
Inbetriebnahme des OpenMANIPULATOR-X und Handlungsplanung mit Partial Order Planning Forward

Fabian Claus

20130004



Bachelorarbeit

Fachbereich Informatik
und Medien
Technische Hochschule Brandenburg

Betreuer: Prof. Dr. Jochen Heinsohn
2. Betreuer: Dipl. Inform. Ingo Boersch

Brandenburg, den TT.MM.JJJJ
Bearbeitungszeit: TT.MM.JJJJ - TT.MM.JJJJ

Inhaltsverzeichnis

Zusammenfassung	III
Abstract	IV
Abbildungsverzeichnis	V
Tabellenverzeichnis	VI
Abkürzungsverzeichnis	VII
1 TODOS	1
2 Einleitung	2
2.1 Aufgabe	2
2.2 Abgrenzung/Motivation	2
3 Grundlagen	3
3.1 ROS2	3
3.1.1 Node	3
3.2 ROS2 Interfaces	3
3.2.1 Message	3
3.2.2 Service	4
3.2.3 Action	5
3.3 ROS2 Node Kommunikation	5
3.3.1 Topic	5
3.3.2 Service und Client	5
3.3.3 Action Server und Client	5
3.4 OpenMANIPULATOR-X	6
3.5 Planung	6
3.5.1 Stanford Research Institute Problem Solver	6
3.5.2 Planning Domain Definition Language	7
3.5.3 PDDL-Plugin für VS Code	9
3.5.4 Partial Order Planning	10
3.5.5 Forward Chaining Partial Order Planning	12
3.5.6 Partial Order Planning Forward	13
3.5.7 Behavior Tree	15
3.6 Planung in Robot Operating System 2 (ROS2)	17
4 Konzept	19
4.1 Inbetriebnahme	19
4.2 Struktur Nodes	19
4.3 Planungsmodell	19

4.4	Erstellung von Übungsbeispielen	20
5	Implementierung	21
5.1	Inbetriebnahme Greifarm	21
5.1.1	Zusammenbau	21
5.1.2	Virtuelle Maschine	22
5.2	Steuerung OMX	22
5.2.1	Anschluss über U2D2	22
5.2.2	OMX-Controller	22
5.2.3	Joint und Task Space	22
5.2.4	Topics	24
5.2.5	Kinematik	24
5.2.6	Teleop	24
5.3	PlanSys2	25
5.3.1	PDDL-Domain	25
5.3.2	Action Nodes	27
5.3.3	Control Gripper Aktion	27
5.3.4	Move Gripper Aktion	28
5.3.5	Package	29
5.4	Praktische Anwendung	29
5.4.1	PlanSys2 Terminal	29
5.4.2	Sussman Anomalie	30
5.4.3	Mehr Blöcke, mehr Positionen	30
5.4.4	Abstürze und Fehlersuche	32
6	Ergebnis	35
6.1	Zusammenfassung	35
6.2	Ausblick	35
6.2.1	MoveIt	35
6.2.2	Vorschläge für Übungen in der Lehre	35
Literaturverzeichnis		38
A	Quellcode	39
A.1	Planungsdomäne	39
A.2	Gripper Package Launch Datei	42
A.3	Move Gripper Action Node	44

Zusammenfassung

Zusammenfassung

Eine Kurzzusammenfassung der Vorgehensweise und der wesentlichen Ergebnisse.

Allgemeine Merkmale

- Objektivität: Es soll sich jeder persönlichen Wertung enthalten.
- Kürze: Es soll so kurz wie möglich sein.
- Verständlichkeit: Es weist eine klare, nachvollziehbare Sprache und Struktur auf.
- Vollständigkeit: Alle wesentlichen Sachverhalte sollen enthalten sein.
- Genauigkeit: Es soll genau die Inhalte und die Meinung der Originalarbeit wiedergeben.

Abstract

Abstract

Obige Zusammenfassung in englischer Sprache.

Abbildungsverzeichnis

1	Konsolenausgabe nach Ausführung des Planers über das PDDL-Plugin	10
2	Visualisierung eines Plans mit dem PDDL-Plugin mit Fokus auf die Aktionen (links) und Fokus auf die relevanten Objekte (rechts)	11
3	Relaxed Planning Graph (RPG) für das Ziel $\{at(Home), has(Product)\}$	14
4	Enforced Hill-Climbing (EHC) Algorithmus [10]	15
5	Behavior Tree (BT) zur Ausführung einer spezifischen Aktion [12]	18
6	Bausatz für den OpenMANIPULATOR-X	21
7	U2D2 Power Hub Board mit montiertem U2D2 (rechts im Bild)	23
8	Geräte-Menü der Virtuelle Maschine (VM) mit angeschlossenem U2D2	23
9	Fernsteuerung des OMX über die Tastatur	25
10	Init (l.) und Home (r.) Pose der Tastatur Teleop	26
11	Startzustand der Sussman Anomalie	30
12	Lösung des Teilziels A auf B	30
13	Lösung des Teilziels B auf C	30
14	Durch Partial Order Planning Forward (POPF) generierter Plan für die Sussman Anomalie	31
15	Visualisierte Lösung der Sussman Anomalie durch POPF	31
16	ROS2 Planning System (PlanSys2) Terminal mit Abfrage der Typen des geladenen Models	33
17	Ausgabe eines Plans in PlanSys2 Terminal	34

Tabellenverzeichnis

1	Integrierte Datentypen für Interfaces und deren C++ Equivalent	4
2	Array Typen und deren C++ Equivalent	4
3	Übersicht für BT Knoten, nach [5]	17

Abkürzungsverzeichnis

Abkürzungsverzeichnis

BT Behavior Tree

EHC Enforced Hill-Climbing

IDL Interface Definition Language

PlanSys2 ROS2 Planning System

PDDL Planning Domain Definition Language

POP Partial Order Planning

POPF Partial Order Planning Forward

RCL ROS Client Library

ROS2 Robot Operating System 2

RPG Relaxed Planning Graph

STRIPS Stanford Research Institute Problem Solver

VM Virtuelle Maschine

1 TODOS

- Neues Bild für Keyboard Teleop 5.2.6
- Bilder für Teleop Posen
- Bild für Servo Abdeckung vorgestanzt
- Überprüfen ob all Anhänge aktuell sind
- Entscheidung Bezeichnung für OMX, Abk., Greifarm, OpenManipulator-X, OpenMANIPULATOR X
- Fehlersuche Abstürze PlanSys2
- Zusammenfassen der Gripper Nodes zu einer parametrisierten Node
- baudrate änderungen für omx controller dokumentieren
- code kommentieren
- launch file für eigenes package beschreiben
- Move node umschreiben um pair anstelle mehrer maps zu nutzen
- PKG readme anpassen: start befehle sowie problem
- domänen dateien aufräumen, umbenennen
- Bilder für PlanSys2 Terminal
- section struktur für ROS2 überdenken
- aspect ratio für Bilder fixen
- Erklärung von durative actions von Implementierung zu Grundlagen verschieben
- Angabe Author für simmons_2001
- einheitlich Knoten oder Nodes, Leaf / Blatt
- Eltern / Kind Knoten oder Vorgänger Nachfolger
- Quellenplatzierung (bei Aufzählungen)
- quellenangabe tabellen
- grafiken für bt
- tabelle mit bt symbolen

2 Einleitung

Dieser Teil der Arbeit sollte folgende Inhalte haben:

- Einführung in die Problemstellung
- Motivation und Herleitung des Themas
- Aufbau der Arbeit

Hinweis: Es hat sich als hilfreich erwiesen, die Einleitung mit der Zusammenfassung bzw. dem Abstract und der Schlussfolgerung zu vergleichen. Damit stellt man sicher, dass diese inhaltlich im Bezug auf Zielsetzung und Motivation übereinstimmen. Der Umfang sollte ca. 5 % der gesamten Arbeit betragen.

Roboter sind ein wichtiger Bestandteil der Forschung und der Lehre. Auch im Fachbereich Informatik und Medien der THB! (THB!) sind Roboter Teil und Kern mehrerer Module. Aktuell werden hier das an der THB! entwickelte AKSEN-Board¹ in Kombination mit LEGO sowie Pioneer 2/3 Roboter² genutzt. Zukünftig sollen auch der ROS2 basierte Roboter TurtleBot3 Waffle Pi³ mit dem Greifarm OMX genutzt werden.

In dieser Arbeit soll, speziell für den OMX, ein erster Überblick über die grundsätzlichen Vorgänge und Prozesse bei dessen Nutzung gegeben werden.

2.1 Aufgabe

Im Rahmen der Bachelorarbeit soll der Greifarm OpenMANIPULATOR-X der Firma Robotis in Betrieb genommen sowie die Möglichkeiten der Steuerung erprobt werden. Weiterhin soll die Steuerung mittels Handlungsplanung ermöglicht werden. Hierfür sind bestehende Bibliotheken und Frameworks zu evaluieren und ein ausgewähltes zu implementieren und an einem praktischen Beispiel zu demonstrieren. Für das gewählte Framework oder den gewählten Planer sollen verschiedene Szenarien getestet werden, die die Möglichkeiten und Grenzen zeigen. Es soll ein ROS2 Package erstellt werden, mithilfe dessen der Einstieg in die Nutzung des OMX vereinfacht werden soll. Abschließend sollen Möglichkeiten der Einbindung in den Lehrbetrieb gezeigt werden.

2.2 Abgrenzung/Motivation

¹[http://ots.th-brandenburg.de/aksen-controller-für-reaktive-roboter.html](http://ots.th-brandenburg.de/aksen-controller-fur-reaktive-roboter.html)

²<http://ots.th-brandenburg.de/pioneer2-und-pioneer3roboter.html>

³<https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/>

3 Grundlagen

3.1 ROS2

ROS2 ist eine Sammlung von Bibliotheken und Werkzeugen für Robotik-Applikationen welche alle OpenSource sind. Die erste ROS2 Release-Version erschien im Dezember 2017 unter dem Namen Ardent Apalon **ros2docs**.

Es werden mehrere ROS Client Libraries (RCLs) zur Verfügung gestellt, welche den Zugriff auf die ROS2-API ermöglichen. Die RCLs für die Sprachen C++ und Python (rclcpp und rclpy) werden dabei direkt vom ROS2-Team verwaltet. Von der Community wurden weitere RCLs, unter anderem für die Sprachen C#, Swift und Rust, entwickelt. Um die Entwicklung der RCLs zu vereinfachen und die Logik sprachen unabhängig zu machen werden Funktionalitäten als C Interfaces zugänglich gemacht, für welche in den RCLs Wrapper geschrieben werden.

3.1.1 Node

Nodes sind die grundlegenden Komponenten, aus welchen eine ROS2 Anwendung besteht. Typischerweise besteht eine Anwendung aus mehreren Nodes. Entsprechend dem Prinzip Separation of Concerns (Trennung der Zuständigkeiten) sollen Nodes so implementiert werden, dass sie jeweils nur eine spezifische Aufgabe haben.

3.2 ROS2 Interfaces

In diesem Abschnitt werden die verschiedenen Arten von Interfaces beschrieben, welche für die im folgenden Abschnitt 3.3 erklärten Methoden zur Kommunikation zwischen mehreren Nodes genutzt werden.

ROS2 Interfaces sind Definitionen, welche Daten mit welchen Typen gesendet werden. Sie werden in der Interface Definition Language (IDL) geschrieben, welche es ermöglicht automatisch Source Code in verschiedenen Sprachen für diese zu generieren.

- Erklären was für den build nötig ist?

3.2.1 Message

Eine Message ist der einfachste Typ der ROS2 Interfaces, welcher gleichzeitig auch ein Baustein für die folgenden Interfaces sein wird.

Message Definitionen haben die Dateiendung `.msg` und werden per Konvention in einem Ordner `msg/` gespeichert. Jede `.msg` Datei besteht aus den folgenden Teilen:

- Felder
- Konstanten

Jedes Feld besteht aus einem Typ und einem durch ein Leerzeichen getrennten Namen: `typ name`, z.B. `bool isDone`. Als Typ können dabei die integrierten Typen (s. Tabelle 1) oder der Name anderer Messages genutzt werden. Zusätzlich kann jeder integrierte

3 Grundlagen

Typ als Array definiert werden (s. Tabelle 2).

