TU Graz

Jakob Auer, BSc

Autonomous Shoring Task

Master's Thesis

Graz University of Technology

Institute for Softwaretechnology
Institute Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Supervisor: Ass.Prof. Dipl.-Ing. Dr.techn. Gerald Steinbauer

Graz, March 2016

Statutory Declaration

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 marked all material which has been quoted either literally or by content from the used sources.

Graz, ______________________ ______________________________

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am ______________________ ______________________________

Datum Unterschrift

[1]Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

A very dangerous job in disaster areas (e.g. after earthquakes) is the underpinning of buildings which are in danger to collapse. This shoring task is done by humans to support the building's structure.

The goal of this thesis was to develop a planning system and a robotic demonstrator that is able to solve the shoring task autonomously. The planner system combines task and motion planning in order to allow flexible planning and robust execution of the shoring in difficult environments. The proposed approach uses abstract and geometric planners which are coupled by a symbol grounding module. In order to allow flexibility in building the structures the abstract planner is based on the STRIPS planning approach. It solves the problem without any geometrical knowledge but uses the geometric planner for the verification of the applicability of actions in the real world. This geometric planner possesses detailed knowledge about the environment and is able to determine if these actions are executable. The abstract planner uses the feedback from the geometric planner to create a plan which is valid from the abstract as well as the geometric point of view. In order to ease the geometric planning the mobile manipulation problem was separated into a navigation and manipulation problem that maps to the corresponding abstract actions.

To demonstrate the feasibility of the proposed approach a prototype implementation was realized. The implementation is based on well known libraries such as the Robot Operating System (ROS), MoveBase and MoveIt!. We built a robot based on a six wheeled differential drive rover and an industrial arm to demonstrate the planning capabilities of the proposed system. The prototype robot was able to successfully build towers in a moderate complex lab environment. It is able to grasp blocks on defined positions and to place them into the tower with respect to the environment and obstacles.

To improve the planning performance several optimizations to the abstract planning system were developed. In order to evaluate the proposed planning system we conducted simulated and real experiments. The simulated experiments showed that the planning process can be handled in suitable time and the planning system is able to find plans that are executable in the real world. The real experiments showed that the implemented real robot system is able to build a shoring tower autonomously and reliably.

In future work we will work on the integration of state-of-the-art planning algorithms and use of direct feedback from the robot's perception system.

Acknowledgments

Firstly, I would like to thank my supervisor Gerald Steinbauer for supporting me during my work on this thesis. He encouraged me with his advise and feedback during the different phases of this work. His guidance helped me in all the time.

I would like to thank my girlfriend Michaela who always supports me in everything I do. Without you Michaela this work would not have been possible.

Thank you to my parents Harriet and Johann for all your support during my whole education and life. You always encouraged me to pursue my goals and I am very grateful for that.

Last but not least I would like to thank all my study colleagues for the great time during our studies in Graz.

Thank you!
Jakob

Contents

Abstract v

Acknowledgements vi

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Goals and Challenges . . . 2
- 1.3 Contribution . . . 3
- 1.4 Outline . . . 4

2 Related Work 5

3 Prerequisites 9
- 3.1 Robot Operating System . . . 9
- 3.2 MoveIt! . . . 10
- 3.3 MoveBase . . . 12
- 3.4 STRIPS . . . 14

4 Concept - System Overview 15
- 4.1 Modularization . . . 15
- 4.2 Abstract Domain Definition . . . 16
  - 4.2.1 Sorts . . . 16
  - 4.2.2 Constants . . . 17
  - 4.2.3 Predicates . . . 18
  - 4.2.4 Functions . . . 18
  - 4.2.5 Actions . . . 20
- 4.3 Symbol Grounding . . . 21
  - 4.3.1 Real World Coordinates . . . 23
  - 4.3.2 Robots Base and End-effector Pose . . . 28
- 4.4 Geometric planning . . . 32
  - 4.4.1 Robot Base Navigation . . . 32
  - 4.4.2 Robot Arm Movement - Pick and Place . . . 34

5 Implementation 35
- 5.1 Handling a planning request . . . 35
- 5.2 Abstract high level planning in PROLOG . . . 39
  - 5.2.1 State representation . . . 39
  - 5.2.2 Successor Arithmetic . . . 43

5.2.3 Actions . . . 44
5.2.4 Solve the planning problem . . . 48
5.2.5 Optimizations . . . 50
5.3 Geometric Planner - Dispatcher . . . 54
5.3.1 Planner Interface . . . 54
5.3.2 Geometry Publisher . . . 54
5.3.3 Gripper Interface . . . 54
5.4 2D Navigation - MoveBase . . . 55
5.4.1 Localization . . . 55
5.4.2 Move the robot . . . 57
5.4.3 Environment Simulation . . . 58
5.5 Object Manipulation - MoveIt! . . . 62
5.5.1 Simulation - Fake Joint Controllers . . . 64
5.6 Real Robot . . . 65

6 Evaluation and Experiments 67
6.1 Abstract Planner Evaluation . . . 67
6.1.1 Logistic Domain . . . 69
6.2 Abstract and Geometric Planner Evaluation . . . 74

7 Conclusion 77

8 Future Work 79
8.1 Perception . . . 79
8.2 Grasping the blocks from the floor . . . 79
8.3 Improved goal position selection . . . 79
8.4 Improved grasping poses . . . 80
8.5 Improved planner . . . 80

Bibliography 85

1 Introduction

1.1 Motivation

The so called shoring task is a job which is done on disaster sites to support damaged buildings. This is used after catastrophes like earth quakes, hurricanes or similar destructive events. For that purpose towers are built with wooden blocks. Usually there are two blocks on each level but there are several different variants of building them. These towers can be built around damaged piles but can also be built as free standing elements to support the ceilings. There are various guides and standards on how the different types of shoring a building works. Two examples are the guides form the United Nations International Search and Rescue Advisory Group (INSARAG)[1] and the United States Army Corps of Engineers' Shoring Operations Guide [2].

Figure 1.1: A real world disaster site [3]

Figure 1.1 shows a real world disaster site where the shoring task could be applied. It is also apparent that this task is very dangerous for the responder. All these buildings could collaps at any time. To protect these workers we want to execute the shoring task autonomously with robots. If the robots do this task the helpers can stay outside these dangerous areas.

The RoboCup Rescue League is an international event where teams of students and researchers compete against each other. Their goal is to build rescue robots which are capable of solving challenges which are necessary in disaster areas (e.g. after earthquakes). Among other challenges the RoboCup Rescue League defined a set of rules for a mobile manipulation and shoring task mission [4][5]. The rules define to build a tower around a given pole (as defined in Figure 1.2) with a block size of $10cm \times 10cm \times 60cm$.

Figure 1.2: Example shoring task [5]

To build these towers it is necessary to have a robot which is able to move on the ground from an area where it grabs blocks to an area where it builds the tower. As mentioned the robot has to be able to grab, carry and place blocks.

1.2 Goals and Challenges

The goal of this thesis was to develop a robot system that is able to build a shoring tower autonomously. The basic concept for solving the shoring task is mobile manipulation. In order to be flexible in the building task as well as the environment we heavily rely on planning. I order to realize such planning we have to combine classical task planning and motion planning for robot arms and platforms. The involved task and motion planner act at very different abstraction levels. The task planner has a very abstract view on the tank and the environment with no geometric information. This abstract planner has a declarative description about the environment and the actions the robot can perform and solves the planning problem from a logic point of view. Because the task planner does not have geometric knowledge about the environment it cannot decide if an action

suitable on the abstract level is executable in the real world. Therefore, the task planner is coupled to a geometric planner which validates the intended abstract actions in the real environment. This module reports back to the abstract planner if a path for requested motion could be found and if the action is executable in reality. This is in particular important in cluttered environment where the individual actions might be executable, but a sequence of them is not even if it is perfectly fine on the abstract level.

To execute these tasks we used a robot based on the Roboterwerk Forbot. This is a six wheeled differential drive rover with integrated batteries and a motor control unit. For object manipulation a Schunk LWA4P arm was mounted on top of this base. In the front area of the robot a Sick LMS100 lidar sensor is used to sense the environment. Figure 1.3 shows the complete structure with the computer and the robot's sensors.

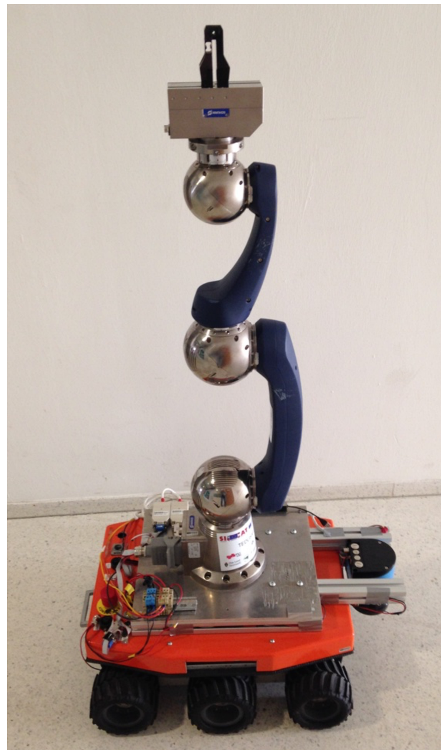

Figure 1.3: The robot

1.3 Contribution

This thesis shows that the combination of an abstract task planning and a geometric motion planning which are highly specialized but coupled planners results in an efficient solution for complex mobile manipulation problems in real world environments. The abstract planner is based on a STRIPS representation that allows to minimize the direct

coding of a solution and is realized as a PROLOG program. The geometric planning was realized using the Robot Operating System (ROS) and it's dedicated planners for 2D navigation(MoveBase) and 3D arm movements and manipulations(MoveIt!). In order to couple the two systems a bridge using Python was developed that takes care about the mapping of abstract to geometric representations and vice versa. The setup was tested in simulation and with a real robot. The results showed that the proposed system is able to build a shoring tower automatically even in cluttered environments.

1.4 Outline

The remainder of this thesis is as following. The next chapter briefly discusses related work. Chapter 3 introduces all the prerequisites used in the following chapters. The next chapter is a functional overview of the proposed system. The Chapter 5 describes in more details the implementation of the concepts. In Chapter 6 an evaluation of the proposed system based on simulations and real experiments is given. The final two chapters deal with future work (Chapter 8) and conclude this thesis (Chapter 7).

2 Related Work

Combined task and motion planning is a very active research field and there exist numerous works related to this topic and the actual thesis. This chapter provides a brief discussion of some similar and different approaches to solve this task.

The book Automated Planning Theory and Practice [6] is a great introduction into planning problems and the methods and approaches to handle them. Daniel Weld gives an overview [7] of propositional methods like Graphplan and compilers which are able to convert planning problems in a way that SAT solver are capable of solving them. Jörg Hoffmann also gives a good introduction into Fast-Forward planning methods in [8].

Christian Dornhege worked on a PDDL (Planning Domain Definition Language) planner with semantic attachments during his PhD [9][10]. These semantic attachments are an extension to a PDDL planner which are capable of calling external binaries. The action descriptions are split into modules consisting of three sub parts. These elements handle the calls for checking if an action is applicable, applying the action and calculate the resulting state and the cost determination for this action. All of these elements have logical parts which are processed by the PDDL planner and a semantic attachment in form of a call to an external library. So if the robot should for example move from A to B the planner is able to first determine the costs and to check if the transfer is possible. If all the requirements are met the action can be applied. They successfully used their system with an PR2 robot in a Tidy-Up-Robot domain. Compared with the planning system we used Dornhege extended the Temporal Fast Downward planner from Freiburg University.

Rachid Alami et al. developed their aSyMov planner [11][12] to target symbolic and geometric reasoning. The planner uses a combination of PDDL on the abstract side and multiple Probabilistic Roadmap Models (PRM) for the geometric validation. The planner deals with the difficult problem of object composition (difference between a robot and the same robot carrying a object) with different PRMs. It has a transfer roadmap for the robot with an object in it's gripper as well as a transit roadmap for the robot alone. From an top-level point of view to these roadmaps, they are linked to each other with grasping and placing operations of the robot. If the robot drives from A to B it is on the transit roadmap. When it then grasps an object on position B and wants to move on to position C it moves on the transfer roadmap. One of the key properties of their planner is that the state and the roadmaps are continuously expanded when necessary. This leads to a very robust plans with a good chance to be valid. So the planning process

is a search through multiple connected roadmaps until the given goal state is reached. They solved a simulated Tower of Hanoi problem with obstacles in the environment which make it very hard to find a valid plan.

L. P. Kaelbling and T. Lozano-Pérez use an hierarchical approach to plan in the now [13] to deal with the non-determinism in real environments. This planner generates an abstract level plan without geometric information and executes it. This approach has the risk of actions which are not executable. The important point in their plans is that all actions and steps are reversible. So if the planner executes actions and gets stuck somewhere because one planned abstract action is not executable in the real world it takes a step back and tries another plan. The realization plan is generated step by step. So if an action is executed successfully the realization plan for the next step is generated with the new observed environment.

Shinya Kimura et al. use a kinematics model for playing Jenga with a robot [14]. They built a kinematics model of the stable tower as well as for the transition periods where the robot grasps and removes a block. They used a generalized model for the five possible patters for each layer. Based on these patterns and a force from the upper layer they are able to calculate the force to the lower layer. Based on these models the robot selects the block which is the most safe to remove. On top of that their implementation also takes the acting force for removing the block into account. The robot only removes the block if the acting force and the kinematics model meet in a way the tower does not break down.

Another very popular and similar planning problem is the Tower of Hanoi challenge. Giray Havur et al. use the action description language C+ and the reasoner CCALC to solve this problem [15]. This reasoner translates the problem to a SAT (satisfiability problem) formulation and uses a SAT solver to calculate the solution. The planning domain is also extended with some geometric constraints which are taken into account during the planning process. To calculate the collision free trajectories they use probabilistic road maps (PRM) or rapid random tree exploration (RRT). With this approach they are able to solve and execute concurrent plans (with two robots) for solving the Tower of Hanoi problem.

Nathaneael Macias and John Wen developed a system to stack Jenga blocks with vision guidance [16]. They use a regular webcam mounted on the robot arm for the vision system. For the recognition and exact localization of the blocks the use ALVAR markers. Each block i tagged with one marker and they are all good visible to the camera. To select the block to grasp they apply several cost functions to each block. They use MATLAB as control and processing unit. The real world demonstrator stacked the blocks with but had some problems with manipulating other block when grasping one. This makes it necessary to perform the very costly block recognition and localization after each block was grasped.

The German Aerospace Center and Örebro University developed a hybrid planner for their humanoid two-arm robot Justin [17]. This planner is a combination of forward chaining task planning and bidirectional rapidly exploring random tree planning. They use geometric predicates in the pre- and post-conditions of operators for the binding between the task an the motion planner. With this additional information the task planner is able to plan in the abstract space with respect to added geometric information. They used these methods to perform real world experiments with their robot.

To connect objects or blocks in the real world with the abstract object in the planning environment all robotics systems need a very capable cognitive system. Symbol grounding [18] or object anchoring [19] is the process of linking sensor information and environment knowledge to actual objects. This connection has to work in both directions, from the abstract to the real world and vice verse. For example if a 3D camera sees a green block and the 2D laser scanner also sees an object on the same position these information can be merged to a description of a block with and identifier. The cognitive system has to be able track and recognize this block. So if it grasps and moves the block it has to be able to detect that the block is on another position. These problems are solved using object recognition algorithms and sensor fusion.

3 Prerequisites

This chapter gives a short introduction to the techniques and tools which are used in this thesis.

3.1 Robot Operating System

The Robot Operating System (ROS) [20] is an open source software project which enables software development for autonomous robots. It is built on a flat architecture and uses peer-to-peer networking technologies to communicate between the modules.
The architecture is defined by following concepts:

- **Nodes**
  Nodes are the computing or processing units. Different problems and processing steps are split into different nodes. Since this is a very modular approach nodes should be exchangeable.

- **Messages**
  If two or multiple nodes are communicating with each other they send messages between them. These messages are always a data structure which consists of a header and a payload. The header stores metadata like a consecutive sequence identificator or a timestamp. This enables ROS to communicate between nodes using different programming languages and even different operating systems.

- **Topics**
  The communication between nodes which is mentioned above is handled with topics. If a node sends a message to another node it publishes a message on a unique topic. Topics have a global scope and can be subscribed by nodes which need the provided information.

- **Master - ROS Core**
  The central module is called master or ROS core. This is a centralized registry of all available nodes and topics. If a node asks for a specific topic, the master looks for this topic in it's registry and returns all required information for a direct communication to the node which publishes on this topic.

- **Services**
  Services are a second way of communication in ROS. They provide a functionality similar to remote procedure calls. It is implemented as a defined pair of two messages. One message for the request and one for the reply. The client libraries also represent this functionality as a remote procedure call.
- **Actions**
  For asynchronous processing ROS uses actions. An action is a set of topics respectively messages which make it possible to send processing information to another node and wait for the result. The other node processes the requested data and gives a status feedback during processing. When the operation is finished the node publishes the result.

- **Transforms**
  For the representation of a geometric binding between links and joints ROS uses the concept of transforms. To do so the whole robot is modeled as a structure of joints and links which have defined transforms between each other. The transform toolset enables the calculation of advanced transformations like the pose of an endeffector link in the root link.

Example

A laser scanner is connected to a ROS computer with CAN. There is a node which gathers all the data sent by the scanner and does the preprocessing. This node then publishes the filtered and processed data to a topic called `scan`. Now all nodes which need the laser's sensor information can subscribe to this topic and receive the preprocessed data with timestamp and other metadata like the geometric link. This could for example be a mapping or localization node. To do so these nodes connect to the master and request the connection information for the topic. With this information they can connect directly.

On the other hand ROS provides developers and users with a vast number of tools. Two examples are the catkin packaging and building environment and RViz which is a customize visualization tool. RViz is able to view the robot and it's sensed environment as well as other information like camera images. It is basically the user interface for visualization of all information about the robot and it's current state.

3.2 MoveIt!

MoveIt! [21] [22] is a ROS framework for motion planning and object manipulation. MoveIt was developed for the arms of the PR2 robot. It is used to plan paths trough a collision domain for robot arms. A collision domain is a virtual projection of the environment and contains all obstacles. Figure 3.1 show the internal architecture of MoveIt!.

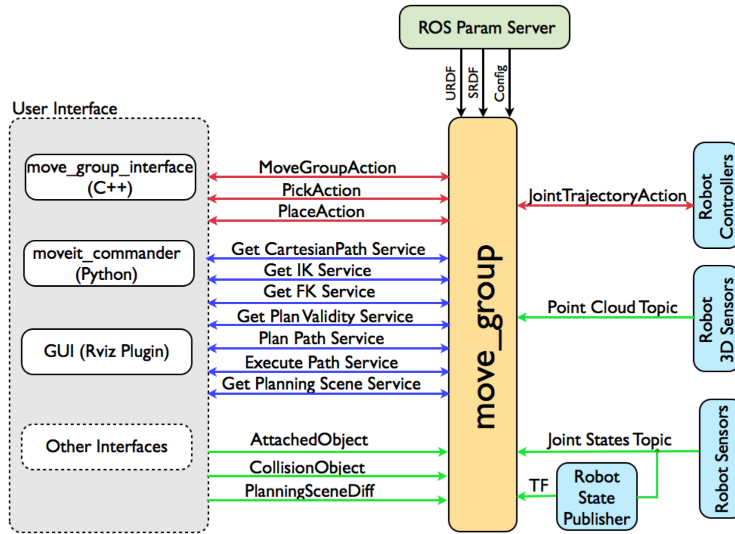

Figure 3.1: Structure of MoveIt! [21]

A MoveGroup is the central element of MoveIt. This module needs the following other modules to be able to operate.

