Parallel Plan Execution on a Mobile Robot with an HTN Planning System

A Resource Based Approach

Bachelor Thesis

Area of studies: Robotics

Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

Department Informatics

presented by:	Lasse Einig
field of study:	Informatics
matriculation number:	591 83 87
first supervisor:	Prof. Dr. Jianwei Zhang
second supervisor:	Prof. Bernd Neumann, Ph.D.

Parallel Plan Execution on a Mobile Robot

A Resource Based Approach

Lasse Einig

Abstract

English

Time is not only a valuable resource concerning resource allocation time on robots for task execution but also concerning duration of development cycles in research and thus the resource human. Experience gained during participation in the master project "Intelligent Robotics" together with insights into the EU-funded project RACE motivated to optimize time and resource efficiency on the *Personal Robot 2* in specific and mobile robots in general. This work gives a short introduction to HTN planners and presents an architectural level to parallelize sequential plans from HTN planners and execute the parallelized plan on mobile robots using ROS software. The benefits of the parallel execution compared to the sequential execution of plans are evaluated in terms of time and consumption of resources. Integration of the parallelization architecture level into the global architecture of the RACE Project is discussed and an outlook to future development of the RACE Project in relation to the parallelization architecture is given.

Lasse Einig

Parallel Plan Execution on a Mobile Robot

A Resource Based Approach

Lasse Einig

Abstract

Deutsch

Zeit ist nicht nur eine wertvolle Ressource, wenn es sich um die Benutzungsdauer von Ressourcen eines Roboters bei der Ausführung von Aufgaben dreht, sondern ebenfalls in den Entwicklungszyklen in der Forschung und damit der Ressource Mensch. Aus dem Master Projekt „Intelligente Robotik“ gewonnene Erfahrungen und Einblicke in das von der EU finanzierte Projekt RACE motivierten zur Optimierung von Zeit- und Ressourceneffizienz auf dem *Personal Robot 2*, bzw. mobilen Robotern im Allgemeinen. Diese Arbeit gibt eine kurze Einführung in das Planen mit HTN Planern und präsentiert ein Architekturlevel zur Parallelisierung von mit HTN Planern erstellten, sequenziellen Plänen sowie in die Ausführung dieser sequenziellen Pläne auf mobilen Robotern auf der Basis der ROS software. Die Verbesserung der parallelen Ausführung im Vergleich zur sequenziellen Ausführung von Plänen wird ausgewertet in Bezug auf Zeit und Ressourcenverbrauch. Die Integration der Parallelisierungsarchitektur in die globale Architektur des RACE Projekts wird erläutert und ein Ausblick der zukünftigen Entwicklung des RACE Projekts im Zusammenhang mit der Parallelisierungsarchitektur gegeben.

Lasse Einig

Parallel Plan Execution on a Mobile Robot

A Resource Based Approach

Lasse Einig

Contents

Nomenclature

API	Application Programming Interface
DOF	Degrees of Freedom
HTN	Hierarchical Task Network
PR2	Personal Robot 2
RACE	Robustness by Autonomous Competence Enhancement
ROS	Robot Operating System
SHOP	Simple Hierarchical Ordered Planner
SIADEX	Assisted Design of Forest Fighting Plans by means of Artifical Intelligence Techniques
SMACH	State MACHine
TAMS	Technical Aspects of Multimodal Systems

List of Figures

Listings

1. Introduction

1.1. Motivation

The master project "Intelligent Robotics" taking place at the Group TAMS, Departement Informatics of the University of Hamburg, offers students valuable insights to developing high-level applications for robots. The platform for development and experiments in use is the *Personal Robot 2* (PR2) developed by Willow Garage. Group TAMS coordinates the EU-funded project RACE [RACE Project 2012], developing a robot able of navigating in and manipulating unknown environments. Current work focuses on Pick-and-Place Operations in catering environments. While observing development on the robot during the project, the delay of the development process due to time consuming test runs could be perceived. Considering the target of cooperating with humans or rather serving humans in a gastronomy surrounding time is also a valuable resource affecting customer satisfaction as well as economic efficiency for the owner. Another view is the environment benefiting from low power consumption by more efficiently executed assignments. Consideration of these observations inspired the scope for this Bachelor thesis.

1.2. Target Objective

The aim of this work is to optimize the plan execution on the mobile service robot PR2 by parallel execution of plans.

In earlier robotics reduced complexity of tasks, like moving a robot or executing simple instructions, led to the development of planners, which generat sequential plans. Still one of the most used is SHOP2 [Nau et al. 2001], a very effective HTN planner. In recent years, the variety of challenges that modern robots face has greatly increased, thus making plans more complex. Current applications in robotics usually combine movement, visual sensoring and object manipulation. The execution of tasks in sequential plans leaves resources unused and increases execution time. Generating parallel plans or executing sequential plans in parallel offers optimized

resource utilization leading to decreased runtime when conducting complex assignments. Not only the obvious efficieny bonus for practical application but also the massively decreasing development period inspire this work. Working on assignments for the robot often forces the developers to run time-consuming test cycles in between optimizing task execution or debugging. Reducing this time effort speeds up the development process.

1.3. Outline

In this work the first step will be the analysis of the parallelization capabilities during the execution of two typical scenarios. Resource idle times will be highlighted, safety and interferences with other robot actions will be considered. In the following, the type of parallelization to be used will be evaluated regarding feasibility and time constraints. The following three methods for parallelization will be focussed: (1) Implementing a new planner or altering an existing planner to allow generation of parallel plans; (2) Interpreting sequential plans in a task coordination layer for parallel execution; (3) Using SMACH State Machines and the *Concurrency* container for optimized execution of tasks. The last step is a practical implementation attempt to execute the scenarios in parallel, which will then be evaluated regarding time save-up and effictivity.

This work will not consider dynamic adaption to occuring disturbances, although this is already an important part of robotic planning. While dynamic adaption might be part of successive works this work will put the main emphasis on time-optimization through parallel execution. This will at first be approached by finding a parallel task execution order by hand, later by adding constraints to current implementation or introducing new architectures to enable the planning level of the robot to parallelize the task execution. Later works motivated by the experiences made might lead to a generalized architecture parallelizing unknown task orders without human intervention needed resulting in improvement of utilization of the robots resources.

2. State of the Art

2.1. Parallel Task Execution in Robotics

Though parallel task execution yields huge space for optimization, only few works in robotics approach this field. Existing attempts can mostly be seperated into two categories: (1) Developing and implementing a planner capable of generating parallel plans and (2) Interpreting generated sequential plans and executing the (sub-)tasks of the plan in parallel.

Most approaches to optimize plan generating in robotics either focus on optimizing a planner for special environments or generating plans in parallel on multi-core architectures [Jacobs et al. 2012][Devaurs et al. 2011]. The few efforts on generating parallel plans did not focus on robotics, but computer aided scheduling in general. The most promising planner is SIADEX which has been extended to include temporal knowledge [Castillo et al. 2006]. The main application area of SIADEX is forest fire fighting planning for the Regional Ministry of the Environment of Andalusia. Though it does not feature robotic applications the similarities of SIADEX and SHOP [Nau et al. 1999] and thus SHOP2 reveal the potential of HTN planners. SIADEX temporal logic enables it to handle time constraints such as earliest starting time, finishing deadlines and time synchronisation. The most interesting enhancement for this work is that of the qualitative ordering. It allows tasks to be decomposed to different types of qualitative ordered subtasks:

- A sequence is a set of subtasks that is to be executed in the exact order in which is has been specified.
- Unordered subtasks may be carried out in any possible order, thus allowing them to be executed in parallel.
- A permutation allows the subtasks to be sorted in any order given by the permutations of the subtasks but not in parallel.

All three qualitative orderings are inherited to lower-level subtasks.[Castillo et al. 2006]

In project scheduling, generating parallel plans has always been important due to the number or workers involved in huge projects. Thus planners capable of scheduling tasks in parallel have been existing for decades. An approach by Luh and Lin [1985] uses states of tasks and taskworkers to find a schedule with least idle times.

Lingard and Richards [1998] offer three theoretical improvements to state-based planning, each with a different focus, enabling the planner to generate parallel plans. The first enhancement puts the emphasis on protecting the achievement of goals. The other two strategies address the duration of goals and actions respectively the duration of the effects resulting from goals and actions. Rintanen et al. [2006] and others present semantics for parallel operators in planning as satifiability, improving the work by Kautz and Selman [1996].

Rather than altering existing planners or developing new planners, parallel task execution on service robots is currently implemented by utilizing a three layer architecture. The three layer architecture was first presented by Gat [1998] introducing a reactive feedback layer, a sequencing layer and a planning layer [Gat 1998]. Taipalus and Halme [2009] use this architecture mainly to use their software on different robot platforms by exchanging the reactive feedback layer. The introduced *Action Pool* architecture enables the robot to execute tasks in parallel, since for each resource of the robot there is one *Action Pool*. The *Action Pool* selects the task to be executed. Actions are elements of tasks, more than one task may drop actions into Action pools, thus allowing i.e. Action Pool for manipulators to execute an action from task A and Action Pool for base movement to execute an action from task B. Another three layer architecture has already been presented by Sousa et al. [1996], who use their task coordination and refinement element to execute a list of elementary actions after examining the generated mission plan. Probably the most advanced architecture is part of the ARMAR-III [Asfour et al. 2006] platform developed at the Forschungszentrum Informatik Karlsruhe (FZI). The coordination layer of the ARMAR-III and its predecessors is presented by Asfour et al. [2004]. The ARMAR robot is subdivided into head, left arm, right arm, torso and mobile platform [Asfour et al. 2004]. Tasks are decomposed into subtasks for each subsystem of the robots. These subtasks are translated to primitive actions. More than one primitive action may be sent to the task execution layer, thus allowing the parallel execution. The task execution layer gives direct feedback to the coordination layer providing the opportunity for the task coordination layer to change the order or type of primitive actions being executed resulting in an adapted plan completion [Asfour et al. 2004]. The coordination of task execution is implemented through a *Petri net*. Each subsystem is represented by a set of two places and two transitions. Places are

corresponding to the state of the subsystem (ready/running) and transitions to the state of the task execution (completed/not completed). In the coordination layer architecture, the subsystems are connected with transistions, each connecting different sets of subsystems, allowing multiple subsystems to be changed from 'ready' to 'running', thus executing the respective primitive action.

Although different approaches to parallelization in plan generation respectively plan execution exist, none of them is applicable to the mobile robots running ROS with a JSHOP2 planning system.

2.2. Evaluation Platform

This work will be conducted using the open-source, state of the art operating system ROS (Robot Operating System) [Berger et al. 2012][Quigley et al. 2009][Hassan and Cousins 2012b] developed by Willow Garage. ROS offers interfaces to multiple robot platforms. Developers may wrap their code into *nodes* and communicate with other *nodes*. Projects can be shared with other developers. For evaluation purposes, the PR2, a high-level robot for research and innovation [Hassan and Cousins 2012a], is used. Besides its physical benefits, the PR2 also comes with a complete simulator for experiments. This simulator may even be used without the robot itself, opening the field of robotic innovation to anyone interested.

One of the current scenarios utilizes the Extended Pick-and-Place Demo for the RACE Project [2012] to move an object from a counter in a restaurant to a table in the restaurant. The restaurant is simulated in the laboratory at the University of Hamburg. While executing, the robot tucks its arms in order not to hit his surroundings when navigating to the table. Once at the table, the PR2 locates the object and, after getting into a position enabling it to perform object manipulation on the table and grabbing the object, moves to the other table, performing similiar actions until it is able to place the object on the table. Locating the object and getting into grasping position forces the robot to execute several movements with arms and torso in order to move close enough to the table without pushing it away with his arms.

The scenario is planned by SHOP2 from a planning domain and a planning problem to be solved. In this case JSHOP2 [Ilghami 2006], a Java implementation of SHOP2 is used.

Developed with the PR2, ROS natively supports a state machine called SMACH [Bohren and Cousins 2010], although SMACH also supports other system than ROS and the PR2. Execution plans for assignments can be transferred to SMACH by

building and executing hierarchical state machines. SMACH uses containers consisting of States. Next to the *StateMachine* container and others, SMACH also features a *Concurrence* container capable of executing more than one state at the same time. Bohren et al. [2011] use this to experience optimal runtime for their PR2 fetching drinks from a refridgerator. SMACH also features data sharing between containers and visualization tools.

The *Personal Robot 2* (PR2) build by Willow Garage features two one-DOF grippers, each fixed to a seven-DOF arm, attached to a vertically adjustable torso located on a four-wheeled omni-directional base. On top of the torso is a two-DOF head, holding sets of stereo cameras and a custom mounted ASUS Xtion PRO LIVE sensor. Another camera is located in each forearm and laser scanners are attached to the torso and the base, the former tilting up and down for a 3D scan of the area in front of the robot. Two Quad-Core i7 Xeon processors are used for computing [Hassan and Cousins 2012c].

The ROS system running on the PR2 allows developers to encapsulate their software in ROS *nodes* using ROS *topics* to transfer data from *node* to *node*. ROS also provides hardware abstraction and more features to simplify the developers work. For more information on the Robot Operating System see [Berger et al. 2012].

2.3. JSHOP2

During previous development on the PR2 by the RACE Project [2012], the decision has been made to prefer the use of JSHOP2 [Ilghami 2006] planning Software. Due to limitations in time and complexity, this work will not break away from this decision, but instead use the current plan generation and alter the plan in order to execute tasks in parallel.

JSHOP2 is a *Hierachical Task Network* (HTN) planner. HTN planning is one of the most popular planning strategies. Other common HTN planners are NONLIN [Tate 1977], SIPE-2 [Wilkins 1991], O-PLAN [Currie and Tate 1991] and UMCP [Erol et al. 1994]. Recently, the most efficient HTN planner is SHOP2 [Ilghami 2006] succeeding its predecessor SHOP [Nau et al. 1999]. In general, HTN planners recursively decompose complex tasks to smaller tasks, often referred to as atomic tasks. A complex task may be moving an object from position A to B. An HTN planner might decompose this complex task to smaller tasks like moving to A, picking up the object, moving to B and dropping the object. Task decomposition depends on the interpretation of atomic tasks. Picking up the object may be an atomic task,

but it may also be a complex task decomposing to detecting the object, moving the arm to the object, grasping and moving the arm back into a desired position. Again depending on the interpretation, these tasks may still be complex tasks. The definition of atomic tasks and complex tasks is done in the domain specification. This specification holds a set of atomic actions called operators and a set of complex tasks called methods. A problem description models an environment and presents a complex task to be executed. This task is decomposed using the methods in the planning domain until decomposition leaves nothing but atomic actions. HTN planners require an additional effort to create and maintain the planning domain, but efficiency in generating plans makes up for this effort. Effiency of HTN planners is a result of less unnecessary paths that have to be entered compared to traditional planners.

JSHOP2 is a Java based implementation of the SHOP2 algorithm by Nau et al. [2001]. SHOP2 is a multi-awarded *Hierarchical Task Network* planner that is domain independent. Both SHOP algorithms know the current state of the tasks at every point of the planning process, since the algorithms generate the plans steps in the exact order of the execution. The main difference to its predecessor SHOP is the partial ordering of subtasks. Partial ordering of subtasks already features some task execution optimization.

