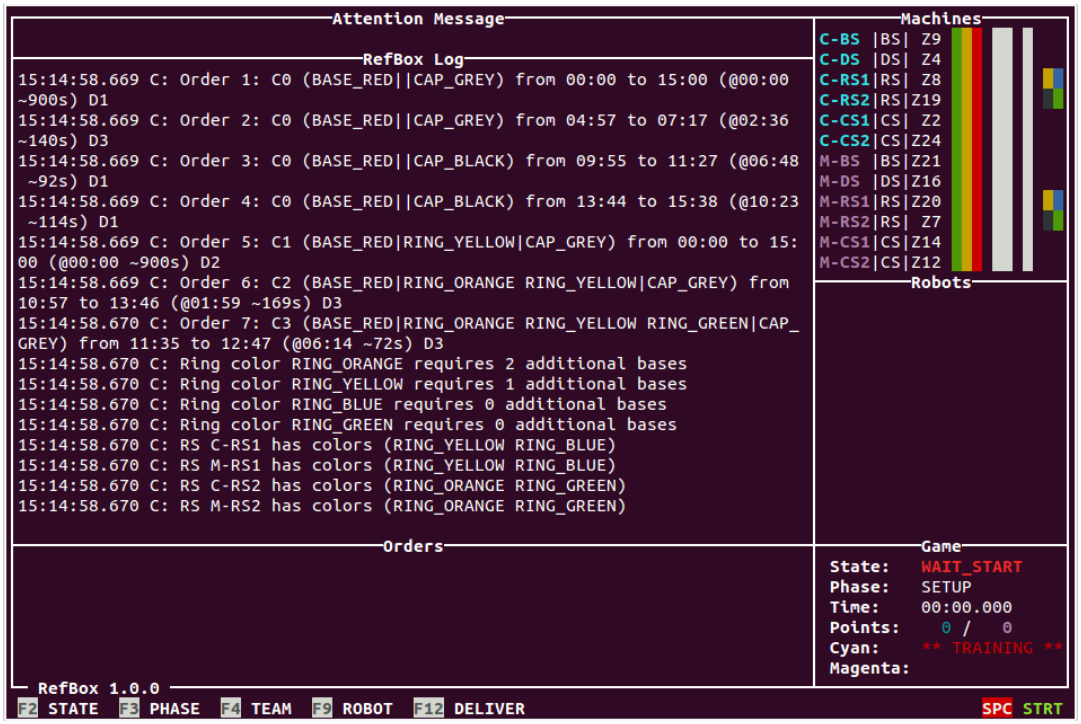


FIGURE 2.3: Screenshot of the Referee Box during the setup phase.

Chapter 3

Problem Formalization

In this chapter, the problem to coordinate a fleet of robots in order to explore an unknown space is formalized as a task allocation problem. To do this, the different components need to be defined, such as the team of robots, the game-field and its zones, the machines to discover, as well as the observations made by the robots. With this, the exploration of the game-field can be described in a formal way.

3.1 Team of Robots, the Gamefield, and Zones

The team of N ($N = 3$ in this particular league) robots is denoted as the set

$$R := \{r_1, r_2, \ldots, r_N\}. \tag{3.1}$$

The gamefield can be seen as an undirected, edge-weighted, planar graph $\mathcal{G}$. The graph contains M ($M = 24$ in this particular league) vertices. The set

$$Z := \{z_1, z_2, \ldots, z_M\} \tag{3.2}$$

is representing the zones. The edges between vertices are defined as the set

$$E = \{\langle z_i, z_j \rangle | z_i, z_j \in Z\}. \tag{3.3}$$

To model a connectivity in terms of a Von Neumann neighborhood with range $r = 1$, a weighting function for each edge is defined as

$$w : E \to W. \tag{3.4}$$

As the graph is undirected, for two vertices z_i and z_j, it holds that $e_{i,j} = e_{j,i}$.

With this, the graph is defined as

$$\mathcal{G} := (Z, E, w). \tag{3.5}$$

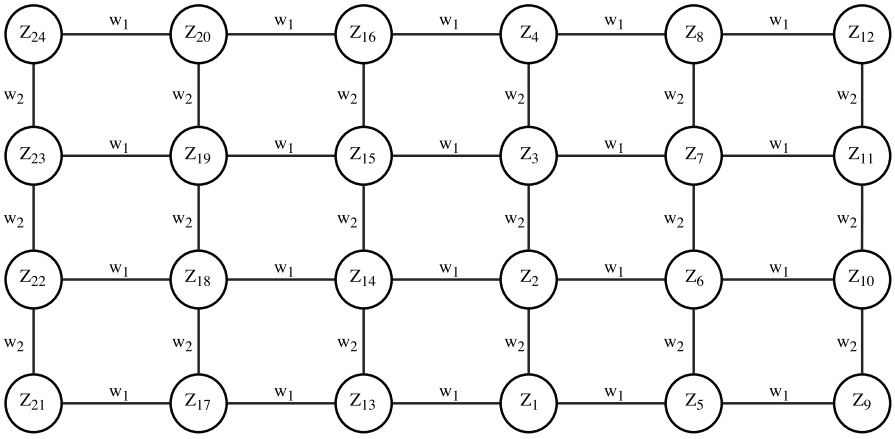


FIGURE 3.1: Gamefield modeled as undirected, edge-weighted, planar graph (finite weighted edges only).

Due to the geometric properties of the game-field (see Figure 2.2), it is sufficient to define two distinct weights

$$W := \{w_1, w_2\}, \quad w_1, w_2 \in \mathbb{R} \tag{3.6}$$

for left/right and up/down respectively. This models the invariance of the cost to move between cells, i.e. moving w.l.o.g. horizontal from one zone to another costs the same for all tuples of neighboring cells, from e.g. Z_{20} to Z_{16} costs as much as for moving from Z_{14} to Z_{18}. Such a graph can be seen in Figure 3.1.

To define the weighting function keeping the indexing of the real game-field (and therefore the relation to that game-field), it is necessary to define the weighting function step-wise as

$$w\left(e_{i,j}\right) = w\left(e_{j,i}\right) = \begin{cases} w_1, & \forall i \in \{1, 2, ..., 11\}, \forall j \in \mathbb{N} : j = i + 1 \\ w_1, & \forall i \in \{13, 14, ..., 23\}, \forall j \in \mathbb{N} : j = i + 1 \\ w_1, & \forall i \in \{1, 2, 3, 4\}, \forall j \in \mathbb{N} : j = i + 12 \\ w_2, & \forall i \in \{1, 2, ..., 8\}, \forall j \in \mathbb{N} : j = i + 4 \\ w_2, & \forall i \in \{13, 14, ..., 20\}, \forall j \in \mathbb{N} : j = i + 4 \\ \infty, & \text{otherwise.} \end{cases} \tag{3.7}$$

Having this, the estimated cost to move from one zone to another can be defined as the accumulated weight over the shortest path $s_{i,j}$ between the two vertices z_i and z_j representing these zones. Such a path is defined as the sequence of edges traversed, i.e.

$$s_{i,j} := \{\langle z_i^0, z_j^0 \rangle, \langle z_i^1, z_j^2 \rangle, \ldots, \langle z_i^k, z_j^k \rangle\} \in S \tag{3.8}$$

where between all consecutive zones there must be a direct path, i.e. an edge with finite weight. The consecutiveness is ensured by

$$\forall m \in \{1, 2, \dots k\} : z_j^m = z_i^{m+1}. \tag{3.9}$$

These paths performed by the robots are collected in the set of all paths S. The length L of such a path is the sum of all weights of the traversed edges, i.e.

$$L : S \to \mathbb{R} \tag{3.10}$$

$$L(s_{i,j}) = \sum_{m=0}^{k-1} w(\langle z_i^m, z_j^m \rangle). \tag{3.11}$$

The shortest path can be found e.g. using Dijkstra's Algorithm described in [8]. This represents only a theoretical lower bound of the costs, i.e. real navigation introduces additional costs due to e.g. obstacles.

3.2 Machines to Discover

The goal of the exploration phase is to detect all of the K ($K = 6$ in this particular league) machines. During this period, the type of the machine needs to be identified, but for the exploration strategy, the type has no effect in the first place. Therefore, the machines can be denoted as

$$M := \{m_1, m_2, \dots, m_K\}. \tag{3.12}$$

Each of these machines is positioned within one randomly chosen zone. Therefore, the assignment of a machine to a zone can be defined as:

$$a_{i,j} = \begin{cases} 1, & \text{if } m_i \text{ is in Zone } z_j \\ 0, & \text{otherwise.} \end{cases} \tag{3.13}$$

This mapping need to be found during this phase. It is assumed that this is known for Zone z_i as one of the robots visits this zone and reported the presence of a machine, i.e. $a_{i,j}$ is set. This assumption holds as one gets the response from the referee box if a machine to zone mapping is reported. This leads to the following condition for a fully discovered field:

$$\text{explored} \Leftrightarrow \sum_i \sum_j a_{i,j} = K \tag{3.14}$$

3.3 Observations

To motivate the chosen modeling of the observation, the way the observations come in needs to be shown. This is mainly done with the AR tag detection described in Sec. 5.7. Each robot is therefore equipped with a front-facing

camera to discover machines on the field. The distance and angle of this machine is then calculated relatively to the robot and further (with the self-localization of the robot) with respect to a common map. Within this map, the robot can estimate the zone this machine is in. All this information about observed machines are then reported as observations as described in this section.

An observation is a tuple of information

$$o := \langle r, z, p, m, t \rangle, \tag{3.15}$$

where $o.r \in R$ denotes the robot's unique identification number, $o.z \in Z$ the zone the observed machine is in, $o.p \in [0, 1]$ the probability of the robot that its observation is correct, $o.m \in M$ the observed machine and $o.t \in \mathbb{N}_0$ the time-stamp of this observation. All this observations are collected in the sequence of observations

$$\Omega := \langle o_1, o_2, \ldots, o_O \rangle \tag{3.16}$$

for O observations.

These observations underly some rules due to the game rules described in Sec. 2.

First of all, there is a defined set of machines MPS, i.e. both teams share the same six machine types, namely a "Base Station" BS, "Delivery Station" DS, two "Ring Stations" RS_1 and RS_2 as well as two "Cap Stations" CS_1 and CS_2 per team, defined as

$$\text{MPS} := \{BS, DS, RS_1, RS_2, CS_1, CS_2\}. \tag{3.17}$$

The teams T are denoted with cyan and magenta and are therefore defined as

$$T := \{\text{cyan}, \text{magenta}\}. \tag{3.18}$$

With this, the set of machines for this particular league can be defined as

$$M_{\text{logistics}} := \{m_{i,j} | \forall i \in \text{MPS}, \forall j \in T\}. \tag{3.19}$$

The relation of the own machine to the machine of the other team can be defined as the counterpart function

$$\text{counterpart} : M_{\text{logistics}} \to M_{\text{logistics}} \tag{3.20}$$

defined as

$$\text{counterpart}(m_{i,j}) = \begin{cases} m_{i,\text{magenta}}, & \text{if } j = \text{cyan} \\ m_{i,\text{cyan}}, & \text{if } j = \text{magenta} \end{cases} \quad \forall i \in \text{MPS}. \tag{3.21}$$

The second relation describes that the machines are distributed in a mirrored way. For this, the relation of mirroring needs to be defined. Mirroring is made about the y-axis of the game-field described in Sec. 2.4. The mirrored zone is

therefore defined as as the mirrored function

$$\text{mirrored} : Z \to Z \tag{3.22}$$

defined as

$$\text{mirrored}(z_i) = \begin{cases} z_{i+\frac{M}{2}}, & \text{if } i \leq \frac{M}{2} \\ z_{i-\frac{M}{2}}, & \text{otherwise.} \end{cases} \tag{3.23}$$

With these relations, we can derive conditions if the set of observations is valid, i.e. is conflict-free. The set is conflict-free if the following holds:

- each reported machine is reported in the same zone (see Equation 3.24).
- if a machine is reported in a zone z_i, observations in the mirrored zone must only be of the same machine type of the other team (see Equation 3.25).

This can be expressed as the integrity constraints:

$$\forall o', o'' \in \Omega : o'.z = o''.z \Leftrightarrow o'.m = o''.m \tag{3.24}$$

$$\forall o', o'' \in \Omega : o'.z = \text{mirrored}(o''.z) \Leftrightarrow o'.m = \text{counterpart}(o''.m). \tag{3.25}$$

3.4 Belief

To calculate a belief of the gamefield, i.e. a probability describing the machine distribution, zones are treated as discrete random variables (compare to Elfes et al. [9] about occupancy grid maps). Such a zone is therefore defined in a stochastic way as

$$\mathcal{Z} : \Psi \to \mathbb{R}, \tag{3.26}$$

where Ψ is the sample space of $\mathcal{Z}$ defined as

$$\Psi := M_{\text{logistics}} \cup \text{free}. \tag{3.27}$$

With this, the probability that the random variable $\mathcal{Z}$ has a specific state, given the observations, can be written using Bayes' theorem as

$$P(\mathcal{Z}|\Omega) = \frac{P(\mathcal{Z})P(\Omega|\mathcal{Z})}{P(\Omega)}, \tag{3.28}$$

i.e. the product of the probability that the zone has a distinct state times the probability of the observations given this state, normalized by the overall probability of the given observations.

As the observations arrive over time, an iterative calculation is needed. For this, the following can be calculated after the observation o_k has arrived:

$$P(\mathcal{Z}|o_1, \ldots, o_k) = \frac{P(o_k|\mathcal{Z})P(\mathcal{Z}|o_1, \ldots, o_{k-1})}{P(o_k)} \tag{3.29}$$

To define this as a valid probability, the probabilities of the state of a zone needs to sum up to 1 for the complete sample space, i.e.

$$\sum_{\psi \in \Psi} P(\mathcal{Z}|\psi) = 1. \tag{3.30}$$

3.5 Exploration as a Minimization Problem

The task to observe all zones can be defined as a minimization problem. The function declaring a zone to be visited by a robot can be defined using the function

$$\text{at} : (R \times \mathbb{N}_0 \times Z) \to \{0, 1\}. \tag{3.31}$$

defined as

$$\text{at}(r, t, z) = \begin{cases} 1, & \text{if robot } r \text{ is in zone } z \text{ at time } t \\ 0, & \text{otherwise.} \end{cases} \tag{3.32}$$

Having this, the constraint that all the zones needs to be explored (exploration constraint) can be defined as

$$\forall z : \sum_{r \in R} \sum_{t \in \mathbb{N}} \text{at}(r, z, t) \geq 1. \tag{3.33}$$

To allow the description, at which zone $z \in Z$ each robot $r \in R$ is at each time-point $t \in \mathbb{N}_0$, an binary allocation variable $X_{r,t,z} \in 0, 1$ is introduced.

As the movement over the zones needs time, i.e. a robot is for defined time-window within an zone, these slots needs to be described too. For this, it is sufficient to define the end-time for each robot $r \in R$ and each time-slot $t \in \mathbb{N}_0$ as $T_{rt} \in \mathbb{R}^+$.

With this it is possible to describe, that each robot has to be in some zone at each time-point (placement constraint), i.e.

$$\forall r \in R, \forall t \in \mathbb{N}_0 : \sum_{z \in Z} X_{r,t,z} = 1. \tag{3.34}$$

Furthermore, overlapping of these time-windows needs to be prevented for the same zone. This is solved with the following constraint (time-slot consistency constraint):

$$\begin{gathered} \forall z \in Z, \nexists r_1, r_2 \in R : r_1 \neq r_2, \forall t_1, t_2 \in \mathbb{N}_0 : t_1 \neq t_2 : \\ (T_{r_1,t_1-1} < T_{r_2,t_2} < T_{r_1,t_1}) \wedge (T_{r_2,t_2-1} < T_{r_1,t_1-1} < T_{r_2,t_2}) \end{gathered} \tag{3.35}$$

To allow movement between adjacent zones only, the neighborhood needs to be defined as

$$\forall z, z' \in Z : z' = \mathrm{NH}(z) \Leftrightarrow w(z, z') < \infty \vee z = z' \tag{3.36}$$

With this, the movement can be limited using the constraint (movement constraint)

$$\forall r \in R, \forall t \in \mathbb{N}_0, \forall z \in Z : X_{r,t,z} = 1 \Rightarrow X_{r,t+1,\mathrm{NH}(z)} = 1. \tag{3.37}$$

The endpoint of each time-slot needs to be greater or equal to the end of the last time-slot and the additional cost for movement and observation $T_{obs} \in \mathbb{N}_0$. The multiplication is only one for the desired pair of states (in both directions, needs to be divided by two) as:

$$\forall r \in R, \forall t \in \mathbb{N}_0 : T_{r,t+1} \geq T_{r,t} + \frac{1}{2} \sum_{z \in Z} \sum_{z' \in Z} X_{r,t,z} X_{r,t+1,z'} \left(w(z, z') + T_{obs} \right) \tag{3.38}$$

With this, the optimization problem results in finding the assignments $X^{opt}_{r,t,z}$ where the last time-point (with index M_r) of all the robots is minimized.

Therefore, the objective function to be minimized is given as

$$X^{opt}_{r,t,z} = \arg\min_{X_{r,t,z}} \left(\max_{r \in R, t \in \mathbb{N}_0} T_{r,t} \right), \text{ s.t. } \mathbb{C} \tag{3.39}$$

considering the constraints $\mathbb{C}$ defined as the set of

- integrity constraints in Equation 3.24 and Equation 3.25
- exploration constraint in Equation 3.33
- placement constraint in Equation 3.34
- time-slot consistency constraint in Equation 3.35
- movement constraint in Equation 3.37.

To keep the problem solvable, a maximum number of steps need to be defined for t. Therefore, the definition of $t \in \mathbb{N}_0$ is only a theoretical one. To solve this, t needs to be restricted to $t \in \{0, 1, 2, \ldots, t_{max}\} \subseteq \mathbb{N}_0$.

Chapter 4

Related Work

In this chapter, several related approaches are discussed to motivate the design choices taken for our architecture. Therefore, different solutions are analyzed regarding their advantages and disadvantages.

To do this, first of all the general design using multiple layers is described using examples from other fields in computer science. Furthermore, the choice of a central high-level component is motivated. To finalize this section, two other solutions for similar problems are presented and compared.

4.1 Layered Approach

Splitting up complex problems into subproblems is a common strategy in computer science. One specific strategy is to split the problems into different layers of abstraction. This approach can be seen solving a lot of problems, e.g. the communication using **T**ransmission **C**ontrol **P**rotocol (TCP) / **I**nternet **P**rotocol (IP) as a implementation of the **I**nternational **S**tandardization **O**rganization (ISO)-**O**pen **S**ystems **I**nterconnection (OSI) reference model [10].

Here the increasing abstraction is clearly visible from layer to layer. Lower layers support higher layers with functionality. Other layer-based solutions can be found for robotic issues too. Brooks et al. [11] describes a robust solution for robot systems. Here, different layers (called levels) provide different abilities with increasing complexity. `Level 0` alone, for example, is only able to deliver raw information and to perform the lowest level hardware actions. `Level 1` builds upon this and uses these abilities to achieve more complex capabilities. Gat continues with this idea in [12]. In this paper, he describes the "Anatomy" of such a three layer architecture. There these dedicated layers are

- *The Controller:* A reactive feed-back control mechanism.
- *The Sequencer:* A reactive plan execution unit.
- *The Deliberator:* Performing time-consuming deliberation tasks.

Similar abstraction layers can be found in the literature with different naming and slightly different definition, e.g. by Bonasso et al. [13] the three tier architecture (3T) with *skill layer*, *sequencing layer*, and *planning layer*.

A slightly modified version of these approaches is used in this work. The problem is split into three distinct layers. One low-level layer is responsible for the abstraction of the hardware. A mid-level layer using the functionality provided by the lower layer, enabling more complex abilities and first error detection and correction. The highest layer is responsible for coordinating the tasks to multiple robots in an optimal way. Therefore, this layer is different from the other two. The lower two layers are implemented per robot and the highest layer exists only once per system. This structure has the advantage of providing a global view. This avoids the problem of local minima using planning in a greedy fashion per robot. This concept is justified in Section 4.2.2.3 and further described in Section 6.1.

4.2 Task Allocation for Multiple Robots with Temporal Constraints

To tackle the challenge of the unknown environment, multiple robots, time constraints and much more, related work in this area needs to be reviewed. To do this, first of all the problem needs to be classified.

4.2.1 Taxonomy

As Gerkey et al. describes in [14], the widely accepted taxonomy for this kind of problems (**M**ulti **R**obot **T**ask **A**llocation (MRTA)) classifies as follows:

- **S**ingle-task **R**obot (SR) versus **M**ulti-task **R**obot (MR): Is a robot capable to perform multiple tasks simultaneously (MT) or not (ST)?
- **S**ingle-robot **T**ask (ST) versus **M**ulti-robot **T**ask (MT): Is a single robot capable to perform a task (SR) or are more than one robot needed (MR)?
- **I**nstantaneous **A**ssignment (IA) versus **T**ime-extended **A**ssignment (TA): Need the tasks to be assigned instantaneous and no planning for future assignments is possible (IA) or is more information available such as all tasks to assign or at least a model of how tasks will arrive over time (TA)?

With this we can classify the problem of the RoboCup Logistic League.

4.2.1.1 Single-Task Robot versus Multi-Task Robot

To pinpoint the type of robots we have, we need to distinguish between the different game phases described in Section 2.5.1 and Section 2.5.2 for the exploration and production phase respectively.

During the exploration phase, each robot can perform multiple tasks at the same time, e.g. move to a machine to detect the shown light pattern and report

on its way to that machine the pose of the other seen machines. Therefore, the robots act as multi-task robots (MR) in this phase.

During the production phase, each task is atomic and needs the full attention of the robots. Therefore, the robots act as single-task robots (SR) in this phase.

4.2.1.2 Single-Robot Task versus Multi-Robot Task

All the tasks are defined in such a way, that they can be done by one robot only too. This can be also seen as it would be possible to participate with a single robot as well. Therefore, the tasks can be classified as single-robot tasks (SR).

A multi-robot task can appear if more than one robot is necessary to solve a task (e.g. too complex for one agent) or in the case of heterogeneous robots, if different robots are needed to accomplish a task. Both is not the case for our scenario.

4.2.1.3 Instantaneous Assignment versus Time-Extended Assignment

Here it is necessary to distinguish between the production and the exploration phase again.

During the exploration phase, the tasks are inferred from the incomplete knowledge about the environment. With this, at each time-point there can be an arbitrary number of open tasks. E.g. with no further information of the gamefield, there is a task for each zone to be explored. With arriving information, some of these tasks are discarded of refined. Therefore, this assignment of tasks cannot be classified clearly with respect to these definitions but have more in common with the instantaneous assignment (IA) category as the task (explore the gamefield) is known in advance.

During the production phase, the tasks arrive in a randomized fashion with no given model behind. As these tasks are then split into subtasks, the assignment of these (dependent) subtasks could be seen as time-extended assignments. Again, the classification is not clear with respect to these definitions, but have more in common with the time-extended assignment (TA).

4.2.1.4 Classification of our Problem

Summarizing, the problem can be classified with the taxonomy of Gerkey et al. [14] as follows:

- Exploration phase
  - Multi-task robots (MR)
  - Single-robot tasks (ST)
  - Time-extended assignments (IA)

- Production phase
  - Single-task robots (SR)
  - Single-robot tasks (ST)
  - Instantaneous assignments (TA)

With this, similar problems can be found as summarized by Nunes et al. in [15]. Here the problem is further separated if a the tasks have to be completed within a given **T**ime **W**indow (TW) and if there are **S**ynchronization and **P**redicence (SP) relations between the tasks.

