UNIVERSITÄT BREMEN

BACHELOR THESIS

Plan Transformation for Autonomous Real World Pick and Place Tasks

Author:
Arthur NIEDZWIECKI

Supervisor:
Prof. Michael BEETZ
Second Supervisor:
Prof. Johannes SCHÖNING
Advisor:
Gayane KAZHOYAN

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Institute for Artificial Intelligence
Computer Science

April 4, 2018

Declaration of Authorship

I, Arthur NIEDZWIECKI, declare that this thesis titled, "Plan Transformation for Autonomous Real World Pick and Place Tasks" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

UNIVERSITÄT BREMEN

Abstract

Faculty 3
Computer Science

Bachelor of Science

Plan Transformation for Autonomous Real World Pick and Place Tasks

by Arthur NIEDZWIECKI

For several centuries, computer scientists try to develop Artificial Intelligence for robots, to make them behave more like human beings. By intensive planning they try to mirror complex activities to be performed by machines. Today there is a huge variety of devices to support household chores and robots to clean the environment autonomously. Humanoid robots are developed to carry out even more complicated tasks in the context of everyday activities. Inspired by human capabilities of adapting to unknown environments, robots are taught to do the same by constantly improving their activities through reconsidering their decisions. Plan-based robotics focus on the design of tasks - from minute motions to extensive activities - considering misbehavior, changes of the environment, the robot's current state and many more, to create stable and reliable components for building even more complex activities.

This thesis concentrates on investigating the behavior of a PR2 robot in a kitchen environment, executing pick and place tasks. Collecting information about the robot's plans enables reasoning about how to improve its actions. By applying alterations on the plans the programmer change the course of actions, trying to optimize the robot's overall performance. Defining rules for transformation, the robot can then improve its task execution autonomously. My approach will demonstrate how to use the CRAM architecture to ascertain valuable information about activities and how they can be used for transformational planning of autonomous robots, performing pick and place tasks in a kitchen environment.

UNIVERSITÄT BREMEN

Zusammenfassung

Fachbereich 3
Informatik

Bachelor of Science

Plan-Transformation für Autonome, Natürliche Holen und Bringen Aufgaben

von Arthur NIEDZWIECKI

Über mehrere Jahrzehnte versuchen Wissenschaftler künstliche Intelligenz für Roboter zu entwickeln, um ihr Verhalten menschlicher zu gestalten. Durch intensive Planung versuchen sie komplexe Aktivitäten widerzuspiegeln, um sie von Robotern ausführen zu lassen. Heute gibt es eine Vielzahl an Geräten, die im Haushalt helfen und Roboter, welche die Umgebung säubern. Humanoide Roboter werden dazu entwickelt noch kompliziertere, alltägliche Aufgaben zu bewältigen. Angelehnt an der menschlichen Fähigkeit sich an unbekannte Situationen anzupassen, wird Robotern beigebracht ihre Aktionen, durch kontinuierliche Rekapitulation ihrer Tätigkeiten, zu verbessern. Plan-basierte Robotik fokussiert die Entwicklung von Aufgaben - von winzigen Bewegungen bis zu umfangreichen Aktivitäten - wobei Fehlverhalten, Veränderung der Umwelt und des Roboters berücksichtigt werden, um durch stabile und zuverlässige Komponenten komplexere Szenarien erschaffen zu können.

Diese Thesis konzentriert sich auf die Analyse des Verhaltens eines PR2 Roboters, der in einer Küche diverse Gegenstände transportiert. Das Sammeln von Informationen über die Pläne des Roboter lässt darüber argumentieren, inwiefern die Pläne zu verbessern sind. Durch Veränderung der Pläne passt der Programmierer den Ablauf der Aktionen an, um zu versuchen das allgemeine Verhalten zu optimieren. Indem Transformationsregeln definiert werden, kann der Roboter seine Aufgabe selbstständig verfeinern. Mein Ansatz zeigt wie man in der CRAM Architektur wertvolle Daten über Aktivitäten erheben kann und wie diese für Plan-Transformation auf autonomen Robotern zu verwenden sind, die in einer Küche Gegenstände transportieren.

Contents

Declaration of Authorship iii

Abstract v

Zusammenfassung vii

1 Introduction 1
- 1.1 General Approach and Research Questions . . . 2
  - 1.1.1 Plans . . . 2
  - 1.1.2 Plan Transformation . . . 3
- 1.2 Contributions . . . 3
- 1.3 Related Work . . . 3
- 1.4 Reader's Guide . . . 5

2 Foundations 7
- 2.1 ROS . . . 7
- 2.2 CRAM . . . 7
  - 2.2.1 Projection Environment . . . 8
  - 2.2.2 Designators . . . 9
  - 2.2.3 Process modules and atomic plans . . . 12
  - 2.2.4 Self-Recovering Plans . . . 13
  - 2.2.5 CRAM Prolog Reasoning . . . 14
  - 2.2.6 Execution Traces and the CRAM Task Tree . . . 15
- 2.3 Transformational Planning . . . 18

3 Methods and Implementation 21
- 3.1 Task tree analysis, Prolog predicates and transformations . . . 21
  - 3.1.1 Scenarios . . . 21
  - 3.1.2 Task tree analysis . . . 23
  - 3.1.3 Transformations . . . 24
  - 3.1.4 Applicability and input schema . . . 28
- 3.2 Generic CRAM plan transformation framework . . . 31

4 Experimental Evaluation 33
- 4.1 Evaluation of Transformation Experiments . . . 33
- 4.2 Evaluation Summary . . . 39

5 Conclusion 41
- 5.1 Summary . . . 41
- 5.2 Discussion . . . 41
- 5.3 Recommendations on Future Work . . . 42

Bibliography 49

Chapter 1

Introduction

During the last years, the interest in autonomous robots became successively bigger. The idea of robots being aware of their surroundings, doing household chores with just a few instructions, has been but a dream for a long time. The developments of useful devices in the recent past aimed for making our daily routine more comfortable. Todays technology infers the intention of the human operating it and helps the operator in various areas, like customizable Apps, auto-completion in instant messengers or health monitoring in smart watches. Nevertheless, those devices are designed to solve only a limited set of problems, provide only a few features each and, in combination, improve our living in multiple ways. But a robot capable of a huge variety of actions executable with as few help as possible, is a completely different scale of complexity. Household robots as we know them nowadays can vacuum the floor, wash the dishes *or* help preparing a meal, like for instance the *ThermoMix*[1]. One single robot is not (yet) capable of doing all this tasks, but multiple research endeavours already try to fill this gap, using robots like Willow Garage's PR2 (Wyrobek et al., 2008) or TUM-Rosie (Beetz et al., 2010). Those robots provide extensive possibilities for implementing complex tasks, e.g. doing dishes, cooking sandwiches, pancakes or popcorn, doing laundry, folding towels and many more (Maitin-Shepard et al., 2010; Beetz et al., 2011).

Autonomous robots doing every day tasks have a hard time doing human chores without being advised, especially when the robot is lacking detailed knowledge about its environment. Ingredients and tools may be spread around a kitchen without the robot knowing their exact position. This may get even more challenging when changing the robot's environment and expecting it to just do as well as previously. Thus, can a robot be an autonomously acting entity, if you have to tell it explicitly what to do, every time? Industrial robots can do the same motion over and over again, without reasoning about the things they do. A programmer wrote an explicit, static manual for such a robot, letting it repeat this task, while prohibiting any deviation. But allowing deviation is the key to autonomy.

An autonomous robot's action should not be constructed as a fix, discrete operation, but in the most abstract way possible. For implementing a robot's autonomy it is highly recommended to phrase tasks in abstract descriptions, supported by cognitive architectures. Such an architecture may be TRANER (Müller, 2008), which is based on McDermotts Reactive Plan Language (Mcdermott, 1993), or the more recent CRAM framework (Mösenlechner, 2016). They both have in common, that actions are written as a plan, which can be executed by describing what the robot has to do. Those plans contain descriptions of lower level actions, which execute their respective plan, which itself may contain descriptions, and so on. In the end, a robot's action as a whole can be described as a hierarchical tree of plans, whose leafs consist of general, atomic motions, executable by the robot. Other approaches use

[1] https://thermomix.vorwerk.de/home/

active reaction on the current situation (Beetz, 2013), implementing continuous reasoning about cognitive inputs, constantly ready to decide between various actions during execution.

Plan transformation is a field of plan based robotics, that aims for improving robots' activities. Much research has been made towards reactive plan transformation (Beetz, 2013; Kruse and Kirsch, 2010; Fedrizzi et al., 2009; Beetz, 1992; Beetz, 2002), which basically includes taking an alternative plan upon encountering a failure or suitable occasion, and replacing the original one with the alternative. Others investigated a whole library of plans, which they transformed into new plans (Müller, 2008). They improved sequences of actions to be more efficient depending on the robot's environment. Analyzing plans after execution provides insights about how the transformed plans have improved the robot's actions. Still, transforming plans after execution is scarcely pursued since it requires an already stable set of flexible plans used to design scenarios executable by robots. By exploiting broad execution traces it enables extensive reasoning possibilities and equivalently powerful transformation mechanisms.

1.1 General Approach and Research Questions

During my work at the Institute for Artificial Intelligence Professor Beetz raised my interest in plan transformation and I wondered, if similar mechanics would allow transformation of CRAM plans as well. CRAM provides a physics simulator which allows collision detection and visualization of plans during execution, I can use the descriptive arguments to construct self-recovering plans, which maintain stability within plans, and there is a powerful logging mechanism, which allows reasoning on the plans after execution. Using these tools I want to investigate to what extent I can apply the ideas from (Müller, 2008) in the CRAM context.

1.1.1 Plans

In plan based robotics a plan represents a robot's action in an abstract way. It contains a sequence of descriptions, that are evaluated to other plans. Furthermore, CRAM plans can be stabilized by handling failures of their underlying actions, making them more flexible.

```
1  (def-cram-function navigate (?location-designator)
2    (with-retry-counters ((nav-retries 3))
3      (with-failure-handling
4          ((navigation-low-level-failure (e)
5             (do-retry nav-retries
6               (retry))))
7        (perform
8         (a motion (type going) (target ?location-designator)))
9        (on-event
10        (make-instance 'cram-plan-occasions-events:robot-state-changed)))))
```

This is a plan for navigating to a specific location. In lines 7-8 the robot executes a navigation to the desired target, but if this movement fails, it is caught in line 4. Lines 5-6 will be executed when navigation fails and retries the movement up to 3 times, because a retry counter is set in line 2. This is just a simple example and the failure handling body can be filled with other actions, for instance, trying to stabilize the robot's state in the environment.

1.1.2 Plan Transformation

Transformational planning allows to improve an action's plan, by analyzing what the robot did, and how. Using plan transformation on existing plans lets a robot improve his actions by himself. I want to test this in different scenarios of pick and place tasks. The vision is, that a generic plan, designed to be suitable for execution in various scenarios, can be tailored to perform better in specific environments. Analyzing what the robot did, searching for specific patterns in the logging data, reveals in which ways the plans' structure can be improved. Partial improvements of the plan are written in transformation rules, which take the data from the analysis to add, remove or change a set of plans.

Müller says, that transformation rules and pattern analysis can be generalized in the following way:

$$\frac{\text{applicability}}{\begin{array}{c}\text{input schema}\\ \text{transformation}\\ \downarrow\\ \text{output plan}\end{array}}$$

Applicability checks if a transformation is suitable for the plan at hand, *input schema* is what the transformations' input parameters are, the *transformation* accounts for altering the plan, returning the *output plan* eventually. Following this pattern I want to implement algorithms to check the former two, and apply the *transformation* on different pick and place scenarios. Later the transformations are evaluated to see, how well they improve the original plans.

1.2 Contributions

This thesis concentrates on enhancing human-written tasks for autonomous robots, narrowed down to fetching and delivering objects in a household environment. I want to find out in which ways such pick and place tasks can be modified to increase the robots performance and still achieving the same goal given by the writer. I will provide an overview on how to design and use transformations on the extensive logging trace of CRAM. For this logging trace I will design general predicates to determine applicability of transformations, to show how to traverse huge data sets, as has been an obstacle in previous researches on plan transformation after execution. To overcome this hurdle I will implement an algorithm to minimize resources while searching for patterns, suitable for transformations. The impact of transformation is determined by evaluating the outcome of transformation rules on pick and place scenarios, using the enhanced reasoning mechanism for gathering interesting data from the simulation. For further research I will implement a generic framework, that can easily include transformations and reasoning predicates into the process of testing plans and possible transformations.

1.3 Related Work

Multiple research groups investigated the possibilities of transformational planning but most of them observed only changing execution for failure handling and avoidance. Introducing the Reactive Planning Language (RPL) (Mcdermott, 1993) and sophisticating reasoning techniques via Prolog, as well as using terms consisting

of symbolic and subsymbolic descriptions, plan transformation gained even more powerful possibilities.

The *TRANER* (TRAnsformational PlanNER for Everyday Activity) introduced by Armin Müller (Müller, 2008) is amongst the broader approaches on transformational planning towards offline plan improvement. Using RPL (Reactive Plan Language) for plan construction and transformation he operates on flexible and reliable plans to make autonomous robots achieve successful action execution in every-day tasks like household chores. With the inference techniques of CRAM (Cognitive Robot Abstract Machine) (Mösenlechner, 2016) the design of plans improved. The CRAM architecture includes tools to ensure reliability, stability and flexibility, as well as the CRAM Prolog predicates for execution trace analysis. Müller's theories about transformational planning and Mösenlechner's architecture are the main accomplishments this thesis is oriented at.

TRANER is most influenced by McDermott's XFRM planner (McDermott, 1992) which is based on transformational improvement of an agent in the grid world. In Michael Beetz' theories on transformational planning (Beetz, 1992) you can see further development through RPL and reasoning over semantic relationships. Using McDermott's foundation, Michael Beetz (Beetz and McDermott, 1997; Beetz, 2000; Beetz, 2001) applies transformational planning techniques on autonomous robots in every-day activities. Also using Müller's TRANER, Mortz Tenorth (Tenorth and Beetz, 2010) demonstrates optimization of pick and place tasks in a kitchen environment.

Another example of transformational planning for mobile manipulation is given by Fedrizzi and Mösenlechner (Fedrizzi et al., 2009). They use *ARP*LACE which resolves descriptions of locations, rather than using fixed poses, to argue about spacial navigation of the robot. Those descriptions were further developed by Mösenlechner into location designators (Mösenlechner, 2016), being used for decision making in transformation rules in this thesis. Reasoning about execution traces was discussed by Demmel and Mösenlechner as well (Mösenlechner, Demmel, and Beetz, 2010) before Mösenlechner enhanced and broadened it for his PHD thesis (Mösenlechner, 2016). Alongside descriptive locations Mösenlechner introduced other *designators* in his work (Mösenlechner, 2016), which are in constant development and enhanced by Gayane Kazhoyan (Kazhoyan and Beetz, 2017).

Sussman's *Hacker* (Sussman, 1973; Sussman, 1975) concentrates on creating and transforming reliable plans to achieve a specific goal using lower-level plans, which are eventually resolved to atomic actions. My simulation uses the bullet physics engine (Mösenlechner and Beetz, 2011; Mösenlechner and Beetz, 2013) to find flaws in decision-making and possible errors in manipulation, which seems closer to real world simulation than the Hacker system.

There are more transformational planners like Chef (Hammond, 1990) and Gordious (Simmons, 1988). Chef and Gordious create sequential plans and mainly cover failure handling. The CRAM execution trace on the other hand contains a hierarchical, easily traversable task tree. Also the transformations presented in this thesis attack optimization in movement of already reliable plans rather that failure handling or avoidance, since most of failure handling is done in CRAM plans already. More of the type, by using hierarchical structures, is the work of Bothelho and Alami (Bothelho and Alami, 2000), in which they explain how to enhance plans in hierarchical, partial order.

An interesting research on reactive, opportunistic action selection is done by Kruse and Kirsch (Kruse and Kirsch, 2010), where the decision to execute an action

depends on the detection of opportunities. If an opportunity fits the preconditions of a plan to execute, it may be executed. Their work is based on reasoning through RPL and decision-making is processed during runtime, which differs from my approach in offline reasoning and transformation, but is similar to preconditions of applying a transformation rule.

Following the same idea, (Beetz et al., 2012) execute their plans based on occasions, represented by fluents that describe complex boolean expressions. Their flexible plans are mainly designed for reactive navigation adaptation in *ARPlace* but the reasoning techniques are similar to the ones used in this thesis.

