Universität Bremen

# Situations- und kontextabhängige Manipulationsaktivitäten autonomer Agenten in alltäglichen Tischdeck-Szenarien

Bachelorarbeit

*Thomas Carstens*

| | |
|---|---|
| Prüfer der Bachelorarbeit: | 1. Prof. Michael Beetz |
| | 2. Dr. Karsten Sohr |
| Betreuer | Jan Winkler |

# Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Bremen, den 18. Mai 2015

---

Thomas Carstens

# Kurzfassung

Roboter werden heutzutage nicht mehr nur für statische Aufgaben in der Industrie genutzt, sondern können mittlerweile auch in Haushalten als Hilfe eingesetzt werden. Hierfür ist es allerdings notwendig, die Pläne der Roboter deutlich dynamischer, im Gegensatz zu den Plänen für Industrieroboter, zu gestalten, da die statischen Pläne für Industrieroboter in einer nicht vordefinierten Umgebung an ihre Grenzen stoßen.
In dieser Bachelorarbeit wird der Ansatz von dynamischen Plänen aufgegriffen und genutzt, um einen solchen Plan zu entwickeln. Das Ziel dieses Planes ist es, dass ein Roboter autonom einen Tisch decken kann. Dabei ist es gelungen, dass der Roboter bei der Suche eines Objektes auf Bedingungen in der Umwelt reagieren kann, um den maximalen Erfolg zu erreichen.
Um das Ziel des Planes zu erreichen, wurde in dieser Bachelorarbeit auf bereits bestehenden Bibliotheken aufgebaut. Diese wurden im Verlauf der Implementierung modifiziert und erweitert, um die gewünschte Funktionsweise sicherzustellen. Außerdem wird eine Methodik vorgestellt, mit welcher sich die Implementierung dieser Bachelorarbeit darstellen lässt.
Abschließend wurden zwei Experimente durchgeführt, welche die Funktionalität des implementierten Planes und der Funktionen zeigt. Dabei wurde darauf geachtet, dass der Roboter das benötigte Objekt nicht direkt im ersten Versuch findet, um darzulegen, dass er autonom ein Objekt suchen und finden kann. Außerdem wurde in den Experimenten gezeigt, dass der Roboter ein Objekt korrekt auf einem Tisch platzieren kann.

Frage an Jan: Sätze einfach switchen? Erwähne ja kurz, dass ich auf Bibs aufsetze und das die modifiziert wurden

# Inhaltsverzeichnis

**Eidesstattliche Erklärung** I

**Abstract** III

**Inhaltsverzeichnis** V

**1 Einführung** 3

**2 Verwendete Software** 5
- 2.1 CRAM . . . . . 5
- 2.2 KnowRob . . . . . 6

**3 Grundlagen** 9
- 3.1 CRAM und LISP Grundlagen . . . . . 9
- 3.2 Darstellungsmethode Behaviour Trees . . . . . 12

**4 Implementierung** 19
- 4.1 Übersicht der Szenen . . . . . 19
- 4.2 Übersicht der Funktionen . . . . . 21
- 4.3 Beschreibung des Plans . . . . . 26
- 4.4 Robot Operating System (ROS) . . . . . 31

**5 Experimente** 33
- 5.1 Tischdeckszenario mit leerem Tisch . . . . . 33
- 5.2 Tischdeckszenario mit dem gesuchten Objekt auf dem Tisch . . . . . 35

**6 Fazit** 37

**7 Ausblick** 39

**Literaturverzeichnis** a

# Todo list

Frage an Jan: Sätze einfach switchen? Erwähne ja kurz, dass ich auf Bibs aufsetze und das die modifiziert wurden . . . . . . . . . . III

Am Ende prüfen, ob alle Verweise etc auf den richtigen Seiten sind! . . . . . . . . . . 1

Am Ende einmal drüber schauen, ob die Zeilen überall passen, ansonsten per Hand eine newline einfügen
Gleiches gilt für Tabellen etc . . . . . . . . . . 3

Frage an Jan: Kapitel in Implementierung mit Funktion ... oder Szene ... beginnen? Damit diese nicht mit kleingeschriebenen Buchstaben anfangen? . . . . . . . . . . 3

Frage an Jan: Wie ist das mit Quellenverweisen? Ich hab teilweise sowas geschrieben wie: In Quelle XY und er macht dann derzeit daraus bspw: In [2]... ist das ok oder lieber anders? . . . . . . . . . . 3

Frage an Jan: Kann ich Knickarme schreiben? . . . . . . . . . . 3

Frage an Jan: kann ich das so schreiben? . . . . . . . . . . 15

Frage an Jan: progn führt ja place theoretisch auch aus, wenn pick nicht klappt. Trotzdem ok? . . . . . . . . . . 16

TODO: Jan nach dem Beispiel fragen und dann anpassen . . . . . . . . . . 33

TODO: Jan nach dem Beispiel fragen und dann anpassen . . . . . . . . . . 35

Am Ende prüfen, ob alle Verweise etc auf den richtigen Seiten sind!

*Kapitel 1*

# Einführung

Am Ende einmal drüber schauen, ob die Zeilen überall passen, ansonsten per Hand eine newline einfügen
Gleiches gilt für Tabellen etc

Frage an Jan: Kapitel in Implementierung mit Funktion ... oder Szene ... beginnen? Damit diese nicht mit kleingeschriebenen Buchstaben anfangen?

Frage an Jan: Wie ist das mit Quellenverweisen? Ich hab teilweise sowas geschrieben wie: In Quelle XY und er macht dann derzeit daraus bspw: In [2]... ist das ok oder lieber anders?

Roboter werden in der heutigen Zeit bereits in vielen Unternehmen beispielsweise als Hilfskraft für Abläufe genutzt. So wurden die ersten Roboter bereits in den 60er Jahren in der Automobilindustrie genutzt, um das Entnehmen und Einsetzen von heißen Spritzgussteilen zu übernehmen [1]. In den weiteren Jahren wurde unter anderem der sogenannte „Stanford-Arm" entwickelt, welcher Manipulationsaufgaben mit sechs Freiheitsgraden übernehmen konnte. Der „Stanford-Arm" wird dabei als Prototyp für die heute bekannten Knickarme gesehen [1].

Frage an Jan: Kann ich Knickarme schreiben?

Mittlerweile werden Roboter nicht mehr ausschließlich in der Automobilindustrie genutzt, sondern finden in vielen weiteren Firmen Anwendung, da es möglich ist, viele Aufgaben automatisiert und parallel ablaufen zu lassen.
Roboter können jedoch auch für weit mehr als für statische und wiederkehrende Industrieaufgaben genutzt werden, beispielsweise wurde von der `Transitions Research Corporation` ein Roboter Namens `Helpmate` entwickelt. `Helpmate` hat dabei die Aufgabe, die Krankenpfleger zu unterstützen und ihnen Gegenstände oder andere benötigte Objekte zu bringen [2]. Doch auch für Privatpersonen gibt es heutzutage interessante Roboter, welche im Alltag helfen können. Ein Beispiel hierfür sind „Cleaning-Robots", welche einer Person beim Putzen eines Raumes helfen sollen [3].
Roboter sind also zu der heutigen Zeit in vielen verschiedenen Aufgabenfeldern tätig, weshalb es auch verschiedene Anforderungen an die Roboter gibt. Ein Industrieroboter befindet sich meist in einer statischen Umgebung und vollzieht somit immer wiederkehrende Aufgaben.

Aus diesem Grund kann er exakt für die benötigte Aufgabe programmiert werden, um diese im Anschluss korrekt lösen zu können.
Ein Haushaltsroboter soll im Gegensatz dazu kein statisches Verhalten widerspiegeln, sondern auf Veränderungen in seiner Umgebung reagieren. Um dieses Verhalten darstellen zu können, werden `Kontrollprogramme` genutzt, welche anhand von Bedingungen Entscheidungen für Roboter treffen. Eine solche Möglichkeit bietet die Plansprache `CRAM` (siehe Kapitel 2.1 auf der nächsten Seite).
Eine solche Plansprache bietet nun die Möglichkeit, dass der Roboter anhand von verschiedenen Bedingungen seinen eigentlichen Ablauf überdenkt. Wenn ein Roboter nun beispielsweise ein Objekt greifen soll und dieser Versuch fehlschlägt, kann der Roboter sein Vorgehen nun selbst überdenken. Eine Bedingung könnte zum Beispiel die Anzahl von Griffversuchen sein, also dass der Roboter nach dem ersten Versuch nicht direkt aufgibt, sondern erneut versucht das Objekt zu greifen, da Objekte in einer dynamischen Umgebung nicht zwingend an einer genau definierten Stelle stehen müssen.
Dies zeigt auch, aus welchem Grund innerhalb der Industrie viele Roboter keine Plansprache benötigen, da dort die Position der Objekte meist genau definiert ist und der Roboter somit genau weiß, wo sich das Objekt befinden muss, damit er es greifen kann.
Dieser Ansatz der Plansprachen wurde nun in diese Bachelorarbeit getragen, um das Szenario eines Roboters, welcher autonom einen Tisch deckt, anzugehen. Um einen Tisch zu decken, müssen verschiedenste Informationen verfügbar sein. Für wen muss gedeckt werden? Was möchten diese Personen essen? Welche Objekte brauche ich für diese Mahlzeiten? Und muss für Frühstück, Mittag oder Abend gedeckt werden? Außerdem muss es Informationen über die Positionen der Objekte auf dem zu deckenden Tisch geben, denn ein Teller soll immer in der Mitte eines Platzes stehen, während ein Glas hinter dem Teller stehen soll.
Auch Informationen über das Objekt an sich werden benötigt. Wie wird das Objekt erkannt? Wie kann das Objekt gegriffen werden? Mit letzteren beiden Fragen wird sich innerhalb dieser Bachelorarbeit allerdings nicht beschäftigt, da das Hauptaugenmerk auf die Erstellung des Planes gelegt wird. Abläufe aus Manipulation und Perzeption sind dabei bereits gegeben.
Ziel dieser Bachelorarbeit ist es, dass der Roboter autonom nach einem Objekt sucht und dieses Objekt anschließend korrekt auf dem zu deckenden Tisch abstellt. Um dieses Ziel zu erreichen, werden verschiedene Software-Systeme genutzt, welche in Kapitel 2 auf der nächsten Seite beschrieben werden. Anschließend werden die Grundlagen, auf welchen diese Arbeit aufbaut, beschrieben (siehe Kapitel 3 auf Seite 9), um danach die eigentliche Implementierung der Funktionen und des Planes zu beschreiben (siehe Kapitel 4 auf Seite 19).
Nachfolgend werden zwei Experimente vorgestellt, in denen geprüft wird, ob die gesetzten Ziele erreicht werden konnten (siehe Kapitel 5 auf Seite 33). Abschließend wird ein Fazit dieser Arbeit gezogen (siehe Kapitel 6 auf Seite 37) und darauf eingegangen, wie man auf dieser Arbeit aufbauen könnte (siehe Kapitel 7 auf Seite 39).

# Verwendete Software

Dieses Kapitel gibt eine kurze Übersicht über die Software-Systeme, welche innerhalb dieser Bachelorarbeit genutzt werden. Es wird dabei lediglich die Idee der Systeme beschrieben, eine genauere Beschreibung, wie die Systeme in dieser Bachelorarbeit genutzt werden, folgt im Kapitel Grundlagen (Kapitel 3 auf Seite 9).