Feldnamen müssen kleingeschrieben sein. Es können alphanumerische Zeichen sowie der Unterstrich zur Trennung von Wörtern genutzt werden. Das erste Zeichen muss ein Buchstabe sein und der Name darf nicht mit einem Unterstrich enden. Zusätzlich darf es keine aufeinander folgenden Unterstriche geben.

Konstanten behalten das Format der Felder bei. Der Name der Konstante wird komplett in Großbuchstaben geschrieben. Zusätzlich bekommen Konstanten einen Wert zugewiesen, welcher nicht innerhalb des Programms geändert werden kann. Die Zuweisung des Wertes erfolgt mit dem = Zeichen: `string EXAMPLE="test".`

Tabelle 1: Integrierte Datentypen für Interfaces und deren C++ Equivalent

Typ	C++
bool	bool
byte	uint8_t
char	char
float32	float
float64	double
int8	int8_t
uint8	uint8_t
int16	int16_t
uint16	uint16_t
int32	int32_t
uint32	uint32_t
int64	int64_t
uint64	uint64_t
string	std::string
wstring	std::u16string

Tabelle 2: Array Typen und deren C++ Equivalent

Typ	C++
static array	std::array<T, N>
unbounded dynamic array	std::vector
bounded dynamic array	custom_class<T, N>
bounded string	std::string

3.2.2 Service

Service Definitionen beschreiben eine Anfrage und eine Antwort. Die Definition hat die Dateiendung .srv und wird im Ordner `srv/` gespeichert. Anfrage und Antwort werden innerhalb der Datei durch – getrennt. Für beide Teile gilt, dass sie gültig sind,

3 Grundlagen

wenn sie einer gültigen Message Definition entsprechen. Ein Beispiel einer einfachen Servicedefinition ist in Listing 1 zu sehen.

```
1 int32 request_int
  string request_string
  ---
  float32 response_float
```

Listing 1: Beispiel einer Service Definition

3.2.3 Action

Action Definitionen bestehen aus den 3 Teilen Anfrage, Ergebnis und Feedback, in dieser Reihenfolge. Wie auch für Service Definitionen gilt, dass die Teile durch – getrennt sind und jeder einzelne Teil gültig ist, wenn er einer gültigen Message Definition entspricht.

3.3 ROS2 Node Kommunikation

Damit verschiedene Nodes untereinander kommunizieren können, gibt es verschiedene Mechanismen, welche sich primär darin unterscheiden, in welche Richtungen Nachrichten gesendet werden und ob es eine direkte Reaktion gibt. Für alle Mechanismen gilt, dass sie von einer Node unter einem bestimmten Namen sowie einem Typ zur Verfügung gestellt werden und von anderen Nodes durch eben diese genutzt werden können.

3.3.1 Topic

Topics entsprechen in etwa einem Newsletter System: eine Node veröffentlicht Daten, welche von allen anderen Nodes empfangen wird, die sich für diese registriert haben. Das Format der Daten entspricht einer gewählten Message Definition.

3.3.2 Service und Client

Services werden für Prozesse genutzt, in denen eine Anfrage gesendet und eine Antwort erwartet wird. Als Typ wird eine Servicedefinition genutzt.

3.3.3 Action Server und Client

Actions sind für länger andauernde Prozesse gedacht. Eine Node erstellt einen Action Server über welchen eine Action zur Verfügung gestellt wird. Andere Nodes können über einen Action Client eine Anfrage senden. Der Server bearbeitet die Anfrage und sendet bei Beendigung eine Antwort mit einem Ergebnis an den Client. Während der Dauer der Action kann der Server den Client optional mit Feedback Nachrichten über den aktuellen Status informieren.

Als Typ wird eine Action Definition genutzt.

3 Grundlagen

3.4 OpenMANIPULATOR-X

Der OpenMANIPULATOR-X ist ein von der Firma ROBOTIS⁴ nach den Prinzipien "OpenSoftware" und "OpenHardware" hergestellter Greifarm. OpenSoftware steht hierbei dafür, dass es ein OpenSource Projekt ist und auf dem OpenSource Projekt ROS2 basiert. OpenHardware steht dafür, dass die meisten Komponenten als STL-Dateien zur Verfügung stehen und als Ersatzteile oder zum Anpassen des Greifarms mittels eines 3D-Druckers selbst hergestellt werden können.

Der OMX(Greifarm?,Abk?) ist eine 5DOF (5 Degrees of Freedom) Plattform, welche mittels 5 Servomotoren⁵ gesteuert wird. Dies ist aufgeteilt in 4DOF für den Arm sowie 1DOF für den Greifer. Es kann eine Last bis 500g getragen werden

3.5 Planung

Planung (auch Handlungsplanung) ist ein Bereich der Künstlichen Intelligenz, welcher sich mit der Lösung von Planungs- und Schedulingproblem befasst [17].

3.5.1 Stanford Research Institute Problem Solver

Der Stanford Research Institute Problem Solver (STRIPS) ist ein automatischer Planer, welcher von Richard Fikes und Nils Nilsson 1971 entwickelt wurde [8]. Mit dem gleichen Namen wird auch die Sprache bezeichnet, welche als Eingabe für diesen Planer genutzt wird. In diesem Abschnitt wird zunächst auf die Sprache eingegangen, welche die Grundlage vieler weiterer Sprachen zur Darstellung von Planungsdomänen und -problemen ist. Anschließend wird der von STRIPS genutzte Algorithmus beschrieben. Ein STRIPS-Modell besteht aus 3 Teilen:

- Startzustand
- Ziele
- Aktionen

Aktionen bestehen wiederum aus:

- Vorbedingungen
- Effekte

Ein STRIPS-Modell kann mathematisch als 4-Tupel (P, O, I, G) beschrieben werden [2]:

1. P : eine endliche Menge von Bedingungen
2. O : eine endliche Menge von Aktionen (auch Operatoren) mit der Form $Pre \Rightarrow Post$:
3. I : der Startzustand und $I \subseteq P$
4. G : die Ziele

Es gilt die *Closed World Assumption* (CWA): alles was nicht explizit als wahr beschrieben

⁴<http://en.robotis.com>

⁵DYNAMIXEL XM430-W350-T

3 Grundlagen

ist, gilt als falsch.

P ist die Menge aller Bedingungen, die relevant sind. Jeder Zustand ist eine Teilmenge $S \subseteq P$.

O ist die Menge der Aktionen die einen Zustand in einen anderen ändern können. Aktionen sind 4-Tupel, bei dem jedes Element eine Menge von Bedingungen sind. Die ersten zwei Mengen beschreiben die Vorbedingungen (Pre): Bedingungen die wahr sein müssen (o^+) sowie Bedingungen, die falsch sein müssen (o^-). Die letzten beiden Mengen beschreiben den Effekt, den die Aktion nach ihrer Ausführung auf einen Zustand hat ($Post$): Bedingungen die wahr werden (o_+) und Bedingungen die falsch werden (o_-).

I beschreibt, welche Bedingungen anfangs wahr oder falsch sind. Eine Bedingung $p \in P$ ist anfangs wahr, wenn $p \in I$ und sonst falsch.

G ist die Menge der Bedingungen die erreicht werden soll, bestehend aus den Zielen die wahr sein sollen (G_+) und denen die falsch sein sollen (G_-). Ein Zustand $S \subseteq P$ ist ein Zielzustand, wenn $G_+ \subseteq S$ und $G_- \cap S = \emptyset$.

Ein Plan besteht aus einer Menge von Aktionen, die vom Startzustand zu einem Zielzustand führen. Dieser Plan ist ein *Total-Order* Plan: alle Aktionen haben eine feste Reihenfolge innerhalb des Plans. Um einen Plan zu finden wird ein Suchbaum erstellt. Die Suche findet im Zustandsraum statt. Jeder Knoten ist ein Zustand. Jede Kante ist die Aktion um den Zustand bzw. Knoten von der die Kante ausgeht zu dem Zustand zu überführen, zu dem die Kante geht. Es wird die GPS Strategie angewandt, um Unterschiede zwischen dem aktuellen Zustand und dem Ziel zu extrahieren und relevante Aktionen für diese Unterschiede zu finden [8].

STRIPS beginnt damit, das Ziel G_0 für den Startzustand M_0 zu beweisen. Gelingt dies, ist das Ziel bereits erreicht und kein Plan notwendig. Ist das Ziel nicht beweisbar, wird der Unterschied zwischen M_0 und G_0 ermittelt und von diesem aus weiter gesucht. Es wird eine Aktion gesucht, die diesen Unterschied weiter verringern kann. Vorbedingungen dieser Aktion werden als Unterziel genutzt. Kann ein (Unter)Ziel für den Zustand M_0 bewiesen werden, werden die Effekte der Aktion auf M_0 angewandt und es entsteht ein Zustand M_1 . Dieses Verfahren wiederholt sich bis ein Zustand entsteht, in dem das Ziel G_0 bewiesen werden kann.

3.5.2 Planning Domain Definition Language

Die Planning Domain Definition Language (PDDL) ist eine Sprache zur Modellierung von Planungsaufgaben. Im Rahmen dieser Arbeit wird nicht auf alle Aspekte von PDDL eingegangen. Es wird sich auf einige grundlegende Teile sowie auf die genutzten Erweiterungen beschränkt.

Die Beschreibung des Problems setzt sich aus folgenden grundlegenden Komponenten zusammen:

- Objekt
- Prädikat
- Startzustand

3 Grundlagen

- Aktion
- Ziel

Objekte sind Dinge in der Welt die für die Planung relevant sind. Dies können bspw. Orte oder Personen sein und werden in PDDL entsprechend Listing 2 beschrieben.

```
1 (: objects kitchen supermarket  
    fabian)
```

Listing 2: Objektbeschreibung in PDDL

Prädikate sind Eigenschaften, die Objekte haben können. Eine Person kann sich an einem Ort befinden oder eine bestimmte Größe haben (s. Listing 3).

```
(: predicates (person_at ?person ?location)  
             (tall ?person))
```

Listing 3: Prädikatbeschreibung in PDDL

Der Startzustand ist die Beschreibung der Welt zu Beginn der Planungsaufgabe. Er besteht aus einer Menge von Instanzen von Prädikaten, also Prädikate die an Objekte gebunden sind (s. Listing 4).

```
(: init (person_at fabian kitchen))
```

Listing 4: Startzustand in PDDL

Aktionen sind die vorhandenen Möglichkeiten, den Zustand der Welt zu verändern. Diese entsprechen den in Abschnitt 3.5.1 beschriebenen Aktionen mit einer Menge von Vorbedingungen und einer Menge von Effekten (s. Listing 5).

```
4 (: action move  
    : parameters (?p ?loc_from ?loc_to)  
    : precondition (and (person_at ?p ?loc_from))  
    : effect (and (person_at ?p ?loc_to)  
                  (not (person_at ?p ?loc_from))))
```

Listing 5: Aktion um eine Person von einem Ort zum anderen zu bewegen in PDDL

Ein Ziel ist eine Menge von Instanzen, die am Ende der Planung Teil des Zustands der Welt sein soll (s. Listing 6).

```
(: goal (and (person_at fabian supermarket)))
```

Listing 6: Ziel in PDDL

Diese Komponenten werden aufgeteilt:

- Domäne
- Problem

3 Grundlagen

Die Domäne enthält die allgemeinen Rahmenbedingungen für das Problem: Prädikate und Aktionen. Die Datei beginnt mit (`define (domain name_der_domäne)`).

Das Problem enthält die Informationen zu einem spezifischen Problem: Objekte, den Startzustand und das Ziel. Die Datei beginnt mit

`(define (problem name_des_problems) (:domain name_der_domäne))`.