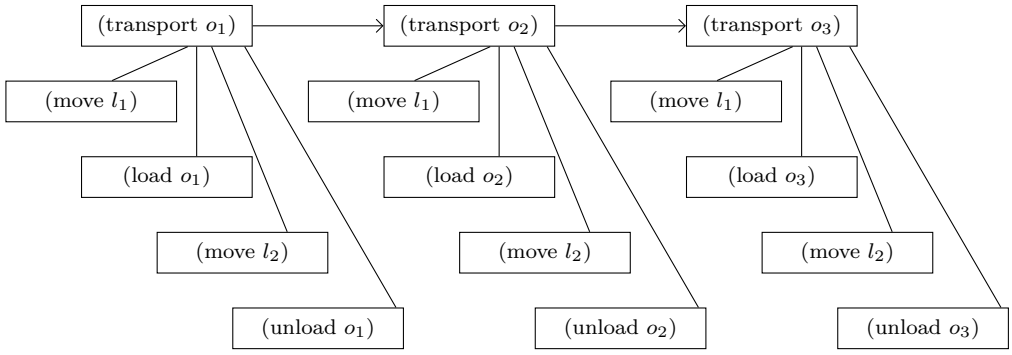

Figure 2.1.: A sample plan for moving three objects generated by SHOP algorithm using total ordering

Figure 2.1 shows a possible plan reduction for moving three objects from location l_1 to location l_2. Total order planning performed by SHOP forces the transporter to move from l_1 to l_2 for each object to be transported. Partial order planning allows interleaving of subtasks. Subtasks of each (transport o_i) must be executed in the correct order ((move l_1) may not be executed after (load o_1)) but each subtask of (transport o_1) may interleave with subtasks of (transport o_2) and (transport o_3).

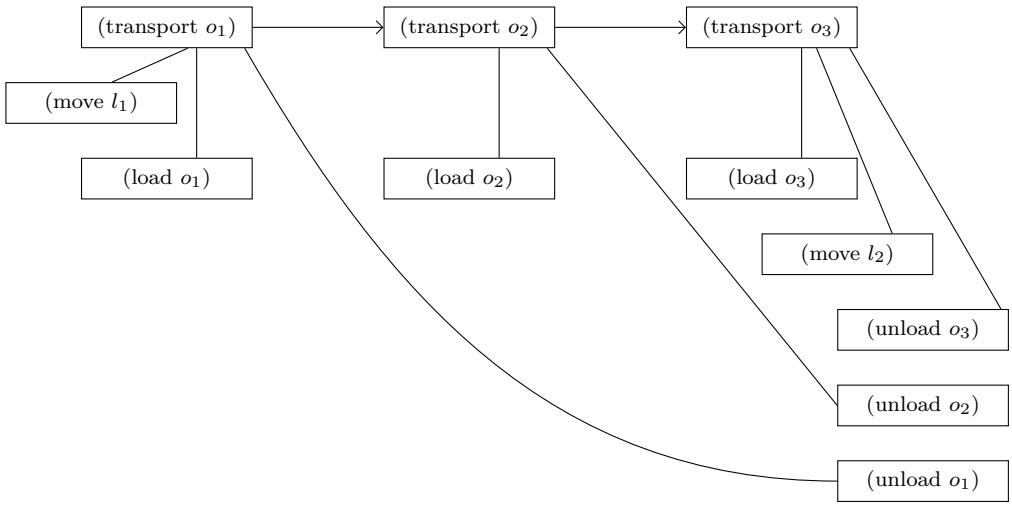

Figure 2.2.: A sample plan for moving three objects generated by SHOP2 algorithm using partial ordering

Figure 2.2 shows a partially ordered example like it could be generated by SHOP2. Due to interleaving subtasks, (move l_1) and (move l_2) subtasks can be executed in a single block and therefore only need to be executed once. Of course this is a simplified example without regarding loading space of the transporter. In JSHOP2 partial ordering is triggered by the keyword *:unordered* which can be placed within a task list. In case of Figure 2.2 the keyword is placed as following:

$$\mathrm{M} = (\text{:unordered } T_1\ T_2\ T_3)$$

where

$$T_i = ((\text{move } l_1)\ (\text{load } o_i)\ (\text{move } l_2)\ (\text{unload } o_i)).$$

Despite its great efficiency and functionality, SHOP2 does not feature a keyword *:parallel* or support the parallel decomposition of methods in any way. In order to generate plans, JSHOP2 needs a domain description and a problem that should be solved. The planning domain description is composed of operators, methods and axioms [Ilghami 2006].

Operators define the execution of tasks. They hold information on requirements to the current state (e. g. posession of objects or knowledge) and the effects of the tasks (e. g. grabbing or dropping an object). Operators also hold an optional cost expression. Listing 2.1 shows the structure of an operator based on moving an object. *!name* corresponds to a primitive task atom, *?arg1* and *?arg2* are

the task arguments. *(on table ?arg1)* is a possible precondition, stating that the *?arg1* is located on a *?table*. The next line is the delete list. Obviously, after grabbing *?arg1* from the *?table*, *?arg1* is no longer on the *?table*. The last required line is the add list. In this case *?arg1* should be placed on top of *?arg2*, therefore *(on ?arg2 ?arg1)* will be true after execution. The *:protection* ensures, that *(on ?arg2 ?arg1)* only becomes true, if *?arg1* is really placed on top of *?arg2*. The add list may be followed by an optional cost operator.

```
(:operator (!name ?arg1 ?arg2)
  (on ?table ?arg1)
  (on ?table ?arg1)
  ((on ?arg2 ?arg1) (:protection (on ?arg2 ?arg1)))
)
```

Listing 2.1: A sample operator in a SHOP planning domain

Methods define decomposition of compound tasks into partially ordered subtasks. Like operators, methods have requirements to the current state. One method may hold more than one set of preconditions and decomposition tasks lists. Depending on the preconditions, a compound task can be decomposed in different ways. A travelling planner for example could decompose a travel depending on the amount of cash available. The result may be a plan to travel by plane or by train. Listing 2.2 shows a quick example of such a method. The methods name is a compound task like travelling to *?arg1*. *plane* and *train* are optional names to help debugging. *(enough-cash ?cash)* and *(not-enough-cash ?cash)* is the precondition for decomposing the task. Depending on the precondition, the compound task name can either bei decomposed to buying a plane or a train ticket and then travelling to *?arg1*. A method may hold an arbitrary amount of decomposition branches.

```
(:method (name ?arg1)
  plane
  (enough-cash ?cash)
  ((!buy-ticket ?plane ?x) (!travel ?arg1))
  train
  (not-enough-cash ?cash)
  ((!buy-ticket ?train ?x) (!travel ?arg1))
)
```

Listing 2.2: A sample method in a SHOP planning domain

Axioms are Horn clauses in a Lisp-like syntax due to SHOP2 implemented in Lisp. They are used to express complex precondition. A precondition to traveling by plane may not only be the amount of cash available but may also depend

on travelling distance and availability of airports. Listing 2.3 shows an example axiom for travelling by plane with the previously listed prerequisits. First the ticket price for travelling to *?x* and the available cash are evaluated and compared, then the distance is checked to be greater than 50 miles. The last preconditions looks for availbale airports near. An axiom may combine preconditions in order to make domain descriptions easier. Preconditions may also be logically arranged.

```
1 (:- (travel-by-plane ?x)
2   ((ticket-price ?x ?t) (cash-available ?c) (call >= ?c ?t)
3     (distance ?x ?d) (call >= ?d 50)
4     (airport-available))
5 )
```

Listing 2.3: A sample axiom in a SHOP planning domain

Although JSHOP2 is a powerful, easily usable planning algorithm, it lacks parallel planning capabilites. The effiency of the SHOP2 algorithm though, allows post-refinement for parallel execution without slowing down planning processes.

2.4. SMACH

SMACH is a python based task-level architecture. The name derives from **S**tate **Mach**ine and is pronounced like the verb "smash"[Bohren 2012]. Although SMACH was built to allow rapid development of robust robot behavior for ROS, it is independent from the operating system but well integrated. This makes SMACH the system of choice for many developers on the PR2 when it comes to creating a complex behavior. Examples include opening doors, plugging into an outlet and fetching drinks from a refrigerator [Bohren and Cousins 2010][Bohren et al. 2011] Unlike usual state machines, the states of which provide a given configuration between the execution, SMACH states are states corresponding to the system performing a task. Each state is an executable task. This concept puts the focus on the execution instead of snap-shots between performing actions. States are connected by their outcomes. The default outcomes for SMACH states are "succeeded", "aborted" and "preemted". Each outcome may be connected to another state, not connected outcomes terminate the state machine.

In SMACH, states also include a small database to hold and pass data to other states, thus states are capable of calculations. Data passing is done by the input and output keys of states. Input keys are similiar to parameters in method heads, output

keys are the parameters passed on the method call. In this way, the state outcomes may be viewed as the methods return value.

SMACH uses *States* and *Containers* to construct complex hierarchical state machines. On the bottom level, states are included in a container. This container may be treated just like a state in a higher level container. This higher level container may consist of containers and states or only containers. Therefore, containers need to have defined outcomes as well as states. SMACH already offers a sufficient range of containers and states, but custom states and containers may be implemented if needed. The SMACH default states are:

Generic State is the base state, it may be used to create own customized states.

CBState stands for **C**all **B**ack **State** and performs a callback to a given function. The CBState will pass the states userdata as well as the given arguments to the function.

SimpleActionState is part of the smach_ros library. It is a state made for executing *actionlib* actions. *actionlib* is an interface for atomic tasks on ROS robots like moving, manipulating, object detection and many more. The interface uses a client-server structure to execute and interrupt the actions. While these actions could be called from a custom state derived from the generic state, the *SimpleActionState* offers an easily usable method. The *SimpleActionState* always has the three default outcomes, which may then be mapped to successive states. The SimpleActionState itself offers four possibilities to pass the goal to the *actionlib* server. (1) Pass an empty goal; (2) Pass a fixed goal; (3) Get the goal from the states userdata and pass it; (4) Pass the return value of a function. Before storing the result of the *actionlib* action performed in the state to the userdata, a function callback may be used to interpret the result.

ServiceState is similiar to the SimpleActionState. Instead of executing *actionlib* actions, the ServiceState may call any service. Instead of passing a goal, the ServiceState will send a request to the service and will receive a response instead of a result. Other than that the usage and the four options to send the request are the same as the SimpleActionState.

MonitorState is still in progress. It is supposed to monitor a topic and block the state machine while the desired message has not been published to monitored topic.

ConditionState calls a condition function to check the result. If the condition is false and the maximum number of allowed checks is reached, it will return

false. If the maximum is not reached it will block execution until the condition becomes true.

SMACH also features four default containers each with a different focus on the task execution order:

StateMachine is the basic container. States may be added to the container and connected by their outcomes. Alike all other containers, the outcomes have to be specified on construction and the optional input and output keys need to be defined. Each container has a userdata that is available to all states and containers inside. The static method *add()* appends states or containers once the container has been opened. Containers must be opened by using the *open()* and *close()* functions or Pythons with keyword.

Concurrence is the most interesting Container for this work and the main reason why SMACH was chosen to implement parallelization of task execution. The *Concurrence* does not connect the contained states by outcomes, instead all contained states are executed in parallel. Once all states have terminated, their outcomes are mapped to a container outcome. *Concurrence* also offers two callbacks to gain further control of the parallel execution. *child_termination_cb()* is called whenever a state of the *Concurrence* terminates and the outcome may be used e. g. to terminate the other, still running, states in case the desired outcome was not archieved. *outcome_cb()* is called when all states have terminated and may be used to alter the *Concurrence* outcome.

Sequence is a container to run a simple, consecutive task execution. Alike the *Concurrence*, the outcomes of the states do not have to be mapped to other states by hand. A *connector_outcome* is specified. Each state is connected to the successive state with this *connector_outcome*. The states are executed in the order in which they are added to the *Sequence*. Since *Sequence* is a modified StateMachine container, it is still possible to connect the outcomes to other states, but the *connector_outcome* will override this connection.

Iterator works like a loop for SMACH. The container will execute the single contained state while the result is equal to the *loop_outcomes*. In order to construct a more complex loop, other containers may be wrapped inside the Iterator.

Since parallel plans consist of nested *Sequences* and *Concurrences*, the SMACH containers and their nested structures offer huge potential for parallel execution of refined sequential plans. The integration into ROS with *SimpleActionStates* and

custom states inherited from the *Generic State*, presents a good interface to interpret parallel plans into execution of atomic actions.

3. Analysis

In order to find a suitable parallel execution order for the sequential plan, this work will analyze the atomic actions and assign resources of the robot to the actions. A parallel execution may not have more than one task in a parallel order that uses the same resource. Tasks that use different resources may be parallelized. In addition, some security constraints may be added to ensure a secure execution order. Resources of the PR2 are:

- Head (including most sensors like vision and depth cameras)
- Torso (moving up and down)
- Base (navigating in the environment including the 2D base laser)
- Right arm (manipulating the environment)
- Left arm (manipulating the environment)

Atomic actions in the JSHOP2 planning domain are operators, while methods are used to decompose complex tasks.

3.1. RACE Scenario: Serving beverages

In this scenario, the robot is supposed to get a beverage from a counter and serve it to a table. The scenario is located in a laboratory restaurant environment at group TAMS, University of Hamburg.

3.1.1. Operators and Resources

The planning domain for the “Serving beverages” scenario holds a total of ten operators. Four of these operators move one arm and only differ in the prerequisites and the arm which is to be moved. One operator for each arm assumes that both arms are in a tucked position, thus leaving the arm that is not moved by the operator in the tucked position after the operator has finished. The other operator for each arm

assumes that only the arm to be moved is tucked and the other arm is already in some kind of manipulation position. After completion of this operator, both arms will be in an untucked state. These arm operators are *!move_arm_to_side left_arm* respectivly *!move_arm_to_side right_arm*. Since these operators only use one of the robots arms, the resource used by this action is the arm (*left_arm*, respectivly *right_arm*).

The torso resource is used only by the *!move_torso ?position* operator which is used to move the torso to a *torso_up_position* or a *torso_down_position*. *!move_base ?to* and *!move_base_blind ?to* use the base resource to navigate to tables or counters in the room. While *!move_base* uses collision avoidance to navigate in a room with obstacles, *!move_base_blind* is only used to navigate from a pre-manipulation pose close to the table or counter in order to get into the required grasping range.

The last two operators (*!pick_up_object ?object ?arm* and *!place_object ?object ?arm ?to*) control picking up and putting down the objects to be served. They do not only require the arm resource to grab the object, but also the head resource to detect the object with vision and depth sensors.