## 2.1 CRAM

Für die Erstellung des Planes wird innerhalb dieser Bachelorarbeit die Plansprache `Cognitive Robot Abstract Machine`, kurz `CRAM`, genutzt. `CRAM` wurde dabei aus verschiedenen Gründen ausgewählt. Zum Einen sind die bereits vorhandenen Grundlagen, auf welchen diese Bachelorarbeit aufbaut, bereits in `CRAM` geschrieben. Außerdem bietet `CRAM` mit einem Reasoning-Verfahren einen wichtigen Baustein an, welcher innerhalb dieser Arbeit genutzt wird. Ein solches Reasoning-Verfahren wird beispielsweise von `SMACH` [1] nicht angeboten. `SMACH` ist eine weitere Möglichkeit, Pläne zu erstellen, und baut dabei auf der Idee von `State Machines` auf. Da das Reasoning Verfahren innerhalb dieser Bachelorarbeit jedoch ein wesentlicher Bestandteil ist, wurde davon abgesehen, `SMACH` als Verfahren zu nutzen.
Auf Grund dessen und, dass die bereits vorhandenen Strukturen in `CRAM` implementiert sind, wurde für diese Bachelorarbeit entschieden, ebenfalls `CRAM` für die Implementierung zu nutzen.
`CRAM` stellt dabei eine eigene Sprache, die `CRAM Plan Language` (`CPL`) zur Verfügung, welche in `LISP` [2] geschrieben ist. `CPL` ist eine Sprache, welche es erlaubt Pläne für Roboter zu schreiben, welche dynamisch ausgeführt werden. Vielmehr bietet `CPL` die Möglichkeit, ein Reasoning während der Planausführung durchzuführen, um auf Gegebenheiten der Umwelt zu reagieren [4].
Um dies zu tun, stellt `CRAM` ein eigenes Reasoning zu Verfügung, also die Möglichkeit, etwas aus gegebenen Daten zu schlussfolgern [4]. Hierfür werden so genannte `Prädikate` genutzt,

---

[1] SMACH Homepage: `http://wiki.ros.org/smach`
[2] Common-Lisp Homepage: `https://common-lisp.net/`

welche in Prolog [3] geschrieben werden. Solche Prädikate haben zunächst immer einen gleichen Aufbau, wie Codeabschnitt 2.1 zeigt.

*Listing 2.1: Aufbau eines Prädikates in CRAM*

```
(<- (PRAEDIKATNAME ATTRIBUT1 ATTRIBUT2))
```

Ein Beispiel für ein bereits bestehendes Prädikat [4] in CRAM zeigt der Codeabschnitt 2.2 . Dieses Prädikat gibt an, dass für das Essen soup eine bowl benötigt wird. Es ist mit Hilfe von diesen Prädikaten also möglich festzulegen, welche Objekte für ein gewünschtes Essen benötigt werden.

*Listing 2.2: Beispiel für ein Prädikat in CRAM welches für Mahlzeit soup das benötigte Objekt bowl angibt*

```
(<- (required-meal-object soup bowl))
```

Um die Informationen aus den Prädikaten extrahieren zu können, werden Prologabfragen benötigt. Diese können auch in normalen CRAM-Funktionen genutzt werden, um die Daten wie gewünscht zu verarbeiten.
In CRAM werden Instanzen wie Objekte durch sogenannte Designatoren beschrieben, welche wie eine Liste von Wert-Schlüssel-Paaren darstellbar sind. Der Codeabschnitt 2.3 stellt dabei beispielhaft dar, wie in CRAM ein Objektdesignator erstellt wird. Es wird dabei ein Objekt vom Typ milkbox für die Person Tim erstellt.

*Listing 2.3: Erstellung eines Objektdesignators vom Typ Milkbox für die Person Tim*

```
(obj (make-designator 'object `((desig-props::type milkbox)
                                 (desig-props::for-guest tim))))
```

In dieser Bachelorarbeit werden Designatoren, neben der Beschreibung von Objekten, auch für Beschreibung von Orten und Positionen genutzt. Ein solcher Designator wird im Verlaufe der Bachelorarbeit als Location-Designator beschrieben.

## 2.2 KnowRob

KnowRob ist eine Wissensdatenbank für Roboter. Es wurde bei der Umsetzung von KnowRob besonders darauf geachtet, dass es vor allem mit autonomen Robotern genutzt werden kann [5].

[3]SWI Prolog Homepage: http://www.swi-prolog.org/
[4]Quelle für das Prädikat: https://github.com/fairlight1337/cram_longterm_pickandplace/blob/master/src/table-setting.lisp

Das Herzstück von KnowRob ist dabei die Wissensbasis, mit welcher es möglich ist, verschiedenste Informationen in einer Datenbank abzuspeichern, sowie Relationen und Eigenschaften zu diesen Informationen zur Verfügung zu stellen [6]. Diese Informationen können über Methoden abgefragt werden und somit von einem Kontrollprogramm genutzt werden.
Neben der Abfrage von Informationen ist es weiterhin ebenfalls möglich, Daten während der Laufzeit zu dieser Datenbank hinzuzufügen [6]. Ein Kontrollprogramm kann also auf die Daten zugreifen, sowie sie erweitern und aktualisieren. Dies bringt die Möglichkeit, flexible Kontrollprogramme zu schaffen, welche auf die Gegebenheiten der Umwelt reagieren können. Es kann sich zum Beispiel ein Szenario vorgestellt werden, in welchem der Roboter eine Mahlzeit vorbereiten soll. Hierfür benötigt er zunächst Informationen über die Mahlzeit an sich, welche Objekte benötigt werden, wo sich diese befinden und weiteres.
Mit Hilfe von KnowRob kann das Kontrollprogramm nun leicht die gewünschten Daten zu gewünschter Zeit aus der Datenbank abfragen, weiterhin kann es auch in die Wissensdatenbank schreiben, welche Objekte der Roboter bereits abgearbeitet hat. Dies bietet den Vorteil, dass der Roboter zu jeder Zeit abfragen kann, welche Objekte noch fehlen, um seine Aufgabe abzuschließen.
Innerhalb dieser Arbeit wird KnowRob vor allem für die Nutzung einer semantischen Karte genutzt. Die semantische Karte speichert dabei alle wichtigen statischen Objekte, in diesem Fall die komplette Küchenumgebung aus dem Labor, in welcher der Roboter innerhalb dieser Bachelorarbeit bewegt wird. Innerhalb dieser Bachelorarbeit gibt es zwei wichtige, statische Positionen im Labor, an denen sich ein Objekt befinden kann. Zum Einen der `kitchen_sink_block`, welcher die Küchenzeile an der Wand beschreibt. Die zweite Position ist das `kitchen_island`, welche die Kücheninsel beschreibt und den zu deckenden Tisch darstellt.
Weiterhin ist es auch möglich, die Daten von gefunden Objekten, wie beispielsweise einer Milchpackung, in dieser semantischen Karte zu speichern und, wenn benötigt, diese Informationen abzurufen.
Die Daten über Objektpositionen sind zunächst essentiell nötig, um während einer Bewegung Kollisionen zu vermeiden. Außerdem kann der Roboter mit Hilfe der semantischen Karte berechnen, wo sich ein Objekt befindet. Befindet sich beispielsweise ein Objekte typischerweise auf dem `kitchen_sink_block`, so kann er über die Position dieses Tisches berechnen, an welchen Positionen sich das Objekt befinden kann, um es auf dem `kitchen_sink_block` zu suchen.
Des Weiteren werden die Positionen der Tische im Verlaufe dieser Bachelorarbeit benötigt, um Entfernungen zu berechnen. Genaueres dazu wird im Kapitel 4.2.6 auf Seite 24 beschrieben.

*Kapitel 3*

# Grundlagen

In diesem Kapitel werden die Grundlagen, welche im Rahmen dieser Bachelorarbeit genutzt werden, genauer beschrieben. So wird zunächst auf die genutzten Grundlagen von `CRAM` eingegangen, während anschließend `Behaviour Trees` beschrieben und erläutert werden. Es wird die grundlegende Struktur dieser erklärt und erläutert, wieso `Behaviour Trees` gut geeignet sind, um die `CRAM`-Implementierungen dieser Bachelorarbeit darzustellen.

## 3.1 CRAM und LISP Grundlagen

Bevor die genutzten Implementierungen, welche bereits vor Beginn dieser Bachelorarbeit in `CRAM` vorhanden waren, beschrieben werden, werden kurz grundlegende Konzepte aus `CRAM` und `LISP` erläutert. Diese erläuterten Konzepte werden innerhalb der Bachelorarbeit verwendet und für das Verständnis einiger Implementierungen benötigt.
Eine wichtige Funktion, welche von `CRAM` angeboten wird, sind die `lazy lists`. Diese Listenart hat gegenüber normalen Listen den großen Vorteil, dass nicht alle Listenelemente ausgegeben werden, sondern immer nur das gerade verwendete Element. Wenn ein Objekt beispielsweise drei verschiedene Positionen hat, an denen sich das Objekt befinden kann, so gibt eine normale Liste direkt alle drei Positionen aus. Allerdings ist es in diesem Fall sinnvoller, `lazy lists` zu nutzen. Diese geben zunächst nur das erste Objekt aus, in diesem Fall die erste Position.

*Listing 3.1: Ausgabe einer normalen Liste. Es werden direkt alle Listeneinträge angezeigt.*

```
(((?OBJ-LOC . "fridge1")) ((?OBJ-LOC . "kitchen_sink_block")))
```

Der Roboter kann nun zunächst die angegebene Position prüfen und wenn er das Objekt dort nicht findet, die nächste Position aus der Liste anfordern. Der Codeabschnitt 3.1 zeigt dabei eine normale Liste, während der Codeabschnitt 3.2 eine `lazy list` darstellt.

*Listing 3.2: Ausgabe einer Lazy List für die gleichen Informationen. Es wird zunächst nur das erste Listenelement angezeigt.*

```
(((?OBJ-LOC . "fridge1"))
.#S(CRAM-UTILITIES::LAZY-CONS-ELEM:GENERATOR#<CLOSURE#{100C4553EB}>))
```

Mit Hilfe von `lazy lists` ist es also möglich, Rechenaufwand zu sparen und Informationen nur auszugeben, wenn sie auch benötigt werden. Dieses Verhalten bietet gerade bei größeren Listen einen Vorteil. Innerhalb dieser Bachelorarbeit wird von mehreren Funktionen das Konzept der `lazy lists` genutzt.

### 3.1.1 Genutzte CRAM Implementierungen

In diesem Kapitel wird auf die genutzten `CRAM`-Implemntierungen eingegangen, welche bereits zu Beginn der Bachelorarbeit zur Verfügung standen. Außerdem wird beschrieben, wie Funktionen innerhalb dieser Bachelorarbeit angepasst wurden, um die gewünschte Funktionalität zu erhalten.

Die Implementierungen, welche in dieser Arbeit genutzt werden, werden von dem `CRAM`-Paket `cram_longterm_pickandplace`[1] bereitgestellt.
Innerhalb dieses Paketes gibt es bereits viele Grundlagen, welche innerhalb dieser Arbeit genutzt werden. Im folgenden wird ein kurzer Überblick über die Methoden und Funktionen beschrieben.

#### 3.1.1.1 required-scene-objects

Um die Prädikate, welche die benötigten Objekte für eine Szene angeben, zu extrahieren, wird `required-scene-objects` genutzt. Hierzu wird das CRAM-Reasoning genutzt, indem innerhalb dieser Funktion eine Prologabfrage genutzt wird (siehe 3.3). Der Aufruf **required-object ?object** gibt dabei ein weiteres Prädikat an, welches alle nötigen Informationen für das `object` aus den vorhanden Prädikaten ausliest. Die Funktion `required-scene-objects` gibt am Ende eine Liste aus allen benötigten Objekten für die ausgewählte Szene zurück.