Um sicherzustellen, dass nur bestimmte Objekte für bestimmte Parameter in Frage kommen (es kann mit der *move* Aktion nur eine Person bewegt werden und der Start und das Ziel sind Orte) wurden ursprünglich weitere Prädikate genutzt um Objekte zu typisieren. Für das aktuelle Beispiel wären das die Prädikate *location(x)* und *person(x)*. Die Aktion *move* wird dann um entsprechende Vorbedingungen erweitert (s. Listing 7).

```
(:action move
  :parameters (?p ?loc_from ?loc_to)
  :precondition (and (person ?p)
    (location ?loc_from)
    (location ?loc_to)
    (person_at ?p ?loc_from))
  :effect (and (person_at ?p ?loc_to)
    (not (person_at ?p ?loc_from))))
```

Listing 7: Move Aktion mit Prädikaten zur Typisierung

Um dies zu vereinfachen, wurde mit PDDL 2.1 unter anderem direkte Typisierung eingeführt. In der Domäne werden die möglichen Typen deklariert, Objekte werden mit einem Typ initialisiert und Aktionen können Parameter auf einen Typ beschränken (s. Listing 8)

```
(:types person
         location)
(:objects kitchen supermarket - location
         fabian - person)
(:action move
  :parameters (?p - person ?loc_from - location ?loc_to - location)
  :precondition (and (person_at ?p ?loc_from))
  :effect (and (person_at ?p ?loc_to)
    (not (person_at ?p ?loc_from))))
```

Listing 8: Typ Unterstützung in PDDL 2.1

Eine weitere Neuerung in PDDL ist die *durative action* zur Modellierung von Aktionen mit einer Dauer für das temporale Planen.

3.5.3 PDDL-Plugin für VS Code

- nach Implementierung verschieben?

Im Marketplace von Visual Studio Code ist ein PDDL-Plugin von Jan Dolejsi verfügbar. Dieses vereinfacht die Entwicklung und das Testen von Domänen und Problemen im

3 Grundlagen

PDDL Format. Es beinhaltet Syntax-Highlighting und Snippets, sowie die Möglichkeit direkt Pläne zu suchen und zu visualisieren. Um Pläne mit POPF zu suchen, muss der Planer in der Overview-Page des Plugins hinzugefügt werden. Der Pfad zum POPF-Planer ist: `/opt/ros/foxy/lib/popf/popf`.

Über das Kontext-Menü einer Domänen- oder Problemdatei lässt sich der Planer starten und es erfolgt eine Ausgabe in der Konsole (s. Abbildung 1) sowie eine Visualisierung wenn ein Plan gefunden wurde (s. Abbildung 2). Die Visualisierung erfolgt dabei zeitlich von links nach rechts. Alle Aktionen werden farblich markiert, wobei eine Aktion auch bei mehrfacher Ausführung die gleiche Farbe erhält. Im oberen Abschnitt gibt es eine einfache Übersicht des Plans: welche Aktionen wurden wann ausgeführt und mit welchen Parametern. Im unteren Abschnitt wird dargestellt, welche Instanzen, zu welchem Zeitpunkt, als Teil welcher Aktion genutzt wurden.

```
/opt/ros/foxy/lib/popf/popf /tmp/domain--2153-51GdAg05j4bm-.pddl /tmp/problem--2153-BsnbQP9gMIfm-.pddl
Constructing lookup tables: [10%] [20%] [30%] [40%] [50%] [60%] [70%] [80%] [90%] [100%]
Post filtering unreachable actions: [10%] [20%] [30%] [40%] [50%] [60%] [70%] [80%] [90%] [100%]
13% of the ground temporal actions in this problem are compression-safe
b (8.000 | 1.000)b (7.000 | 2.251)b (6.000 | 3.501)
Resorting to best-first search
b (8.000 | 1.000)b (7.000 | 1.251)b (6.000 | 2.501)b (5.000 | 3.501)b (4.000 | 3.751)b (3.000 | 3.751)b (2.000 | 4.751)b (1.000 | 5.001)g;;; Solution Found
; Time 0.12
0.000: (move-gripper gripper s1l3 s1l2) [1.000]
1.001: (unstack gripper yellow blue s1l2 s1l1) [0.250]
1.251: (move-gripper gripper s1l2 s3l1) [1.000]
2.251: (place gripper yellow s3l1 s3) [0.250]
2.501: (move-gripper gripper s3l1 s1l1) [1.000]
3.501: (grab gripper blue s1l1 s1) [0.250]
3.751: (move-gripper gripper s1l1 s2l2) [1.000]
4.751: (stack gripper blue white s2l2 s2l1) [0.250]
Planner found 1 plan(s) in 0.238secs.
```

Abbildung 1: Konsolenausgabe nach Ausführung des Planers über das PDDL-Plugin

3.5.4 Partial Order Planning

Partial Order Planning (POP) ist eine Form des Planens, in dem die Aktionen eines Plans nur soweit geordnet sind, wie es notwendig ist [6]. Dies steht im Gegensatz zu Total-Order Planern wie STRIPS.

Es wird nach dem Principle Of Least Commitment verfahren. Das bedeutet, dass Entscheidungen soweit wie möglich aufgeschoben und erst getroffen werden, wenn es absolut nötig ist. Dies umfasst nicht nur die Reihenfolge von Aktionen, sondern auch die Bindung von Variablen.

Ein Partial-Order Plan kann als 4-Tupel [9] beschrieben werden.

- eine Menge von Aktionen
- eine Menge von Einschränkungen der Reihenfolge
- eine Menge von kausalen Zusammenhängen
- eine Menge offener Vorbedingungen (Vorbedingungen einer Aktion, die nicht durch eine andere Aktion erfüllt werden)

Die Menge der Aktionen enthält Instanzen von Aktionen der Domäne. Eine Aktion der Domäne kann daher mehrfach vorkommen. Einschränkungen der Reihenfolge $A < B$ bilden ab, dass Aktion A vor Aktion B ausgeführt werden muss. Zyklische Einschrän-

3 Grundlagen

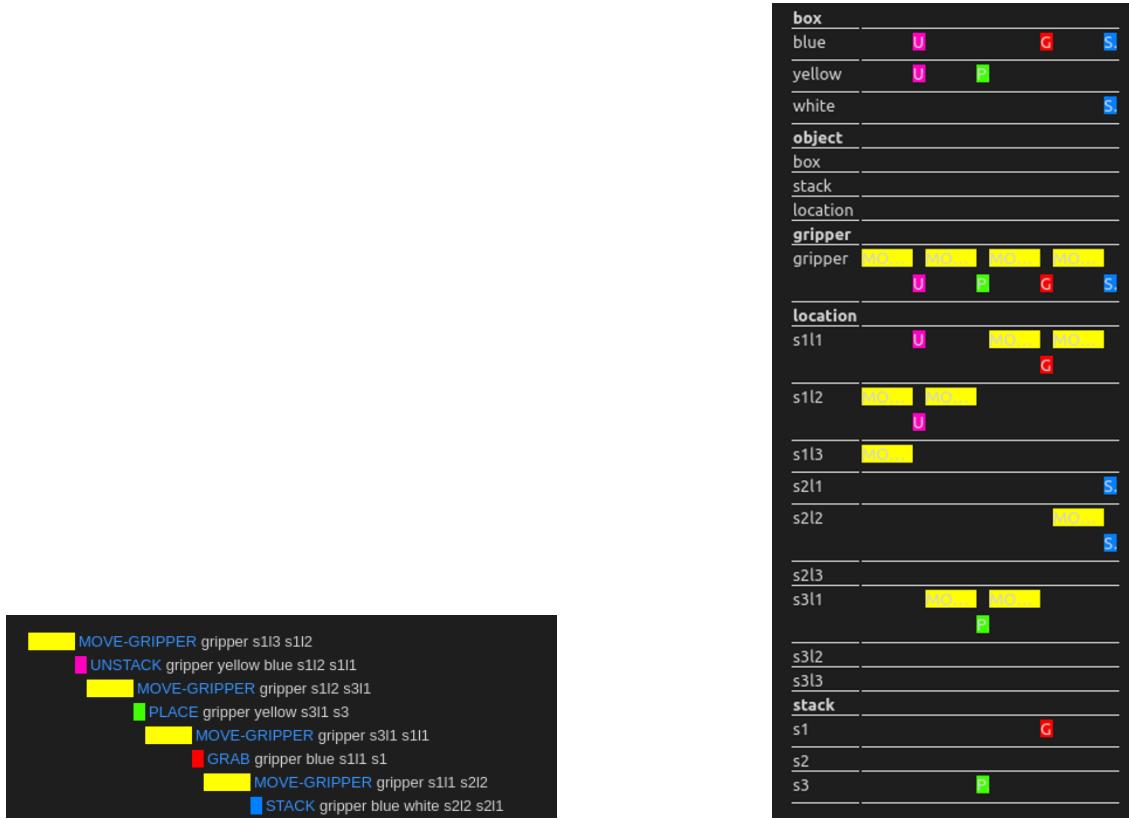


Abbildung 2: Visualisierung eines Plans mit dem PDDL-Plugin mit Fokus auf die Aktionen (links) und Fokus auf die relevanten Objekte (rechts)

kungen sind nicht erlaubt. Ein kausaler Zusammenhang $A \xrightarrow{\alpha} B$ beschreibt, dass die Aktion A die Vorbedingung α der Aktion B erfüllt.

Die Suche erfolgt im Plan Space. Jeder Knoten im Graph entspricht einem Teilplan und jede Kante einer Änderung des einen zum anderen Plan. Dies kann durch das Hinzufügen eines weiteren Schrittes oder einer Einschränkung sowie der Bindung einer Variable geschehen. Die Wurzel des Graphen besteht aus einem Plan mit zwei Schritten, die den Startzustand und das Ziel darstellen. Der Startzustand wird durch eine Aktion repräsentiert, die keine Vorbedingungen und alle Fakten als Effekt hat. Das Ziel erhält alle Zielfakten als Vorbedingung und hat keinen Effekt. Zusätzlich enthält der Plan die Einschränkung, dass die Aktion für den Start vor der für das Ziel ausgeführt werden muss ($Start < Ziel$).

Ein Plan ist eine Lösung, wenn [9]:

- keine Aktion des Plans eine offene Vorbedingung hat
- und wenn keine Threats existieren

Ist der aktuelle Plan keine Lösung werden folgende Schritte vorgenommen [9]:

- eine offene Vorbedingung p einer Aktion B wählen

3 Grundlagen

- eine existierende Aktion A , die die Vorbedingung p erfüllt wählen oder eine neue Aktion A erzeugen
- den kausalen Zusammenhang $A \xrightarrow{p} B$ und die Reihenfolge $A < B$ hinzufügen
- wenn A eine neue Aktion ist, die Reihenfolge $Start < A$ hinzufügen
- Probleme zwischen dem neuen kausalen Zusammenhang und den existierenden Aktionen lösen sowie zwischen den existierenden kausalen Zusammenhang und A , wenn A eine neue Aktion ist.
- entsteht durch das Hinzufügen der Reihenfolgen ein Zyklus kann der Plan keine Lösung mehr werden

Ein Problem ist eine Beziehung zwischen einer Aktion C und einem kausalen Zusammenhang $A \xrightarrow{p} B$, bei der die Aktion C einen Effekt $\neg p$ hat. Sollte C nach A aber vor B ausgeführt werden, ist die Vorbedingung p für B nicht mehr gegeben. Probleme können durch *Promotion* oder *Demotion* gelöst werden [6]. Bei *Promotion* wird der problematische Schritt C nach dem kausalen Zusammenhang ausgeführt: die Reihenfolge $B < C$ wird hinzugefügt. Bei *Demotion* wird der problematische Schritt C vor dem kausalen Zusammenhang ausgeführt: die Reihenfolge $C < A$ wird hinzugefügt.

3.5.5 Forward Chaining Partial Order Planning

Die Prinzipien des POP sollen nun mit Vorwärtsverkettung kombiniert werden. Für Vorwärtsverkettung mit einem Total-Order Plan können Zustände als Tupel (F, V, Q, P, C) beschrieben werden [4]:

- F ist eine Menge von Fakten die im Zustand gelten
- V enthält die Werte numerischer Variablen
- Q ist eine List von Aktionen die begonnen haben aber noch nicht beendet sind
- P ist der Plan um den Zustand zu erreichen
- C ist eine Liste aller zeitlichen Einschränkungen der Schritte in P

Um eine Aktion zum Plan hinzuzufügen müssen deren Start- und Endpunkte hinzugefügt werden. Dies muss nicht in aufeinander folgenden Schritten geschehen. Wird eine Aktion hinzugefügt werden F und V entsprechend der Effekte angepasst. Die Voraussetzung um eine Aktion hinzufügen zu können ist, dass Q keine Aktion enthält, zu der die Effekte der neuen Aktion einen Konflikt darstellen würden.