3.1.2. Methods and Decomposition

A typical problem for the planning domain locates the robot in the same room as a counter and a table. Three objects, including one clean coffee cup, are located on the counter. Both arms are in an untucked state, the torso is in an upper position. The objective for the robot is to serve the coffee cup to the table (*serve_cup table_1*). *serve_cup ?to* is decomposed to *move_object ?object ?to* after checking for the correct object type (coffee cup) and if the object is in a clean state. *move_object ?object ?to* can be decomposed to *move_object ?object ?from ?to*, if the precondition: *?object* is located on a counter or table (*?from*) is fulfilled, or into *put_object ?object ?to*, if the precondition robot is holding on to *?object* is fulfilled. At this point, the object is still located on the counter, so the first decomposition is applied. *move_object ?object ?from ?to* first decomposes to *get_object ?object ?from*, then again to *move_object ?object ?to*. After execution of *get_object ?object ?from*, the robot should hold on to the object and therefore decompose *move_object ?object ?to* to *put_object ?object ?to*. Both *get_object ?object ?from* and *put_object ?object ?to* decompose similarly.

In case the robot is not already close enough to the table or the counter to get the object from, or put the object down to, the robot will first drive to a pre-manipulation

position (*drive_robot ?fromarea* respectively *drive_robot ?toarea*), then retry the decomposition. The *drive_robot* method decomposes, depending on the preconditions, to the operators *!tuck_arms*, *!move_torso* and *!move_base*. Tucking arms and moving down the torso will only be applied, if the robot is not holding an object.

If the robot is close enough to the manipulation location, *get_object ?object ?from* and *put_object ?object ?to* decompose to manipulation methods *grasp_object ?object* and *put_down ?object ?to*. Both manipulation methods decompose to assuming the manipulation pose (coming from the pre-manipulation pose), performing the manipulation and leaving the manipulation pose. Assuming the manipulation pose decomposes to the operators *!move_torso torso_up_position*, *!move_arm_to_side [left_arm, right_arm]* and *!move_base_blind* while leaving the manipulation pose decomposes only to *!move_base_blind*. The manipulation operator performed during assuming and leaving the manipulation pose is *!pick_up_object ?object [left_arm, right_arm]* respectively *!place_object ?object [left_arm, right_arm] ?to*.

3.1.3. Plan and Parallelization

The planning process usually comes up with four different plans, of which the first plan is executed. A typical sequential plan resulting from the planning process is:

1. *!tuck_arms both_arms*
2. *!move_torso torso_down_position*
3. *!move_base counter_1_pre_manipulation_pose*
4. *!move_torso torso_up_position*
5. *!move_arm_to_side left_arm*
6. *!move_base_blind counter_1_manipulation_pose*
7. *!pick_up_object coffee_cup_1 left_arm*
8. *!move_base_blind counter_1_pre_manipulation_pose*
9. *!move_base table_1_pre_manipulation_pose*
10. *!move_base_blind table_1_manipulation_pose*
11. *!place_object coffee_cup_1 left_arm table_1*
12. *!move_base_blind table_1_pre_manipulation_pose*

This plan is devided in two sections as this work will later show. At first, the latter section is reviewed. It starts at item *6*. The movement action with no collision avoidance cannot be paralellized with moving the arm or picking up an object. Moving the arm to the side while moving closer to the table or the counter with no collision avoidance exessively increases the risk of damaging the robot or its environment, including humans sitting at the table. The same is true of manipulating an object increased by the fact, that the robot may not be able to grab the object because it is out of range for the arm or increases probability for singularities in arm planning. Therefore item *6* offers a good breach dividing the plan into two segments. The following tasks *7 – 12* alternately have a *!move_base_blind* task and another task. Again, moving tasks with no collision avoidance are unsafe to parallelize. In addition, the task *7*, picking up the object must be complete before backing off from the table or counter to successfully complete the manipulation. Item *8* must be finished before moving with object detection in task *9*, since the object detection that close to the table might force the robot to shut down since no collision free path can be found. Tasks *8*, *9* and *10* use the same resource (moving with the base) and may not be executed in a parallel order as well. Again, item *11* may not be executed in parallel with *10* or *12*, because either the area to place the object at is not in range or the object has not been put down yet.

For the second segment of the plan, this only leaves a sequential task order to be executed, shown in figure 3.1. Although the sequential order of the tasks *8 – 10* will result from the resource collision, a constraint must be introduced to avoid the parallel ordering of the other tasks with the “blind” navigation.

The former section of the sequential plan offers parellelization potential. This work will review two different approaches. The first approach is a more secure approach as it puts a constraint on the task *3* to avoid driving through the environment with untucked arms and a torso in upper position. Untucked arms bear a slightly higher risk of collision and increase collision detection complexity. The torso in an upper position raises the center of mass of the robot, thus reducing stability of acceleration in the horizontal plane. The second approach will ignore these risks, as the robot is still able to decelerate without tilting over if the torso is in an upper position. Test runs in the *gazebo* simulation environment and the practical experiments in the test environment at group TAMS will point out difficulties with collision detection and avoidance when executing the parallel approaches.

The collision avoidance will be examined in a practical experiment (chapter 5) in the test environment at group TAMS and the *gazebo* simulation environment.

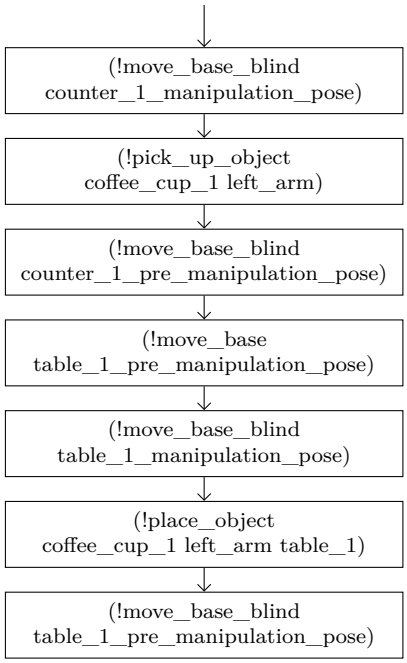

Figure 3.1.: Sequential section of *Serving Beverages* scenario. Due to security threats by movements without collision detection, there are no parallelization capabilities in this section.

For both approaches, tasks *1*, *2*, *4* and *5* may be executed in parallel, regarding:

- Tasks *1* and *5* use the same resource, therefore *1* has to be finished before *5*.
- Tasks *2* and *4* use the same resource, therefore *2* has to be finished before *4*.
- Task *3* may not be started after *4* or *5*.

These constraints add up to the parallel execution orders shown in figure 3.2 and 3.3. While figure 3.2 shows a more secure approach by adding a security constraint to the *!move_base counter_1_pre_manipulation_pose* operator, the execution order in figure 3.3 increases parallel potential by allowing arm and torso movements while moving the base. In figure 3.2, the *!move_base* operator is assigned the *Base* resource, as well as the *Torso*, *Left arm* and *Right arm* resource. Therefore, the operator *!move_base* must wait for the completion of *!tuck_arms both_arms* and *!move_torso torso_down_position* operators and it must be finished before *!move_torso torso_up_position* and *!move_arm_to_side left_arm* operators may be executed. In figure 3.3, the *!move_base* operator is assigned only the *Base* resource.

Expectation is, that the approach without security constraint is most effective, if the time that task *3* needs is longer than the time of the parallel execution of tasks *1* and

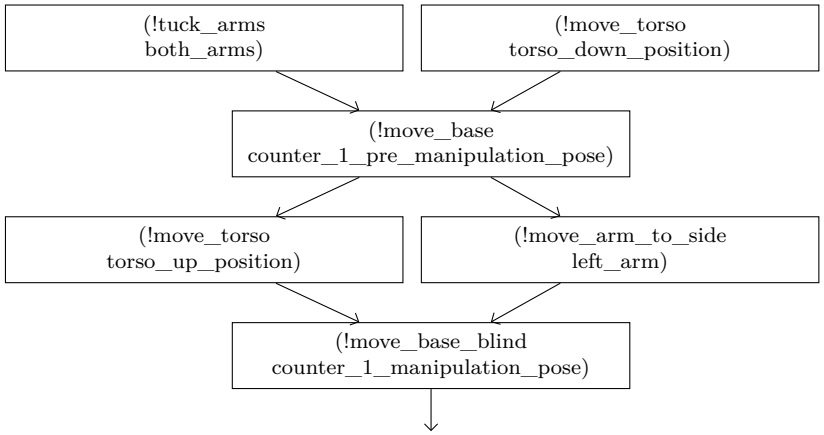

Figure 3.2.: Parallel section of the *Serving Beverages* scenario. A security constraint is added to *!move_base counter_1_pre_manipulation_pose* forcing the robot to finish arm and torso movements before moving the base to the new position and waiting for the base movement to finish before the arms and torso may operate again. This prevents the robot from hitting objects or humans in the environment, respectively increases the tilt stability while moving due to a lower center of mass.

2 in addition with the time of *4* and *5*. Up to a certain distance from the robots' start position from the counter, the time benefit will increase while the distance increases. For the approach with security constraint, the time benefit should stay constant.

3.2. Theoretic Scenario: Loading a dishwasher

In this scenario, the robot is supposed to load a set of plates and cups into the upper basket of a dishwasher. Cups are placed on the right side, plates on the left side. This scenario yet lacks practical application and will therefore be analyzed in theory.

3.2.1. Operators and Resources

The planning domain for the theoretical scenario *Loading Dishwaser* holds a set of operators similar to the "Serving beverages" domain. Both domains hold *!tuck_arms*, *!move_torso*, *!move_base* and *!move_base_blind* operators as they are essential for navigating to and from manipulation poses. The domain for "Loading a dishwasher" also holds the same set of *!move_arm_to_side* operators. These operators require the same resources as in chapter 3.1.1.

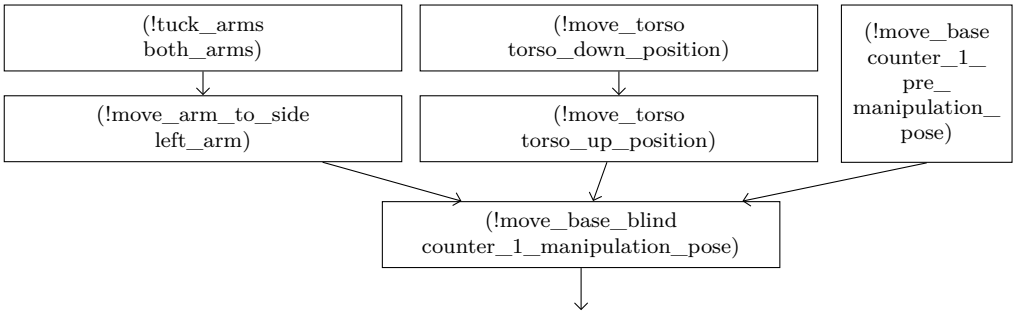

Figure 3.3.: Parallel section of the *Serving Beverages* scenario. No security constraint is added. Possible threat is posed to the environment but parallel capabilities are increased.

Picking up plates and cups and loading them to the dishwasher requires four new operators: (1) *pick_up_plate*, (2) *pick_up_cup*, (3) *place_plate* and (4) *place_cup*. (1) and (2) obviously require the robot to detect the desired object and use an arm to grab the object. Since cups are located on the right of the dishwasher basket, (2) uses the right arm and (1) uses the left arm. Operators (3) and (4) place the object into the dishwasher. The dishwasher and its basket have predefined measurements and the robots tracks the amount of objects in the dishwasher while loading. Predefined positions in the basket allow the robot to place objects in the dishwasher "blindly". That is, the robot does not use its sensors to detect valid positions in the dishwasher. (1) requires the *Head* resource to detect the objects and the *Left arm* resource to grab the plate. (2) also requires the *Head* resource for detection, but the *Right arm* resource to grab the cup. (3) requires only the *Left arm* resource to place the object, while (4) only requires the *Right arm* resource to place the object.

3.2.2. Methods and Decomposition

The theorectical problem for the planning domain locates the robot in the hallway near the door to the kitchen with the dishwasher. Four plates and five cups are placed near the dishwasher in a position enabling the robot to grab the objects and place them into the dishwasher without moving its base. This position may even be a tray mounted to the robot. The objective for the robot is to place all objects of type plate or cup in manipulation range into the dishwasher (*load_dishwasher*). While the robot is not in its manipulation pose, *load_dishwasher* decomposes to *drive_robot ?position*. Similar to *Serving beverages* (chapter 3.1.2), *drive_robot* decomposes to *!tuck_arms both_arms*, *!move_torso torso_down_position* and *!move_base pre_manipulation_pose*.

Now that the robot is in its pre-manipulation position, *load_dishwasher* decomposes *load ?object* until no more objects are left. The *?object* type is determined by the amount of objects of the certain type left to load. In this example, the planner will decide to start loading a cup and continue loading plates and cups in an alternating order. *load ?object* will decompose to *!pick_up_plate* and *!place_plate*, respectively *!pick_up_cup* and *!place_cup*. If the robot is not yet in the manipulation position before the pick-up operator, *load ?object* will decompose to *!move_torso torso_manipulation_position*, *!move_arm_to_side right_arm*, *!move_arm_to_side left_arm* and *!move_base_blind manipulation_pose*.

Once no more objects are left to place in the dishwasher, *load_dishwasher* will decompose to *!move_base_blind pre_manipulation_pose*, *!tuck_arms both_arms* as well as *!move_torso torso_down_position* and the planning process is finished.

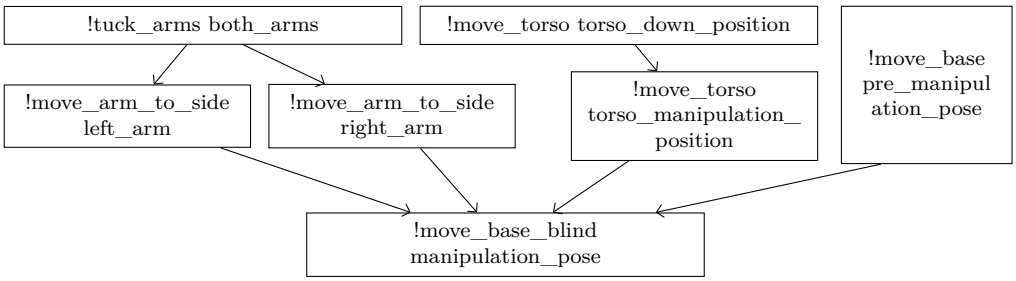

Figure 3.4.: Parallel section of the *Loading Dishwasher* scenario. No security constraint is added. Possible threat is posed to the environment but parallel capabilities are increased.

3.2.3. Plan and Parallelization

Since no planning domain exists yet, an assumed sequential execution order is used for parallelization. A typical sequential plan resulting from the planning process might be:

1. *!tuck_arms both_arms*
2. *!move_torso torso_down_position*
3. *!move_base pre_manipulation_pose*
4. *!move_torso torso_manipulation_position*
5. *!move_arm_to_side left_arm*
6. *!move_arm_to_side right_arm*

7. *!move_base_blind manipulation_pose*
8. *!pick_up_cup*
9. *!place_cup*

   ...

   10. – 25. Pick and place operators are repeated four times

   !pick_up_plate

   !place_plate

   !pick_up_cup

   !place_cup

   ...

26. *!move_base_blind pre_manipulation_pose*
27. *!tuck_arms both_arms*
28. *!move_torso torso_down_position*

Again, the plan is divided into two sections. Other than the *Serving Beverages* scenario, both sections of the *Loading Dishwasher* scenario hold parallelization capabilities. Similar to the *Serving Beverages* scenario, the first section offers two different approaches to parallel execution (Chapter 3.1.3). Like the first scenario, the problematic task is moving the base to the pre-manipulation pose, which is enclosed in tucking and untucking arms, as well as moving the torso up and down. Both approaches are described in this chapter and the evaluation of the *Serving Beverages* scenario in chapter 5 will give information on executability of the approaches.