With this, several examples can be found for the problem occurring during the exploration phase (MR, ST). These are solved e.g. by Su et al. in [16] and Amador et al. in [17] for problems with given time windows, and by Jones et al. [18] and Tang et al. [19] for problems with synchronization and precedence relations between the tasks. Approaches faces both, TW and SP, can be found in Pujol et al. [20] and Parker et al. [21].

Similar problems to those in the production phase (SR, ST) can be found too for the time-extended assignment type. Here the same separation with TW and SP can be done. Solutions for TW problems can be found in Melvin et al. [22] and Ponda et al. [23], for SP problems McIntire et al. [24] and Luo et al. [25] provide a solution. Approaches for both, TW and SP problems are also given for this domain by Korsah et al. [26] and Gombolay et al. [27].

The problem faced in the RoboCup Logistics League can be classified with respect to several properties as described in [28], too. Summarizing, the following attributes can be found:

- Cooperative and Competitive: The robots work together in a team, but at the same time, they try to achieve more points than the opposing team.
- Partial Observable: Not all information is given, e.g. the position of the robots from the other team is unknown.
- Non-Deterministic: The execution of actions such as gripping may fail. For some of them, there is no a trivial way to model this in a stochastic fashion.
- Sequential: The execution of an action at time-point t_1 influences the state of the whole system at time-point t_2, where $t_2 > t_1$. This means that actions happened in the past may affect the behavior in the future.
- Continuous: As the time-slots to deliver a product or the position of the robot are real-valued numbers, the domain is inherently continuous.
- Known Environment: The behavior of the environment is known, i.e. if a robot feeds a particular semi-finished product into one particular machine, the outcome is known.

- Explicit Time: The time needed to perform a production step is randomized from game to game but constant during one game.
- Infinite Domain: Depending on the chosen granularity, the domain is infinite (continuous time, the position of the robot) or finite by using discretization.

These properties emphasize again the complexity of this league and the issues to be solved. To address this problem in a structured way, different approaches can be found. One of them is to structure the problem using different layers with increasing abstraction as used in this work.

4.2.2 Typical Approaches

The solution for this problems can be separated in a first step by the used control hierarchy, i.e. if a central monitoring and controlling unit is used, or if some kind of swarm or distributed solution is chosen.

4.2.2.1 Centralized Solutions

Using a centralized solution means to have one central point where all the information is collected and where the decision which tasks needs to be performed next is made. This can be realized with a (physically) dedicated additional unit or by choosing one of the robots to handle this tasks besides its main activity.

This approach allows it to generate a global view of the situation and to generate a global optimum scheduling. Furthermore, it allows a separation of the task generation / scheduling and the actual task execution. This benefits come with the downside of a single point of failure, as if the central unit breaks, the whole system is unable to perform further.

An often used approach to find optimal solutions is the Branch-and-Bound algorithm. Here, the state space of possible solutions is represented as a tree. The root of the tree corresponds here to the original problem to be solved and all other nodes are subproblems of the original problem. Defining a bounding function for the costs of each subproblem in the leafs leads then to the possibility to find the optimal solution in a computational efficient way. Modifications of this algorithm such as Branch-and-Cut [29], Branch-and-Price [30], and Branch-Price-and-Cut [31] are further developments of this approach used in the field of vehicle routing problems (which is similar to these problems). These approaches are summarized well by Gini in [32].

Formalization To solve such a problem, it is necessary to formulate it as a constraint satisfaction problem. One example for the field of vehicle routing problems is given in [29]. Here, the problem to find a minimum number of vehicles required to visit a defined set of nodes is presented. The problem is

solved with respect to time window and capacity constraints (similar to our problem).

To apply e.g. the Branch-and-Cut algorithm, the problem needs to be formulated as an integer programing problem. Such a problem maximizes an term

$$\max \mathbf{c}^T \mathbf{x} \tag{4.1}$$

with respect to the constraints

$$\mathbf{A}\mathbf{x} \leq \mathbf{b} \tag{4.2}$$
$$\mathbf{x} \geq \mathbf{0} \tag{4.3}$$
$$\mathbf{x} \in \mathbf{Z}^n. \tag{4.4}$$

Describing a problem as such an integer programing problem is beyond the scope of this thesis (as this only motivates the choice of the software architecture, centralized vs. decentralized). Having such a formalization, the problem can be solved using e.g. the Branch-and-Cut Algorithm.

4.2.2.2 Decentralized Solutions

In the scope of multi robot task allocation, the most used approaches are distributed constraint optimization and marked-based methods. Therefore we will focus on these two in this section.

Distributed Constraint Optimization Problem Based Methods MRTA problems can be formulated as **D**istributed **C**onstraint **O**ptimization **P**roblem (DCOP) as Maheswaran et al. proposes in [33]. This approach has the disadvantage that solving it exactly is in the class of **N**on-deterministic **P**olynomial-time hard (NP-hard) problems and therefore impractical [34]. Therefore, approximated solutions are used, e.g. methods like Max-Sum methods have been used in the RoboCup Rescue League [20], [35], [36] and for task allocation in sensor networks [37]. An advanced version of this algorithm can be found in the work of Ramchurn et al. in [35], called Fast Max-Sum Algorithm.

Another approximation is the **L**ow-Communication **A**pproximation **D**istributed **C**onstraint **O**ptimization **P**roblem (LA-DCOP) presented by Scerri et al. in [38] and Farinelli et al. in [39]. This approach uses some kind of token passing to chose a random agent. This solution tends to guide the search towards a greedy solution.

Marked-Based Methods Within the field of distributed task allocation, so-called marked-based methods are very popular. There are several reasons for that - one of them is, that it can be implemented rather easy and scales very well. The main idea of almost all the concepts is that robots buy tasks and bid with respect to their own cost (e.g. how long it takes them to complete this task, for inhomogeneous teams how appropriate is the robot for this task).

Such sequential auction-based algorithms are widely used, e.g. by Heap et al. [40], Sariel-Talay et al. [41] and Nanjanath et al. [42]. These solutions present robust and well scaling solutions for the distributed task allocation problem.

Gerkey et al. [43] describes one possibility of this approach and calls it MURDOCH. To keep the approach effective and flexible, an anonymous communication scheme is used. This means that all the communication is broadcast-based. This has the advantage that robots can subscribe and unsubscribe for such a broadcast domain without any additional effort for the auctioneer. Furthermore, the broadcast approach is more efficient especially with increasing number of robots.

These auctioning approaches need a lot of communication to be able to come to an equilibrium, i.e. the local optimum solution. To reduce the traffic, so called consensus algorithms have been introduced in [44], [45], [23].

The flexibility introduced with this decentralized algorithms comes with a downside as well. Most of these algorithms operate in a local scope and a greedy fashion. Furthermore, it is hard to derive theoretic performance bounds for these decentralized MRTA problems.

4.2.2.3 Chosen Approach

After analyzing the pros and cons of distributed and centralized approaches, we came to the conclusion to use the centralized one. The downside of the single point of failure could be caught by a stable knowledge-base management, i.e. to keep the state of the world in a persistent and conflict-free manner. In combination with a fault detection system, this single point of failure issue can be compensated rather well.

The downsides of the decentralized approach cannot be handled as good as they have to be in a industrial environment. Here it is necessary to provide theoretical bounds in the future to be able to work on real-time systems. This cannot be handled easily with distributed approaches.

Therefore, the chosen solution is centralized with a global team-server coordinating a fleet of autonomous robots, with all the benefits of being able to accumulate all the informations from all the robots and to keep the possibility of a global optimal scheduling order of the tasks.

Most of the published solutions uses stateless methods (e.g. using **U**ser **D**atagram **P**rotocol (UDP)) to communicate with the agents. As the number of connected robots is low in our scenario, and UDP is not reliable, a dedicated connection to each robot is used instead (i.e. using TCP). This has several advantages as one can be sure that a package is delivered (or a lost connection is reported). The disadvantage of handling all the different connections to the different clients can be neglected as it is rather easy to allow an automatic connection of each robot to the scheduling system. This combines the advantages of both approaches in an optimal way.

4.2.3 Carologistics

The problems solved in this thesis appear in the context of a competition. Therefore, other teams provide solutions for this too. The solution of one of these teams, called Carologistics from Aachen, Germany, is described in this section.

In contrast to our solution, the Carologistics team do not use the ROS as the low-level robot framework (as described in Section 5.4), but Fawkes[1].

4.2.3.1 Fawkes Robot Software Framework

An alternative solution for autonomous robots is presented by the Niemueller et al. [46]. There, the component-based software architecture paradigm is used to keep the system scalable and maintainable.

Component-Based Software Architecture The idea of this component-based software architecture goes back to McIlroy's idea [47] to produce reusable software components. As Collins et al. [48] states, such a system needs to meet four properties:

- A component is a compiled unit of development.
- A component implements (one or more) well-defined interfaces.
- A component provides access to an inter-related set of functionalities.
- A component may have its behavior customized in well-defined manners without access to the source code.

4.2.3.2 Software Architecture

The solution of the Carologistics team is mainly separated into three pieces as described by Niemüller et al. in [49] and is shown in Figure 4.1.

The three layer of the architecture allows a gradual abstraction of the world. The three different layer are present per robot.

Low-Level Components In the lowest layer, so called components are collected providing low-level functionality as hardware driver for the **Li**ght **D**etection **A**nd **R**anging (LiDAR), the interface to the Robotino platform, and vision related functionality. Some of the components in this layer are ported from ROS we are using, e.g. the implemented **A**daptive **M**onte **C**arlo **L**ocalization (AMCL).

[1] www.fawkesrobotics.org

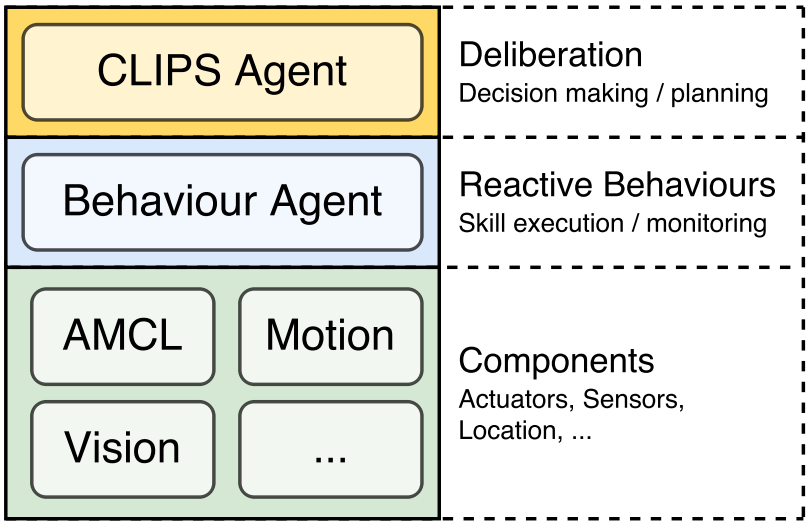


FIGURE 4.1: Software framework of the Carologistics team using component-based software architecture and Fawkes as basis.

Reactive Behaviors The part between the deliberation and the low-level components is designed as a Lua-Based behavior engine. With this, simple and complex so called "Skills" can be modeled similar to a hybrid state machine, where actions are executed in the states and boolean jump conditions denote state changes. Doing so, it acts as a gateway to the low-level hardware functionality, such as e.g. simple way-point navigation or as complex "Skills" as to retrieve a product from a machine.

Deliberation The top-level control, here called deliberation, is implemented using a CLIPS rule engine [50]. This allows an incremental reasoning and therefore to derive at any time-point, for each robot, a local optimal decision. In this point this solution differs from our approach a lot. The lack of a global point of control leads to sub-optimal solutions as each robot performs on its own in a greedy fashion. Furthermore, the advantage of no single point of failure as it is introduced with our high-level team-server comes here into account too, as information about e.g. machine reservation needs to be shared between the robots. For this, one robot is chosen to be some kind of master coordinating these reservations. This is an additional reason for our decision to chose a clear centralized structure, so we have that one (containable) disadvantage, and on the other hand, all the advantages of this central point of knowledge and computational power.

4.2.3.3 Approaches for Common Problems

As the Carologistics team participates in the same league as our team, they have to solve the same problems with their framework. This offers a great opportunity to compare these approaches. Some of them are presented in this section.

Light Pattern Detection The detection of the light-pattern of each machine is solved within multiple steps.

First of all, the robots align in front of the machine at a defined position. Using the LiDAR, it is possible to align in a such a precise way. Having this, and knowing the design (and therefore transformations) of the robot, it is possible to determine the position of the machine light in the camera frame of the mounted camera for this purpose. With this, it is possible to shrink the region of interest a first time.

As a second step, the geometric properties of the lights are used. The information, that there have to be three stacked rectangles, where one is red, one is yellow and one is green is used. Finding the orange one turned out to be hard to detect, so they decided to use the red and green parts only. Combining these geometric constraints with an blob detection in the YUV color space and clustering (here it is necessary to define what is green, yellow, red using a cylinder in this color space) leads to the three **R**egion **O**f **I**nterest (ROI) containing the light.

To decide if a light is on or off, the colors within these regions are compared within the YUV color space again. Here it is necessary to define what "on" and what "off" looks like too.

As it can be seen this requires a lot of configuration and parametrization and depends a lot on the lightning and the robot's setup (e.g. if the position of the camera changes, this have to be updated in the parameters too). All this parameters are not necessary with our solution as we use machine learning to become invariant to different lighting and to find the ROI using the constant gradients of these lights (described in Section 5.6).

Conveyor Belt Detection As the input of the conveyor is rather small, the alignment needs to be very precise to be able to put products into a machine or to retrieve products from a machine.

For this, a detection using three steps is presented. First, the machine is detected using the applied **A**ugmented **R**eality (AR) tag. As this tag is applied arbitrary at the machine, this would introduce the need to calibrate this for each machine and each side.

Therefore, as a second step the mounted laser scanner is used to further increase the precision, but this do not solve the issue for the per-machine calibration as the conveyors can be mounted arbitrary too.

To overcome this issue, a 3D camera (Intel RealSense F200) is used. The gathered point cloud is then used to perform a plane search to detect the input of the conveyor belt.

Chapter 5

Prerequisites

In this chapter components and algorithms we use are described. All the used functionality from existing libraries is described here, this comprises the protocol buffers used for layer abstraction, the simulation environment Gazebo, the procedural reasoning system used in the mid-level, the robotics framework ROS for the low-level, and machine learning approaches for the light pattern detection and the machine detection using AR tags.

5.1 Google Protocol Buffer

Google's Protocol Buffer (short protobuf) is a language- and platform-neutral protocol to serialize structured data. Therefore, it is a similar solution to e.g. XML but it is more lightweight, faster and allows simpler integration. To communicate between different programming languages, messages are defined in a specific syntax. Having this, the protobuf compiler can generate messages (e.g. files with object definitions for the case of C++ or Java) for a variety of programming languages.

5.1.1 Protobuf Syntax

Protobuf messages are defined usually in `.proto` files. For this, several keywords are defined. To declare a new message, the `message` keyword can be used. To explain the syntax, the example in Listing 5.1 is used. In this example, the message `Person` is defined. It contains the required fields `name` and `age`. These fields have to be set, otherwise it is not possible to serialize the message. The optional fields `sex` and `size` may be set. As it can be seen, enumerations can be defined and used too (`Sex`). To allow lists of elements, the `repeated` keyword can be used. With this, it is possible to add a list of persons as children.

The number next to the message entries (e.g. `= 1;`) defines the order of serialization. This should not be confound with the numbers at the enum definition; these numbers define the internal representation of the entries as integers.

Listing 5.1: Example Protobuf message.

```
message Person {
  required string name  = 1;
  required int32  age   = 2;
  optional Sex sex      = 3;
  optional double size  = 4;
  repeated Person child = 5;

  enum Sex {
    MALE   = 1;
    FEMALE = 2;
  }
}
```

5.1.1.1 Datatypes

As it can be seen in the example message, different primitive datatypes can be used. All of these datatypes have their corresponding type in the supported programming languages. An overview from the official Google documentation can be seen in Table 5.1.

Proto	C++	Java	Python	Go
double	double	double	float	*float64
float	float	float	float	*float32
int32	int32	int	int	*int32
int64	int64	int	int	*int64
uint32	uint32	int	int	*uint32
uint64	uint64	int	int	*uint64
sint32	int32	int	int	*int32
sint64	int64	int	int	*int64
fixed32	int32	int	int	*int32
fixed64	int64	int	int	*int64
sfixed32	int32	int	int	sfixed64
int64	int64	int	int	*int64
bool	bool	boolean	bool	*bool
string	string	String	str	*string
bytes	string	ByteString	str	[]string

Table 5.1: Supported protobuf datatypes and corresponding C++, Java, Python and Go type.

As it can be seen, there are several different types to e.g. represent `int64`. They differ in the serialization and encoding of the data as described in Section 5.1.1.2.

5.1.1.2 Encoding

To serialize the data in an efficient way, so called "Varints" are used. These encoding of variables uses one ore more bytes to store an integer. As an arbitrary number of bytes can be used, the end of the stream needs to be declared. For this, the most **M**ost **S**ignificant **B**it (MSB) is used. If this bit is not set, this is the last byte of information to this integer. The remaining seven bits contain the two's complement representation of the number. The arriving bytes are then in a least significant byte first order. This is explained best using an example. To convert the number 301 into a byte stream, first of all it needs to be converted into the binary system:

$$301_{dec} = 100101101_{bin} = 00000100101101_{bin}$$

Having this, packages of seven bits length need to be built as `0000010 0101101`. As the numbers are sent in least significant byte order, the second byte is sent first. To mark that a second byte is following, the MSB is set to one. At the second byte, the MSB is set to zero. The resulting byte stream is then `10101101 00000010`.

As it can be seen, this encoding is very efficient for small positive numbers. Negative numbers on the other hand are very inefficient to encode this way. This type of encoding is therefore used for `int32`, `int64`, `uint32`, and `unit64`.

Signed numbers are treated special. To use this special encoding, the types `sint32` and `sint64` can be used. Here, the so called Zig-Zag encoding is applied. This leads to an encoding, where small (absolute value) numbers result in a small Varint encoded value. This is done by reordering the numbers starting by zero and map in positive and negative direction alternating. This mapping is shown in Table 5.2.

Original Value	Encoded As
0	0
-1	1
1	2
-2	3
2	4
-3	5
3	6
...	...
2147483647	4294967294
-2147483648	4294967295

TABLE 5.2: Zig-Zag encoding for `sint32` and `sint64` protobuf types.

This encoding can also be calculated as `o = (n « 1)^(n » 31)` for `sint32` and as `o = (n « 1)^(n » 63)` for `sint64` (« and » denotes arithmetic shift left and shift right, and `^` denotes an exclusive or).

Listing 5.2: Example for the usage of protobuf with C++.

```
1  #include <string>
2  #include "person.pb.h"
3
4  int main(int argc, char* argv[]) {
5      GOOGLE_PROTOBUF_VERIFY_VERSION;
6
7      example::Person person;
8      person.set_name = "Smith";
9      person.set_age = 42;
10     person.set_sex = example::Person::Sex::MALE;
11
12     example::Person* son = person.add_child;
13     son->set_name = person.get_name();
14     son->set_age = 16;
15     son->set_size = 1.75;
16     son->set_sex = example::Person::Sex::FEMALE;
17
18     google::protobuf::ShutdownProtobufLibrary();
19     return 0;
20 }
```

Additionally to this variable length encoding, fixed length encoding can be forced using `fixed32` and `fixed64`. `float` and `double` types are also encoded with a fixed length of 32 and 64 bits respectively.

`string`, `bytes`, embedded messages and repeated fields are represented using a length-delimiting protocol, i.e. the data is encoded as a Varint representing the length following by the actual data.

5.1.2 Usage of the Serialized Messages

The defined protobuf messages can be used to generate e.g. native C++ or Java code. Having this, these files provide access to these messages in a programming language specific way. As an example, the data access using C++ and Java is shown in this section.

5.1.2.1 Protobuf Messages with C++

The fields of the message can be written and read by provided public interfaces of the generated classes. This can be explained with the example protobuf message given in Listing 5.1.

As it can be seen in Listing 5.2, after the generated header (`person.pb.h`) is included, the protobuf message can be used as an ordinary class. Each of the members can be accessed using the defined setter. In the same fashion, these values can be retrieved again (see line `13`).

Listing 5.3: Example for the usage of protobuf with Java.

```
1  package com.example;
2
3  import com.example.PersonProtos.Person;
4
5  public static void main(String[] args) {
6    Person john = Person.newBuilder()
7                          .setName("Smith")
8                          .setAge(42)
9                          .setSex(Person.Sex.MALE)
10                         .addChild(
11                         Person.newBuilder()
12                               .setName("Smith")
13                               .setAge(16)
14                               .setSize(1.75)
15                               .setSex(Person.Sex.FEMALE) )
16                        .build();
17 }
```

A special case for adding a value is to add an element to a repeated field. To do so, one need to call the `add_` method of this field. This returns the reference to the added field allowing to fill it.

The macro in line `5` ensures that the executable is linked to a version of the protobuf library compatible to the used version in the proto files. The `ShutdownProtobufLibrary()` function deletes all global objects allocated by the protobuf library.

5.1.2.2 Protobuf Messages with Java

The same functionality as shown in C++ is now shown using Java in Listing 5.3.

Here, a slightly different syntax is used, e.g. the Java style camelcase for the getter and setter (e.g. `setName("Smith")` and `getName()`). Furthermore, the builder pattern is used to generate new messages.

These messages can now be transmitted e.g. over the network or between programs written in different (or the same) programming language. How to reconstruct these received messages is described in the next section.

5.1.3 Reconstruction of Protobuf Messages

As protobuf messages are e.g. sent over the network, the received data is a simple stream of bytes. These bytes can be interpreted arbitrarily. To find the correct type (e.g. the sent type), a trick is applied. To do so, an defined enumeration with a pair of values is added to each message. An example of such a message can be found in Listing 5.4.

Listing 5.4: CompType Protobuf enum.

```
enum CompType {
    COMP_ID  = 1000;
    MSG_TYPE =    1;
}
```

The reconstruction of the message is shown using Java. This can be done with all the other programming languages in a similar way too.

To be able to reconstruct a message, it is necessary to register all types of messages you want to receive, i.e. to save the mapping between the enumeration and the Java class. This can be seen in Listing 5.5. This method adds a object derived from `GeneratedMessage` (i.e. all protobuf messages) to the array of keys. Here, the method `getDefaultInstance` of the given protobuf message is invoked. This returns the descriptor of this object. With this, the defined (in the proto file) enum and its values can be found, saved, and used later to e.g. generate an empty protobuf message from the `COMP_ID` and `MSG_TYPE` pair or to interpret a received byte stream by comparing this pair.

To handle a received message, the code shown in Listing 5.6 is used. Here, the raw input stream parsed using the defined data structure of the message:

- 1 byte protocol version
- 1 byte cipher
- 2 bytes reserved
- 4 bytes payload size
- 2 bytes component id (`COMP_ID`)
- 2 bytes message type (`COMP_ID`)

With the `COMP_ID` and the `COMP_ID`, an empty message of the correct type can be generated. Furthermore, the data can be read in using the payload size. This is done in the `parse_input()` function returning this pair of protobuf message and data. With this, the data can be parsed into the corresponding empty message using e.g. using Java's `instanceof` functionality.

5.2 Gazebo

Gazebo is a robotic simulator allowing it to model robots in a virtual world. With this, robots and its surroundings are defined as 3D objects. An physics engine provide realistic interaction between the objects and the world, e.g. simulates collisions and gravity. Sensors can be modeled using this plug-ins too.