Vorwärtsverkettung im Zustandsraum hat, durch den Total-Order Plan, den Vorteil, dass nicht explizit nach Problemen gesucht werden muss. Neue Aktionen werden immer am Ende hinzugefügt, wodurch sie kein Problem für die Vorbedingungen und Effekte früherer Aktionen darstellen können. Gleichzeitig kann keine folgende Aktion ein Problem für diese darstellen. Dieser Vorteil kommt mit dem Nachteil der frühen Bindung und einer festgelegten Reihenfolge zwischen Aktion, zwischen denen kein Zusammenhang besteht.

Es soll ein Kompromiss zwischen den Vorteilen von Total- und Partial-Order Planung gefunden werden. Dem Tupel werden weitere Elemente hinzugefügt:

3 Grundlagen

- $F^+(F^-)$, wobei $F^+(p)(F^-(p))$ den Index eines Schrittes a angeben, indem der Fakt p als letztes hinzugefügt (oder gelöscht) wurde
- FP , wobei $FP(p)$ eine Menge von Paaren $\langle i, d \rangle \in (N_0 \times \{0, \epsilon\})$:
 - $\langle i, 0 \rangle \in FP(p)$ beschreibt, dass Schritt i am Ende eines offenen Intervalls liegt, in dem p wahr sein muss.
In PDDL gibt es diese für Aktionen mit der *over all* Bedingung p , wobei i der Endpunkt dieser Aktion ist.
 - $\langle i, \epsilon \rangle \in FP(p)$ beschreibt, dass Schritt i der Start eines Intervalls ist, indem p wahr sein muss.
Dies entspricht *at start* oder *at end* Bedingungen in PDDL die relevant für den Schritt i sind.

3.5.6 Partial Order Planning Forward

POPF ist ein am King's College London entwickelter Planer, der die in Abschnitt 3.5.4 genannten Methoden implementiert [4]. Als Heuristik wird ein temporaler RPG genutzt. Ein RPG ist ein Planungsgraph, in dem versucht wird ein vereinfachtes Problem zu lösen. Die Vereinfachung besteht daraus, alle negativen Effekte von Aktionen zu ignorieren.

Der Graph besteht aus einer Sequenz $F_0, A_0, \dots, F_{t-1}, A_{t-1}, F_t, A_t$: Aktions- und Fakten schichten, die sich abwechseln. Die Wurzel des Graphen ist eine Faktenschicht, die den aktuellen Zustand abbildet. Eine Aktionsschicht enthält alle Aktionen, die in der vorhergehenden Faktenschicht anwendbar sind. Eine Faktenschicht enthält alle Fakten, die durch alle Aktionen der vorhergehenden Aktionsschicht hinzugefügt werden. Da nur positive Effekte betrachtet werden, enthält eine Faktenschicht auch alle Fakten der vorherigen Faktenschicht und eine Aktionsschicht alle Aktionen der vorherigen Aktionsschicht.

Ein Beispiel eines RPG ist in Abbildung 3 dargestellt. Im Startzustand gelten die Fakten *at(Home)* und *has(Money)*, dargestellt in F_0 . Die einzige Aktion, deren Vorbedingungen erfüllt ist, ist *move(Home, Shop)* und wird der Aktionsschicht A_0 hinzugefügt. Pfeile, die zu Aktionen gehen, zeigen die Fakten, die als Vorbedingung gelten. Pfeile, die zu Fakten gehen, zeigen die Aktionen durch deren Effekt sie hinzugefügt wurden. In F_1 wird als Effekt der Aktion *move(Home, Shop)* der Fakt *at(Shop)* hinzugefügt. Da negative Effekte ignoriert werden, gilt *at(Home)* weiterhin. Da in F_1 nun *at(Shop)* und *has(Money)* gelten, kann in A_1 die Aktion *buy(Product, Money)* angewendet werden, was in dem Fakt *has(Product)* in F_2 resultiert.

Angenommen, das Ziel ist *has(Product)*, so ist dieses Ziel in F_2 erfüllt. Mit $t = 2$ als Wert der Heuristik, wissen wir, dass mindestens zwei Aktionen nötig sind um vom Startzustand zum Ziel zu gelangen. Da *at(Home)* vom Startzustand aus durchgängig gilt, erhält jedoch auch das Ziel $\{\text{at}(Home), \text{has}(Product)\}$ den Heuristikwert 2.

Die Erstellung des Graphen wird gestoppt wenn eine Faktenschicht alle Ziele erfüllt oder wenn eine Faktenschicht der vorherigen entspricht. Werden zwischen zwei Fakten-

3 Grundlagen

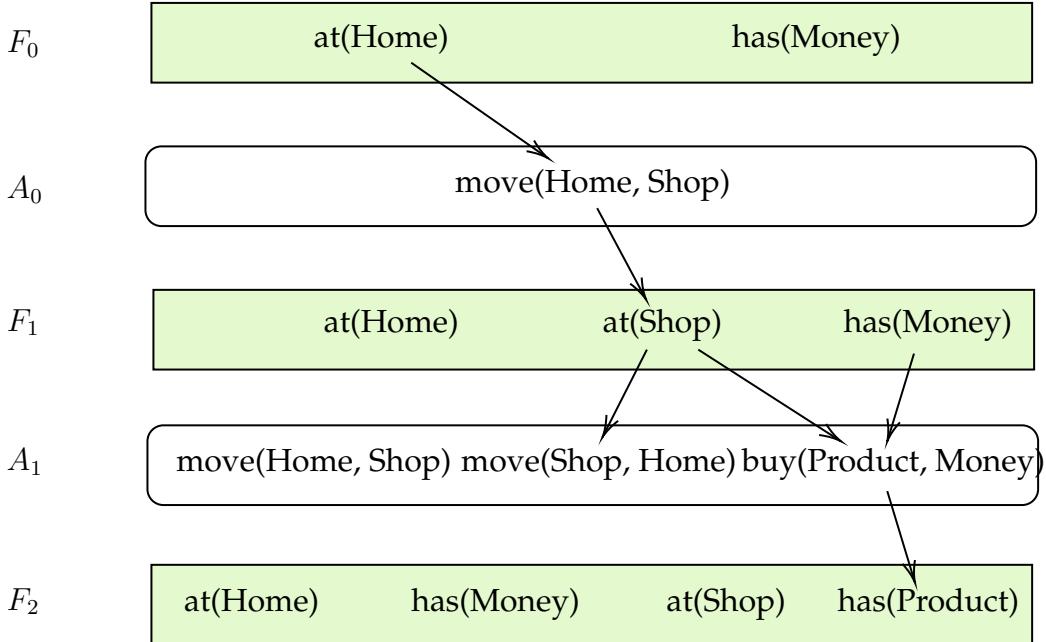


Abbildung 3: RPG für das Ziel $\{at(Home), has(Product)\}$

schichten keine neuen Fakten hinzugefügt, können auch in zukünftigen Faktenschichten keine neuen Fakten mehr hinzugefügt werden. Endet ein Graph auf diese Weise ist das vereinfachte Problem nicht lösbar und der Zustand erhält einen Heuristikwert von unendlich. Andernfalls wird ein vereinfachter Plan aus dem Graphen extrahiert. Die Länge des optimalen vereinfachten Plans ist eine zulässige Heuristik. Der optimale Plan lässt sich jedoch nicht effizient extrahieren [10]. Daher wird ein suboptimaler Plan extrahiert, wodurch die Heuristik nicht mehr zulässig ist.

Um einen RPG für temporales Planen zu erweitern, werden Aktionen in Startaktionen A_{\leftarrow} und Endaktionen A_{\rightarrow} aufgeteilt. Der Index t einer Schicht entspricht nun einer Zeitmarke.

Als Suchverfahren wird zunächst EHC versucht. Der in Abbildung 4 beschriebene Algorithmus beginnt im Startzustand S . Von diesem aus wird eine *breadth-first* Suche gestartet. Diese findet den nächstbesten Nachfolger, also den nächsten Zustand S' , dessen Heuristik besser als die des aktuellen Zustands ist. Gelingt dies, wird der Weg von S nach S' zum Plan hinzugefügt. Die Suche wird gestoppt, wenn ein Zustand mit der Heuristik 0 gefunden wird. Ist die Heuristik des aktuellen Zustands $\neq 0$ und wird kein Zustand mit einer besseren Heuristik gefunden, schlägt der Algorithmus fehl. Zustände werden in einer *Queue* gespeichert und für eine Iteration der Suche wird der erste Zustand S' von dieser entfernt und die Heuristik berechnet. Die Suche ist erfolgreich, wenn die Heuristik für diesen Zustand S' besser als die von S ist. Andernfalls werden die Nachfolger von S' zum Ende der *Queue* hinzugefügt. Wiederholte Zustände werden durch ein *HashTable* der bereits besuchten Zustände vermieden. Die Suche schlägt fehl, wenn keine weiteren Zustände erreicht werden können.

3 Grundlagen

```
initialize the current plan to the empty plan <>
S := I
while h(S) ≠ 0 do
    perform breadth first search for a state S' with h(S') < h(S)
    if no such state can be found then
        output "Fail", stop
    endif
    add the actions on the path to S' at the end of the current plan
    S := S'
endwhile
```

Abbildung 4: EHC Algorithmus [10]

Wird in einer Iteration kein besserer Zustand gefunden, stoppt EHC ohne eine Lösung zu finden. Dies geschieht, da EHC Entscheidungen, eine Aktion in den Plan aufzunehmen, nie rückgängig macht. Der Algorithmus ist daher nur vollständig, für Planungsaufgaben die keine "dead ends" enthalten. Ein *Dead End* für eine Planungsaufgabe (O, I, G) ist definiert, als ein Zustand S , der erreichbar ist und von diesem aus keine Plan das Ziel erreichen kann:

$$\exists P : S = \text{Result}(I, P) \text{ und } \neg \exists P' : G \subseteq \text{Result}(S, P')$$

Schlägt die Suche mit EHC fehl, wird eine neue *best-first* Suche ausgehend vom ursprünglichen Startzustand gestartet.

3.5.7 Behavior Tree

BTs sind ein modulares System, das in der Videospieleindustrie entwickelt wurde [5]. Sie ermöglichen aus simplen Aktionen komplexe Verhaltensmuster zu modellieren ohne das Aktionen Kenntnis über andere Aktionen haben müssen [1]. Entsprechend [13] ist ein BT ein gerichteter azyklischer Graph $G(V, E)$ mit $|V|$ Knoten und $|E|$ Kanten. Ein Eltern Knoten von einem Paar verbundener Knoten ist der, von dem die Kante ausgeht. Der Knoten mit der eingehenden Kante ist ein Kind Knoten. Knoten ohne Kinder heißen Blätter. Ein Knoten ohne Eltern heißt Wurzel. In einem BT gibt es genau eine Wurzel mit genau einem Kind.

Nach [5] beginnt die Ausführung eines BT bei der Wurzel, die in einem bestimmten Intervall ein Signal erzeugt, dass die Ausführung eines Knoten erlaubt ("tick"). Dieses Signal wird an das Kind der Wurzel gesendet. Ein Knoten wird ausgeführt, wenn es dieses Signal erhält und auch nur dann. Wird ein Knoten ausgeführt, gibt er einen Status zurück. Der Status ist einer von drei möglichen [11]:

- Running: die Ausführung läuft noch
- Success: die Ausführung war erfolgreich
- Failure: die Ausführung ist fehlgeschlagen

Wie in [13] beschrieben, werden verschiedene Arten von Knoten auf verschiedene Ty-

3 Grundlagen

pen beschränkt. Blätter sind Ausführungsknoten und haben einen der folgenden Typen:

- Aktion
- Bedingung

Knoten mit Kindern, mit Ausnahme der Wurzel, sind Kontrollfluss Knoten und haben einen der folgenden Typen:

- Selektor
- Sequenz
- Parallel
- Dekorator

Ein Selektor Knoten führt seine Kinder der Reihe nach aus bis das erste einen Status *Running* oder *Success* zurückgibt. Nachfolgende Kinder werden dann nicht mehr ausgeführt und der Selektor gibt den entsprechenden Status zurück. Wurden alle Kinder ausgeführt und keins hat einen Status *Running* oder *Success* zurückgegeben, gibt der Selektor den Status *Failure* zurück.

Ein Sequenz Knoten führt seine Kinder der Reihe nach aus, bis das erste einen Status *Running* oder *Failure* zurückgibt. Nachfolgende Kinder werden dann nicht mehr ausgeführt und die Sequenz gibt den entsprechenden Status zurück. Wurden alle Kinder ausgeführt und keins hat einen Status *Running* oder *Failure* zurückgegeben, gibt die Sequenz den Status *Success* zurück.