- **Configuration**
  The MoveIt! configuration handles the geometric constraints of the robot. It comprises the robots URDF (Unified Robot Description Format) and SRDF (Semantic Robot Description Format) models as well as a general MoveIt configuration.

- **Robot Controllers**
  The robot controllers handle the motion commands MoveIt! generates for a real or simulated robot. On a real robot it converts joint trajectories to hardware commands. To perform smooth movements these trajectories consist of joint values as well as their speeds and accelerations.

- **Environment**
  The robot's 3D sensor data is represented in Octomaps [23], an octree-based probabilistic representation, and is used as the input for the different planning algorithms.

- **Joint States and Transforms**
  To plan new movements MoveIt has to have knowledge about the robots joint states as well as the transforms between the robot's links.

- **User Interface**
  There are several ways to control a MoveGroup. To control it from code there are interfaces for C++ as well as for Python. RViz also has an plugin to control a MoveGroup from an graphical interface.

With this information the MoveGroup uses a generic interface to various motion planners. Some examples are the Open Motion Planning Library (OMPL) [24] or the Search-based Planning Library (SBPL). MoveIt also uses path smoothers and additional constraints for the arm movements. An example for a constraint would be that a glass of water should always held upright so nothing is spilled.

Figure 3.2 shows the robot in the MoveIt! visualization. The blue elements are the powerball joints and the gray ones are the links between them. The orange base is the Forbot rover. On top of the arm there is the Schunk gripper.

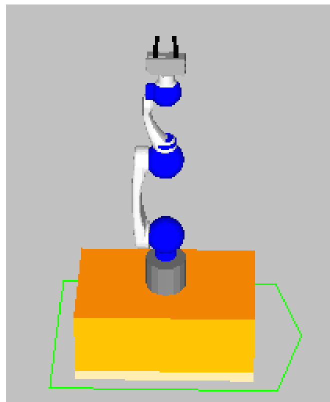

Figure 3.2: The Schunk arm in MoveIt! on the robot(orange) with the joints(blue), the links(gray) and the gripper on the top

3.3 MoveBase

MoveBase [25][26] is a 2D navigation stack for robots in ROS. MoveBase consists of multiple modules. Figure 3.3 shows the modules and how they work together. The key components are the following modules.

- **Costmaps**
  Costmaps are gridmaps which are used to calculate the path. Obstacles and the area around these obstacles are inflated with an expensive area to move in. Free space is rated with cheap values. The global costmap is built with the information provided by the map. All walls and obstacles which are part of the map are inflated with a defined radius. The closer an obstacle is the more expensive is the area. The

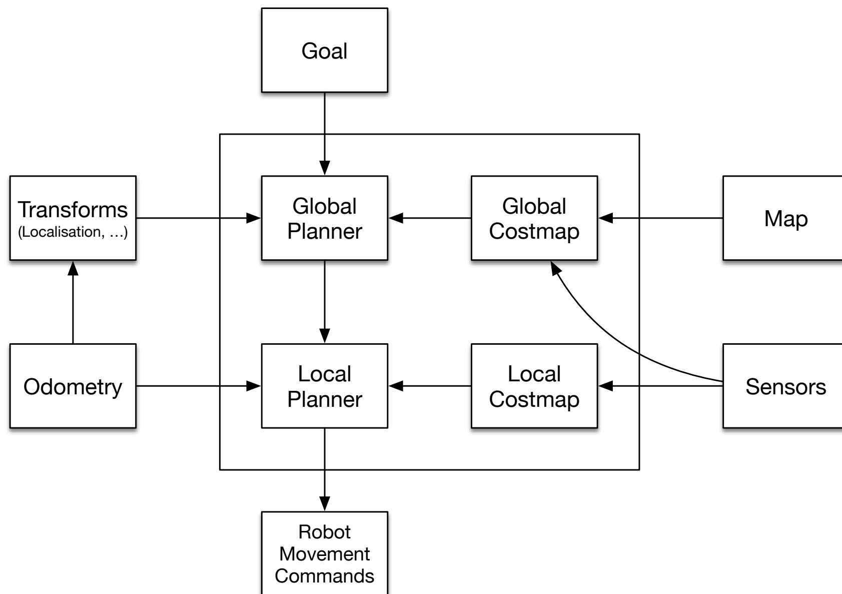

Figure 3.3: Structure of MoveBase

local costmap takes the current sensor information to create a dynamic costmap. This works for example with the measurements a laser range finder provides. All the obstacles reported by the sensor are inflated in the same way as in the global costmap. The local costmap has a very limited field of view. A human which is walking through the environment is visible in the local costmap but not in the global costmap. The size of the costmaps, their inflation radii and more parameters can be configured.

- **Planners**
  The planners use the costmaps and the localization information to create plans for the movement through the environment. The global planner calculates a path from the current position to the defined goal position using the global costmap. This path is then processed by the local planner. It works with an navigation function based on A* [27]. The local planner follows the path and handles the local obstacle avoidance. To do so it uses implementations of the trajectory rollout [28] and dynamic window approaches [29]. So basically if the laser scanner senses an obstacle in the local area of the robot the global plan is not changing. The local planner handles this obstacle and moves around it if possible. It is directly connected to the robot controller which executes the velocity commands. The executed path is a weighted joining of the global plan and a direct movement to the goal. These weights can be configured in the parametrization.

- **Recovery Behavior**

If the robot gets stuck the system has a defined recovery behavior. This consists of several steps like a conservative reset, a clearing rotation to reorient again or a aggressive reset.

3.4 STRIPS

The STRIPS (Stanford Research Institute Problem Solver) Algorithm [6] [30] is a problem solver for abstract defined worlds. These worlds are defined a state which is a list of prepositions and possible actions. Each problem is defined by a initial state S and a goal state G. To reach the goal the planner can apply a set of operations $O = \{O_1...O_n\}$ on the state. Each operation consists of four properties:

1. Preconditions which must be fulfilled P_{pos}
2. Preconditions which must not be fulfilled P_{neg}
3. State elements which are removed from the current state C_-
4. State elements which are added to the current state C_+

If an operation is applied to the current state, first the preconditions have to be part of the state.

$$P_{neg} \nsubseteq S_i \wedge P_{pos} \subseteq S_i \tag{3.1}$$

If this validation is positive the operation can be applied to the current state.

$$S_n = S_{n-1} + C_+ - C_- \tag{3.2}$$

This new state represents a new node in the tree of possible states. The transitions between these nodes are applicable operations. The application of these operations is repeated until the state fulfills the goal.

$$G \subseteq S_i \tag{3.3}$$

There are some drawbacks regarding this problem solver.

- The state search algorithms are incomplete
  For example it is not possible to find a solution for the simple problem of interchanging the values of two variables.

- It does not always find the best solution
  There can be a vast number of correct plans and it is not guaranteed that the plan is optimal. It is possible and likely that the found plan is not the shortest one.

4 Concept - System Overview

This chapter gives an overview of the design principles we use for combined task and motion planning. It introduces in detail the abstract and geometric planning process.

4.1 Modularization

To handle the planning requests the system is split up into three different modules.

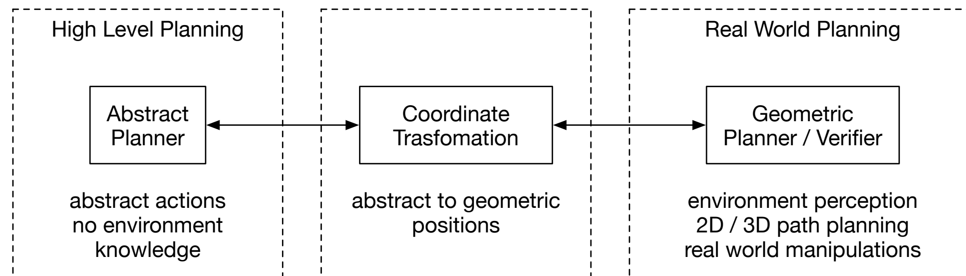

Figure 4.1: Modules of the planning process

The first module is the abstract high-level planner. This module handles the abstract logic-based planning process without detailed geometric information. The real world planner verifies that the abstract actions selected by the high-level planner is executables using a simulation of the real world. These two modules are connected by a symbol grounding module. This module handles the transformation of abstract positions to real world coordinates. The following sections describe these three modules in more details.

4.2 Abstract Domain Definition

The abstract planning domain is modeled using order sorted first order logic [31]. Figure 4.2 shows an example real world environment. On the left side we see the *Heap* area which stores all blocks which are not built into the tower yet. On the right side we see the *Tower* area which stores the tower and the *Robot* which is positioned next to the tower. The tower is already built up with three blocks a, b and c. In the *Heap* area there two more blocks (d and e) which are available for the robot to grasp if it is positioned near them.

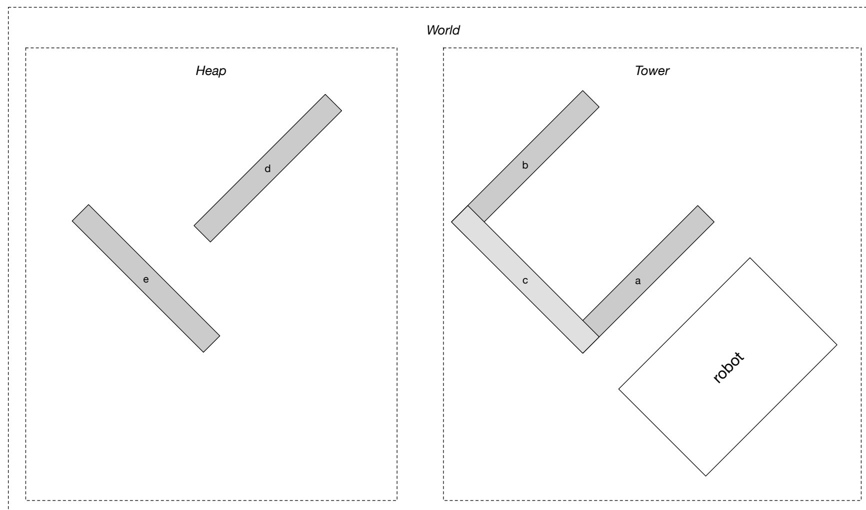

Figure 4.2: An example environment

4.2.1 Sorts

All terms in this domain (e.g. objects, positions) must have a defined sort. The sorts a used to bind function parameters to a defined type. The following sorts are defined:

- Integer:
  Integers are numbers in the space of $\mathbb{N}_0$. We assume a theory for Presburger arithmetic is part of the domain description.

- Objects:
  This sort defines the set of available objects in the environment. These objects are robots, towers, heaps and blocks.

- Areas:
  The environment might be partioned in several areas.

- Block positions:
  The position of a block can be defined in multiple ways. It is possible to place a block on the heap, in the robot's gripper or in the tower.

- Robot positions:
  The robot is able to move in the environment. It's position is defined relative to another object in the world.

- General positions:
  General positions are a super-sort of the available areas, block positions and robot positions.

- Directions:
  Directions are the orientations of a block in a tower.

4.2.2 Constants

The world is the definition of the whole environment.

$$W = World \tag{4.1}$$

This world is separated in areas. They are of the sort *Areas*. One area is the space around the tower and the other one is defined as the heap of available blocks. This sort is a set of constants.

$$A = \{Heap, Tower\} \tag{4.2}$$

In these areas there are several different objects. These objects are sets of robots R, towers T, heaps H and blocks B.

$$R = \{r_1\} \tag{4.3}$$
$$T = \{t_1\} \tag{4.4}$$
$$H = \{h_1\} \tag{4.5}$$
$$B = \{b_1, b_2, \ldots, b_n\} \tag{4.6}$$

When blocks are put on the tower they have be aligned in the same direction on each level. Figure 4.3 shows the possible directions. On the left side it shows a north-south direction and on the right side an east-west alignment. This results in the following constant definition.

$$D = \{North, East\} \tag{4.7}$$

The naming $North$ and $East$ is only used for the better understanding and is an abstract definition which does not correspond to the cardinal directions.

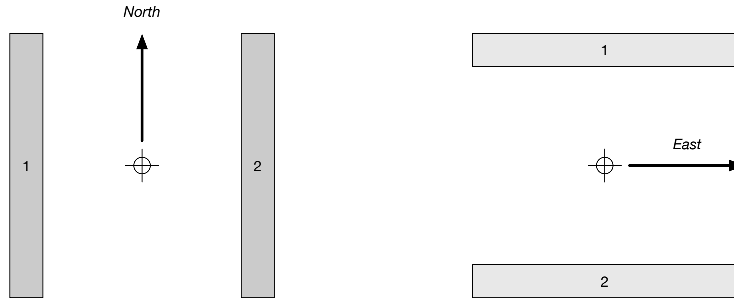

Figure 4.3: The direction of a level in the tower

4.2.3 Predicates

The fact that object O is located at position P is expressed by the predicate $pos(O, P)$ where O is of sort objects and P is of sort general positions. An object can only be on one position at a time.

$$\forall o \exists ! p \, [pos(o, p)] \tag{4.8}$$

Example

A possible position for the heap would be:

$$pos(h_1, Heap) \tag{4.9}$$

This would define that the heap h_1 is in the area $Heap$. In the example from Figure 4.2 the h_1 would represent the new introduced heap object in the $Heap$ area on the left side.

4.2.4 Functions

All positions are defined as functions and are reificated for their usage in the pos predicate. There are several different types of position functions:

Manipulation positions are the possible locations where the robot is able to execute a manipulation operation. They are from type robot position and are defined as a function of the object O which will be manipulated and an consecutive integer n as identifier for the multiple positions around this object.

$$mpos(O, n) \tag{4.10}$$

Example

A position for the robot in the example from Figure 4.2 would be:

$$pos\,(r_1, mpos\,(t_1, 1)) \tag{4.11}$$

This would mean that the robot r_1 is at a manipulation position at tower t_1 with the identifier 1.

A block can have different position types. They have all in common that they are from the sort block position. The first position of a block will most likely be a position on the heap h.

$$hpos(h) \tag{4.12}$$

If the robot grasps a block it's position changes to an grasp position defined by the robot r.

$$rpos(r) \tag{4.13}$$

If the robot puts a block on the tower the block's new position is defined by the tower position function. This function is defined by the tower t, the level l, the level's direction d and a consecutive integer n as position on this level.

$$tpos\,(t, l, d, n) \tag{4.14}$$

The number of the blocks on each level is defined by the type of tower which should be built. As shown in Figure 4.4 n would be in $\{1, 2\}$ for shoring towers and $\{1, 2, 3\}$ for Jenga towers.

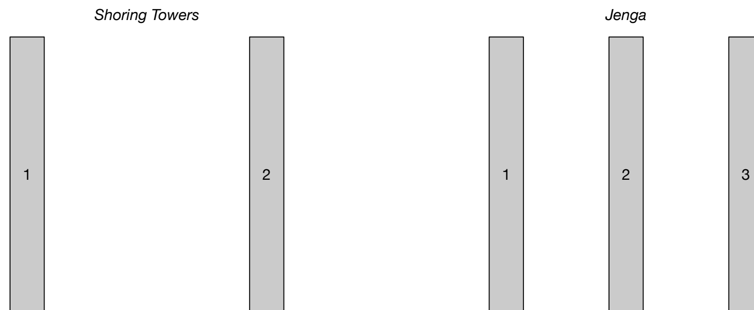

Figure 4.4: Block positions in a shoring tower and in Jenga towers

Example

If a block is put on the tower it could have the following position.

$$pos(a, tpos(t_1, 1, North, 1)) \tag{4.15}$$

This represents the fact that the block a is placed on tower t_1 on the first level with an north-south direction.

To build a valid block the following rules apply:

- All blocks on the same level have to have the same direction and different position identifiers.

$$\begin{aligned}\forall t, o_1, o_2, l, d_1, d_2, p_1, p_2 :\ & pos(o_1, tpos(t, l, d_1, p_1)) \wedge pos(o_2, tpos(t, l, d_2, p_2)) \rightarrow \\ & \rightarrow p_1 \neq p_2 \wedge d_1 = d_2\end{aligned} \tag{4.16}$$

- A block can only be put on a level if the level below is finished and has no empty spots.

$$\forall t, o, l, d, p : pos(o, tpos(t, l, d, p)) \rightarrow \forall p', l'.\ \exists o', d' : l' < l \wedge pos(t, l', d', p') \tag{4.17}$$

- The direction of each level has to be different than the direction of the level below.

$$\begin{gathered}\forall t, o, l, d, p : pos(o, tpos(t, l, d, p)) \rightarrow \\ \rightarrow \forall o', l', d', p' : l = l' + 1\ \wedge\ pos(o', tpos(t, l', d', p')) \rightarrow d' \neq d\end{gathered} \tag{4.18}$$

4.2.5 Actions

Move the robot

The operation to move a robot r form the origin position o to it's goal g is defined as follows:

$$moveTo(r, o, g) \tag{4.19}$$

The sort of r is objects and o and g are both robot positions.

Preconditions:

$$\neg pos(r, g) \wedge o \neq g \tag{4.20}$$

These preconditions imply that the robot's current position has to be different than the goal position.

Effect:

$$pos(r, g) \wedge \neg pos(r, o) \tag{4.21}$$

This operation results in a new position for the robot which is different than the old position.

Grasp a block

The operation of grasping a block b by a robot r is defined as follows:

$$grasp(r, b) \tag{4.22}$$

r and b are both from the sort objects.

Preconditions:
To grasp a block the robot r has to be on a position at the block b and the gripper has to be empty.

$$\exists y : pos(r, mpos(b, y)) \wedge \neg\exists\ b' : pos(b', rpos(r)) \tag{4.23}$$

Effect:

$$pos(b, rpos(r)) \wedge \neg\exists x : pos(b, x) \wedge x \neq rpos(r) \tag{4.24}$$

Put a block

The operation of a robot r putting a block b on the position p is defined as follows:

$$put(r, b, p) \tag{4.25}$$

With r and b from sort objects and p from the sort block position.

Preconditions:
To put a block b on the position p the block has to be held by the robot, the robot has to be at the position and the desired position has to be free.

$$\begin{aligned} &\exists t', l', d', p' : p = tpos(t', l', d', p') \wedge pos(b, rpos(r)) \wedge \\ &\exists n : pos(r, mpos(t', n)) \wedge \neg\exists b' : pos(b', p) \end{aligned} \tag{4.26}$$

Effect:

$$pos(b, p) \wedge \neg pos(b, rpos(r)) \tag{4.27}$$

4.3 Symbol Grounding

Since the abstract planner has no geometric knowledge but the geometric planner needs this information there is a symbol grounding module between them. This module takes environment information and abstract information to calculate the real world coordinates and poses for the geometric planner.
Figure 4.5 shows an example environment. The environment has the global coordinate frame *world frame*. Moreover, the environment comprises the two areas heap and tower with its coordinate frames *heap frame* respectively *tower frame*.

Besides that, the environment contains the heap h1 and a tower t1 with its coordinate frames *h1 frame* respectively *t1 frame*. The blocks a, b and c are already part of the tower and so they are placed in the t_1 frame. The blocks d and e are part of the heap h_1 on the left side. The robot is positioned next to block a in the area around tower t_1.