The high-level distributed architecture *HiDDeN* (Gateau, Lesire, and Barbier, 2013) provides reparation of plans by analysis of a hierarchical decomposition of the robots' tasks. Alternative repairing solutions for subtasks need to be defined a prior, which will then replace the previous, malfunctioning subtask. Like in the CRAM execution trace(Mösenlechner, Demmel, and Beetz, 2010) they rely on observing a task tree, but in contrast to their transformations the transformation rules described in this thesis observe a collaboration of tasks at a higher level and apply changes upon multiple subtasks within one transformation.

1.4 Reader's Guide

This thesis describes how plan transformation can be realized in the CRAM architecture, specialized on pick and place tasks. It will explain how to collect the information needed in transformations and how they affect the program's structure.

Chapter 2 gives an overview of the technologies used for implementing plan transformation. It contains basic knowledge about ROS, CRAM and its components, as well as the CRAM functionality for plan transformation before and after execution.

Chapter 3 explains my implementation of CRAM specific plan transformation, reasoning over an extensive log using a declarative, logical language. Furthermore, it show additional implementations and difficulties to overcome, in order to achieve sophisticated plan transformation.

Chapter 4 analyzes in which ways the transformations explained in Chapter 3 have impact on pick and place scenarios and to what extent they improved or worsened execution of the scenarios.

Chapter 5 concludes what insight has been gained while implementing and evaluating the transformations. Also it discusses future work on transformations in the CRAM architecture.

Chapter 2

Foundations

In this chapter I want to talk about the Robot Operatin System (ROS), the Cognitive Robot Abstract Machine (CRAM) published in (Mösenlechner, 2016), and some ideas on plan transformation from (Müller, 2008). The CRAM architecture implements symbolic, discriptive plan design and failure handling mechanisms and Prolog reasoning techniques to retrieve information from extensive logging traces and knowledgebases. A particular log will be seved by the CRAM execution trace, containing most of the information about executed plans in a scenario, represented as an offline structure. Within this execution trace I operate on a hierarchical represantation of all plans having bee executed in the scenario under inverstigation. Furthermore, I will introduce the simulation environment (bullet world) where the execution of plans will be visualized and tested.

2.1 ROS

The Robot Operating System (ROS[1]) (Quigley et al., 2009) is a dynamic framework for writing programs for robots. It enables communication between multiple, exchangable nodes where each has a specific task to do and in collaboration can achieve a greater goal. The main idea for ROS was to distribute the implementation of robust robot programs. Using ROS gives the ability to outsource programs into seperate processes. ROS is used in this thesis for the simulation environment, representing the robot and kitchen and further reasoning frameworks like the semantic map.

2.2 CRAM

CRAM (Cognitive Robot Abstract Machine) is a computational, cognitive framework written in Lisp. It combines a high-level descriptive language for action planning of autonomous robots and sophisticated reasoning mechanisms. This provides the programmer the ability to write plans in an absract way, without concern about the underlying hardware components and sensors, nor the knowledge about discrete information in the environment.
Lacking, or ignoring this low-level information, the programmer can use CRAM prolog reasoning queries to get insight into *What does the robot know?*, *Does he see that object?* or *Why did he decide to do this?*. This kind of reasoning can be used at runtime as well as after execution. Reasoning after execution is called *offline reasoning*. Executing plans creates an execution trace, which is like an extensive log of all the memory the robot has about the actions he executed previously.
The bullet world projection environment is a simulator in which the robot's movements and cognitive processes can be visualized (figure 2.1) and tested. It provides

[1] http://www.ros.org/about-ros/

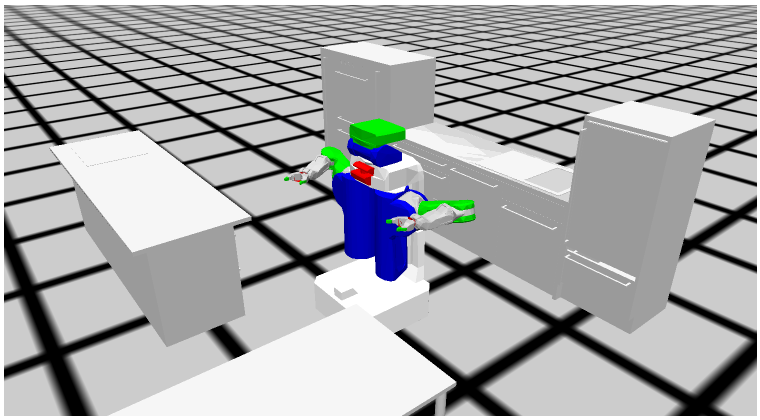

FIGURE 2.1: Robot and kitchen simulated in the bullet environment.

FIGURE 2.2: The kitchen and PR2 robot in the Institute for Artificial Intelligence Bremen.

the programmer with convenient tools to spawn and move objects, manipulation of robots and the environment, as well as gravitational simulation and reasoning about collisions.

2.2.1 Projection Environment

The bullet environment is a multi-purpose lightweight simulator. A Lisp wrapper for the *Bullet* simulator[2] was developed in (Mösenlechner and Beetz, 2011; Mösenlechner and Beetz, 2013) to provide visualization of CRAM plans during their execution. The wrapper allows us to spawn customized meshes of objects, and robots in a semantic format or Unified Robot Description Format (URDF). I use this to set up the kitchen environment, the PR2 robot and design scenes consisting of spawned objects, placed at various locations, depending on the tasks the robot is told to perform.
What is the most important feature of the simulation is that I can temporally try out a variety of motions in an instant to check, if it would lead to unwanted behaviour. Via an inverse kinematic (IK) solver I can decide to keep or reject the robots

[2] https://pybullet.org/wordpress/

movement if it would lead to an undesired outcome. Taking the action of placing an object as an example, an undesired outcome would be that the robot collides with the environment or the movement, when releasing the object and simulating gravity afterwards, would result in an unstable state of the object or environment. Such conditions can be reasoned upon immediatly and the motion revoked, when rendered malicious, usually retried with a slight alteration in movement. Being more specfic, the movement is executed with a different solution from the IK solver. Retrying an action with slight adjustments is also implemented within self recovering plans, which are explained in section 2.2.4. For a detailed hands-on tutorial for manipulating the CRAM bullet world, the IAI provides insightful tutorials[3]. In this section I will only mention some specific tools more frequently used in this thesis. The packages *btr* and *btr-utils* (nickname for *cram-bullet-reasoning*) provide all the necessary functionality to comprehend and manipulate the simulated world programatically. To retreive information about the world I can inspect the *btr:*current-bullet-world** object.
Since pick and place tasks concentrate on the transition of objects I need to gain access to their simulated representation. I can do this with *(btr:object btr:*current-bullet-world* object-name)* where *object-name* is to be provided as a symbol. I use keywords, case-insensitive symbols, for object names throughout the whole implementation to encourage absolutely distiguishable terminology.
Simulated objects always contain a name, type, the world they are in and a rigid body. The rigid body contains the bounding box of the objects mesh. Deeper details, like poses of bounding boxes, can only be investigated through the C-interface, which makes debugging complicated but protects the programmer from getting lost in unnecessary details about the bullet projection environment. The *btr* package does not only provide us most of the functionality I need, it also returns all information almost[4] in first class representation. Changing an objects parameter is immediately resolved in the simulation. Alongside the Lisp REPL (Read-Eval-Print-Loop), having this kind of a responsive simulation gives programmers the ability to rapidly prototype new plans.

2.2.2 Designators

Designators are at the core of abstract plan construction. They consist of a symbolic and subsymbolic description of what you want to express, be it a locations, actions or objects. The symbols used within a designator are then resolved by the prolog engine into plans, which contain other designators. This lets us design command descriptions for the robot in a hierarcical way, building up a tree of plans which can be reasoned about, again with Prolog (see 2.2.5). When costructing a designator keep in mind, that they are resolved by the CRAM prolog module, which supports most of SWI-Prologs syntax. Value holding variables used within designator instantiation must be prefixed with a question mark (*?*) to let the prolog interpreter know, that it has to handle the variable's value instead of using it as a symbol, more specific, a keyword. Lets look into the different kinds of designators that will be of use for us.

Location Designators are descriptions of a point, an area or a somehow relative spacial position. You can describe the location of a pose like this

[3] `http://cram-system.org/tutorials/advanced/bullet_world`

[4] Lisp getter-functions like *pose* do not actually read slot values of an object but request them from the C-Interface instead. Changing object slot-values is also wrapped to eventually interact with the low-level interface, nervertheless it feels like working directly on the simulated object.

```
(a location
   (pose ?my-pose))
```

where *?my-pose* is the variable name of a *cl-tf:pose-stamped* object. *cl-tf* is the Lisp package that contains all pose transformation logic I use throughout the implementation. To create a location designator evaluated to the point (1 1 1) and identity rotation (0 0 0 1) you need to first create the pose-stamped object. Since I use stamped poses instead of normal poses, the constructor needs an origin frame, to which the pose is relative to, and a timestamp, which would normally be set when the pose is created during plan execution.

```
(let ((?my-pose (cl-tf:make-pose-stamped
                 "map" 0.0
                 (cl-tf:make-3d-vector 1.0 1.0 1.0)
                 (cl-tf:make-identity-rotation))))
     (a location
        (pose ?my-pose)))
```

You see that, even if designators are written in descriptive symbolic notation, they can be hard-coded, although straight-forward declarations like these are not very welcome in the dynamic plans I want to have. To make it more dynamic I can assign a location to the origin of an object.

```
(a location
   (obj (an object
            (type :cup))))
```

Since I use another designator within the location designator, the object designator needs to be resolved before the location can point to a descrete pose. As soon as the described object is percieved, the location can be resolved to the objects position.

Another form of location can be described using areas. This form uses names of components from the environment to create a set of possible locations that each fit to the described components location.

```
(a location
   (on "CounterTop")
   (name "iai_kitchen_sink_area_counter_top"))
```

Here I want to create an area of possible locations all fit to the description, namely the surface of the sink area in the kitchen.

Object Designators cover the description of objects in the world. As already shown in an example for location designators, an object can simply consist of its type.

```
(an object
    (type :cup))
```

For the manipulation of an object this description does completely suffice my needs. By perceiving any object of the given type the bullet world resolves the object designator, given that an object of this type is found. The object designator then gets a name, stamped pose, color etc. which can be used for further manipulation and reasoning. Through the course of execution time the pose assigned to the object stays the same at this specific timestamp. If the object is moved, the designator changes its pose value, adding timestamps, which allows us to reason about the position of objects at various times.

Action Designators can describe any series of movement the robot may be able to execute. The latest design of action designators can be seen in (Kazhoyan and Beetz, 2017). To perceive an object, for example, I will use the following action

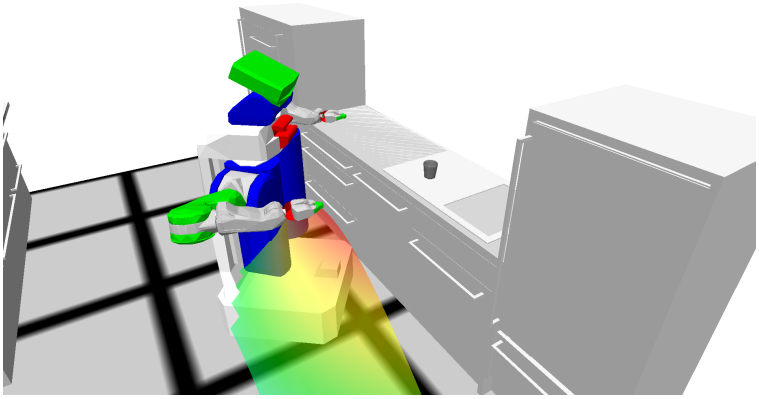

FIGURE 2.3: The robot looks at the cup after searching for it.

```
(an action
    (type detecting)
    (object (an object
                (type :cup))))
```

which tries to find an object of type *:cup* in the robots current field of vision. When this action is successful it returns the resolved object with subsymbolic values. Before I can see an object I first need to navigate into the right position with

```
(an action
    (type navigating)
    (location ?loc))
```

where *?loc* again is a location designator. Combining these two actions and the one to move the head, which is not worth mentioning for my purposes, you can describe a more sophisticated plan that searches for an object in the robots environment.

```
(desig:an action
          (type searching)
          (object (an object
                      (type :cup)))
          (location (a location
                       (on "CounterTop")
                       (name "iai_kitchen_sink_area_counter_top")))))
```

Searching for an object requires the objects description and a probable location to look for it. After executing the searching action the robot should be in the state shown in figure 2.3. You can see the generated area of locations for navigating near the location provided, a gaussian distributed area of promising locations to navigate to. Having found the desired object the robot can now pick it up with his grippers. In combination with the searching action I have an action of type *fetching* in my repertoire, that uses the same parameters as the searching action.

```
(an action
    (type fetching)
    (object ?obj)
    (location ?loc))
```

Fetching an object consists of navigating to, looking at and detecting the object as well as moving the arms in a way, that the gripper at the end of an arm can securely grasp the object while preventing collision with the surface and other obstacles. The result of this action is shown in figure 2.4 How such a plan is designed in detail will be explained in section 2.2.4.

Now with the object in out gripper I can deliver it to an other location. Delivering an

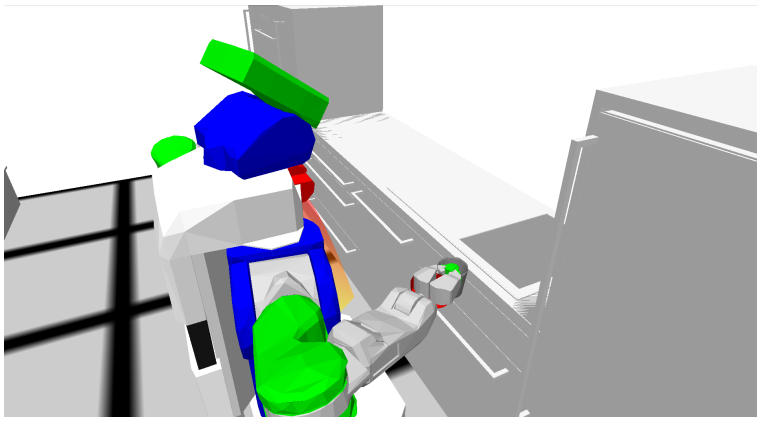

FIGURE 2.4: The robot fetches the cup from the table with his right arm.

object contains navigating to the depot location and putting the object down, again without causing any collision with the environment or other objects.

```
1 (an action
2     (type delivering)
3     (object ?obj)
4     (target ?loc))
```

All those actions combined can be wrapped up in a transportation action that first tries to fetch an object to then deliver it onto the given location.

```
1 (an action
2     (type transporting)
3     (object ?obj)
4     (location ?fetching-loc)
5     (target ?delivering-loc))
```

Until now you have seen all important actions to write simple pick and place plans. Simply moving the robots base and manipulating his joints does not make a plan yet, I need some mechanisms for recovering the robots state if he fails to execute a task properly. As can be seen the commands are written in a way of controlling a robot without any definition of robot-specific code.

2.2.3 Process modules and atomic plans

Resolving action designators to plans, those to action designators and again to plans eventually comes to an end, when a designator describes an atomic motion. Those motions will only further be resolved to process modules. Motion designators being resolved to process modules is the border between descriptive planning and robot specific execution of code. Everything above is completely independent of the underlying hardware, hence process modules can be seen as an interface between planning and hardware related code execution.
Changing the plans and related actions above can be executed regardless of what is beyond process modules, as long as the process modules resolve to useful code. Due to this border I can freely create, change and revise plans in the above structures. All the designators and plans explained in section 2.2.2 have been tested in real world scenarios, using the PR2 robot in the kitchen environment of the Institute for Artificial Intelligence Bremen. Exchanging just the lowest level atomic plans, that are usually used in the live demo, with functions executed in a simulation, I can test the higher level plans in an environment close to the real world robot.