Ein Parallel Knoten führt alle Kinder aus, ohne auf den Status des vorherigen zu warten. Überschreitet die Anzahl der Kinder die *Success* zurückgeben einen benutzerdefinierten Wert *S* gibt der Knoten *Success* zurück. Haben $N - M + 1$ Kinder *Failure* zurückgegeben, wobei *N* die Anzahl der Kinder ist, wird *Failure* zurückgegeben, da der Wert *S* nicht mehr erreicht werden kann [5]. Alternativ kann ein zweiter benutzerdefinierter Wert *F* eingeführt werden. Überschreitet die Anzahl der Kinder die *Failure* zurückgeben diesen Wert *F* wird direkt *Failure* zurückgegeben [13]. *S* und *F* müssen $\leq M$ sein. Wird keiner dieser beiden Werte überschritten, wird *Running* zurückgegeben.

Ein Dekorator Knoten hat genau ein Kind. Er kann durch eine Bedingung entscheiden ob das Kind ausgeführt wird und kann einen modifizierten Status zurück geben. Beispiele für Dekoratoren sind:

- Invertierung/Negation: der *Success* und *Failure* Status des Kinds wird invertiert
- Zeitbeschränkung: gibt das Kind den Status *Running* für länger als eine benutzerdefinierter Zeitspanne *T* zurück, wird *Failure* zurückgegeben und das Kind nicht mehr ausgeführt

[5]

- Visualisierungs bsp.

3 Grundlagen

Knoten	Symbol	Rückgabewert		
		Success	Failure	Running
Selektor		Wenn 1 Kind "Success" zurückgibt	Wenn alle Kinder "Failure" zurückgeben	Wenn 1 Kind "Running" zurückgibt
Sequenz		Wenn alle Kinder "Success" zurückgeben	Wenn 1 Kind "Failure" zurückgibt	Wenn 1 Kind "Running" zurückgibt
Parallel		Wenn $\geq M$ Kinder "Success" zurückgeben		Wenn keine der Bedingungen für "Success" oder "Failure" erfüllt sind
Dekorator		benutzerdefiniert	benutzerdefiniert	benutzerdefiniert
Bedingung		Wenn wahr	Wenn falsch	Nie
Aktion		Bei vollständiger Ausführung nicht möglich ist	Wenn vollständige Ausführung nicht möglich ist	Während der Ausführung

Tabelle 3: Übersicht der BT Knoten, nach [5]

3.6 Planung in ROS2

- config für andere Planer

Für die Planung in ROS2 wird PlanSys2 genutzt. In diesem Abschnitt wird das in [12] beschriebene System zusammenfassend beschrieben. Es wird dabei jedoch nur auf die für diese Arbeit relevanten Teile eingegangen.

PlanSys2 ist ein modulares OpenSource Planungsframework. Ursprünglich als Nachfolger für ROSPlan [3] geplant, bietet es viel umfangreichere Funktionen. Hierfür werden unter anderem neue Funktionen von ROS2 wie LifeCycle Knoten oder Multicast Kommunikation genutzt.

Eins der Designprinzipien bei der Entwicklung von PlanSys2 ist Modularität und Erweiterbarkeit. Alle Komponenten haben klar definierte Schnittstellen, wodurch sie einfach erweitert oder ausgetauscht werden können.

Die Hauptkomponente ist der *Planner* Knoten, der den tatsächlichen Planer aufruft. Verschiedene Planer können über Plugins aufgerufen werden, die beschreiben, wie der Planer aufzurufen ist und wie der generierte Plan zu verstehen ist. Standardmäßig sind POPF sowie TFD (Temporal Fast Downward) [7] verfügbar und konfiguriert.

Der *Domain Expert* Knoten liest die Domäne im PDDL Format ein. Er kann mehrere verschiedene Domänen zu einer kombinieren. Dies ermöglicht es mehrere modulare Packages zu nutzen, die jeweils nur den für sie relevanten Teil einer Domäne beinhalten sowie deren Aktionen implementieren. Der Knoten dient zur Validierung (z.B. ob ein Prädikat, dass hinzugefügt werden soll mit den gegebenen Objekten gültig ist) und dem Planer die Domäne zur Verfügung zu stellen.

Der *Problem Expert* Knoten enthält alle Informationen zum gegebenen Problem: Objekte, Prädikate, Funktionen und Ziele. Diese werden immer über den *Domain Expert* validiert. Wenn ein Plan angefordert wird, werden die gespeicherten Informationen zu einem Problem im PDDL Format konvertiert und dem *Planner* Knoten zur Verfügung gestellt.

Der *Executor* Knoten ist dafür zuständig einen generierten Plan auszuführen. Der Plan wird in einen BT konvertiert, der diesen ausführt.

3 Grundlagen

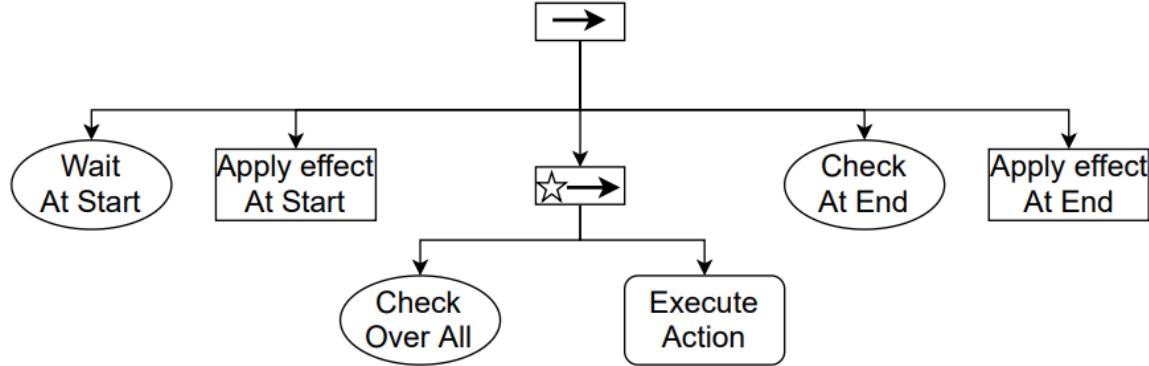


Abbildung 5: BT zur Ausführung einer spezifischen Aktion [12]

Ein weiteres Konzept ist die Unterstützung für mehrere Roboter sowie eine Spezialisierung bei der Ausführung von Aktionen. Alle Knoten im gleichen Netzwerk können miteinander kommunizieren. Es kann mehrere Knoten geben die die selbe Domänenaktion ausführen, sich aber auf bestimmte Parameter spezialisieren. Mehrere Roboter können einen Knoten für die gleiche Aktion (*move roboter start ziel*) ausführen, wobei sich jede darauf spezialisiert, die Aktion auszuführen deren erster Parameter mit dem entsprechenden Roboter übereinstimmt. Die gleiche Aktion kann je nach vorhandenen Parametern von verschiedenen Knoten ausgeführt werden.

BTs eignen sich für die Ausführung eines Plans, da sich mit ihnen sehr gut sequentielle und parallele Vorgänge beschreiben lassen. Zunächst wird von einem Plan ein Ausführungs-Graph erstellt, der die Abhängigkeiten zwischen Aktionen zeigt. Von diesen lässt sich ableiten welche Aktionen parallel ausgeführt werden können und welche sequentiell ausgeführt werden müssen. Semantisch entspricht jedes Blatt im Graphen einer Aktion. In der Praxis ist jedes Blatt ein eigenständiger BT, der nicht nur die selbst Aktion ausführt, sondern auch die Bedingungen überprüft und die Effekte ausführt (s. Abbildung 5).

PlanSys2 hat drei große Limitationen. PDDL Unterstützung beschränkt sich aktuell auf PDDL 2.1, mit dem Ziel zukünftig die aktuellste Version zu unterstützen. Wird eine Aktion und damit ein Plan abgebrochen, gibt es keine Garantie, dass der Zustand der Welt konsistent bleibt. Die Fähigkeit von Planern, die zunächst einen allgemeinen Plan liefern und diesen während der Ausführung verbessern, wird nicht genutzt. Es wird jedoch daran gearbeitet Übergänge von der Ausführung eines Plans zu einem anderen zu ermöglichen.

4 Konzept

labe

4 Konzept

In diesem Abschnitt wird der Ansatz beschrieben, mit dem in dieser Arbeit vorgegangen werden soll.

4.1 Inbetriebnahme

Im ersten Schritt muss der OMX in Betrieb genommen werden. Dies beinhaltet den Zusammenbau und Konfiguration sowie die Installation nötiger Software. Ein besonderes Augenmerk wird dabei darauf gelegt, Details festzuhalten welche in den gegebenen Anleitungen nicht explizit angegeben werden oder leicht zu übersehen sind.

4.2 Struktur Nodes

Um eine einfache sowie übersichtliche Steuerung des Greifarms zu ermöglichen wird die Funktionalität in mehrere ROS2 Nodes aufgeteilt. Generell werden folgende Funktionalitäten benötigt: der Planer, die Speicherung des aktuellen Zustands der Welt, die Ausführung des Plans sowie die Möglichkeit Eingaben zu verarbeiten und an die entsprechenden Nodes weiterzuleiten.

Diese Aufteilung entspricht auch einer guten Aufteilung und Trennung der Verantwortungen um daraus ROS2 Nodes zu machen. Hier haben die Nodes folgende Verantwortlichkeiten:

Die Planungs-Node muss mit einer gegebenen Domäne und einem Problem einen Plan bestehend aus einer Reihe von Aktionen zurückgeben.

Die Welt-Node hält den aktuellen Zustand der Welt bzw. des Problems und muss diesen konsistent halten.

Die Ausführungs-Node muss entsprechend eines Plans die gegebenen Aktionen ausführen.

Die Eingabe-Node ermöglicht die Erstellung eines Problems mit einem Welt-Zustand und einem Ziel.

4.3 Planungsmodell

Für die Erstellung von Plänen wird eine Domäne im PDDL Format erstellt. In der Domäne werden die Aspekte des Stapelns von Blöcken mit denen des Greifers kombiniert. Zusätzlich werden *durative actions* genutzt, um die Dauer der einzelnen Aktionen zu modellieren.

4.4 Erstellung von Übungsbeispielen

Abschließend soll eine Übersicht, mit Möglichkeiten den OMX in die Lehre einzubinden, erstellt werden. Dies umfasst spezifische Beispiele sowie allgemeine Ansatzpunkte, um die Benutzung des OMX sowie dessen grundlegenden Konzepte zu vermitteln.

5 Implementierung

5.1 Inbetriebnahme Greifarm

Der OMX wird als Bausatz geliefert. Für die Inbetriebnahme ist daher der Zusammenbau und die Installation der entsprechenden Software nötig.

5.1.1 Zusammenbau

Der Bausatz des OMX besteht aus ca. 60 Teilen (ohne Schrauben, s. Abbildung 6). Einige der mit den Servomotoren mitgelieferten Teile werden dabei nicht benötigt, da der Bausatz des OMX diese auch enthält oder ersetzt (z.B. längere Kabel). Von allen Schrauben wurde außerdem Ersatz mitgeliefert.

Der Zusammenbau erfolgte nach der auf der Webseite verfügbaren Bauanleitung VERWEIS ANLEITUNG. Zu beachten ist, dass hier vorausgesetzt wird, dass den Servos bereits die IDs 11 (Basis des Greifarms) bis 15 (Greifer) zugewiesen wurden. Dies kann über die Software DYNAMIXEL Wizard⁶ gemacht werden: die Servos einzeln über das U2D2 (s. Abschnitt 5.2.1) an den PC anschließen, die ID setzen und den Servo entsprechend markieren oder die ID merken. Weiterhin müssen bei den Abdeckungen der Servos 12 und 14 die vorgestanzten Abdeckungen herausgebrochen werden. Dies ist in der Anleitung leicht zu übersehen. Weiterhin wird angenommen, dass das Horn der Servos bereits angebracht ist. Hierbei ist darauf zu achten, dass die Einkerbung an Horn und Servo übereinstimmen.

Der komplette Zusammenbau erfolgte in ca. 2,5 Stunden.