To allow the simulation of the given multi-robot scenario in the RoboCup Logistics League, a simulated environment (see Figure 5.1) was developed by Zwilling

Listing 5.5: Register a protobuf message.

```
public <T extends GeneratedMessage>
       void add_message(Class<T> c) {
  try {
    Method m = c.getMethod("getDefaultInstance",
                           (Class<?>[]) null);
    T msg = (T) m.invoke((Object[]) null, (Object[]) null);
    Descriptors.EnumDescriptor desc = msg
                                      .getDescriptorForType()
                                      .findEnumTypeByName
                                       ("CompType");

    int cmp_id = desc.findValueByName("COMP_ID")
                     .getNumber();
    int msg_id = desc.findValueByName("MSG_TYPE")
                     .getNumber();
    Key key = new Key(cmp_id, msg_id);
    _msgs.put(key, msg);
    } catch ...
  }
}
```

Listing 5.6: Reconstruct a received (beacon signal) message.

```
private void handle(@Nullable InputStream in) {
  if (null == in) {
    return;
  }

  try {
    Pair<GeneratedMessage, byte[]> msg = parse_input(in);
    if (null == msg || null == msg.getKey()) {
      received_invalid_packet = true;
      return;
    }

    if(msg.getKey() instanceof
                   BeaconSignalProtos.BeaconSignal) {
      BeaconSignalProtos.BeaconSignal beacon_signal;
      try {
        beacon_signal = BeaconSignalProtos.BeaconSignal
                            .parseFrom(msg.getValue());
        handleBeaconMsg(beacon_signal);
      } catch ...
      else {
        System.err.println("Unkown Message!");
      }
    } catch ...
  }
}
```

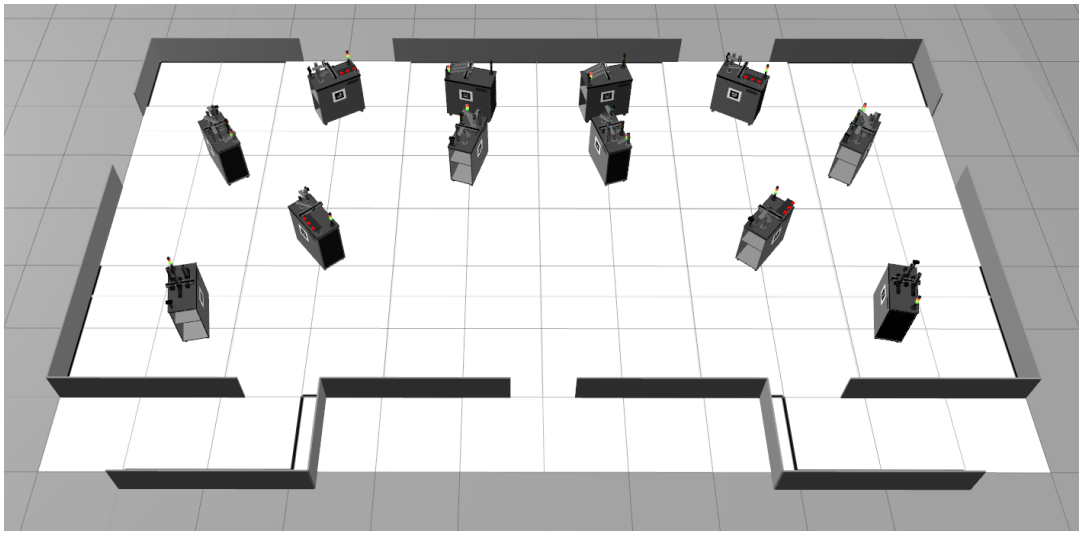


FIGURE 5.1: Simulated arena using Gazebo.

et. al. [51]. In that approach, Gazebo is used to simulate the physical behavior of the robots as well as their surroundings (e.g. MPS).

5.2.1 Layer of Abstraction

The provided simulation contains the official RoboCup Logistic game field and the modular production systems. The functionality of the machines is mocked up using Gazebo plug-ins. With this, the behavior is close to the real world scenario.

We modeled our robots, i.e. the Robotino base with our customized construction with cameras, for this simulator to allow us to get the same sensor readings as the real robot. For this, the cameras and the LiDAR is simulated. With our clear layer structure described in Section 6.1 it is possible to exchange the lowest layer with this simulation of the robots transparently. With this, there is no difference for the middle layer and the team-server between a simulated game and a real game.

5.2.2 Interfaces

To use the simulation and interact with it as with a real robot and to setup the environment, an interface is necessary.

5.2.2.1 Robot Plugins

To interact with the simulation, plugins are used. Those plugins are needed to mock up e.g. detection of the light pattern as the actual pattern is not drawn accurately in the simulation. Here, the ground truth of the actual light pattern

is only returned, if the robot stands in front of the machine in a certain distance, the correct angle and right side of the machine.

5.2.2.2 Referee Box Plugin

To setup the machines for each game in a random way, the so-called "Machine Placement" plugin is used, provided by the league as a library. This ensures a random distribution of the machines conforming to the rules described in 2.4.

5.3 Procedural Reasoning System (PRS)

A PRS can be used to perform complex tasks in a dynamic environment and is based on the belief-desire-intention architecture.

5.3.1 Belief-Desire-Intention (BDI) Architecture

The **B**elief **D**esire **I**ntention (BDI) model is one of the fundamental and widely used approach to design intelligent agents as described in [52]. In this software model, three different data structures are defined, the belief, the desire, and the intention.

5.3.1.1 Belief

To be able to act in an intelligent and reactive way, it is necessary to model the knowledge about the environment. This information is stored and kept up do date as the belief of the agent.

5.3.1.2 Desire

The desires of the agent are the goals to achieve and can, therefore, be seen as the motivation of the agent to perform an action.

5.3.1.3 Intention

The intention is a hierarchical set of plans describing the necessary steps to take to achieve subgoals and therefore to satisfy a desire.

5.3.2 Open PRS

The used implementation of the procedural reasoning system is the open version described in [53]. The main part of this system is the so-called PRS-Kernel

consisting of the database, library of plans and a task graph. Examples of such plans are shown in Section 6.4.

Database The database contains facts of the environment, e.g. the position of the robot and is constantly updated. Therefore, this part of the kernel corresponds to the belief of the BDI software model. Entries in the database are not necessarily static, i.e. they can be concluded as evaluable predicates in an external piece of code (i.e. as a dynamically linked library).

Library of Plans The library of plans contains sequences of actions and tests to achieve particular goals. This part implements the intention of the BDI software model. Open PRS cannot combine plans. Therefore, a plan, or a sequence of sub plans, must exist explicit to achieve a goal successfully.

Task Graph The task graph is a dynamic set of tasks currently executed. This can be seen as the processes running in an operating system.

5.3.2.1 Procedure

A procedure or a plan in the sense of PRS is a data structure used to model information about how to achieve a subgoal or to react to a specific event. The procedure can be split into the following parts.

Body In the body, the steps to perform are defined. This can be done either in a graphical way or LISP-style syntax.

Invocation Condition This defines the goal this procedure may fulfill or the events to which this procedure is meant to react to.

Context The boundary conditions in which this procedure is applicable is defined here.

Documentation The documentation part can be used to add information about the purpose of this procedure in running text.

Effects All facts to conclude to or retract from the database if and only if the plan was successful are collected here.

5.3.2.2 Goals

One of the main design principles using (Open) PRS is the concept of goals. Goals can be seen as a desired state and can be split up in different type.

Achieve The goal to achieve a certain state `S`, written as `(!S)`. This goal is satisfied if `S` is in the database. An example for this would be `(!gripperOpened)`, being successful if and only if the the term `gripperOpened` is in the database or a plan with the invocation condition `gripperOpened` exists and is successful.

Test Test can be used to check for facts `F` in the database, written as `?F`. This goal is satisfied if and only if the fact `F` is in the database. This mechanism can also be used to bind variables, e.g. to get the robot's position from the database, `(?(robotPosition $x $y $theta))` can be used to save the position and angle of the robot in the variables x, y and $theta$ respectively.

Wait This goal can be used to introduce time and to wait for specific state `S`, written as `(^ S)`. With this, the procedure waits until S is in the database or a plan with this invocation part returns successfully. An example would be to wait for the gripper to become ready like `(^ (gripperState ready))`. This goal can never fail and would sleep forever. To avoid this, often timeouts are introduced using the disjunction with an timer, i.e. `(^ (V (gripperState ready) (elapsed_time (time) 60))`, where `V` is the disjunction of the two terms and `(elapsed_time (time) 60)` is successfully after 60 s passed by.

Preserve This goal passively preserves a state `S`, written as `(#S)`. This is mainly used for so-called guarded actions, e.g. to guarantee that the robot's status is running when the gripper should be opened as `(&(!gripperOpened)(#(robotStatus running)))`.

Maintain To actively maintain a state `S` this goal can be used, written as `%S`. An example would be to maintain the battery level above 20 % during reaching the position $(10, 20)$ and the angle of $0°$ as `(&(!(robotPositioned 10, 20, 0)(%(battery_level 0.20)))`. If in this case the maintain goal `(%(battery_level 0.20))` fails, the system interrupts the execution the conjunct goal `(!(robotPositioned 10, 20, 0))` and tries to reestablish the state maintained (actively), i.e. calls a plan to recharge the battery.

Assert This goal simply writes a fact `F` into the database, written as `(=>F)`. This can be used to save several states or error messages in the database, e.g. `(=>(failedAt sendingOverNetwork (time)))` to log errors with the time.

Listing 5.7: Register a function for an open PRS action.

```
make_and_declare_action("SendRobotInfo", sendRobotInfo, 1);
```

Listing 5.8: Implementation of a custom open PRS action.

```
Term* sendRobotInfo(TermList term) {
  unsigned int robot_number;
  if (PUGetOprsParameters(term, 1, INTEGER, &robot_number)) {
    try {
      gateway->sendRobotInfo(robot_number);
    } catch ...
    return build_t();
  }
  return build_nil();
}
```

Retract With this, a fact `F` can be removed from the database, written as `(~ >F)`. This can be used to remove old information from the database, e.g. `(~ (failedAt sendingOverNetwork $x)` to remove all errors occurred during sending over the network.

5.3.3 Interface

To communicate with the open PRS kernel, different interfaces are used. To set a goal, the message passer interface provided by open PRS is used. To communicate with the low level, so-called actions, evaluable predicates, and evaluable functions can be used.

5.3.3.1 Invoke an Action

An user-defined action can be implemented using C++. For this, a name of the action is chosen to be used in open PRS. In our example, the action is called `SendRobotInfo`. To connect this action with a function, it needs to be linked with the corresponding function pointer to the function, the code in Listing 5.7 is used. Here, the name in open PRS, the function pointer and the number of arguments is defined.

Having this, the function is called if the conditions for the action in open PRS are fulfilled. Such a custom action can be seen in Listing 5.8.

Such a function needs to have a specific signature. The return value needs to be of the type `Term*`. With this type, a failed or succeeded state can be returned using the builder `build_nil()` and `build_t()`. Although the function is executed by open PRS in a blocking way, it is possible to return with `wait_id()` and do something in an additional process. If this is done, the function is

Listing 5.9: Register a function for an open PRS evaluable predicate.

```
make_and_declare_eval_pred("waypointExists",
                           waypointExists, 1, TRUE);
```

Listing 5.10: Implementation of a custom open PRS evaluable predicate.

```
PBoolean waypointExists(TermList terms) {
  char* waypoint;
  if (PUGetOprsParameters(terms, 1, ATOM, &waypoint)) {
    return RosActionManager::getInstance()
                            ->checkWaypointExists(waypoint);
  }
  return FALSE;
}
```

called later again until one of the final return values is returned (`built_t` or `built_nil`). This allows introduction of concurrency.

The given arguments have to be of the type `TermList`. This structure contains the given parameters (provided in open PRS). To get these parameters out of this structure, the helper function `PUGetOprsParameters` can be used, giving it the `TermList`, number of arguments, type of this argument (e.g. `INTEGER`) and the location to store this data.

Such an action can therefore only execute an external functionality and return if this succeeded or not. If one needs a customized functionality returning an boolean value, e.g. for a branch in open PRS, a so called evaluable predicate can be used.

5.3.3.2 Evaluable Predicate

User-defined predicates need to be registered in an similar way as actions too. This can be seen in Listing 5.9.

The implementation of this function can be seen in Listing 5.10. Here, the signature is special again. The returning value is now of the type `PBoolean`, i.e. it can be `TRUE` or `FALSE`. This can be seen as the return value. The argument of the function is again a `TermList`, and therefore the elements can be fetched in the same way as in the action case. Here, instead of an integer, a string is given.

If it is not enough to return true or false, i.e. if it is necessary to return e.g. a string, an evaluable function can be used for this.

Listing 5.11: Register a function for an open PRS evaluable function.

```
make_and_declare_eval_funct("getZoneOfMachineTf",
                            getZoneOfMachineTf, 1);
```

Listing 5.12: Implementation of a custom open PRS evaluable function.

```
Term* getZoneOfMachineTf(TermList terms) {
  char* tf_point;
  std::string tf_point_string;
  if(PUGetOprsParameters(terms, 1, ATOM, &tf_point)) {
    tf_point_string = std::string(tf_point);
    return RosActionManager::getInstance()
                            ->getZoneToTF(tf_point_string);
  }
  return build_nil();
}
```

5.3.3.3 Evaluable Function

The registration of an evaluable function can be seen in Listing 5.11 and is similar to the other two types of extensions.

This function needs to have the same structure as an action (as it can be seen in Listing 5.12). The return value type `Term*` can not only handle failed and succeeded, but all the other open PRS types too. As it can be seen, if the code fails, `build_nil()` is returned. Otherwise the return value of the called function is returned. This method can be seen in Listing 5.13. Here (if everything works as expected), an `Term*` is built using `MAKE_OBJECT`. Having this, the type of this structure (`atom` is a string in open PRS) and the value can be set and returned.

5.4 Robot Operating System (ROS)

The software framework used in the lowest layer is ROS [54]. ROS allows comfortable inter-process communication using a publisher/subscriber mechanism as well as the possibility to encapsulate functionality as actions provided by action servers.

5.4.1 Nomenclature

The ROS-Framework introduces several concepts called nodes, topics, messages, and services.

LISTING 5.13: Implementation of the functionality in the evaluable function.

```
Term* RosActionManager::getZoneToTF(std::string tf_name) {
  if(zone_of_tf_->waitForServer(TIME_OUT) == false) {
    return build_nil();
  }
  grips_way_point_navigation_msgs::ZoneOfTFGoal
                                          zone_to_tf_goal;
  zone_to_tf_goal.tf_string = tf_name;
  zone_of_tf_->sendGoal(zone_to_tf_goal);
  bool finished_before_timeout = zone_of_tf_
                                  ->waitForResult(TIME_OUT);
  if (finished_before_timeout) {
    grips_way_point_navigation_msgs
                             ::ZoneOfTFResultConstPtr result;
    result = zone_of_tf_->getResult();
    Term* res = MAKE_OBJECT(Term);
    res->type = ATOM;
    res->u.string = const_cast<char*>(result
                                      ->zone_name.c_str());
    return res;
  }
  return build_nil();
}
```

5.4.1.1 Nodes

Different parts of the software, i.e. modules can be separated using so-called nodes. A node in the view of the operating system is the process that performs the main computation needed for this module. The term node arises from the visualization of processes as nodes and communication paths between processes as edges in a graph.

5.4.1.2 Topics

To enable process intercommunication, so-called topics are introduced. Such a topic is defined by a name, i.e. a string like `map`. Every node is, therefore, able to publish information to these topics and nodes can subscribe such a topic, i.e. a publisher-subscriber mechanism is used. With this, it is possible to provide information asynchronously to (arbitrarily many) other processes.

5.4.1.3 Messages

The communication using the topics is done with strictly typed data structures called messages. These messages can contain primitive types as integers, floating point numbers and booleans as well as arrays of these types, constants and other messages (arbitrarily deep nested).

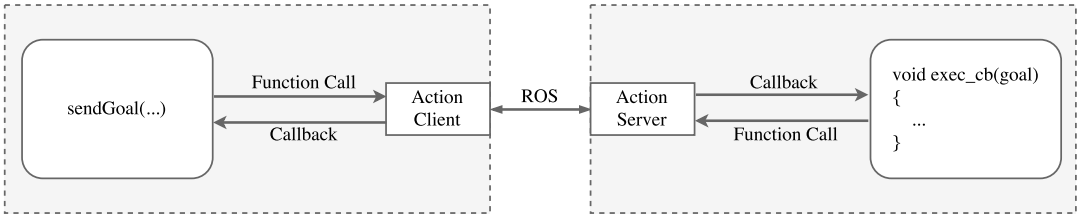


FIGURE 5.2: Client-Server interaction using ROS Actions.

5.4.1.4 Service

To allow synchronous communication scheme as well, services are defined. Such a service is defined by a pair of messages, one for the request and one for the response. The node offering the service offers such a service under an defined name (string). The node can then call the service by sending a request message and wait for the response.

5.4.1.5 actionlib

Using ROS actions, it is possible to model simple functionality as dedicated, preemptable tasks. An example of such an action would be the feature to open or close the gripper or to align in front of a machine. Using a defined message (e.g. action message), a client can initiate an action as e.g. to align in front of a machine by sending such a defined message. The result is sent back again also using defined messages on a subtopic, i.e. `alignment_serverresult`. This functionality is encapsulated within the `actionlib`[1] library.

5.5 Way-Point Navigation

To allow the navigation of the robot to defined way-points, ROS `move_base`[2] is used. This approach uses two distinct strategies to generate a path to a given point, namely a global and a local planner.

5.5.1 Global Planner

The `global_planer`[3] uses the given map of the game-field (i.e. the outer border and the insertion area) to generate a plan to the target point. The generation of such a plan can be achieved using several approaches as e.g. an A* search on the grid map.

This is quite straight forward for our system, as no machine is present in this map and so the planner has to plan in an empty field. This leads in most of

[1] wiki.ros.org/actionlib
[2] wiki.ros.org/move_base
[3] wiki.ros.org/global_planner

the cases to an straight line towards the goal. Reactive behavior have to be implemented therefore using the local planner.

5.5.2 Local Planner

For the local approach, the global path is used as an goal, i.e. to stay on this path and to avoid obstacles. The local planner uses the data from the LiDAR to generate a costmap, i.e. a map of its surroundings with the obstacles. These obstacles are inflated with the radius of the robot to allow planning for a point instead of the robots geometry. With this, different trajectory generating algorithms can be used. For our approach, the **D**ynamic **W**indow **A**pproach (DWA)[4] was used. Here, the costmap is transformed into the velocity space. A detailed description of this algorithm can be found in the work of Fox et al. in [55].

5.6 Vision-Based Object Recognition

The object recognition is split up into two phases. First, the image is reduced to the ROI as it can be seen in Fig. 6.6 for the example of machine light detection using a **H**istogram of **O**riented **G**radients (HOG) [56] detector. This ROI is then scaled to a defined image size and serialized to a vector with respect to the three color channels. This vector is then classified using a **F**eed-**F**orward **N**eural **N**etwork (FF-NN).

5.6.1 Histogram of Oriented Gradients

To identify the object in the image, i.e. to find the ROI, an feature detection algorithm is used. For this, the histogram of oriented gradients descriptor is used. This approach works similar to e.g. **S**cale-**I**nvariant **F**eature **T**ransform (SIFT) as it describes each pixel with an descriptor. In this case, the histogram of the gradient orientations of the neighboring pixels is used.

To train this, labeled images are necessary where the ROI is defined. An **S**upport **V**ector **M**achine (SVM) is then trained to classify each pixel if the desired object is here or not for a pyramid of different image resolutions (for scale invariance). This feature extraction and classification is done using the `dlib`[5] library.

Standing in front of a machine, the mounted camera on the top captures an image as it can be seen in Figure 6.5 in Section 6.3.3. The trained HOG-Detector, respectively the learned SVM, is then capable of extracting the ROI as they can be seen in Figure 6.6 (in Section 6.3.3 too). This extraction works

[4] wiki.ros.org/dwa_local_planner
[5] dlib.net

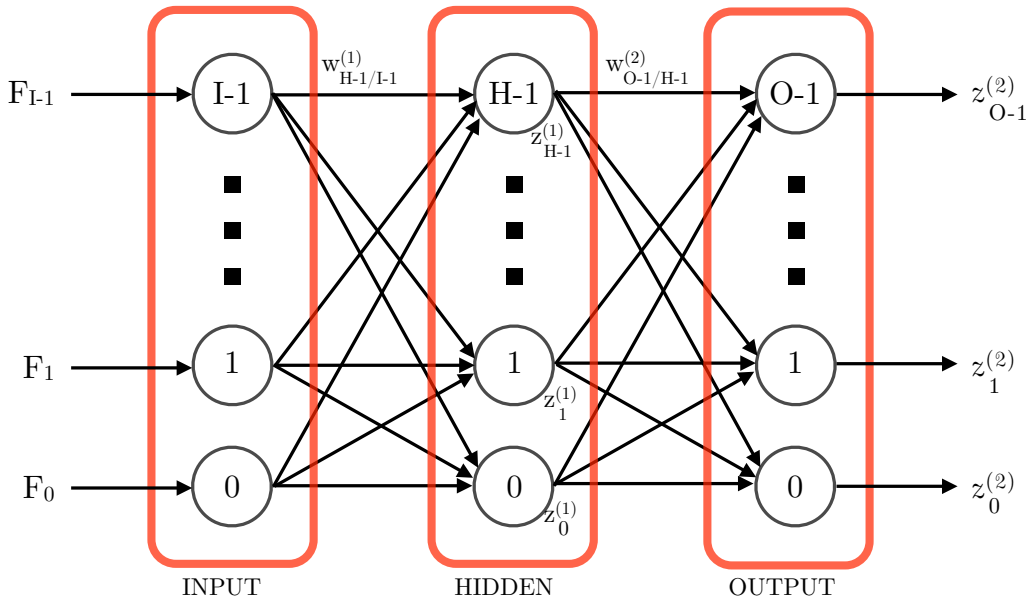


FIGURE 5.3: Structure of a generic artificial feed-forward neural network with one hidden layer.

with almost no false-positive results. Having this, the problem is reduced to a standard classification problem which can be solved using a neural network.

5.6.2 Neural Network

To classify these extracted ROIs, a classic feed-forward neural network (see [57]) with one hidden layer can be used as shown in Figure 5.3. This generic network has I inputs, H hidden units and O output neurons. The intermediate layer is called hidden as their units are not accessible from the outside directly. This representation hides the presence of an additional bias for each neuron. This is done by adding a constant 1 as input feature F_0. Therefore the weights from neuron 0 to the hidden layer represent the bias. This is equivalent to the representation of the network with an additive bias. The same scheme is used to represent the bias in the other layers too.