2.2.4 Self-Recovering Plans

Self recovering plans have been an important field in transformational planning for a while now (Hammond, 1990; McDermott, 1992; Beetz and McDermott, 1997; Liberatore, 1998; Müller, 2008; Gateau, Lesire, and Barbier, 2013). In CRAM plan recovery behaves like catching and handling an exception in other programming structures, only that in CRAM plans it is the failure of a plan that is caught and handling a failure consists of actions that try to stabilize the robot's and environment's situation. This recovery prepares the robot for retrying the action that previously failed. In a way this recovery mechanism can already be called plan transformation, since the execution of such a plan generates non-deterministic behavior and changes its sequence of actions by adding and changing tasks during execution. Although the course of additional action may differ in each execution, the plan itself stays the same. This means, when a plan is executed successfully, the outcome of the plan would leave the environment always in the same state, for example, an action for transporting the object *O* from location *A* to *B*, at the end of the action the object *O* should be at location *B*, regardless of how often or in which ways the plan has been recovered, provided the plan terminates successfully after all.
Besides catching and handling a failure, I can set how many times a plan should be re-executed after failure handling to prevent the scene from life-locking. The macro for using retries is called *cpl:with-retry-counters ((counter n))*. Catching failures is done with the macro *cpl:with-failure-handling ((failure-class (e))) &body)*, where the plan(s) to catch failures from is executed in the *&body*. Let's take a reduced version of the searching plan as an example. The upper half implements handling failures, at the bottom you can see the plans executed.

```
(cpl:def-cram-function searching-for (?object-designator
                                      ?search-location)

  (cpl:with-retry-counters ((attempts 5))
    (cpl:with-failure-handling
      ;; BEGIN handling failure.
      ((common-fail:object-not-found-object (e)
        ;; Print error.
        (roslisp:ros-warn (pp-plans search-for-object) "~e" e)
        ;; Retry if attempty left.
        (cpl:do-retry attempts
          ;; Try next searching location.
          (handler-case
            ;; Check next navigation solution.
            (setf ?search-location (desig:next-solution ?search-location))
            ;; If none available throw error to signal higher level plan.
            (desig:designator-error ()
              (roslisp:ros-warn (pp-plans search-for-object)
                "Designator cannot be resolved: ~a. Propagating up." e)
              (cpl:fail 'common-fail:object-nowhere-to-be-found)))
          (if ?search-location
            ;; Retry search with new location if available.
            (cpl:retry)
            ;; Throw error if no alternative location is found.
            (cpl:fail 'common-fail:object-nowhere-to-be-found))
          ;; If attempts and locations are out, throw error.
          (cpl:fail 'common-fail:object-nowhere-to-be-found))))
      ;; END handling failure.

      ;; BEGIN actual plan execution.
      ;; Prepare (new) search location from failure handling.
      (let* ((?pose-search-loc (desig:reference ?search-location))
             (?nav-location
               (desig:a location
                        (visible-for pr2)
                        (location (desig:a location
                                           (pose ?pose-search-loc))))))
```

```
38          ;; Navigate to (new) location.
39          (exe:perform (desig:an action
40                                  (type navigating)
41                                  (location ?nav-location)))
42          ;; Look at (new) target pose.
43          (exe:perform
44            (desig:an action
45                      (type looking)
46                      (target (desig:a location
47                                       (pose ?pose-search-loc))))))
48
49        ;; Detecting the desired object.
50        (exe:perform (desig:an action
51                                (type detecting)
52                                (object ?object-designator)))
53        ;; END actual plan execution
54        )))
```

The lines 4 to 28 implement handling the failure *object-not-found*, while 30 to 52 contain the actual plan. First I want to execute the plan, in which I navigate to a location (39-41), where the PR2 robot can see the provided search location. After navigating the robot should look at this searching-pose (43-47), where the robot tries to detect any object described in *?object-designator* (50-52). Navigating and looking are separate plans, containing failure handling as well. But what if no such object could be found? Within detecting an object a failure will be thrown, which is caught in the upper section of my searching plan.
To recover from this failure, I request the next solution of my *?search-location* designator (15), an alternative position and point of attention should be gained. If there is any (13 & 21-26), try the plan again (23), but only if it did not exceed the retry counter yet (4 & 11 & 27).

Besides handling failures it is important for self-recovering plans to be as independent of other plans as possible. Using plans within plans requires a certain amount of trust in their reliability, which is provided by handling failures and recovery mechanism. Looking at an object, for example, sometimes requires opening drawers and other containers, clearing occlusions by other objects etc. The more careful those actions are designed, the greater their independence and reliability. Independence is among the most important prerequisites when reasoning over and transforming plans in a larger scale, as will be discussed later in this thesis.

2.2.5 CRAM Prolog Reasoning

Prolog is a programming language designed for querying knowledge bases which contain inference rules. CRAM Prolog is a primitive Prolog interpreter written in Common-Lisp, inspired by the SWI-Prolog dialect. It was implemented to gain the abilities of Prolog in the CRAM context, to be able to reason about knowledge bases most commonly provided by KnowRob (Tenorth and Beetz, 2009) knowledge representations. Using CRAM Prolog for parsing semantic contexts lets us reason about my extensive log, the execution trace (2.2.6), and operate the projection environment (2.2.1). Querying information via Prolog always returns a lazy associative list of solutions that fit the requests predicate, hence locations resolved from designators might very well be an infinite list of possible locations that all match the description provided. A lazy list is like a normal list, but does only compute every element as soon it is needed. In some cases I only need the first element of a list, and to avoid unnecessary computation, I can easily get the head of such a list without calculating

each element. I can use the Lisp command *car* or the CRAM Prolog command *lazy-car* to access the head of a lazy list, or compute the next element with *lazy-cdr*. The Lisp command *cdr* usually returns the tail of a list, namely the whole list without the first element. I can also force the a lazy list to calculate and return all elements with the Prolog command *force-ll*. How predicates look like in detail are explained in their respective sections.

2.2.6 Execution Traces and the CRAM Task Tree

The CRAM execution trace can be seen as a large scale logging framework, containing monitored data in first class representation. It contains the CRAM task tree, a hierarchical structure of all executed plans and their subplans, including the parameters used during execution, their current status and outcome. Per definition of trees the task tree is free of directed or undirected cycles. Each node has only one parent where parent and node must differ. Also each node can be reached from the root node.
Online traces enable programmers to read and adjust plans during execution. Investigation of failures after execution can be done by traversing the task tree and checking what the robot was thinking when executing a specific task. To construct such a task tree, plans must be executed from within a named cram-top-level plan[5].

```
1 (cpl:def-top-level-cram-function my-top-level-plan ()
2   (exe:perform some-action)
3   (exe:perform some-other-action))
```

Here *my-top-level-plan* will be the root node of the task tree, *some-action* and *some-other-action* are resolved to *cram-plans*, being direct children of the root plan. The runtime object representing an executable plan is called a *task*. Executing the top-level task generates the task tree, attaching subplans as a child to the higher-level plan, spanning up the previously mentioned task hierarchy. Tasks contain a significant amount of information:

```
1 (defstruct task-tree-node
2   (code nil)
3   (code-replacements (list))
4   (parent nil)
5   (children nil)
6   (path nil)
7   (lock (sb-thread:make-mutex)))
```

Here you can see that a task-tree node contains not only a list of children, but also a link to its parent. The path makes it easy to find a specific task in the tree. The lock slot determines in which state the task currently is. For representing the code of a task there is another struct:

```
1 (defstruct code
2   sexp
3   function
4   task
5   parameters)
```

This struct contains an S-expression (symbolic expression) of the plan executed. The function is the reference to the actual Lisp function. The task slot contains the function and parameters, as well as further, negligible information.
So far the *code-replacement* slot of a task-tree node has not been mentioned, for this is the most important feature for plan transformation. This slot can be filled with a list of function references, which are executed instead of the original function. When

[5] You can run anonymous top-level plans as well, but reasoning over the trace afterwards is not recommended if any possible, and transformation of such a task-tree not purposeful at all.

executing the top-level plan with his task-tree already built, each task-tree node is checked, whether the *code-replacements* slot is filled, and if so, the contained function reference is executed instead of the original tasks code, while still preserving all information of the old task execution. If a programmer would clear a tasks *code-replacements* slot the original plan represented by this task is executed again, without any trace of the replacements. Adding replacements to nodes do not even change their path or add new tasks to the task-tree. Issues related to this beaviour will be discussed in section 5.2.
To gain access to the task-tree I use the *cpl* and *cpl-impl* package (*cram-language*, *cram-language-implementation*).

```
1 (cpl:get-top-level-task-tree 'my-top-level-plan)
```

Through the SBCL inspector I can investigate the retrieved object and traverse the task-tree by hand. But there are way better tools that fit my purpose. To identify specific tasks in the task tree, (Mösenlechner, 2016) implemented some useful predicates. They were designed to detect tasks and their designator's properties (see section 2.2.2) in the CRAM task tree. Via CRAM Prolog I can reason about the content of the task tree without having to traverse it all the way. Lets look at the following top-level plan, transporting three objects from one location to the pose (1 1 1) with identity rotation[6].

```
1  (def-top-level-cram-function my-top-level ()
2    (dolist (?object-type '(:breakfast-cereal :milk :cup))
3      (exe:perform (an action
4                       (type transporting)
5                       (object
6                         (an object
7                             (type ?object-type)))
8                       (location
9                         (a location
10                           (on "CounterTop")
11                           (name "iai_kitchen_sink_area_counter_top")))
12                      (target
13                        (a location
14                           (pose (cl-tf:make-pose-stamped
15                                  "map" 0.0
16                                  (cl-tf:make-3d-vector 1 1 1)
17                                  (cl-tf:make-quaternion 0 0 0 1)))))))))
```

Executing this code should result in the following task tree in figure 2.5. In figure 2.5 you can see that the task tree's root has but one child, which represents the top level plan. The top level plan has three subtasks, the transporting actions of the three items of type *:cereal-box*, *milk* and *cup*. In the code segment of the last subtask, transporting the cup, the corresponding action can be found as parameter of the code struct. The target pose is already resolved. The path for each subtask is unique, even if the designators called only differ in the object type. To distinguish them from another, a *:call n* suffix is added to the path of each siblings of similar S-expressions. I can now use this task tree for further examples on reasoning and code replacement. Calling a Prolog predicate evaluates the predicates condition on the task-tree and returns the first solution as soon as found, while the rest can be retrieved lazily. To get all subtasks of the root node I first need to get the specific node, then access its subtasks

```
1 (prolog '(and (top-level-task my-top-level-plan ?root-task)
2               (subtask ?root-task ?sub-task)))
```

As explained in 2.2.5 a lazy list computes each element as soon as needed, while the first result is returned right away. I can extract the first element of a lazy list with

[6]The pose should differ between the objects, but this is just an example.

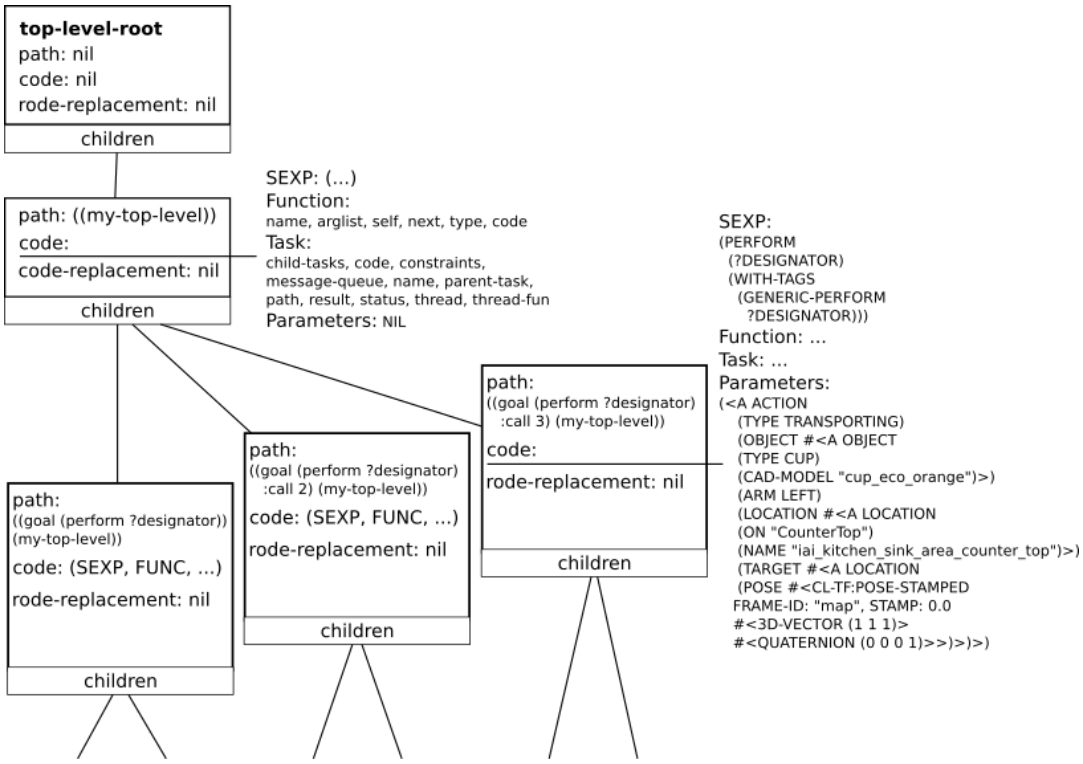

FIGURE 2.5: An example of the task tree transporting three items.

the command *car* and calculate the next element with *lazy-cdr*. The head of this particular lazy list contains two pairs: *?root-task* is the car (first element) of the first pair and *?sub-task* the car (first element) of the second pair. The cdr (second element) of the first pair is the root task of the task tree and the cdr of the second pair is the first solution of a subtask found.

```
(((?root-task . <top-level-root-task>)
  (?sub-task . <my-to-level-task-object>))
 <lazy-cdr-object>)
```

I can now simply car this entry, if I only need one solution, or, for example, use lazy-cdr to maintain lazyness and get the next solution, as well as force the list to evaluate every possible solution. Force-calculating every solution might require huge resources, hence I try to avoid that. Another example searches the task-tree for a task, whose plans designator is of the specified action-type.

```
1 (task-specific-action ?top-level-name ?subtree-path
2                       ?action-type ?task ?designator)
```

This predicate returns an associative list of tasks and designators, where the designator has the same properties as when it was called within the tasks plan. I only search in the task-tree given by *?top-level-name*, unter the path *?subtree-path* for an action of type *?action-type*. An example could look like this:

```
1 (prolog '(task-specific-action my-top-level-plan ((my-top-level))
2                                :transporting ?task ?desig)))
```

Here I get the rightmost transporting task from figure 2.5 as head of the lazy list. In *?task* the actual task is stored, in *?desig* I get the action designator stored in the parameter slot of the code struct.
Calculating all possible tasks that suffice this condition leaves the predicate to traverse the whole task tree below the subtree-path given. These predicates are just

TABLE 2.1: CRAM Prolog predicates for traversing the task tree.

Predicate	**Result**
top-level-task (?tl-name ?tl-node)	Root node of task tree named ?tl-name.
task-full-path (?node ?path)	Path of given node.
task (?tl-name ?path ?node)	All tasks under ?path.
subtask (?task ?subtask)	All direct subtasks of ?task.
subtask+ (?task ?subtask)	All subtasks of ?task.
task-sibling (?task ?sibling)	All tasks that share the same parent with ?task.
task-parameter (?task ?parameter)	Argument (designator) the ?task was called with.
task-specific-action (?tl-name ?path ?action-type ?task ?designator)	Task and designator of type ?action-type under ?path.

some of those I need to design the transformation rules. The most important predicates used are listed in table 2.1.

2.3 Transformational Planning

Basically, transformational planning tries to change the course of a seemingly sequential execution to achieve a better performance overall. Transforming plans themself or their hierarchy in general needs some guidance; rules must be designed, investigated if they are applicable. The rule must consist of a concrete idea about how it changes the code. My current approach is mostly influenced by Müller's offline plan improvement (Müller, 2008) and the cognition and predicate based rules from (Beetz et al., 2012) and (Mösenlechner, 2016). Müller suggests a simple structure for transformation rules. If it is reasonable to apply a transformation rule will be determined by its *applicability*. An *input schema* contains the set of plans to be transformed. By applying the *transformation* form upon the *input schema* I get the transformed *output plan*.

$$\frac{\text{applicability}}{\begin{array}{c}\text{input schema}\\ \text{transformation}\\ \downarrow\\ \text{output plan}\end{array}}$$

Müller applies his transformation upon plans in his plan library, adding the newly transformed plans and increasing the library with every iteration of transformations. An overview of Müller's TRANER structure can be seen in figure 2.6. For all the plans in the library he can execute, evaluate, transform and re-execute as he will, revoking malfunctioning plans and keeping the well-functioning ones.
In my implementation plans are not stored in a library, but can be defined at various locations and distributed over multiple projects, without compromising its functionality. Like in Müller's structure I execute plans implicitly through higher level plans, creating an execution trace (2.2.6), which can be evaluated via Prolog. Executing the right predicates in CRAM Prolog lets us reason about a rules applicability and provides an input schema for the transformation. An example transformation could be to rearrange the order in which multiple objects are acted on as in figure 2.7. This transformation takes two plans for transporting an object and switches their execution order in the higher level plan Set table, which describes what the robot should

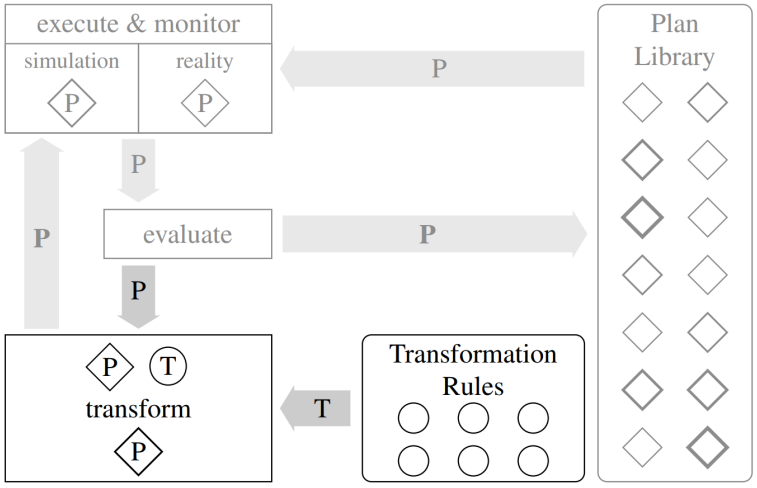

FIGURE 2.6: The basic structure of TRANERs transformation cycle.