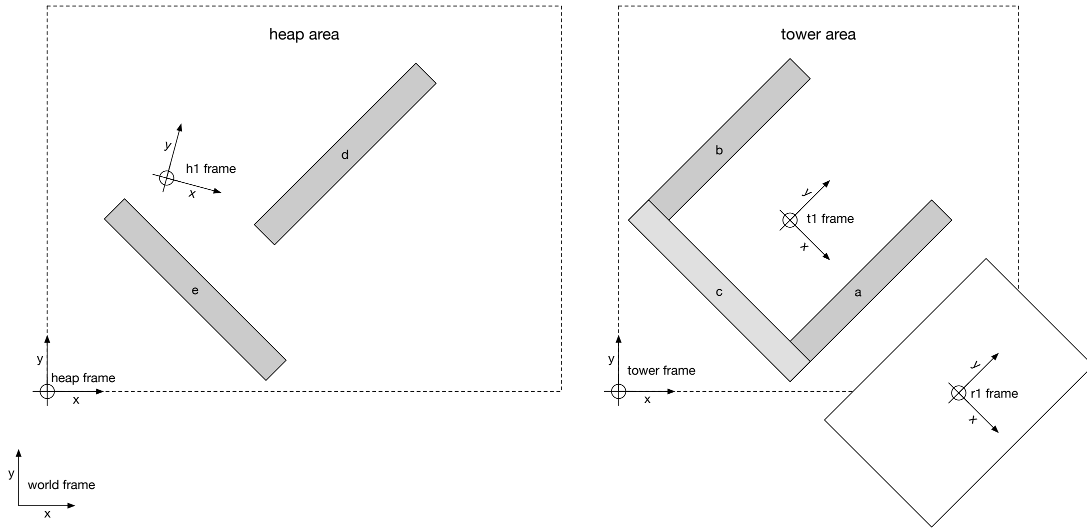

Figure 4.5: An example of a real environment with a heap on the left side and a tower and the robot on the right side

The abstract description for the example environment would be as follows:

$$\begin{aligned} &pos(Tower, World)\\ &pos(Heap, World)\\ &pos(t_1, Tower)\\ &pos(h_1, Heap)\\ &pos(a, tpos(t_1, 1, East, 1))\\ &pos(b, tpos(t_1, 1, East, 3))\\ &pos(c, tpos(t_1, 1, North, 3))\\ &pos(d, hpos(h_1))\\ &pos(e, hpos(h_1))\\ &pos(r_1, mpos(t_1, 1)) \end{aligned} \tag{4.28}$$

4.3.1 Real World Coordinates

The calculation of world coordinates for a frame is a concatenation of multiple transforms. Each element in the abstract world description has an equivalent transformation in the real world.
Figure 4.6 shows an example transformations (green arrows) for block's frame a. The block's frame ($a\ frame$) is part of the tower's frame ($t_1\ frame$) which is in the tower area ($tower\ frame$). The tower area is part of the world frame. The abstract description of this green path and their corresponding transformations are defined as follows.

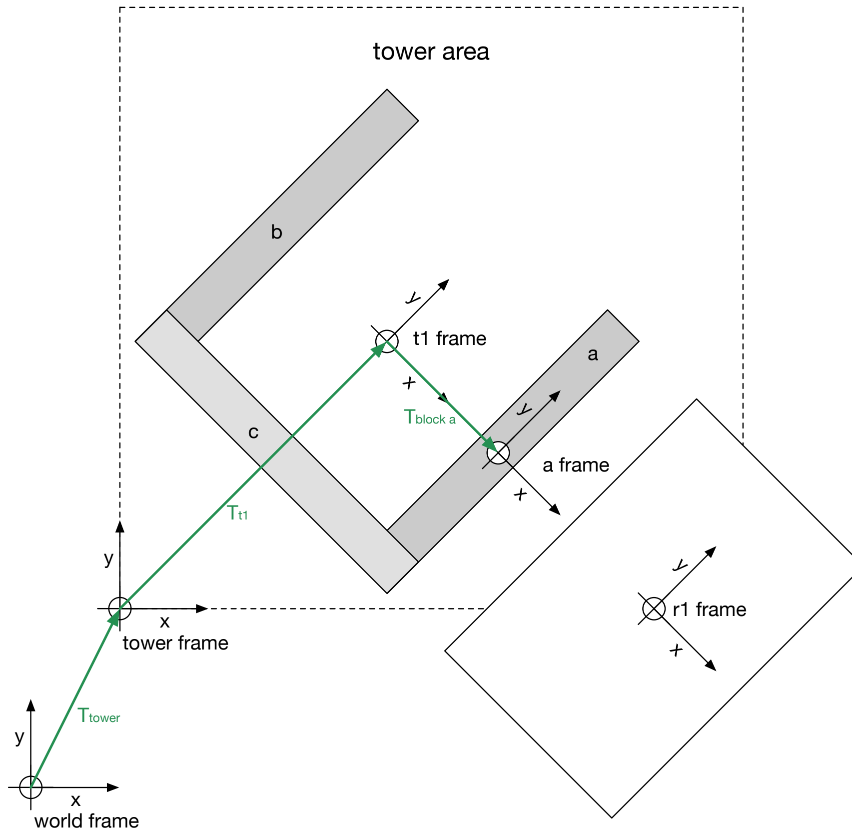

Figure 4.6: Transformations from the world frame to a block with the green arrows as the transform tree

Equation 4.29 is a lineup of the abstract definitions for all the abstract definitions for block a and it's real world equivalents.

$$\begin{aligned} pos(Tower, World)\ represents\ T_{tower} \\ pos(t_1, Tower)\ represents\ T_{t_1} \\ pos(a, tpos(t_1, 1, East, 1)\ represents\ T_{block_a} \end{aligned} \tag{4.29}$$

The concatenated transformations result in the following equation which projects block coordinates into world coordinates.

$$\begin{bmatrix} x_{world} \\ y_{world} \\ z_{world} \\ 1 \end{bmatrix} = T_{tower} \cdot T_{t_1} \cdot T_{block_a} \cdot \begin{bmatrix} x_{block_a} \\ y_{block_a} \\ z_{block_a} \\ 1 \end{bmatrix} \tag{4.30}$$

T_{tower} is defined as a function of the position $[x_{tower}\ y_{tower}]^T$ and angle φ_{tower} of the area in the real world.

$$T_{tower} = f\left(\begin{bmatrix} x_{tower} \\ y_{tower} \end{bmatrix}, \varphi_{tower}\right) \tag{4.31}$$

Very similar to that T_{t_1} is defined as a function of the tower's position $[x_{t_1}\ y_{t_1}]^T$ and angle φ_{t_1} in the tower frame.

$$T_{t_1} = f\left(\begin{bmatrix} x_{t_1} \\ y_{t_1} \end{bmatrix}, \varphi_{t_1}\right) \tag{4.32}$$

The position of the block's link in the tower link has dependencies to the real world (coordinates and block sizes) as well as to the abstract definition.

$$T_{block_a} = f\left(\begin{bmatrix} x_{block_a} \\ y_{block_a} \\ z_{block_a} \end{bmatrix}, \varphi_{block_a}\right) \tag{4.33}$$

With a block defined by it's size $[b_{width}\ b_{length}\ b_{height}]^T$ and the identifier of the block's position N this results in the following functions:

$$\begin{bmatrix} x_{block_a} \\ y_{block_a} \end{bmatrix} = f(Direction, N, b_{width}, b_{length}) \tag{4.34}$$

$$z_{block_a} = f(Level, b_{height}) \tag{4.35}$$

$$\varphi_{block_a} = f(Direction) \tag{4.36}$$

Each of these transformations is defined by a rotation and a translation. Equation 4.37 defines the one of these transformations.

$$\begin{bmatrix} x_{parent} \\ y_{parent} \\ z_{parent} \\ 1 \end{bmatrix} = T_{child} \cdot R_{child} \cdot \begin{bmatrix} x_{child} \\ y_{child} \\ z_{child} \\ 1 \end{bmatrix} \tag{4.37}$$

With matrices for the rotation and translation this results in the following equations.

$$\begin{bmatrix} x_{parent} \\ y_{parent} \\ z_{parent} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & x_{child} \\ 0 & 1 & 0 & y_{child} \\ 0 & 0 & 1 & z_{child} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} cos\varphi_{child} & -sin\varphi_{child} & 0 & 0 \\ sin\varphi_{child} & cos\varphi_{child} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{child} \\ y_{child} \\ z_{child} \\ 1 \end{bmatrix} \tag{4.38}$$

Example

To calculate the real world coordinates of block a from the example in Figure 4.6 the calculation would be as follows:
In the abstract description the block is defined by the following statements:

$$pos(Tower, World)$$
$$pos(t_1, Tower)$$
$$pos(a, tpos(t_1, 1, East, 1))$$

The transform to the block's frame is defined by the concatenation of transforms from the world to the tower area to the tower t_1 and it's relative position in this frame.

$$\begin{bmatrix} x_{world} \\ y_{world} \\ z_{world} \\ 1 \end{bmatrix} = T_{tower} \cdot T_{t_1} \cdot T_{block_a} \cdot \begin{bmatrix} x_{block_a} \\ y_{block_a} \\ z_{block_a} \\ 1 \end{bmatrix}$$

The transformation T_{tower} from the world frame to the origin of the $Tower$ area is defined by it's translation $\begin{bmatrix} 10 \\ 2 \end{bmatrix}$ and no rotation.

$$T_{tower} = \begin{bmatrix} 1 & 0 & 0 & x_{tower} \\ 0 & 1 & 0 & y_{tower} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 10 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The t_1 frame is then defined by the transformation relative to the $Tower$ area. Let's assume this transformation is a translation $\begin{bmatrix} 3 \\ 3 \end{bmatrix}$ and a rotation with $\frac{-\pi}{4}$.

$$T_{t_1} = \begin{bmatrix} 1 & 0 & 0 & x_{t_1} \\ 0 & 1 & 0 & y_{t_1} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} cos\varphi & -sin\varphi_{t_1} & 0 & 0 \\ sin\varphi & cos\varphi_{t_1} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} =$$

$$= \begin{bmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} cos\frac{-\pi}{4} & -sin\frac{-\pi}{4} & 0 & 0 \\ sin\frac{-\pi}{4} & cos\frac{-\pi}{4} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} =$$

$$= \begin{bmatrix} cos\frac{-\pi}{4} & -sin\frac{-\pi}{4} & 0 & 3 \\ sin\frac{-\pi}{4} & cos\frac{-\pi}{4} & 0 & 3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The transformation to the block frame T_{block_a} is then a function of the block's size and it's abstract definition. The abstract definition is:

$$tpos(TowerId, Level, Direction, N) = tpos(t_1, 1, East, 1)$$

The frame's translation and rotation in the t_1 frame is defined with

$$\begin{bmatrix} x_{block_a} \\ y_{block_a} \\ z_{block_a} \end{bmatrix} = \begin{bmatrix} f(Direction, N, b_{width}, b_{length}) \\ f(Direction, N, b_{width}, b_{length}) \\ f(Level, b_{height}) \end{bmatrix} = \begin{bmatrix} \frac{b_{length}-b_{width}}{2} \\ 0 \\ 0 \end{bmatrix} =$$

$$= \begin{bmatrix} \frac{0.60-0.10}{2} \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.25 \\ 0 \\ 0 \end{bmatrix}$$

$$\varphi_{block_a} = f(Direction) = 0$$

This results in the following transformation matrix

$$T_{block_a} = \begin{bmatrix} 1 & 0 & 0 & 0.25 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} cos\,0 & -sin\,0 & 0 & 0 \\ sin\,0 & cos\,0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} =$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0.25 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformed in real world coordinates the block's center position would results in:

$$p_{rw} = T_{tower} \cdot T_{t_1} \cdot T_{block_a} \cdot \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 13.1768 \\ 0 & 1 & 0 & 4.8232 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 13.1768 \\ 4.8232 \\ 0 \\ 1 \end{bmatrix}$$

4.3.2 Robots Base and End-effector Pose

To calculate the robot's base pose the same methods as above are applied (Equation 4.38). The difference is that the position in the tower link is a different. On position is in parallel to the block as depicted in Figure 4.5 ($pos(r, mpos(t_1, 1))$). The green arrows represent the transform tree and the red arrow is the resulting transform between the robot r_1 and the block a.

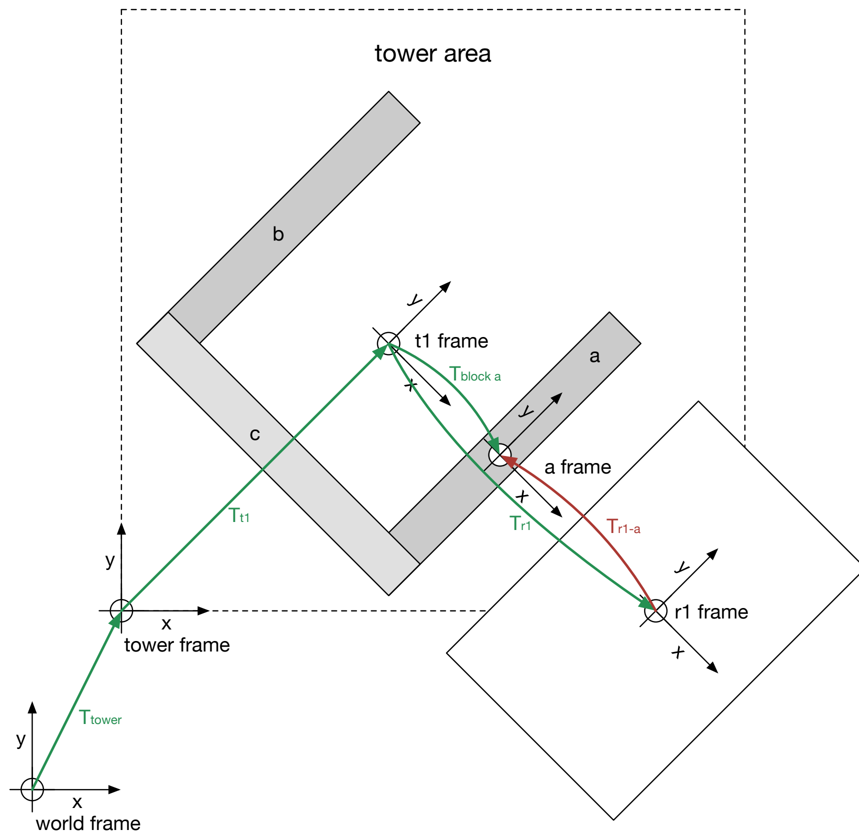

Figure 4.7: Transformations for the robot and it's end-effector pose with the green arrows as the transform tree and the red arrow as the resulting relative transform between robot and block

To calculate the transformations to the robot's frame r_1 a concatenation of transformations have to be applied. This is similar to the calculation of a block's transformation in the tower (see last section).

$$\begin{aligned} pos(Tower, World) \ represents \ T_{tower} \\ pos(t_1, Tower) \ represents \ T_{t_1} \\ pos(r_1, mpos(t_1, 1)) \ represents \ T_{r_1} \end{aligned} \tag{4.39}$$

$$\begin{bmatrix} x_{world} \\ y_{world} \\ z_{world} \\ 1 \end{bmatrix} = T_{tower} \cdot T_{t_1} \cdot T_{r_1} \cdot \begin{bmatrix} x_{r_1} \\ y_{r_1} \\ z_{r_1} \\ 1 \end{bmatrix} \tag{4.40}$$

The robot's transform T_{r_1} in the tower frame is defined by the following functions.

$$\begin{bmatrix} x_{r_1} \\ y_{r_1} \end{bmatrix} = f(N, b_{width}, b_{length}) \tag{4.41}$$

$$\varphi_{r_1} = f(N) \tag{4.42}$$

The calculation of the correct grasping and placing position is slightly more difficult. The goal pose for the arm is calculated with a position which is a simple translation to a point in 3D and an orientation. The translation is a relative value to the robot's link so the block's link in the tower's link t_1 has to be remapped to a position in the robot's link r_1. This remapping is realized with the transformation $T_{r_1 a}$ (red arrow).

Since the block's position is defined in the t_1 frame we first need to create the translation matrix from t_1 frame to the r_1 frame.

The relation between coordinates in the block's frame and the tower's t_1 frame is defined as follows.

$$\begin{bmatrix} x_{t_1} \\ y_{t_1} \\ z_{t_1} \\ 1 \end{bmatrix} = T_{block_a} \cdot \begin{bmatrix} x_{block_a} \\ y_{block_a} \\ z_{block_a} \\ 1 \end{bmatrix} \tag{4.43}$$

A similar relation is valid for the robot's frame r_1.

$$\begin{bmatrix} x_{t_1} \\ y_{t_1} \\ z_{t_1} \\ 1 \end{bmatrix} = T_{r_1} \cdot \begin{bmatrix} x_{r_1} \\ y_{r_1} \\ z_{r_1} \\ 1 \end{bmatrix} \tag{4.44}$$

These two equations both calculate coordinates in the t_1 frame so they can be equated.

$$T_{r_1} \cdot \begin{bmatrix} x_{r_1} \\ y_{r_1} \\ z_{r_1} \\ 1 \end{bmatrix} = T_{block_a} \cdot \begin{bmatrix} x_{block_a} \\ y_{block_a} \\ z_{block_a} \\ 1 \end{bmatrix} \tag{4.45}$$

To get the block's coordinates in the robot's frame the inverse matrix of the robot's transformation has to be applied.

$$\begin{bmatrix} x_{r_1} \\ y_{r_1} \\ z_{r_1} \\ 1 \end{bmatrix} = T_{block_a} \cdot T_{r_1}^{-1} \cdot \begin{bmatrix} x_{block_a} \\ y_{block_a} \\ z_{block_a} \\ 1 \end{bmatrix} \tag{4.46}$$

This transformation is limited by the robot arm's workspace. The grasping position on the block define the coordinates in the $block_a$ frame. In the implementation for this thesis these coordinates are set to the middle of the block which is represented by the following equation.

$$\begin{bmatrix} x_{grasp} \\ y_{grasp} \\ z_{grasp} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ f(Level) \end{bmatrix} \tag{4.47}$$

Other values would increase the robot's flexibility but would also change the robot's base positions and so drastically increase the size of the state graph. The implementation of this is a future task on top of this thesis.

The orientation of the arm's end effector is defined in quaternions. Quaternions are a way to describe rotations in the 3 dimensional space with a four element vector:

$$\mathbf{q} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} \tag{4.48}$$

This vector represents a complex value with three different complex parts.

$$\mathbf{q} = q_0 + q_1 \cdot \mathbf{i} + q_2 \cdot \mathbf{j} + q_3 \cdot \mathbf{k} \tag{4.49}$$

It is possible to add, multiply, divide (inverse) quaternions. The only thing quaternions don't have is a commutative multiplication. If several quaternions are computed together to a final quaternion the resulting quaternion represents a unit vector of a rotation axis and the angle of the rotation around this axis.
To grasp a block the gripper has to point straight downwards. This is handled with a rotation by π around the vertical axis (yaw). The pitch and roll angle are both zero. This results in the following quaternion.

$$\mathbf{q} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{4.50}$$

Example

To calculate the real world coordinates of the robot the same steps as in the previous example have to be executed.

$$\begin{bmatrix} x_{world} \\ y_{world} \\ z_{world} \\ 1 \end{bmatrix} = T_{tower} \cdot T_{t_1} \cdot T_{r_1} \cdot \begin{bmatrix} x_{block_a} \\ y_{block_a} \\ z_{block_a} \\ 1 \end{bmatrix}$$

The transformation depends on different variables.

$$\begin{bmatrix} x_{r_1} \\ y_{r_1} \end{bmatrix} = \begin{bmatrix} f(N, b_{width}, b_{length}) \\ f(N, b_{width}, b_{length}) \end{bmatrix} = \begin{bmatrix} 0.9 \\ 0 \end{bmatrix}$$

$$\varphi_{block_a} = f(Direction) = \frac{\pi}{2}$$

This results in the following transformation matrix

$$T_{r1} = \begin{bmatrix} cos\ 0 & -sin\ 0 & 0 & 0.9 \\ sin\ 0 & cos\ 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The end-effector pose of the robot's arm is then calculated as a relative value to the robot's link r. The translation is defined as follows:

$$\begin{bmatrix} x_{r_1} \\ y_{r_1} \\ z_{r_1} \\ 1 \end{bmatrix} = T_{block_a} \cdot T_{r_1}^{-1} \cdot \begin{bmatrix} x_{block_a} \\ y_{block_a} \\ z_{block_a} \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_{r_1} \\ y_{r_1} \\ z_{r_1} \\ 1 \end{bmatrix} = \begin{bmatrix} cos\ 0 & -sin\ 0 & 0 & 0.25 \\ sin\ 0 & cos\ 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} cos\ 0 & -sin\ 0 & 0 & 0.9 \\ sin\ 0 & cos\ 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} x_{block_a} \\ y_{block_a} \\ z_{block_a} \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_{r_1} \\ y_{r_1} \\ z_{r_1} \\ 1 \end{bmatrix} = T_{block_a} \cdot T_{r_1}^{-1} \cdot \begin{bmatrix} x_{block_a} \\ y_{block_a} \\ z_{block_a} \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_{r_1} \\ y_{r_1} \\ z_{r_1} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -0.65 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{block_a} \\ y_{block_a} \\ z_{block_a} \\ 1 \end{bmatrix}$$

The z-value for the end-effector (at the moment set to zero) is defined by the gripper's geometry and has to be set to a proper value for the block's level in the tower. As mentioned above the quaternion for the end-effector's orientation is set to $q = \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix}^{-1}$.