As activation function, the log-sigmoid transfer function (logsig) is used in most of the cases which is defined as

$$\sigma(x) := \frac{1}{1 + e^{-x}}. \tag{5.1}$$

This function has the properties to limit the output to the range $[0, 1]$ and is skew-symmetric around the point $(0, 0.5)$ (see Figure 5.4). This function allows learning nonlinear relationships between the input and the output and due to the limited range of this function, the intermediate resulting numbers stay small

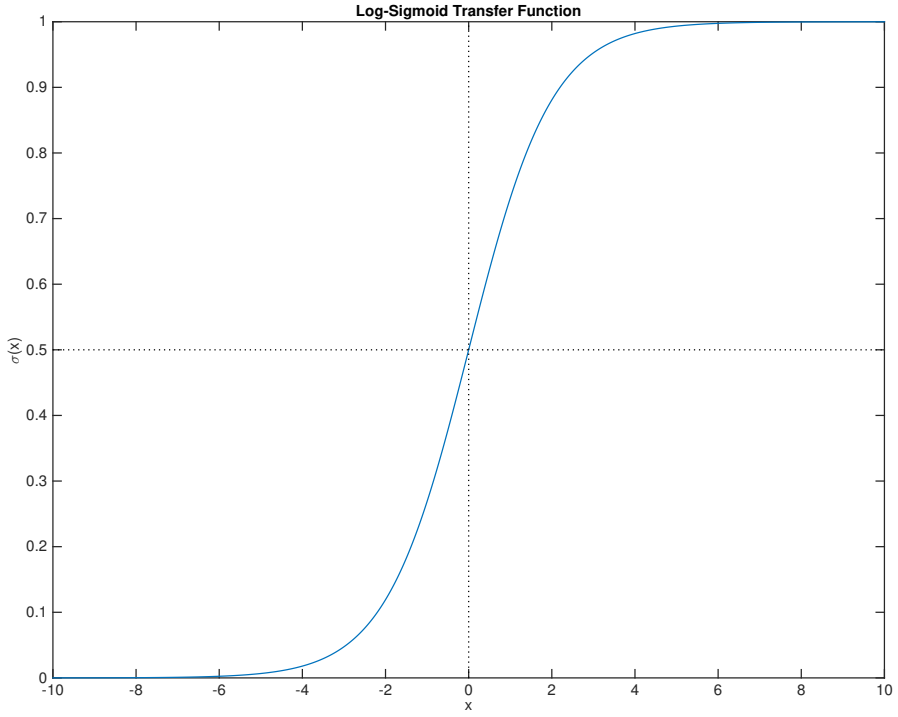


FIGURE 5.4: Log-sigmoid $\sigma(x)$ function for $x \in [-10, 10]$.

and numerical problems are avoided. Furthermore, it allows a simpler decision rule at the output.

With this definitions shown in Figure 5.3, the network performs the following calculations:

$$z_h^{(1)} = \sigma\left(\sum_{i=0}^{I-1} w_{h/i}^{(1)} \cdot F_i\right), \quad \forall h \in \mathbb{N}, 0 \leq h < H \tag{5.2}$$

where the subscript j numbers the outputs of the first layer, denoted with the superscript (1) referencing to the first layer. The same operation is performed again to generate the outputs as:

$$\begin{aligned} z_o^{(2)} &= \sigma\left(\sum_{h=0}^{H-1} w_{o/h}^{(1)} \cdot z_h^{(1)}\right) \\ &= \sigma\left(\sum_{h=0}^{H-1} w_{o/h}^{(1)} \cdot \sigma\left(\sum_{i=0}^{I-1} w_{h/i}^{(1)} \cdot F_i\right)\right), \quad \forall o \in \mathbb{N}, 0 \leq o < O \end{aligned} \tag{5.3}$$

Due to the properties of the logsig function, the outputs $z_0^{(2)}$ to $z_6^{(2)}$ are in the range $[0, 1]$ if an image is given as feature vector F. The outputs can be represented by an output vector Z consisting of these elements. To further normalize the outputs to sum up to one and therefore giving them the meaning

of an approximation of the probability P_o, that the shown image is out of the class o, the so-called softmax function (also called normalized exponential) is used [58]. This is the more generalized form of the logsig function and is calculated as:

$$P_o = \varphi(z_o^{(2)}) = \frac{e^{z_o^{(2)}}}{\sum_{i=0}^{O-1} e^{z_i^{(2)}}}, \quad \forall o \in \mathbb{N}, 0 \leq o < O. \tag{5.4}$$

This value can then be used to decide which object was shown and the confidence can be used to decide if the result is used or not. In the simplest case, the estimated class $\hat{C}$ is chosen as the maximum value, i.e.

$$\hat{C} = \underset{i}{\operatorname{argmax}} \left(P_i\right). \tag{5.5}$$

5.7 AR Tag Detection

The detection of the tags is implemented using the ALVAR framework[6]. For this, a ROS[7] wrapper is already available, called `ar_track_alvar`. Configuring this package with the correct camera, the used types of tags, and the expected output frame of the found tags, is all what is needed for this to work for one robot. Having this, the node publishes the transformation (ROS `tf`[8]) to each detected AR tag, i.e. the position and orientation of this tag in the given frame.

As we need several robots working with a single ROS master (simulation of multiple agents), it was necessary to adopt the source code to use customized prefixes for each robot. This is implemented using an additional launch parameter, named `tf_prefix`. With this, all the robots can work independently.

The used camera is a simple 2D camera, and therefore the inferred orientation of the tags is calculated due to the characteristic distortion of the tags in the camera image. This leads to decreasing accuracy with tags far away and at sharp angles. To cope with this, a filter is implemented discarding these measurements and using only those within a certain distance and a defined cone relative to the robot.

[6] virtual.vtt.fi/virtual/proj2/multimedia/alvar/index.html
[7] wiki.ros.org/ar_track_alvar
[8] wiki.ros.org/tf

Chapter 6

Software Architecture

In this chapter we describe the developed software architecture with the RoboCup Logistic League as a concrete area of deployment. The developed architecture can easily be adapted to similar fields of application too.

6.1 Layered Architecture

The software architecture is split up into three layers as it can be seen in Figure 6.1. Lower layers provide functionality to higher layers in a fashion similar to ISO Open System Interconnection (ISO OSI) reference model [10]. The lowest layer is mainly based on ROS and provides basic functionality to the upper layers, such as open or close the gripper, move to way-points or to classify the observed light-pattern. The next layer is based on the BDI engine openPRS (see Section 5.3.2). Here, first error handling and more complex tasks are implemented, such as aligning in front of a machine and grab a product. On top of this, the third layer is responsible for the communication to the referee box and distributes work-packages (e.g. fetch a base, mount a ring, etc.) in such a way, that overall the number of achieved points is maximized.

All tasks are initiated in the highest layer. These work-packages are then decomposed into sub-goals and posted down the layered architecture until the lowest layer is reached. Any bypassing of this strict layer structure has to be avoided to maintain a clear semantics.

The interfaces communicate with serialized data using Google's protocol buffer (protobuf)[1]. With this defined messages it is possible to exchange all of the layers. Even different programming languages are possible as protobuf supports (version 3) C++, Java (including JavaNano), Python, Go, Ruby, Objective-C, and C#. Several third party implementations are available too, supporting C, Perl, PHP, Scala, Julia and Apple's Swift. This variety of languages allows enormous flexibility.

One exception here is the communication between the mid-level and the low-level. To reduce the overhead of the communication, ROS' mechanisms are used

[1] developers.google.com/protocol-buffers/

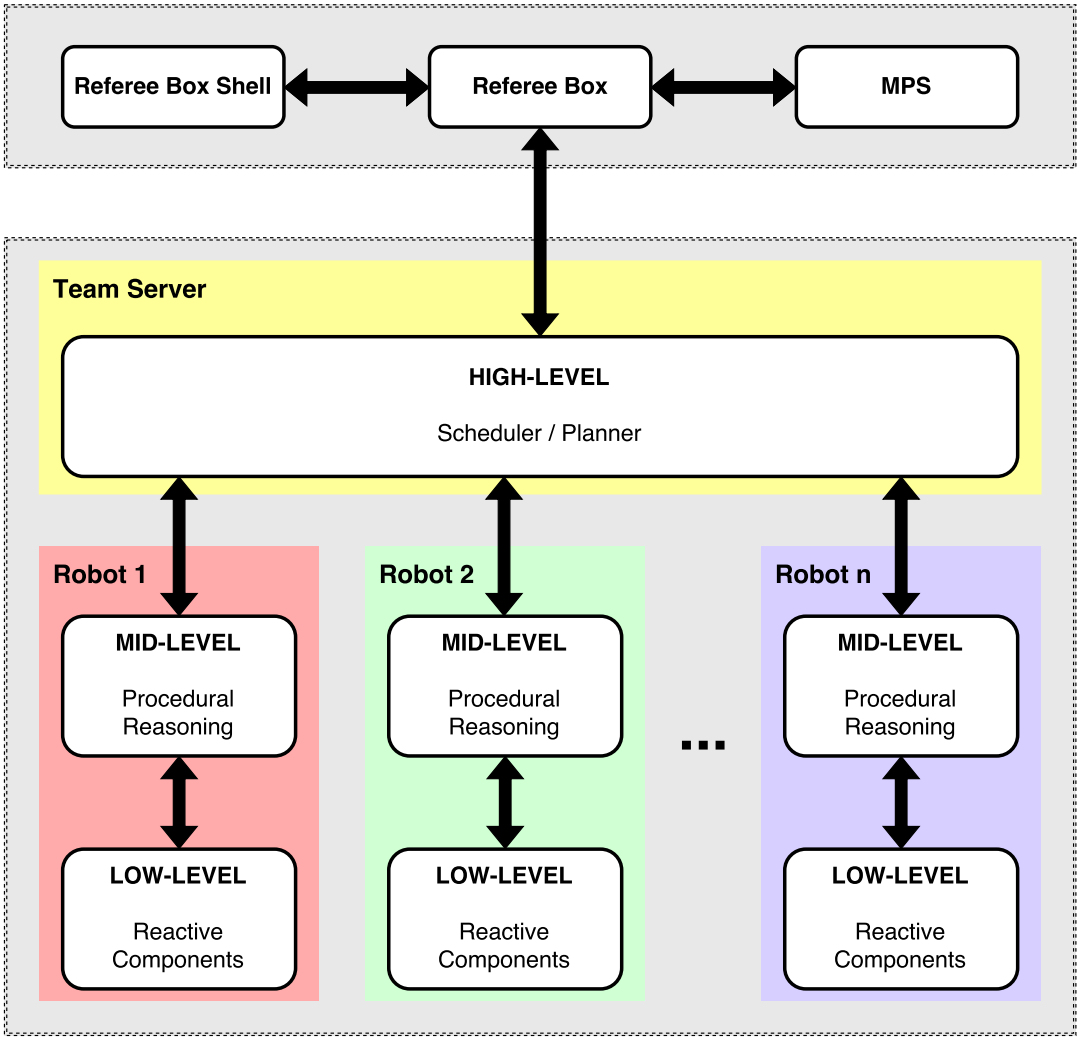


FIGURE 6.1: Layered software architecture with centralized high-level, distributed mid-level and low-level components.

to communicate. This can be easily wrapped with protobuf messages too in the Protobuf-ROS interface.

6.2 Communication Scheme

The communication between the different layers is based on Google's Protocol Buffer protocol where it is possible as it can be seen in Fig 6.2 and described in [6]. This serialization protocol allows a strict separation between the components. Protobuf is used as the communication to the referee box is given (and there it is used). To avoid confusion using different approaches, and as this protocol is very efficient and widely used, it is used throughout the whole project. Open PRS is integrated using its "Message Passer" and evaluable predicates and actions (see Section 5.3.3 for further details). The last communication to ROS is done using standard ROS topics and actions (see Section 5.4).

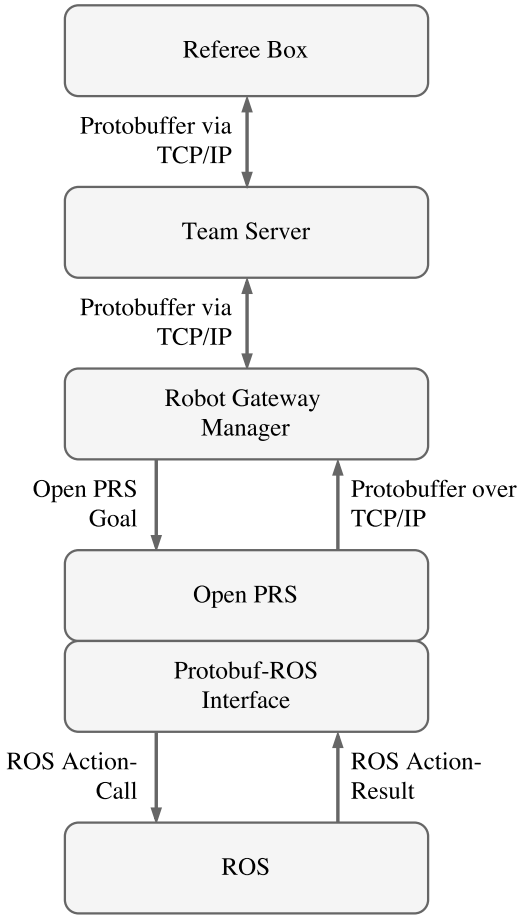


FIGURE 6.2: Communication protocols between the different layers in the system.

6.2.1 Recovery of Message Type

The functionality to communicate with a protobuf server is provided by the `protobuf_comm` library[2]. As we use this communication scheme in our framework too (client and server), the communication with a client needs to be implemented too. To distinguish the raw messages, the same technique is used therefore, i.e. to infer the type of received message, it is necessary to add a specific enum to each of the messages. The used comparison ID is a unique pair of `COMP_ID` and `MSG_TYPE`. An example is shown in Listing 6.1. With this, the type of received messages can be reconstructed. This reconstruction depends

[2]github.com/robocup-industrial/protobuf_comm

LISTING 6.1: CompType Protobuf enum.

```
enum CompType {
    COMP_ID  = 1000;
    MSG_TYPE =    1;
}
```

on the used programing language, e.g. using Java it is possible to use reflection to check for the existence of this element and parse the value as it is described in Section 5.1.3. An overview of the used messages and the comparison type pair is given in Table 6.1.

Message	COMP_ID	MSG_TYPE
BeaconSignal	2000	1
VersionInfo	2000	3
LightSpec	2000	10
Machine	2000	12
MachineInfo	2000	13
GameState	2000	20
RobotInfo	2000	30
Robot	2000	31
OrderInfo	2000	41
Order	2000	42
MachineReportEntry	2000	60
MachineReport	2000	61
MachineReportInfo	2000	62
ExplorationSignal	2000	70
ExplorationZone	2000	71
ExplorationInfo	2000	72
PrepareMachine	2000	101
RingInfo	2000	110
RobotStatus	2000	204
PrsTask	5006	701

TABLE 6.1: Used comparison types for protobuf communication.

6.2.2 Communication between Referee Box and Scheduler

The communication between the referee box and our software framework is restricted to our highest level, the scheduler/planner. For this communication, two dedicated channels are used, an encrypted private channel and a public channel. The used messages between the referee box and our framework (defined in the refbox manual[3]) are the following:

- Private Channel
  - Machine Info (see Section 6.2.2.1)
  - Machine Report (see Section 6.2.2.2)
  - Machine Report Info (see Section 6.2.2.3)
- Public Channel

[3] www.robocup-logistics.org/refbox

Listing 6.2: MachineInfo Protobuf message.

```
message MachineInfo {
  repeated Machine machines    = 1;
  optional Team    team_color = 2;
}
```

  - Game State (see Section 6.2.2.4)
  - Exploration Info (see Section 6.2.2.5)
  - Version Info (see Section 6.2.2.6)
  - Robot Info (see Section 6.2.2.7)
- Both Channels
  - Beacon Signal (see Section 6.2.2.8)
  - Order Info (see Section 6.2.2.9)
  - Ring Info (see Section 6.2.2.10)

The `CompType` is suppressed in the following sections to maintain a better overview of the messages. In general, each of the protobuf message has a unique `COMP_ID` and `MSG_TYPE` pair as described in Section 5.1.

6.2.2.1 Machine Info

The machine info consists of a list of machines (defined in Listing 6.3) and the team color (defined in Listing 6.24) and is shown in Listing 6.2. This can be used to send instructions to the machines (i.e. prepare machine) and for the referee box sends this message periodically to inform about the machines state and configuration.

6.2.2.2 Machine Report

The machine report message consists of the team color (defined in Listing 6.24) and a list of machine report entries (defined in Listing 6.5) and is described in Listing 6.4. This message is used by the scheduler/planner to report machine types and the light shown to the referee box.

6.2.2.3 Machine Report Info

The machine report info message consists of a list of reported machines and the team color (defined in Listing 6.24) and is described in Listing 6.6. This message is sent from the referee box to the scheduler/planner to acknowledge reported machines.

LISTING 6.3: Machine Protobuf message.

```
message Machine {
  required string    name                =  1;
  optional string    type                =  2;
  optional string    state               =  3;
  optional Team      team_color          = 10;

  repeated LightSpec lights              =  7;
  optional Pose2D    pose                =  8;
  optional Zone      zone                = 11;
  optional bool      correctly_reported  =  9;

  // Number bases loaded if machine is a Ring Station
  optional uint32    loaded_with         = 13 [default = 0];
  // Available ring colors at this machine if machine is a Ring Station
  repeated RingColor ring_colors         = 14;

  // Instructions to send to the machines
  optional PrepareInstructionBS instruction_bs     = 16;
  optional PrepareInstructionDS instruction_ds     = 17;
  optional PrepareInstructionRS instruction_rs     = 18;
  optional PrepareInstructionCS instruction_cs     = 19;

  // Additional exploration information.
  optional ExplorationState exploration_type_state = 21;
  optional ExplorationState exploration_zone_state = 22;
}
```

LISTING 6.4: MachineReport Protobuf message.

```
message MachineReport {
  required Team               team_color = 2;
  repeated MachineReportEntry machines   = 1;
}
```

LISTING 6.5: MachineReportEntry Protobuf message.

```
message MachineReportEntry {
  required string name = 1;
  optional string type = 2;
  optional Zone   zone = 3;
}
```

Listing 6.6: MachineReportInfo Protobuf message.

```
message MachineReportInfo {
  repeated string reported_machines = 1;
  required Team   team_color        = 2;
}
```

6.2.2.4 Game State

The game state message contains the current game state (i.e. if the game is running), the game phase (e.g. exploration, production), the time since game start (defined in Listing 6.21), the points for both teams as well as the team names ans is described in Listing 6.7. This message is sent periodically by the refbox to the teams.

6.2.2.5 Exploration Info

The exploration info message contains the exploration signals (defined in Listing 6.9 with the mapping from light pattern to a random string and the exploration zones (defined in 6.10 (i.e. the zones to explore) and is described in Listing 6.8. This message is sent during the exploration phase periodically by the refbox.

6.2.2.6 Version Info

The version info is sent by the refbox to distribute the current refbox version and is described in Listing 6.11.

6.2.2.7 Robot Info

The robot info contains a list of robots (defined in Listing 6.13 and is described in Listing 6.12. The robot message contains therefore all information about one robot, i.e. the name, the corresponding team, the team color, robot number and more. This message is sent by the refbox to acknowledge all reported signals using the beacon signal described in Section 6.2.2.8.

6.2.2.8 Beacon Signal

The beacon signal contains status information such as time (defined in Listing 6.21) and a sequence number and is described in Listing 6.15. This message is used as an alive-message and is sent periodically by all active robots to the refbox and by the refbox to the teamserver.

Listing 6.7: GameState Protobuf message.

```
message GameState {
  required Time   game_time      = 1;

  required State  state          = 3;
  required Phase  phase          = 4;

  optional uint32 points_cyan    = 5;
  optional string team_cyan      = 6;

  optional uint32 points_magenta = 8;
  optional string team_magenta   = 9;

  enum State {
    INIT       = 0;
    WAIT_START = 1;
    RUNNING    = 2;
    PAUSED     = 3;
  }

  enum Phase {
    PRE_GAME    = 0;
    SETUP       = 10;
    EXPLORATION = 20;
    PRODUCTION  = 30;
    POST_GAME   = 40;
  }
}
```

Listing 6.8: ExplorationInfo Protobuf message.

```
message ExplorationInfo {
  repeated ExplorationSignal signals = 1;
  repeated ExplorationZone   zones   = 2;
}
```

Listing 6.9: ExplorationSignal Protobuf message.

```
message ExplorationSignal {
  required string    type   = 1;
  repeated LightSpec lights = 2;
}
```

Listing 6.10: ExplorationZone Protobuf message.

```
message ExplorationZone {
  required Zone zone       = 2;
  required Team team_color = 3;
}
```

Listing 6.11: VersionInfo Protobuf message.

```
message VersionInfo {
  required uint32 version_major  = 1;
  required uint32 version_minor  = 2;
  required uint32 version_micro  = 3;
  required string version_string = 4;
}
```

Listing 6.12: RobotInfo Protobuf message.

```
message RobotInfo {
  repeated Robot robots = 1;
}
```

Listing 6.13: Robot Protobuf message.

```
message Robot {
  required string     name                          =  1;
  required string     team                          =  2;
  required Team       team_color                    = 12;
  required uint32     number                        =  7;
  required string     host                          =  3;
  required Time       last_seen                     =  4;

  optional Pose2D     pose                          =  6;
  optional Pose2D     vision_pose                   = 11;
  optional RobotState state                         =  8;

  optional float      maintenance_time_remaining =  9 [default↩
    = 0.0];
  optional uint32     maintenance_cycles            = 10;
}
```

Listing 6.14: RobotState enum.

```
enum RobotState {
  ACTIVE       = 1;
  MAINTENANCE  = 2;
  DISQUALIFIED = 3;
}
```

Listing 6.15: BeaconSignal Protobuf message.

```
message BeaconSignal {
  required Time    time        = 1;
  required uint64  seq         = 2;

  required uint32  number    = 8;
  required string team_name   = 4;
  required string peer_name   = 5;
  optional Team    team_color = 6;

  optional Pose2D pose        = 7;
}
```

Listing 6.16: OrderInfo Protobuf message.

```
message OrderInfo {
  repeated Order orders = 1;
}
```

6.2.2.9 Order Info

The order info contains a list of orders (described in Listing 6.17) to produce and is described in Listing 6.16. This message is sent by the refbox periodically and informs about new orders. Therefore it contains all the information of these orders, e.g. base color, ring color, cap color, quantity, delivery time-slot and delivery gate.

6.2.2.10 Ring Info

The ring info contains the rings (defined in Listing 6.19) and is defined in Listing 6.18. This message is sent periodically by the refbox during the production phase and supports information about the needed raw material for each ring.

6.2.2.11 Common Sub-Messages

Sub-messages used by the messages above are described in this section.

LightSpec Message The LightSpec message (defined in Listing 6.20) is used to describe a pattern of light, i.e. the state of each single lightbulb (red, orange, green).

Time Message To have a common sense of time, a time message is defined in Listing 6.21. It contains two fields, for the seconds and nanoseconds since the Unix epoch in UTC.

Listing 6.17: Order Protobuf message.

```
message Order {
  required uint32     id                          =  1;
  required Complexity complexity                  =  2;

  required BaseColor base_color                   =  3;
  repeated RingColor ring_colors                  =  4;
  required CapColor  cap_color                    =  5;

  required uint32     quantity_requested          =  6;
  required uint32     quantity_delivered_cyan     =  7;
  required uint32     quantity_delivered_magenta  =  8;

  required uint32     delivery_period_begin       =  9;
  required uint32     delivery_period_end         = 10;

  required uint32     delivery_gate               = 11;

  enum Complexity {
    C0 = 0;
    C1 = 1;
    C2 = 2;
    C3 = 3;
  }
}
```

Listing 6.18: RingInfo Protobuf message.

```
message RingInfo {
  repeated Ring rings = 1;
}
```

Listing 6.19: Ring Protobuf message.

```
message Ring {
  required RingColor ring_color   = 1;
  required uint32     raw_material = 2;
}
```

Listing 6.20: LightSpec Protobuf message.

```
message LightSpec {
  required LightColor color = 1;
  required LightState state = 2;
}
```

Listing 6.21: Time Protobuf message.

```
message Time {
  required int64 sec  = 1;
  required int64 nsec = 2;
}
```

Listing 6.22: Pose2D Protobuf message.

```
message Pose2D {
  required Time  timestamp = 1;

  required float x         = 2;
  required float y         = 3;
  required float ori       = 4;
}
```

Pose2D Message To communicate the pose of a robot, the Pose2D message is defined in Listing 6.22. Here the coordinates of the robot in the global frame and the orientation of the robot is contained.

Machine Instruction Messages Each machine is capable to perform different types of production steps. To instruct the machines, different machine instructions are defined in Listing 6.23.

6.2.2.12 Common enums

Enums used in multiple messages are defined the following section.