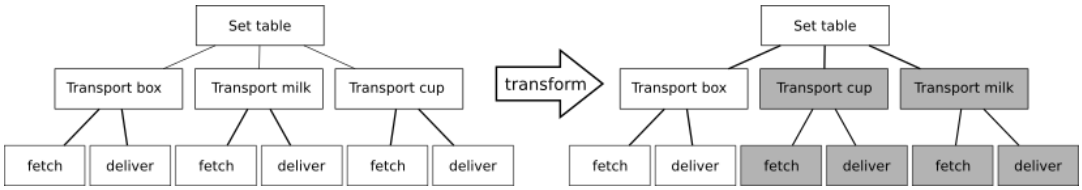

FIGURE 2.7: Example transformation for changing execution order.

do in an abstract way.
Besides regrouping and rearranging actions Müller proposed other transformation patterns, like using containers, where he can stack multiple objects upon another and carry just the bottommost. Another idea is to exploit resources more efficiently. An easy example for this would be to use both grippers in every suitable situation. Talking about pick and place tasks, a robot can save time by carrying two objects at the same time, one in each gripper, to save extra navigation actions.
On the contrary there are situations which are a bit harder to apply general rules upon, e.g. manipulation of the room interior. An open cupboard can spare the opening action when fetching something from inside, but can also prevent the robot from accessing location, which were easily accessible before. Reasoning about this behavior can get difficult with increasing complexity of the scenario.

Implementational constrains of the CRAM task tree prevent us from exchanging tasks by other tasks, for their names are part of their unique path in the tree, as will be discussed in Section 5.2. But still I can apply transformation techniques on the execution trace by using code replacements. To transform the plan like in figure 2.7 and using the transporting plan visualized in figure 2.5 I first need to acquire the nodes to be transformed from the task tree.

```
1 (prolog '(and
2           (task-specific-action my-top-level ((my-top-level))
3                                 :transporting ?first-task ?first-desig)
4           (task-sibling ?first-task ?second-task)
5           (task-parameter ?second-task ?second-desig)
6           (task-full-path ?first-task ?first-path)
7           (task-full-path ?second-task ?second-path)))
```

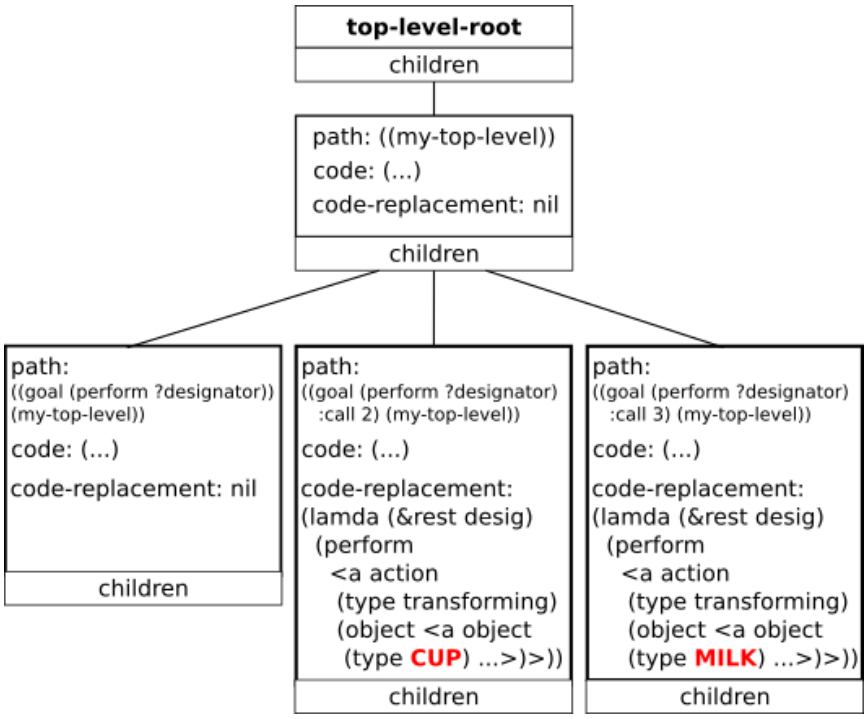

FIGURE 2.8: The task tree after code replacement to switch actions.

Now I have one task-path and its action designator in *?first-path* and *?first-desig*, and the second components in *?second-path* and *?second-desig*. To add a code replacement to a task I can call the *replace-task-code* function. In the following code I swap the action designators between the two tasks.

```
1  (replace-task-code '(TRANSFORMATION-1)
2                     #'(lambda (&rest desig)
3                         (declare (ignore desig))
4                         (perform ?second-desig))
5                     ?first-path
6                     (cpl-impl::get-top-level-task-tree 'my-top-level))
7
8  (replace-task-code '(TRANSFORMATION-2)
9                     #'(lambda (&rest desig)
10                        (declare (ignore desig))
11                        (perform ?first-desig))
12                    ?second-path
13                    (cpl-impl::get-top-level-task-tree 'my-top-level))
```

The first argument should be the S-expression of the function executed, but even if there is bogus in there, it does not affect the replacement. As explained in section 2.2.6 the replaced code, namely the lambda function containing the perform keyword, will be executed instead of the tasks original code. The task tree should now look like in figure 2.8.

Using Müllers and Mösenlechners definitions of a good plan structure, how to reason on the execution trace and how transformation rules should be built, I can begin transforming the execution trace by myself. How far I can go with code replacements will be explained in chapter 3, where I analyze the execution traces of different scenarios, define my own transformation rules and develop Prolog predicates to reason about applicability.

Chapter 3

Methods and Implementation

In the previous chapter I introduced the CRAM architecture, designators, the execution trace, predicates, the bullet projection and plan transformation. Based on those technologies I can design transformation rules and their respective predicates to check applicability. To explain the concepts of this chapter better, I introduce three example scenarios that generate the plans I am going to transform. I will investigate the plans towards how they can be improved afterwards and design predicates to extract the improvable parts from the task tree. In parallel to implementing the predicates I will present their corresponding transformation rules. To make it easier for the next programmer interested in transforming CRAM plans I implemented a generic mechanism to register, prioritize and automatically apply transformation rules on the plan at hand.

3.1 Task tree analysis, Prolog predicates and transformations

A transformation rule contains the manual of how to change a plan. To determine if a transformation is useful for a scenario, I need to check its *applicability* by executing their respective Prolog predicate. Remembering what have been said about transformation rules in Section 2.3 the pattern of a transformations can be simplified to the following structure:

$$\frac{\text{applicability}}{\begin{array}{c}\text{input schema}\\ \text{transformation}\\ \downarrow\\ \text{output plan}\end{array}}$$

By the end of this Section I will have three transformation rules that might be applied in the scope of pick & place tasks. My first rule will change the course of fetch and deliver actions in such ways, that the robot tries to use both his grippers to transport multiple objects at once. The second rule implicates using a tray to carry even more objects at once. Lastly I want to decrease unnecessary environment manipulation, meaning that the robot can leave the container open until having collected all items from within.

3.1.1 Scenarios

I have a total of three scenarios that all include the PR2 robot and the kitchen. Each scenario represents a different sequence of pick and place tasks that all have the common goal of putting a list of objects from various locations in the kitchen onto the table in the very center of the kitchen, called *kitchen island*. Those scenarios are executed and tested in the bullet projection environment. A scenario is defined as a

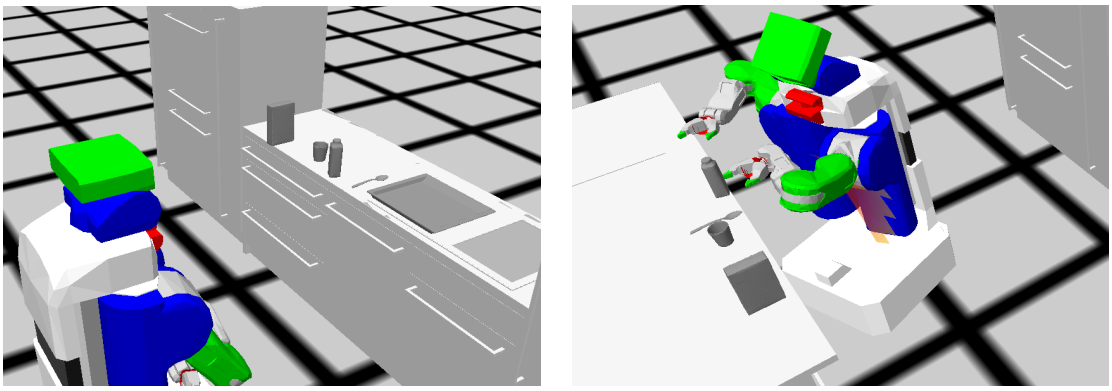

FIGURE 3.1: Scenario 1. Setup for transporting four objects from the sink counter (left) to the kitchen island (right).

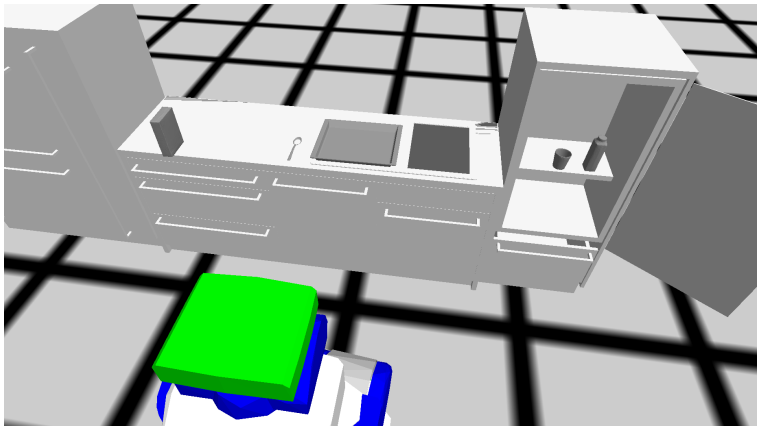

FIGURE 3.2: Scenario 3. Cereal box, spoon and tray are on the sink area, cup and milk are in the fridge (right).

top level plan, containing all the necessary object and location designators used in actions of type *transporting*. Transporting actions include fetching an object from one location and delivering it to another.

My first scenario contains four objects (spoon, cereal box, milk and cup) on the table near the sink (see Figure 3.1), called *sink-area*.

In the second scenario's setup all desired objects are stored in a closed fridge. In the end those objects should be placed on the kitchen island like in scenario 1. Based on the actions of type *transporting*, consisting of the subactions *fetching* and *delivering*, I add two more actions to this process, *accessing-container* and *closing-container*. I combine this set of actions in an action of type *transporting-from-container*. Before transporting an object from within a container, the container must be opened first and closed after this object has been collected (see Figure 3.3).

By combining the main features of scenario 1 and 2 I create a third setup, where two objects are placed on the sink, two more are stored in the fridge. Again the robot needs to collect all four objects from their respective locations. First he acquires the two on the sink area, then he takes care of those in the fridge (see Figure 3.2).

When implementing scenario 2 and 3 I experienced, that collision detection in the bullet environment needs to be enhanced. Each object in the bullet simulation has its bounding box, that fully surrounds the object's mesh. Since I want to fetch objects from within such a bounding box the robot would always detect a collision

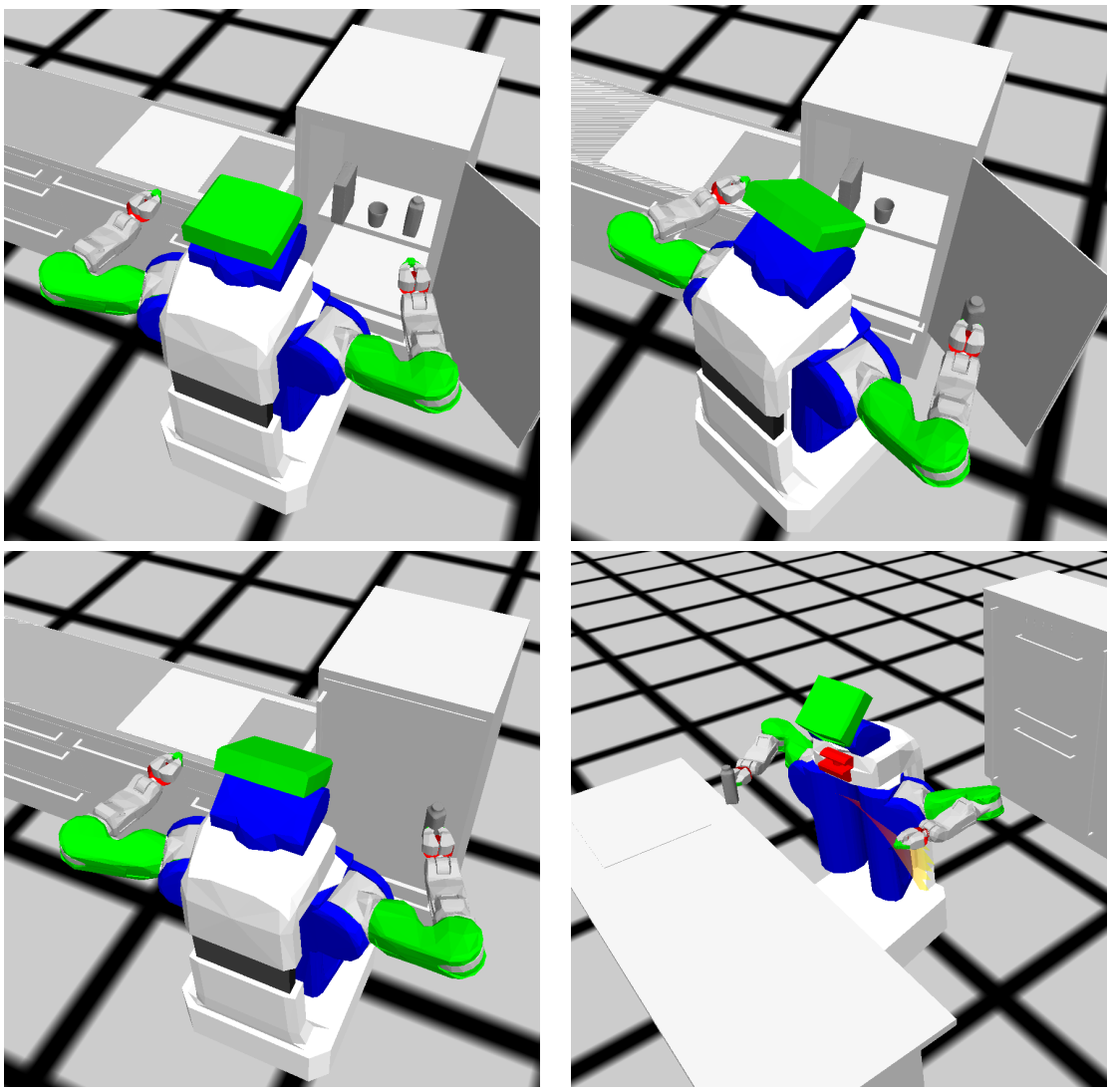

FIGURE 3.3: Scenario 2. (top left) Open the fridge to see three objects inside, (top right) fetch the milk, (bottom left) close the fridge door and (bottom right) deliver the object onto the kitchen island.

with this container, making it impossible to grab something from inside. Unfortunately I did not find a solution to avoid this without altering the fridge meshes, like spawning the fridge's walls separately instead of the fridge as a whole. In order to make movement inside a container possible without falsely detecting a collision, the bounding boxes must be aligned close to the mesh, instead of simply wrapping a bounding box around it. To avoid this problem I implemented a mechanism, that allows collision with the container after opening it. After closing the container I enable collision detection with the container again.