Abbildung 6: Bausatz für den OpenMANIPULATOR-X

⁶VERWEIS ODER LINK

5 Implementierung

5.1.2 Virtuelle Maschine

Zur Nutzung des Greifarms wurde eine VM mit VirtualBox⁷ von Oracle aufgesetzt. Als Betriebssystem der VM wurde das für ROS2 Foxy empfohlene [15] Ubuntu 20.04⁸ gewählt. Danach wurde entsprechend der Anleitung für den OpenMANIPULATOR-X [14] zuerst ROS 2 Foxy über das Installations-Skript von ROBOTIS und im Anschluss die für den Greifarm benötigten Packages installiert.

5.2 Steuerung OMX

Die Steuerung des OMX erfolgt über die vom *open_manipulator_x_controller* *open_manipulator_x_controller* zur Verfügung gestellten Topics und Services.

5.2.1 Anschluss über U2D2

Der OMX wird über das U2D2 und das U2D2 Power Hub Board (s. Abbildung 7) per USB an den PC angeschlossen.

Das U2D2 konvertiert die Signale der DYNAMIXEL und ermöglicht die Kontrolle über den PC. Zur Nutzung des U2D2 in der VM muss das Gerät *FTDI USB <-> Serial Converter* über das Geräte-Menü der VM an diese gebunden werden (s. Abbildung 8).

5.2.2 OMX-Controller

Der OMX-Controller ist ein Package, welches automatisch beim Installieren der für den OMX benötigten Software installiert wird. Er kann über die entsprechende Launch-Datei mit dem Befehl

```
ros2 launch open_manipulator_x_controller  
    open_manipulator_x_controller.launch.py
```

gestartet werden.

Für eine Nutzung des OMX ist dieses Package zwangsläufig nötig, da es alle Steuerungsmöglichkeiten sowie Informationen über den OMX zur Verfügung stellt.

5.2.3 Joint und Task Space

Der OMX kann in den zwei verschiedenen Arbeitsräumen *Joint Space* und *Task Space* betrachtet werden.

Im *Joint Space* werden die Gelenke und ihre aktuellen Winkel betrachtet.

Der *Task Space* entspricht einer Betrachtung in einem kartesischen Koordinatensystem, mit dem Motor mit der ID 11 als Ursprung. Zusätzlich zur Position gibt es eine Orientierung für jede Achse (Roll-Nick-Gier, engl. roll-pitch-yaw).

⁷<https://www.virtualbox.org>

⁸<https://releases.ubuntu.com/20.04/>

5 Implementierung

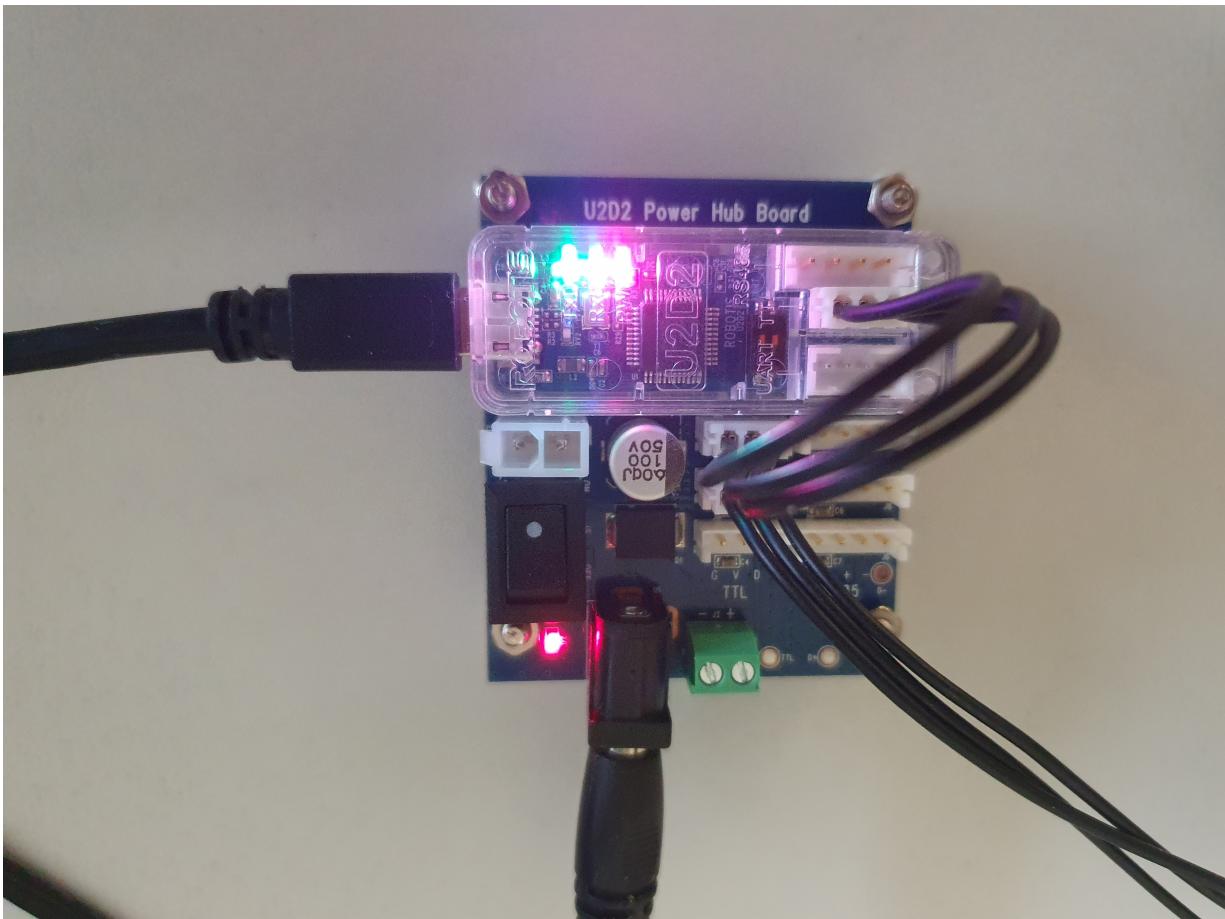


Abbildung 7: U2D2 Power Hub Board mit montiertem U2D2 (rechts im Bild)

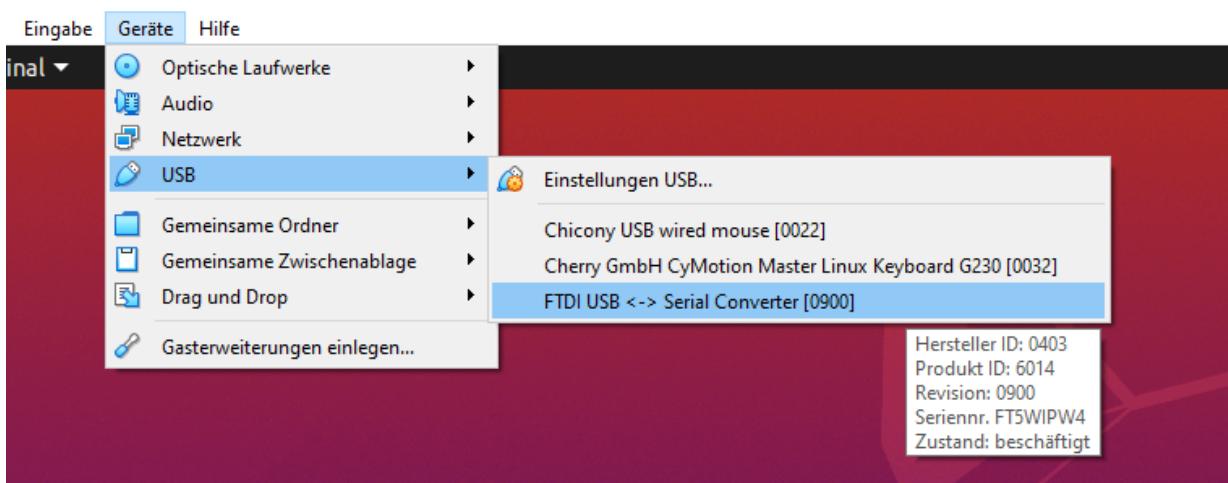


Abbildung 8: Geräte-Menü der VM mit angeschlossenem U2D2

5 Implementierung

5.2.4 Topics

Der OMX-Controller veröffentlicht drei Topics, die Informationen über den aktuellen Status geben:

- /states
- /joint_states
- /kinematics_pose

/states stellt allgemeine Informationen über die Motoren und die Bewegung des OMX zur Verfügung. Ob die Motoren aktiviert sind, wird über das Feld *open_manipulator_actuator_state* mit den möglichen Werten *ACTUATOR_ENABLE* und *ACTUATOR_DISABLE* angegeben. Der Bewegungsstatus, mit den möglichen Werten *IS_MOVING* und *STOPPED*, steht im Feld *open_manipulator_moving_state*.

/joint_states stellt Informationen über alle Gelenke zur Verfügung. Dies umfasst den Namen, den aktuell verbrauchten Strom, die Position (der Winkel des Motors) sowie die Geschwindigkeit.

/kinematics_pose stellt Informationen über die Pose im Task Space zur Verfügung. Eine Pose besteht dabei aus einer Position und einer Orientierung und bezieht sich auf den Greifer. Die Position enthält die Koordinaten der Mitte des Greifers. Die Orientierung gibt die Rotation als Quaternion an.

5.2.5 Kinematik

Der OMX lässt sich über zwei Formen der Kinematik steuern. Zum eine über die direkte Kinematik (auch Vorwärtsskinematik, engl. forward kinematics, FK), was einer Steuerung im Joint Space entspricht und zum anderen über die inverse Kinematik (engl. inverse kinematics, IK), was einer Steuerung im Task Space entspricht.

Die komplette kinematische Steuerung erfolgt über Services, die sich dementsprechend darin unterscheiden, ob Winkel für die Gelenke oder Posen für den Greifer angegeben werden. Bei der Steuerung über Posen besteht wiederum die Möglichkeit Position und Orientierung oder nur eines der beiden zu steuern und den aktuellen Wert des anderen beizubehalten.

Die komplette Liste der Services ist in der Anleitung des OMX⁹ zu finden.

5.2.6 Teleop

Mit einem laufenden OMX-Controller kann der OMX auch ohne extra Programmierung direkt ferngesteuert werden. Mögliche Geräte zur Steuerung sind die Tastatur sowie Playstation- und XBOX-Controller. Für ROS2 Foxy wird aktuell allerdings nur die Steuerung über die Tastatur unterstützt.

Der OMX kann dabei sowohl im Task Space als auch im Joint Space kontrolliert werden

⁹https://emanual.robotis.com/docs/en/platform/openmanipulator_x/ros_controller_msg/#message-list

5 Implementierung

(s. Abbildung 9). Zusätzlich gibt es 2 vordefinierte Posen (Init und Home) in die der OMX bewegt werden kann (s. Abbildung 10).

```
Control Your OpenManipulator!
-----
Task Space Control:
    (Forward, X+)
    W           Q (Upward, Z+)
(Left, Y+) A   D (Right, Y-)   Z (Downward, Z-)
    X
    (Backward, X-)

Joint Space Control:
- Joint1 : Increase (Y), Decrease (H)
- Joint2 : Increase (U), Decrease (J)
- Joint3 : Increase (I), Decrease (K)
- Joint4 : Increase (O), Decrease (L)
- Gripper: Increase (F), Decrease (G) | Fully Open (V), Fully Close (B)

INIT : (1)
HOME : (2)

CTRL-C to quit

Joint Angle(Rad): [0.000, 0.000, 0.000, 0.000, 0.000]
Kinematics Pose(Pose X, Y, Z | Orientation W, X, Y, Z): 0.000, 0.000, 0.000 | 0.000, 0.000, 0.000, 0.000
```

Abbildung 9: Fernsteuerung des OMX über die Tastatur

5.3 PlanSys2

[12]

Für die Implementierung der in Abschnitt 4.2 genannten Funktionalitäten wird das Framework PlanSys2 genutzt. Es übernimmt dabei alle Funktionalitäten: die Verwaltung der Daten zur Domäne und dem aktuellen Problem erfolgt über die domain-expert und problem-expert Nodes. Die planer Node ist für die komplette Planung zuständig und die Executioner Node für die Ausführung des Plans.