4.4 Geometric planning

The geometric planning module is responsible for verifying and executing abstract action in the real world. For each action it receives an environment state and the action. This environment state is then built into the real world environment the geometric planner gathers with it's sensors. Then it executes the action and returns the execution status of the action. The planning for the robot's base and the arm is handled by two different modules.

4.4.1 Robot Base Navigation

In order to place or grab a block the robot first has to navigate to a position where it is possible to execute this grabbing or placing action. To do so the robot has to sense and create or use a map of the environment. It then has to locate itself in this environment. These positions where the robot has to move to are dependent on how a block has to be grabbed or placed. The circles in Figure 4.8 show some possible positions for the robot's base from where it could place a block.

If the robot has to move from the heap area to the tower because it grabbed a new block it has to plan a path. The important thing in this task is that it has to find a valid path which does not touch obstacles and result in the correct pose to place the block. Figure 4.8 shows a possible path from a position where the robot grabbed the block d and goal position to place the block on the tower.

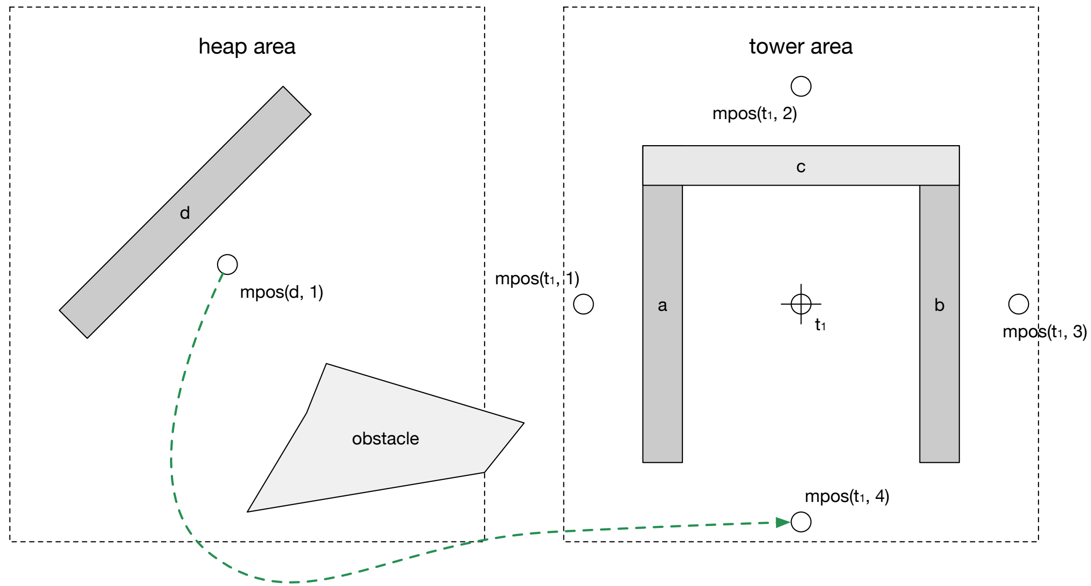

Figure 4.8: Possible path between two positions

The robot base navigation submodule uses the robots sensor information to locate the robot and find paths to the required goal positions. On the other hand it could be possible that some points are not valid. This could for example happen because there is an obstacle or that there is no free path to reach this point. Figure 4.9 for example shows a situation where some points are not reachable. This of course results in very restricted constraints for the possible tower variants which can be built although they are possible on the abstract level. This is the reason why the geometric planning is necessary.

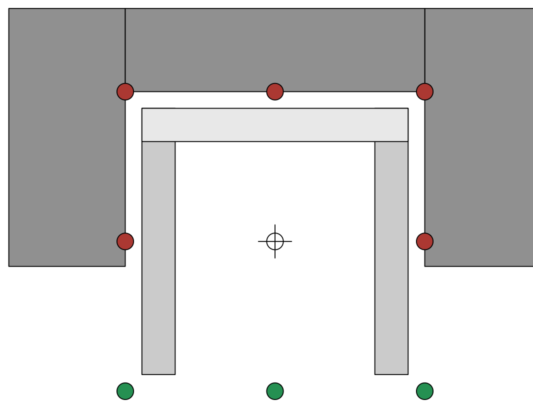

Figure 4.9: Possible Robot Positions in an environment

4.4.2 Robot Arm Movement - Pick and Place

When the robot is near a block it has to grab this block. To do so the arm has to move from it's current position to a pose where it can grab the block. The Robot Arm Movement submodule calculates and executes the path for the robots pick and place operations. The area the arm is able to operate in is limited by the arms workspace. Figure 4.10 shows the arms from the technical point of view.

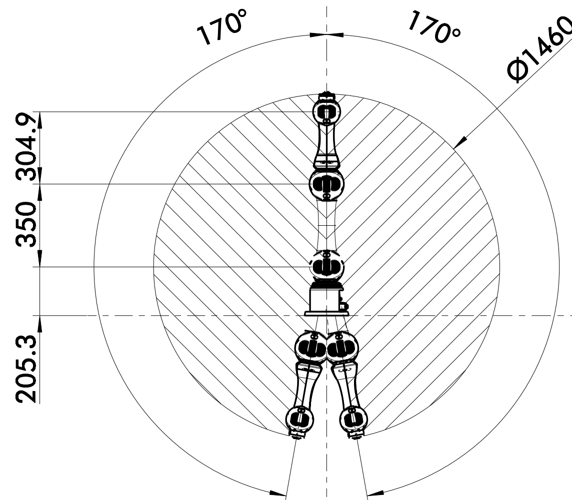

Figure 4.10: The robot arm's workspace[32](right side)

Additional to that the above workspace is limited by the robots base and obstacles in the environment. These elements are added to a collision domain which is used by the planner to find collision free movements. The goal of each planning request is a smooth movement from pose `A` to `B` without any collisions. To prevent a collision of the arm with other links of the arm due to the high degree of freedom the planning algorithm uses a 3D model of the arm.

To prevent collisions with any obstacles the planner uses a 3D representation of the robot's environment constructed from the robot's sensor data and it's knowledge about the environment. This knowledge is represented by the environment map and the known state of the block's positions. So for example the planner knows that there are already three block in the tower and is able to extend the collision domain with this information. From this representation it is possible to calculate the obstacles in the arm's workspace. With these obstacles the robot's movement path can be calculated and executed. If the module does not find a path or finds collisions with obstacles this planning request fails.

5 Implementation

Figure 5.1 shows the implemented planning system. On the left side of the figure is the abstract planner which is implemented in PROLOG and described in Section 5.2. The right side represents the geometric planner and verifier which is built in the ROS environment. Dependent on the action which should be verified the dispatcher passes the operation to MoveBase for 2D navigation or to MoveIt! for object manipulation. This is described in detail in Section 5.4.

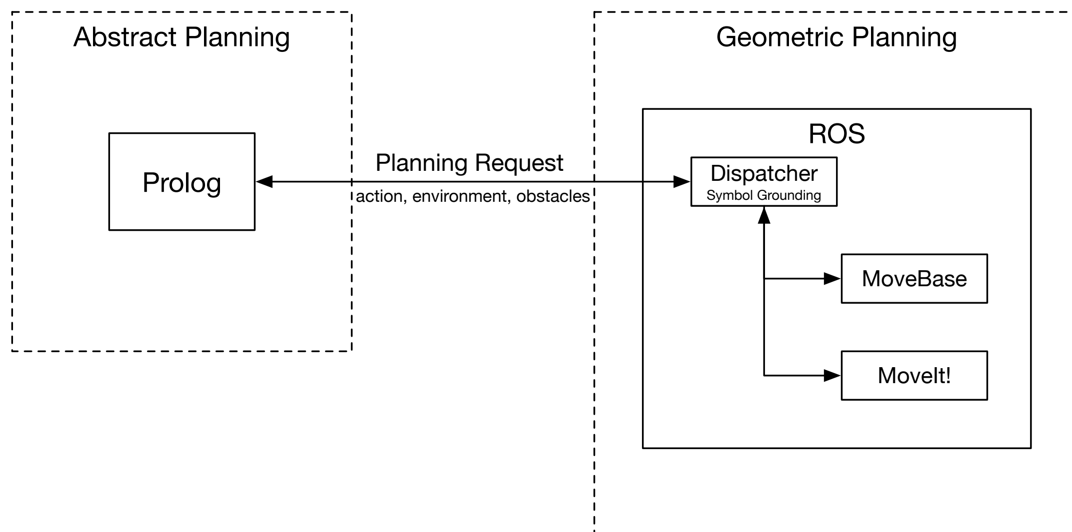

Figure 5.1: Overview of the implementation

5.1 Handling a planning request

As mentioned before several steps are needed to handle a planning request. Figure 5.2 shows an overview how these requests are handled. The gray numbers in the figure correspond to the items in the following detailed description.

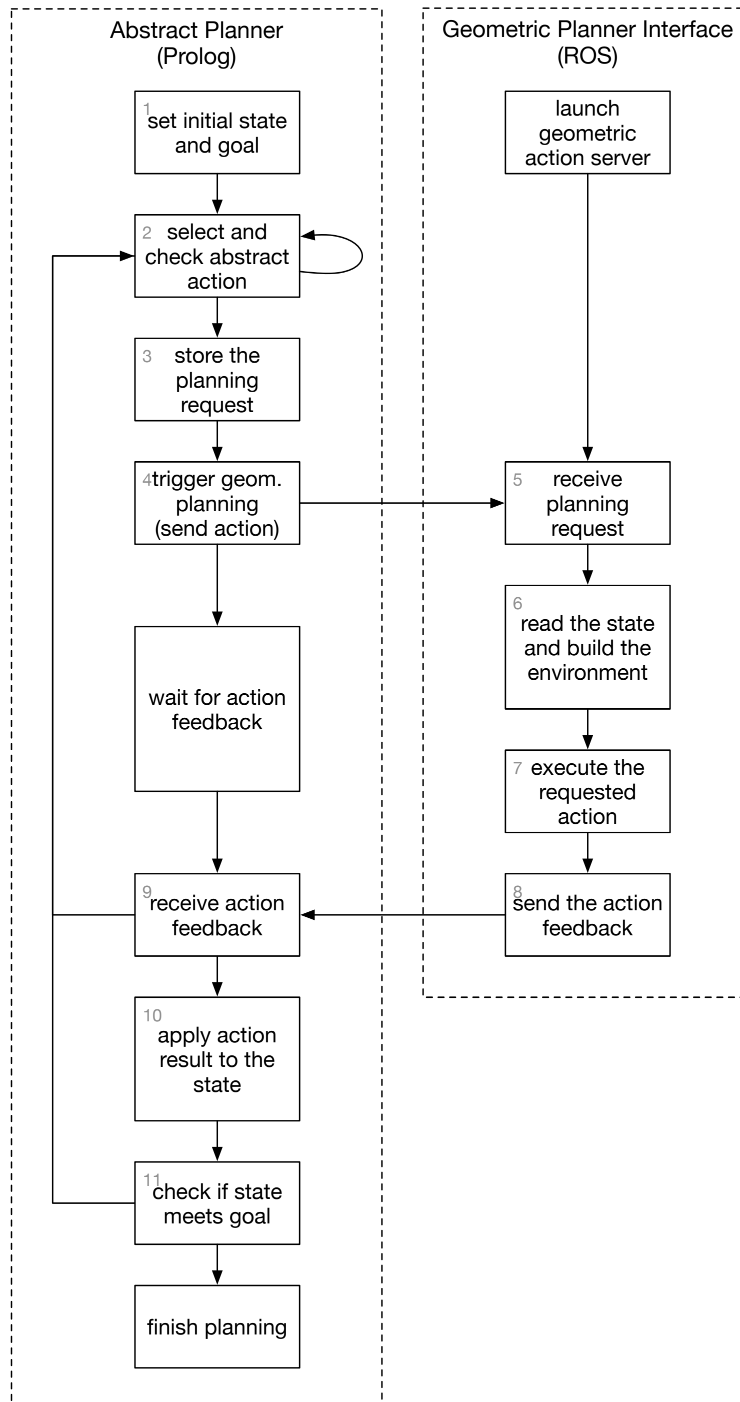

Figure 5.2: Sequence of a planning request

1. Set initial state and goal
   To start with the planning an initial state is defined. This state is an abstract representation of the starting conditions in the environment. On top of that a goal which should be reached is defined. This goal could for example be a tower with a height of two levels.

2. Select and check an abstract action
   The abstract planner selects an action from the pool of possible ones. It also applies various optimizations to accelerate the planning process. These optimizations are very performance relevant because verifying an action in the logic space is a matter of milliseconds. On the other hand does it take several seconds to verify an action in the geometric world. So each action which can be excluded from the geometric planning brings a boost in the planning time. If the selection and the optimizations were successful the planner switches to the next step. Otherwise it selects another action.

3. Store the planning request
   To execute a geometric request the planner needs knowledge about the state and the environment. These states represent the abstract position of the blocks and the robot etc. (see Section 5.2.1). The state is written as a string to a file.

4. Trigger the geometric planning
   To trigger the geometric planning and to switch to the ROS world the planner has to leave the PROLOG context. To do so it triggers a Python script which has a connection to ROS. This script launches a ROS action client and informs the geometric action planner that there is a new planning job.

5. Receive a planning request
   When the action server receives a planning request the geometric planning process is triggered.

6. Read the state and build the environment
   The geometric planner needs an environment to plan in. This environment is based on a map. This map can be based on a stored map or can be a environment sensed with the LIDAR scanner. On top of that the planner projects the abstract state into this world. It sets the robots position and builds up the part of the tower which is already finished.

7. Execute the requested action
   After building up the environment the planner plans and executes the action. This can either be a request for the 2D navigation or the object manipulation. The 2D navigation is used to move the robot from position A to position B without colliding with an obstacle. The 3D object manipulation is used to grab and place a block.

8. Send the action feedback

The result of the planning process is sent back to the client. This is a simple feedback that indicates a successful or failed planning request.

9. Receive action feedback
   The action client interprets this feedback and passes it to the PROLOG planner. This happens through the return value of the Python script.

10. Handle the planning result in the abstract world
    Back in the PROLOG context the planner has a feedback if this action is valid in the real world. If this is the case the action's effect can be applied to the current state. If the geometric planning feedback indicates that the action is not executable the planner drops this action and has to select a different one and try again with the same as the current state.

11. Check if the current state meets the goal
    The new state which was created by applying the actions effect has to be checked with the goal state. If the this state meets the goal requirements the planning process is finished. If not another new action has to be selected starting with the new state.

Example

Let's assume the robot is at a position in the area around the tower.

1. The abstract planner selects a new action and checks if this is a valid one. This will be for example to move the robot from the heap to the tower.
2. The current state and the action to test is stored to an output file.
3. PROLOG calls the Python script which then triggers an action in ROS.
4. The action is received in the ROS world and the action handler is called.
5. The handler builds the environment based on the state file.
6. The geometric planner triggers a planning request to MoveBase which tries to find a path for the arm to the requested goal. MoveBase returns that this action is executable.
7. This result is sent to the action client.
8. The action client passes to the PROLOG planner that this is a geometrically valid action.
9. The planner adds this action to it's plan and proceeds with the next action.

5.2 Abstract high level planning in PROLOG

The high level planning module is a STRIPS planner written in PROLOG (SWI PROLOG). It is a practical implementation with some simplifications of the concept of the last chapter.
The planner is based on the SIMSTRIPS planner from Tim Smith [33]. This planner has a defined initial state and a goal state which should be reached. To reach the goal state the planner tries available operations. If the planner applies an action to the current state it first has to verify if all preconditions are fulfilled. If this is the case the geometric planner is called with the current state and the abstract action. For this purpose a binding to the geometric planner and verifier (Section 5.4) was implemented. If the geometric verification was also successful the planner applies the action to the current state. The state is a list representation of the current valid logical predicates. The effect of actions is the adding and removing of some elements from this current state list. This working principle leads to a very large state graph with each possible state as a node and actions as transitions between these nodes. Since PROLOG uses a depth first search in the state tree the planner was extended with some optimizations (Section 5.2.5) to improve the planning performance. In addition to that the planner was also extended with a functionality to handle negative preconditions for action. These are preconditions which must not be fulfilled.

5.2.1 State representation

The state of the environment is represented as a list of propositions. This list uses helper predicates to follow the set semantic of STRIPS. The state is manipulated by the actions with add and remove operations. The following state propositions are defined:

Positions

In chapter subsection 4.2.4 the logical definition of positions was introduced. This definition is handled with *pos* predicates like the following ones.

$$pos(r_1, mpos(t_1, 1)) \tag{5.1}$$

$$pos(a, tpos(t_1, 1, North, 1)) \tag{5.2}$$

The first predicate means that the robot r_1 is currently at a position at the tower t_1 with the identifier 1. The second one means that the block a is at a position in the tower t_1 on level 1 with a defined orientation and identifier.

In this planner we use some simplifications. There is no identifier for the robot used because the planning process does not handle multiple robots. The same applies to the tower and the heap. There is only one tower and one heap. In the position statements the values `tower` and `heap` represent their single logical equivalents t_1 and h_1.

The position of an object is not handled with a global *pos* predicate which is valid for all objects. Instead we use a specific position definition for the robot and a separate definitions for the blocks.

Robots position

In the logical definition the robot's position is defined with $pos(r_1, mpos())$ statement. In Prolog the robot's position is defined with a `position`-statement. So for example $pos(r_1, mpos(t_1, 1))$ would be projected to `position(tower, n, 1)` with n as a simple placeholder.

Since the planner has a defined manipulation space the possible positions for the robot are defined with several `possiblePos` statements. This is necessary because Prolog does not support sorts and so all possible positions have to be defined in advance. All these statements are defined by their parameters which represent their area (`tower` or `heap`), the block (e.g. `a` or `n` as placeholder) and an consecutive identifier.