For reasons described in chapter 3.1.3, task 7 marks the border between the two sections. The first section of *Loading Dishwasher* is similar to the first section of *Serving Beverages*. The only difference is, instead of moving one arm to the side like in *Serving Beverages*, *Loading Dishwasher* requires both arms to be moved to the side. Since the second arm movement does not conflict with any other tasks in the execution order but with *tuck_arms both_arms*, which has to be executed both before *move_arm_to_side left_arm* and *move_arm_to_side right_arm*, the second arm movement can simply be inserted to into the parallel container (Figures 3.4 and 3.5).

Other than this, the first section is the same as the *Serving Beverages* scenario with the same restrictions and will therefore be skipped. The latter part of the

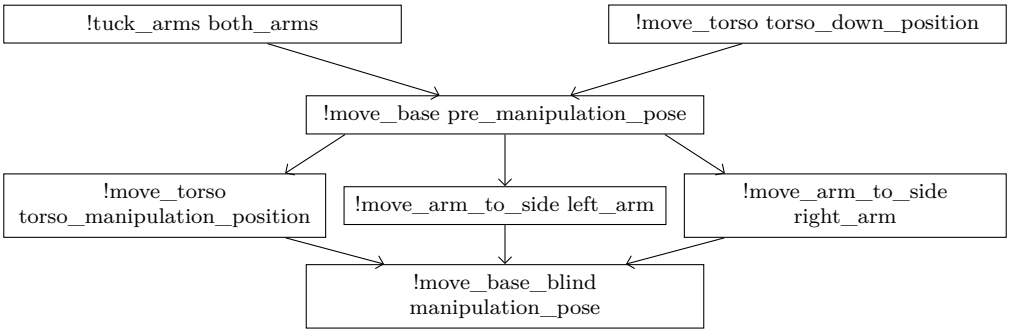

Figure 3.5.: Parallel section of the *Loading Dishwasher* scenario. A security constraint is added to *!move_base pre_manipulation_pose* forcing the robot to finish arm and torso movements before moving the base to the new position and waiting for the base movement to finish before the arms and torso may operate again. This prevents the robot from hitting objects or humans in the environment, respectively increases the tilt stability while moving due to a lower center of mass.

plan execution though is completely different from the *Serving Beverages* scenario. The problematic task from the *Serving Beverages* scenario, the *!move_base_blind* action is only found twice in the latter part of the *Loading Dishwasher* scenario. The first appearance is the section dividing task and has no effect on the parallelization capabilities of this section, only for the overall execution order. The main part of this section is made up by repetitive pick and place actions. Two different types of objects are recognized, picked up and placed in the dishwasher basket. The object types are *cup* and *plate*. Each object has its own pick and place actions. Picking up any of the objects requires object detection and therefore requires the *Head* resource with the cameras and depth sensors. The *!pick_up_cup* action additionally requires the *Right arm* and the *!pick_up_plate* action requires the *Left arm*. The arm resources are the same for the place actions, but the head is not required. This leads to two different types of dependencies in the parallel execution order, as marked in figure 3.6. The first type of dependency is the *Head* resource. In figure 3.6 the *Head* resource is represented by a dotted line. The second type of dependency is the *Arm* resource, which is divided into *Right arm* (represented by a thick dashed line) and *Left arm* (represented by a thin dashed line). Figure 3.6 shows a thick dashed path from *!pick_up_cup cup1* to *!place_cup cup5* on the left side and a thin dashed path from *!pick_up_plate plate1* to *!place_plate plate4* on the right side. These paths are "synchronized" by the *Head* resource (showing a zigzagged path between left and right path).

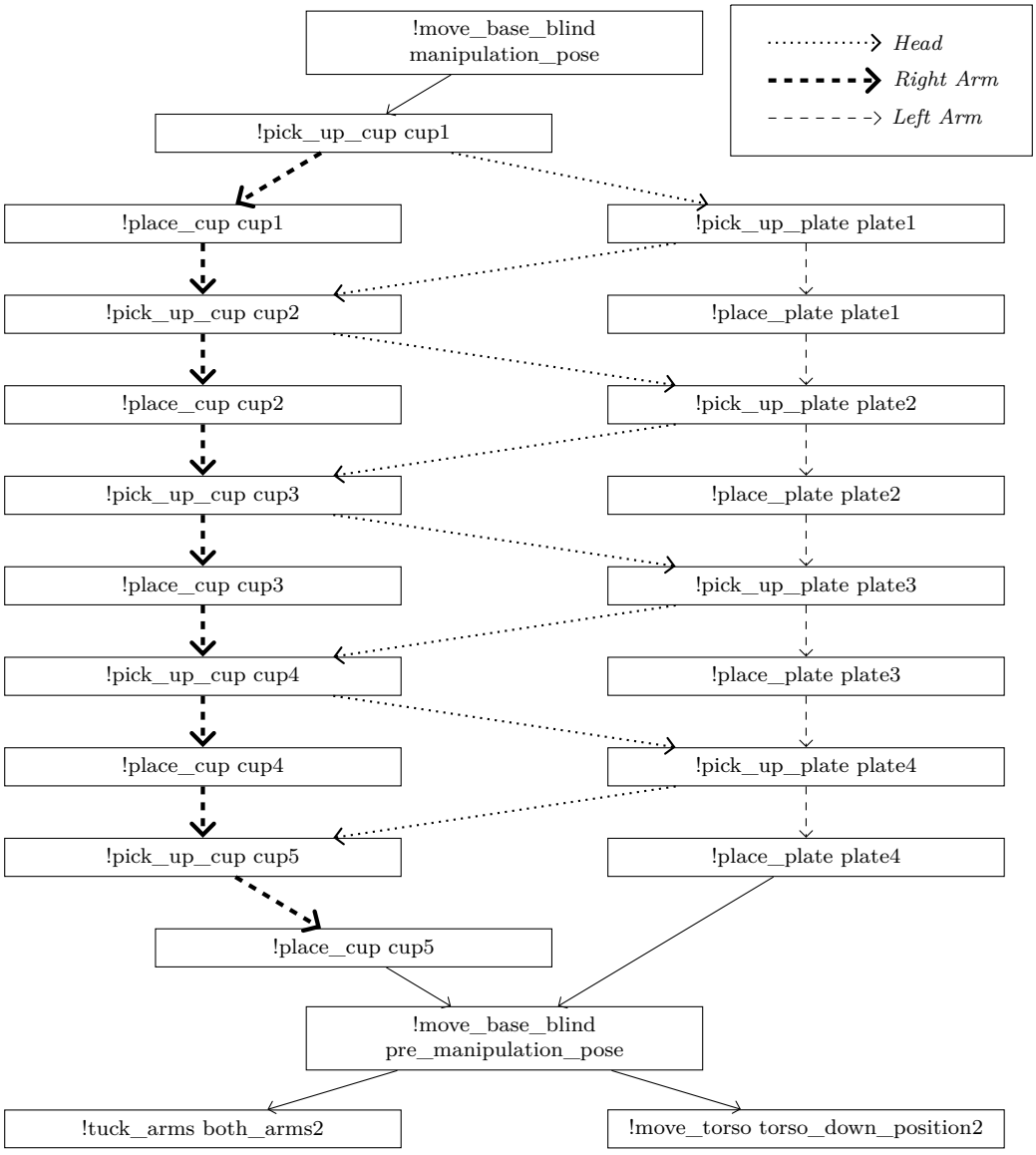

Figure 3.6.: Latter section of *Loading Dishwasher* scenario. Repetive Execution of loading cups and plates with different arms offers huge parallelization capabilities, nearly cutting execution time in half. Dependencies from *Head*, *Left arm* and *Right arm* resources in the repetitive part are distinguished between by different line styles shown in the legend.

After finishing with the pick and place actions, the second *!move_base_blind* action forces the parallel execution to a break until continuing with tucking arms and moving torso in parallel again (like the first section).

Like in chapter 3.1.3, expectation is for the approach without security constraints (Figure 3.4) to be more effective depending on the distance to be travelled. The main

benefit though should be gained in the latter section while executing the pick and place actions. Obviously, best results should be achieved if the amount of cups and plates only differ by a maximum of one item. Since there is a little sequential execution left (picking and placing the first item and both *!move_base_blind* actions) larger amounts of objects to be placed in the dishwasher basket should increase percental benefit compared to the sequential plan execution.

This chapter analyzed the parallelization potential of the two scenarios dealt with in this work. Optimal parallelized execution orders by human have been presented, resources, dependencies and possible security threats have been highlighted. After examining the implementation for automated parallelization, the results will be evaluated in chapter 5 with the specifications made in this chapter.

4. Implementation

4.1. Basic structure

This chapter will give a quick look into the implementation of the parallelization algorithm and into the creation and execution of the SMACH state machine. SMACH is currently only available with a Python API, so Python is used. The implementation has three overall levels (algorithm 1). The first level is the JSHOP2 planner. Running the JSHOP2 planner and getting the planning result is trivial to this work, hence it will not be explained. Still, listing A.1 shows the basic routine to call the planner. The second level parses the plan returned by the JSHOP2 planner and performs the parallelization operations. The third level creates a SMACH state machine from the parallelized plan and executes it.

Algorithm 1: Main structure

Data: planningGoal

```
1 begin
2     plan ← GetJSHOP2Plan(planningGoal);
3     parallelPlan ← ParallelizePlan(plan);
4     ExecuteStateMachine(parallelPlan);
```

4.2. Parallelization algorithm

The parallelization algorithm is devided into two parts. The first part is the *ParallelizePlan()* algorithm (algorithm 2). It parses and prepares the JSHOP2 plan. Afterwards, a recursive algorithm (*ParallelizePlanRecursively()*, algorithm 3) runs through the prepared plan arranging each task in its parallel order.

4.2.1. Parsing and Preparation

After parsing the JSHOP2 formatted list, the operators are matched with a list of resources (Listing A.2) and an object of the type *Action* is created for each operator.

Algorithm 2: ParallelizePlan()

input: JSHOP2 representation of the sequential plan

```
1  plan ← ParsePlanToList;
2  begin ResourceAllocator(plan)
       // match operators (op) in plan with resources (res) definition
3      r_list ← [[op_1,[res_1,...,res_n]],...,[op_n,[res_1,...,res_n]]];
       // create Action object from each entry in r_list
4      a_list ← [Action_1(op_1,id,res_1),...,Action_n(op_n,id,res_n)];
5      begin ParallelizationPreparation(a_list)
           // create links between op's from res's
6          l_list ← Link(a_list);
           // remove duplicates from l_list
7          l_list ← Unique(l_list);
           // generate dependencies (dep) from l_list, sorted by
              Action.ID
8          foreach Action i in a_list do i.dep ← Sorted(ID,[dep_i,...,dep_n]);
           // generate successors (suc) from l_list
9          foreach Action i in a_list do i.suc ← [suc_i,...,suc_n];
           // remove redundant links
10         foreach Action i in a_list do i.dep ← CheckDep(i);
11         foreach Action i in a_list do i.suc ← CheckSuc(i);
           // find start Actions from a_list
12         s_list ← {Action ∈ a_list|len(Action.dep)=0};
13         actions_left ← {Action ∈ a_list|len(Action.dep)!=0};
14         switch len(s_list) do
15             case 0
                   // dead graph with no start Actions
16                 raise Exception;
17             case 1
                   // one start Action
                   // begin parallelization with sequential list (SList)
18                 start ← SList(s_list);
                   // repeat until actions_left is empty
19                 ParallelizePlanRecursively(start, actions_left);
20             otherwise
                   // multiple start Actions
                   // begin parallelization with parallel list (PList)
21                 start ← PList(s_list);
                   // repeat until actions_left is empty
22                 ParallelizePlanRecursively(start, actions_left);
```

This list of operators is the only input, besides the sequential JSHOP2 plan, that is required for the parallelization. The list of operators, currently implemented as a Python file, holding a list of the operators, each holding a list of resources, is written by human. New operators for other planning domains may be added by appending an item to the list, which matches the following syntax:

[*!operator_name arg_1 arg_n*, ['RESOURCE1', ..., 'RESOURCEn']]

Resources currently implemented in the operators list are (see also chapter 3):

- *H* for the head
- *T* for the torso
- *B* for the base, respectively planar movement
- *RA* for the right arm
- *LA* for the left arm

The decisions made by the algorithm on which tasks may be executed in parallel, is solely based on the required resources by the task.

The *Action* created after matching the resources, holds the operator as *name*, a unique *id* and the list of resources required by the operator, which will now be referred to as *Action*. The class *Action* (Listing A.3) also holds fields for dependencies (*depends*) and successors (*next*) which will be used to link the nodes, as well as *visited_from* and *marked* required for creating links and removing redundant links (algorithm 2, lines 1 – 4). To create the links between the *Actions*, the list of *Actions* is treated like the list of *Vertices* in graph theory. An algorithm (listing A.4) similar to algorithms used to create graphs from dependencies is used to generate the list of *Edges* connecting the *Vertices* based on the *Vertices* requiring the same resource (Listing A.4). Unfortunately, this algorithm creates duplicate links, so these need to be removed (algorithm 2, line 7). The graph algorithm creates an *Edge* from each *Vertice* that uses the same resource. For the approach shown in figure 3.2, the algorithm creates a link from *!tuck_arms both_arms* to *!move_base counter_1_pre_manipulation_pose*, to *!move_arm_to_side left_arm* and *!move_base_blind counter_1_manipulation_pose*. Since *!move_arm_to_side left_arm* depends on *!move_base counter_1_pre_manipulation_pose*, the link between *!tuck_arms both_arms* and *!move_arm_to_side left_arm* is a redundant link. These redundant links are removed after mapping the list of links to the *depends* and *next* field of each *Action* (algorithm 2, lines 8 – 11). The preparation for the

parallelization is now complete. A list of *Actions* with dependencies and successors as well as a graph in set notation ($G = \{V, E\}$) exist.

Lines 12 – 22 of algorithm 2 prepare the recursive parallelizing algorithm (algorithm 3). The list of *Actions* is devided into a start list containing *Actions* with an empty *depends* field and a list containing all other *Actions* to keep track of the *Actions* already contained in the parallel list. Since the start list *Actions* do not depend on any other *Action*, they may be executed at first. Depending on the amount of *Actions* in the start list, the first container is created. In case the start list is empty, then the graph is incorrect and the algorithm will halt. With only one *Action* in the start list, a sequential container is created and the *Action* is placed inside. Multiple *Actions* in the start list are placed in a parallel container. Note, that the *Sequence* and *Concurrence* containers provided by SMACH (Chapter 2.4) are not used yet, but rather objects of the type *SList* and *PList* derived from Pythons *List* type. The start list and the list of left over *Actions* are passed to the recursive paralellization algorithm.