Es müssen hier lediglich die Domäne erstellt sowie die tatsächliche Funktionalität der einzelnen Aktionen implementiert werden. Das Wort Action im Namen der Nodes und den folgenden Kapiteln bezieht sich hier auf die Aktionen der Domäne, nicht auf ROS2 Actions.

5.3.1 PDDL-Domain

Beim erstellen der Domäne ist zu beachten, das POPF nicht alle Funktionen unterstützt, die mit PDDL beschrieben werden können. Dies beinhaltet unter anderem existentielle sowie negative Vorbedingungen. Die vollständige Liste ist auf ?? zu finden.

Für den Blockworld Teil der Domäne gibt es zunächst den Typ `box` und Prädikate, die

5 Implementierung



Abbildung 10: Init (l.) und Home (r.) Pose der Tastatur Teleop

das Stapeln von Blöcken darstellen. Dies umfasst das Verhältnis von Blöcken aufeinander ((box_on ?b_above ?b_below - box)) sowie ob ein Block der oberste eines Stapels und damit frei zum bewegen ist ((clear ?b - box)). Als Aktionen werden das Nehmen sowie Ablegen eines Blocks benötigt. Da in den Bedingungen und Effekten einer Aktion nur mit Parametern gearbeitet werden kann, welche auch Parameter der Aktion selbst sind, wird hier weiter aufgeteilt in Nehmen/Ablegen eines Blocks von/auf einen anderen Block sowie von/auf einen leeren Stapel. Für das Bewegen von Blöcken ergeben sich die 4 Aktionen GRAB, PLACE, STACK, UNSTACK.

Während für das reine Stapeln der Blöcke die exakte Position dieser nicht wichtig ist, wird diese jedoch für den Greifer benötigt, welcher vor diesen Aktionen an die richtige Stelle bewegt werden muss. Hier werden die weiteren Typen location und stack sowie die Prädikate (box_at ?b - box ?l - location),
(gripper_at ?g - gripper ?l - location),
(location_above ?l_above ?l_below - location) und
(is_base_loc ?l - location ?s - stack) eingeführt.

Das Prädikat clear ist notwendig, da POPF weder existentielle noch negative Vorbedingungen unterstützt. Es wird genutzt um zu überprüfen, ob ein Block der oberste eines Stapels ist, was es ermöglicht diesen zu nehmen oder einen anderen auf diesem abzulegen. Durch existentielle und negative Vorbedingungen könnte dies für die Aktion den Block B zu greifen, durch eine Bedingung der Form "es existiert kein Block C für den gilt:(box_on C B)" ersetzt werden.

Da alle Aktionen eine Dauer haben und PlanSys2 auch nur diese unterstützt, werden alle Aktionen als durative-action modelliert. Es wird jeder Aktion eine Dauer zugeordnet (:duration (= ?duration 0.25)) welche auch für Metriken genutzt werden kann. Zusätzlich bekommen Bedingungen und Effekte einen temporalen Zusatz: at start und at end können sowohl für Bedingungen als auch Effekte benutzt werden, um festzulegen zu welchem Zeitpunkt der Aktion diese Bedingung zutreffen oder

5 Implementierung

der Effekt eintritt. Für Bedingungen gibt es außerdem `over all` damit eine Bedingung über die komplette Dauer der Aktion zutreffen muss damit sie gültig ist.

Da Bedingungen dadurch nicht mehr strikt vor Ausführung der Aktion gelten müssen, wird das Schlüsselwort `:precondition` zu `:condition` geändert. Im aktuellen Szenario wird `over all` dafür genutzt, allen Block-Aktionen die Bedingung zu geben, dass sich der Greifer über die komplette Zeit an der gleichen Stelle wie der entsprechende Block befinden muss.

Nur in der MOVE-GRIPPER Aktion wird `at start` für einen Effekt genutzt:

```
(at start (not (gripper_at ?g ?l_from))).
```

Sobald die Aktion beginnt befindet sich der Greifer nicht mehr an der Ausgangsposition und erst am Ende der Aktion wird die neue Position als Fakt gesetzt:

`(at end (gripper_at ?g ?l_to)).` Würden beide Effekte erst am Ende der Aktion eintreten, würde der Planer mehrere MOVE-GRIPPER Aktionen parallel ausführen (s. Abbildung ??). Dies ist zum einen physisch nicht möglich, hinterlässt die Welt aber auch in einem ungültigen Zustand, da der Greifer sich dann an mehreren Positionen gleichzeitig befinden würde.

5.3.2 Action Nodes

Obwohl in der Domäne mehr als 2 Aktionen beschrieben sind, beschränken sich die ausgeführten Aktionen auf ein Bewegen des OMX sowie die Kontrolle des Greifers. Für diese wird jeweils eine Node erstellt. Damit diese Nodes von PlanSys2 genutzt werden können, müssen sie von der Basisklasse `plansys2::ActionExecutorClient` erben. Da diese von PlanSys2 nur in C++ zur Verfügung gestellt wird, müssen auch die Aktionen in C++ implementiert werden.

In jeder Aktion muss die Methode `void do_work()` implementiert werden. Diese wird mit einem bestimmten Zeitintervall aufgerufen während die Aktion aktiv ist. Das Intervall wird im Konstruktor gesetzt (s. Zeile ?? in ??). Das Mapping einer Node zu einer Aktion erfolgt durch das Setzen des Parameters `action_name` nach dem Erstellen der Node (s. Zeile ?? in ??).

5.3.3 Control Gripper Aktion

Die Logik Node zur Steuerung des Greifers wird in der Klasse `ControlGripperAction` (s. ??) implementiert. Da es vom OMX kein Feedback gibt, ob oder wann etwas gegriffen wurde, wird die Aktion mit einer fixen Wartezeit implementiert: wenn die Aktion gestartet wird, wird ein Request zur Steuerung des Greifers erzeugt (s. Zeile ?? in ??) und gesendet und nach einer bestimmten Zeit die Aktion beendet. Um unnötige Aufrufe der Methode während des Wartens zu verhindern wird die Wartezeit über das Zeitintervall zur Ausführung der Node gesetzt. Die Aktion wird hierdurch immer beim 2. Aufruf der Methode `do_work` beendet.

Zum Senden des Requests wird ein ROS2 Client für den Service `goal_tool_control` mit dem Typ `open_manipulator_msgs::srv::SetJointPosition` erstellt. Um sowohl die Aktionen zum Öffnen sowie Schließen des Greifers mit einer Node imple-

5 Implementierung

mentieren zu können wird ein Parameter eingeführt, welcher über den Konstruktor gesetzt wird und die Bewegung des Greifers bestimmt. Für jede Aktion wird eine Node mit entsprechenden Parametern erzeugt. Diese Nodes werden alle einem `MultiThreadedExecutor` hinzugefügt. Dadurch können alle Nodes gemeinsam gestartet werden und laufen in einem Prozess, sind durch mehrere Threads aber unabhängig voneinander.

5.3.4 Move Gripper Aktion

Die Logik Node zur Bewegung des OMX wird in der Klasse `MoveGripperAction` (s. ??) implementiert. Insgesamt umfasst dies das Senden des Bewegungsrequests an den vom OMX-Controller erstellten Service `goal_task_space_path_position_only` mit dem Typ `open_manipulator_msgs::srv::SetKinematicsPose` (s. Zeile ?? in??) sowie das Beenden der Aktion wenn der OMX an der Zielposition angekommen ist. Ob die Bewegung beendet ist wird über den im Topic `states` gesendeten Bewegungsstatus geprüft. Dieser kann die beiden Werte `IS_MOVING` und `STOPPED` annehmen. Der aktuelle Status wird mit dem letzten empfangenen verglichen: war der letzte Status `IS_MOVING` und der aktuelle `STOPPED`, waren wir in einer Bewegung, die jetzt beendet ist.

Um die Zielkoordinaten der Bewegung zu erhalten, muss der String Parameter der Aktion in Zahlenwerte konvertiert werden. Zunächst werden aus dem String, welcher im Format `s<Nummer des Stapels>l<Höhe innerhalb eines Stapels>` vorliegt, die 2 Integerwerten mit Hilfe einer regular Expression extrahiert. Diese werden im 2. Schritt über Maps zu Y und Z Koordinaten konvertiert. Die X Koordinate ist im Rahmen dieser Arbeit fest auf den Wert 0,25 gesetzt.

Um eine kollisionsfreie Bewegung sicherzustellen findet diese nicht direkt von der aktuellen zur Zielposition statt. Stattdessen werden mehrere Zwischenpositionen errechnet und entsprechende Bewegungen zuerst ausgeführt. Zunächst wird der OMX bei gleichbleibenden X und Y Koordinaten auf eine Z Koordinate von 0,25 bewegt. Dies entspricht einer Höhe in der keine Blöcke liegen können. Die zweite Zwischenposition bleibt auf der gleichen Höhe (Z Koordinate 0,25) und bewegt sich zur X und Y Koordinate der Zielposition. Diese Zwischenposition befindet sich exakt oberhalb der Zielposition, so dass eine vertikale Bewegung zur Zielposition ohne Kollision möglich ist.

Diese Zwischenpositionen werden übersprungen, wenn die aktuelle Position des OMX bereits dem korrekten Stapel der Zielposition entspricht. Hierbei werden die aktuellen Koordinaten auf 2 Dezimalstellen gerundet.

Für alle benötigten Positionen wird ein Request erzeugt und einer Queue hinzugefügt. Der Auslöser, den nächsten Request zu senden ist jeweils das Ende der vorherigen Bewegung.

Um Zugriff auf die aktuelle Position des OMX zu erhalten wird ein weiterer Subscriber für das Topic `kinematics_pose` benötigt, welches eine Message vom Typ `open_manipulator_msgs::msg::KinematicsPose` veröffentlicht.

Um eine Beschädigung der Motoren zu vermeiden, soll die Bewegung bei einer Kollision abgebrochen werden. Hierfür sind zwei Teile nötig: die Kollisionserkennung sowie

5 Implementierung

das Abbrechen der Bewegung. Für die Kollisionserkennung wird der Stromverbrauch der einzelnen Motoren genutzt. Dieser steigt, wenn der Motor mehr Leistung bringen muss, wenn versucht wird sich gegen ein Hindernis zu bewegen.

Dieser ist in der Message des Topics *joint_states* als *effort* enthalten. Die letzten *x* Werte (konfigurierbar über die Variable *EFFORT_COUNT*, standardmäßig 10) werden für jeden Motor gespeichert (s. Zeile ?? in ??) und ausgewertet (s. Zeile ?? in ??). Die Werte jedes Motors werden einzeln ausgewertet. Es wird das Maximum und das Minimum der Werte genommen und geprüft ob die absolute Differenz größer als der festgelegte Grenzwert (konfigurierbar über die Variable *EFFORT_THRESHOLD*, standardmäßig 500) ist. Ist dies für mindestens einen Motor der Fall gilt die Aktion als fehlgeschlagen und der entsprechende Status wird zurückgegeben (s. Zeile ?? in ??). Eine fehlgeschlagene Aktion resultiert gleichzeitig in einem fehlgeschlagenen Plan. Ein Abbrechen der Aktion verhindert jedoch nicht, dass der OMX versucht das aktuelle Bewegungsziel zu erreichen. Es wird ein neues Bewegungsziel erstellt, das sich an der aktuellen Position befindet.

5.3.5 Package

- pkg build config

Alle erstellten Komponenten werden in einem eigenen ROS2 Package gebündelt. Dies umfasst die C++ Dateien für die Aktionen sowie die Domänendatei. Es wurde eine Launch Datei erstellt um das komplette System über einen Befehl starten zu können. In der Launch Datei wird zum einen PlanSys2 und zum anderen die Nodes für alle Aktionen gestartet. Weiterhin wird hier festgelegt, welche Domäne genutzt wird.

Das Package enthält außerdem eine Readme Datei in der die nötigen Schritte beschrieben sind um das gesamte System, einschließlich OMX, zu starten. Weiterhin wurden mehrere Problemszenarien erstellt, die sowohl als PDDL Datei als auch als Befehle für das PlanSys2 Terminal zur Verfügung stehen.

5.4 Praktische Anwendung

Um das System praktisch anwenden zu können wurden mehrere farbige Würfel aus LEGO gebaut (s. Abbildung ??). Auf der Basisplatte des OMX wurden die Positionen für drei Stapel markiert. Es werden mehrere Szenarien getestet um die allgemeine Anwendbarkeit zu überprüfen.