*Listing 3.3: CRAM-Reasoning Aufruf welcher die benötigten Objekte aus für eine Szene ausgibt*

```
(crs:prolog `(required-object ?object))
```

[1] Original Github-Repository: `https://github.com/fairlight1337/cram_longterm_pickandplace`

#### 3.1.1.2 perceive-a

Die erste Funktion, welche im Plan genutzt wird, ist `perceive-a`. Diese Funktion wird genutzt, um Objekte zu erfassen. Es gibt dabei verschiedene Möglichkeiten, sie zu nutzen.
Zunächst kann angegeben werden, welches Objekt gesucht wird, und `perceive-a` gibt dieses zurück, wenn es erfasst wurde. Wenn das Objekt nicht gefunden wird, wird `nil` zurückgegeben.
Außerdem ist es möglich, anzugeben, ob der Kopf beim Erfassen bewegt werden darf.
Neben diesen beiden Möglichkeiten kann auch angegeben werden, ob `perceive-a` bis zum Erfolg versucht, ein Objekt zu finden oder ob nach drei Versuchen abgebrochen werden soll. Dieses Verhalten wurde innerhalb der Bachelorarbeit implementiert, da die Objekte an verschiedenen Positionen platziert sein können und es somit nicht zu einer Endlosschleife innerhalb von `perceive-a` kommt. Um jedoch auch zu gewährleisten, dass die bisherigen Funktionsaufrufe für andere Pläne nicht beeinträchtigt werden, wurde die Implementierung mit einem stationären Attribut, `ignore-object-not-found`, durchgeführt.
Somit ist es möglich, `perceive-a` weiterhin wie gewohnt aufzurufen.
Es gibt allerdings die Möglichkeit, den Wert von `ignore-object-not-found` auf `t`, also `True` zu setzen. Dies hat zur Folge, dass innerhalb von `perceive-a` der Wert `retry-task`, welcher auf drei gesetzt ist, genutzt wird. Es wird in diesem Fall lediglich dreimal versucht, ein Objekt an dieser Position zu erkennen.

#### 3.1.1.3 pick-object

Die nächste genutzte Funktion ist `pick-object`, welche ein übergebenes Objekt greift.
Auch in dieser Methode wurde ein stationäres Attribut `ignore-object-not-found` hinzugefügt, ähnlich wie bei `perceive-a`. Dieses wird ebenfalls dazu genutzt, maximal dreimal zu versuchen, das Objekt zu greifen und danach abzubrechen. Dies ist nötig, da so verhindert wird, dass eine Endlosschleife entstehen kann, falls das Objekt fehlerhaft erkannt wurde.

#### 3.1.1.4 place-object

Um ein gegriffenes Objekt abzustellen, wurde die Funktion `place-object` genutzt. Es wird dabei neben dem Objekt, welches abgestellt werden soll, auch angegeben, an welcher Position das Objekt abgestellt werden soll. Diese Position wird als `Location-Designator` angegeben.

Wie bei der Funktion `place-object`, besteht auch bei der Funktion `pick-object` die Möglichkeit anzugeben, ob der Roboter sich während des Vorganges bewegen darf.

### 3.1.2 Genutzte Prädikate

Wie bereits in Kapitel 2.1 auf Seite 5 beschrieben, bietet `CRAM` ein eigenes Reasoning-Verfahren an, welches innerhalb dieser Bachelorarbeit genutzt wird. Im folgenden wird grundlegend auf die genutzten Prädikate eingegangen. Dabei wird nicht die genaue Implementierung dieser beschrieben, sondern es wird dargelegt, welche Prädikate vorhanden sind und wofür diese genutzt werden. Es werden dabei ebenfalls nicht alle Prädikate erläutert, sondern lediglich wichtige, ausgewählte Prädikate beschrieben.
Zunächst wird das Prädikat `preference` beschrieben, mit welchem verschiedene Informationen abgerufen werden können. Zum Einen ist es möglich, mit dem Prädikat die Sitzposition einer Person zu speichern und abzurufen. Hierzu muss innerhalb des Prädikates das zweite Attribut auf `seat` gesetzt werden.
Wenn das zweite Attribut jedoch auf `dish` gesetzt ist, gibt dies die Mahlzeit an, welches eine Person essen möchte. Anhand des Prädikats `preference` werden also die Sitzposition sowie die Mahlzeit einer Person gespeichert.
Das zweite, wichtige Prädikate ist `required-meal-object`. Mittels diesem Prädikat ist es möglich zu erfahren, welche Objekte und Gegenstände für eine Mahlzeit benötigt werden.

## 3.2 Darstellungsmethode Behaviour Trees

Im Rahmen dieser Bachelorarbeit werden `Behaviour Trees` als Darstellungsmittel genutzt, um verschiedene Ansätze der Implementierung zu veranschaulichen. In diesem Kapitel werden zunächst `Behaviour Trees` an sich beschrieben und erklärt. Außerdem wird dargelegt, wieso `Behaviour Trees` gut geeignet sind, um die Strukturen des implementierten `CRAM`-Planes zu veranschaulichen.

### 3.2.1 Behaviour Trees

Im folgenden Kapitel wird nun zunächst die allgemeine Funktionsweise von `Behaviour Trees` erläutert. Im Anschluss werden ausgewählte Komponenten dieser vorgestellt. Die Komponenten wurden anhand dessen ausgewählt, ob sie im weiteren Verlauf der Bachelorarbeit zur Veranschaulichung des Plans benötigt werden.

`Behaviour Trees` wurden erstmals Mitte 2000 in der Informatik eingesetzt, Geoff Dromey nutze sie im Bereich der Software Entwicklung [7]. Dabei boten die `Behaviour Trees` einen relativ leichten Umgang mit Aktionen und Vorbedingungen, welche für gewisse Aktionen erfüllt sein müssen. Dies hatte zur Folge, dass der Ansatz der `Behaviour Trees` auch innerhalb von Spielen wie `Halo 2` oder `Spore` genutzt wurde [7].
`Behaviour Trees` können dazu hierarchisch aufgebaut werden, sodass an der Wurzel die übergeordneten `tasks` stehen und, je tiefer es in den Baum hineingeht, konkretere `tasks` beschrieben werden. In [7] wird zum Beispiel ein `Behaviour Tree` für das Verhalten eines

Wachhundes innerhalb eines Computerspiel genutzt, wie Abbildung 3.1 zeigt.
Die Highlevel tasks, also die tasks an der Wurzel, sind in diesem Falle patrol, investigate und attack, während die nachfolgenden tasks beispielsweise bite sind. Hierbei sieht man bereits den angesprochenen Unterschied zwischen den einzelnen tasks. Während attack noch ein übergeordnetes Verhalten ist, da ein Hund auf verschiedene Weisen attackieren kann, ist die nachfolgende Aktion bite ein konkreteres Verhalten.

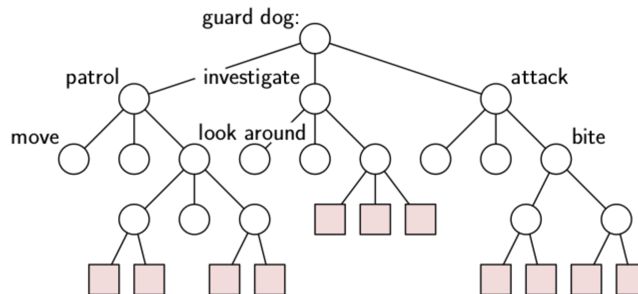


***Abbildung 3.1:*** *Ein hierarchischer Behaviour Trees am Beispiel eines Wachhundes* [2]

Somit ist es möglich, die tasks, welche der Wachhund durchführt, immer spezifischer zu gestalten. Am Ende dieser Aktion, also an den Blättern, stehen dann Interaktionen mit dem Spiel an sich.
Um diese Blätter zu erreichen, müssen conditions erfüllt werden. Zum Beispiel ist der Hund verletzt? Wenn dies der Fall ist, kann er nicht in den Zustand attack, da er für diesen gesund sein muss. Wichtig hierbei ist, dass die conditions nicht veränderbar sind.
Im Gegensatz dazu können die actions Änderungen am Spiel vornehmen. Eine Action kann zum Beispiel das Aufheben eines Gegenstandes sein. Das Zusammenspiel von conditions und actions wird task genannt. Tasks haben dabei ein Set von conditions, welche für die Ausführung zutreffen müssen. Wenn alle conditions zutreffen, führt der task actions aus, welche das Verhalten des tasks darstellen [7].
Nachfolgend werden zwei konkrete Komponenten eines Behaviour Trees, der Selector und die Sequence, beschrieben.

### 3.2.1.1 Selector

In diesem Abschnitt wird ein Selector beschrieben. Zur Verdeutlichung wird folgendes Szenario angenommen:
Es soll geprüft werden, ob in einer Liste von Objekten das Objekt Milch vorhanden ist. In der Liste befinden sich die Objekte Löffel, Müsli, Milch und Schüssel, wobei die Reihenfolge der Aufzählung gleicht.
Ein Selector, wie in Abbildung 3.2 auf der nächsten Seite, prüft nun nacheinander die einzelnen Möglichkeiten. In diesem konkreten Beispiel wird nun zunächst das erste Objekt der

[2] Bild aus: Artificial Intelligence for Games: Decision Making [7]

Liste, also der Löffel, mit der `Milch` verglichen. Da diese aber nicht äquivalent sind, wird das nächste Objekt, das Müsli geprüft. Da auch das Müsli nicht die `Milch` ist, wird nun das dritte Objekt der Liste geprüft.
Der Selector hat in diesem Fall nun das Objekt `Milch` gefunden. In diesem Fall beendet sich der Selector mit der Rückgabe, dass das Objekt in der Liste gefunden wurde. Alle weiteren Möglichkeiten, in diesem Fall das Objekt `Schüssel`, werden nicht weiter betrachtet. Ein Selector prüft also nur solange, bis er den **ersten** Treffer landet.

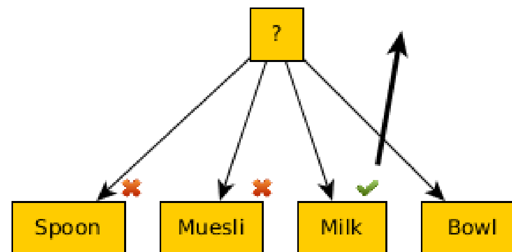


***Abbildung 3.2:*** *Beispiel für einen Behaviour Tree Selector, welcher das Objekt Milch aus einer Liste finden soll*

Wenn man nun annimmt, dass zum Beispiel ein `Pfannkuchenmix` in der Liste gesucht werden würde, so würde der `Selector` alle Möglichkeiten prüfen und am Ende zurückgeben, dass sich der `Pfannkuchenmix` nicht in der Liste befindet.

#### 3.2.1.2 Sequence

Nachdem im vorherigen Abschnitt der `Selector` beschrieben wurde, wird nun eine `Sequence` erläutert. Auch hierfür wird erneut ein Szenario angenommen, welches in Abbildung 3.3 auf der nächsten Seite dargestellt wird:
Der Roboter soll eine Milch greifen und sie an einer neuen Position wieder abstellen. Hierfür benötigt er die Aktionen `grasp`, `move` und `place`.
Eine `Sequence` zeichnet sich dadurch aus, dass sie nur gültig ist, wenn **alle** Aktionen ausgeführt werden konnten.
Anhand des gegebenen Szenarios könnte es folgenden Ablauf geben:
Der Roboter greift die Milch und fährt zu der Position, an der die Milch abgestellt werden soll. Nun soll der Roboter die Milch abstellen, allerdings ist der Platz, an dem die Milch stehen soll, bereits durch ein anderes Objekt belegt. Die `Sequence` schlägt nun fehl, da `place` nicht ausgeführt werden kann. Im Gegensatz zum `Selector` wird nun auch nicht mehr die nächste `Action`, in diesem Fall `placed = t` ausgeführt. Es muss für eine erfolgreiche `Sequence` also jede `Action` korrekt ausgeführt werden. In jedem anderen Fall schläft die `Sequence` fehl.