Team enum The two teams, CYAN and MAGENTA are defined with the team enum described in Listing 6.24.

Light Color and State enum The different light colors and the possible states of these lights are defined in Listing 6.25 and Listing 6.26.

Product Color enums The different colors for the rings, bases and caps are defined in Listing 6.27.

Listing 6.23: MachineInstructions Protobuf message.

```
message PrepareMachine {
  required Team   team_color                    = 1;
  required string machine                       = 2;

  optional PrepareInstructionBS instruction_bs = 4;
  optional PrepareInstructionDS instruction_ds = 5;
  optional PrepareInstructionRS instruction_rs = 6;
  optional PrepareInstructionCS instruction_cs = 7;
}

enum MachineSide {
  INPUT  = 1;
  OUTPUT = 2;
}

enum CsOp {
  RETRIEVE_CAP = 1;
  MOUNT_CAP    = 2;
}

message PrepareInstructionBS {
  required MachineSide side  = 1;
  required BaseColor   color = 2;
}

message PrepareInstructionDS {
  required uint32 gate = 1;
}

message PrepareInstructionRS {
  required RingColor ring_color = 1;
}

message PrepareInstructionCS {
  required CsOp operation = 1;
}
```

Listing 6.24: Team enum

```
enum Team {
  CYAN    = 0;
  MAGENTA = 1;
}
```

Listing 6.25: LightColor enum

```
enum LightColor {
  RED    = 0;
  YELLOW = 1;
  GREEN  = 2;
}
```

Listing 6.26: LightState enum

```
enum LightState {
  OFF   = 0;
  ON    = 1;
  BLINK = 2;
}
```

Listing 6.27: Product Color enums.

```
enum RingColor {
  RING_BLUE   = 1;
  RING_GREEN  = 2;
  RING_ORANGE = 3;
  RING_YELLOW = 4;
}

enum BaseColor {
  BASE_RED    = 1;
  BASE_BLACK  = 2;
  BASE_SILVER = 3;
}

enum CapColor {
  CAP_BLACK = 1;
  CAP_GREY  = 2;
}
```

6.2.3 Communication between Scheduler and Mid-Level Control

The communication between the scheduler and the mid-level is done with protobuf messages too. For this, a single channel is used, and only one message is defined for this purpose, the `PrsTask` message.

6.2.3.1 PrsTask

The communication between the highest layer and the middle layer uses the capability of Google's protobuf to test if a field is filled. Therefore, only one central message is needed to command tasks to the lower layer and return an execution result to the upper layer. The message is defined in Listing 6.28. This message contains the team color, a unique task ID, and a robot ID. Additionally it can contain one or more tasks and an execution result. For the RoboCup Logistic League, five of this optional messages are sufficient (can be easily extended in this place for future improvements):

- `ExploreMachineTask`
- `ReportAllMachinesTask`
- `GetWorkPieceTask`
- `PrepareCapTask`
- `DisposeWOrkPieceTask`
- `DeliverWorkPieceTask`

These tasks are sent to the mid-level and invoke a plan there. Therefore, sending such a message can be seen as the invocation of the corresponding plan as described in Section 6.4.

ExploreMachineTask This message is used to command a robot to move to a given machine, or if this machine is not known at the moment, to a defined zone and to explore it, i.e. detect light and zone of the machine as described in Section 6.5.2

ReportAllMachines To command the robot to send all it's known machines and their position in the game field (i.e. the zone), this message can be used.

GetWorkPieceTask With this command, the robot moves to the given station and retrieves a workpiece. The result of this task (if succeeded) is holding a workpiece in the gripper.

Listing 6.28: PrsTask Message.

```
message PrsTask {
    required Team                 teamColor             =  1;
    required uint32               taskId                =  2;
    required uint32               robotId               =  3;

    optional ExecutionResult      executionResult       =  4;

    optional ReportAllMachinesTask reportAllMachinesTask =  5;
    optional ExploreMachineTask   exploreTask           =  6;
    optional GetWorkPieceTask     getWorkPieceTask      =  7;
    optional PrepareCapTask       prepareCapTask        =  8;
    optional DisposeWorkPieceTask disposeWorkPieceTask  =  9;
    optional DeliverWorkPieceTask deliverWorkPieceTask  = 10;

    enum ExecutionResult {
        SUCCESS = 1;
        FAIL    = 2;
    }
}

// Move to the given machine if known in TF or to the zone otherwise and explore the machine there.
message ExploreMachineTask {
    required string machine = 1;
    required string side    = 2;
    required string zoneId  = 3;
}
// Send all observations to the team server.
message ReportAllMachinesTask {
    required bool report = 1;
}
// Get a workpiece at the given station and hold the product in the gripper.
message GetWorkPieceTask {
    required string providing_station = 1;
}
// Get to the cap station and put one base with cap into the machine.
message PrepareCapTask {
    required string cap_station = 1;
    required string cap_color   = 2;
}
// Get to the station and remove the workpiece and dispose it.
message DisposeWorkPieceTask {
    required string station = 1;
}
// Deliver the product in the gripper to the given station.
message DeliverWorkPieceTask {
    required string delivered_station = 1;
}
```

Listing 6.29: Conveyor detection ROS action server message.

```
# Goal
string offset_id
---
# Result
float32 distance_x
float32 distance_y
float32 angle
float32 machine_width
geometry_msgs/PoseStamped pose
---
# Feedback
float32 progress
```

PrepareCapTask This command can be used to prepare a cap station to mount a cap, i.e. to insert the right base with a cap.

DisposeWorkPieceTask To remove a workpiece from a machine, e.g. to remove the empty base at the cap station, this task can be used. The robot moves to the given machine and removes the workpiece.

DeliverWorkPieceTask If the robot holds a workpiece in its gripper, this task can be used to transport the product to the given machine.

6.3 Low-Level Layer

In this layer, primitive functionality, i.e. simple skills, are implemented (as this layer is the closest to the hardware and the physical world). We are using functionality provided by the ROS software stack to solve these problems. The functionality developed is implemented (as commonly used when working with ROS) as either an action server or as a node publishing information on a particular topic.

6.3.1 Conveyor Detection Node

The conveyor detection is implemented as an action server (see Section 5.4). The action is started by sending the corresponding goal containing a string defining the point of interest. This can be the input of the conveyor, the output, for the CS one of the three positions at the shelf, or the input for additional bases at the RS. Having this, the MPS is detected as described in Section 6.3.1.1 and the distance in x- and y-direction, as well as the angle θ to this point, and the measured machine width, is returned. The used action message can be found in Listing 6.29.

Data: Array of laser beams L, maximum distance to machine *dist_ max*, maximal valid discontinuity *epsilon*
Result: Position of the most left point of the machine x
center_beam_index = floor(size(L)/2);
if *L[center_ beam_ index] > dist_ max* **then**
  throw exception that no machine is found;
end
if *no AR-tag visible* **then**
  throw exception that object in front is no machine;
end
discontinuity = false; most_left_idx = size(L); old_point = calc_coordinates(L[center_beam_index]);
for *i = center_ beam_ index-1 to 0* **do**
  new_point = calc_coordinates_(L[i]);
  if *euclid_ distance(old_ point, new_ point) > epsilon* **then**
    most_left_idx = i;
    discontinuity = true;
    break;
  end
end
if *discontinuity is false* **then**
  throw exception that left corner of machine is not visible;
end
return x = calc_coordinates(L[most_left_idx]);

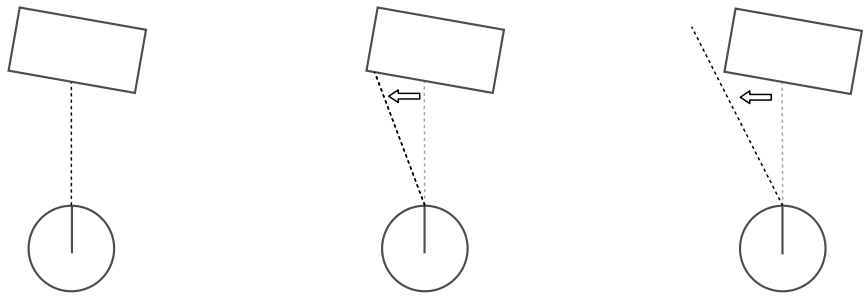
Algorithm 1: Detect most left coordinate of machine.

6.3.1.1 Machine Detection

As a first sanity check, it is ensured that there is an AR-tag visible and the found machine is at a reasonable distance (i.e. limited by a maximum distance d_{max}). If there is no AR-tag, there is no machine, and it is not reasonable to align.

If there is a machine, a simple line detection algorithm is used to determine the most left and most right coordinates of the machine (and therefore the center or any other point defined relative to these outer points). The used algorithm to detect the edges of the machine can be seen in Algorithm 1 as well in Figure 6.3. The difference in calculating the counterpart, i.e. the right edge, is simple iterating from the central laser beam upwards instead of downwards. Therefore this can be packed into one function. For simplicity reasons, only the left case is shown here.

The function, transforming the laser beam to coordinates in the robot's frame, uses the angle of the laser beam relative to the robot and the distance measurement of this beam, to calculate the position of the reflection. Having this, the Euclidean distance between adjacent beams can be calculated and used as a termination criterion.



(a) The central laser beam is used for the first detection of the machine.

(b) Checking each laser beam for continuity.

(c) Discontinuity as the corresponding laser beam misses the machine.

FIGURE 6.3: Line detection algorithm used for conveyor alignment.

LISTING 6.30: Conveyor alignment ROS action server message.

```
# Goal
string offset_id
int16 state
int16 state_align_safe_dist     = 3
int16 state_raw_alignment_safe = 2
int16 state_fine_alignment      = 1
---
# Result
bool success
---
# Feedback
float32 progress
```

Having both points, the most left and the most right of this line, the coordinates of the center between these points as well as the orientation of the machine can be calculated.

6.3.2 Conveyor Alignment Node

To initiate the alignment in front of a machine, this action server can be used. For this, three different alignment modes are available: alignment at a safe distance, raw alignment, and fine alignment. These three states of alignment can be passed as arguments of this action server as it can be seen in Listing 6.30. Here it can also be seen how an enumeration is defined such an action message (`state_align_safe_dist`, `state_raw_alignment_safe`, and `state_fine_alignment`).

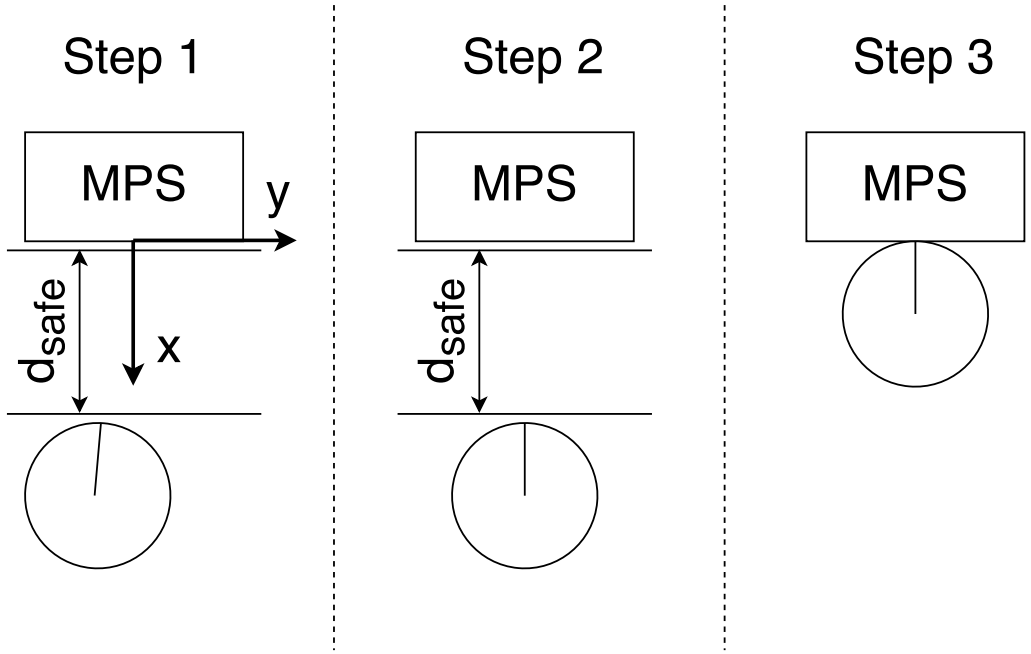


FIGURE 6.4: Results of each alignment step and used coordinate system. Alignment at a safe distance (step 1), raw-alignment at a safe distance (step 2), and fine-alignment (step 3).

6.3.2.1 Alignment Controller

The alignment in front of a machine is a critical part as it is necessary to be able to interact with the machine. It is not possible to use the way-point navigation for this for several reasons. First of all, the exact position of the machines is not known, i.e. a exact way-point in front of this machine cannot be deduced. Furthermore, it is necessary to enter the machine front in a specific way, i.e. to move normal to the machine front as otherwise a collision with the machine could happen. This leads to an other problem with the way-point navigation for this issue, as it is necessary to drive "into" the machine, i.e. the collision domain of the robot needs to overlap with the obstacle (in different heights, i.e. the gripper needs to be over the machine, but as only 2D information is given, this cannot be modeled) and this is an invalid state for the move-base.

To be able to overcome this, a multi-step solution was developed. First of all, the robot drives to a defined way-point (relative to the machine) at a safe distance using the way-point navigation. When arrived, a three-step alignment procedure is started:

1. alignment at a safe distance.
2. raw-alignment at a safe distance.
3. fine-alignment.

The results of each of these steps are depicted in Figure 6.4.

In each of these steps, a different controller structure is used as described below. As the used robot is equipped with omni-directional wheels, independent control rules can be formulated for each direction (x, y) and the rotation.

Alignment at a Safe Distance This mode can be used to align in front of a machine at a safe distance where rotation of the robot is possible without colliding with the MPS. This alignment mode ensures a safe state and is, therefore, the first and last step of the alignment process.

The machine is detected and measured using the methods described in Section 6.3.1.1. Having this, the robot can align relative to this machine at a safe distance. This is necessary to allow the robot to rotate safely (i.e. if it would be too close after the way-point navigation, it could interfere with the machine by turning around). This is done using a simple P-controller structure for the velocity in x-direction (normal to the machine) as

$$v_x(t) = P_k \left(\Delta x(t) - d_{safe} \right) \tag{6.1}$$

where $\Delta x(t)$ is the actual distance to the machine in x-direction (as it can be seen in Figure 6.4) at time t and d_{safe} the defined safe distance to the machine. P_k is a parameter defining how aggressive the controller reacts to differences in the distances.

This movement is stopped as the difference between the actual position and the safe distance becomes smaller than a certain value. This can be choses rather high as this alignment have not to be precise at all, as only a safety distance needs to be achieved and no precise position is needed.

Raw-Alignment at a Safe Distance To allow a fast alignment, a second mode is introduced. Using this, a PI-controller is used for all the three parameters (velocity in x- and y-direction and angular speed). This mode should be only used after an alignment at a safe distance has been performed as this controller is optimized for fast alignment and can lead to a significant overshooting if the control error is too big.

As the robot is now in a safe position, the orientation of the robot can be corrected without the danger of a collision. So, in this step, the angle to the machine is corrected as well as the position of the robot relative to the machine in x- and y direction (as it can be seen in Figure 6.4). This can be formulated

as a controller rule (PI-controller) in three variables as

$$v_x(t) = P_k\left(\Delta x(t) - d_{safe}\right) + I_k \int_{\tau=0}^{t} \left(\Delta x(\tau) - d_{safe}\right) d\tau \tag{6.2}$$

$$v_y(t) = P_k\left(\Delta y(t) - y_t\right) + I_k \int_{\tau=0}^{t} \left(\Delta y(\tau) - y_t\right) d\tau \tag{6.3}$$

$$\omega_x(t) = P_k\left(\Delta \theta(t) - \theta_t\right) + I_k \int_{\tau=0}^{t} \left(\Delta \theta(\tau) - \theta_t\right) d\tau \tag{6.4}$$

where y_t and θ_t denotes the target distance in y-direction (parallel to the machine) and the target angle relative to the machines (rotation about z, in terms of a right hand coordinate system). I_k is the parameter to chose defining the impact of the integral part.

Fine-Alignment The third and final step is different than the before taken ones as it is close to the machine. Overshooting of the controller would mean here a collision with the machines. Therefore, a special controller structure was chosen limiting the overshooting using saturation (anti-windup strategy, based on Astrom et al. in [59]).

For this, the saturation operator $sat_{x_{min}}^{x_{max}}$ needs to be defined as

$$\mathrm{sat}_{x_{min}}^{x_{max}}(x) := \max(x_{min}, \min(x_{max}, x)) \tag{6.5}$$

where $\max(a, b)$ and $\min(a, b)$ is defined as

$$\max(a, b) := \begin{cases} a, & \text{if } a > b \\ b, & \text{else} \end{cases} \tag{6.6}$$

$$\min(a, b) := \begin{cases} a, & \text{if } a < b \\ b, & \text{else.} \end{cases} \tag{6.7}$$

With this, the controller rule can be written as

$$v_x(t) = I_k \int_{\tau=0}^{t} \left(\Delta x(\tau) - x_t\right) d\tau - T_k \left(v_x(t - \Delta t) - \mathrm{sat}_{v_{min}}^{v_{max}}(v_x(t - \Delta t))\right) \tag{6.8}$$

$$v_y(t) = I_k \int_{\tau=0}^{t} \left(\Delta y(\tau) - y_t\right) d\tau - T_k \left(v_y(t - \Delta t) - \mathrm{sat}_{v_{min}}^{v_{max}}(v_y(t - \Delta t))\right) \tag{6.9}$$

$$\omega(t) = I_k \int_{\tau=0}^{t} \left(\Delta \theta(\tau) - \theta_t\right) d\tau - T_k \left(\omega(t - \Delta t) - \mathrm{sat}_{\omega_{min}}^{\omega_{max}}(\omega(t - \Delta t))\right) \tag{6.10}$$

Listing 6.31: HOG detector ROS action server message.

```
# Goal
int32 num_images
---
# Result
float32[] probabilities
---
# Feedback
float32 progress
```

where $v_x(t - \Delta t)$, $v_y(t - \Delta t)$, and $\omega(t - \Delta t)$ denotes the velocities in the last time-step and T_k is a constant to tune the behavior of the anti-windup approach.

Implementation Details To implement the alignment controller described, e.g. to calculate the integral parts as valid summations, a constant update period is necessary. As we do not use a real-time system, it is not possible to guarantee this. The controller is updated as precise as possible using `ros::Rate`[4]. An alternative would be to use the timer library[5] provided by ROS (also no real-time guarantees).

6.3.3 HOG Detector Node

Detecting the lights observed by the mounted camera is done using this action server. The detection is done using a HOG detector with an support vector machine to extract the ROI (see Section 5.6.1) and an FF-NN (see Section 6.3.3.2) to classify these regions. The result of this action is an array containing seven probabilities for each combination of lights or all zeros if no lights are found as it can be seen in Listing 6.31

6.3.3.1 ROI Extraction

The view of the robot standing in front of such a machine can be seen in Figure 6.5. With this, the extracted ROI found with the HOG detector (as described in Section 5.6.1) is shown in Figure 6.6.

6.3.3.2 Feed-Forward Neural Network

The results of the HOG-detector, i.e. the ROI, is normalized to a defined size (e.g. 192×64 pixels). This color image can then be interpreted as an $192 \times 64 \times 3$-matrix as there are 3 color channels. Due to the fact that after the preprocessing this classification problem is rather easy, the features are

[4] wiki.ros.org/roscpp/Overview/Time
[5] wiki.ros.org/roscpp/Overview/Timers

FIGURE 6.5: View of the robot in front of a machine as seen by the light detection camera.

FIGURE 6.6: Region of interest found using a histogram of oriented gradients detector.

LISTING 6.32: Gripper controller ROS action server message.

```
# Goal
bool open
---
# Result
bool successfull
---
# Feedback
float32 progress
```

chosen to be the serialized matrix, i.e. all rows with all columns and all colors concatenated as a vector.

The used Network structure can be seen in Figure 5.3 in Section 5.6.2. As the serialized image is used as feature vector F, I is given as the product of image width, image height and number of colors and the additional constant feature representing the bias, i.e. $I = 192 \cdot 64 \cdot 3 + 1 = 36865$. The number of outputs O is given as the number of possible combinations of the three lights green, yellow and red, each with the binary states on and off, except the state where all lights are off, i.e. $O = 2^3 - 1 = 7$. The number of hidden units needs to be chosen empirically. As the number of features used is high and therefore the complexity of the network increases enormously with increasing number of hidden neurons (each input neuron is connected to each hidden neuron), 8 units are chosen.

Training Having this configuration, such a neural network is trained using the scaled conjugate gradient algorithm [60]. Additionally, a validation set is used to avoid overfitting. Training of the network is done using Matlab's neural network toolbox[6]. Therefore the training samples (100 per class) are split randomly into training- (70%), validation- (15%) and test-set (15%).

6.3.4 Gripper Controller Node

The gripper controller node offers a interface to the motors of the gripper to allow the opening and closing of this. For this purpose, again the action server approach is used. The server accepts messages containing whether the gripper should close or open as it can be seen in Listing 6.32. The used stepper motor allows to detect if the opening or closing procedure has worked or not (desired position is reached). This information is used to give a feedback, i.e. if the operation was successfully or not. Further sensors are not available and so this node is kept simple.

[6] de.mathworks.com/help/nnet/

Listing 6.33: Move to way-point ROS action server message.

```
# Goal
string wayPoint
---
# Result
bool success
---
# Feedback
```

6.3.5 Navigation Node

To navigate through the game-field, ROS `move_base` framework[7] is used. A simple interface to this navigation capabilities is achieved via an action server accepting strings describing defined way-points, e.g. at the corner of a zone or at the front of a machine as it is defined in the action message in Listing 6.33. For all of these way-points, a transformation (ROS `tf`[8]) is defined in an configuration file. The result of calculating the transformation from this point to the global can then be used as the target position for for the `move_base`.

6.3.6 Machine Position Publisher Node

To map the machines into the game-field and to allow movement targeted to a machine dynamically, the machine position publisher subscribes to the topic published by the AR framework and converts these points (i.e. tag for input, tag for output) to one single machine and publishes the transformation of this machine center. To make this work reliable, the found tag positions, and therefore the deduced machine position is filtered by an exponential decay filter.

The result of this node is a pair of way-points for each seen machine, one for the input and one for the output. As all the relevant machine parts (conveyor input, shelf) are defined relative to this machine center as transformation too (in a configuration file), this has the effect, that if a valid transformation for the machine center is published, all the derived transformations become valid too.

6.3.7 Check Existance of Way-Point Node

To check if a robot has already seen a machine (i.e. to be able to move relative to this machine, e.g. in front of it during the exploration phase), this action server can be used. Here, the transformation tree of the robot is checked determining if this position is already known. This functionality can be called with an action message as described in Listing 6.34.

[7] wiki.ros.org/move_base
[8] wiki.ros.org/tf

Listing 6.34: Way-point exists ROS action server message.

```
# Goal
string wayPoint
---
# Result
bool exists
---
# Feedback
```

Listing 6.35: Zone of way-point ROS action server message.

```
# Goal
string tf_string
---
# Result
string zone_name
---
#Feedback
```

6.3.8 Get Corresponding Zone of Way-Point

This action server can be used to derive the zone a given way-point is in, i.e. to perform a discretization as it is defined in the game-field layout to deduce e.g. the zone a machine is in. This is needed during the exploration phase to find the zone a machine is located in. If the way-point is not existent, an empty string is returned. This functionality can be used with the action massage defined in Listing 6.35.