3.1.2 Task tree analysis

The task tree is my main source of information and target of every transformation made. To investigate my possibilities I can look into the task tree of a scenario from Section 3.1.1. In scenario 1 I have four objects to carry from one location to another. The task tree of scenario 1 can be seen in Figure 3.4.

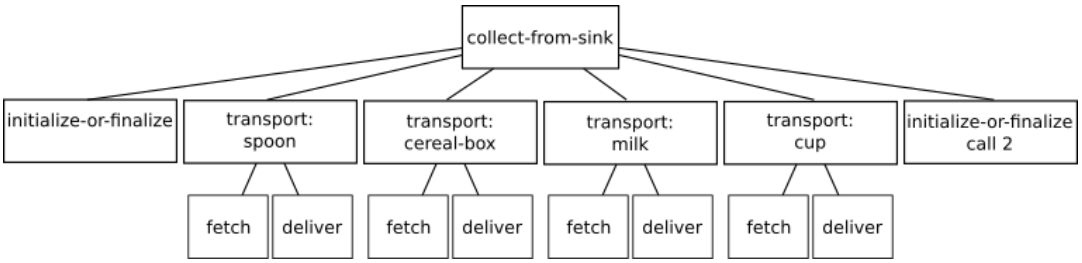

FIGURE 3.4: Scenario 1 task tree, transporting four objects with initializing and finalizing the plan.

The scenario 2 task tree looks similar the one from scenario 1 in Figure 3.4, instead of simply transporting four objects, in the pick and place tasks from scenario 2 I need to open the container, fetch the object from within, close the door and then deliver the object. Therefore, I need to define an additional action type, that I call *transporting-from-container*. You can see the scenario 2 task tree in Figure 3.5.

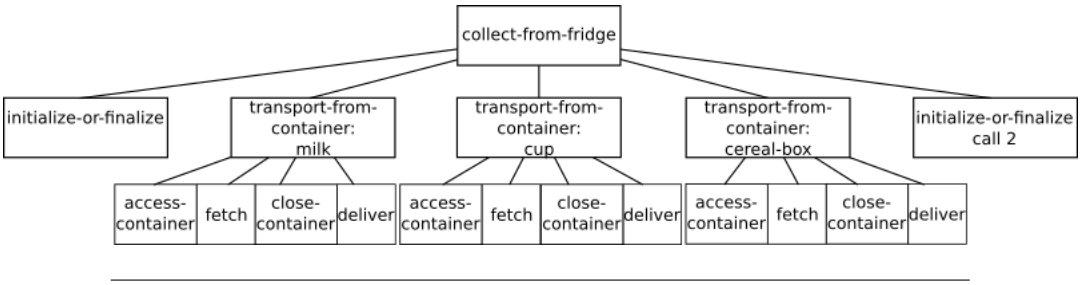

FIGURE 3.5: Scenario 2 task tree,

There is a third scenario combining simple *transporting* actions and *transporting-from-container* actions. I use two actions of each kind: *breakfast-cereal* and *spoon* will be in the sink area, *milk* and *cup* are in the fridge. I first gather the objects from the sink area, then account for those in the fridge (see Figure 3.6).

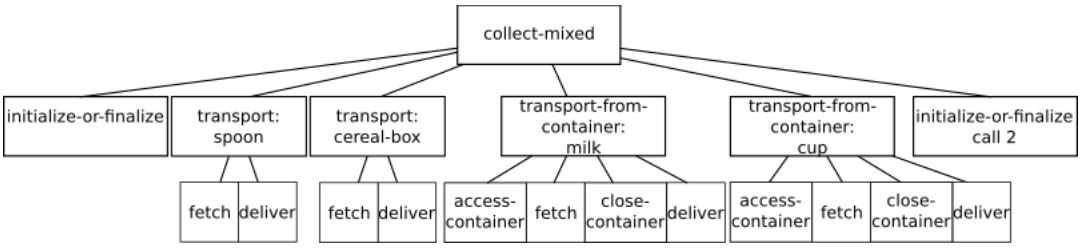

FIGURE 3.6: Scenario 3 task tree,

3.1.3 Transformations

In this Section I concentrate on the *transformation* and *output plan* part, while *applicability* and *input schema* are discussed in Section 3.1.4. Following I will explain three transformations that are applicable to the scenarios explained in Section 3.1.1. The scenarios 1, 2 and 3 provide different task trees, that allow different application of transformations.

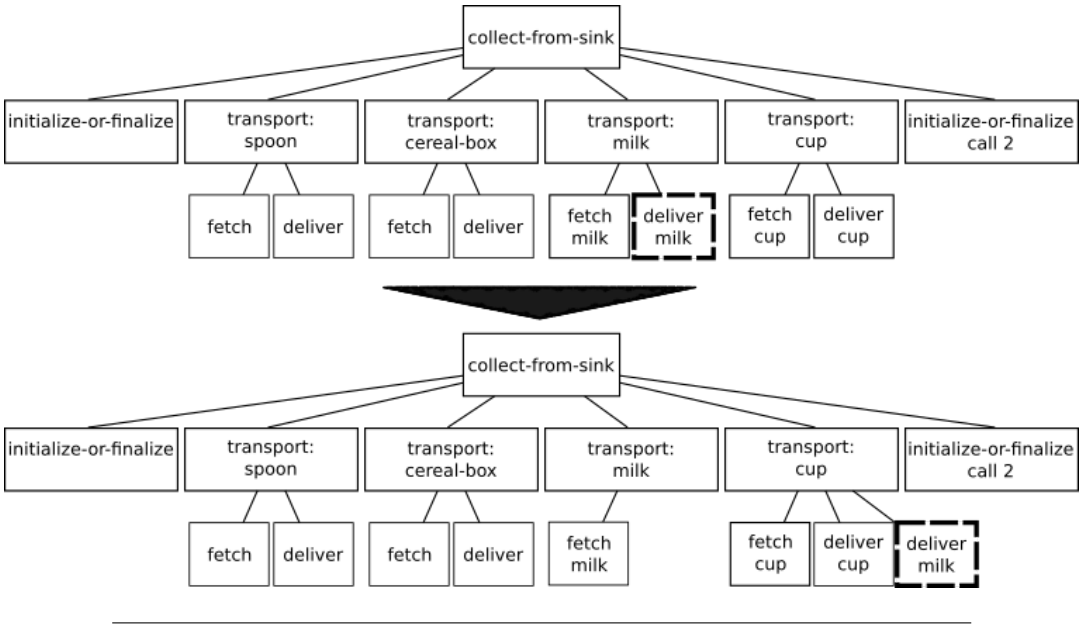

FIGURE 3.7: Scenario 1 task tree transformed with both-hands-rule. It delays one *delivery* action to be executed after the second transport.

both-hands-rule My first transformation is called *both-hands-rule* and enables transporting two objects at the same time. This is done by extracting two actions of type *transporting* from the task tree that start at the same location, then switching the order of their *fetching* and *delivering* actions. Where in the original plan the robot first fetches one object and delivers it immediately, I delay the deliverance and instead append the first *delivering* action after the transport of the second object. After transformation the two *transporting* actions will be resolved to fetching A, fetching B, delivering B and delivering A. By changing part of the scenario's plan the transformation results in a partially changed *output-plan*. The *both-hands-rule* can be applied to scenario 1 and 3, while in scenario 2 this transformation would collide with the opening and closing of the fridge door. How this rule would change the task tree of scenario 1 can be seen in Figure 3.7.

tray-rule For the second transformation I want to make use of extra objects obtainable from the kitchen. A tray fits my purpose, since I want to carry more that one object at a time. With the help of a tray I can put my objects onto this carrier and transport the carrier instead of each object separately. In this demonstration I can put up to three objects on the tray and still carry the tray safely, which means, the objects can stand on the tray without being stacked and there is still enough room to grab the tray with a gripper. In the transformation I need to change the location where the objects are delivered to, from the kitchen island to a pose on the tray. To make this process more flexible I first search for the tray after fetching an object, transform the tray's pose to a suitable location above the tray and swap the original object's delivery location with the one above the tray. After the last object has been put on the carrier object, I fetch the tray and deliver it to the kitchen island. Besides, I must constraint the transformation to only use *transporting* actions that start at the same location (sink area) and end at nearby coordinates (kitchen island), otherwise an object carried by tray might either be placed at the destination falsely, or the *fetching* actions to obtain an object would be undesirably long. What this transformation

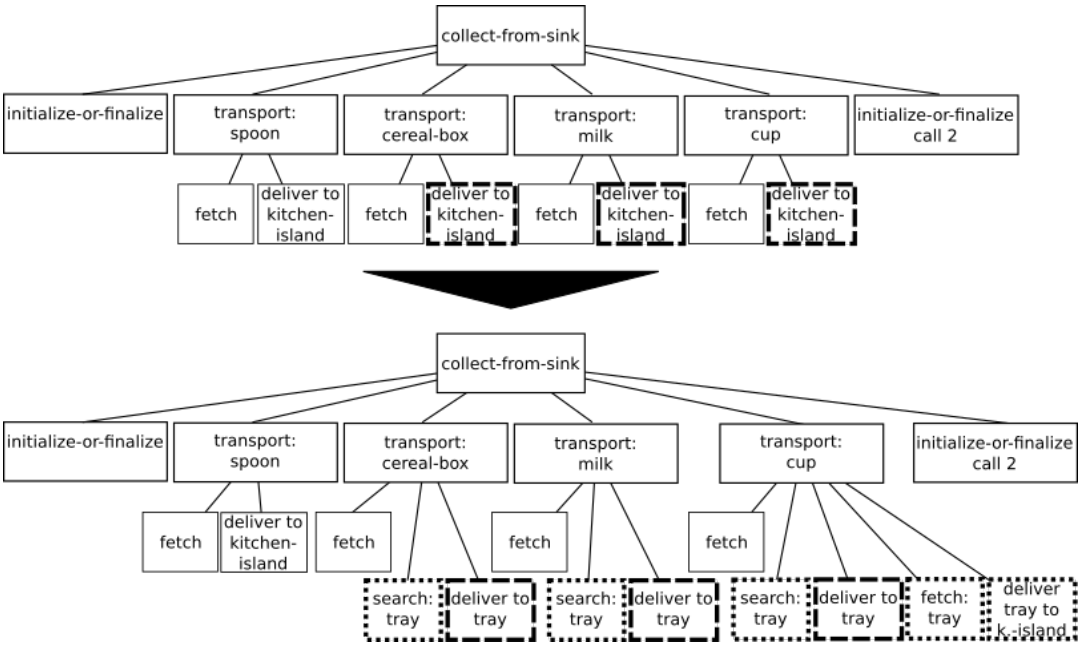

FIGURE 3.8: Scenario 1 task tree transformed with tray-rule. The rule changes the delivery location of up to three objects to a pose on the tray and appends a *fetching* and *delivering* action for the tray.

does to the scenario 1 task tree is illustrated in Figure 3.8. The objects transported by a carrier (the tray) are illustrated in Figure 3.9. Scenario 1 and 3, can be transformed with this rule, but scenario 2 does not contain any *transporting* actions.

By the time when I was developing this transformation there was no mesh for a tray, and also no poses for gripping an object like that. I found a suitable mesh from an other project of the Institute for Artificial Intelligence and adapted its features for the bullet world. The DAE mesh was converted to STL, and its position and scale have been configured. As mentioned before, the bullet world's meshes are surrounded by a bounding box, used to calculate collisions. Simply placing the tray mesh onto the *sink-area* wouldn't let the gripper grasp the tray without colliding with the kitchen below, therefore I needed a base between the kitchen surface and the tray. This base is just a flat block, lifting the tray up a bit. Furthermore, the motion for grasping the tray needs poses, from where the tray can be accessed by the gripper. I provided those gripping poses by copying the poses for gripping a dinner plate and adjusting it for the tray mesh.

The bullet reasoning package did not support transporting objects carried by other objects, like using a tray to transport objects on it. To make this possible I got inspired by how the robot carries objects in his grippers and adapted it to the tray transport. For all objects that shall be moved with the carrier, I attach those to the carrier item in the bullet world. For this to happen I extend the *item* class in the bullet reasoning package by a list of attachments. When the carrier is moved I observe the carrier's pose before and after movement, calculate the transformation and apply this transformation to all attached objects. Additionally I must set the mass of all attached objects to zero to prevent gravity from interfering in the time between movement of the carrier and transition of the attached objects. After the carrier has been moved and placed at its destination, the mass can be reset to the default value and the carrier's attachment list is wiped clean.

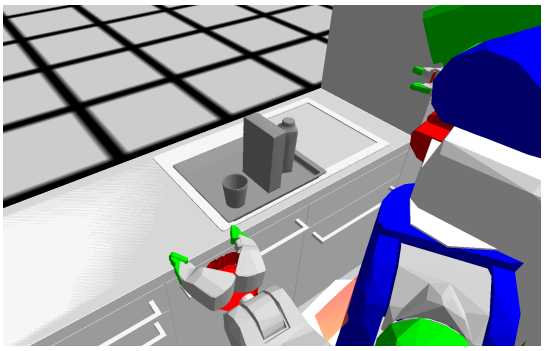

FIGURE 3.9: Scenario 1 after tray-rule transformation. The objects breakfast-cereal, milk and cup are put on the tray.

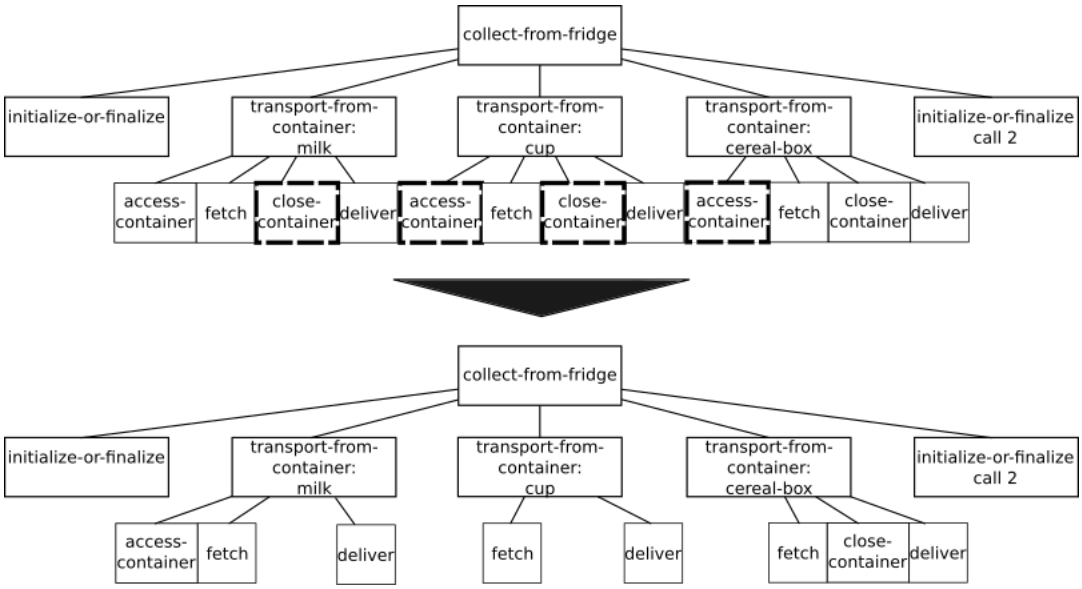

FIGURE 3.10: Scenario 2 task tree transformed with environment-rule. It skips opening and closing containers between the first *access-container* and the last *closing-container* action

environment-rule In the third transformation I want to improve the process of collecting multiple objects from drawers, cupboards and other containers. It is called *environment-rule* and can be applied on plans that include *transporting-from-container* actions like scenario 2 and 3. My main idea is to leave doors and drawers open as long a possible, hence reducing the action of opening and closing containers to a minimum. For this I obtain all *transporting-from-container* actions and their sub-actions *accessing-container* and *closing-container*. The action for accessing the container includes navigating to the container and opening it. The other one, *closing-container* also navigates the robot to the container but closes it. When applying the *environment-rule* some actions will be ignored: from the first transport I only remove the *closing-container* action, from the last I ignore *accessing* the container. For all intermediate actions I can remove both, the *accessing* and *closing* actions. In Figure 3.10 you can see the *environment-rule* being applied on the task tree from scenario 2. Leaving containers open may cause some problems. Doors and cupboards might block the robot from achieving certain actions that would be easier to execute if those

containers would be shut close. Determining when to shut a container is a complex issue that requires reasoning about navigation paths and their dependencies to containers blocking this path. For my scenarios there is no problem in leaving the fridge door open, except for the fact, that nobody should leave a fridge open too long.