Listing 5.1 is an example configuration of the environment with the following meaning:

1. The robot is currently at position 1 in the area `tower`
2. There are four positions around the tower
3. Each block has two fictive positions to grab it when the block is on the heap.

```
1     position(tower, n, 1),
2     possiblePos(tower, n, 1),
3     possiblePos(tower, n, 2),
4     possiblePos(tower, n, 3),
5     possiblePos(tower, n, 4),
6     possiblePos(heap,  a, 1),
7     possiblePos(heap,  a, 2),
8     possiblePos(heap,  b, 1),
9     possiblePos(heap,  b, 2),
10    possiblePos(heap,  c, 1),
11    possiblePos(heap,  c, 2),
12    possiblePos(heap,  d, 1),
13    possiblePos(heap,  d, 2),
```

Listing 5.1: The robot's start and possible position

Blocks

A block has multiple different properties. Compared to the logical definition in the Concept chapter where we define a block with an $pos(Id, Pos)$ statement we use a different but similar definition in Prolog. In this case we define a block with a name, an area, a level and a position. The name is the unique identifier of each block. The area

can either be `heap` or `tower` representing that the block is anywhere in the environment (heap) or already built into the tower. The position represents the position id mentioned in the section above. Listing 5.2 shows some blocks in an example environment. Block `a` and `b` are already built into the tower and the blocks `c` and `d` are on the heap.

```
1     block(a,tower,1,1),
2     block(b,tower,1,2),
3     block(c,heap,0,0),
4     block(d,heap,0,0),
```

Listing 5.2: Some block statements

These Prolog statements would be equivalent to the following ones defined in the abstract logical domain.

$$pos(t_1, Tower)$$
$$pos(a, tpos(t_1, 1, dir, 1))$$
$$pos(b, tpos(t_1, 1, dir, 2))$$
$$pos(c, Heap)$$
$$pos(d, Heap)$$
$$pos(Tower, World)$$
$$pos(Heap, World)$$

The direction dir is not set in this example because this is defined in another way. This is described in the next section.

Positions of a block in the tower

When a block is put on the tower there are several different positions (as visible in Figure 5.3). From the logic point of view we defined a block's position in a tower with

$$pos(id, tpos(t, l, d, n)) \tag{5.3}$$

To handle this the planner needs to have knowledge about how many blocks (n) it can put in parallel on each level (l) and how many different directions (d) for the block's alignment exist. Figure 5.3 is a top view of the tower with the four resulting possible positions.

The number of possible parallel block on each level is defined with a `free(level, id)` statement. The direction for each level is set with an `direction(level, dir)` statement. And all possible directions are defined with `possibleDir` statements. Listing 5.3 is an example configuration.

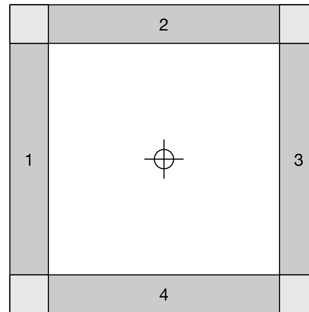

Figure 5.3: Possible positions for a block in a tower

```
1    free(1,1),
2    free(1,2),
3    direction(1,0),
4    possibleDir(1),
5    possibleDir(2)
```

Listing 5.3: Position statements

In detail this Listing means that:

1. There are two free positions on level 1 which have the ids 1 and 2
2. The direction for level 1 is not selected yet (value is set to zero)
3. There are two possible directions (one and two) for the block's alignment

Constructing state

To know on which level the robot is working at the moment and which levels are already finished the following statements in Listing 5.4 are necessary. These statements mean that the robot is currently working on level 1(`line 1`) and none of the levels are finished(`line 2`).

```
1    constructing(1),
2    done(0)
```

Listing 5.4: Construction statements

The following example shows the Prolog representation of a previous mentioned environment.

Example

The state for the example environment from Figure 4.5 and Equation 4.28 would be defined as follows:

```
1      position(tower, n ,1),
2      possiblePos(tower, n, 1),
3      possiblePos(tower, n, 2),
4      possiblePos(tower, n, 3),
5      possiblePos(tower, n, 4),
6      possiblePos(heap,  d, 1),
7      possiblePos(heap,  d, 2),
8      possiblePos(heap,  e, 1),
9      possiblePos(heap,  e, 2),
10     block(a,tower,1,1),
11     block(b,tower,1,2),
12     block(c,tower,2,1),
13     block(d,heap,0,0),
14     block(e,heap,0,0),
15     free(2,2),
16     direction(1,1),
17     direction(2,2),
18     possibleDir(1),
19     possibleDir(2),
20     done(0),
21     done(1),
22     constructing(2)
```

Listing 5.5: State of an example environment

5.2.2 Successor Arithmetic

Since it is not possible to handle arithmetic operations inside the strips process the planner uses a successor arithmetic to represent simple mathematical operations. If the planner for example uses the level 0 to work and wants to increase the level it has to apply an incrementation.
To do so it surrounds the current value with a successor function (`succ(0)`). With the next incrementation the value is surrounded again (`succ(succ(0)`) and so on. The mathematical evaluation of these values happens when the values are really needed (e.g. `succ(succ(0)) = 2`).

5.2.3 Actions

The following Listings show the Prolog representations of each action. The operation's parameters have the following structure.

1. The operations name (`NAME`) as an identifier string
2. Positive preconditions (`POS_PRECOND`) as a set of propositions which have to be fulfilled
3. Negative preconditions (`NEG_PRECOND`) as the propositions which must not be fulfilled
4. Items to remove from the state list (`DEL_ITEMS`)
5. Items to add to the state list (`ADD_ITEMS`)

Written in the PROLOG format SIMSTRIPS is using this results in the following syntax.

```
1 opn(NAME,
2     POS_PRECOND,
3     NEG_PRECOND,
4     DEL_ITEMS,
5     ADD_ITEMS).
```

Listing 5.6: Operation Syntax

Move the robot

To build a tower it is necessary to move the robot between different locations. In the Concept chapter we defined the preconditions for moving a robot r from an origin location o to an goal position g as follows:

$$\neg pos(r, g) \land o \neq g \tag{5.4}$$

This basically means that the robot can move from origin (`position(PLACE_OLD,POS_OLD,ID_OLD)`) to any possible position (`possiblePos(PLACE,POS,ID)`) if this new position is free. In Prolog this simple precondition was built on top of that so that the goal position has to be in a different area than the origin one (neg. precond. `position(PLACE,_,_)`).

The effect of this action is that the current position (`position(PLACE_OLD,POS_OLD,ID_OLD)`) is replaced by the new one (`position(PLACE,POS,ID)`).

```
1 opn(moveTo(PLACE, POS, ID),
2     [position(PLACE_OLD,POS_OLD,ID_OLD), possiblePos(PLACE,POS,ID)],
3     [position(PLACE,_,_)],
4     [position(PLACE_OLD,POS_OLD,ID_OLD)],
5     [position(PLACE,POS,ID)]).
```

Listing 5.7: Action to finish a level

Grab a block

The grab action is the logical operation of how to grab a block from the heap. As mentioned in section 4.2.5 it is possible to execute this action if the robot r is on a position near the block b and the gripper has to be empty.

$$\exists y : pos(r, mpos(b, y)) \wedge \neg\exists\ b' : pos(b', rpos(r)) \tag{5.5}$$

This results in a changed position definition of the block.

$$pos(b, rpos(r)) \wedge \neg\exists x : pos(b, x) \wedge x \neq rpos(r) \tag{5.6}$$

In Prolog this is defined as an action named `grabFromHeap(A)`.
Prerequisites for this action:

- The block has to be on the Heap (line 2)
- The robot's position has to be near the block (line 2)
- No other block is grabbed (line 3)

To check if there is no other block grabbed we use a proposition with Prolog's wild card `_`. This allows Prolog to use any term instead as an anonymous variable. We use this proposition as a negative precondition so it checks if there is any block with the give parameters but with any identifier (line 2).

If it these prerequisites are fulfilled the block changes its position definition from `heap` to `grab`. On top of that this action also removes the possible positions for the robot on the heap near this block. This is used to decrease the size of the possible positions for future actions because these positions are definitely not needed anymore in the planning process.

```
1 opn(grabFromHeap(A),
2     [block(A,heap,0,0), position(heap,A,_)],
3     [block(_,grab,0,0)],
4     [block(A,heap,0,0), possiblePos(heap,A,1), possiblePos(heap,A,2)],
5     [block(A,grab,0,0)]).
```

Listing 5.8: Action to grab a block from the heap

Put the first block on a new level

The action of putting a block on the tower is split up into two different actions. These actions are "putting the first block on the tower" and "putting a new block parallel to an existing one".
If a level is finished then a new block has to be put as first one on the next level. On top of the logical definition of the standard put action in the following equations the Prolog planner has to deal with the state of each level.

$$\exists b, t', l', d', p' : p = tpos(t', l', d', p') \wedge pos(b, rpos(r)) \wedge \exists n : pos(r, mpos(t', n)) \wedge \neg\exists b' : pos(b', p) \tag{5.7}$$

Effect:

$$pos(b, p) \wedge \neg pos(b, rpos(r)) \tag{5.8}$$

As mentioned before we added a `constructing` and a `done` state for each level. On top there hast to be a `direction` selected for each new level. Since this action puts the first block on each level it has to handle some of these additional tasks.

These additional and the logical prerequisites sum up to the following preconditions:

1. The robots position has to be at the tower. This part of the logical definition in Equation 5.7 ($\exists n : pos(r, mpos(t', n))$) projects to the Prolog statement `position(tower,_,_)` which represents that the robot is at any position around the tower.

2. A block (b) has to be grabbed by the robot ($pos(b, rpos(r))$) which is represented by the positive precondition `block(A, grab, 0, 0)`.

3. There is no existing block on this new level (neg. precond. `block(_,tower,LEVEL,_)`) and the direction for this level is not selected yet (pos. precond. `direction(LEVEL,0)`).

4. The previous level has to be finished (finished levels are marked with `done(PRE_LEV)`) and the new level has to be marked as under construction `constructing(succ(PRE_LEV))`.

5. The direction for this level is not yet selected (`direction(LEVEL,0)`).

6. There have to be positions on this level which are marked as free (`free(LEVEL,POS)`). The free statements are added to the states by other actions (mentioned later).

If all these preconditions are fulfilled the following effects can be applied:

1. The direction for this level is selected. This is handled by the the selection of a possible direction (pos. precond. `possibleDir(DIR)`) which is different than the direction of the previous level (neg. precond. `direction(PRE_LEV,DIR)`). Then the selected direction is set for this level (add state `direction(LEVEL,DIR)`).

2. The block is put on the level. As logically defined above with $pos(b, p) \wedge \neg pos(b, rpos(r))$ the block is removed from the robots gripper by removing `block(A,grab,0,0)` from the state and put to the new position by adding `block(A,tower,LEVEL,POS)` to the state.

The following listing is a definition of the full action.

```
1 opn(putFirst(A,LEVEL,POS,DIR),
2     [block(A,grab,0,0), free(LEVEL,POS), direction(LEVEL,0),
         position(tower,_,_), possibleDir(DIR), done(PRE_LEV),
         constructing(succ(PRE_LEV))],
3     [block(_,tower,LEVEL,_), direction(PRE_LEV,DIR)],
4     [free(LEVEL,POS), block(A,grab,0,0), direction(LEVEL,0)],
5     [block(A,tower,LEVEL,POS), direction(LEVEL,DIR)]).
```

Listing 5.9: Action to put the first block on each level

Put a block parallel to another one

If the first block was already dropped another block can be put in parallel to the first one. To do so the current state has to meet the following preconditions:

1. A block has to be grabbed by the robot which is represented by the positive precondition `block(A, grab, 0, 0)`.

2. This is not the first block on this level which is indicated by another block on this level (pos. precond. `block(B,tower,LEVEL,_)`).

3. The direction is already selected for this level (pos. precond. `direction(LEVEL,DIR)`). A already selected direction has a value different to zero (neg. precond. `direction(LEVEL,0)`).

4. There is a free position on this level (pos. precond. `free(LEVEL,POS)`)

With these preconditions fulfilled the same effect as above (removing the block from the gripper and adding it to the tower) can be applied.

```
1 opn(putParallel(A, B, LEVEL, POS, DIR),
2     [block(A,grab,0,0), block(B,tower,LEVEL,_), free(LEVEL,POS),
         position(tower,_,_), direction(LEVEL,DIR), constructing(LEVEL)],
3     [direction(LEVEL,0)],
4     [free(LEVEL,POS), block(A,grab,0,0)],
5     [block(A,tower,LEVEL,POS)]).
```

Listing 5.10: Action put a block parallel to another one

Constructing and finishing a level

As mentioned above the planner marks levels as finished (`done(LEVEL)`) and under construction (`constructing(LEVEL)`) during the process of building a tower. This is necessary to easily determine the already finished levels and the level it is currently working on in each planning step.
The `finishLevel` action does this job by marking the finished level with `done` and the preparation of the next level. The preconditions for this action are:

1. The level under construction has already two blocks stored (pos. precond. `block(_,tower,LEVEL,1)` and `block(_,tower,LEVEL,2)`)

2. There are not a additional free positions on this level (neg. precond. `free(LEVEL,_)`)

3. As an optimization this action can only be executed after putting a block on the tower. This is valid if the robot is at any position at the tower (pos. precond. `position(tower,_,_)`) and no block is grabbed (`block(_,grab,_,_)`)

If these preconditions are met this action results in the following effects:

1. The current level is marked as finished (`done(LEVEL)`) and the constructing marker is removed from this level (`constructing(LEVEL)`).

2. The successor level is marked as under construction and prepared for usage in the next planning steps. In this preparation free positions are added to the level (`free(succ(LEVEL),1)` and `free(succ(LEVEL),2)`) and the levels direction is set to undetermined (`direction(succ(LEVEL),0)`).

```
1 opn(finishLevel(LEVEL),
2     [constructing(LEVEL), position(tower,_,_), block(_,tower,LEVEL,1),
         block(_,tower,LEVEL,2)],
3     [free(LEVEL,_), block(_,grab,_,_)],
4     [constructing(LEVEL)],
5     [done(LEVEL), constructing(succ(LEVEL)), free(succ(LEVEL),1),
         free(succ(LEVEL),2), direction(succ(LEVEL),0)]).
```

Listing 5.11: Action to finish a level

5.2.4 Solve the planning problem

The search in the state graph is a combination of multiple queries to different modules. Listing 5.12 is the code snippet of the planning process. The following enumeration is a detailed description of the code snippet. The snippet is divided into four regions.

1. Logical check
   The first query is to to find an operation which is valid from the logical point of view. This happens with a selection of an operation out of all possible ones. As defined above all actions are defined with `opn` statements. The `opn` call in line three selects arbitrary action out of the defined ones. This call determines the operation itself, it's preconditions and it's effects.
   The current state is then checked against the positive preconditions (line 4). This happens with a check if the positive precondition definition is a subset of the current state. The same happens with the negative precondition in line 5.

2. Call to the geometric planner
   The geometric check of an action is a simple call to a python script (line 8). This script triggers the geometric planning process and it's return value indicates if the action is possible in the real environment. To do so the current state is written to a file(line 7) which is then read by the script. The script returns either zero for a failed geometric planning or one for a successful geometric planning. Figure 5.2 shows a sequence diagram of how the two sides communicate with each other.

3. Apply the action to the state
   If the action is valid from both the logical and the geometric side the current state and plan is adapted. Then the the planner is triggered again with the new state.

4. Exit condition
   When the plan meets the goal requirements (line 17 and 18) the planner prints the plan and exits the planning process.

```
1  solve(State, Goal, NotGoal, Sofar, Plan):-
2      //Logical Check
3      opn(Op, Preconditions, NotPreconditions, Delete, Add),
4      is_subset(Preconditions, State),
5      is_not_subset(NotPreconditions, State),
6      //Call to the geometric planner
7      write_state(State, Op),
8      shell('python plan.py', ReturnVal),
9      ReturnVal == 1,
10     //Apply the action to the state
11     delete_list(Delete, State, Remainder),
12     append(Add, Remainder, NewState),
13     solve(NewState, Goal, NotGoal, [Op|Sofar], Plan).
14
15 solve(State, Goal, NotGoal, Plan, Plan):-
16     //Exit condition
17     is_subset(Goal, State),
18     is_not_subset(NotGoal, State),
19     write_sol(Plan).
```

Listing 5.12: The tree search in the state tree

5.2.5 Optimizations

Length limitation

Since PROLOG uses depth first search we need to prevent that it searches too far in the depth in one of the search tree's branches. The plans in the following example (Listing 5.13 and 5.14) are both correct. The problem with the plan in Listing 5.14 is that it is too long even though it is correct. The steps 15 and 16 are not really necessary.

Example

```
1 moveTo(heap,a,1)
2 grabFromHeap(a)
3 moveTo(tower,n,1)
4 putFirst(a,1,1,1)
5 moveTo(heap,b,1)
6 grabFromHeap(b)
7 moveTo(tower,n,1)
8 putParallel(b,a,1,2,1)
9 finishLevel(1)
10 moveTo(heap,c,1)
11 grabFromHeap(c)
12 moveTo(tower,n,1)
13 putFirst(c,2,1,2)
14 moveTo(heap,d,1)
15 grabFromHeap(d)
16 moveTo(tower,n,1)
17 putParallel(d,c,2,2,2)
18 finishLevel(2)
```

Listing 5.13: Plan with ideal length

```
1 moveTo(heap,a,1)
2 grabFromHeap(a)
3 moveTo(tower,n,1)
4 putFirst(a,1,1,1)
5 moveTo(heap,b,1)
6 grabFromHeap(b)
7 moveTo(tower,n,1)
8 putParallel(b,a,1,2,1)
9 finishLevel(1)
10 moveTo(heap,c,1)
11 grabFromHeap(c)
12 moveTo(tower,n,1)
13 putFirst(c,2,1,2)
14 moveTo(heap,d,1)
15 moveTo(tower,n,1)
16 moveTo(heap,d,2)
17 grabFromHeap(d)
18 moveTo(tower,n,1)
19 putParallel(d,c,2,2,2)
20 finishLevel(2)
```

Listing 5.14: Longer than ideal but correct plan

To ensure the shortest plan iterative deepening [34] is used. This approach limits the search depth for each iteration.

```
1 IterativeDeepening()
2 {
3   depth := 0;
4   while (depth < infinity)
5   {
6     DepthLimitedSearch (depth);
7     depth := depth + 1;
8   }
9 }
```

Listing 5.15: Iterative Deepening

In this specific case it is possible to calculate the ideal length of a plan. Instead of an iteration from zero to a specific plan length it is a simple limitation to the absolute length.

To place one block at least the following steps are necessary:

1. Move robot to a grasp position at the heap
2. Grasp a block
3. Move to a position at the tower
4. Place the block

For each level there is also one additional step added. This action finishes this level. The goal of the planning process is defined by the height of the tower. With two blocks per level the count of blocks is calculated by $N = 2 \cdot Height$.
The length of a ideal path can be calculated as follows:

$$n = 4 \cdot N + \frac{N}{2} \tag{5.9}$$

with N as the number of blocks and n the ideal plan length

If we analyze Listings 5.13 and 5.14 with Equation 5.9 the ideal plan length for 4 blocks is

$$n = 4 \cdot 4 + \frac{4}{2} = 16 + 2 = 18 \tag{5.10}$$

As already described above Listing 5.13 has the ideal length of 18 steps whereas Listing 5.14 with 20 steps is longer than the ideal length.

Toggle prevention

It happens that PROLOG reaches a state where the robot toggles between to operations respectively states. The following example shows one of these cases:

Example

```
1 moveTo(heap,a,1)
2 grabFromHeap(a)
3 moveTo(tower,n,1)
4 moveTo(heap,a,1)
5 moveTo(tower,n,1)
6 moveTo(heap,a,1)
7 moveTo(tower,n,1)
8 moveTo(heap,a,1)
```

Listing 5.16: Toggling between two states

The problem with these lists of actions which do the same again and again is that they are useless but valid operations. To prevent this a small check was added to the planner. This check prohibits this behavior and drastically decreases the search tree for the planner.