6.4 Mid-Level Layer

The mid-level layer implements intermediate actions, i.e. combines primitive low-level skills and first error recovery and is implemented using the BDI engine open PRS[9] described in Section 5.3.2. Therefore, this section is focused on the implementation of some functionality using this paradigm. As examples for the idea behind this, three plans are described. One allow to align the robot in front of a machine (Section 6.4.2.1), the plan to retrieve a base (Section 6.4.2.2), as well as the procedure to explore a machine on the game-field (Section 6.4.2.3).

6.4.1 Abstraction of the Physical World

Having this additional layer between the hardware and the high-level allows a more fine-grained abstraction process. Within this layer, no knowledge about

[9] git.openrobots.org/projects/openprs

poses (i.e. x-, y-, z-position and orientation) is needed. The world is modeled in this sense using abstract way-points and the movement between these points is a simple call of an action. This enables us to model complex skills the robot is able to do and to provide these functionality to the high-level control.

6.4.2 Examples

To discuss the function of the mid-level actions, three examples are used to do so. With these examples, the used procedural reasoning engine is explained in an practical way.

6.4.2.1 Align in Front of a Machine

To align in front of a machine, the plan shown in Figure 6.7 can be used. This plan (started internally as the subgoal of an other plan) is started, if the invocation part is called and the context is true. In this case, the plan is started by posting e.g. the goal `(!(align C-CS1 input shelf1 10))` only if the predicate `running` is in the database (set if the game is in running mode). In this example, the robot would move to the input of the first CS of team cyan and aligns at the first position at the shelf. The action to move to the way-point will retry ten times, e.g. if the goal is not reachable as another robot is in the path or something similar. The plan consists therefore of several subgoals. As it can be seen, after the start the robot tries to move to the given way-point using `(!(failSafeWaypoint ...))`. If this fails, the predicate `(moveToWayPoint failed)` is written to the database to allow execution observation. If not, using the evaluable predicate `facemachine`, it is checked if after the movement the correct machine is in front of the robot. Is this correct, the robot aligns at a safe distance as described in Section 6.3.2.1 and sends the result `SUCCESS` to the high-level or `FAIL` if it failed. The plan is exited in the fail state if something went wrong (`(!(fail)`), otherwise the plan exits in success state.

6.4.2.2 Retrieve a Base

The plan to retrieve a base can be called e.g. by posting the goal `(retrieve_base C-BS RED 10)`. Here, the starting exclamation mark is not used as this plan needs to be callable via the message passer (i.e. the passer adds this as predicate to the database and then the plan is invoked) by the scheduler. In the shown example, the robot would try to get a red base from the BS of team cyan and would retry the movement ten times. As a first step in this plan, the ID of the robot is fetched from the database. Having this, the robot moves to the given machine. Arriving there, the station is configured using the action `(!(configureBS))`. If all this worked, the robot aligns in front of this machine, grabs the product and aligns at a safe distance again. If one of the steps fails, the reason for the failure is written to the database.

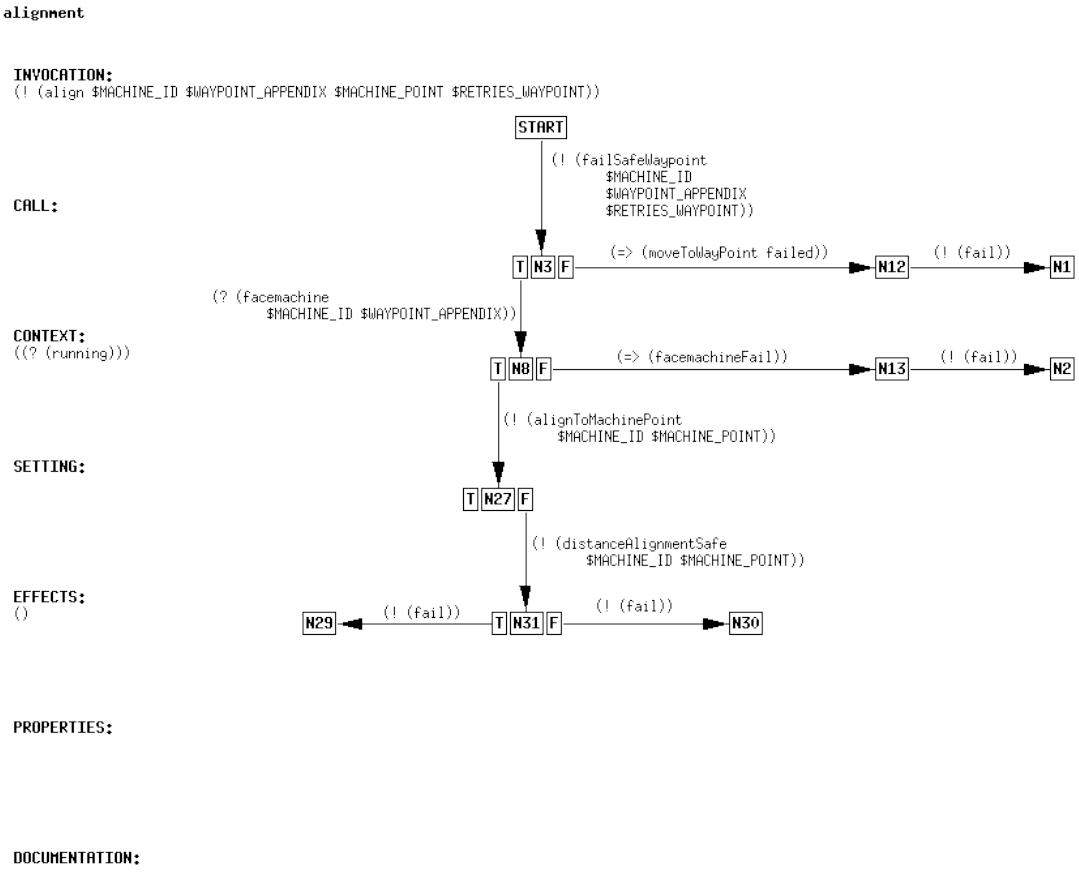


FIGURE 6.7: OP to align in front of a machine.

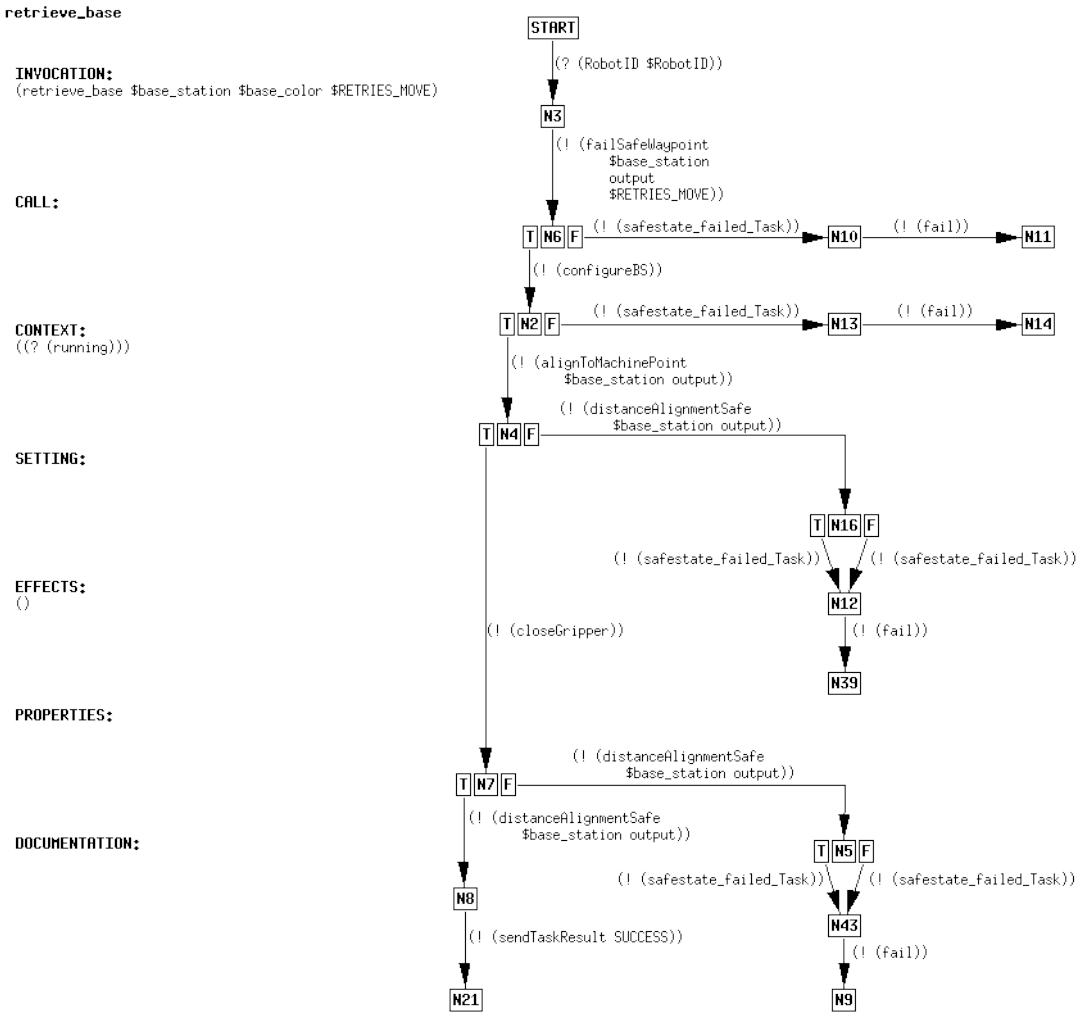


FIGURE 6.8: OP to retrieve a base.

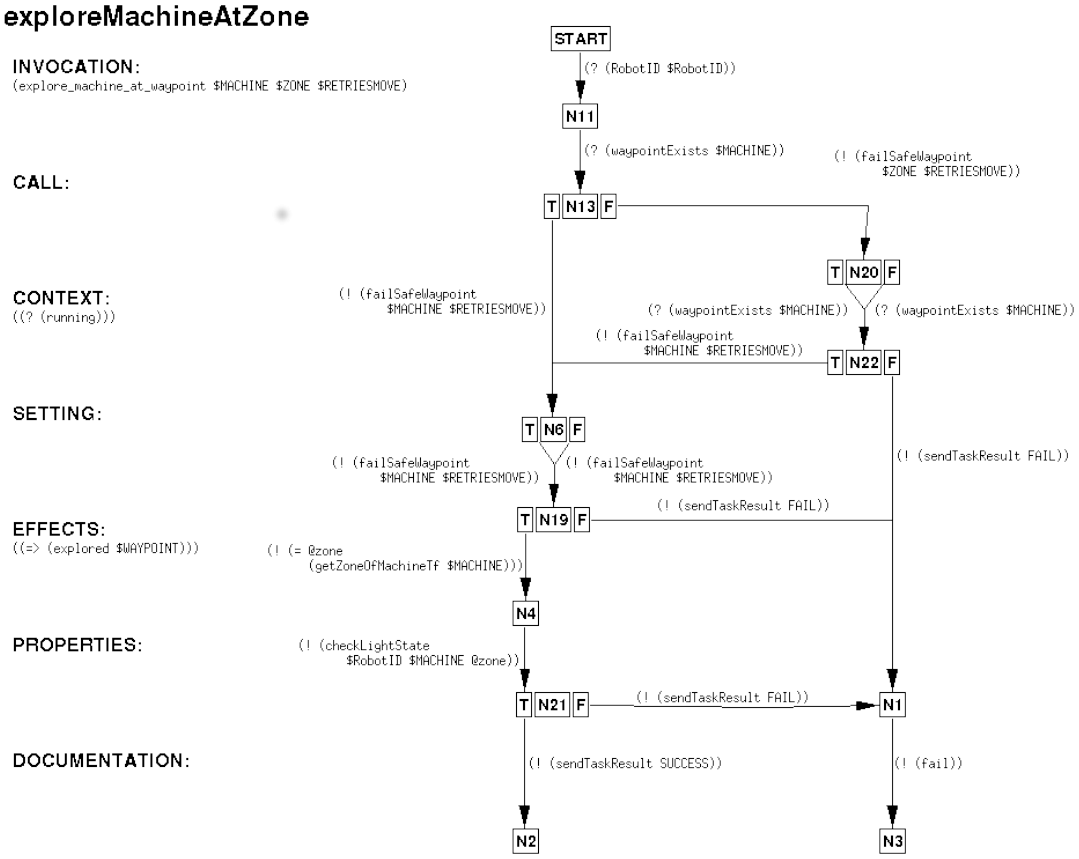


FIGURE 6.9: OP to explore a waypoint.

6.4.2.3 Explore a Machine

The plan to explore machines during the exploration phase (triggered by the scheduler) can be seen in Figure 6.9. This plan can be called by posting e.g. the goal `(explore_machine_at_waypoint C-CS1 Z10 10)`. This goal can be interpreted as to explore the machine `C-CS1` at zone `Z10`. The machine is optional here, it is just a initial guess. If the robot already have seen this machine, i.e. the corresponding way-point (by checking the evaluable predicate `waypointExists`), the robot moves directly to this machine. Otherwise, the robot will move to the given zone. Arriving at the zone, the procedure checks again if now the machine is known, i.e. if the robot has seen the machine on the way to the zone. If so, the robot moves to the way-point in front of the machine to detect the shown light-pattern. If not, failed is returned. If the machine is detected, and it is possible to align to it, the zone the machine is in is checked again (the given zone is a guess only) and the light-pattern is classified. If this works, the collected data is sent to the calling instance and the plan exits in the succeeded state, otherwise failed is returned.

6.5 High-Level Layer

The highest layer of our architecture is written in Java. The previous version was written using C++. The current implementation is described in this section as well as the used algorithms to distribute the tasks for the exploration phase.

6.5.1 Granularity and Flexibility

To keep the scheduler flexible, it is split up into two parts: the team-server and the scheduler (or several schedulers). The team-server is therefore responsible for the communication to the referee box and to the robots and collects the information received. The second part, the scheduler uses this data to generate new tasks. With this, several independent schedulers can be developed to fulfill specific tasks (e.g. different exploration scheduler approaches).

6.5.1.1 Team-Server

The team server opens the connection to the referee box and offers the connection to arbitrary many robots. This is done using TCP sockets in both directions. The used serialization protocol is Google's Protocol Buffer using the `protobuf_comm` library. For the communication, different handler are implemented for the different channels. Data arriving from the robots is handled within the `RobotHandler`. Here beacon messages, task results, and collected data are received, parsed and stored in a central database (using **S**tructured **Q**uery **L**anguage (SQL)) containing independent tables (conversion between the protobuf message and the shown objects below is done using the `Dozer` framework[10]). With this, the data persistence can be managed. Data arriving from the referee box is handled similarly. Here, data is arriving at two channels. As some messages arrive at both channels, an generic handler and two derived handler are implemented as shown in Figure 6.10. The received data is here parsed as described in Section 5.1.

Some of the used database schemes can be seen in Table 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9, 6.10, and 6.11 and are described in this section. In these tables, the table name is at the top row followed by the primary key in the second row.

BeaconSignal One of the fundamental information sent is the so-called beacon signal. This message is sent by the robots and the referee box periodically. To save all the received data in an database, two different tables are defined to store these two messages. As they overlap in some elements, a base class `BeaconSignal` is used to handle the common elements as it can be seen in Table 6.2. The additional elements are handled separately in Table 6.3 and Table 6.4. Here e.g. the time is stored and can be used to derive if a robot is active, i.e. if it has sent a beacon signal in the last few seconds.

[10] dozer.sourceforge.net

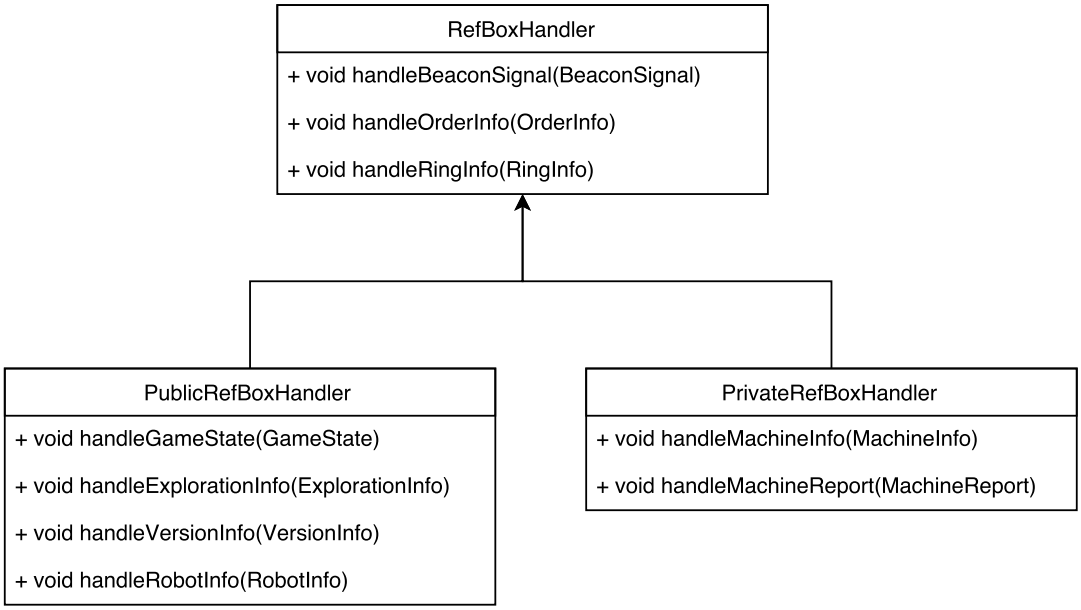


FIGURE 6.10: Inheritance of the `RefBoxHandler`.

BeaconSignal
Long beaconSignalID
Long seq TeamColor teamColor long timeNanoSeconds

TABLE 6.2: BeaconSignal Table.

BeaconSignalFromRefbox : BeaconSignal
Long beaconSignalID
String refBoxName String refBoxTeamName

TABLE 6.3: BeaconSignalFromRefbox table extending the BeaconSignal table sharing the same primary key.

BeaconSignalFromRobot : BeaconSignal
Long beaconSignalID
Long robotId String robotName String teamName double poseX double poseY double poseTheta

TABLE 6.4: BeaconSignalFromRobot table extending the BeaconSignal table sharing the same primary key.

GameState The referee box publishes information about the game periodically. To log this information and to provide it in a defined way to the scheduler,

the scheme defined in Table 6.5 is used. Here, information about the actual game time, team names and points is stored.

GameState
Long Id
String teamCyan String teamMagenta Long pointsCyan Long pointsMagenta Long gameTimeNanoSeconds

TABLE 6.5: GameState table to be filled with information received from the referee box.

LightSignal To report a machine's light signal, it is necessary to report a defined random string. This mapping from pattern to string is provided by the refbox and is stored as it can be seen in Table 6.6.

LightSignal
Long Id
String code LightPattern pattern

TABLE 6.6: LightSignal table containing the dynamic mapping between the light pattern and the string to report.

Ring Some rings need additional raw material to be mounted. The amount of this raw material is defined per game in a randomized fashion. This information is distributed by the referee box and stored in Table 6.7.

Ring
RingColor ringColor
long rawMaterial

TABLE 6.7: Ring table containing the amount of raw-material to provide for each ring.

MachineInfoRefBox During the exploration phase, the referee box distributes information about the game-field. This information is collected and made accessible to the scheduler via a scheme defined in Table 6.8.

ProductOrder Orders during the production phase are sent by the referee box at random time points. To collect this information, it is stored in the scheme depicted in Table 6.9.

MachineInfoRefBox
Long ZoneId
String name
String type
String state
TeamColor teamColor
Boolean correctlyReported
ExplorationState explorationTypeState
ExplorationState explorationZoneState
RingColor ring1
RingColor ring2

TABLE 6.8: MachineInfoRefBox table to be filled with information about the machines received by the referee box.

ProductOrder
Long Id
Complexity complexity
Long deliveryPeriodBegin
Long deliveryPeriodEnd
BaseColor baseColor
CapColor capColor
RingColor ring1
RingColor ring2
RingColor ring3
Long quantityRequested
Long quantityDeliveredCyan
Long quantityDeliveredMagenta

TABLE 6.9: ProductOrder table to model the orders received from the referee box.

RobotObservation Each robot explores the game-field during the exploration phase and reports its observations to the high-level. Here, the team-server collects this data in a scheme defined in Table 6.10.

RobotObservation
Long observationId
Long robotId
String machineName
String machineZone
Double machineZoneConfidence
Long observationTimeAbsolut
Long observationtimeGame

TABLE 6.10: RobotObservation table containing all the observations received from the robots during the exploration phase.

LightObservation If a robot discovers a machine and detect the light pattern of it, it reports its observations. This information is collected as a table too, shown in Table 6.11.

LightObservation
Long observationId
Long internId
String machineName
String machineType
String machineZone
LightPattern lightPattern
Double confidence
Long observationTimeAbsolut
Long oversvationTimeGame

TABLE 6.11: LightObservation table containing all the light observations received from the robots during the exploration phase.

This team server contains one or more schedulers. These schedulers have an defined interface, i.e. they have to define a method to generate the next task and can access the database described above.

6.5.1.2 Scheduler

The scheduler(s) can be implemented completely independent of the team-server, i.e. the communication part of the high-level control. For this, two interfaces need to be defined. The scheduler has access to the database generated by the team-server containing all collected data, i.e. all observations sent by the robots and all the data provided by the refbox in a central point. On the other side, each scheduler needs to implement the scheduler-interface, i.e. a method which returns the next tasks to perform as a `PrsTask` as described in Section 6.2.3.1 and defined in Listing 6.28. Having this, the scheduler can be easily exchanged and different schedulers for different scenarios can be developed independently. This complies with the overall design of this framework to be as modular and flexible as possible. One specific implementation of such a scheduler is presented in Section 6.5.2.

6.5.2 Exploration Scheduler

The developed exploration scheduler implements the defined scheduler interface and can therefore be used by the team-server to generate new tasks. To do this, the scheduler uses the information collected by the team-server in the database to generate a knowledge-base. This knowledge-base is modeled in a way it makes it easy to derive the next task. The class diagram of this scheduler can be seen in Figure 6.11. In this diagram, the most important classes and its members (reduced to those necessary for understanding) are shown. In this diagram, the

interface can be seen, i.e. the defined scheduler interface to communicate to the team-server and the interface to the database the **D**ata **A**ccess **O**bject (DAO) design pattern, e.g. the `gameStateDao`. A central point in this architecture is the `GameField` acting as a central knowledge representation.

6.5.2.1 Knowledge-Base

The knowledge of the game is modeled as a matrix of zones representing the game-field. This leads to a very natural way of looking at it and makes it easy to infer information. Each of these zones holds the information about it, i.e. the zone number, its physical neighbors, its mirrored zone (as the game-field is symmetric), the zone center, and its exploration state. Furthermore, each zone contains a list of observations made at this zone. This means, that information about observed machines (as their type, time of observation and the observing robot, optionally the shown light pattern) is stored. Having this, a belief of the game-field can be derived, i.e. a set of machines distributed in the game area which does not conflict with the constrains defined in Section 3.

As the game field has to be symmetric, the information of both sides can be merged to one half of the field. For this, the right one is chosen. The algorithm to merge the information is described in Algorithm 2. Here, the information of the zones in the right half are merged with those on the other side. For this, the affiliation of the machines observed need to be inverted for the mirrored ones, i.e. magenta becomes cyan and vice versa.

Input: gamefield
Output: gamefield
1: **for all** z ∈ gamefield.rightHalf **do**
2: z.addObsMirrored()
3: **end for**
4: gamefield.mirrorRightHalfToLeft()
5: **return** gamefield
Algorithm 2: Algorithm to merge the observations in the symmetric game-field halves.