The actions *transport-from-container*, *access-container* and *close-container* were not yet developed in the bullet context, also no generic plans exist for manipulating a container. Still I need those plans to be able to argue over them and design transformations like the *environment-rule*. To keep the implementational overhead low I set up those plans and resolved them directly to simulation-specific code.

3.1.4 Applicability and input schema

To check applicability of a transformation I can use Prolog predicates. Those predicates traverse the task tree to find patterns feasible for improvement, done by transformation. In this Section I will also explain what the *input schema* of those transformations are, since checking *applicability* and getting the input schema includes almost the same predicates. I can determine the applicability of a transformation by executing predicates designed to do that. If such a predicate returns a solution, this solution can be used for the transformation. If it resolves to NIL, the rule is not applicable. I have three transformation rules whose applicability predicates vary, since they look for different patterns in the task tree.

both-hands-rule This transformation requires at least two transporting actions to be applicable. Those transports must begin at the same location, otherwise the robot could navigate across the map just to fill his second gripper. This might lead to a further distance navigated than before transformation. To find out if *both-hands-rule* is applicable and get the required input parameters for the transformation, I use the following predicate.

```
1  (<- (task-transporting-siblings (?first-path ?first-fetching-desig)
2                                  (?second-path ?first-delivering-desig))
3      (top-level-name ?top-level-name)
4      (top-level-path ?path)
5      (task-transporting-action ?top-level-name ?path ?second-task ?_)
6      (task-transporting-action ?top-level-name ?path ?first-task ?_)
7      (without-replacement ?first-task)
8      (without-replacement ?second-task)
9      (not (== ?first-task ?second-task))
10     (task-location-description-equal ?first-task ?second-task)
11     (task-full-path ?first-task ?first-path)
12     (task-full-path ?second-task ?second-path)
13     (task-fetching-action ?top-level-name ?first-path ?_ ?first-fetching-desig)
14     (task-delivering-action ?top-level-name ?first-path ?_ ?first-delivering-desig))
```

In lines 3 and 4 I get the task tree's name and the path of the top level plan in the tree. With this I can search for two *transporting* actions in the tree (lines 5-6). The argument *without-replacement* prevents, that I use nodes that already have been transformed. Transforming nodes that already have been used in transformations, could lead to an unstable state of the plan. Why this is problematic is discussed in Section 5.2.

I want to compare the locations of my two *transporting-actions* and for that I must make sure, they are not the same action anyway (lines 9-10). For comparing the locations I can rely on the location's description (see designators in Section 2.2.2). If the predicate finds any solutions so far (lines 5-10) I can start extracting the information necessary for transformation. What I need are the paths of the two nodes that are responsible for the *transporting* actions, those are the tasks to transform. Moreover, I want to have the *fetching* and *delivering* actions from the first *transporting* action, this

will be my *input schema*. The transformation then transforms the first *transporting* action to only contain its *fetching* action. In the second transport I end up with the original *transporting* action with *fetching* and *fetching*, but also the *delivering* action from the first transport.

When executing the predicate I can simply use the first solution. Applying this rule to scenario 1 I first transform the third and fourth transport, and after applying this rule again the first and second transports are transformed to be executed with two grippers (see transformation in Figure 3.7).

tray-rule Carrying objects by using a tray can be useful, when at least two transports are in the plan, that start and end at an approximately near location. In my demo I can place up to three objects on the tray and still carry it securely. For this transformation I do the applicability check and retrieving the input schema separately. This predicate accounts for applicability.

```
1  (<- (task-transporting-with-tray (?delivering-path))
2      (top-level-name ?top-level-name)
3      (top-level-path ?path)
4      (location-distance-threshold ?dist-threshold)
5      (task-transporting-action ?top-level-name ?path ?last-task ?_)
6      (task-transporting-action ?top-level-name ?path ?before-last-task ?_)
7      (not (== ?last-task ?before-last-task))
8      (without-replacement ?last-task)
9      (without-replacement ?before-last-task)
10     (task-location-description-equal ?last-task ?before-last-task)
11     (task-targets-nearby ?last-task ?before-last-task ?dist-threshold)
12     (task-full-path ?last-task ?last-path)
13     (task-delivering-action ?top-level-name ?last-path ?delivering-task ?_)
14     (task-full-path ?delivering-task ?delivering-path))
```

In contrast to the *both-hands-rule* I don't only look at the starting location, but also check the location where the objects are brought to. Comparing the target location can't be done by looking at the descriptions of their location designators, since they only contain discrete poses. Instead I must calculate the distance between those locations to argue if they are delivered at nearby spots. For that I have a constant distance threshold set to 50 cm, that I get from the *location-distance-threshold* predicate (line 4). In lines 5-6 I gather two transports, make sure they are not the same and replaceable (lines 7-9), and check if they start at the same location (line 10) like in *both-hands rule*. Line 11 detects if those transports' target locations are approximately at the same position, using the 50 cm *dist-threshold*. As preparation for getting the input schema, I retrieve the path of just one of the delivering actions, that I will be calling the *anchor*, since this task will be fixed while the amount of other transports, suitable for the *tray-rule*, may vary. I can use this predicate to check applicability because the anchor's path will only be set (not NIL) when all the predicate's arguments resolve positively, and this only happens if there are at least two nearby *transporting* actions.

Now that I got the path of one *delivering* action's task, I want to have all the other delivering actions that can be used for transformation. I use the path of my *anchor* from the *task-transporting-with-action* to get a list of all other *delivering* actions. For getting those deliveries I use the following predicate and feed it with my *anchor*'s path.

```
1  (<- (task-transporting-with-tray-other-deliveries ?first-delivering-path (?other-
       path))
2    (bound ?first-delivering-path)
3    (top-level-name ?top-level-name)
4    (top-level-path ?path)
5    (location-distance-threshold ?dist-threshold)
6    (task-of-path ?first-delivering-path ?del-task)
```

```
7    (task-delivering-action ?top-level-name ?path ?other-del-task ?_)
8    (not (== ?other-del-task ?del-task))
9    (parent+ ?del-task ?task-transport)
10   (task-type ?task-transport :transporting)
11   (parent+ ?other-del-task ?other-task-transport)
12   (task-type ?other-task-transport :transporting)
13   (without-replacement ?task-transport)
14   (without-replacement ?other-task-transport)
15   (task-location-description-equal ?task-transport ?other-task-transport)
16   (task-targets-nearby ?task-transport ?other-task-transport ?dist-threshold)
17   (task-full-path ?other-task-transport ?other-transp-path)
18   (task-delivering-action ?top-level-name ?other-transp-path ?other-delivering-task
         ?_)
19   (task-full-path ?other-delivering-task ?other-path))
```

Although I am only interested in the *delivering* actions, I still need their superior *transporting* actions to check my location constraints (start and end location approx. equal). I get those transports with the *parent+* argument, which recursively receives each parent of the observed deliveries (lines 9-12). By checking the location constraints I reduce my search space to only those transports I need, and from those I derive their *delivery* actions. In the end my *input schema* contains one delivery path (*?first-delivering-path*) and all other delivery paths (*?other-path*). This set of paths is then fed into the *tray-rule* transformation, which alters the deliveries in a way, that the objects are not put directly onto the kitchen island, but instead on the tray (see tray-rule in Figure 3.8).

environment-rule This rule is only applicable to plans that contain transports from within a container. Like for the *both-hands-rule* I determine *applicability* and get the *input schema* by the same predicate, which searches for *transporting-from-container* actions, that share the same container position.

```
1  (<- (task-transporting-from-fridge ?navigate-action ?accessing-path ?closing-path)
2    (top-level-name ?top-level-name)
3    (top-level-path ?path)
4    (top-level-task ?top-level-name ?root)
5    (task-specific-action ?top-level-name ?path :transporting-from-container ?
         transport-action ?_)
6    (task-specific-action ?top-level-name ?path :transporting-from-container ?compare
         ?_)
7    (not (== ?transport-action ?compare))
8    (without-replacement ?transport-action)
9    (without-replacement ?compare)
10   (location-distance-threshold ?threshold)
11   (task-nearby ?compare ?transport-action ?threshold :located-at)
12   (task-full-path ?transport-action ?transport-path)
13   (task-specific-action ?top-level-name ?transport-path :accessing-container ?access
         ?_)
14   (task-full-path ?access ?accessing-path)
15   (task-specific-action ?top-level-name ?accessing-path :navigating ?navigate ?
         navigate-action)
16   (task-specific-action ?top-level-name ?transport-path :closing-container ?closing
         ?_)
17   (task-full-path ?closing ?closing-path))
```

Again I search for pairs of tasks that start at the same location. For my purposes I can say, that similar starting-locations mean, that the transports involve the same container (10+11). This is not a sophisticated, general solution for distinguishing containers but for my case it is accurate enough. In contrast to the *both-hands-rule*, I need all solutions of the *task-transporting-from-fridge* predicate, instead of just the first one. The list of all solutions, which contains all transports involving the container, will be my *input schema* for the *environment-rule*. My parameters *?navigate-action*, *?accessing-path* and *?closing-path* build a 3-tuple, and the solution list is a list of those tuples. The *?navigate-action* moves the robot to the container, *?accessing-path*

points to the task that is used to open the container, and *?closing-path* points to the closing task. In the transformation it is decided, which of the latter tasks should be discarded and which should be kept, while the navigation action must always be executed before fetching an object from the container (see transformation in Figure 3.10).

The most helpful development has been in traversing the task tree in reasonable time. Analyzing huge execution traces like the task tree has already been a problem in (Müller, 2008) and it indeed had negative impact on resolving the predicates shown in Section 3.1.4. Since CRAM Prolog predicates take very long time and massive resources when traversing the tree to find tasks with the specified features, there was a solution in need, and was found. Instead of feeding the whole task tree into a predicate, I can minimize the task tree a prior with the help of Lisp. For example, using Prolog for searching a task that contains an action designator of type *transporting* did return the first task very fast, but the more Prolog needed to search in the tree, the slower it became and eventually flooded the RAM until SBCL (Steel Bank Common Lisp) ran out of heap size. With Lisp, on the other hand, I can take the root node of the tree, receive all children of the root recursively and create a list of lists of all tasks in the whole tree, then I flatten it to a list containing all tasks without any sublists. Now I can filter this list depending on what I search for and return the remaining tasks back to the Prolog predicate. This process sounds costly but is actually done within a few milliseconds depending on the tree size and search complexity.

If filtering the task tree through Lisp is not an option, it may be difficult to design predicates that are fast enough to find the desired solution before you run into memory issues. Traversing the task tree can be a computationally expensive task. If you can't reduce the memory demand of a predicate, you can at least give the SBCL heap more space for calculation. Add following line to the slime config file, to expand memory allocation to 6GB. This should give the predicates enough heap size for small task trees.

```
1 (setq inferior-lisp-program "sbcl --dynamic-space-size 6144")
```

3.2 Generic CRAM plan transformation framework

Previously I talked about three transformations and their applicability rules. The rule's applicability is checked by executing a Prolog predicate designed to search for a pattern used in the rule. Hence a transformation consists of an applicability predicate on the one hand, that can also retrieve the input schema, and the transformation on the other hand, which decides how to change/improve the plan by using the input schema. Executing the transformation eventually results in an output plan. This routine of checking for applicability and applying rules is generalized and automated in the following framework. Every programmer can design rules and predicates suitable for the plan at his or her hands. The generic transformation framework provides registration of transformations and their predicates. Additionally, one can enable or disable certain rules and prioritize them, in case multiple rules may be applicable for the same plan. In Figure 3.11 it is illustrated how the framework works.

To register a transformation you can call *register-transformation-rule* with the name of a transformation function and its applicability predicate. This will add the function and predicate to a hash table in which the transformation name is the key and the predicate the value. You can disable certain rules with *disable-transformation-rule*

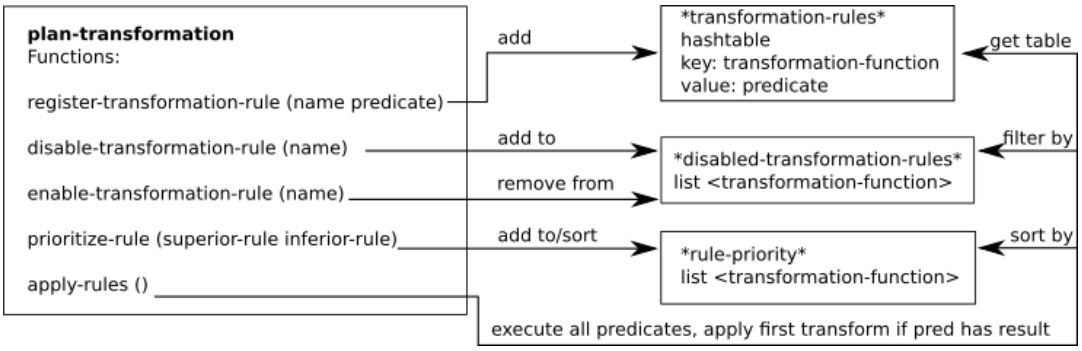

FIGURE 3.11: The generic transformation rule framework allows the registration and prioritization of rules.

and enable them with *enable-transformation-rule*, which basically adds and removes the name from a list of names. All transformation names in this list will be ignored when checking for rule applicability. Since multiple transformations may be applicable at once, the programmer can prioritize the registered rules with *prioritize-rule*. After registering and sorting the transformations you can call *apply-rules*, executing the predicates of **transformation-rules**, except those listed in **disabled-transformation-rules**, in order given by **rule-priority**. From those rules, whose predicates are resolved positively, the first of those transformations is applied on the plan. Applying rules can be done as long as predicates still evaluate positively. If you now launch the top level CRAM again, the transformed *output plan* will be executed.

Chapter 4

Experimental Evaluation

To determine to what extent the transformations affect the scenarios I need to evaluate them. In this chapter I will analyze and compare the distances of the robot moved, how failure prone the plans are and by how much I can reduce the amount of actions performed.

To calculate the performance of a plan I gather all actions of type *navigating*, which contain the robot's navigation poses, and *moving-tcp*, containing poses to where the gripper is moved. After sorting them by their time stamp, to get them in a chronological order, I calculate the euclidean distance between one pose and the next, creating a list of distances of each navigation and gripper movement. Summing them all up I get an estimation of the total distance the robot has moved throughout a scenario.

Inspired by (Müller, 2008), who took between 168 and 336 samples, my evaluation is based on 200 executions per variation of a plan. I will not bother the reader with the whole dataset, but concentrate on mean values of the navigation distance and gripper distance, as well as the total of executed actions, and how prone to failures the plans are. Furthermore, I calculate the variance and significance for a one-tailed t-test. My hypothesis for each evaluated variant is, that transforming the original plan improves the performance of the plan in terms of either the total distance navigated or grippers moved. If the significance is below 0.05, the transformation counts as a significant improvement. Therefore, the hypothesis H_0 to disprove is that there is no significant improvement (significance bigger than 0.05).

The table's columns consist of the means of navigation and gripper distance, which represent the mean total cost of those actions in the plan. In σ I put the variance of the deviations between the parameters from the original and the transformed plan. *Sign.* stands for the significance of the deviations, after calculating the t-value and considering the t-value table. The total distance contains the sum of all costs, #*Actions* are the total number of actions and *Error* has the percentage of the plans over all samples, that contain critical failures in *fetching* or *delivering* actions.

4.1 Evaluation of Transformation Experiments

My samples will cover scenario 1 transformed with the *tray-rule* and *both-hands-rule*, collecting 200 samples each. In the original scenario 1 I use four items to transport. To see the difference in using different amounts of items I will also collect 50 samples of scenario 1 using three and two items, applying the *tray-rule* and *both-hands-rule* as well. Regarding scenario 2, where only the *environment-rule* is interesting to observe, there will be no empirical evaluation. Since there is no kinematic implementation for opening or closing a container I lack of a cost function for those actions. I could estimate the cost of opening the fridge door by taking the approximate distance of an

action type *picking-up* twice, but evaluating data based on such estimations did not seem very purposeful to me. Nevertheless I will take 50 samples to show that evaluations not necessary. Equivalently the mixed scenario 3 has parts from scenario 1 and 2, but since I already evaluate transformations on scenario 1 and container transport is not observed (scenario 2), I rely on the outcome of the scenario 1 transformation samples. When collecting samples I run the whole plans in the bullet simulation, in which I expect some actions to fail. By experience I know, that execution traces with unsuccessful *fetching* or *delivering* actions yield unreasonable data for evaluation with highly fluctuating amounts of counted actions and recorded distances, therefore the data of such traces is discarded.