Prevention of a row of same actions

Since it is possible to calculate the ideal length of a plan it does not make sense to execute two actions of the same type after each other. For example if the robot is at `position(heap, b, 1)` and wants to navigate to `position(tower, n, 1)` both plans in Listing 5.17 and 5.18 are valid. The iterative deepening optimization is also preventing this behavior. But it takes much longer to find a valid plan without this prevention.

Example

```
1 grabFromHeap(b)
2 moveTo(heap,a,1)
3 moveTo(tower,n,1)
4 putFirst(a,1,1,1)
```

Listing 5.17: Longer than ideal but correct plan

```
1 grabFromHeap(b)
2 moveTo(tower,n,1)
3 putFirst(a,1,1,1)
```

Listing 5.18: Plan with ideal length

These optimization change the Prolog planning algorithm as defined in Listing 5.19. The selected action and the current plan is checked against optimizations before the geometric planner is triggered. The toggle prevention is the simple call in line 7 which checks if the second last action is the same one as the current selected. Line 8 and 9 implement the plan length limitation and line 10 the call to the last action check optimization.

```
1  solve(State, Goal, NotGoal, Sofar, Plan):-
2      //Logical Check
3      opn(Op, Preconditions, NotPreconditions, Delete, Add),
4      is_subset(Preconditions, State),
5      is_not_subset(NotPreconditions, State),
6      //Optimizations
7      not(nth1(2,Sofar,Op)),
8      length(Sofar, LEN),
9      LEN =< 18,
10     last_action_check(Sofar, Op),
11     //Call to the geometric planner
12     write_state(State, Op),
13     shell('python plan.py', ReturnVal),
14     ReturnVal == 1,
15     //Apply the action to the state
16     delete_list(Delete, State, Remainder),
17     append(Add, Remainder, NewState),
18     solve(NewState, Goal, NotGoal, [Op|Sofar], Plan).
19
20 solve(State, Goal, NotGoal, Plan, Plan):-
21     //Exit condition
22     is_subset(Goal, State),
23     is_not_subset(NotGoal, State),
24     write_sol(Plan).
```

Listing 5.19: The tree search in the state tree

5.3 Geometric Planner - Dispatcher

As mentioned before the geometric planning is handled by two different planner modules - MoveIt! for object manipulation and MoveBase for 2D navigation in the environment. The dispatching of the requests to the correct planner is handled by the planner interface.

5.3.1 Planner Interface

The planner interface class implements an action server which handles the planning requests from the PROLOG planner. It also handles the following preprocessing tasks:

- Transform abstract to geometric positions
  It calculates the calculation of the geometric positions in the environment from the abstract positions. Section 4.3 shows the methods how to calculate the values.

- State interpretation
  The state string from the PROLOG planner is interpreted and the environment is built from these states. From this state string the robot's position as well as the state of the tower and so on is extracted.

- Action interpretation
  The requested action and all it's parameters are processed and evaluated.

In the next step the interpreted actions are sent to the ROS modules which handle the geometric planning. These modules then return if the planning request was successful. The result is then passed back to the action client which requested the evaluation.

5.3.2 Geometry Publisher

The geometry publisher is a simple helper node which publishes all necessary values to make the planning progress visible. It always runs in the background so the user is able to visualize the progress of the planning requests. The published values are for example the new 2D movement planning goal or the goal pose of an arm movement.

5.3.3 Gripper Interface

Since the gripper needs no real planning because we only open or close it the planner interface is able to do that directly through a gripper interface. The gripper interface handles the request to open and close the gripper. The gripper's force is limited to $60N$. The state open is defined with a width between the two fingers of $11cm$. The closed state is set to $5cm$. If there is a block in the gripper and the gripper should close it automatically stops when the force is exceeded.

5.4 2D Navigation - MoveBase

This section deals with the implementation of the 2D Navigation for the robot. First the localization of the robot in the environment is described and then the geometrical path planning in this environment.

5.4.1 Localization

The robot uses a laser scanner by Sick for the localization in the environment. This localization is handled in two different ways.

- If there is no map available for a new environment `gmapping` [35][36] is used. It provides a simultaneous mapping of the environment and localization in the new created map. For this purpose `gmapping` combines the sensor information of the laser scanner and the odometry information from the robot. This is mainly used to create a new map which can be saved with the `map_saver` tool.

- When the map was successfully created the standard toolchain can be used. This toolchain consists of the ROS packages `amcl` [37][38] and `mapserver` [39]. The `mapserver` loads a map and publishes it and `amcl` uses the odometry and laser scanner sensor information to localize in this map.

Figure 5.4 shows a screenshot of the localization of the robot in an environment.

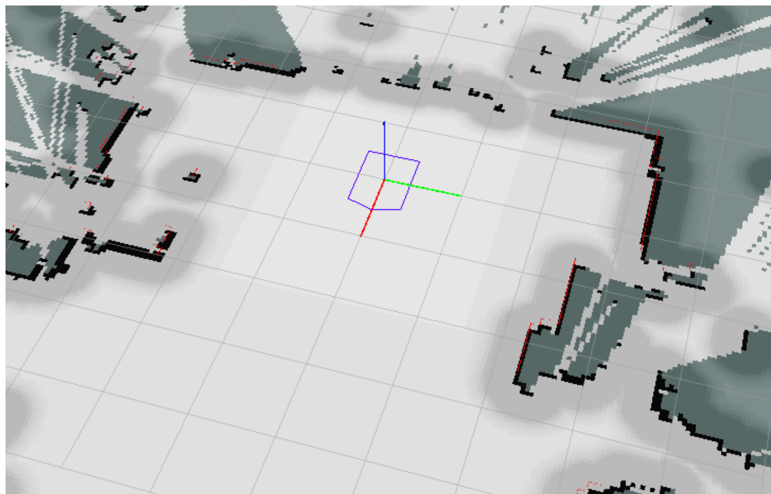

Figure 5.4: Localization of the robot in the map (free area in gray, obstacles in black and unmapped areas in dark gray) with the robot as the blue polygon

The black borders are the walls and obstacles in the map. The thin red lines are the laser range finder's sensor information. The coordinate system with the red, green and blue

axis is the robot's base link. The rectangle like polygon around this coordinate system is the robot's footprint. The light area in the middle of the figure is free space (which is the room the robot is in) with the darker areas near the black borders which are the inflated cost intensive areas. The even darker areas outside outside the black borders are areas which are not yet captured by the mapping system.

This results in the following transformations tree (Figure 5.5) inside ROS. The transformation between the `odom` and the `base_link` frames are handled by the robot controllers odometry publisher. `amcl` moves the `odom` frame in order to guarantee the correct transforms between `map` and `base_link`.

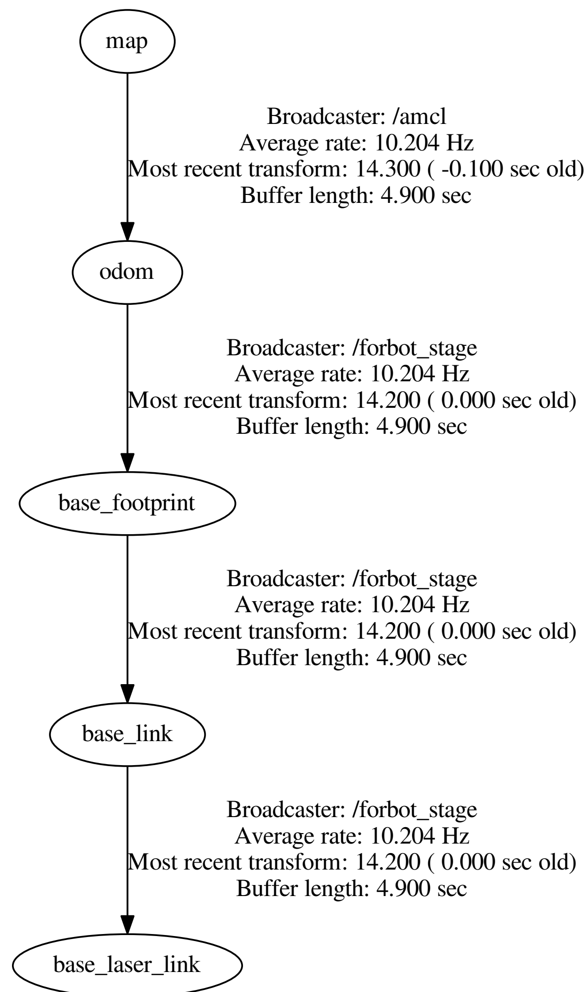

Figure 5.5: Robots base transforms

5.4.2 Move the robot

To move the robot's base the ROS toolkit MoveBase [25][26] is used. As described in section 3.3 MoveBase consists of multiple modules. These modules have to be configured with separate config files.

The local planner defines some hardware limits of the robots. These hardware limits are for example maximum and minimum velocities, acceleration limits or rotation speeds. These values are very dependent on the robot and can be found in the robot's datasheets.

The costmaps are configured with three different parametrizations. There is a global configuration which defines the most important stuff like the robots footprint, the sensor source and how detected obstacles are handled. This also includes for example an inflation radius which defines how far obstacles are inflated.

The specific parameters for the global and local costmap define the geometrical frames they use, the update frequency and the map size. In common the local costmap uses a smaller map size (we use $3m \times 3m$) than the global costmap ($10m \times 10m$).

Figure 5.6 show Move Base executing a request.

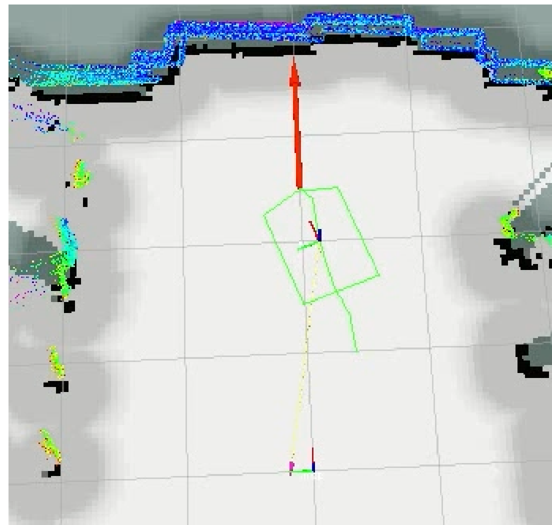

Figure 5.6: Move base executing a path (green line) for the robot (green polygon) to a specified goal (red arrow) in a map (free areas in gray, obstacles in black and unmapped areas in dark gray) with sensor data (colored dots)

The red arrow is the requested goal pose. The coordinate system on the bottom of the figure is the map's origin. The second coordinate system is the robot's base link which

is surrounded by the green polygon representing the robots footprint. On the left and the right side there are obstacles which are marked black and surrounded with the gray inflation. The green line is the global planned path which the robot tries to follow. The colorful dots all around the area represent the sensor information provided by the laser scanner for localization with an added decay time.

5.4.3 Environment Simulation

ROS Stage [40] is a simulator for a robot and it's environment based on the Stage project [41]. Stage has a defined map as environment and a robot with various sensors. The sensor values are calculated by raytracing through the simulated environment. ROS Stage publishes a transform of the given robot as well as the sensor information. In the scope of this project a simple robot with a laser range finder was used.

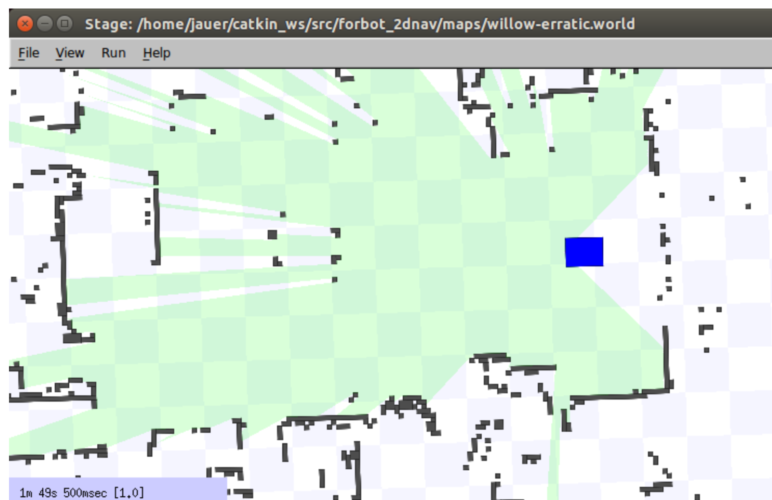

Figure 5.7: Stage ROS with a map with the robot in blue and the laser sensor data in green

During the simulation it is necessary to change the environment by moving the blocks. ROS Stage is not able to handle dynamic added objects. For this purpose an extension to the existing Stage ROS project was developed [42]. Since the Stage coordinate system is different to the map coordinates in ROS we had to develop a transformator between these two environments.

The localization of the robot in ROS publishes a transform between the `map` and `base_link`. This transformation is used to calculate the transform matrix between `base_link` and `map`. The following equation show how to calculate this matrix, the corresponding code

is in Listing 5.20 lines 3-9.

$$M_{base-map} = [T_{map-base} \cdot R_{map-base}]^{-1} \tag{5.11}$$

On the other hand publishes the Stage Node the ground truth of the robots position. This information is continuously monitored. It is possible to calculate the transform matrix between the Stage frame and the robots base link (lines 12-19).

$$M_{stage-base} = T_{stage-base} \cdot R_{stage-base} \tag{5.12}$$

To calculate the transform matrix between the Stage and the map link the former matrices have to be multiplied with each other (lines 22-27).

$$M_{stage-map} = M_{stage-base} \cdot M_{base-map} \tag{5.13}$$

The python method in Listing 5.20 allows us to add blocks to the simulated robots environment (plus make them visible in the laser scan) with simple map coordinates which are available in the ROS environment.

```
1  def get_stage_to_map_transform(self):
2      #robots trans in the map frame and calc matrices
3      (trans, rot) = self._transform_listener.lookupTransform('map',
           'base_link', rospy.Time(0))
4      map_base_rotation = tf.transformations.quaternion_matrix(rot)
5      map_base_translation = tf.transformations.translation_matrix(trans)
6      map_base_matrix = map_base_translation.dot(map_base_rotation)
7
8      #get the transform from base_link to map
9      base_map_matrix = inv(map_base_matrix)
10
11     #robots trans in the Stage frame and calc matrices
12     robot_pose = self._base_pose_ground_truth
13     robot_position = robot_pose.pose.pose.position
14     robot_orientation = robot_pose.pose.pose.orientation
15     stage_base_translation = tf.transformations.translation_matrix(
16         [robot_position.x, robot_position.y, robot_position.z])
17     stage_base_rotation = tf.transformations.quaternion_matrix(
18         [robot_orientation.x, robot_orientation.y, robot_orientation.z,
               robot_orientation.w])
19     stage_base_matrix = stage_base_translation.dot(stage_base_rotation)
20
21     #calculate the resulting translation and rotation
22     result = stage_base_matrix.dot(base_map_matrix)
23
24     quat = tf.transformations.quaternion_from_matrix(result)
25     euler = tf.transformations.euler_from_quaternion(quat)
26     trans = tf.transformations.translation_from_matrix(result)
27     return (trans, euler[2])
```

Listing 5.20: Map to Stage transform

This functionality is used to place blocks into the simulated environment during the planning process. For example Figure 5.8 shows the state of the following example:

```
    position(tower, n ,2),
    possiblePos(tower, n, 1),
    possiblePos(tower, n, 2),
    possiblePos(tower, n, 3),
    possiblePos(tower, n, 4),
    possiblePos(heap,  d, 1),
    possiblePos(heap,  d, 2),
    possiblePos(heap,  e, 1),
    possiblePos(heap,  e, 2),
    block(a,tower,1,1),
    block(b,tower,1,2),
    block(c,tower,2,1),
    block(d,tower,2,1),
    direction(1,1),
    direction(2,2),
    possibleDir(1),
    possibleDir(2),
    done(0),
    done(1),
    done(2)
```

Listing 5.21: State of an example environment

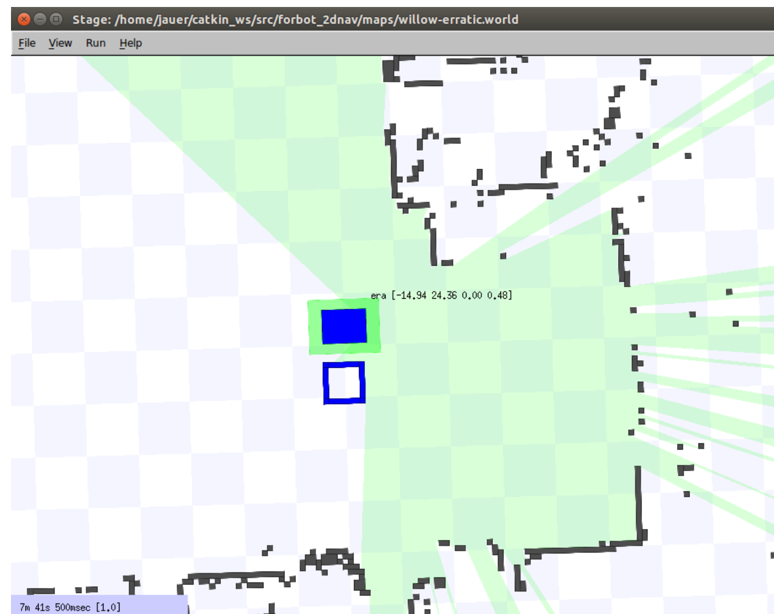

Figure 5.8: Example state with a tower(blue blocks) and a robot(blue rectangle and green sensor data) in Stage

The figure shows the robot as a blue rectangle and the tower as the square shaped

structure of four blocks below the robot. The light green area is the robot's sensor data.

5.5 Object Manipulation - MoveIt!

To move the arm MoveIT needs all the information about the transform tree and the joint values if the robot. To control the joints and gather their current sensor information a ROS node was developed by the TU Graz RoboCup team. This node uses the `IPA CANopen` driver [43] to communicate with the arm via CAN. On top of this communication layer is a controller based on `ROS Control` [44]. This controller also commands the Schunk WSG gripper.

For the geometric description of the arm the public available data from Schunk was used. These information are maintained in the `Schunk Modular Robotics` [45] package by IPA. The `schunk_description` package includes 3D meshes and URDF descriptions for the Schunk LWA4P. Only the joint limits in the URDF were slightly manipulated to prevent that the arm crashes into itself. Figure 5.9 shows the resulting view of planning process in RVIZ.

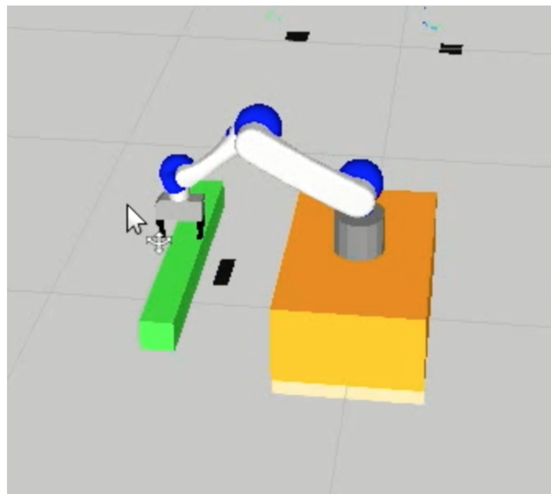

Figure 5.9: MoveIt with a Schunk LWA4P

To use this infrastructure the controller has to publish the current joint values and the transform tree. Figure 5.10 shows the complete transform tree of the running system. This is an extenden tree from Figure 5.5. The arms foot link is attached to the base link of the Forbot. The tree also includes the transforms for the gripper and an ASUS Xtion sensor.
For path planning the standard OMPL planner which is included in MoveIt was used. This planner delivered very solid results for the motion planning tasks.