Having this, conflicts in this knowledge-base needs to be found and eliminated. Therefore, a metric needs to be defined to chose those observations that are treated as correct. For this, the number of observations of a machine in a zone is used. This procedure is described in Algorithm 3. Here, for each machine the zone with the highest observation count is searched. Having this, all other observations of this machine are discarded. The result of this is a game field with no inconsistencies.

This belief of the world can then be used to generate the next task to perform, i.e. to explore the zone where it is optimal to go to for the robot next. For this, the zones have different exploration states:

- `UNEXPLORED`: No robot has been at this zone.

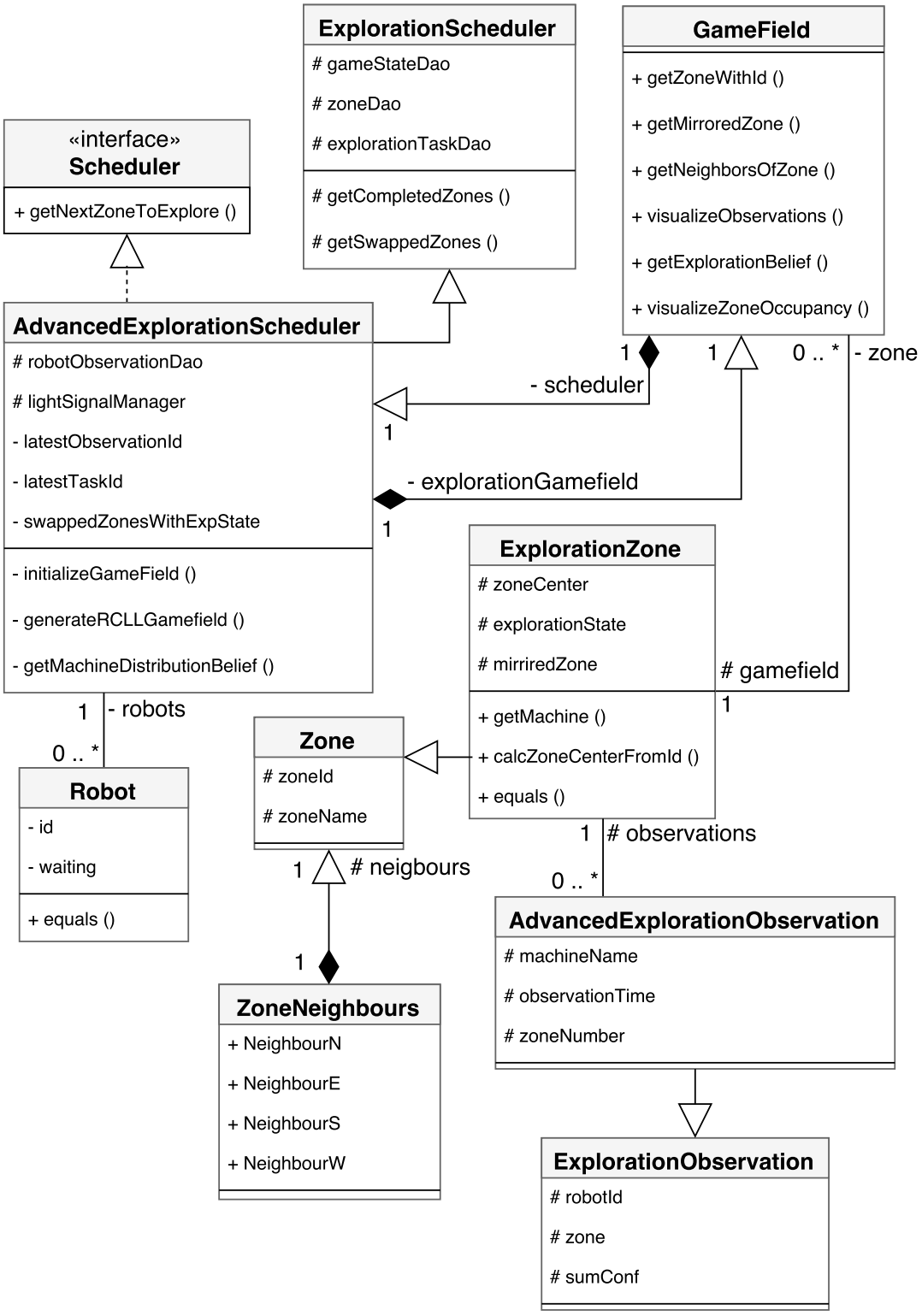


FIGURE 6.11: Class diagram of the exploration phase scheduler.

Input: gamefield, MPS
Output: gamefield
1: mergeMirroredObservations(gamefield)
2: **for all** m ∈ MPS **do**
3: (zone, count) = getZoneWithHighestCount(gamefield, m)
4: removeAllObsForMachine(gamefield, m)
5: addObsCountForMachineInZone(zone, count, m)
6: **end for**
7: **return** gamefield
Algorithm 3: Algorithm to generate a conflict-free game field given all possible machines (MPS).

- `EXPLORED`: A robot already has been at this zone but no light pattern was detected.
- `LIGHT_FOUND`: A robot already has been at this zone and a light pattern was detected but is not reported to the referee box yet.
- `REPORTED`: A robot already has been at this zone and a light pattern was detected and it already reported to the referee box.
- `SCHEDULED`: The zone is scheduled at the moment, i.e. a robot tries to explore it.

With this, the zones are scheduled in this order, i.e. first, zones where no robot has been are scheduled, than the explored ones, and so on. To exploit the knowledge of the two swapped zones as described in Section 2.4 (the own zones are advertised but the contained machine type and shown light pattern not), these zones are scheduled first (i.e. a special heuristic). The algorithm is described in Algorithm 4. As it can be seen, the fall-back solution if no machine has been detected so far (and the swapped zones are already reported or scheduled) is to chose a random one.

Having these tasks generated, the robots explore the given zones and report their observations. If no machine is detected in this zone, the count of this machine is reduces by a decay factor k, e.g. for $k = 0.5$, the number of observations is reduced by a half. The arriving observations are collected by the team-server, and as a certain confidence is achieved, i.e. the number of observations of a machine in a particular zone reaches a threshold O_{min}, these zones are reported. For the light observation, it is necessary that the machine of the observed light is observed in the zone, the machine is in using the derived belief. If this is correct the light pattern is reported. As it can be seen, there are two major parameters for the exploration phase to be chosen, i.e. k and O_{min}. The influence of this parameters is evaluated in Section 7.

Input: gamefield
Output: zone

```
1: belief = getConflictFreeGameField(gamefield)
2: mirr = getSwappedZonesNotExplored()
3: unex = getUnexploredZonesSortByMCount(belief)
4: ex = getExploredZonesSortByMCount(belief)
5: light = getZonesWLightReportSortByMCount(belief)
6: zones = queue(mirr, unex, ex, light)
7: removeZonesWithZeroCount(zones)
8: if zones = ∅ then
9:    nZone = randomZone()
10: else
11:    nZone = zones.firstElement()
12: end if
13: nZone.state = SCHEDULED
14: return nZone
```

Algorithm 4: Algorithm to get next zone to explore.

Chapter 7

Evaluation

In the context of this thesis, a complete robot framework for autonomous agents in the context of Industrie 4.0 was developed. Therefore, it is only possible to evaluate specific parts of this software.

The complete software stack is used during the exploration phase, i.e. high-level scheduling algorithms, the mid-level using the procedural reasoning engine down to the low-level components to perform e.g. way-point navigation. Therefore, it makes a lot of sense to use this as an evaluation testbed. For this purpose, the RoboCup Logistic League simulation using Gazebo with the Referee Box can be used to generate some kind of evaluation criteria, e.g. the awarded points during this phase.

As described in Section 6.5.2, for the exploration scheduler there are two main parameters defining the behavior: the penalizing factor k reducing the number of observations for a machine in a zone if there is no machine detected; and the threshold O_{min} defining when a machine is reported. The performance is compared to a similar approach that was used in the RoboCup Logistics League championship in 2016.

7.1 Evaluation Setup

The used base-line for the developed scheduler and the overall testing environment is described in this section. Furthermore, the chosen reward values are motivated and defined.

7.1.1 Base-Line Scheduler

To have a reference value for the implemented approach, a similar scheduler as it was used in the last competition was implemented for comparison purpose. This scheduler distributes the zones to explore in a random fashion, i.e. which zone next is chosen randomly. Having one zone, one of the free robots is chosen to perform this task. If a machine is found in this zone, the robot tries to detect the light pattern. The zone of the machine is then derived from the accumulated observations (i.e. maximum votes) and reported.

If a machine of the opponent team is detected, the result is mirrored and reported. With this, we have a rather advanced base-line for our scheduler (using this approach, we were able to achieve the third place in the RoboCup world championship).

7.1.2 Evaluation Criteria

First of all, an **E**valuation **C**riteria (EC) needs to be found reflecting the performance of the system. The resulting numbers after each game are:

- Number of correct reported zones Z_r.
- Number of wrong reported zones Z_w.
- Number of correct reported lights L_r.
- Number of wrong reported lights L_w.

Having this, derived values can be calculated (as described in Section 2.5 and the rulebook[1]). First of all, the gained points due to correct reported zones P_z as

$$P_z = 3 \cdot Z_r - 2 \cdot Z_w. \tag{7.1}$$

Similarly, the points for correct reported lights P_l can be calculated as

$$P_l = 5 \cdot L_r - 4 \cdot L_w \tag{7.2}$$

and finally, the overall points P as

$$P = P_z + P_l. \tag{7.3}$$

The number of achieved points depend strongly on the chosen game-field arrangement. Therefore, an other factor is needed to describe the overall accuracy of the system. For this, the number of zones reported is defined as

$$R_z = Z_r + Z_w \tag{7.4}$$

and the number of lights reported is defined as

$$R_l = L_r + L_w. \tag{7.5}$$

With this, the accuracy of the system can be defined as

$$\text{acc}_Z = \frac{Z_r}{R_z} \tag{7.6}$$

$$\text{acc}_L = \frac{L_r}{R_l} \tag{7.7}$$

[1] www.robocup-logistics.org/rules

To analyze the influence of different scheduling/perception parameters, several combinations are tested:

- $k \in (0.0, 0.5, 1.0)$
- $O_{min} \in (0, 25, 50)$

For each of the parameter pairs, ten runs are performed (i.e. $3 \cdot 3 \cdot 10 = 90$ simulations). The bar notation (e.g. $\overline{\text{acc}}_z$) is used to denote an average over the runs.

7.1.3 Experiment Procedure

To ensure independent runs, between all the simulations the complete software stack was restarted. To allow reconstruction of the experiments, the complete information flow was logged in a SQL database. The dump of this database as well as the output of the referee box is saved.

For this evaluation, 90 runs have been performed with the proposed scheduler using different parameter pairs. Additionally, 20 runs are performed with the base-line scheduler for comparison reasons.

7.2 Influence of the Parameters

In this section, the influence of the parameters k and O_min is examined and the results are compared to those of the base-line approach.

7.2.1 Dependency Between the Factors

With the defined evaluation criteria in Section 7.1.2, we can examine the dependency of these two factors on the performance. With this, the dependency of the accuracy to report a zone can be seen in Figure 7.1. Here it is clearly visible that the influence of the threshold O_{min} is very strong on the mean of the accuracy. This behavior is further examined in Section 7.2.3. On the other hand, the penalizing factor k do not have such a big influence. This is further examined in Section 7.2.3.

7.2.2 Influence of the Threshold Factor

The influence of the threshold factor on the accuracy can be seen in Figure 7.2. Here, the influence of the penalizing factor is very small.

To get the influence of the threshold factor only, the results of the different values for k are combined. Having this, a box plot can be used to visualize the results in an structured way. In such a plot, the median (red line), a box

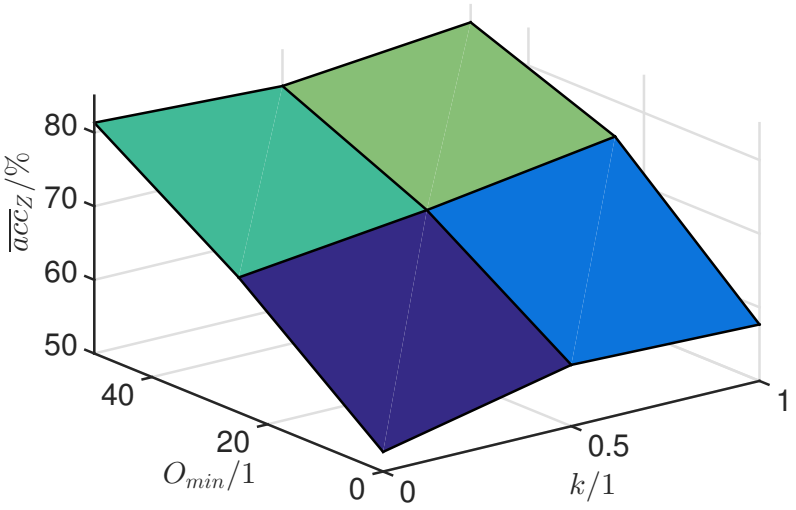


FIGURE 7.1: Influence of the penalizing factor k and the minimum observations threshold O_{min} on the averaged (10 runs) accuracy $\overline{\text{acc}}_Z$ of the zone reports.

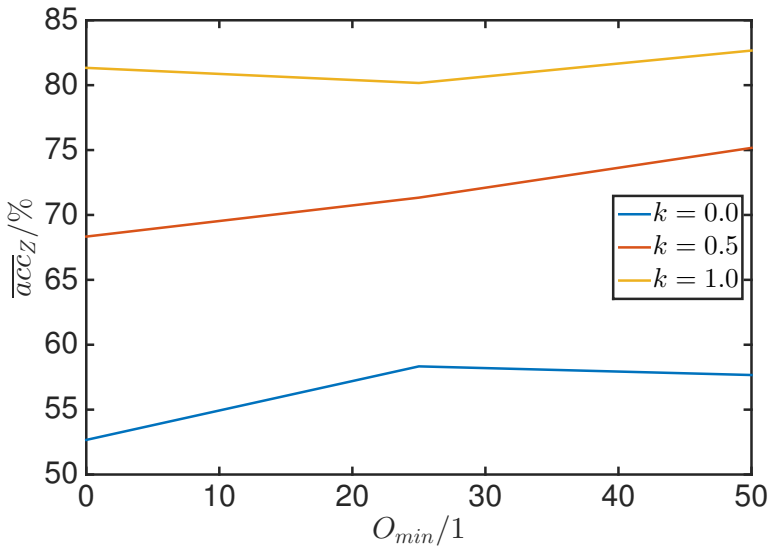


FIGURE 7.2: Influence of the minimum observations threshold O_{min} on the averaged (10 runs) accuracy $\overline{\text{acc}}_Z$ of the zone reports.

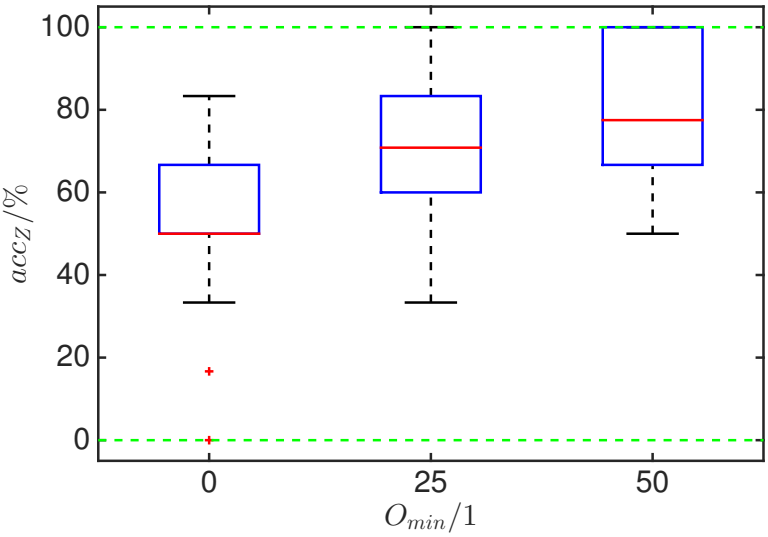


FIGURE 7.3: Box plot of the the zone reporting accuracy acc_Z over the minimum observations threshold O_{min}.

limited by the upper and lower quartile (blue box), the whiskers (black lines) defining the overall value range, and outliers (red plus) defined as values as far away from the median as 1.5 times the difference between the upper and lower quartile are illustrated. Such a plot for the overall accuracy over the different values of O_{min} can be seen in Figure 7.3. Here, the accuracy increases with increasing threshold. This behavior makes sense as more and more observations of a machine in a zone are needed before the zone is reported. Therefore, this factor also influences the overall number of reported zones. This relation can be seen in Figure 7.4. Here, for $O_{min} = 0$ almost every time all six machines are reported with only some outliers at five. On the other hand, reporting six zones is seen as an outlying value for the configuration with $O_{min} = 50$. Therefore, a compromise needs to be made between a good accuracy, and therefore reliable and correct reports and the readiness to take risks (i.e. to send a wrong report). A good way to find such a value is to take a look at the achieved points due to zone reports. This reflects the game policies of "how bad" a wrong report is, i.e. how much penalty is given. A box plot of the received points over the parameter O_{min} can be seen in Figure 7.5. This picture gives us several options to chose the value O_{min} depending on the willingness to take risks. As it can be seen, the highest points are achieved using $O_{min} = 25$ as a threshold for reporting, but this can lead to lower overall points too. The median of the configuration using a threshold of 50 is almost the same as the one of 25, but here it is not very likely to achieve more points. Therefore, a value around 25 or between 25 and 50 might be chosen for a good compromise.

7.2.3 Influence of the Penalizing Factor

The influence of the of the penalizing factor k is examined in the same way as the threshold before. For this, the impact on the zone reporting accuracy

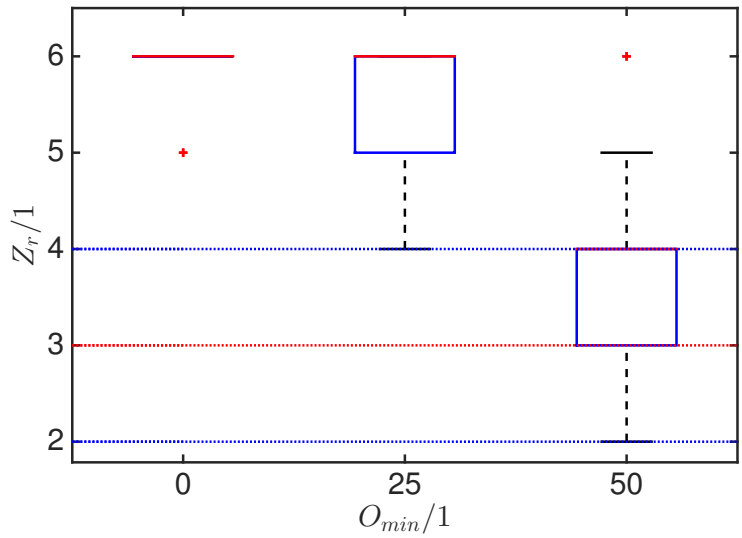


FIGURE 7.4: Box plot of the the number of reported machines (correct and wrong) R_Z over the minimum observations threshold O_{min}. Results of the base-line approach drawn dotted in red for the median and blue for the quartiles.

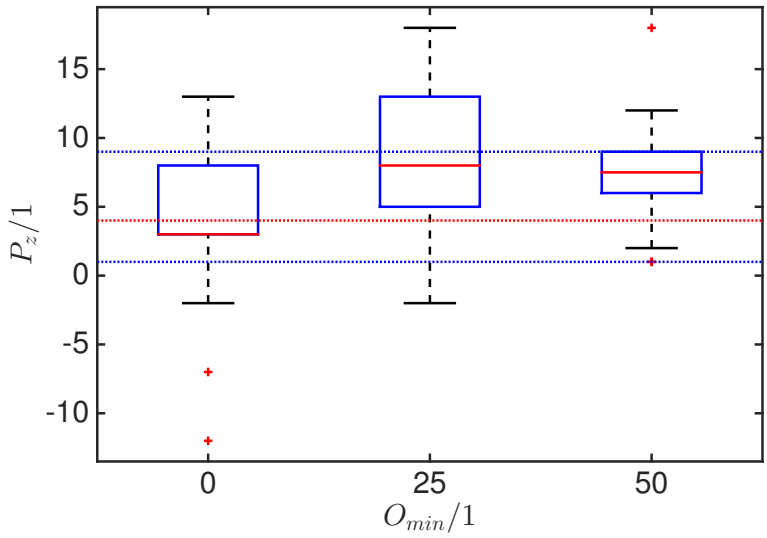


FIGURE 7.5: Box plot of the the number of received points P_z over the minimum observations threshold O_{min}. Results of the base-line approach drawn dotted in red for the median and blue for the quartiles.

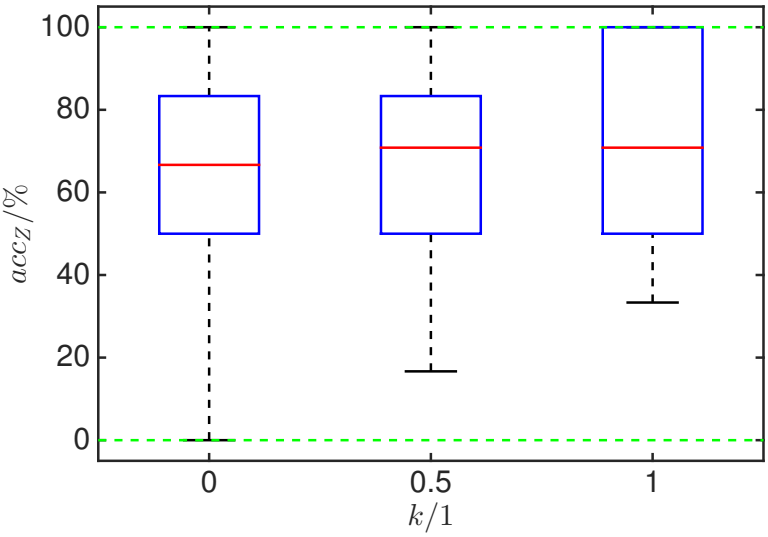


FIGURE 7.6: Box plot of the the zone reporting accuracy acc_Z over the penalizing factor k.

is shown for all values of O_{min} in Figure 7.6. Here, again, a higher value of k leads to a better accuracy. With this result alone, $k = 1$ would seem ideal, but if we take a look at the influence on the overall reported machines, we need to make a compromise again. This dependence depicted in Figure 7.7 suggests a value around 0.5. Again, the way to find a compromise is to take a look at the gained points for the reports, depicted in Figure 7.8. Similar to the case for the threshold, the decision to chose this factor depends on the readiness to take risks. A reasonable value for this factor would be between 0.5 and 1.0.

7.2.4 Influence on the Reported Lights

The influence on the number of reported lights needs not to be taken into account for two reasons: the accuracy of the reports is at almost 1.0, independent of the parameters and the points gained through reported lights behave like those described above for the reported zones. With this, the chosen parametrization for a good result in zone exploration leads to a high number of points for the lights too. This can be again seen in the plots for the achieved points through light reporting in Figure 7.9 and Figure 7.10.

7.3 Comparison with Base-Line Scheduler

The results of the base-line scheduler are drawn with dotted lines in the plots, i.e. red dotted line for the median, and blue dotted lines for the upper and lower quartile. The results are the combination of 20 independent runs.