Scenario 1 with both-hands-rule Focusing the *both-hands-rule* applied twice on the original plan, there is significant improvement in navigation distance and gripper movement (see Table 4.1 and Figure 4.1).

TABLE 4.1: Evaluation of the *both-hands-rule* transformation on scenario 1.

Scenario	Avg. Nav. dist.	σ^2 Nav. dist.	Sign. Nav. dist.	Avg. Grip. dist.	Avg. Total dist.	Avg. #Act.	% Err.
original	37.96			84.66	122.62	191.39	6%
both-hands-rule x2	26.30 -30.72%	5.76	<0.001	80.20 -5.27%	106.50 -13.15%	193.48 +1,09%	10,5% +4,5%

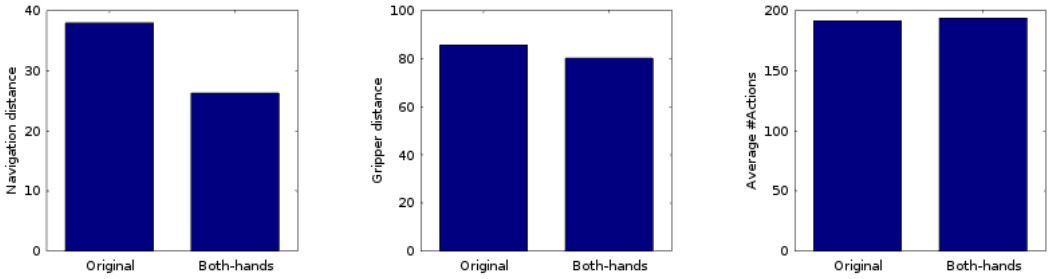

FIGURE 4.1: (Left) Navigation distance, (middle) gripper distance, (right) #Actions. Comparing evaluation data of scenario 1 with both-hands-rule.

From transporting four elements one by one I change the plan to fetching two elements at once, before transporting them, trying to reduce the navigated distance between the starting area and the goal. The navigation distance can be improved by 30.72%, while the amount of actions stay almost the same, since I don't add or remove any in the transformation. Unfortunately the transformed plan shows more failures, from 6% in the original to 10.5% with *both-hands-rule* transformation. Holding two objects, one in each gripper, apparently makes the plan less stable. Significance is calculated to 65.04, which results in a significance below 0.001, which is a significant improvement.

Scenario 1 with tray-rule In table 4.2 and Figure 4.2 you can see the impact of the *tray-rule*.

TABLE 4.2: Evaluation of the *tray-rule* transformation on scenario 1.

Scenario	Avg. Nav. dist.	σ^2 Nav. dist.	Sign. Nav. dist.	Avg. Grip. dist.	Avg. Total dist.	Avg. #Act.	% Err.
original	37.96			84.66	122.62	191.39	6%
tray-rule	36.85	8.39	<0.001	113.79	150.64	248.60	12%
	-2.92%			+34.41%	+22.85%	+29,89%	+6%

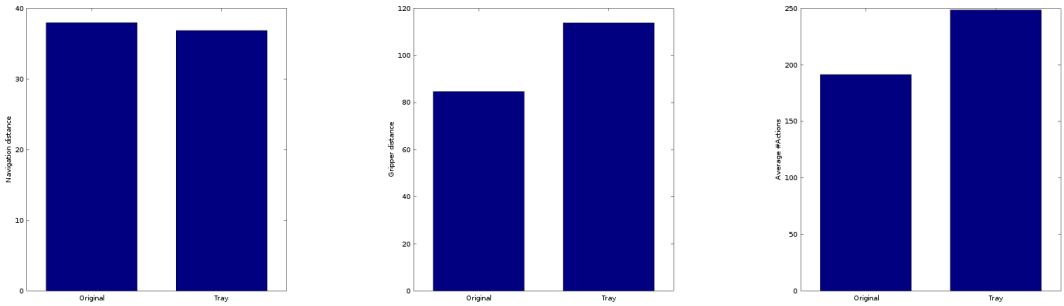

FIGURE 4.2: (Left) Navigation distance, (middle) gripper distance, (right) #Actions. Comparing evaluation data of scenario 1 with tray-rule.

By transforming the original plan the total of navigated distance decreases. For the *tray-rule* it decreases by only 2.92% and after calculation this counts as a significant improvement. Due to to the transformation I now need to transport a total of five items, and the save in distance between starting location and goal is not big enough to compensate for the extra transporting action. If start and goal would me further apart, using a tray would perform better. Since I have additional actions by transporting the tray, also the average distance of grippers moved increases, as well as the total amount of actions. By including the tray the plan gets more complex and more failure prone than the original plan, from 6% to 12%. Calculating the t-value for the navigation distance difference between original and transformed plan samples results in 5.01, which gets me a significance below 0.001, hence it is below 0.05 and counts as significant.

Scenario 1, both-hands-rule with 3 items Taking a look at Table 4.3 and Figure 4.3 you can see the impact of the *both-hands-rule* on the pick and place task including 3 items, first the breakfast-cereal, then milk, and finally the cup.

Using this rule on only three items is not as effective as using it twice on four items, but still improves the original plan. Also notable is the high frequency of errors, which is due to an issue in the bullet simulation with the breakfast-cereal item. The box is difficult to grasp as the first element of three items, constantly choosing inappropriate gripping positions, unable to reach the item. This problem does not occur with other items, but for the sake of consistency I stuck to the original order of items being transported. Since I discard all failed execution and only consider successful executions to maintain consistent evaluation data, the data should still be

TABLE 4.3: Evaluation of the *both-hands-rule* transformation on scenario 1 with 3 items.

Scenario	Avg. Nav. dist.	σ^2 Nav. dist.	Sign. Nav. dist.	Avg. Grip. dist.	Avg. Total dist.	Avg. #Act. dist.	% Err.
original	31.93			45.49	77.42	164.52	60%
both-hands-rule	28.52 -10,68%	19.07	0.01	48.64 +6.92%	66.16 -14.22%	157.43 -4.31%	70% +10%

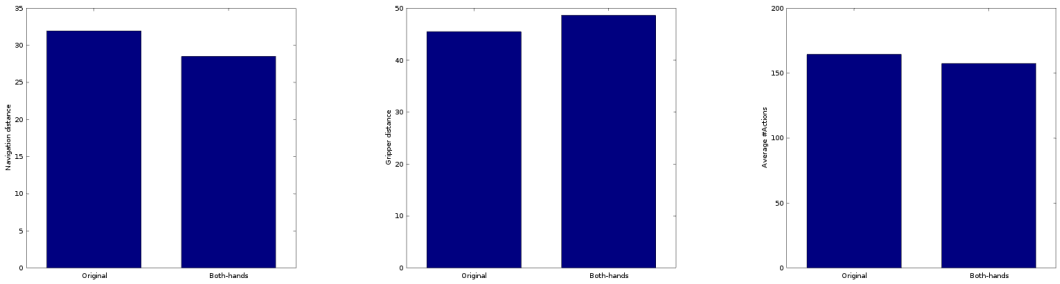

FIGURE 4.3: (Left) Navigation distance, (middle) gripper distance, (right) #Actions. Comparing evaluation data of scenario containing 3 items with both-hands-rule.

reliable enough to spot major differences to the demo with four items. In contrast to scenario 1 with the *both-hands-rule* applied twice, the navigation distance decreased less, but still very well. Comparing the original and the transformed plans' navigation distance I get a t-value of 2.81, which gives us 0.01 significance (see Table 4.3 and Figure 4.4). Like in the demo with four items the gripper distance doesn't really change, but the failure frequency did, probably due to the breakfast-cereal issue.

Scenario 1, tray-rule with 3 items The *tray-rule* is applied on a scenario with three items as well. Its outcome is shown in Table 4.4.

TABLE 4.4: Evaluation of the *tray-rule* transformation on scenario 1 with 3 items.

Scenario	Avg. Nav. dist.	σ^2 Nav. dist.	Sign. Nav. dist.	Avg. Grip. dist.	Avg. Total dist.	Avg. #Act.	% Err.
original	31.93			45.49	77.42	164.52	60%
tray-rule	31.20 -2.29%	5.04	0.4	79.36 +74.46%	110.55 +25.70%	267.46 +62,57%	54% -6%

As in the evaluation of the *tray-rule* transformation on scenario 1 you can see a slight improvement in the navigation distance, but immensely higher gripper distances and amounts of actions. The gripper distance is even bigger than in scenario 1, which may be because we lack the fourth item's distance data, making the impact of the *tray-rule* clearer. The t-value for the navigation distance is 0.92, which

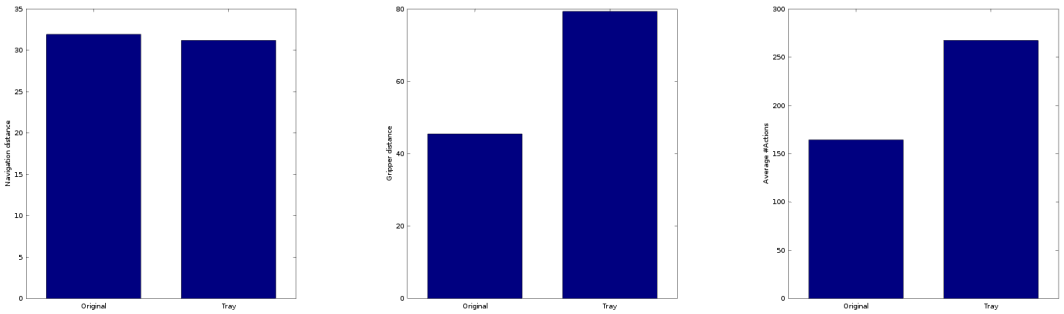

FIGURE 4.4: (Left) Navigation distance, (middle) gripper distance, (right) #Actions. Comparing evaluation data of scenario containing 3 items with tray-rule.

corresponds to a significance of 0.4, hence an improvement is not at hand. Again, increasing the distance between start and goal may render this rule more profitable.

Scenario 1, both-hands-rule with 2 items A scenario with only two items should be optimal to the how the *both-hands-rule* performs. In Table 4.5 and Figure 4.5 it is shown how well it improves the plan.

TABLE 4.5: Evaluation of the *both-hands-rule* transformation on scenario 1 with 2 items.

Scenario	Avg. Nav. dist.	σ^2 Nav. dist.	Sign. Nav. dist.	Avg. Grip. dist.	Avg. Total dist.	Avg. #Act.	% Err.
original	19.88			35.19	55.07	103.94	2%
both-hands-rule	13.04 -34.41%	2.27	<0.001	35.15 -0.11%	48.19 -12.49%	101.61 -2.24%	12% +10%

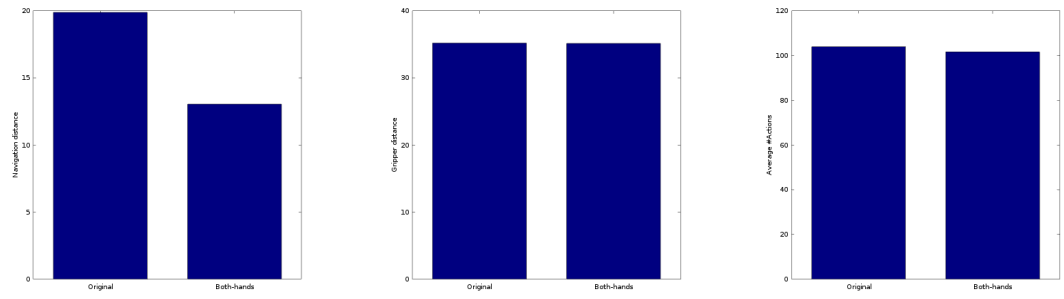

FIGURE 4.5: (Left) Navigation distance, (middle) gripper distance, (right) #Actions. Comparing evaluation data of scenario containing 2 items with both-hands-rule.

Like with four items, a scenario with two items can be greatly improved with the *both-hand-rule*. The robot navigates shorter distances, by 34.41% compared to transporting the two items separately. Gripper distance doesn't change much, as

expected and already observed in the previous experiments. With a t-value of 30.26 this transformation changes the behavior significantly.

Scenario 1, tray-rule with 2 items Now I want to now if I can improve an environment with two items by transporting them with the help of a tray. The corresponding data can be seen in Table 4.6.

TABLE 4.6: Evaluation of the *tray-rule* transformation on scenario 1 with 2 items.

Scenario	Avg. Nav. dist.	σ^2 Nav. dist.	Sign. Nav. dist.	Avg. Grip. dist.	Avg. Total dist.	Avg. #Act.	% Err.
original	19.88			35.19	55.07	103.94	2%
tray-rule	27.28 +37.22%	4.00	<0.001	68.38 +94.32%	95.66 +80.97%	173.15 +66,59%	8% +6%

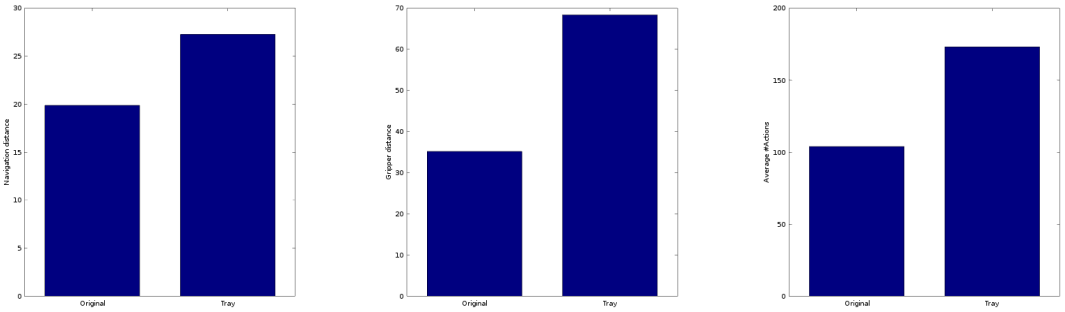

FIGURE 4.6: (Left) Navigation distance, (middle) gripper distance, (right) #Actions. Comparing evaluation data of scenario containing 2 items with tray-rule.

For only two elements, the *tray-rule* produces no improvement. From two transporting actions I make three, increasing the navigated distance even higher than in the untransformed plan. As expected, the gripper distance increases drastically. In conclusion I can say, that the *tray-rule* is not made for scenarios with such few items to carry.

Scenarios 2 and 3 with environment-rule After taking 50 samples for scenario 2 before and after applying the *environment-rule* I compared their results. With the navigated distance of 36.34 (before) to 37.67 (after) I consider this difference unimportant for evaluation, as well as the gripper movement distance of 42.68 (before) to 41.68 (after). Only the difference in the average amount of actions form 125.47 (before) to 122.44 (after) gives a hint about the actions, responsible for opening and closing the container, removed by the transformation. Since I have no means to calculate the cost of those actions, apart from acknowledging their absence, I must forfeit evaluating these scenarios.

4.2 Evaluation Summary

As you can see, there are some improvements by transforming the plans. It depends on the scenario's setup which transformations are suitable and result in the best improvement. The *tray-rule* for example does not perform good in scenarios with short transports. If there would be a scenario, where start and goal were further apart, I consider the *tray-rule* a good choice. As long as the improvement in navigation is higher than the cost for extra transitions of items, the *tray-rule* may perform better than the original plan, but it would take wider environments than a small kitchen to discover this threshold. Considering transportation supported by carriers is most recommended for pick and place scenarios with many items, while carrying just a few items is not worth the extra actions.

For the *both-hands-rule* I find it a serious improvement throughout all examples. The gain of saved navigation is higher in scenarios with an even amount of items, while the cost of gripper movement is almost unchanged. If possible, exploiting a robot's resources pays off, at least regarding the use of all grippers available.

Chapter 5

Conclusion

Here I conclude what information has been gained throughout this thesis, giving a summary of my approach, having a discussion about my concepts and implementation, and eventually I give some recommendations on future work regarding this topic.

5.1 Summary

Plan transformation has been pursued for over three decades, developed and implemented in various ways and today still an interesting field of research. Automatically letting a robot improve its behavior is a goal that many scientists try to achieve. Plan-based robotics is one of those fields of research, building reasonable, sequential or hierarchical, comprehensive manuals for robotic actions. Plan transformation, an application of plan-based control, concentrates on finding and changing patterns in a robot's activities.