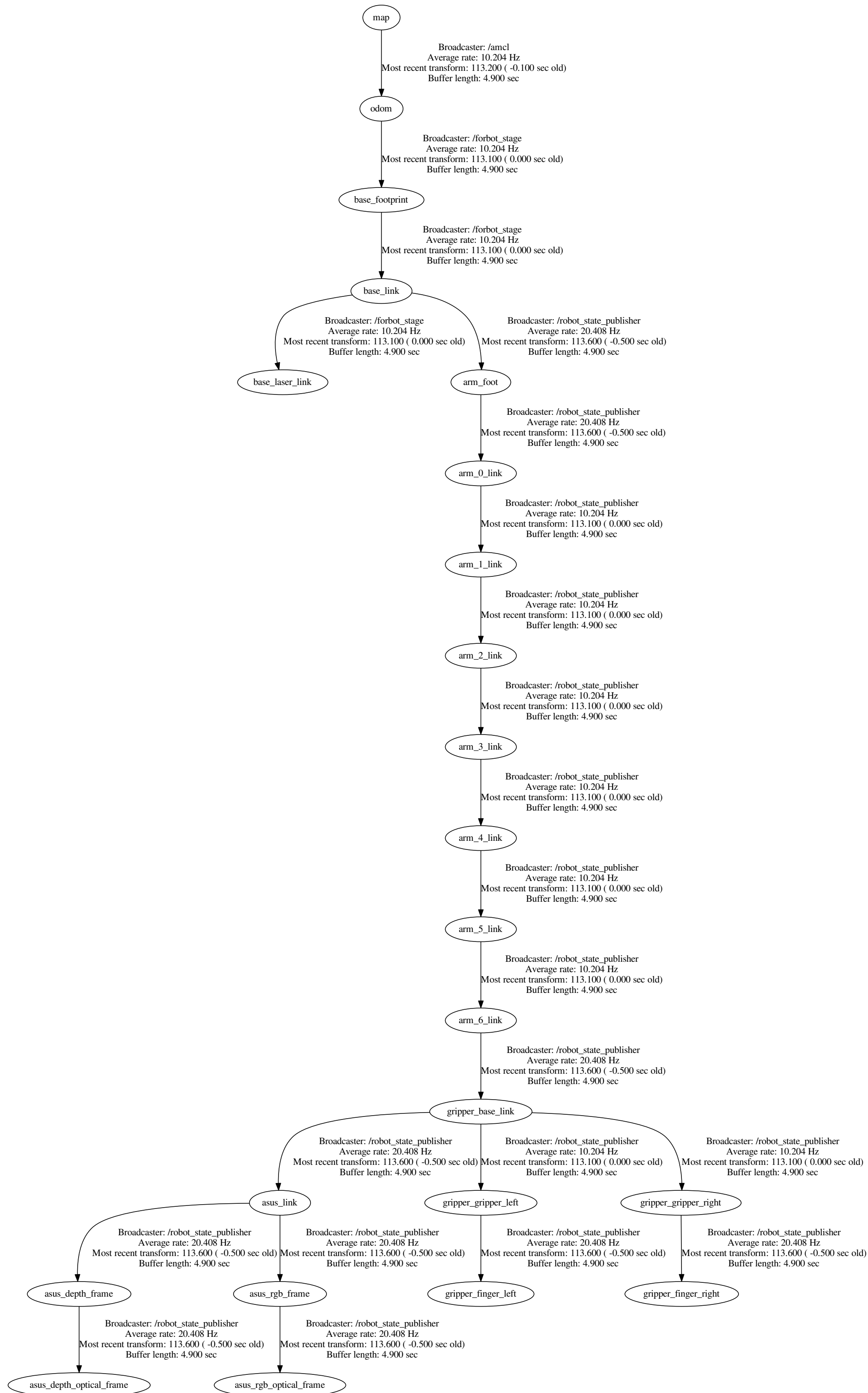

Figure 5.10: Robots base and arm transforms

Figure 5.11 shows the state defined in Listing 5.21. The figure shows the robot as an orange box with the Schunk arm mounted on top. The four blocks of the tower are shown in green in front of the robot. The red lines show the robot's sensor data.

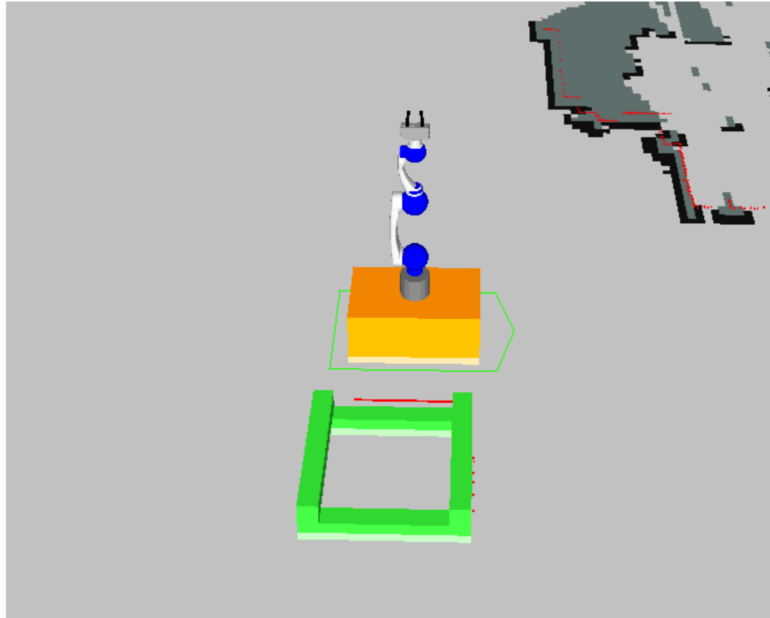

Figure 5.11: Example state in MoveIt! with a tower (green blocks) and the robot in the background (orange base with the arm on top)

5.5.1 Simulation - Fake Joint Controllers

To simulate the arm's movements and grasp operations during the planning process we use fake joint controllers. These controllers act like real controllers. They are able to interpret the trajectories calculated by MoveIt and generate joint values from them. They take the simulated joints current angle, speed and acceleration values and apply the trajectories on them. For external modules there is no apparent difference to real controllers. The fake joint controllers are a part of the ROS demo package for the Schunk arm.

5.6 Real Robot

As mentioned before a Roboterwerk Forbot combined with a Schunk LWA4P is used as robot. In addition to that a Sick LMS100 laserscanner is mounted in front of the Forbot base. Figure 5.12 shows the robot in the lab. The Forbot base is the orange, six wheeled rover. The Schunk arm is mounted on top of this base. The light blue laser scanner is mounted under the notebook.

Figure 5.12: The real robot working on a tower

The Forbot and the Schunk arm are connected to two different CAN buses. Each bus is connected to the notebook with a PeakCAN USB to CAN controller. This was necessary because the two systems were incompatible. The notebook handles all the planning and execution tasks. In addition to that a gamepad is connected to the notebook for eventually manual control of the system. The emergency stop button is mounted on the back of the rover. If this button is pressed all systems are electrically disconnected. The only systems which stay connected is the arm's logic controllers and the notebook has it's own power supply.
The robot's base is controlled by a node based on `ROS Controllers`. This node handles the velocity commands generated by the `MoveBase`. The odometry is tuned for the floor in the lab.

6 Evaluation and Experiments

This chapter evaluates the proposed planning system. The first section analyzes the performance behaviors of the abstract planner without a coupled geometric planner. The second section then deals with the fully coupled system.

6.1 Abstract Planner Evaluation

For this evaluation the geometric planner is nearly disabled. The only thing which is checked from a geometric point of view is that if the robot wants to drop a block on a position the robots base pose has to be valid to do so. For example when the robot's position is `position(tower, n, 1` it is only possible to place a block on a level with direction 1 and block position 1. To compare the different optimizations the planner had to build a tower with four blocks which represents a height of two levels. The following variants are compared:

1. No optimizations
   This does not lead to a valid plan. The problem is that if the plan length is not limited the PROLOG solver runs in the problem of depth search. It analyzes one part of the state tree until infinity. This happens because the planner always tries to apply the actions in a defined order. The abstract planner stops with a not valid plan when it's heap size is exceeded.

Example

Let's assume the robot is in on a heap position at block A. Defined by the planners action order it always tries to apply `moveTo` operations at first. So since `moveTo` is a valid and executable position the planner will append it to the plan and will for example move to the tower. In the next step exactly the same happens. Since the planner tries to apply a `moveTo` action again and this action is valid it will add e.g. the action and moves back to the heap. Then it will move to the tower and so on and so forth. This problem occurs no matter which action is selected to be checked as first by the planner.

2. Length limitation
   The limitation of the length leads to a valid plan. The problem with this solution

is that PROLOG has to try a vast amount of different states and different actions. This is extremely relevant for a good performance.

3. Toggle Prevention
This optimization prevents the planner to toggle between two positions. This decreases the number of plans with a behavior as mentioned in the example above but it does not eliminate it. This optimization uses the length limitation as base.

4. Action check
For this specific problem it makes sense to use a optimization which does not allow two actions of the same type in a row as parts of a plan. In detail this means that for example a `moveTo` action can no be followed by a `moveTo` action. This results in a massive restriction of the search space and the operations to execute. The result is a very performant abstract planner. This optimization uses the toggle prevention as base.

To evaluate the different optimizations the tower planning process is executed with different complexities. Table 6.1 lists the different variants which were executed with the different optimizations. The four variants first differ in the amount of blocks which results in different heights. The second difference between the variants is how many grab positions per block the planner has. If there are two possible positions to grab a block instead of one the system gets more complex. Variant 4 is the version which is used in the final implementation of this thesis.

Variant	Tower Height	Block Count	Grab Positions / Block
1	1	2	1
2	1	2	2
3	2	4	1
4	2	4	2

Table 6.1: Variants of building a tower with increasing complexity

Table 6.2 shows the evaluation result of the experiments. The values represent the count of the operations which had to be tried by the planner. There is a significant performance increase in the planning performance when the optimizations are enabled. The first two rows solve the simple problem and due to the very reduced size of the space tree the optimizations do not have any effects on the performance. When analyzing the variants three and four in the table the difference between a simple length limitation and all optimizations is significant. The execution of the tests with optimizations was a matter of milliseconds. The execution with length limitation took two days. Since Prolog's depth first search is deterministic all these tests were run once. The results marked with * are not terminating.

Variant	Planningsteps			
	No optimizations	Lenght limitation	+ Toggle prevention	+ Action check
1	*	41	41	41
2	*	41	41	41
3	*	7.804.597	347.912	92
4	*	373.119.842	36.196.482	92

Table 6.2: Results of the evaluation

6.1.1 Logistic Domain

To evaluate the performance of the abstract planner and it's optimizations the well known logistics competition problem [46] was modeled. The logistic competition deals with problem of packages which have to be picked up and delivered to the recipient. To solve this problems it is possible to use trucks and planes whereas trucks are only able to drive in a city and planes are used to transport packets between airports. The goal is to find an optimal plan which should be the most efficient way to deliver all packets.

This problem was solved completely without geometric evaluation. The problem has the following predicates:

- Position
  This predicate defines possible positions. It is defined by a city and an id within this city. All available positions are defined in the initial state.
- Package
  This is the package which should be transported. All packages are defined by a name and a position with city and id.
- Truck
  The trucks are used to move packages within a city. Each truck is defined by a name and a position. Trucks are able to transport one package within the bounds of a city. So it is possible to move package from `(city1, 1)` to `(city1, 2)` but not to `(city2, 1)`.
- Airplane
  Airplanes are used to transport objects between cities. The airplane is defined by name and position. Airplanes must have airports as start and end position.
- Airport
  Airports are the points between airplanes are able to move. So if you want to move an object between two cities both have to have an airport.

Based on these predicates the planner is able to execute the following actions:

- Load Truck
  If a truck is at the package's position it is possible to put it into the truck.
- Unload Truck
  The reverse action to load a truck. The unloaded object will then be at the position where the truck was.
- Drive Truck
  A truck is able to drive between the positions in a city. A truck can either be empty or loaded with a package.
- Load Airplane
  Same as loading a truck. The package has to be on the same position as the plane.
- Unload Airplane
  Same as unloading a truck as well. The unloaded object is then on the position of the plane.
- Fly Airplane
  An airplane is able to move between airports. The airplane can either be empty or loaded with an object.

Example

The following listing shows the start state of a simple example for a logistics task of shipping a package from one city to another one. The position of the package is `(city1, 1)` and the goal position is `(city2, 1)`. The definition of the goal state is `package(a, city2, 1)`.

```
1     package(a, city1, 1),
2     truck(truck1, city1, 2),
3     pos(city1,1),
4     pos(city1,2),
5     plane(plane1, city1, 2),
6     airport(city1, 2),
7     airport(city2, 2),
8     pos(city2,1),
9     pos(city2,2),
10    truck(truck2, city2, 2)
```

Listing 6.1: Logistics Domain - Simple Example

The result of the planning process is the following plan:

```
1	driveTruck(truck1,city1,1)
2	loadTruck(a,truck1)
3	driveTruck(truck1,city1,2)
4	unloadTruck(truck1)
5	loadPlane(a,plane1)
6	flyPlane(plane1,city2,2)
7	unloadPlane(plane1)
8	loadTruck(a,truck2)
9	driveTruck(truck2,city2,1)
10	unloadTruck(truck2)
```

Listing 6.2: Logistics Domain - Simple Example - Solution

In detail this plan means:

1. The truck has to drive from his origin position to the package
2. The package has to be loaded into the truck
3. The truck drives to the airport
4. On the airport the truck has to unload the object
5. Afterwards the package is loaded into the plane
6. The plane flies to the second city
7. After arrival in the other city the plane has to be unloaded
8. The packages has to be loaded into the truck
9. The truck has to drive to the goal position
10. On the goal position the truck has to be unloaded

To compare the results the logistics problem is also solved in two variants.

1. Ship one object to another city (example above)
2. Ship two packages but each in the same city
3. Ship two packages - one within the same city, the other has to be shipped to another city by plane

Variant	Planningsteps			
	No optimizations	Lenght limitation	+ Toggle prevention	+ Action check
1	*	132.834	76.860	14.868
2	*	66.264	45.966	15.285
3	*	1.514.868	986.538	214.848

Table 6.3: Results of the evaluation

These results outline again the significant performance improvements of the optimizations. These optimizations get the more important the more complex the task is as you

can see if you compare line 3 with line 1 and 2 in Table 6.3. Again results marked with * are not terminating.

Since all these problems were executed with a length limitation with the precalculated minimum length we also evaluated the variant three with iterated deepening. The problem was triggered with the a maximum plan length starting with one up to the precalculated minimum length of 9. Table 6.4 shows the results of each step. A very interesting point in this table is that the exhaustive search in the state tree with plan length 7 and 8 does not lead to a valid plan but take more steps than the final iteration with a plan length of 9. The sum of operations to execute if the plan length is not known increases to 1.343.142 with iterative deepening which is six times higher than the planning time with a pre calculated plan length.

Plan length	Step count	Plan found
1	102	No
2	348	No
3	1.212	No
4	4.386	No
5	16.032	No
6	59.070	No
7	220.002	No
8	827.142	No
9	214.848	Yes
Sum	1.343.142	

Table 6.4: Iterative deepening for logistics problem variant 3

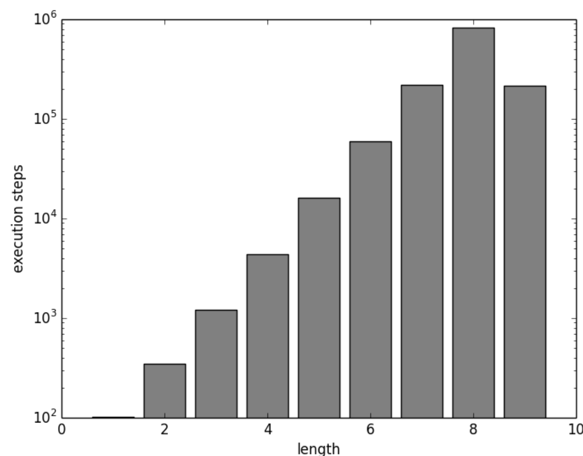

Figure 6.1: Planning steps for iterated deepening on a logarithmic scale

To improve the performance of iterative deepening the process could be parallelized like proposed in [47]. To do so the planner would launch the first N (where N is the number of CPU cores) runs in parallel. So for example on a quad core processor there would always run four iterations in parallel. If one of the iterations finds a valid plan for the problem the whole execution is terminated. This results in a drastic increase of the planning performance in the domain of planning time. This would be the most promising approach for calculating solutions for problems with no pre-calculateable length on modern computers.

6.2 Abstract and Geometric Planner Evaluation

To evaluate the complete system consisting of the abstract and the geometric planner the robot had to build towers with four blocks in different environments. To do so we ran the simulations in these environments and evaluated the runtimes, stepcount and plan length. The environments were defined as follows.

1. Simple task
   This is a simplified map of the standardized example from by Willow Garage which is packaged with Stage ROS. For this purpose some obstacles were removed to have a larger free space.

2. 2D navigation obstacle
   This task uses the same map as the simple task. The difference is that an obstacle was added next to the tower so that the planning for the 2D navigation gets more complex. Figure 6.2 shows the obstacle (red) near the tower. The blue rectangle (green encased) is the robot. The light green area is the sensor information of the robot's simulated lidar sensor (raytracing). The blue square shaped element are the blocks of the tower.

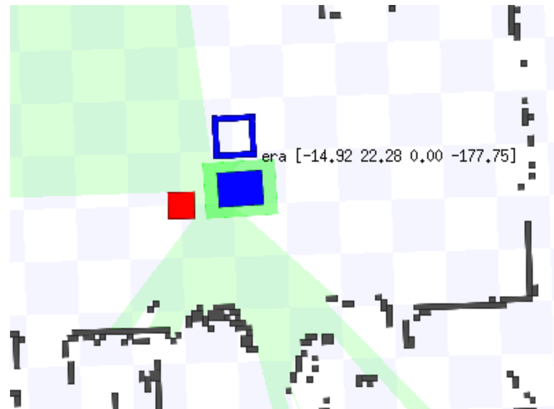

Figure 6.2: A simple obstacle (red) near the tower (blue blocks) and the robot (blue rectangle)

3. Tower obstacle
   This task also uses the same environment as the simple task. To evaluate the combination of abstract and geometrical planning there is an obstacle on the floor where the tower has to be built. This forces the planner to rotate the tower by 90 degrees. Figure 6.3 shows the result of the experiment. The green elements represent the blocks. The smaller but also block like shaped obstacle is shown under the tower (red arrow). In this case the planner would at first try to put a

block at this position and the geometric planning fails because of this obstacle. So the planner tries to build a tower with a different direction for the first level. This will then success and result in a tower as shown in the figure.
If this obstacle would not be placed there the planner would build a tower which would have a block on this position (the tower rotated by 90 degrees).

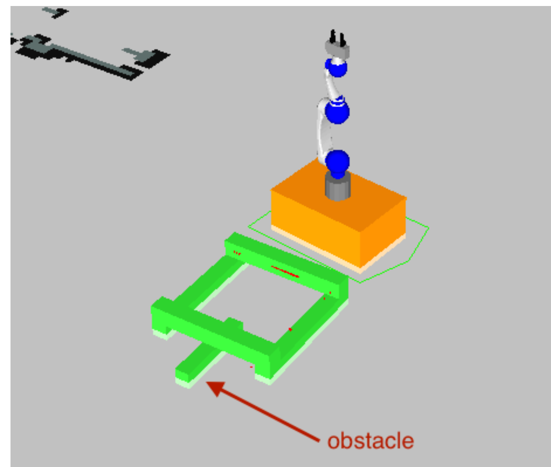

Figure 6.3: A simple obstacle beneath a tower (green blocks) and the robot (orange) with the arm (gray and blue) in the background

4. 2D + tower obstacle
   Uses an obstacle for the 2D navigation(as in 2.) as well as an tower obstacle(like in 3.).