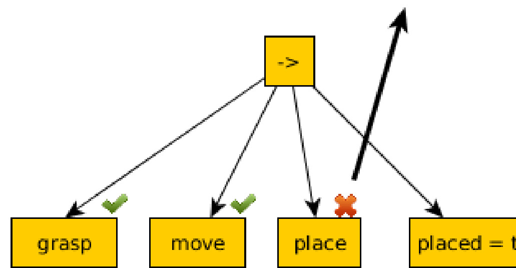


***Abbildung 3.3:*** *Beispiel für eine Behaviour Tree Sequence*

### 3.2.2 Behaviour Trees und CRAM

In diesem Abschnitt wird kurz darauf eingegangen, aus welchen Gründen sich dazu entschieden wurde, innerhalb dieser Bachelorarbeit die `Behaviour Trees` zu nutzen. Zunächst ist es wichtig klarzustellen , dass innerhalb dieser Bachelorarbeit `Behaviour Trees` nicht implementiert werden, sondern lediglich als Darstellungsmöglichkeit genutzt werden, um den geschrieben Plan veranschaulicht beschreiben zu können.

Frage an Jan: kann ich das so schreiben?

`Behaviour Trees` lassen sich bei `CRAM` gut als Darstellungsmittel nutzen, da `CRAM`-Pläne ebenfalls `conditions` nutzen können, um zu entscheiden, welche Aktion am sinnvollsten ausgeführt werden soll.
Außerdem sind die bereits angebrachten Beispiele für eine `Sequence` und einen `Selector` ebenfalls gute Beispiele dafür, dass `Behaviour Trees` sich innerhalb dieser Arbeit anbieten, genutzt zu werden. Denn die dort beschriebenen Beispiele sind grundlegende Aktionen und Abläufe, die innerhalb dieser Bachelorarbeit genutzt wurden.
Auf Grund dessen wurde entschieden, dass `Behaviour Trees` eine gute und elegante Möglichkeit sind, die Komplexität des Planes zu veranschaulichen. Dies kann an zwei Beispielen belegt werden. Das erste Beispiel vergleicht zunächst eine `CRAM` Implementierung für eine Pick und Place Aktion (siehe Codeabschnitt 3.4 auf der nächsten Seite) mit einer `Sequence` aus einem `Behaviour Tree`, welche in Abbildung 3.4 auf der nächsten Seite dargestellt ist. Die `Sequence` stellt dabei anschaulich dar, welche Funktionalität die implementierte `CRAM` Funktion besitzt.

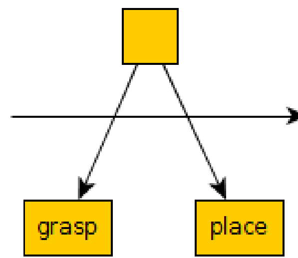


***Abbildung 3.4:*** *Beispiel für einen Sequence, welche erst ein Objekt greifen und dieses anschließend wieder abstellen soll*

Frage an Jan: progn führt ja place theoretisch auch aus, wenn pick nicht klappt. Trotzdem ok?

*Listing 3.4: Eine Funktion welche ein Objekt greif und anschließend abstellt*

```
(defun bt-sequence (obj)
    (progn
      (pick-object obj)
      (place-object obj bring-to))))
```

Als zweites Beispiel wurde innerhalb von `CRAM` eine Funktion implementiert (siehe (siehe Codeabschnitt 3.5)), welche das erste Element einer Liste überprüfen soll und entsprechend von diesem eine Aktion starten soll. Diese Implementierung stellt dabei abstrakt dar, wie entschieden wird, welcher Tisch näher am Roboter ist.
Diese Implementierung kann nun ebenfalls mit Hilfe der `Behaviour Trees` dargestellt werden, indem ein `Selector` genutzt wird (siehe Abbildung 3.5). An diesem Beispiel ist ebenfalls zu sehen, dass der `Selector` die Implementierung anschaulich darstellt.

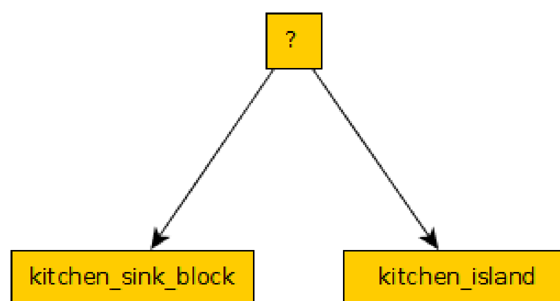


***Abbildung 3.5:*** *Beispiel für einen Selector, welcher auswählt, ob der kitchen_sink_block näher am Roboter ist als das kitchen_island*

*Listing 3.5: Eine Funktion welche aus einer Liste das erste Element prüft und entsprechend eine Aktion startet*

```
(defun bt-selector (loc-list)
  (cond ((equal (first loc-list) "kitchen_island")
            ''Drive to kitchen_island'')
        ((equal (first loc-list) "kitchen_sink_block")
            ''Search at kitchen_sink_block''))))
```

*Kapitel 4*

# Implementierung

Innerhalb dieses Kapitels wird die Implementierung, welche innerhalb dieser Bachelorarbeit erarbeitet wurde, vorgestellt. Es werden dabei zunächst zwei ausgewählte Szenen beschrieben, welche innerhalb dieser Bachelorarbeit implementiert und genutzt wurden. Im Zuge dessen wird zunächst dargestellt, wie Szenen grundsätzlich implementiert werden, um daraufhin auf die Beschreibung der Szenen einzugehen.
Anschließend wird auf die implementierten Funktionen eingegangen, welche genutzt werden, um das Ziel dieser Bachelorarbeit zu erreichen. Nachdem diese beschrieben und deren Verhalten dargelegt wurde, wird auf den Plan eingegangen, welcher innerhalb dieser Bachelorarbeit erstellt worden sind, um die beschriebenen Ziele zu erreichen. In diesem Zusammenhang werden, wie bereits angesprochen, `Behaviour Trees` als Darstellungsmittel genutzt. Die Implementierungen, welche innerhalb dieser Bachelorarbeit erarbeitet wurden, sind dabei in einem eigenen Repository [1] zu finden.

## 4.1 Übersicht der Szenen

Innerhalb dieser Bachelorarbeit wurden aus Testzwecken verschiedene Szenen geschrieben. Da diese jedoch zum Großteil nicht relevant für den Inhalt dieser Bachelorarbeit sind, werden nur ausgewählte Szenen beschrieben. Zum einen wird die Szene `set-scene-thomasthesis-experiment-01` beschrieben, welche für die Experimente genutzt wird. Außerdem wird die Szene `set-scene-thomasthesis-test-02` beschrieben, da diese für die Tests (siehe Kapitel 4.2.14 auf Seite 26) innerhalb dieser Bachelorarbeit genutzt wurde.
Zunächst wird jedoch kurz beschrieben, wie Szenen aufgebaut sind.

[1] Geforktes Github-Repository: `https://github.com/thocar/cram_longterm_pickandplace`

### 4.1.1 Aufbau einer Szene

Eine Szene benötigt mehrere Informationen, damit sie genutzt werden kann. Zunächst gibt es eine Funktion, welche die Szene als aktuelle Szene auswählt (siehe Codeabschnitt 4.1). In dieser Funktion muss festgelegt werden, welcher `Gast` am Tisch sitzt, welche `Tageszeit` gerade ist, also für was gedeckt werden soll, und für welchen `Tag` gedeckt werden soll. Letztere Information wird jedoch derzeit noch nicht betrachtet.

*Listing 4.1: Beispiel für eine definierte Funktion um eine Szene festzulegen*

```
(defun set-scene-1 ()
  (set-scene-detail 'guests `(bob))
  (set-scene-detail 'meal-time 'breakfast)
  (set-scene-detail 'week-day 'saturday))
```

Nun müssen die Informationen verarbeitet werden. Hierzu werden die Prädikate vom `CRAM-Reasoning` genutzt. Dabei werden unter anderem die Informationen zusammengefasst. In Codeabschnitt 4.2 wird zum Beispiel dargestellt, welche Mahlzeit die Person Bob zum Frühstück essen möchte.

*Listing 4.2: Beispiel für eine Zuweisung einer Mahlzeit für ein Frühstück für Bob*

```
(<- (preference bob dish cornflakes)
        (context-prop meal-time breakfast))
```

Aus diesen Informationen kann nun wiederum geschlussfolgert werden, welche Objekte und Gegenstände aufgedeckt werden müssen (siehe Codeabschnitt 4.3).

*Listing 4.3: Beispiel für eine Zuweisung der benötigten Objekte für die Mahlzeit Cornflakes*

```
(<- (required-meal-object cornflakes bowl))
(<- (required-meal-object cornflakes cornflakes))
(<- (required-meal-object cornflakes milkbox))
(<- (required-meal-object cornflakes spoon))
```

Über die verschiedenen Prädikate können nun alle benötigten Objekte hergeleitet werden. Für das oben dargestellte Beispiel würden also die Objekte `bowl`, `cornflakes`, `milkbox` und `spoon` benötigt werden.

Neben den gezeigten Prädikaten gibt es noch weitere Prädikate, wie beispielsweise die Zuweisung eines Objektes an eine Position auf dem zu deckenden Tisch, auf welchem Platz welche Person sitzt oder an welchen Orten ein Objekt üblicherweise gelagert wird. Allerdings würde es den Rahmen dieser Bachelorarbeit überschreiten, für alle Prädikate ein Beispiel zu geben und diese zu erläutern, da der Aufbau eines Prädikates immer dem selben Muster folgt.

### 4.1.2 set-scene-thomasthesis-experiment-01

Die Szene für die Experimente wurde auf ein Objekt beschränkt, da dies ausreichend ist, um die Funktionalität des Planes zu zeigen. Für die Experimente soll so lediglich eine Packung Milch für Tim auf dem `kitchen_island` abgestellt werden. Es wurde als Objekt dabei die Milch ausgewählt, weil diese von der Perception und der Manipulation am sichersten verarbeitet werden kann und somit weniger Probleme, die unabhängig von diesem System bestehen, auftreten. Der Codeabschnitt 4.4 zeigt die ausgewählten Prädikate, welche Teile dieser Szene spezifizieren.

*Listing 4.4: Ausgewählte Prädikate für die Szene für die Experimente*

```
(<- (preference tim dish muesli)
        (context-prop meal-time breakfast))
....
(<- (required-meal-object muesli milkbox))
```

Für die Position der Milch auf dem zu deckenden Tisch wurde der Standardwert gesetzt. Die Milch soll also möglichst weit vorn auf dem Tisch stehen, damit man diese gut erreichen kann.

### 4.1.3 set-scene-thomasthesis-test-02

Diese Szene wurde lediglich für das Testen von verschiedenen Funktionen, ohne einen Roboter zu nutzen, geschrieben. Sie enthält vier verschiedene Objekte für die Person `Bob`, nämlich einen `Teller`, ein `Messer`, eine `Gabel` sowie einen `Pizzaschneider`. Die Prädikate sind dabei analog zu den bisherigen geschrieben worden.