4.2.2. Parallelizing

Algorithm 3 shows a simplified representation of the recursive parallelization algorithm. Each time the algorithm adds an *Action* from the list of left over *Actions* (referred to as *actions_left*), the current *item* is run through the algorithm again and the algorithm is run until *actions_left* contains no more *Actions*. *ParallelizePlanRecursively()* runs through its input, which may be an *SList* or a *PList*, containing *Actions*, *SLists* and *PLists*. Each element in the top-level list is examined depending on its type. The element will be referred to as *item*:

Action has three sub-types:

I. The *item* has exactly one successive *Action*, this successor has exactly one dependency and this dependency is the *item*. Obviously, the two *Actions* are linked with a single link and no interferences. If the *item* is already contained in a *SList*, then the *Action* may simply be appended to the *SList* behind the *item*. If the *item* is contained in a *PList*, the *Action* may not be inserted into the *PList*, so a *SList* is created to wrap the *PList* and the *Action*.

II. The *item* has multiple successors, all these successors have a single dependency, which is the *item*. This case is quite similar to the first, the

Algorithm 3: ParallelizePlanRecursively()

input: s_list; actions_left

```
1  foreach item in s_list do
2      switch Type(item) do
3          case Action
4              if len(item.suc) = 1 and len(item.suc[0].dep) = 1 and item.suc[0].dep =
                   item then
5                  if item contained in SList then SList.append(item);
6                  else if item contained in PList then
7                      PList.append(new SList(PList.pop(item),item.suc));
8              else if len(item.suc) > 1 and All(i in item.suc: i.dep = 1 and i.dep[0] =
                   item) then
9                  if item contained in SList then
10                     SList.insert(behind item: new PList(item.suc[0 ... n]));
11                 else if item contained in PList then
12                     PList.append(new SList(PList.pop(item), new PList(item.suc[0
                           ... n])));
13             else if len(item.suc) = 1 and len(item.suc[0].dep) > 1 then
14                 new SList, newPList;
15                 foreach element in s_list do
16                     if element ∈ item.suc[0].dep then PList.append(element);
17                     else SList.append(element);
18                 s_list ← new SList(PList, item, SList[0 ...n]);
19         case PList
20             if len(item.suc) = 1 and All(item.suc[0] = item.suc[0]) and
                   len(item.suc[0].dep) = len(item) then
21                 PList.parent.append(item.suc[0]);
22             if len(item.suc) > 1 and All(item.suc.dep ∈ item) then
23                 PList.parent.append(new PList(item.suc));
24         case SList
25             if len(item.suc) > 1 and All(len(item.suc.dep) = 1 and item.suc.dep[0] =
                   last(item)) then
26                 SList.parent.append(new PList(item.suc[0 ... n]));
27             if len(item.suc) = 1 then
28                 if len(item.suc[0].dep) = 1 and item.suc[0].dep[0] = last(item) then
29                     s_list ← new List(new SList(item, item.suc[0].dep[0]),
                           All(element ∈ actions_left|element ∉ SList));
30                 if len(item.suc[0].dep) > 1 and item.suc[0].dep[0 ... n] ∉ actions_left
                       then
31                     s_list ← new List(new SList(item, new PList(item.suc[0].dep[0
                           ... n]), All(element ∈ actions_left|element ∉ PList)));
```

only difference is, that the multiple successive *Actions* are encapsulated into a *PList* before appending or wrapping them.

III. The *item* has a single successor, but this successor has multiple dependencies. All elements in the start list (respectively the *s_list*) are sorted into elements that the successors depends on, and those that it does not. The elements that the successor depends on are placed in a *PList*, the others in a *SList*. The *s_list* is replaced by the *PList*, the *item* and the elements of the *SList*.

PList has two subtypes:

I. Each element of the *item* has exactly one successive *Action*, this *Action* is the same for each successor and each successor has the same amount of dependencies as the *item* has elements. The successive *Action* is inserted behind the *item* in the same list containing the *item*.

II. Each element has multiple successors and all dependencies of these successors are contained in the *item*. This denotes two parallel sections in direct dependance, thus, all successors can be placed in a new *PList* and can be treated like the single *Action* in I.

SList has three subtypes. The successor of an *SList* is the successor of the last item in the *SList*, the dependency obviously is the dependency of the first item:

I. The *item* has multiple successors and all successors have a single dependency, which is the *item*. All successors may be placed in a *PList* and this list is appended to the list containg the *item* behind that.

II. The *item* has a single successor, this successor has a single dependency which is the last item in *item*. *s_list* is replaced by a list containing the *item*, the successor of *item* and the other *Actions* contained in *s_list*. The list types depend on the previously contained list types and the amount of *Actions* and lists contained in *s_list*.

III. The *item* has a single successor with multiple dependencies, which are not in *actions_left*. Similar to II., *s_list* is replaced by a list containing the *item*, the successors and the other *Actions*. Other than II., the successors are contained in a *PList*.

Algorithm 4: ExecuteStateMachine()

input: parallelized list of `Actions`

```
1  begin CreateSMACH(a_list)
2      while a_list do
3          switch a_list.pop(0) do
4              case [
5                  create new Concurrence;
6                  insert Concurrence into currently opened Container;
7                  open Concurrence
8              case (
9                  create new Sequence;
10                 insert Sequence into currently opened Container;
11                 open Sequence
12             case ]
13                 close Concurrence;
14                 open Concurrence.parentContainer;
15             case )
16                 close Sequence;
17                 open Sequence.parentContainer;
18             case Action
19                 insert State(Action) into currently opened Container;
20 if CreateSMACH then
21     state_machine.run
```

4.3. Creating and Executing SMACH State Machine

As seen in algorithm 1, *ExecuteStateMachine()* is passed the parallelized plan resulting from algorithm 2 and 3. This list now contains elements that identify a container or an *Action*. The algorithm runs through the complete list until it is empty, removing the elements in order from the first to the last. There are five possibilites for the element:

I. [denotes opening a *PList*, respectively a parallel container or a SMACH *Concurrence*. The *Concurrence* is created, appended to a list of open containers to keep track of the current depth, and *Concurrence.open()* opens the container. Every *Action* is now inserted into this container until it is closed.

II. (denotes opening a *SList*, respectively a SMACH *Sequence*. See I.

III.] denotes closing a *PList*. The *Concurrence* is closed using *Concurrence.close()*. Since this container is completed, it is removed from the list of open containers, reducing the depth. The last container in the list of open containers is then opened (*lastContainer.open()*).

IV.) denotes closing a *SList*. See I.

V. **Action** denotes an *Action*. *Actions* are transformed into SMACH states depending on the type of *Action* and added to the currently opened container.

Once the algorithm reached the end of the list, the list of opened containers contains only the top-level container. This container is the SMACH state machine to be executed.

For all currently possible planning scenarios, all possible split and join events are covered by the algorithm, evolution in planning scenarios will have to be covered by changes in the algorithm.

5. Evaluation

In this chapter the experiments for the *Serving Beverages* (Chapter 3.1) and for the *Loading Dishwasher* scenario are evaluated. While the *Serving Beverages* scenario was simulated and carried out practically with the robot itself, the experiments for *Loading Dishwasher* are only theoretical. Afterwards, global capabilities concerning the *RACE* architecture are discussed.

The first step is the parallelization process itself. Listing 5.1 shows the parallelized result for the approach with no security constraints. This approach will be referred to as *offensive* approach. Listing 5.2 shows the result for the approach with security constraint, forcing the robot to wait for arm and torso actions to complete before moving the base, which will be referred to as *defensive* approach. For the offensive approach, the optimal parallelization by human was presented in figure 3.3, 3.1, and also in figure A.1. In figure A.1 (as well as figures A.2, A.3 and A.4), containers are highlighted with a transparent background box. A box containing vertically aligned *Actions* or sub-boxes represents a *Sequence* container, while a box containing horizontally aligned items represents a *Concurrence* container. The plans returned by the algorithm in listings 5.1, 5.2, A.5 and 5.3 on the other hand represent containers in pairs of parentheses. Round brackets represent a *Sequence* container, squared brackets represent a *Concurrence* container. The indent of the *Action* represents the container depth discussed in chapter 4.3. The container depth in the human-optimized plans is represented by the grey level.

5.1. RACE Scenario: Serving beverages

5.1.1. Parallelization

Comparing listing 5.1 and figure A.1 reveals a flawless parallelization. The algorithm created the exact same container tree as the human optimization did. The first opening bracket represents the lightest grey box in figure A.1. Both contain the seven sequential elements and a parallel container represented by the pair of

squared brackets. This parallel container contains the *!move_base Action* and the two sequential containers.

```
1  (
2      [
3          (
4            -!tuck_arms both_arms-
5            -!move_arm_to_side left_arm-
6          )
7          (
8            -!move_torso torso_down_position-
9            -!move_torso torso_up_position-
10         )
11       -!move_base counter_1_pre_manipulation_pose-
12     ]
13   -!move_base_blind counter_1_manipulation_pose-
14   -!pick_up_object coffee_cup_1 left_arm-
15   -!move_base_blind counter_1_pre_manipulation_pose-
16   -!move_base table_1_pre_manipulation_pose-
17   -!move_base_blind table_1_manipulation_pose-
18   -!place_object coffee_cup_1 left_arm table_1-
19   -!move_base_blind table_1_pre_manipulation_pose-
20 )
```

Listing 5.1: Plan resulting from the parallelization algorithm optimizing the *Serving Beverages* scenario using no security constraints.

Listing 5.2 though shows a minimal difference. While the beginning is the same as figure A.2, starting with the overall sequential container, containing the first parallel container, followed by the single *!move_base Action*, after this *Action*, an unnecessary *Sequence* is inserted. This container is the result of merging the lists in algorithm 3, lines 27 – 29. Although the automated parallelization returns a plan with an additional sequential container, this does not effect the parallel execution of the plan, as the additional *Sequence* is contained in a *Sequence* itself and only contains sequential *Actions*. The resulting parallel execution can be expected to be the same as the human optimized plan.

5.1.2. Simulated Experiment

As the *gazebo* simulator returns the same results as the practical experiments and is only used to determine if the execution of the plan performs correctly and presents no threat to humans, the environment or the robot itself, the experiment itself is not described. The only important result from the simulated experiment concerns the type of plan to be executed in the practical experiment. During the simulated experiment, while executing the *offensive* approach, the robot had multiple collisions

```
1  (
2      [
3        -!tuck_arms both_arms-
4        -!move_torso torso_down_position-
5      ]
6    -!move_base counter_1_pre_manipulation_pose-
7      (
8          [
9            -!move_torso torso_up_position-
10           -!move_arm_to_side left_arm-
11         ]
12       -!move_base_blind counter_1_manipulation_pose-
13       -!pick_up_object coffee_cup_1 left_arm-
14       -!move_base_blind counter_1_pre_manipulation_pose-
15       -!move_base table_1_pre_manipulation_pose-
16       -!move_base_blind table_1_manipulation_pose-
17       -!place_object coffee_cup_1 left_arm table_1-
18       -!move_base_blind table_1_pre_manipulation_pose-
19     )
20 )
```

Listing 5.2: Plan resulting from the parallelization algorithm optimizing the *Serving Beverages* scenario using security constraints.

with the environment while moving with yet untucked arms. Thus, the offensive approach cannot be evaluated in a practical experiment since it will damage the robot. Further collision avoidance development is required in order to detect collision of the dynamic elements of the robot with the environment.

5.1.3. Practical Experiment

Resulting from chapter 5.1.2, only the *defensive* approach (listing 5.2 and figure A.2 is executed. Figure 5.1 presents the execution duration of the sequential plan with and without parallelization. The complete parallel execution takes 269 seconds, while the sequential execution takes 371 seconds. The parallelization saves 27.5 % on the execution of the *Serving Beverages* scenario. Since this is the only scenario evaluated in a practical experiment in this work, the first two tasks will be examined as well. These two tasks, *!tuck_arms both_arms* and *!move_torso torso_down_position*, are elementary tasks to any scenario involving movement, which is again elementary to any mobile robot. For collision avoidance and stability during acceleration, the *PR2* must tuck its arms and move the torso down before travelling. Thus almost any possible scenario involves these first two tasks. Execution of these tasks takes 39 seconds in the classic sequential plan and 24 seconds in the optimized parallel plan, offering a benefit of 15 seconds (39.5 %) to almost any scenario, even if the scenario does not hold any other parallelization capabilities.

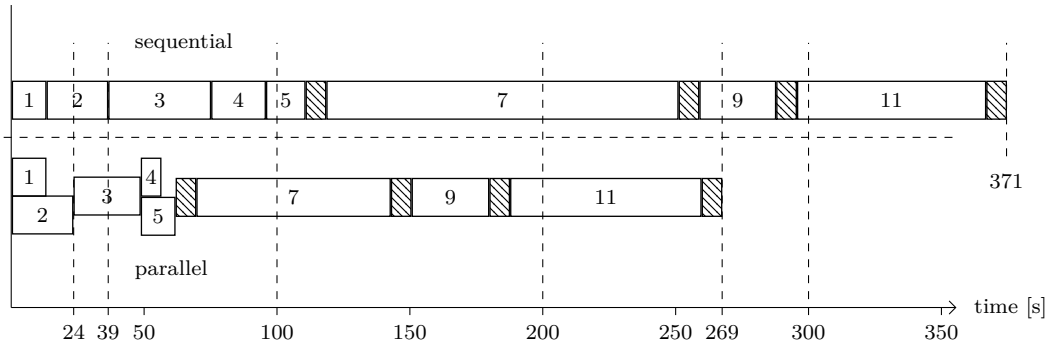

Figure 5.1.: Result from the practical experiment of the *Serving Beverages* scenario. The upper section shows the overall execution time and the execution time of each task in the sequential order. The lower section shows the overall execution time, the execution time of each task and the parallel ordering of the parallelized plan. A significant difference of 102 seconds, respectively 27.5 % is observed after optimizing the sequential plan.

Figure 5.2 shows the CPU load in percent of the robot during execution of the parallel, respectively sequential plan. Besides a small shift to the right for the sequential graph, caused by the increased duration of the first five tasks, no significant difference is observed. Both execution types produce similar processor load over time, but since the parallel execution requires 27.5 % less execution time, it also requires about 27.5 % less processor capacity. Additionally, this result offers the ability to further increase parallelization.