In this thesis I elaborated the use of transformation on plans, written in the CRAM architecture. As can be seen, plan transformation techniques can be realized on CRAM plans by using code replacements. All required data is fetched from the execution trace's task tree, containing the hierarchical plan structure, as well as broad information to reason upon. Throughout the combination of CRAM Prolog and Lisp, extracting valuable data is even possible in larger scale logs, as I found out when using predicates to traverse the task tree. My analysis of large scale logs has not been aimed at the tree's depth, but more at the amount of information each node contains. Furthermore, those transformations have been tested in a bullet simulation, providing a realistic environment for simulating the actions of robots. For pick and place plans there are a lot of possible enhancements, as I have shown just a few, but more did I contribute basic concepts about the approach of developing such transformation rules with the tools at hand for analysis and application of changes.

5.2 Discussion

One limitation of CRAM code replacements is that it does not change the task-tree's original structure. I can only replace the tasks code by other code, but am not able to add, remove or merge multiple task nodes. It is due to the flexibility of top-level execution to use a plan's path as identifier of a node that the original plan hierarchy cannot be altered since it would not fit the original program structure anymore. In other words, when a top level plan is executed, the executed plan's hierarchy always stay the same, creating the same task tree again. If there would be an extra node in the task tree, that node would never be reached, since there is no corresponding plan in the hierarchy of the top level plan, that would create it, hence the new node

is never checked on code replacements, neither does it occur in any way during execution. I am bound to the task tree's structure, given by the top level plans hierarchy of underlying plans. Using code replacements on the task tree's nodes is one way of changing a plans behavior, but larger scale transformations, that require additional actions (like the *tray-rule*) are hard to realize with the restriction of a fixed plan hierarchy.

When a plan is transformed in a way like the *tray-rule*, where an extra action is added in the code replacement, this extra action is added to the task tree as soon as the transformed plan is executed. This leaves the task tree in a weird structure, where, on one hand, I have the extra action inside the replacement of the last *delivering* action, on the other hand, a newly created task node containing the tray's *transporting* action, which would never be executed by the top level plan in its original form. After all this additional task node does not cause malfunction of the transformed plan, but gives insight on how the execution trace is built. Every executed plan must be represented in the task tree, hence the tray transport action, contained in the code replacement, is added as a node to the task tree. This behavior yields positive and negative aspects.

Having additional, unattended, loose nodes in the task tree makes reasoning more difficult. My Prolog predicates analyze the tree on patterns, in which such nodes would appear as well, because they are not that easily distinguishable from the usual nodes. On the contrary, such nodes, created after execution of a freshly transformed plan, might have the potential to overcome the problem of a static, immutable task tree, elaborated above, overcoming the restriction of replacing a node's code instead of actually adding and removing a node. In Section 5.3 I will propose my ideas on how to use this behavior to our advantage.

5.3 Recommendations on Future Work

Transformation of CRAM plans is far from easily applicable, and has much potential for improvement. The task tree, for instance, must be more flexible, but also consistent, to serve as a reliable reasoning source. Alterations can only be applied by attaching a code replacement on plans' nodes, while the original code is still present. During execution the original code of a node is bypassed and the replacement will be executed, if a code replacement exists. In the current state of development, the task tree is not suitable for extensive acts of transformation, since transformation of nodes, running with code replacements, can lead to inconsistent misbehavior.

An idea for an alternative concept contains two aspects: (1) it must be possible to completely replace a task tree's node, not just inject a code replacement, and there must be a way to add and remove whole subtrees; (2) in order to execute such alterations, the task tree must be executable independently from the top level plan. With this, the top level plan is only used once to create the task tree, and transformations can then be applied only on the task tree, regardless of the top level plan's structure. Furthermore, such a task tree can be granulated into subtrees, equivalent to the hierarchy of CRAM plans, which can be collected into a plan library like in (Müller, 2008), making their transformation much easier. It is already possible to execute subtrees separately but using this on a transformed task tree is yet to be developed.

When it comes to analyzing large datasets, it is imperative to use tools that work fast enough to traverse the data in reasonable time and resource dimensions. SWI Prolog may conquer this issue but CRAM Prolog does not by itself. Using Lisp

functions to compress the knowledge base will help immensely in this process. On the other hand, it may be more appropriate to enhance the CRAM Prolog algorithms, to make them powerful enough for solving this task. For now, evaluation of the task tree is difficult to do by CRAM Prolog predicates alone.

During evaluation of the scenarios I constantly encountered issues the longer my samples were running, therefore I recommend splitting the evaluation in smaller chunks to prevent data loss. I implemented function to export data from the REPL to files on my hard drive, since it occasionally stopped working after several runs. I reckon that the issue lies within recording the execution trace. It could be helpful to attach custom cleanup mechanisms to the *on-top-level-cleanup-hook*, which is implemented by the execution trace, or improve the recording and garbage collection of the execution trace itself.

In the field of plan-based control of robotic systems, plan transformation is among the more promising approaches for improving a robot's behavior. Transformational planning is applicable on CRAM plans in various ways already. Regarding the usage of the task tree to manipulate plans after execution, it still takes a bit of development and refactoring to accomplish the many advantages of transforming plans, but my approach takes some steps into the right direction.

List of Figures

2.1 Kitchen Environment in the Bullet Simulation 8
2.2 IAI Kitchen environment 8
2.3 Searching action result 11
2.4 Fetching action result 12
2.5 Task tree example 17
2.6 TRANER Structure 19
2.7 Example Transformation changing execution order 19
2.8 example transformed task tree 20

3.1 Scenario 1 22
3.2 Scenario 3 22
3.3 Scenario 2 23
3.4 Scenario 1 task tree 24
3.5 Scenario 2 task tree 24
3.6 Scenario 3 task tree 24
3.7 Scenario 1 transformed by both-hands-rule 25
3.8 Scenario 1 transformed by tray-rule 26
3.9 Scenario 1 objects collected on tray 27
3.10 Scenario 2 transformed by environment-rule 27
3.11 Generic transformation framework 32

4.1 Histogram scenario 1 both-hands-rule 34
4.2 Histogram scenario 1 tray-rule 35
4.3 Histogram scenario with 3 items and booth-hands-rule 36
4.4 Histogram scenario with 3 items tray-rule 37
4.5 Histogram scenario with 2 items and booth-hands-rule 37
4.6 Histogram scenario with 2 items tray-rule 38

List of Abbreviations

CRAM	**C**ognitive **R**obot **A**bstract **M**achine
IAI	**I**nstitut of **A**rtificial **I**ntelligence
IK	**I**nverse **K**inematic
Lisp	**Lis**t **P**rocessing Language
ROS	**R**obot **O**perating **S**ystem
RPL	**R**eactive **P**rogramming **L**anguage
TRANER	**TRA**nsformational Plan**NER** for Everyday Activity

Bibliography

Beetz, Michael (1992). *Decision-theoretic Transformational Planning*. Tech. rep. DFKI, p. 22. URL: https://www.dfki.de/web/forschung/publikationen/renameFileForDownload?filename=RR-92-07.pdf&file_id=uploads_1733.

— (2000). *Concurrent Reactive Plans, Anticipation and Forestalling Execution Failures*. Vol. 1772. Lecture Notes in Computer Science. Springer. ISBN: 3-540-67241-9. DOI: 10.1007/3-540-46436-0. URL: https://doi.org/10.1007/3-540-46436-0.

— (2001). "Structured Reactive Controllers". In: *Autonomous Agents and Multi-Agent Systems* 4.1/2, pp. 25–55. DOI: 10.1023/A:1010014712513. URL: https://doi.org/10.1023/A:1010014712513.

— (2002). "Plan Representation for Robotic Agents". In: *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems, April 23-27, 2002, Toulouse, France*. Ed. by Malik Ghallab, Joachim Hertzberg, and Paolo Traverso. AAAI, pp. 223–232. ISBN: 1-57735-142-8. URL: http://www.aaai.org/Library/AIPS/2002/aips02-023.php.

— (2013). "Cognition-Enabled Autonomous Robot Control for the Realization of Home Chore Task Intelligence". In: *SOFSEM 2013: Theory and Practice of Computer Science, 39th International Conference on Current Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic, January 26-31, 2013. Proceedings*, p. 106. DOI: 10.1007/978-3-642-35843-2_9. URL: https://doi.org/10.1007/978-3-642-35843-2_9.

Beetz, Michael and Drew V. McDermott (1997). "Expressing Transformations of Structured Reactive Plans". In: *Recent Advances in AI Planning, 4th European Conference on Planning, ECP'97, Toulouse, France, September 24-26, 1997, Proceedings*. Ed. by Sam Steel and Rachid Alami. Vol. 1348. Lecture Notes in Computer Science. Springer, pp. 64–76. ISBN: 3-540-63912-8. DOI: 10.1007/3-540-63912-8_76. URL: https://doi.org/10.1007/3-540-63912-8_76.

Beetz, Michael et al. (2010). "Generality and legibility in mobile manipulation". In: *Auton. Robots* 28.1, pp. 21–44. DOI: 10.1007/s10514-009-9152-9. URL: https://doi.org/10.1007/s10514-009-9152-9.

Beetz, Michael et al. (2011). "Robotic roommates making pancakes". In: *11th IEEE-RAS International Conference on Humanoid Robots (Humanoids 2011), Bled, Slovenia, October 26-28, 2011*. IEEE, pp. 529–536. ISBN: 978-1-61284-866-2. DOI: 10.1109/Humanoids.2011.6100855. URL: https://doi.org/10.1109/Humanoids.2011.6100855.

Beetz, Michael et al. (2012). "Cognition-Enabled Autonomous Robot Control for the Realization of Home Chore Task Intelligence". In: *Proceedings of the IEEE* 100.8, pp. 2454–2471. DOI: 10.1109/JPROC.2012.2200552. URL: https://doi.org/10.1109/JPROC.2012.2200552.

Bothelho, S. and Rachid Alami (2000). "Robots that Cooperatively Enhance Their Plans". In: *Distributed Autonomous Robotic Systems 4, Proceedings of the 5th International Symposium on Distributed Autonomous Robotic Systems, DARS 2000, October 4-6, 2000, Knoxville, Tennessee, USA*. Ed. by Lynne E. Parker, George A. Bekey, and Jacob Barhen. Springer, pp. 55–68. ISBN: 4-431-70295-4.

Fedrizzi, Andreas et al. (2009). "Transformational planning for mobile manipulation based on action-related places". In: *14th International Conference on Advanced Robotics, ICAR 2009, 22-26 June 2009, Munich, Germany*. IEEE, pp. 1–8. URL: http://ieeexplore.ieee.org/document/5174701/.

Gateau, Thibault, Charles Lesire, and Magali Barbier (2013). "HiDDeN: Cooperative plan execution and repair for heterogeneous robots in dynamic environments". In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, Tokyo, Japan, November 3-7, 2013*. IEEE, pp. 4790–4795. DOI: 10.1109/IROS.2013.6697047. URL: https://doi.org/10.1109/IROS.2013.6697047.

Hammond, Kristian J. (1990). "Explaining and Repairing Plans That Fail". In: *Artif. Intell.* 45.1-2, pp. 173–228. DOI: 10.1016/0004-3702(90)90040-7. URL: https://doi.org/10.1016/0004-3702(90)90040-7.

Kazhoyan, Gayane and Michael Beetz (2017). "Programming robotic agents with action descriptions". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2017, Vancouver, BC, Canada, September 24-28, 2017*. IEEE, pp. 103–108. ISBN: 978-1-5386-2682-5. DOI: 10.1109/IROS.2017.8202144. URL: https://doi.org/10.1109/IROS.2017.8202144.

Kruse, Thibault and Alexandra Kirsch (2010). "Towards Opportunistic Action Selection in Human-Robot Cooperation". In: *KI 2010: Advances in Artificial Intelligence*. Ed. by Rüdiger Dillmann et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 374–381. ISBN: 978-3-642-16111-7.

Liberatore, Paolo (1998). "On the Compilability of Diagnosis, Planning, Reasoning about Actions, Belief Revision, etc". In: *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98), Trento, Italy, June 2-5, 1998*. Ed. by Anthony G. Cohn, Lenhart K. Schubert, and Stuart C. Shapiro. Morgan Kaufmann, pp. 144–155.

Maitin-Shepard, Jeremy et al. (2010). "Cloth grasp point detection based on multiple-view geometric cues with application to robotic towel folding". In: *IEEE International Conference on Robotics and Automation, ICRA 2010, Anchorage, Alaska, USA, 3-7 May 2010*. IEEE, pp. 2308–2315. DOI: 10.1109/ROBOT.2010.5509439. URL: https://doi.org/10.1109/ROBOT.2010.5509439.

McDermott, Drew (1992). *Transformational Planning Of Reactive Behavior*. Tech. rep.

Mcdermott, Drew (1993). *A Reactive Plan Language*. Tech. rep.

Mösenlechner, Lorenz (2016). "The Cognitive Robot Abstract Machine: A Framework for Cognitive Robotics". PhD thesis. Technical University Munich, Germany. URL: http://nbn-resolving.de/urn:nbn:de:bvb:91-diss-20160520-1239461-1-3.

Mösenlechner, Lorenz and Michael Beetz (2011). "Parameterizing actions to have the appropriate effects". In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2011, San Francisco, CA, USA, September 25-30, 2011*. IEEE, pp. 4141–4147. ISBN: 978-1-61284-454-1. DOI: 10.1109/IROS.2011.6094883. URL: https://doi.org/10.1109/IROS.2011.6094883.

— (2013). "Fast temporal projection using accurate physics-based geometric reasoning". In: *2013 IEEE International Conference on Robotics and Automation, Karlsruhe, Germany, May 6-10, 2013*. IEEE, pp. 1821–1827. ISBN: 978-1-4673-5641-1. DOI: 10.1109/ICRA.2013.6630817. URL: https://doi.org/10.1109/ICRA.2013.6630817.

Mösenlechner, Lorenz, Nikolaus Demmel, and Michael Beetz (2010). "Becoming action-aware through reasoning about logged plan execution traces". In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems, October 18-22, 2010, Taipei,*

Taiwan. IEEE, pp. 2231–2236. ISBN: 978-1-4244-6674-0. DOI: 10.1109/IROS.2010.5650989. URL: https://doi.org/10.1109/IROS.2010.5650989.

Müller, Armin (2008). "Transformational Planning for Autonomous Household Robots using Libraries of Robust and Flexible Plans". PhD thesis. Technische Universität München. URL: http://mediatum2.ub.tum.de/node?id=645588.

Quigley, Morgan et al. (2009). "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software*.

Simmons, Reid G. (1988). "A Theory of Debugging Plans and Interpretations". In: *Proceedings of the 7th National Conference on Artificial Intelligence. St. Paul, MN, August 21-26, 1988*. Ed. by Howard E. Shrobe, Tom M. Mitchell, and Reid G. Smith. AAAI Press / The MIT Press, pp. 94–99. ISBN: 0-262-51055-3. URL: http://www.aaai.org/Library/AAAI/1988/aaai88-017.php.

Sussman, Gerald J. (1973). *A Computational Model of Skill Acquisition*. Tech. rep. Cambridge, MA, USA: Massachusetts Institute of Technology. Dept. of Mathematics. URL: http://hdl.handle.net/1721.1/12183.

Sussman, Gerald Jay (1975). *A Computer Model of Skill Acquisition*. New York, NY, USA: Elsevier Science Inc. ISBN: 044400159X.

Tenorth, Moritz and Michael Beetz (2009). "KNOWROB - knowledge processing for autonomous personal robots". In: *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems, October 11-15, 2009, St. Louis, MO, USA*. IEEE, pp. 4261–4266. ISBN: 978-1-4244-3803-7. DOI: 10.1109/IROS.2009.5354602. URL: https://doi.org/10.1109/IROS.2009.5354602.

— (2010). "Priming transformational planning with observations of human activities". In: *IEEE International Conference on Robotics and Automation, ICRA 2010, Anchorage, Alaska, USA, 3-7 May 2010*, pp. 1499–1504. DOI: 10.1109/ROBOT.2010.5509161. URL: https://doi.org/10.1109/ROBOT.2010.5509161.

Wyrobek, Keenan A. et al. (2008). "Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot". In: *2008 IEEE International Conference on Robotics and Automation, ICRA 2008, May 19-23, 2008, Pasadena, California, USA*. IEEE, pp. 2165–2170. DOI: 10.1109/ROBOT.2008.4543527. URL: https://doi.org/10.1109/ROBOT.2008.4543527.