As it can be seen in the figures before (evaluating the different parameters of the scheduler), the developed scheduler works for almost all parameters chosen

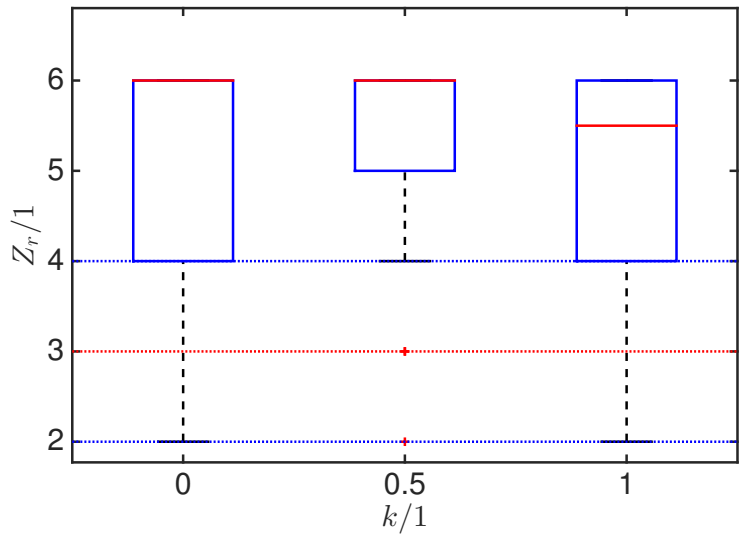


FIGURE 7.7: Box plot of the the number of reported machines (correct and wrong) R_Z over the penalizing factor k. Results of the base-line approach drawn dotted in red for the median and blue for the quartiles.

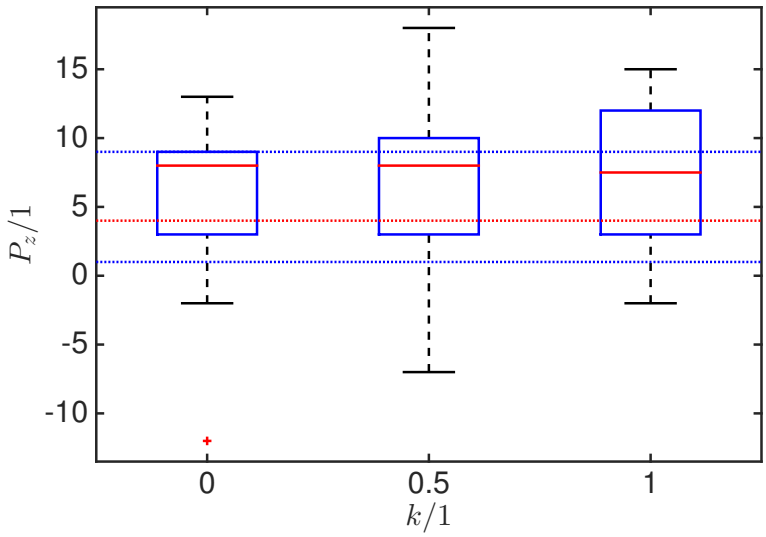


FIGURE 7.8: Box plot of the the number of received points P_z over the penalizing factor k. Results of the base-line approach drawn dotted in red for the median and blue for the quartiles.

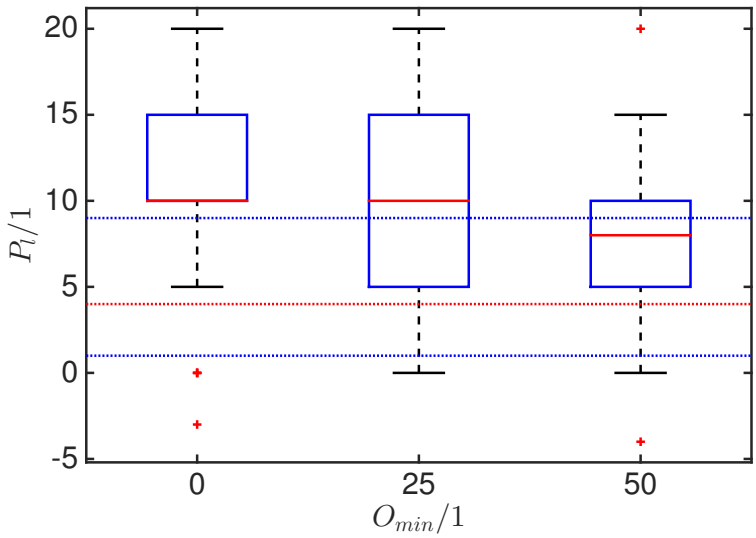


FIGURE 7.9: Box plot of the the number of received points P_l over the minimum observations threshold O_{min}. Results of the base-line approach drawn dotted in red for the median and blue for the quartiles.

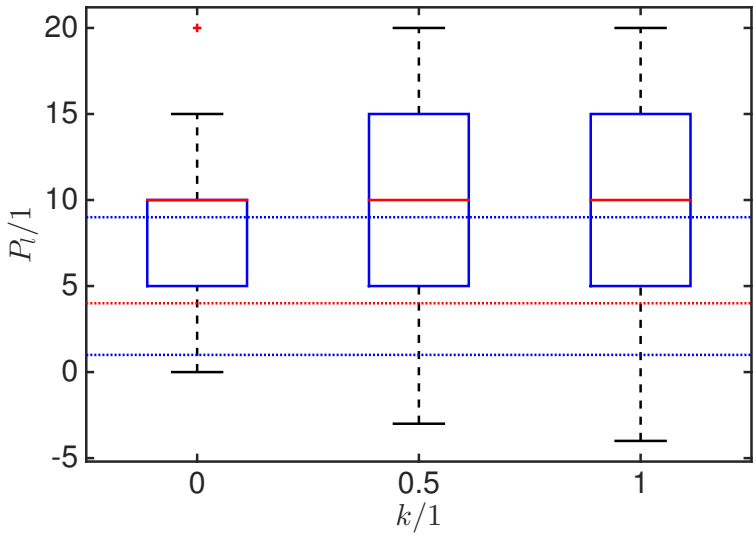


FIGURE 7.10: Box plot of the the number of received points P_l over the penalizing factor k. Results of the base-line approach drawn dotted in red for the median and blue for the quartiles.

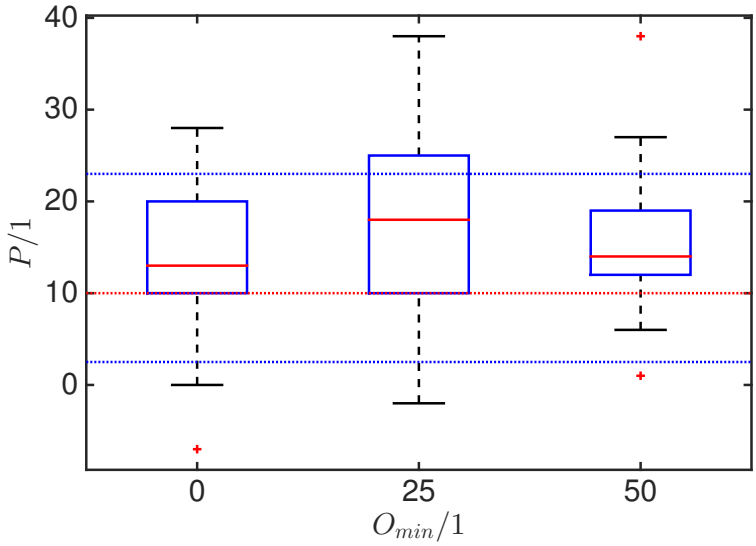


FIGURE 7.11: Box plot of the the number of overall received points P over the minimum observations threshold O_{min}. Results of the base-line approach drawn dotted in red for the median and blue for the quartiles.

better (i.e. higher median) and with less variance. Therefore, the approach works more reliable. This can be further analyzed by viewing at the overall points gained as it can be seen in Figure 7.11 and Figure 7.12. Here, the same behavior can be observed: the median of the achieved overall points is higher and the result is more stable if the correct parameters are used (smaller quantiles and therefore lower variance).

The results show, that the implemented scheduler works better than the previous implementation used at the RoboCup Logistic world championship, especially with parameters k between 0.5 and 1 and $O_{min} = 25$.

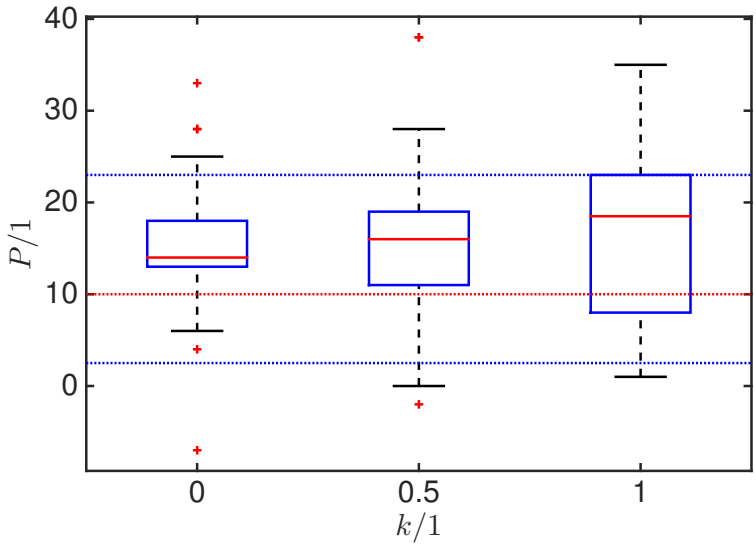


FIGURE 7.12: Box plot of the the number of overall received points P over the penalizing factor k. Results of the base-line approach drawn dotted in red for the median and blue for the quartiles.

Chapter 8

Conclusion

In this master's thesis, a robot framework for a fleet of autonomous robots was presented. The framework is designed in a layered and flexible way as each layer can be exchanges easily. The architecture consists of a centralized high-level control and distributed mid- and low-level layers. The communication is mainly based on TCP using Google's protocol buffers for the serialization. As an practical example, the software stack needed for the RoboCup Logistics league is implemented and tested. Furthermore, the problem to explore an unknown factory hall is formalized and an exploration scheduler is implemented.

8.1 Modularity

The modularity of the developed framework has already shown its advantages as the high-level is already implemented in a second version. The first implementation used C++, and the presented version, in this thesis, Java for this. This change in the programing language was possible due to the clear layered architecture and the use of the language independent serialization protocol.

With this, the power of this framework is not limited, i.e. all of the layers can be redesigned and new functionality can be added very easy. This makes this work a great fundamental for future projects in the scope of autonomous multi-robot systems with a centralized high-level. The abstraction between the levels allows independent development and evaluated.

8.2 Improvement of Reliability in Industrie 4.0

With this framework, each layer can be tested individually. This lead to a huge increase in testability. For this, an adjacent layer can simply be mocked up by implementing the interfaces, e.g. the low-level can be modeled using deterministic behavior to test the mid-level and high-level components. On the other side, actions can be sent to the low-level from a test-engine as well, testing the behavior of the hardware components.

8.3 Participation and Results

The described architecture has already been tested by participating in the RoboCup 2016 in Leipzig, Germany. The software version used there was in a beta state with a minimalist scheduler as described in the team description paper in [61]. The architecture and low-level components were able to show off their robustness and efficiency, and therefore it was possible to reach the third place as a newcomer against opponents competing in this league for already several years.

Chapter 9

Future Work

With this robot framework as a basis, more complex functionality can be added in each of the three layers. The next evolution steps of this framework contains such improvements in all three layers. A short preview of the main planned additional functionality is given in this chapter.

9.1 Production Scheduler

At the high-level control, i.e. the highest layer, at the moment only a scheduler to explore the game-field is implemented. To be able to produce ordered products during the production phase, a production scheduler needs to be developed too. For this, the arriving orders need to be split up into atomic subtasks. These tasks can then be scheduled using a earliest-deadline-first approach or one of the presented algorithms in Section 4

This additional scheduler can be added to the present schedulers as described in Section 6.

9.2 Diagnosis for Error Detection and Handling

The low-level layer of the developed framework starts to become rather complex, i.e. a lot of different processes are started there. To be able to detect errors, different approaches can be used as described in [62] or [63]. As an example, the nominal behavior of the processes (e.g. frequency of communication, used bandwidth) can be used to model a normal operating mode. Deviations from this behavior can therefore be recognized and an error handling or error recovery can be started. This would introduce an enormous improvement in stability and reliability.

Bibliography

[1] Nasser Jazdi. "Cyber physical systems in the context of Industry 4.0". In: *Automation, Quality and Testing, Robotics, 2014 IEEE International Conference on*. IEEE. 2014, pp. 1–4.

[2] Heiner Lasi et al. "Industry 4.0". In: *Business & Information Systems Engineering* 6.4 (2014), p. 239.

[3] Rainer Drath and Alexander Horch. "Industrie 4.0: Hit or hype?[industry forum]". In: *IEEE industrial electronics magazine* 8.2 (2014), pp. 56–58.

[4] Tim Niemueller et al. "RoboCup logistics league sponsored by Festo: a competitive factory automation testbed". In: *Robot Soccer World Cup*. Springer. 2013, pp. 336–347.

[5] Hiroaki Kitano et al. "Robocup: The robot world cup initiative". In: *Proceedings of the first international conference on Autonomous agents*. ACM. 1997, pp. 340–347.

[6] Marco Wallner et al. "A Robust and Flexible Software Architecture for Autonomous Robots in the Context of Industrie 4.0". In: *Austrian Robotics Workshop* (2017), pp. 67–73.

[7] Tim Niemueller et al. "Knowledge-Based Instrumentation and Control for Competitive Industry-Inspired Robotic Domains". In: *KI-Künstliche Intelligenz* 30.3-4 (2016), pp. 289–299.

[8] Edsger W Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[9] Alberto Elfes. "Occupancy grids: A stochastic spatial representation for active robot perception". In: *Proceedings of the Sixth Conference on Uncertainty in AI*. Vol. 2929. 1990.

[10] Hubert Zimmermann. "OSI reference model-the ISO model of architecture for open systems interconnection". In: *IEEE Transactions on communications* 28.4 (1980), pp. 425–432.

[11] Rodney Brooks. "A robust layered control system for a mobile robot". In: *IEEE journal on robotics and automation* 2.1 (1986), pp. 14–23.

[12] Erann Gat et al. "On three-layer architectures". In: *Artificial intelligence and mobile robots* 195 (1998), p. 210.

[13] R Peter Bonasso et al. "Experiences with an architecture for intelligent, reactive agents". In: *Journal of Experimental & Theoretical Artificial Intelligence* 9.2-3 (1997), pp. 237–256.

[14] Brian P Gerkey and Maja J Matarić. "A formal analysis and taxonomy of task allocation in multi-robot systems". In: *The International Journal of Robotics Research* 23.9 (2004), pp. 939–954.

[15] Ernesto Nunes, Mitchell McIntire, and Maria Gini. "Decentralized allocation of tasks with temporal and precedence constraints to a team of robots". In: *Simulation, Modeling, and Programming for Autonomous*

Robots (SIMPAR), IEEE International Conference on. IEEE. 2016, pp. 197–202.

[16] Xing Su, Minjie Zhang, and Quan Bai. "Coordination for dynamic weighted task allocation in disaster environments with time, space and communication constraints". In: *Journal of Parallel and Distributed Computing* 97 (2016), pp. 47–56.

[17] Sofia Amador, Steven Okamoto, and Roie Zivan. "Dynamic multi-agent task allocation with spatial and temporal constraints". In: *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*. International Foundation for Autonomous Agents and Multiagent Systems. 2014, pp. 1495–1496.

[18] E Gil Jones, M Bernardine Dias, and Anthony Stentz. "Time-extended multi-robot coordination for domains with intra-path constraints". In: *Autonomous robots* 30.1 (2011), pp. 41–56.

[19] Fang Tang and Lynne E Parker. "A complete methodology for generating multi-robot task solutions using asymtre-d and market-based task allocation". In: *Robotics and Automation, 2007 IEEE International Conference on*. IEEE. 2007, pp. 3351–3358.

[20] Marc Pujol-Gonzalez et al. "Efficient inter-team task allocation in robocup rescue". In: *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*. International Foundation for Autonomous Agents and Multiagent Systems. 2015, pp. 413–421.

[21] James Parker et al. "Exploiting spatial locality and heterogeneity of agents for search and rescue teamwork". In: *Journal of Field Robotics* (2015).

[22] Justin Melvin et al. "Multi-robot routing with rewards and disjoint time windows". In: *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*. IEEE. 2007, pp. 2332–2337.

[23] Sameera Ponda et al. "Decentralized planning for complex missions with dynamic communication constraints". In: *American Control Conference (ACC), 2010*. IEEE. 2010, pp. 3998–4003.

[24] Mitchell McIntire, Ernesto Nunes, and Maria Gini. "Iterated multi-robot auctions for precedence-constrained task scheduling". In: *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*. International Foundation for Autonomous Agents and Multiagent Systems. 2016, pp. 1078–1086.

[25] Lingzhi Luo, Nilanjan Chakraborty, and Katia Sycara. "Distributed algorithms for multirobot task assignment with task deadline constraints". In: *IEEE Transactions on Automation Science and Engineering* 12.3 (2015), pp. 876–888.

[26] G Ayorkor Korsah et al. "xBots: An approach to generating and executing optimal multi-robot plans with cross-schedule dependencies". In: *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. IEEE. 2012, pp. 115–122.

[27] Matthew C Gombolay, Ronald Wilcox, and Julie A Shah. "Fast Scheduling of Multi-Robot Teams with Temporospatial Constraints." In: *Robotics: Science and Systems*. 2013.

[28] Tim Niemueller, Gerhard Lakemeyer, and Alexander Ferrein. "The robocup logistics league as a benchmark for planning in robotics". In: *WS on Planning and Robotics (PlanRob) at Int. Conf. on Aut. Planning and Scheduling (ICAPS)*. 2015.

[29] Jonathan F Bard, George Kontoravdis, and Gang Yu. "A branch-and-cut procedure for the vehicle routing problem with time windows". In: *Transportation Science* 36.2 (2002), pp. 250–269.

[30] Anders Dohn, Esben Kolind, and Jens Clausen. "The manpower allocation problem with time windows and job-teaming constraints: A branch-and-price approach". In: *Computers & Operations Research* 36.4 (2009), pp. 1145–1157.

[31] Cynthia Barnhart, Christopher A Hane, and Pamela H Vance. "Using branch-and-price-and-cut to solve origin-destination integer multicommodity flow problems". In: *Operations Research* 48.2 (2000), pp. 318–326.

[32] Maria Gini. "Multi-robot Allocation of Tasks with Temporal and Ordering Constraints". In: (2017), pp. 4863–4869.

[33] Rajiv T Maheswaran et al. "Taking DCOP to the real world: Efficient complete solutions for distributed multi-event scheduling". In: *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*. IEEE Computer Society. 2004, pp. 310–317.

[34] Robert Junges and Ana LC Bazzan. "Evaluating the performance of DCOP algorithms in a real world, dynamic problem". In: *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 2*. International Foundation for Autonomous Agents and Multiagent Systems. 2008, pp. 599–606.

[35] Sarvapali D Ramchurn et al. "Decentralized coordination in robocup rescue". In: *The Computer Journal* (2010), pp. 1447–1461.

[36] James Parker, Alessandro Farinelli, and Maria Gini. "Max-sum for allocation of changing cost tasks". In: *International Conference on Intelligent Autonomous Systems*. Springer. 2016, pp. 629–642.

[37] Alessandro Farinelli, Alex Rogers, and Nick R Jennings. "Agent-based decentralised coordination for sensor networks using the max-sum algorithm". In: *Autonomous agents and multi-agent systems* 28.3 (2014), pp. 337–380.

[38] Paul Scerri et al. "Allocating tasks in extreme teams". In: *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*. ACM. 2005, pp. 727–734.

[39] Alessandro Farinelli et al. "Assignment of dynamically perceived tasks by token passing in multirobot systems". In: *Proceedings of the IEEE* 94.7 (2006), pp. 1271–1288.

[40] Bradford Heap and Maurice Pagnucco. "Minimising Undesired Task Costs in Multi-Robot Task Allocation Problems with In-Schedule Dependencies." In: *AAAI*. Citeseer. 2014, pp. 2542–2548.

[41] Sanem Sariel-Talay, Tucker R Balch, and Nadia Erdogan. "Multiple traveling robot problem: A solution based on dynamic task selection and

robust execution". In: *IEEE/ASME Transactions On Mechatronics* 14.2 (2009), pp. 198–206.

[42] Maitreyi Nanjanath and Maria Gini. "Repeated auctions for robust task execution by a robot team". In: *Robotics and Autonomous Systems* 58.7 (2010), pp. 900–909.

[43] Brian P Gerkey and Maja J Matarić. "Sold!: Auction methods for multirobot coordination". In: *IEEE transactions on robotics and automation* 18.5 (2002), pp. 758–768.

[44] Michael M Zavlanos, Leonid Spesivtsev, and George J Pappas. "A distributed auction algorithm for the assignment problem". In: *Decision and Control, 2008. CDC 2008. 47th IEEE Conference on*. IEEE. 2008, pp. 1212–1217.

[45] Han-Lim Choi, Luc Brunet, and Jonathan P How. "Consensus-based decentralized auctions for robust task allocation". In: *IEEE transactions on robotics* 25.4 (2009), pp. 912–926.

[46] Tim Niemueller, Sebastian Reuter, and Alexander Ferrein. "Fawkes for the robocup logistics league". In: *Robot Soccer World Cup*. Springer. 2015, pp. 365–373.

[47] M Douglas McIlroy et al. "Mass-produced software components". In: *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*. 1968, pp. 88–98.

[48] Mark Collins-Cope. "Component based development and advanced OO design". In: *White Paper* (2001).

[49] Tim Niemueller et al. "The carologistics approach to cope with the increased complexity and new challenges of the RoboCup logistics league 2015". In: *Robot Soccer World Cup*. Springer. 2015, pp. 47–59.

[50] Robert M Wygant. "CLIPS—A powerful development and delivery expert system tool". In: *Computers & Industrial Engineering* 17.1-4 (1989), pp. 546–549.

[51] Frederik Zwilling, Tim Niemueller, and Gerhard Lakemeyer. "Simulation for the RoboCup logistics league with real-world environment agency and multi-level abstraction". In: *RoboCup 2014: Robot World Cup XVIII*. Springer, 2014, pp. 220–232.

[52] Michael Georgeff et al. "The belief-desire-intention model of agency". In: *Intelligent Agents V: Agents Theories, Architectures, and Languages*. Springer, 1998, pp. 1–10.

[53] Francois Fe'lix Ingrand et al. "PRS: A high level supervision and control language for autonomous mobile robots". In: *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*. Vol. 1. IEEE. 1996, pp. 43–49.

[54] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. 3.2. 2009, p. 5.

[55] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. "The dynamic window approach to collision avoidance". In: *IEEE Robotics & Automation Magazine* 4.1 (1997), pp. 23–33.

[56] Navneet Dalal and Bill Triggs. "Histograms of oriented gradients for human detection". In: *Computer Vision and Pattern Recognition, 2005. CVPR*

2005. IEEE Computer Society Conference on. Vol. 1. IEEE. 2005, pp. 886–893.

[57] Richard Lippmann. "An introduction to computing with neural nets". In: *IEEE Assp magazine* 4.2 (1987), pp. 4–22.

[58] John S Bridle. "Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition". In: *Neurocomputing*. Springer, 1990, pp. 227–236.

[59] Karl Johan Astrom and Lars Rundqwist. "Integrator windup and how to avoid it". In: *American Control Conference, 1989*. IEEE. 1989, pp. 1693–1698.

[60] Martin Fodslette Møller. "A scaled conjugate gradient algorithm for fast supervised learning". In: *Neural networks* 6.4 (1993), pp. 525–533.

[61] Sarah Haas et al. "RoboCup Logistics League TDP Graz Robust and Intelligent Production System GRIPS". In: (2016).

[62] Safdar Zaman et al. "An integrated model-based diagnosis and repair architecture for ROS-based robot systems". In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE. 2013, pp. 482–489.

[63] Raymond Reiter. "A theory of diagnosis from first principles". In: *Artificial intelligence* 32.1 (1987), pp. 57–95.