The simulations are executed with the same parameters as the real robot works. This means that simulation durations also represent a rough estimation of the execution duration in a similar real environment.
The results in Table 6.5 show that the planner always finds a plan with the requested minimal length of 18 steps and it also takes the same amount of abstract planningsteps to find this plan. The planning duration varies because the more difficult problems need more time for the geometric verification. Figure 6.4 is a comparison of the execution times for the different problems. Because of MoveIt!'s non-determinism we used five iterations for this evaluation. The reason for the non-determinism are the randomized geometric planning algorithms.

Complexity	Environment	Avg. Duration	Std. Dev.	Length	Planningsteps
1	Simple	348.6s	2.30	18	92
2	2D Obstacle	352.2s	4.97	18	92
3	Tower obstacle	359.0s	2.73	18	92
4	2D + Tower obstacle	425.6s	3.04	18	92

Table 6.5: Results of the evaluation with five executions per complexity

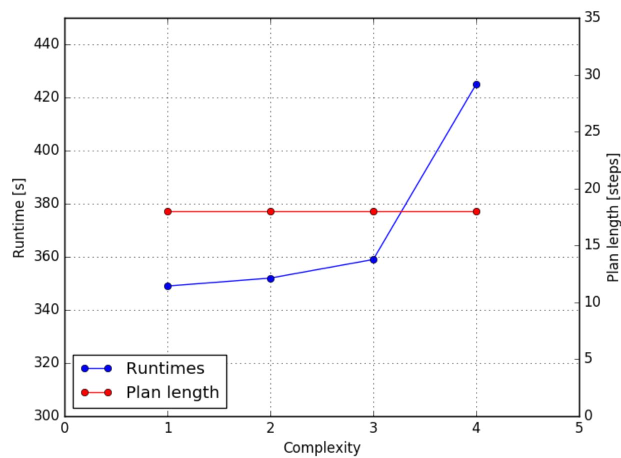

Figure 6.4: The execution time of the planning with increasing complexity

7 Conclusion

This chapter is a recap of the work of this thesis and a short discussion about the solved challenges. The implementation of a combined task and motion planner showed promising result but there is still room for improvement (see Future Work).

In this thesis we developed a combined task and motion planner. The abstract task planner was developed in Prolog and acts as the global planner with no geometric knowledge. This abstract planner is coupled to a motion planner. The motion planner uses ROS and it's modules MoveIt! and MoveBase to plan the robot's movements and the object manipulations. These two modules are separated but coupled. The abstract planner selects actions based on it's high level knowledge and state of the environment. It passes the action to the geometric planner for the verification in the real world. This geometric planning request is then processed with real world environment information. It determines if the selected action is executable. The result of this geometric verification is then used for further abstract planning. This structure gives the planning system the flexibility to solve complex tasks.

The combination of abstract and geometric planning leads to a very robust planning and plan execution. The abstract planner has a very small domain definition as well as problem descriptions but this is enough knowledge to build a tower in any environment. The planning and execution of the task in a simulated environment worked as expected and only failed in very complex environments because it was not possible to find geometrical plans with the used techniques (grasping methods, positioning, etc.). The optimizations and restrictions in the abstract planner guarantee an ideal plan length. As discussed in the evaluation these optimizations drastically decrease the number of actions the abstract planner has to try to find a valid plan. This leads to a lower amount of actions which have to be verified by the geometrical planner which results in reduced overall planning durations. On the other hand the planning and execution with real world sensor data and a real robot also worked pretty good. The robot was able to grasp blocks from a defined position and build a tower. There were some limitations with the executions of the plan but they were caused by some hardware problems of the robots six-wheeled base and had nothing to do with the planning methods. At the moment the implementation is limited to towers which need two blocks per level. But the whole system is designed in a way that changes in the rules of building the tower are very simple. So in fact this planner is also able to deal with more complex towers.

The experiments and evaluation showed that some very simple optimizations for the abstract planner resulted in a significant performance boost. This optimizations prevent exponentially rising planning times and allow the fast evaluation. In addition to that these optimizations were designed to be very specific for this planning task but the evaluation showed a contrary situation. These optimization also applied for the wide known logistics domain problems.

Overall this system is capable to handle real world planning tasks and in is usable for practical applications. There is still some room for improvement to handle even more difficult problems and environments (see Future Work).

8 Future Work

This chapter outlines additional work which could improve the outcome for the shoring task problem.

8.1 Perception

To improve the robots perception of the environment an additional sensor could be added. To do so an ASUS Xtion or Microsoft Kinect could be mounted on the gripper of the arm. This would enable the robot to dynamically measure the exact position of the tower. This is one of the problems at the moment. Due to the inaccuracy of the navigation of the robot's base (in exceptional cases up to $10cm$) this is sometimes a real problem. At the moment the robot has fixed relative (to the base) positions to drop the block. The ideal case would be that the robot drives to the desired position and calculates then the goal position for putting the block. Additional perception would solve this because sometimes the blocks are not visible in the laser scan.

8.2 Grasping the blocks from the floor

Due to the insufficient perception it is sometimes not possible detect a block. To grasp a block from the floor is also not feasible. At the moment the robot drives to the determined position and waits there for a few seconds with an open gripper. The robot expects that someone places a block in it's gripper. In addition to the perception the robot needs modifications to the gripper. The maximum opening width of the gripper is $11cm$ and the block size is $10cm$. To grasp a block with a clearance of only $1cm$ is a really hard task with all the small inaccuracies in the system. With changed fingers the gripper could extend it's opening width and make the grasping task significantly easier.

8.3 Improved goal position selection

At the moment the robot is able to choose between four fixed positions around the tower. If the robot is in a very difficult environment this could lead to problems. Figure 4.5 show the currently used positions. If these positions would be more flexible, maybe even randomized or at least multiple different fixed positions near each block this could enable the robot to drop blocks even with some more complex obstacles near the block. This

feature would go hand in hand with improved grasping mentioned in Section 8.4. One drawback of multiple positions for each block around the tower would be a bigger state tree. This can cause significant performance problems (see Chapter 6).

8.4 Improved grasping poses

The current implementation always expects that a block is grasped in the middle. If it is possible to select different base positions for putting a block on the tower it makes also sense to change the way the robot grasps a block. This would lead to even more possible states in the state tree, but it would make it possible for the robot to find plans in even more complex environments.

8.5 Improved planner

The integration of a more advanced planner will result in more performant planning results in difficult planning environments. Particularly with regard to some improvements mentioned above an improved planner would also be necessary to deal with the bigger search space.
With a better abstract planning performance it would also be possible to add some geometric background information to the abstract planner. This would decrease the amount of planning requests to the geometric planner and improve the overall planning performance because these requests are the most time intense subtasks.

List of Figures

1.1 A real world disaster site . . . 1
1.2 Example shoring task . . . 2
1.3 The robot . . . 3

3.1 Structure of MoveIt! . . . 11
3.2 The Schunk arm in MoveIt! . . . 12
3.3 Structure of MoveBase . . . 13

4.1 Modules of the planning process . . . 15
4.2 An example environment . . . 16
4.3 The direction of a level in the tower . . . 18
4.4 Block positions in a shoring tower and in Jenga towers . . . 19
4.5 An example of a real environment . . . 22
4.6 Transformations from the world frame to a block . . . 23
4.7 Transformations for the robot and it's end-effector pose . . . 28
4.8 Possible path between two positions . . . 33
4.9 Possible Robot Positions in an environment . . . 33
4.10 The robot arm's workspace . . . 34

5.1 Overview of the implementation . . . 35
5.2 Sequence of a planning request . . . 36
5.3 Possible positions for a block in a tower . . . 42
5.4 Localization of the robot in the map . . . 55
5.5 Robots base transforms . . . 56
5.6 Move base executing a path . . . 57
5.7 Stage ROS . . . 58
5.8 Example state in Stage . . . 60
5.9 MoveIt with a Schunk LWA4P . . . 62
5.10 Robots base and arm transforms . . . 63
5.11 Example state in MoveIt! . . . 64
5.12 The real robot working on a tower . . . 65

6.1 Planning steps for iterated deepening . . . 72
6.2 A simple obstacle near the tower . . . 74
6.3 A simple obstacle in the tower area . . . 75
6.4 The execution time of the planning with increasing complexity . . . 76

Listings

5.1 The robot's start and possible position 40
5.2 Some block statements 41
5.3 Position statements 42
5.4 Construction statements 42
5.5 State of an example environment 43
5.6 Operation Syntax 44
5.7 Action to finish a level 44
5.8 Action to grab a block from the heap 45
5.9 Action to put the first block on each level 47
5.10 Action put a block parallel to another one 47
5.11 Action to finish a level 48
5.12 The tree search in the state tree 49
5.13 Plan with ideal length 50
5.14 Longer than ideal but correct plan 50
5.15 Iterative Deepening 50
5.16 Toggling between two states 51
5.17 Longer than ideal but correct plan 52
5.18 Plan with ideal length 52
5.19 The tree search in the state tree 53
5.20 Map to Stage transform 59
5.21 State of an example environment 60

6.1 Logistics Domain - Simple Example 70
6.2 Logistics Domain - Simple Example - Solution 71

Bibliography

[1] International Search and Rescue Advisory Group. *INSARAG Guidelines*. URL: http://www.insarag.org/en/methodology/guidelines.html (cit. on p. 1).

[2] *United States Army Corps of Engineers' Shoring Operations Guide*. Feb. 2016. URL: http://www.scrd.ca/files/File/Community/EmergencyOps/USAR%20Shoring%20Operations%20Guide%203rd%202013.pdf (cit. on p. 1).

[3] *USAR Advanced Training Unit*. Feb. 2016. URL: http://www.fireblast.com.cn/english/Mobile-Fire-Training-05.html (cit. on p. 1).

[4] H. Levent Akin, Nobuhiro Ito, Adam Jacoff, Alexander Kleiner, Johannes Pellenz, and Arnoud Visser. "RoboCup Rescue Robot and Simulation Leagues." In: *AI Magazine* 34.1 (2013), pp. 78–86. URL: http://dblp.uni-trier.de/db/journals/aim/aim34.html#AkinIJKPV13 (cit. on p. 2).

[5] *RoboCupRescue Robot League Rules*. Feb. 2016. URL: http://wiki.ssrrsummerschool.org/doku.php?id=rrl-rules-2015 (cit. on p. 2).

[6] Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory and Practice*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004, pp. 76–78. ISBN: 1558608567 (cit. on pp. 5, 14).

[7] Daniel S. Weld. "Recent Advances in AI Planning." In: *AI MAGAZINE* 20 (1999), pp. 93–123 (cit. on p. 5).

[8] Jörg Hoffmann. "FF: The Fast-Forward Planning System." In: *AI magazine* 22 (2001), pp. 57–62 (cit. on p. 5).

[9] C. Dornhege, M. Gissler, M. Teschner, and B. Nebel. "Integrating symbolic and geometric planning for mobile manipulation." In: *Safety, Security Rescue Robotics (SSRR), 2009 IEEE International Workshop on*. Nov. 2009, pp. 1–6. DOI: 10.1109/SSRR.2009.5424160 (cit. on p. 5).

[10] Christian Dornhege. "Task Planning for High-Level Robot Control." https://www.freidok.uni-freiburg.de/data/10122. PhD thesis. University of Freiburg, 2015 (cit. on p. 5).

[11] Stéphane Cambon, Fabien Gravot, and Rachid Alami. *aSyMov: Toward More Realistic Robot Plans*. Rapport LAAS 03472. LAAS-CNRS, Oct. 2003 (cit. on p. 5).

[12] Stéphane Cambon, Fabien Gravot, and Rachid Alami. "A Robot Task Planner that Merges Symbolic and Geometric Reasoning." In: *ECAI*. IOS Press, Sept. 29, 2009, pp. 895–899. ISBN: 1-58603-452-9 (cit. on p. 5).

[13] L. P. Kaelbling and T. Lozano-Pérez. "Hierarchical task and motion planning in the now." In: *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. May 2011, pp. 1470–1477. DOI: 10.1109/ICRA.2011.5980391 (cit. on p. 6).

[14] S. Kimura, T. Watanabe, and Y. Aiyama. “Force based manipulation of Jenga blocks.” In: *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*. Oct. 2010, pp. 4287–4292. DOI: 10.1109/IROS.2010.5651753 (cit. on p. 6).

[15] G. Havur, K. Haspalamutgil, C. Palaz, E. Erdem, and V. Patoglu. “A case study on the Tower of Hanoi challenge: Representation, reasoning and execution.” In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. May 2013, pp. 4552–4559. DOI: 10.1109/ICRA.2013.6631224 (cit. on p. 6).

[16] N. Macias and J. Wen. “Vision guided robotic block stacking.” In: *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*. Sept. 2014, pp. 779–784. DOI: 10.1109/IROS.2014.6942647 (cit. on p. 6).

[17] Lars Karlsson, Julien Bidot, Fabien Lagriffoul, Alessandro Saffiotti, Ulrich Hillenbrand, and Florian Schmidt. “Combining task and path planning for a humanoid two-arm robotic system.” In: *Combining Task and Motion Planning for Real-World Applications (ICAPS workshop)*. 2012, pp. 13–20 (cit. on p. 7).

[18] Stevan Harnad. “The Symbol Grounding Problem.” In: *Physica D: Nonlinear Phenomena* 42 (1990), pp. 335–346. DOI: 10.1016/0167-2789(90)90087-6. URL: http://groups.lis.illinois.edu/amag/langev/paper/harnad90theSymbol.html (cit. on p. 7).

[19] Silvia Coradeschi and Alessandro Saffiotti. “An introduction to the anchoring problem.” In: *Robotics and Autonomous Systems* 43.2–3 (2003). Perceptual Anchoring: Anchoring Symbols to Sensor Data in Single and Multiple Robot Systems, pp. 85–96. ISSN: 0921-8890. DOI: http://dx.doi.org/10.1016/S0921-8890(03)00021-6. URL: http://www.sciencedirect.com/science/article/pii/S0921889003000216 (cit. on p. 7).

[20] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. “ROS: an open-source Robot Operating System.” In: *ICRA Workshop on Open Source Software*. 2009 (cit. on p. 9).

[21] Ioan A. Sucan and Sachin Chitta. *MoveIt!* URL: http://moveit.ros.org (cit. on pp. 10, 11).

[22] Steve Cousins Sachin Chitta Ioan Sucan. “MoveIt! [ROS Topics].” In: *IEEE Robotics & Automation Magazine* 19 (2012), pp. 18–19 (cit. on p. 10).

[23] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. “OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees.” In: *Autonomous Robots* (2013). Software available at http://octomap.github.com. DOI: 10.1007/s10514-012-9321-0. URL: http://octomap.github.com (cit. on p. 11).

[24] M. Moll I. A. Sucan and L. Kavraki. *The Open Motion Planning Library (OMPL)*. 2010. URL: http://ompl.kavrakilab.org/ (cit. on p. 12).

[25] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, and K. Konolige. "The Office Marathon: Robust navigation in an indoor office environment." In: *Robotics and Automation (ICRA), 2010 IEEE International Conference on.* May 2010, pp. 300–307. DOI: 10.1109/ROBOT.2010.5509725 (cit. on pp. 12, 57).

[26] *ROS move base.* URL: http://wiki.ros.org/move_base (cit. on pp. 12, 57).

[27] Q. Chen, Z. Sun, T. Chen, and L. Cao. "Local Path Planning for Autonomous Land Vehicle Based on Navigation Function." In: *Intelligent Computation Technology and Automation (ICICTA), 2011 International Conference on.* Vol. 2. Mar. 2011, pp. 1122–1125. DOI: 10.1109/ICICTA.2011.566 (cit. on p. 13).

[28] Brian P. Gerkey and Kurt Konolige. "Planning and control in unstructured terrain." In: *In Workshop on Path Planning on Costmaps, Proceedings of the IEEE International Conference on Robotics and Automation (ICRA.* 2008 (cit. on p. 13).

[29] D. Fox, W. Burgard, and S. Thrun. "The dynamic window approach to collision avoidance." In: *IEEE Robotics Automation Magazine* 4.1 (Mar. 1997), pp. 23–33. ISSN: 1070-9932. DOI: 10.1109/100.580977 (cit. on p. 13).

[30] Richard E. Fikes and Nils J. Nilsson. *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving.* Tech. rep. 43R. SRI Project 8259. 333 Ravenswood Ave, Menlo Park, CA 94025: AI Center, SRI International, May 1971 (cit. on p. 14).

[31] Arnold Oberschelp. "Sorts and Types in Artificial Intelligence: Workshop, Eringerfeld, FRG, April 24–26, 1989 Proceedings." In: Berlin, Heidelberg: Springer Berlin Heidelberg, 1990. Chap. Order sorted predicate logic, pp. 7–17. ISBN: 978-3-540-46965-0. DOI: 10.1007/3-540-52337-6_16. URL: http://dx.doi.org/10.1007/3-540-52337-6_16 (cit. on p. 16).

[32] *Schunk LWA4p Powerball Technical Data.* Feb. 2016. URL: http://mobile.schunk-microsite.com/fileadmin/user_upload/broshures/SCHUNK_Technical_data_LWA4P.pdf (cit. on p. 34).

[33] Tim Smith. *Artificial Intelligence Programming in PROLOG.* URL: http://www.inf.ed.ac.uk/teaching/courses/aipp/ (cit. on p. 39).

[34] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms.* 2nd. McGraw-Hill Higher Education, 2001. ISBN: 0070131511 (cit. on p. 50).

[35] G. Grisettiyz, C. Stachniss, and W. Burgard. "Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling." In: *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on.* Apr. 2005, pp. 2432–2437. DOI: 10.1109/ROBOT.2005.1570477 (cit. on p. 55).

[36] *ROS gmapping.* URL: http://wiki.ros.org/gmapping (cit. on p. 55).

[37] B. Zhang, J. Liu, and H. Chen. "AMCL based map fusion for multi-robot SLAM with heterogenous sensors." In: *Information and Automation (ICIA), 2013 IEEE International Conference on.* Aug. 2013, pp. 822–827. DOI: 10.1109/ICInfA.2013.6720407 (cit. on p. 55).

[38] *ROS amcl.* URL: http://wiki.ros.org/amcl (cit. on p. 55).

[39] *ROS mapserver.* URL: http://wiki.ros.org/map_server (cit. on p. 55).

[40] *ROS Stage.* URL: http://wiki.ros.org/stage_ros (cit. on p. 58).

[41] *Stage.* URL: http://rtv.github.io/Stage/ (cit. on p. 58).

[42] *TUG ROS Stage.* URL: https://github.com/RoboCupTeam-TUGraz/stage_ros (cit. on p. 58).

[43] *IPA CANopen Driver.* 2015. URL: https://github.com/ipa320/ipa_canopen (cit. on p. 62).

[44] *ROS Control.* 2016. URL: http://wiki.ros.org/ros_control (cit. on p. 62).

[45] *Schunk Modular Robotics.* 2016. URL: https://github.com/ipa320/schunk_modular_robotics (cit. on p. 62).

[46] Drew Mcdermott. "The 1998 AI Planning Systems Competition." In: *AI Magazine* 21 (2000), pp. 35–55 (cit. on p. 69).

[47] A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications.* Amsterdam, The Netherlands, The Netherlands: IOS Press, 2009, pp. 495–496. ISBN: 1586039296, 9781586039295 (cit. on p. 73).