## 4.2 Übersicht der Funktionen

Im folgenden Kapitel werden die im Laufe der Bachelorarbeit, implementierten Funktionen beschrieben und erläutert. Dabei wird zunächst eine Übersicht gegeben und im Anschluss auf die einzelnen Funktionen eingegangen. Die Funktionen wurden dabei teilweise in Zusammenarbeit mit dem Betreuer implementiert.
Innerhalb dieser Bachelorarbeit wurden die folgenden Funktionen implementiert:

| Funktionsname | Kapitelreferenz |
|---|---|
| extract-objectdesig-and-bringto | 4.2.1 |
| get-objectlocation-from-object | 4.2.2 |
| search-object-at-object-locations | 4.2.3 auf der nächsten Seite |
| current-robot-pose | 4.2.4 auf der nächsten Seite |
| semantic-object-pose | 4.2.5 auf der nächsten Seite |
| distance-to-semantic-object | 4.2.6 auf Seite 24 |
| sorted-semantic-object-distances | 4.2.7 auf Seite 24 |
| in-front-of | 4.2.8 auf Seite 24 |
| go-in-front-of-island | 4.2.9 auf Seite 24 |
| look-at-pose | 4.2.10 auf Seite 25 |
| perceive-all | 4.2.11 auf Seite 25 |
| perceive-table | 4.2.12 auf Seite 25 |
| object-in-list | 4.2.13 auf Seite 25 |
| Testfunktionen | 4.2.14 auf Seite 26 |

***Tabelle 4.1:*** *Übersicht der innerhalb der Bachelorarbeit implementierten Funktionen*

### 4.2.1 extract-objectdesig-and-bringto

Die Funktion `extract-objectdesig-and-bringto` extrahiert wichtige Informationen für den Plan aus den Prädikaten.
Diese Informationen werden als Object- beziehungsweise Location-Designator in einer Liste abgespeichert. Der `Object-Deisgnator`, welcher im Laufe der Bachelorarbeit lediglich mit `object-designator` oder `object-desig` beschrieben wird, enthält dabei die Informationen, um welches Objekt es sich handelt und für welche Person dieses Objekt benötigt wird.
Der `Location-Designator`, im Laufe der Bachelorarbeit wird dieser meist mit `bring-to` beschrieben, enthält hingegen die Information, an welchem Ort das Objekt abgestellt werden soll, also auf welchen Tisch sowie an welcher Position auf diesem Tisch.
Am Ende wird eine Liste mit allen benötigten Objekten zurückgegeben, wobei es für jedes Objekt eine eigene Liste gibt, in denen die Informationen gespeichert sind.

### 4.2.2 get-objectlocation-from-object

Die Positionen, an denen ein Objekt gelagert sein kann, wurden mit Hilfe von Prädikaten dargestellt, wie der Codeabschnitt 4.5 zeigt.
Diese Prädikate wurden analog für die weiteren, in dieser Bachelorarbeit verwendeten Objekte geschrieben. Außerdem wurde als Standardposition `kitchen_sink_block` festgelegt.
Die Funktion `get-objectlocation-from-object` verarbeitet diese Prädikate und gibt eine Liste von Positionen für ein Objekt zurück.

*Listing 4.5: Beispielprädikat für die Positionszuweisung für das Objekt milkbox*

```
(<- (object-position milkbox "fridge1"))
```

Des Weiteren gibt es Prädikate, welche die verschiedenen Positionen genauer definieren (siehe Codeabschnitt 4.6).

*Listing 4.6: Beispielprädikat für die Definition für die Position fridge1*

```
(<- (location-details "fridge1" (in container)))
```

### 4.2.3 search-object-at-object-locations

Mit Hilfe der Liste von möglichen Positionen von Objekten kann der Roboter nun das Objekt suchen. Dies wird mit Hilfe der Funktion `search-object-at-object-locations` durchgeführt.
Dazu wird innerhalb der Funktion die Liste der Positionen durchgegangen. Dabei wird bei dem Objekt, welches gesucht wird, der Location-Designator `at` hinzugefügt. In diesem wird gespeichert, wo sich - nach der derzeitigen Annahme - das Objekt befindet.
Im Folgenden wird dann `perceive-a` mit diesem Objekt aufgerufen und die Rückgabe von `perceive-a` wird an den Plan weitergereicht. Die Rückgabe kann dabei entweder `nil`, wenn das Objekt nicht erkannt wurde, oder das Objekt mit der genauen Position sein.
Bei den Orten `fridge1` und `drawer1` wird derzeit lediglich eine Warnung ausgegeben und anschließend die nächste Position überprüft, da es zum Zeitpunkt der Implementierung dieses Planes nicht möglich war, den Kühlschrank oder die Schublade zu öffnen.

### 4.2.4 current-robot-pose

Um im späteren Verlauf die Distanz zwischen dem Roboter und verschiedenen Objekten, zum Beispiel einem Tisch, zu ermitteln, wird die aktuelle Roboterposition benötigt. Mit dieser Funktion wird ein `Pose-Stamped` mit der aktuellen Roboterposition erstellt.

### 4.2.5 semantic-object-pose

Neben der Roboterposition wird natürlich auch die Position des Objektes benötigt, zu welchem eine Distanz berechnet werden soll. Diese Objektposition wird dabei mit `semantic-object-pose` erstellt, indem der Funktion der Name des Objektes übergeben wird.
Die Funktion holt sich nun aus der semantischen Karte die Pose, welche das Objekt beschreibt.

### 4.2.6 distance-to-semantic-object

Mit Hilfe der aktuellen Roboterposition sowie der Position des Objektes kann nun eine Distanz berechnet werden. Dabei wird das Objekt, zu welcher die Distanz berechnet werden soll, an die Funktion übergeben. Die Distanz zwischen den beiden Positionen wird nun mit Hilfe von `tf` (weiteres siehe in Kapitel 4.4 auf Seite 31) berechnet, indem die Vektor Distanz zwischen den beiden Positionen berechnet wird. Als Ergebnis wird eine Dezimalzahl zurückgegeben, welche die Distanz zwischen den beiden Positionen darstellt.

### 4.2.7 sorted-semantic-object-distances

Es ist nun möglich, mit der Funktion `distance-to-semantic-object` eine Distanz zwischen dem Roboter und einem Objekt zu berechnen. Diese Funktion kann nun genutzt werden, um von einer Liste von Objekten die Distanz zum Roboter zu berechnen. Die Ergebnisse werden dabei in einer neuen Liste gespeichert, welche nach der Größe der Distanz sortiert ist. Es ist also das Objekt mit der geringsten Entfernung an der ersten Stelle und das Objekt mit der größten Entfernung an der letzten Stelle.
Somit ist es möglich zu berechnen, welches Objekt die kleinste Distanz zum Roboter hat.

### 4.2.8 in-front-of

Die Funktion `in-front-of` bietet die Möglichkeit, den Roboter an eine gewünschte Position zu fahren. Um diese Aktion ausführen zu können, benötigt die Funktion eine Position, sowie einen Boolean für `try-indefinitely`. Dieser gibt an, ob unbegrenzt lange versucht werden soll, die Position zu erreichen.
Wenn dieser Wert auf `t` gesetzt ist, wird die Navigation solange ausgeführt, bis die angegeben Position mit dem Roboter erreicht wurde. Andernfalls wird der Navigationsablauf lediglich einmal ausgeführt. Außerdem stellt der Roboter sicher, dass er im nachfolgenden Ablauf an dieser Position bleibt, bis eine neue Position angefordert wird.

### 4.2.9 go-in-front-of-island

Um nun vor das `kitchen_island` zu fahren, wird das Makro `in-front-of` genutzt. Dieses wird mit einer festen Pose, welche die Position vor dem `kitchen_island` beschreibt, aufgerufen. Außerdem wird `try-indefinitely` auf `nil` gesetzt, da der Roboter für das erfassen der Objekte auf dem Tisch keine exakt korrekte Position benötigt. Es reicht aus, dass der Roboter sich an einer Position befindet, aus welcher er auf den Tisch gucken kann.

### 4.2.10 look-at-pose

Da es möglich ist, dass ein Objekt nicht an einer Position zu finden ist, an welcher der Roboter dies erwartet, wurde die Funktion `look-at-pose` geschrieben. Diese ermöglicht es, dass der Roboter seinen Blick auf eine übergebene Position richtet. Die Position muss dabei als `Pose` übergeben werden.

### 4.2.11 perceive-all

Die Funktion `perceive-all` gibt alle Objekte, welche dem übergebendem Objekttyp entsprechen, in einer Liste zurück. Im Gegensatz zu `perceive-a` wird also nicht nur ein Objekt zurückgegeben, wenn mehrere, gleiche Objekte erkannt werden.
Außerdem ist es möglich, alle Objekte, welche der Roboter im Blickfeld hat, in einer Liste zu speichern. Dies geschieht, indem als Objekt `nil` übergeben wird, also wenn kein bestimmtes Objekt gesucht wird.

### 4.2.12 perceive-table

Mit Hilfe von `look-at-pose` und `perceive-all` können nun Objekte auf dem `kitchen_island` erkannt werden. Zunächst wird `look-at-pose` mit der Pose vom `kitchen_island` aufgerufen. Die Pose wird über die Funktion `semantic-object-pose` berechnet.
Wenn der Roboter nun auf den Tisch schaut, wird `perceive-all` mit einem `nil` Objekt aufgerufen und die Liste, welche `perceive-all` ausgibt, zurückgegeben.

### 4.2.13 object-in-list

Wenn nun `perceive-table` ausgeführt worden ist, hat der Roboter lediglich eine Liste von Objekten, welche sich auf dem `kitchen_island` befinden. Um nun herauszufinden, ob in dieser Liste das gesuchte Objekt ist, wird die Funktion `object-in-list` genutzt.
Diese benötigt einen Object-Designator von dem gesuchten Objekt und eine Liste von Objekten, beispielsweise die von `perceive-table`. Wenn sich ein Objekt vom gleichen Typ wie des gesuchten Objektes in der Liste befindet, wird dieses Objekt aus der Liste zurückgegeben. Andernfalls wird `nil` zurückgegeben. Um dieses Verhalten zu erreichen, wird die `find` Funktion genutzt.
Somit ist es auch möglich, diese Funktion unabhängig von `perceive-table` zu nutzen.

### 4.2.14 Testfunktionen

Innerhalb dieses Kapitels werden kurz und prägnant die implementierten Testfunktionen beschrieben, welche die Teile der implementierten Funktionen testen, ohne dass eine Verbindung zu einem Roboter nötig ist.

Als erste Testfunktion wird `find-object` beschrieben. Diese Funktion extrahiert mittels der bereits beschriebenen Funktion `get-objectlocation-from-object` zunächst alle möglichen Positionen, an denen das Objekt zu finden ist. Das Verhalten dieser Funktion ähnelt dabei der Funktion `search-object-at-object-locations`, mit dem Unterschied, dass nicht versucht wird, das Objekt zu erkennen, sondern lediglich das Objekt mit der neuen `at-Location` zurückgegeben wird. So kann geprüft werden, ob das Erstellen und Updaten der Designatoren korrekt funktioniert.
Die Funktionen `test-scene-one-object` und `test-scene-experiment-01` nutzen lediglich die Funktion `find-object`, um das Verhalten bei verschiedenen Szenen zu prüfen.

Die nächste Testfunktion ist `test-object-in-list`. Hierfür wird ein beliebiges Objekt erstellt und geprüft, ob sich dieses in einer Liste von Objekten befindet. Diese Liste enthält dabei lediglich alle Objekte der Szene `set-scene-thomasthesis-test-02` (siehe Kapitel 4.1.3 auf Seite 21). Erstellt man nun ein Objekt mit dem Typ `milkbox`, so ist das Resultat `nil`, da in der Szene keine Milch benötigt wird.
Wird allerdings ein Objektdesignator mit dem Typ `knife` erstellt, so wird die Rückgabe, welche in 4.7 dargestellt ist, ausgegeben, da das Objekt in der Liste gefunden worden ist.

*Listing 4.7: Rückgabe von test-object-in-list wenn sich das Objekt in der gegebenen Szene befindet*

```
(#<OBJECT-DESIGNATOR ((TYPE KNIFE) (FOR-GUEST BOB)) {1006F55AC3}>)
```

Der Wert `{1006F55AC3}` stellt lediglich die interne Speicher-Adresse des Designators dar und ist auf Grund dessen nicht weiter zu beachten.

Die letzte Testfunktion `test-objectlocation-from-object` erstellt ebenfalls ein Objekt, für welches dann lediglich die Funktion `get-objectlocation-from-object` aufgerufen wird. So konnte überprüft werden, ob die Zuweisung von Objektpositionen zu Objekten korrekt funktioniert.