5.2. Theoretic Scenario: Loading a dishwasher

5.2.1. Parallelization

Chapter 5.1.2 showed that the *offensive* approach for the *Serving Beverages* scenario is not applicable. Since the difference between the *offensive* and the *defensive* approach is in the first section of both scenarios and the first section of both scenarios are nearly the same (chapter 3.2.3), it can be assumed, that the *offensive* approach for the *Loading Dishwasher* scenario is not applicable as well. For the sake of completeness, the result of the automated parallelization for the *offensive* approach can be found in listing A.5 and a close look shows the accordance with figure A.3. Note, that listings 5.3 and A.5 are missing the tasks from 14 to 20 (chapter 3.2.3), which have been removed to improve clarity of the representation. In the *defensive* approach in listing 5.3 and figure A.4, the result of the automated parallelization shows

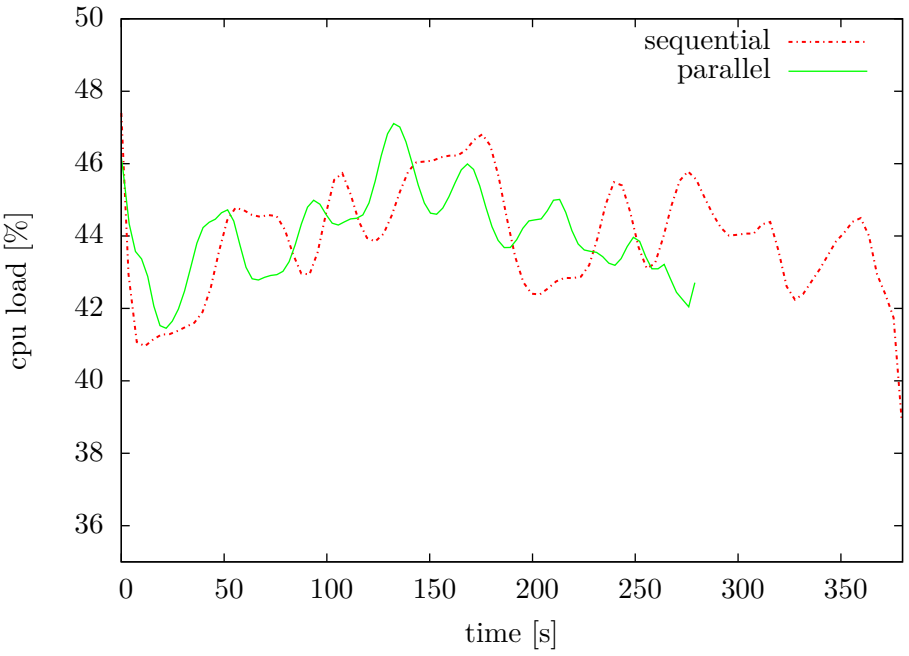

Figure 5.2.: Processor load during the practical experiment of the *Serving Beverages* scenario. Actually measured values of the computer cluster have been smoothed with a Beziér polynom and the arithmetic average was calculated. Both graphs show similiar result, the sequential line is shifted to the right due to increased execution time within the first five tasks. Overall processor load is steady comparing the sequential and the parallel execution of tasks. Figure A.5 shows processor load of both computers in the cluster for both experiments.

similiar deviation from the human-optimized plan as the *Serving Beverages* scenario did before. Unnecessary but redundant sequential containers are opened on lines 7 and 27 and closed on the lines 40 – 41. On a first glimpse, lines 29 – 33 look out of order, but a closer look shows that the tasks *!pick_up_cup cup5* and *!place_cup cup5* being encapsulated in a sequential container does not effect the execution duration of the parallelized plan. In fact, it is the only possible execution order, as the automated parallelization algorithm is not able to create links that leap over tasks.

5.2.2. Experiments

As the *Loading Dishwasher* scenario is a theoretic scenario, simulated and practical experiments can not yet be performed. The result from chapter 5.2.1 gives an idea on the possibilities of the parallelization algorithm. Reduction of approximately 40 %

```
1  (
2      [
3        -!tuck_arms both_arms-
4        -!move_torso torso_down_position-
5      ]
6    -!move_base pre_manipulation_pose-
7      (
8          [
9            -!move_torso torso_manipulation_position-
10           -!move_arm_to_side left_arm-
11           -!move_arm_to_side right_arm-
12         ]
13       -!move_base_blind manipulation_pose-
14       -!pick_up_cup cup1-
15         [
16           -!place_cup cup1-
17           -!pick_up_plate plate1-
18         ]
19         [
20           -!pick_up_cup cup2-
21           -!place_plate plate1-
22         ]
23         [
24           -!pick_up_plate plate2-
25           -!place_cup cup4-
26         ]
27         (
28             [
29                 (
30                   -!pick_up_cup cup5-
31                   -!place_cup cup5-
32                 )
33               -!place_plate plate4-
34             ]
35           -!move_base_blind pre_manipulation_pose-
36             [
37               -!tuck_arms both_arms-
38               -!move_torso torso_down_position-
39             ]
40         )
41     )
42 )
```

Listing 5.3: Plan resulting from parallelization algorithm on the *Loading Dishwasher* scenario using security constraints. Note, that between line 24 and 25 portions of the repetitive output have been skipped.

can be expected. This can be derived from: (1) The measured benefits in the first section of the *Serving Beverages* scenario, which is similar in both scenarios; (2) The shortening of the tasks list length by 50 % in the object manipulation section by always executing two tasks in parallel. In (2), an overhead due to uneqal task length of picking and placing is expected.

5.3. Integration into current Architecture of RACE Project

Although the results from automated parallelization look promising, capabilities are currently very limited. With the current setup, only the operators in listing A.2 may be contained in the planning domain for the HTN planner. In case a new operator is added to the domain, the operator and its required resources have to be added to the list. Additionally, the current operators may be decomposed into smaller operators which provide more parallelization capabilities. This “size” of the atomic tasks and

the need for a human to decide on the required resources of an operator pose a drawback to the efficiency and the applicability of the parallelization algorithm to real world problems.

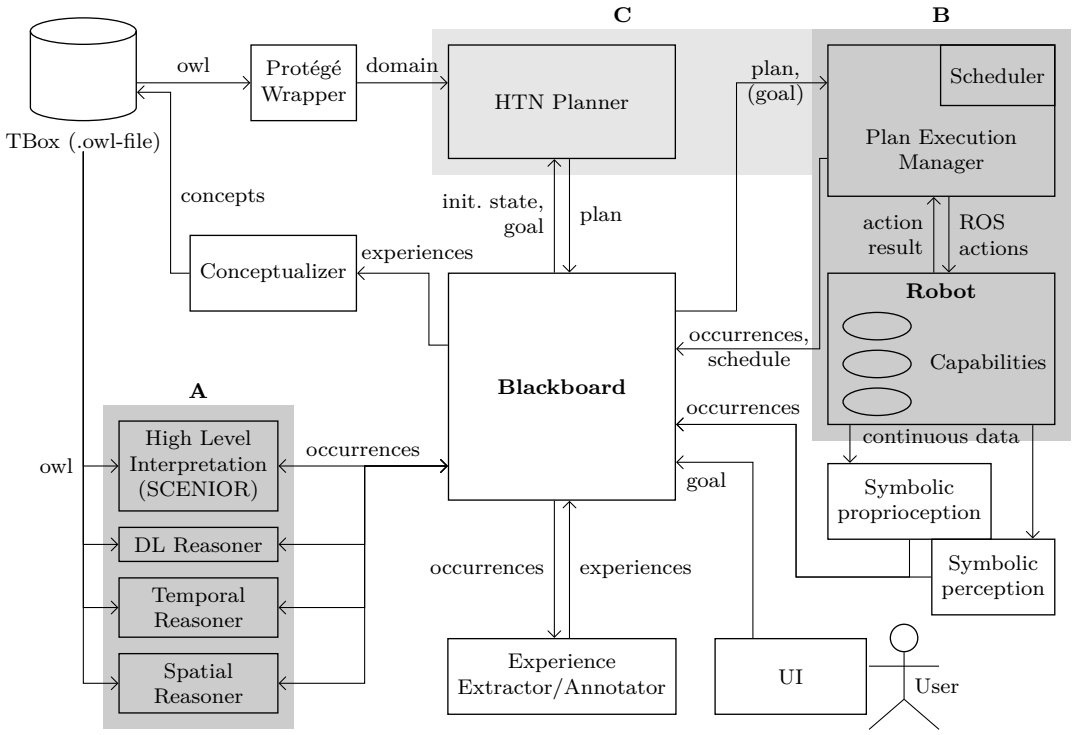

Figure 5.3.: Global architecture of the RACE project. A Blackboard is the center of the structure managing the communication. Block **A** highlights the *Reasoners*, blocks **B** and **C** hold the modules used in this work.

Figure 5.3 shows the current architecture of the RACE Project [2012]. Communication is currently planned to run via a *Blackboard* system [Nii 1986]. The *Blackboard* connects the areas **B** and **C** with area **A**. **B** contains the parallelization algorithm and the interpreter for the SMACH state machine creator, **C** contains the JSHOP2 planning system. **A** contains multiple *Reasoners* which are currently developed by part of the RACE project. One of the objectives of the *Reasoners* is to be able to decide from its experience, whether an operator is an atomic task or should be decomposed to smaller operators and, this is most important, which tasks require which resources including resource-blocking security constraints as seen for the *defensive* approach. Eventually, one of these *Reasoners* will be able to inherit the role of the hard-coded operator resources in listing A.2 described in chapter 4.2.1. This will remove the need to edit the list of operator resources by hand and remove the last non-autonomous part from the planning process, since judging on the atomic

tasks will also allow the *Reasoner* to create the planning domain for JSHOP2. Another possibility presented by the *Reasoner* is the dynamic adaption of resources required by an operator. The *offensive* approach described in chapter 3.1.3 with no security constraints may be applicable in an environment with no or few obstacles, while environments with many obstacles and narrow passages require the *defensive* approach with security constraints. Variables in the environment, like the size of the object, the height of the obstacles, the lighting conditions and others, may also change the required resources for some tasks. The *Reasoner* will be able to decide which operator requires which resources based on experiences made in similar environments before. The *Reasoner* will also be able to decide from its experience gained during previous executions, which task is the best atomic operator for a specific scenario. Depending on the scenario, the size of the atomic operator for the same task may differ. The *Reasoner* will also be able to create new scenario planning domains from its knowledge base. This will not only allow the parallelization of nearly any problem without human interaction but also increase the parallelization capabilities, as the *Reasoner* finds the optimal atomic tasks for the parallel execution of the plan. Effienceny may thus increase to even more than 27.5 %.

JSHOP2 and other SHOP2 implementations are available for multiple platforms and the Python programming language also supports multiple operating systems. Since the parallelization itself is not dependend on the interpreter for the SMACH state machine the parallelization itself is available for any platform supporting Java and Python and only requires an automatically or human generated resource mapping.

This chapter examined the equivalency of the automated parallelization algorithm presented in chapter 4 and the human-optimized parallel order presented in chapter 3, showed the increased efficiency concering execution time and processor load and pointed out the opportunities inside the RACE project concerning autonomous parallelization.

6. Conclusion

This work aims to improve execution time and resource management on mobile robots. Not only to increase efficiency and reduce expenses for resources and energy, but also reduce the duration of development cycles for applied research. Up to the authors knowledge there are currently no applicable solutions to improving the task execution on mobile robots using *Hierarchical Task Network* planners by executing tasks in parallel.

Although this work was created and evaluated on the mobile platform *Personal Robot 2*, the results are applicable to almost any platform running the *Robot Operating System* and other systems supporting the SMACH state machines, since *JSHOP2* and other HTN planners run on any platform. A quick introduction to *JSHOP2* and *SMACH* was presented in chapter 2 and two scenarios for the evaluation were analysed in chapter 3. After a simplified look into the implementation in chapter 4, the parallel execution was evaluated in chapter 5.

The evaluation identified security issues due to the parallelization of planar movements and torso and joint movement which could be resolved. For one of the scenarios discussed in this work, the evaluation of the parallel execution revealed a time benefit of 27.5 % compared to the sequential execution with a constant processor load. Last of all, a solution to the remaining shortcoming in autonomy, resulting from the required human-written operator file, was presented with the architecture of the RACE Project [2012] in chapter 5.3. More experiments may be required to increase the value of the evaluation.

The parallelization approach presented is already used in the current development process of the RACE project at group TAMS. Evaluation in chapter 5 shows the almost universal area of application for the parallelization presented in this work.

7. Outlook

The RACE Project [2012] aims to place a mobile robot in a catering environment. The evaluation in chapter 5 revealed a time benefit of 102 seconds (27.5 %) for a single test run. A setting with multiple robots in a working environment with 16 or 24 working hours not only significantly lowers operating costs. A restaurant requiring 24 working hours of five robots with sequential plans not only saves energy and maintenance costs, but also saves expenses for a whole robot, as four robots with parallel plans might be able to carry out the same work as five sequential robots.

The evaluation already showed the benefits from parallelization in current scenarios with human-composed resource allocation and plans. With fully working *Reasoners*, which are currently developed, the benefits from parallelization with optimized size of atomic tasks will exceed the results from evaluation and increase autonomy of planning and optimizing.

Bibliography

Asfour et al. 2006

Asfour, T. ; Regenstein, K. ; Azad, P. ; Schroder, J. ; Bierbaum, A. ; Vahrenkamp, N. ; Dillmann, R.: ARMAR-III: An Integrated Humanoid Platform for Sensory-Motor Control. In: *Humanoid Robots, 2006 6th IEEE-RAS International Conference on*, 2006, pp. 169 – 175 2.1

Asfour et al. 2004

Asfour, Tamim ; Ly, D. N. ; Regenstein, Kristian ; Dillmann, Rüdiger: Coordinated Task Execution for Humanoid Robots. In: Jr., Marcelo H. A. (Editor) ; Khatib, Oussama (Editor): *ISER* Bd. 21, Springer, 2004 (Springer Tracts in Advanced Robotics). – ISBN 978–3–540–28816–9, 259 – 267 2.1

Berger et al. 2012

Berger, Eric ; Conley, Ken ; Faust, Josh ; Foote, Tully ; Gerkey, Brian ; Leibs, Jeremy ; Quigley, Morgan ; Wheeler, Rob: *Robot Operating System (ROS)*, 2012. http://www.ros.org, Last checked: 10.09.2012 2.2

Bohren and Cousins 2010

Bohren, J. ; Cousins, S.: The SMACH High-Level Executive [ROS News]. In: *Robotics Automation Magazine, IEEE* 17 (2010), dec., no. 4, pp. 18 – 20. http://dx.doi.org/10.1109/MRA.2010.938836. – DOI 10.1109/MRA.2010.938836. – ISSN 1070–9932 2.2, 2.4

Bohren et al. 2011

Bohren, J. ; Rusu, R.B. ; Jones, E.G. ; Marder-Eppstein, E. ; Pantofaru, C. ; Wise, M. ; Mosenlechner, L. ; Meeussen, W. ; Holzer, S.: Towards autonomous robotic butlers: Lessons learned with the PR2. In: *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, 2011. – ISSN 1050–4729, pp. 5568 – 5575 2.2, 2.4

Bohren 2012

Bohren, Jonathan: *SMACH Package Documentation and Wiki*, 2012. http://www.ros.org/wiki/smach, Last checked: 08.08.2012 2.4

Castillo et al. 2006

Castillo, Luis ; Fdez-Olivares, Juan ; García-Pérez, Óscar ; Palao, Francisco: Temporal enhancements of an HTN planner. In: *Proceedings of the 11th Spanish association conference on Current Topics in Artificial Intelligence*. Berlin, Heidelberg : Springer-Verlag, 2006 (CAEPIA'05). – ISBN 3–540–45914–6, 978–3–540–45914–9, 429 – 438 2.1

Currie and Tate 1991

Currie, Ken ; Tate, Austin: O-Plan: the open planning architecture. In: *Artif. Intell.* 52 (1991), November, no. 1, 49 – 86. http://dx.doi.org/10.1016/0004-3702(91)90024-E. – DOI 10.1016/0004–3702(91)90024–E. – ISSN 0004–3702 2.3

Devaurs et al. 2011

Devaurs, D. ; Simeon, T. ; Cortes, J.: Parallelizing RRT on distributed-memory architectures. In: *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, 2011. – ISSN 1050–4729, pp. 2261 – 2266 2.1