## 4.3 Beschreibung des Plans

Innerhalb dieses Kapitels wird nun, mit Hilfe der `Behaviour Trees`, der eigentliche Plan beschrieben und erklärt. Dabei wird der Plan zunächst in einzelnen Teilen erklärt, um die Übersicht zu wahren. Nachdem diese Teile beschrieben wurden, wird abschließend der Zusammenhang zwischen diesen Teilen erklärt.

Die einzelnen Teile des Plans sind dabei das `Greifen und Abstellen`, die Erkennung eines Objektes auf dem `kitchen_sink_block`, sowie das Erkennen eines Objektes auf dem `kitchen_island`.
Diese Teilaspekte des Plans werden seperat beschrieben, da es immer wiederkehrende Aktionen sind, welche an verschiedenen Stellen des Plans auftreten und somit nur einmal zu Beginn beschrieben werden müssen.

### 4.3.1 Greifen und Abstellen

Zu Beginn wird das Greifen und Abstellen innerhalb des Planes beschrieben. Hierfür werden vier Aktionen durchgeführt, welche nacheinander ausgeführt werden. Zunächst wird das Objekt, welches von der Perzeption erkannt wurde und gegriffen werden soll, an das eigentliche Objekte hinzugefügt. Dies geschieht mittels der Funktion `equate`, welche von `CRAM` zur Verfügung gestellt wird. Diese Aktion ist notwendig, da an dem von der Perzeption erkannten Objekt genaue Informationen über die Position dieses Objektes gespeichert sind.
Anschließend wird die Funktion `pick` (siehe Kapitel 3.1.1.3 auf Seite 11) aufgerufen, um das Objekt zu greifen. Um das Objekt nun abzustellen, wird `place` (siehe Kapitel 3.1.1.4 auf Seite 11) aufgerufen. Als Position, an welcher das Objekt abgestellt werden soll, wird `bring-to` (siehe Kapitel 4.2.1 auf Seite 22) übergeben. Abschließend wird der Boolean `placed-object` auf `t` gesetzt, damit der Roboter weiß, dass er das Objekt bereits abgestellt hat.
Da alle Aktionen, welche ausgeführt werden, auf die vorherige Aktion aufbauen und ohne diese nicht ausgeführt werden können, kann dieser Teil auch als `Sequence` in einem Behaviour Tree dargestellt werden. Abbildung 4.1 zeigt eine solche `Sequence` für das Greifen und Abstellen eines Objektes.

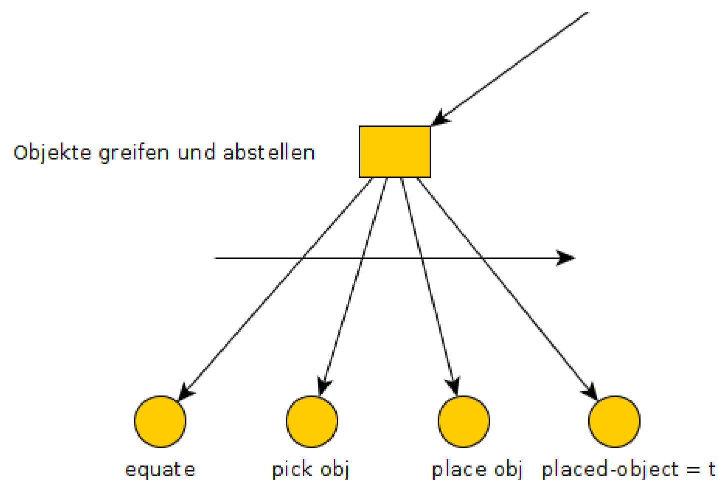


***Abbildung 4.1:*** *Ablauf einer Pick und Place Aktion als Sequence aus einem Behaviour Tree*

### 4.3.2 Erkennen auf dem kitchen_sink_block

Innerhalb dieser Bachelorarbeit gibt es zwei mögliche Positionen, an denen sich Objekte befinden können. Eine dieser Positionen ist der `kitchen_sink_block`, auf welchem die Objekte auch im Normalfall zu finden sind.
Um ein Objekt auf dem `kitchen_sink_block` zu finden, wird die Funktion `search-object-at-object-locations` (siehe Kapitel 4.2.3 auf Seite 23) genutzt. Innerhalb des Planes wird nun geprüft, ob die Rückgabe der Funktion ein Objekt oder nil ist. Entsprechend der Rückgabe wird nun entweder das Objekt gegriffen oder andernfalls abgebrochen oder versucht, das Objekt erneut zu erkennen.
Da bei der Erkennung nur eine der zwei Möglichkeiten eintreffen kann, kann dieser Teil auch als `Selector` in einem Behaviour Tree dargestellt werden, wie in Abbildung 4.2 zu sehen ist.

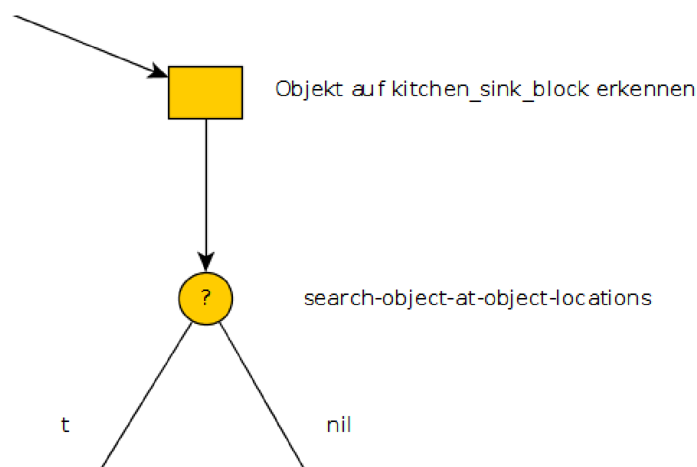


***Abbildung 4.2:*** *Ablauf für die Erkennung eines Objektes auf dem kitchen_sink_block als Selector aus einem Behaviour Tree*

### 4.3.3 Erkennen auf dem kitchen_island

Die zweite mögliche Position, auf welcher Objekte zu finden sind, ist das `kitchen_island`. Wenn der Roboter sich dazu entscheidet, auf diesem Tisch nach einem Objekt zu suchen, startet er einen Ablauf, welcher sich auch als `Sequence` darstellen lässt (siehe Abbildung 4.3 auf der nächsten Seite).
Die erste Aktion des Roboters ist es, zum `kitchen_island` zu fahren, welchen er mit der Funktion `go-in-front-of-island` (siehe Kapitel 4.2.9 auf Seite 24) erreicht. Nun kann der Roboter die Objekte, welche auf dem `kitchen_island` stehen, mit der Funktion `perceive-table` (siehe Kapitel 4.2.12 auf Seite 25) erkennen und verarbeiten.
Da innerhalb eines kompletten Durchlaufes lediglich einmal geprüft werden soll, welche Objekte auf dem `kitchen_island` stehen, wird der Boolean `checked-table` nun auf `t` gesetzt. Abschließend prüft der Roboter nun, ob das gesuchte Objekt innerhalb der Liste zu finden ist, welche die erkannten Objekte vom `kitchen_island` enthält. Abhängig von dem Ergebnis

wird nun entweder das Objekt gegriffen und an der richtigen Position abgestellt oder es wird geprüft, ob sich das Objekt an einem anderen Ort befinden könnte.

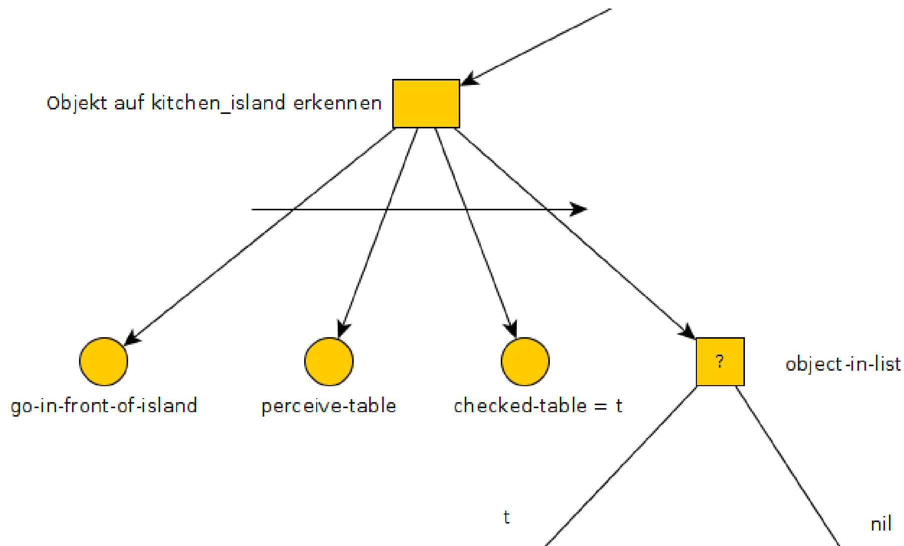


***Abbildung 4.3:*** *Ablauf für die Erkennung eines Objektes auf dem kitchen_island als Selector aus einem Behaviour Tree*

### 4.3.4 Beschreibung des Gesamtplanes

Nachdem nun die drei grundlegenden Abläufe des Planes beschrieben wurden, wird nun das Zusammenspiel dieser Abläufe beschrieben. Dabei wird zunächst beschrieben, wie sich der Plan verhält, wenn sich der Roboter näher am `kitchen_island` befindet. Anschließend wird der Ablauf beschrieben, wenn sich der Roboter näher am `kichen_sink_block` befindet.
Zu Beginn des Planes lokalisiert sich der Roboter und prüft, an welchem Tisch er näher steht. Wenn das `kichen_island` näher ist, wird zunächst überprüft, ob die Objekte auf dem Tisch bereits bekannt sind. Dies erfährt der Roboter über den Boolean `checked-table`. Wenn die Objekte noch nicht bekannt sind, möchte der Roboter prüfen, ob und welche Objekte auf dem Tisch stehen. Hierzu nutzt er den Ablauf, welcher in 4.3.3 auf der vorherigen Seite beschrieben wurde. Er fährt also zum Tisch und erkennt dort die gegebenenfalls stehenden Objekte. Sollte der Boolean `checked-table` angeben, dass der Tisch bereits einmal verarbeitet wurde, wird dieser Schritt übersprungen.
In beiden Fällen wird nun mit derselben Aktion fortgefahren. Es wird nämlich geprüft, ob sich das gesuchte Objekt auf dem Tisch befindet. Dies wird mit Hilfe der Funktion `object-in-list` durchgeführt (siehe Kapitel 4.2.13 auf Seite 25). Befindet sich das Objekt nun in der Liste, so versucht der Roboter das Objekt zu greifen und abzustellen. Der Ablauf hierzu wurde bereits in 4.3.1 auf Seite 27 beschrieben. Außerdem wurde der Ablauf des Erkennens von Objekten auf dem `kitchen_island`, sowie dem Greifen und Abstellen zur Veranschaulichung ebenfalls in der Abbildung 4.4 auf der nächsten Seite dargestellt.
Wenn sich das Objekt jedoch nicht auf dem `kitchen_island` befindet, prüft der Roboter, an welchen Positionen das Objekt stehen könnte. Hierzu wird die Methode `search-object-`

at-objects-location genutzt (siehe Kapitel 4.2.3 auf Seite 23). Da innerhalb dieser Bachelorarbeit lediglich der kitchen_sink_block als Position in Frage kommt, fährt der Roboter nun dorthin und sucht das Objekt auf diesem Tisch (siehe Kapitel 4.3.2 auf Seite 28). Wenn er das Objekt findet, versucht er es zu greifen und auf dem kitchen_island abzustellen (siehe 4.3.1 auf Seite 27). Wenn das Objekt allerdings nicht an dieser Position zu finden ist, hat der Roboter alle Positionen abgesucht, an denen sich ein Objekt innerhalb dieser Bachelorarbeit befinden kann. Der Plan schließt für dieses Objekt in diesem Fall mit einem object-not-found ab und es wird mit dem nächsten Objekt fortgefahren, sofern weitere Objekte benötigt werden.