Erol et al. 1994

Erol, Kutluhan ; Hendler, James A. ; Nau, Dana S.: UMCP: A Sound and Complete Procedure for Hierarchical Task-network Planning. In: *AIPS*, 1994, pp. 249 – 254 2.3

Gat 1998

Gat, Erann: On Three-Layer Architectures. In: *Artificial Intelligence and Mobile Robots*, MIT Press, 1998 2.1

Hassan and Cousins 2012a

Hassan, Scott ; Cousins, Steve: *Personal Robot 2 (PR2)*, 2012. http://www.willowgarage.com/pages/pr2/overview, Last checked: 10.09.2012 2.2

Hassan and Cousins 2012b

Hassan, Scott ; Cousins, Steve: *Robot Operating System (ROS)*, 2012. http://www.willowgarage.com/pages/software/ros-platform, Last checked: 10.09.2012 2.2

Hassan and Cousins 2012c

Hassan, Scott ; Cousins, Steve: *Willow Garage*, 2012. http://www.willowgarage.com/, Last checked: 03.07.2012 2.2

Ilghami 2006

Ilghami, Okhtay: Documentation for JSHOP2 / Departement of Computer Science, University of Maryland. College Park, MD 20742, USA : University of Maryland, 05 2006 (CS-TR-4694). – Technical Report 2.2, 2.3, 2.3

Jacobs et al. 2012

Jacobs, Sam A. ; Manavi, Kasra ; Burgos, Juan ; Denny, Jory ; Thomas, Shawna ; Amato, Nancy M.: A scalable method for parallelizing sampling-based motion planning algorithms. In: *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, 2012. – ISSN 1050–4729, pp. 2529 – 2536 2.1

Kautz and Selman 1996

Kautz, Henry ; Selman, Bart: Pushing the envelope: planning, propositional logic, and stochastic search. In: *Proceedings of the thirteenth national conference on Artificial intelligence – Volume 2*, AAAI Press, 1996 (AAAI'96). – ISBN 0–262–51091–X, 1194 – 1201 2.1

Lingard and Richards 1998

Lingard, A. R. ; Richards, E. B.: Planning parallel actions. In: *Artif. Intell.* 99 (1998), März, no. 2, 261 – 324. http://dx.doi.org/10.1016/S0004-3702(97)00080-5. – DOI 10.1016/S0004–3702(97)00080–5. – ISSN 0004–3702 2.1

Luh and Lin 1985

Luh, J.Y.S. ; Lin, C.S.: Scheduling parallel operations in automation for minimum execution time based on pert. In: *Computers & Industrial Engineering* 9 (1985), no. 2, 149 – 164. http://dx.doi.org/10.1016/0360-8352(85)90014-2. – DOI 10.1016/0360–8352(85)90014–2. – ISSN 0360–8352 2.1

Nau et al. 2001

Nau, Dana ; Avila, Héctor Muñoz ; Cao, Yue ; Lotem, Amnon ; Mitchell, Steven: Total-order planning with partially ordered subtasks. In: *Proceedings of the 17th international joint conference on Artificial intelligence – Volume 1*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2001 (IJCAI'01). – ISBN 1–55860–812–5, 978–1–558–60812–2, 425 – 430 1.2, 2.3

Nau et al. 1999

Nau, Dana ; Cao, Yue ; Lotem, Amnon ; Munoz-Avila, Hector: SHOP: Simple Hierarchical Ordered Planner. In: *Proceedings of the 16th international joint conference on Artificial intelligence - Volume 2*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1999 (IJCAI'99), 968 – 973 2.1, 2.3

Nii 1986

Nii, H. P.: Blackboard Systems, Part One: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures. In: *AI Magazine* 7 (1986), no. 2, pp. 38 – 53 5.3

Quigley et al. 2009

Quigley, Morgan ; Conley, Ken ; Gerkey, Brian P. ; Faust, Josh ; Foote, Tully ; Leibs, Jeremy ; Wheeler, Rob ; Ng, Andrew Y.: ROS: an open-source Robot Operating System. In: *ICRA Workshop on Open Source Software*, 2009 2.2

RACE Project 2012

RACE Project: *Robustness by Autonomous Competence Enhancement Project*, 2012. http://www.project-race.eu/, Last checked: 03.07.2012 1.1, 2.2, 2.3, 5.3, 6, 7

Rintanen et al. 2006

Rintanen, Jussi ; Heljanko, Keijo ; Niemelä, Ilkka: Planning as satisfiability: parallel plans and algorithms for plan search. In: *Artif. Intell.* 170 (2006), September, no. 12 – 13, 1031 – 1080. http://dx.doi.org/10.1016/j.artint.2006.08.002. – DOI 10.1016/j.artint.2006.08.002. – ISSN 0004–3702 2.1

Sousa et al. 1996

Sousa, J.B. ; Pereira, F.L. ; Silva, E.P. da ; Martins, A. ; Matos, A. ; Almeida, J. ; Cruz, N. ; Tunes, R. ; Cunha, S.: On the design and implementation of a control architecture for a mobile robotic system. In: *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on* Bd. 3, 1996, pp. 2822 – 2827 2.1

Taipalus and Halme 2009

Taipalus, Tapio ; Halme, Aarne: An action pool architecture for multi-tasking service robots with interdependent resources. In: *Proceedings of the 8th IEEE international conference on Computational intelligence in robotics and automation*. Piscataway, NJ, USA : IEEE Press, 2009 (CIRA'09). – ISBN 978–1–4244–4808–1, 228 – 233 2.1

Tate 1977

Tate, Austin: Generating project networks. In: *Proceedings of the 5th international joint conference on Artificial intelligence – Volume 2*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1977 (IJCAI'77), 888 – 893 2.3

Wilkins 1991

Wilkins, David E.: Can AI planners solve practical problems? In: *Comput. Intell.* 6 (1991), Januar, no. 4, 232 – 246. http://dx.doi.org/10.1111/j.1467-8640.1990.tb00297.x. – DOI 10.1111/j.1467–8640.1990.tb00297.x. – ISSN 0824–7935 2.3

Eidesstattliche Erklärung

Ich, Lasse Einig, Matrikel-Nr. 591 83 87, versichere hiermit, dass ich meine Bachelorarbeit mit dem Thema

> *Parallel Plan Execution on a Mobile Robot with an HTN Planning System – A Resource Based Approach*

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Mir ist bekannt, dass ich meine Bachelorarbeit zusammen mit dieser Erklärung fristgemäß nnach Vergabe des Themas in dreifacher Ausfertigung und gebunden im Prüfungsamt der Universität Hamburg, Departement Informatik abzugeben oder spätestens mit dem Poststempel des Tages, an dem die Frist abläuft, zu senden habe.

Ich bin mit der Veröffentlichung in der Bibiliothek des Departement Informatik an der Universität Hamburg einverstanden.

Hamburg, den 18. Oktober 2012

Lasse Einig

A. Appendix

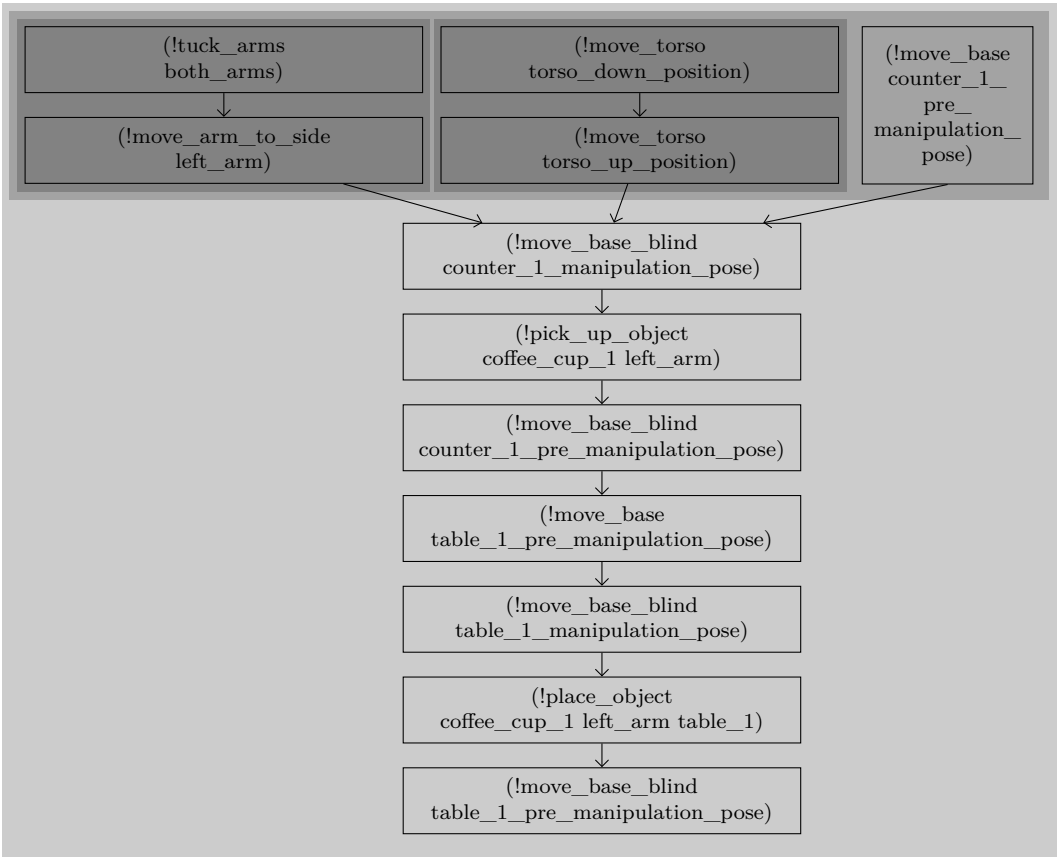

Figure A.1.: Offensive approach to parallelizing the *Serving Beverages* scenario. No security constraint is added to *!move_base counter_1_pre_manipulation_pose*, posing possible threat to the environment during execution. Due to security threats by movements without collision detection, there are no parallelization capabilities in the latter section. The grey boxes represent parallel and sequential sections as explained in chapter 5.

```
class PlanExecutor():
    def __init__(self):
        rospy.loginfo('connecting to jshop2_planner...')
        self.jshop2_planner_client = actionlib.SimpleActionClient(
            'jshop2_planner', PlanningAction)
        self.jshop2_planner_client.wait_for_server()
        rospy.loginfo(' successfully  connected to jshop2_planner.')

    def run(self):
        rospy.loginfo(' calling  JSHOP2 planner...')
        planning_goal = PlanningGoal()
        planning_goal.tasks = ["serve_cup table_1"]
        self.jshop2_planner_client.send_goal(planning_goal)
        self.jshop2_planner_client.wait_for_result()
        planning_result = self.jshop2_planner_client.get_result()
        rospy.loginfo('JSHOP2 planner returned.')
        try:
            plan = planning_result.plans.pop(0)
        except IndexError:
            rospy.logerr("no plan found!")
            return
```

Listing A.1: Section of the implementation showing the initialisation of the JSHOP2 planner, sending the planning goal to the JSHOP2 planner and retrieving the plan.

```
operators = [['!tuck_arms both_arms',['RA','LA']],
             ['!tuck_arms left_arm',['LA']],
             ['!tuck_arms right_arm',['RA']],
             ['!move_base ?to',['B','RA','LA','T']],  #secure
             ['!move_base_blind ?to',['RA', 'LA', 'T', 'B', 'H']],
             ['!move_torso ?position',['T']],
             ['!move_arm_to_side left_arm',['LA']],
             ['!move_arm_to_side right_arm',['RA']],
             ['!pick_up_object ?object left_arm',['LA', 'H']],
             ['!pick_up_object ?object right_arm',['RA', 'H']],
             ['!place_object ?object left_arm ?to',['LA', 'H']],
             ['!place_object ?object right_arm ?to',['RA', 'H']],
             # Dishwasher Operators
             ['!pick_up_plate ?object',['LA', 'H']],
             ['!pick_up_cup ?object',['RA', 'H']],
             ['!place_plate ?object',['LA']],
             ['!place_cup ?object',['RA']]]
```

Listing A.2: Section of the implementation showing the list of operators with resources. *#secure* marks the resources for the defensive approach. The last four operators are required by the *Loading Dishwasher* scenario only.

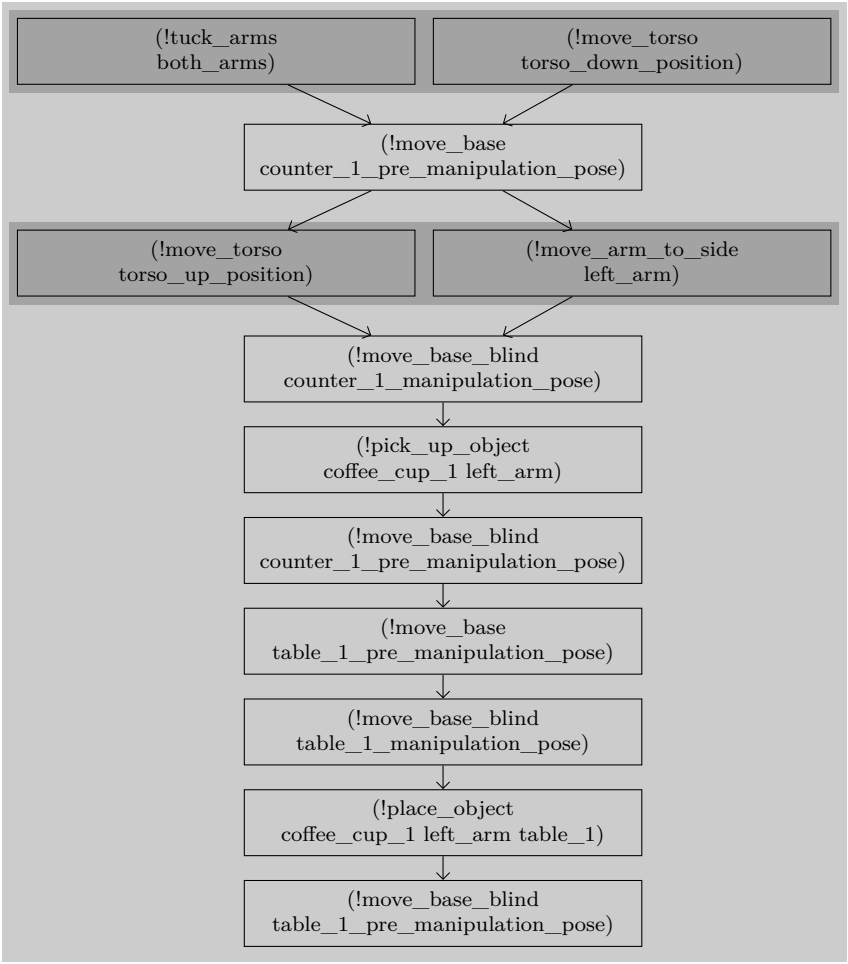

Figure A.2.: Defensive approach to parallelizing the *Serving Beverages* scenario. A security constraint is added to *!move_base counter_1_pre_manipulation_pose* forcing the robot to finish arm and torso movements before moving the base to the new position and waiting for the base movement to finish before the arms and torso may operate again. This prevents the robot from hitting objects or humans in the environment, respectively increases the tilt stability while moving due to a lower center of mass. Due to security threats by movements without collision detection, there are no parallelization capabilities in this section. The grey boxes represent parallel and sequential sections as explained in chapter 5.