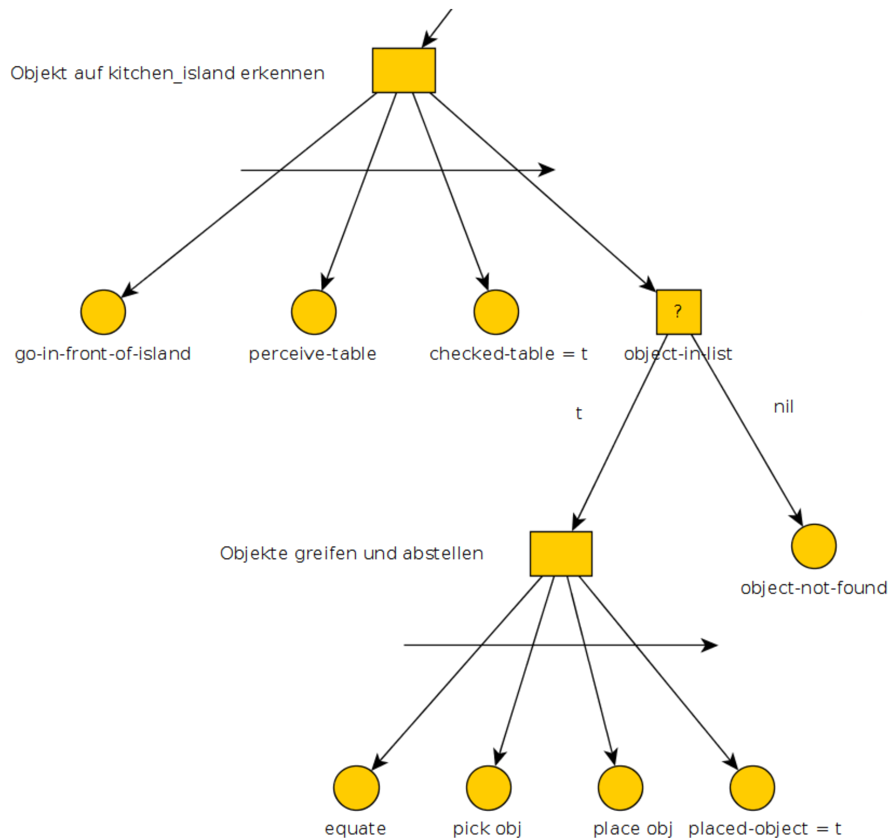


***Abbildung 4.4:*** *Komplettablauf für das Erkennen, Greifen und Abstellen von Objekten auf dem* kitchen_island

Nachdem nun das Verhalten des Planes beschrieben wurde, wenn das kitchen_island näher am Roboter ist, wird nun der zweite Fall betrachtet. In diesem Fall wäre der Roboter näher zum kitchen_sink_block positioniert. Anschließend beginnt der Roboter damit, das gesuchte Objekt auf dem kitchen_sink_block zu finden. Diese Suche wird ebenfalls mit der Funktion search-object-at-objects-location durchgeführt. Findet der Roboter das Objekt direkt, kann er damit beginnen, das Objekt zu greifen und an der richtigen Position auf dem kitchen_island abzustellen. Andernfalls prüft der Roboter, ob er bereits weiß, welche Objekte auf dem kitchen_island stehen.

Der nun folgende Ablauf ist ähnlich zu dem, wenn der Roboter am Anfang näher zum `kitchen_island` steht. Er fährt also zunächst, sofern er den Tisch noch nicht erfasst hat, zum `kitchen_island`, um dort die Objekte zu erfassen. Sollte der Tisch bereits vorher einmal angefahren worden sein, um dort die Objekte zu erkennen, wird dieser Schritt übersprungen und mit dem nächsten fortgefahren.
In diesem prüft der Roboter, ob das Objekt auf dem Tisch steht. Wenn dem so ist, greift er das Objekt und stellt es an der korrekten Position ab. Andernfalls hat er die beiden möglichen Positionen geprüft und schließt den Plan für dieses Objekt ab, da er es nicht gefunden hat.

Abschließend kann festgehalten werden, dass der Plan bereits auf einige, nicht vorhergesehene Umstände reagieren kann und diese verarbeitet.
So kann damit umgegangen werden, dass ein Objekt nicht an der für das Objekt, vorgesehenen Position steht. In diesem Fall wird eigenständig geprüft, ob das benötigte Objekte bereits auf dem zu deckenden Tisch steht.
Des Weiteren kann von diesem Plan jedes beliebige Objekt verarbeitet werden, sofern es von der Perzeption die benötigten Objektinformationen erhält und die Manipulation dazu in der Lage ist, dass Objekt zu greifen. Es wäre also prinzipiell, mit kleinen Veränderungen, möglich, diesen Plan nicht für das Tischdecken zu nutzen, sondern auch, um beispielsweise einen Tisch abzudecken.

## 4.4 Robot Operating System (ROS)

Damit der Plan innerhalb dieser Bachelorarbeit auf einem echten Robotersystem getestet werden kann, wird das `Robot Operating System`, kurz `ROS`, genutzt. Das Robot Operating System ist ein Framework für Roboter, welches auf Linux Distributionen aufbaut. Es ist allerdings mittlerweile auch möglich, `ROS` auf anderen Systemen, wie zum Beispiel Windows, zu nutzen.
Mit Hilfe des `Robot Operating Systems` ist es möglicht, Software für Roboter zu implementieren und diese zu testen. Dabei kann auf verschiedene Tools und Bibliotheken zurück gegriffen werden.
Die Bibliotheken und Tools können dabei in verschiedenen Programmiersprachen geschrieben werden, beispielsweise `c++, java, lisp` oder `python`. Somit kann je nach Anforderung, zum Beispiel wenn die Perfomance wichtig ist, die bestmögliche Sprache gewählt werden.
Ein großer Vorteil von `ROS` ist die große Community, da mit Hilfe dieser viele Probleme schnell gelöst werden können. Außerdem gibt es somit auch eine große Anzahl von verschiedenen Tools und Bibliotheken, aus denen man diejenige wählen kann, welche den persönlichen Anforderungen optimal entsprechen.
Neben den Tools und Bibliotheken gibt es auch Anwendungen, beispielsweise um Tests zu simulieren, welche die Daten verarbeiten.
Eine Bibliothek, welche innerhalb dieser Bachelorarbeit genutzt wurde, ist `MoveIt!` [2]. Mit Hilfe von `MoveIt!` ist es möglich, Berechnungen für die Bewegung des Roboters durchzuführen.

[2] MoveIt!: `http://moveit.ros.org/`

Eine weitere Bibliothek, welche von ROS zur Verfügung gestellt wird, ist `tf` [3]. Mit `tf` ist es möglich, sogenannte `Frames` zu erstellen und Umrechnungen zwischen diesen durchzuführen. `Frames` geben in `tf` die Position von Objekten oder Punkten an, wobei ein Frame in Relation zu einem Weiteren stehen muss, damit eine Position in einen anderen Frame umgerechnet werden kann. Zur Visualisierung solcher Berechnungen gibt es das Tool `rviz` [4]. In diesem kann man über so genannte `Marker` die Positionen, welche berechnet wurden, visualisieren und die Bewegungen besser nachvollziehen. Weiterhin stellt `rviz` die Möglichkeit zur Verfügung, alle derzeit vorhandenen Frames, inklusive deren Relationen, darzustellen.

[3]tf: http://wiki.ros.org/tf
[4]rviz: http://wiki.ros.org/rviz

# Experimente

Um die Funktionalität der Implementierung zu prüfen, wurden im Rahmen dieser Bachelorarbeit zwei verschiedene Szenarien betrachtet, welche im Folgenden beschrieben werden.
Im ersten Szenario soll der Roboter ein Objekt vom `kitchen_sink_block` auf den `kitchen_island` aufdecken, wobei er perfekte Bedingungen vorfindet. Das Objekt befindet sich somit am vordefinierten Platz und der zu deckende Tisch ist leer.
Das zweite Szenario betrachtet die Generalität der Implementierung, indem innerhalb dieses Experimentes das benötigte Objekt bereits auf dem `kitchen_island` steht.
Im Folgenden werden diese beiden Szenarien beschrieben und die Ergebnisse vorgestellt.

## 5.1 Tischdeckszenario mit leerem Tisch

In diesem Kapitel wird das erste Experiment beschrieben. Dabei wird zunächst der Aufbau sowie das Ziel, welches erreicht werden soll, beschrieben. Im Anschluss wird das Ergebnis dargestellt, welches auf dem Roboter erreicht wurde. Das erhaltene Ergebnis wird zum Abschluss mit dem erwarteten Ergebnis verglichen.

Das Ziel dieses Experimentes ist es, dass der Roboter autonom ein Objekt sucht und es, nachdem er es gefunden hat, an den korrekten Platz auf dem `kitchen_island` abstellt. Es ist dabei ausreichend, mit einem Objekt zu testen, da der Ablauf für mehrere Objekte lediglich wiederholt wird.
Wenn also für ein Objekt ein korrekter Ablauf festgestellt wird, kann davon ausgegangen werden, dass auch bei mehreren und anderen Objekten der Ablauf korrekt sein wird.
Für dieses Experiment wird die Szene `set-scene-thomasthesis-experiment-01` (siehe Kapitel 4.1.2 auf Seite 21) genutzt. Es soll also eine Milch aufgedeckt werden. Die Milch steht dabei, wie vom Roboter erwartet, auf dem `kitchen_sink_block`. Außerdem steht der Roboter näher am `kitchen_island` als am `kitchen_sink_block`.
Es wird nun der folgende Ablauf erwartet, welcher auch in der Abbildung 5.1 auf der nächsten Seite dargestellt ist.

TODO: Jan nach dem Beispiel fragen und dann anpassen

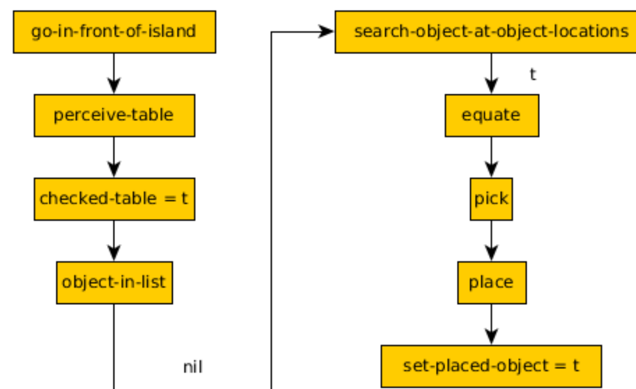


***Abbildung 5.1:*** *Ablaufdiagramm für das erste Experimente*

Der Roboter berechnet, dass es kürzer ist, zunächst zum `kitchen_island` zu fahren und zu schauen, ob sich die gesuchte Milch dort befindet. Da sich die Milch jedoch nicht dort befindet, prüft der Roboter nun die Positionen, an denen sich die Milch normalerweise befindet. Neben dem Kühlschrank, welcher jedoch übersprungen wird (siehe Kapitel 4.2.3 auf Seite 23), wird der `kitchen_sink_block` angegeben.
Der Roboter soll also nun zum `kitchen_sink_block` fahren und dort prüfen, ob er die Milch findet. Da die Milch auf dem `kichen_sink_block` steht, sollte der Roboter im Anschluss die Milch greifen, wenn er sie korrekt erkannt hat. Anschließend soll der Roboter mit der Milch zurück zum `kitchen_island` fahren, da sich dort der zu deckende Tisch befindet. Wenn der Roboter die Position erreicht hat, soll er die Milch zentral abstellen. Mit dieser Aktion hat der Roboter das Experiment abgeschlossen.