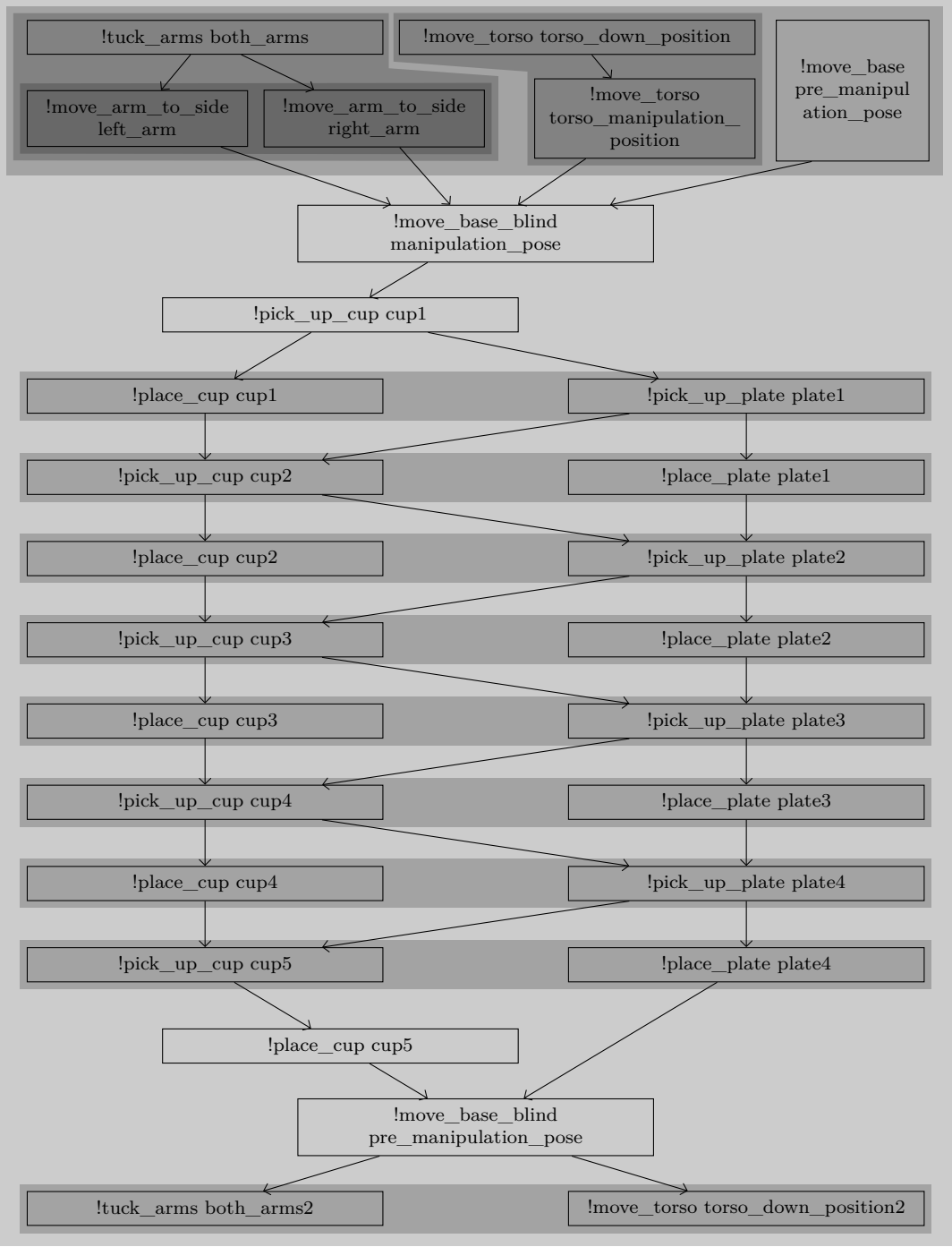

Figure A.3.: Offensive approach to parallelizing the *Loading Dishwasher* scenario. No security constraint is added, posing possible threat to the environment. The grey boxes represent parallel and sequential sections as explained in chapter 5.

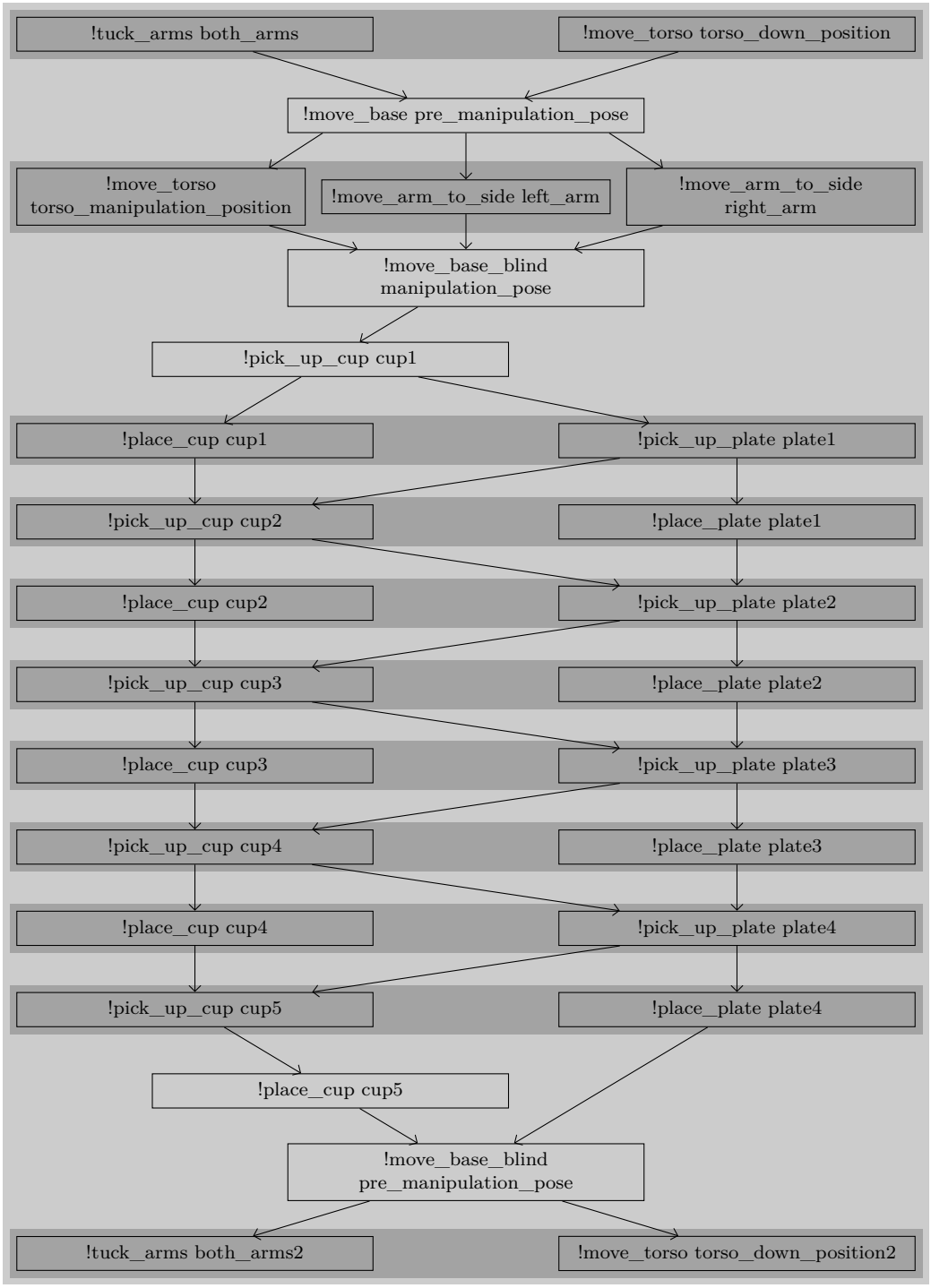

Figure A.4.: Defensive approach to parallelizing the *Loading Dishwasher* scenario. A security constraint is added to *!move_base counter_1_pre_manipulation_pose*. The grey boxes represent parallel and sequential sections as explained in chapter 5.

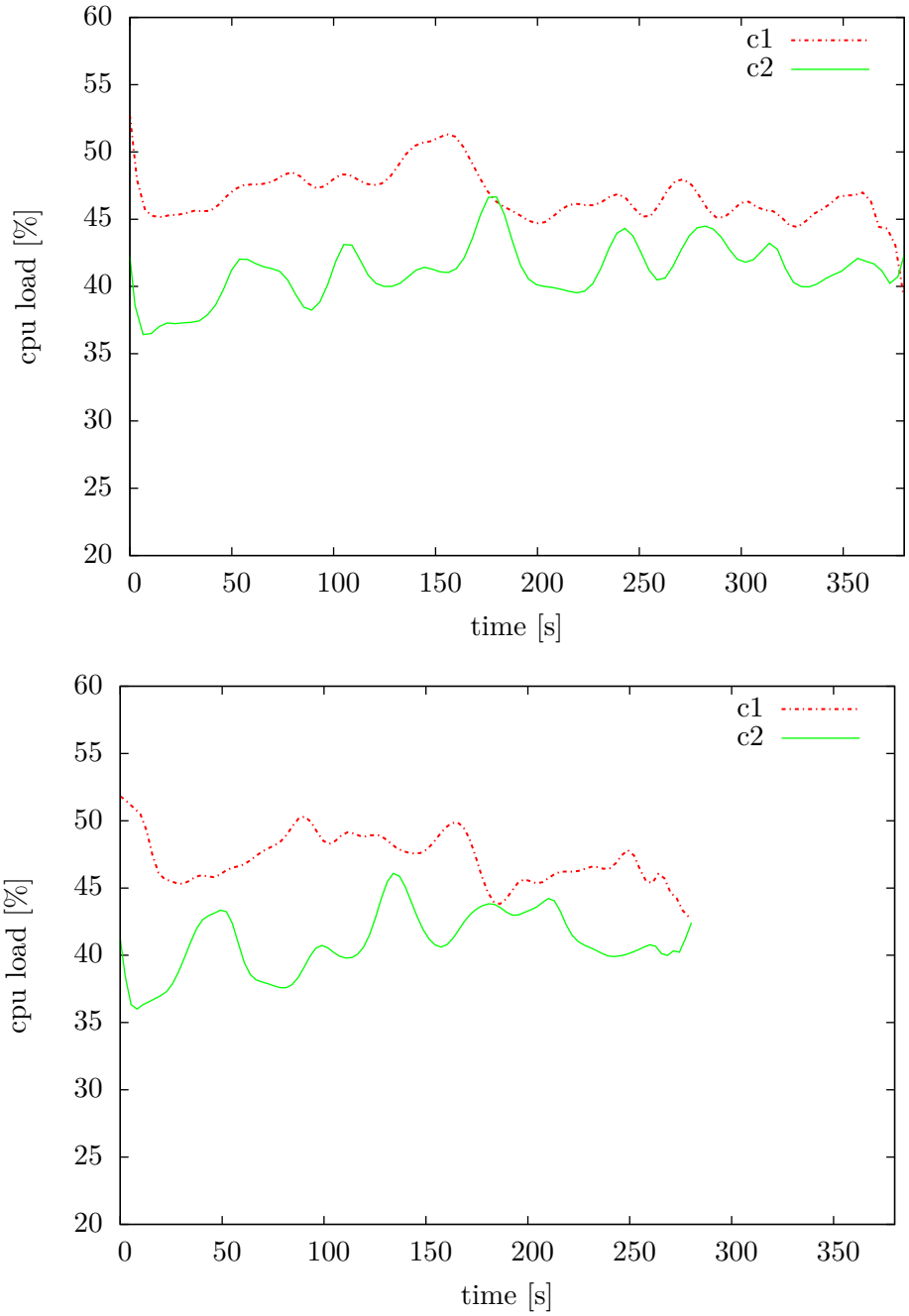

Figure A.5.: The upper figure shows the processor load during the experiment with sequential execution for the two computers *c1* and *c2* in the computer cluster of the PR2, the lower figure shows the processor load during the experiment with the parallel execution. Actually measured values of the computer cluster have been smoothed with a Beziér polynom.

```
class Action(object):
    def __init__(self, name, id_nr, ressources=[], depends=[]):
        self.id = id_nr
        self.name = name
        self.ressources = ressources
        self.depends = depends
        self.visited_from = 0
        self.next = []
        self.marked = 0

    def depends_on(self, b):
        return b in self.depends

    def ressource_req(self, res_type):
        return res_type in self.ressources

    def __repr__(self):
        return self.name
```

Listing A.3: Section of the implementation showing the *Action* class required by the complete parallelization process. The class holds fields for resources, depencies, successors, name and ID, as well as fields for the algorithm to remove redundances.

```
def plan(actions):
    links = []
    for action in actions:
        for ressource in action.ressources:
            for dep_on in actions:
                if dep_on == action:
                    break
                if ressource in dep_on.ressources:
                    links.append([action, dep_on])
    return links
```

Listing A.4: Section of the implementation showing the algorithm to create links between *Actions* depending on the resources required by the *Action*. This algorithm is similar to algorithms to create graphs.

```
1  (
2      [
3          (
4            −!tuck_arms both_arms−
5              [
6                −!move_arm_to_side left_arm−
7                −!move_arm_to_side right_arm−
8              ]
9          )
10         (
11           −!move_torso torso_down_position−
12           −!move_torso torso_manipulation_position−
13         )
14       −!move_base pre_manipulation_pose−
15     ]
16   −!move_base_blind manipulation_pose−
17   −!pick_up_cup cup1−
18     [
19       −!place_cup cup1−
20       −!pick_up_plate plate1−
21     ]
22     [
23       −!pick_up_cup cup2−
24       −!place_plate plate1−
25     ]
26     [
27       −!pick_up_plate plate2−
28       −!place_cup cup4−
29     ]
30     (
31         [
32             (
33               −!pick_up_cup cup5−
34               −!place_cup cup5−
35             )
36           −!place_plate plate4−
37         ]
38       −!move_base_blind pre_manipulation_pose−
39         [
40           −!tuck_arms both_arms−
41           −!move_torso torso_down_position−
42         ]
43     )
44 )
```

Listing A.5: Plan resulting from parallelization algorithm on the *Loading Dishwasher* scenario using no security constraints. Note, that between line 27 and 28 portions of the repetitive output have been skipped

Acknowledgement

This work was created at group TAMS, Department Informatics, University of Hamburg.

I would like to thank Prof. Jianwei Zhang for the trust reposed in me by supervising my work and giving me the opportunity to work on the PR2. and Prof. Bernd Neumann for his professional advise.

I would also like to thank Denis Klimentjew and Sebastian Rockel for supporting me with this thesis by words and deeds.

On a personal note, I would like to thank my parents, Peter and Barbara Einig, for giving me the opportunity to study unburdened, and my brother Kolja Einig for his helpful suggestions on my writing. I also would like to thank Claudia Ringelhan for supporting and motivating me throughout the writing process.