### 5.1.1 Ergebnis

Der Ablauf dieses Experimentes lief wie erwartet. Der Roboter fuhr zuerst zum `kitchen_island`, um dort nach der Milch zu suchen. Nachdem er die Milch dort nicht gefunden hatte, fuhr er zum `kitchen_sink_block`. Hier wurde die Milch nun mittels der Funktion `search-object-at-object-locations` gefunden.
Anschließend wurde die Milch korrekt gegriffen und der Roboter fuhr zurück zum `kitchen_island`, um die Milch an der korrekten Position abzustellen. Wie Abbildung 5.2 auf der nächsten Seite zeigt, wurde die Milch korrekt abgestellt.

***Abbildung 5.2:*** *Die Milch wurde korrekt auf dem kitchen_island abgestellt.*

## 5.2 Tischdeckszenario mit dem gesuchten Objekt auf dem Tisch

Innerhalb dieses Kapitels wird nun das zweite Experiment beschrieben, welches innerhalb dieser Bachelorarbeit durchgeführt wurde. Es wird, wie im Experiment zuvor, zunächst der Aufbau und erwartete Ablauf beschrieben. Anschließend wird das Ergebnis des Experimentes dargstellt und mit den Erwartungen verglichen.

Wie im vorherigen Experiment wird auch innerhalb dieses Experimentes mit lediglich einem Objekt gearbeitet. Hierzu wird erneut die Szene `set-scene-thomasthesis-experiment-01` (siehe Kapitel 4.1.2 auf Seite 21) genutzt, bei welcher eine Milch aufgedeckt werden soll. Der Unterschied zum vorherigen Experiment ist jedoch, dass sich die Milch nicht wie erwartet auf dem `kitchen_sink_block` befindet, sondern bereits auf dem `kitchen_island` steht. Außerdem befindet sich der Roboter dieses Mal näher am `kitchen_sink_block`.
Anhand dieses Aufbaus wird nun der folgende Ablauf erwartet, welcher ebenfalls in einem Ablaufdiagramm (siehe Abbildung 5.3 auf der nächsten Seite) dargestellt ist.

TODO: Jan nach dem Beispiel fragen und dann anpassen

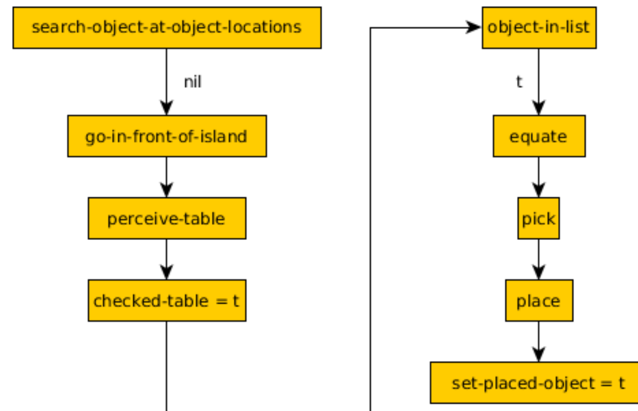


***Abbildung 5.3:*** *Ablaufdiagramm für das zweite Experimente*

Der Roboter fährt zunächst zum `kitchen_sink_block`, da dieser näher an seiner Position ist als das `kitchen_island`. Da dies auch die Position ist, an welcher der Roboter erwartet, dass er die Milch findet, sucht er sie nun. Da die Milch allerdings nicht auf dem `kitchen_sink_block` steht, kann der Roboter sie hier auch nicht finden.
Nun prüft der Roboter, ob er bereits einmal geprüft hat, welche Objekte auf dem `kitchen_island` stehen. Da dies nicht der Fall ist, fährt er nun zum `kitchen_island` und sucht hier nach der Milch. Der Roboter findet die Milch und greift sie nun, um sie an der richtigen Position auf dem `kitchen_island` abzustellen. Mit dieser Aktion hat der Roboter das Experiment abgeschlossen.

### 5.2.1 Ergebnis

Wie bereits im ersten Experiment, verlief auch das zweite Experiment wie erwartet. Der Roboter fuhr zunächst zum `kitchen_sink_block` und suchte die Milch an dieser Position, allerdings fand er sie nicht. Anschließend fuhr der Roboter zum `kitchen_island`, um zu prüfen, ob die Milch bereits auf dem Tisch steht.
Nachdem er die Milch dort gefunden hatte, wurde diese angehoben um sie an die korrekte Position abzustellen. Diese ähnelt der Position, welche bereits in der Abbildung 5.2 auf der vorherigen Seite aus dem ersten Experiment zu sehen ist.

Nachdem nun beide Experimente inklusive deren Ergebnisse vorgestellt wurden, wird nun ein kurzes Fazit zu den Experimenten dargelegt. Es konnte innerhalb der Experimente gezeigt werden, dass das Verhalten des Plans korrekt ist. Außerdem wurde auch gezeigt, dass der Roboter durch den Plan auf nicht ideale Umstände reagieren kann, zum Beispiel wenn das gesuchte Objekt nicht an den definierten Positionen zu finden ist.
Zudem konnte ebenfalls dargelegt werden, dass ein korrekter Ablauf für das Greifen und Abstellen vorhanden ist. Ebenfalls wurde gezeigt, dass der Roboter das Objekt an der dafür vorgesehenen Position auf dem `kitchen_island` abstellt.

*Kapitel 6*

# Fazit

Innerhalb dieser Bachelorarbeit wurde das Ziel verfolgt, dass ein Roboter autonom einen Tisch decken kann. Dieses Ziel sollte dabei mittels der Plansprache `CRAM` erreicht werden. Zu Beginn dieser Arbeit wurde in der Einführung (siehe Kapitel 1 auf Seite 3) das genaue Ziel dieser Bachelorarbeit wie folgt definiert:

> Ziel dieser Bachelorarbeit ist es, dass der Roboter autonom nach einem Objekt sucht und dieses Objekt anschließend korrekt auf dem zu deckenden Tisch abstellt.

Um dieses Ziel zu erreichen, wurden zunächst die bestehenden Grundlagen analysiert und anschließend erweitert. Dabei wurden zu Beginn eine neue Objektposition eingeführt, welche den Ort beschreibt, an welcher ein Objekt in der Regel zu finden ist. Aufbauend darauf konnte nun eine Funktion implementiert werden, welche die Suche für ein Objekt ausführt (siehe Kapitel 4.2.3 auf Seite 23).
Da diese Funktion allerdings nur an den vordefinierten Objektpositionen sucht, wurde innerhalb des Planes ein Mechanismus implementiert. Dieser prüft maximal einmal in einem kompletten Planablauf, ob und welche Objekte sich auf dem zu deckenden Tisch befinden. Somit kann der Roboter alle Positionen, welche innerhalb dieser Bachelorarbeit verfügbar sind, absuchen und prüfen.
Außerdem wurde implementiert, dass der Roboter zu Beginn eines Planablaufes ermittelt, zu welchem der beiden Tische (`kitchen_island` und `kitchen_sink_block`) er näher steht. Anhand dieser Information kann der Roboter entscheiden, ob es sinnvoller ist zuerst zum `kitchen_sink_block` zu fahren oder zuvor zu prüfen, ob und welche Objekte auf dem `kitchen_island` stehen.
Anschließend wurden zwei Experimente durchgeführt, welche den Fokus auf die Suche eines Objektes sowie das Abstellen eines Objektes legen (siehe Kapitel 5 auf Seite 33). Dabei wurde nur mit einem Objekt getestet, da dies ausreichend ist, um die Funktionalität der Implementierungen zu zeigen. Da die Experimente erfolgreich abgeschlossen werden konnten, kann festgehalten werden, dass das gesetzte Ziel innerhalb dieser Bachelorarbeit erreicht wurde.
Der Roboter kann, mit Hilfe der implementierten Funktionen und des Planes, autonom ein Objekt suchen, es greifen und korrekt auf einem vorgegebenen Tisch abstellen.

*Kapitel 7*

# Ausblick

Nachdem der Inhalt und die Ergebnisse dieser Bachelorarbeit dargelegt wurden, wird nun kurz auf einige mögliche, ausgewählte Ansätze eingegangen, welche auf diese Arbeit aufbauen könnten. Ein erster Ansatz, um die Autonomie des Roboters zu erhöhen, wäre die Berechnung von benötigten Händen, damit ein Objekt sicher gegriffen werden kann. Derzeit wird die Anzahl der Hände über ein Prädikat fest einem Objekt zugewiesen, allerdings ist es erstrebenswert, dass der Roboter dies abhängig vom Objekt selbst berechnen kann, damit auch unbekannte Objekte sicher gegriffen werden können. Für diese Berechnungen könnten Informationen der Perzeption, beispielsweise die Größe sowie das Volumen des Objektes, genutzt werden.

Neben der Berechnung der benötigten Hände könnte auch das Abstellverfahren innerhalb des Planes verbessert werden. Der Roboter könnte über die benötigten Objekte zunächst berechnen, welche Objekte wo auf dem gedeckten Tisch stehen sollen und entsprechend dieser Positionen den Tisch decken. So könnte er zunächst die Objekte auf dem Tisch abstellen, welche weiter hinten stehen sollen, um anschließend die Objekte abzustellen, die weiter vorne stehen. Dies würde Probleme vermeiden, dass der Roboter ein Objekt abstellt und danach ein weiteres Objekt hinter diesem Objekt abstellen möchte, jedoch keine Bewegung berechnen kann, um das Objekt an dieser Position abzustellen.

Daran anknüpfend könnte dem Roboter beigebracht werden, nicht benötigte Objekte vom Tisch zu räumen. Dies könnte zusammenhängend mit dem erfassen aller Objekte auf dem Tisch (siehe Kapitel 4.2.12 auf Seite 25) geschehen, indem der Roboter überprüft, ob die Objekte auf dem Tisch alle für die gegebene Szene benötigt werden. Nachfolgend könnte er alle nicht benötigten Objekte wegräumen.

Zusammenhängend damit könnte auch die allgemeine Generalität des Planes erweitert werden, indem der Roboter neben dem Decken eines Tisches eben diesen auch abräumen könnte. Weiterhin könnte auch die Informationsausgabe des Roboters verbessert werden, da derzeit nicht bekannt wird, wenn ein Objekt vom Roboter nicht gefunden wird. Es könnte zum Beispiel implementiert werden, dass nach der Ausführung des Planes eine Liste zurückgegeben wird, in welcher alle Objekte enthalten sind, welche nicht gefunden wurden und somit auch nicht auf den Tisch gestellt werden konnten.

# Literaturverzeichnis


[1] Dr. Dirk Jacob. Roboter in der automobilindustrie.

[2] John M. Evans. Helpmate: An autonomous mobile robot courier for hospitals. In *Intelligent Robots and Systems '94. 'Advanced Robotic Systems and the Real World', IROS '94. Proceedings of the IEEE/RSJ/GI International Conference on (Volume:3 )*. IEEE, 1994.

[3] Paolo Fiorini and Erwin Prassler. Cleaning and household robots: A technology survey. *Autonomous robots*, 9(3):227–235, 2000.

[4] Michael Beetz, Lorenz Mösenlechner, Moritz Tenorth, and Thomas Rühr. Cram – a cognitive robot abstract machine. In *5th International Conference on Cognitive Systems (CogSys 2012)*, 2012.

[5] Moritz Tenorth and Michael Beetz. KnowRob—knowledge processing for autonomous personal robots. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 4261–4266. IEEE, 2009.

[6] M. Tenorth and M. Beetz. KnowRob: A knowledge processing infrastructure for cognition-enabled robots. *The International Journal of Robotics Research*, 32(5):566–590, April 2013.

[7] Dave Mount. Artificial intelligence for games: Decision making. 2013.