5.4.1 PlanSys2 Terminal

Das PlanSys2 Terminal ist eine separate Node, welche es ermöglicht, das System durch Eingabe über die Konsole zu steuern. Die wichtigsten Befehle sind hierbei *get*, *set* und *run*. Mit *get* können die Typen, Prädikate und Aktionen der geladenen Domäne sowie des aktuellen Problems angezeigt werden (s. Abbildung 16). Der *set* Befehl wird genutzt, um das aktuelle Problem zu verändern. Es können Instanzen, Prädikate sowie das Ziel erstellt und geändert werden. Über den Befehl *run* lässt sich der Plan für das

5 Implementierung

aktuelle Ziel ausführen. Es ist auch möglich nur einen Plan zu suchen, ohne ihn direkt auszuführen. Dies ist mit `get_plan` möglich (s. Abbildung 17).

Während der Ausführung eines Plans wird im Terminal die aktuelle Aktion sowie dessen Fortschritt (gesteuert über den Feedback Kanal innerhalb der Aktion) angezeigt.

5.4.2 Sussman Anomalie

Die Sussman Anomalie [16] beschreibt eine Schwäche von Planern, bei der bereits ausgeführte Teile bzw. erreichte Unterziele rückgängig gemacht werden müssen um andere bzw. das Gesamtziel zu erreichen. Dieses Problem kann durch drei Blöcke beschrieben werden, die sich Anfangs in den Positionen befinden, die in Abbildung 11 gezeigt wird. Das Ziel ist es, dass *A* auf *B* und *B* auf *C* gestapelt wird, wobei jeweils immer nur ein Block gewegt werden kann. Wird versucht als erstes *A* auf *B* zu erfüllen, wird *C* zur Seite gelegt und *A* auf *B* platziert (s. Abbildung 12). Dieser Schritt muss nun wieder rückgängig gemacht werden um *B* auf *C* zu platzieren.

Wird stattdessen versucht als erstes *B* auf *C* zu erfüllen, so kann dies direkt durch bewegen von *B* erreicht werden (s. Abbildung 13). Um das Ziel *A* auf *B* zu erfüllen muss dieser Schritt nun wieder rückgängig gemacht werden.

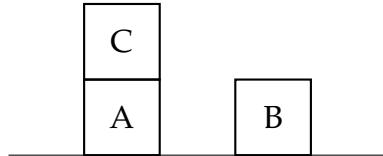


Abbildung 11: Startzustand
der Sussman Anomalie

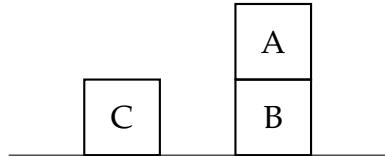


Abbildung 12: Lösung des
Teilziels *A* auf *B*

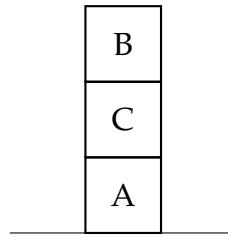


Abbildung 13: Lösung des
Teilziels *B* auf *C*

Um diese Anomalie zu handhaben müssen Planer beide Ziele und deren Schritte kombinieren. POPF generiert für dieses Ziel den in Abbildung 14 gezeigten Plan, der in Abbildung 15 visualisiert wird. Der grün hervorgehobene Block ist jeweils der, der in diesem Schritt bewegt wurde.

5.4.3 Mehr Blöcke, mehr Positionen

5 Implementierung

```

0.000: (move-gripper gripper s1l3 s1l2) [1.000]
1.001: (unstack gripper c a s1l2 s1l1) [0.250]
1.251: (move-gripper gripper s1l2 s3l1) [1.000]
2.251: (place gripper c s3l1 s3) [0.250]
2.501: (move-gripper gripper s3l1 s2l1) [1.000]
3.501: (grab gripper b s2l1 s2) [0.250]
3.751: (move-gripper gripper s2l1 s3l2) [1.000]
4.751: (stack gripper b c s3l2 s3l1) [0.250]
5.001: (move-gripper gripper s3l2 s1l1) [1.000]
6.001: (grab gripper a s1l1 s1) [0.250]
6.251: (move-gripper gripper s1l1 s3l3) [1.000]
7.251: (stack gripper a b s3l3 s3l2) [0.250]
Planner found 1 plan(s) in 1.073secs.

```

Abbildung 14: Durch POPF generierter Plan für die Sussman Anomalie

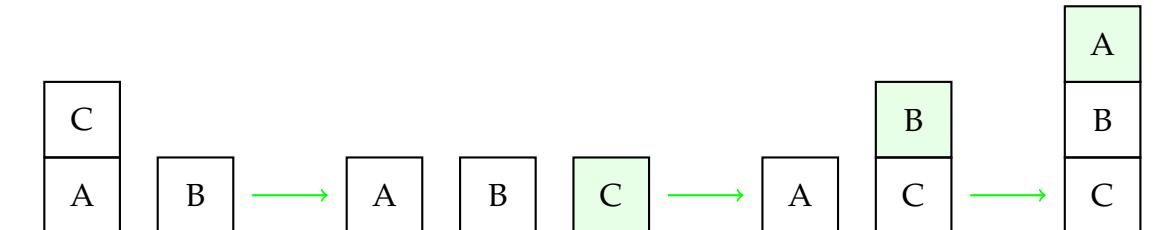


Abbildung 15: Visualisierte Lösung der Sussman Anomalie durch POPF

- evtl sinnvolle Auswertung der maximalwerte inkl. graphen

Die Anzahl der möglichen Blöcke und Positionen soll ausgereizt werden. Bei einem Szenario mit drei Stapeln mit jeweils drei Positionen kann ein maximum von 9 Blöcken platziert werden. Dies verhindert jedoch das Bewegen von Blöcken, da die einzige freie Position zu der ein aufgenommener Block bewegt werden kann, seine Ursprungsposition ist. Damit alle Blöcke bewegt werden können, muss die Anzahl der freien Positionen mindestens der Höhe der Stapel entsprechen. Gibt es bei einer Stapelhöhe von drei und drei Stapeln nur zwei freie Positionen tritt das gleiche Problem für alle untersten Blöcke der Stapel auf: die einzige gültige Position nach dem greifen ist ihre Ausgangsposition (s. Abbildung ??) Um mehr Blöcke verwenden zu können müssen also die Anzahl der Stapel und/oder die maximale Höhe der Stapel erweitert werden.

Für beide Varianten gibt es Einschränkungen, die zu beachten sind. Für höhere Stapel ist der Bewegungsraum des OMX zu beachten. Über dem äußereren Rand der Grund-

5 Implementierung

platte beträgt die maximale Höhe ca 25cm. Da sich der Greifarm mit einem geöffnetem Block frei oberhalb der Stapel bewegen können muss, muss die maximale Höhe eines Stapels unterhalb dieses Werts liegen. Die exakte Maximalhöhe ist von der Größe der Blöcke abhängig.

Für weitere Stapel nehmen wir als Voraussetzung, dass alle Stapel sich nebeneinander befinden sollen. Stapel die sich hinter einem anderen befinden, können vom OMX nicht garantiert frei angesteuert werden. Je näher sich die Stapel am OMX befinden, desto größer ist der Winkel zwischen ihnen. Aufgrund der Größe des Greifers muss der Abstand zwischen mehreren Stapeln umso größer sein, je näher sie sich am OMX befinden. Stapel sollen sich daher am äußeren Rand der Grundplatte befinden.

Da sowohl ein erhöhen der Anzahl der Stapel als auch deren Höhe durch die Größe der Blöcke behindert wird, wurden kleinere Blöcke gebaut. Dies ermöglicht bis zu 5 Stapel mit einer maximalen Höhe von 5 Blöcken. Das theoretische Limit an Blöcken wird damit auf 25, bzw. 20 um Bewegungen zu ermöglichen, erhöht.

Bereits Tests mit Zielen, deren Plan aus nur wenigen Schritten besteht, zeigen, dass dies den Suchraum so stark erweitert, dass Pläne nicht oder erst nach langer Zeit gefunden werden. Ein Testfall mit 20 Blöcken und dem Ziel den obersten Block eines Stapels auf den eines anderen zu legen (optimale Planlänge: 4) führte dazu, dass der Planer ca 13 Minuten lief. Obwohl laut Konsolenausgabe ein Plan gefunden wurde, wird dieser nicht ausgegeben (s. Abbildung ??).

Hier sollten jetzt evtl Graphen und eine Auswertung stehen

Während bei den wenigen, größeren Blöcken eine Namensgebung entsprechend der Farbe ausreichend war, wurden die kleineren Blöcke durchnummieriert. Das PlanSys2 Terminal erlaubt zwar die Erstellung von Objekten und Prädikaten, die diese nutzen, mit rein numerischen Namen, stürzt jedoch beim Versuch ein Ziel mit diesen zu erstellen ohne aussagekräftige Fehlermeldung ab. Ein weiteres Problem, das sich bei der Nutzung der kleinen Blöcke zeigt, ist das diese, aufgrund des geringeren Gewichts, beim Öffnen des Greifers an einer Seite kleben bleiben. Obwohl die Präzision des OMX selbst hoch genug ist, ist die Position der platzierten Blöcke dadurch nicht mehr genau. Durch die Größe des Greifers sowie der Blöcke lässt sich der jeweils unterste Block eines Stapels nicht mehr korrekt greifen. Dieser Block kann zwar normal und sicher bewegt werden, benötigt beim Abbrechen jedoch eine angepasste Position.

5.4.4 Abstürze und Fehlersuche

Während der Ausführung eines Plans kam es wiederholt zu Abstürzen der PlanSys2 Node. Diese Abstürze verhinderten eine vollständige Ausführung des Plans und erforderten einen Neustart des PlanSys2-Systems. Der Absturz erfolgte immer mit einem Exit Code 11, der auf ein SEGMENTATION FAULT hinweist. Um die Stelle des Absturzes zu finden wurde PlanSys2 zunächst in der distributed anstelle der Monolithic Version gestartet. Hierdurch werden alle Nodes in eigenen Prozessen gestartet. Dies ermöglichte ein Eingrenzen auf die Executor Node.

Für das weitere Debugging wurde der GNU Project Debugger (GDB) sowie xterm (muss-

5 Implementierung

te separat installiert werden) genutzt. In der Launch-Datei für den Executor wurde der Node prefix=['xterm -e gdb -ex run --args'] , hinzugefügt, um beim Start der Node auch eine Debugging-Session zu starten.

Über den backtrace nach einem Absturz wurde die Methode

ActionExecutor::request_for_performers() als Ursache für den Absturz identifiziert. Innerhalb dieser Methode wurde das Problem in der Zeile

action_hub_pub_->publish(msg); gefunden. Zum Zeitpunkt eines Absturzes hatte die Variable action_hub_pub_ den Wert null. Um den Absturz zu verhindern wurde ein null-Check implementiert. Obwohl dieser dafür sorgt, dass einige Messages nicht gesendet werden, führte diese Änderung zu keinen offensichtlichen Nebenwirkung und Pläne werden normal ausgeführt.

```
fabian@fabian-VirtualBox:~/colcon_ws2$ ros2 run plansys2_terminal plansys2_terminal
[INFO] [1636970274.917224956] [terminal]: No problem file specified.
ROS2 Planning System console. Type "quit" to finish
> get problem
    Usage:
        get problem [instances|predicates|functions|goal]...
> get model
    Usage:
        get model [types|predicates|functions|actions|predicate|function|action]...
> get model types
Types: 4
    box
    gripper
    location
    stack
> █
```

Abbildung 16: PlanSys2 Terminal mit Abfrage der Typen des geladenen Models

5 Implementierung

```
> set goal (and(box_on white yellow)(box_on blue white))
> get plan
plan:
0      (move-gripper gripper s2l2 s1l2)      1
1.001   (unstack gripper yellow blue s1l2 s1l1) 0.25
1.251   (move-gripper gripper s1l2 s3l1)      1
2.251   (place gripper yellow s3l1 s3)  0.25
2.501   (move-gripper gripper s3l1 s2l1)      1
3.501   (grab gripper white s2l1 s2)    0.25
3.751   (move-gripper gripper s2l1 s3l2)      1
4.751   (stack gripper white yellow s3l2 s3l1) 0.25
5.001   (move-gripper gripper s3l2 s1l1)      1
6.001   (grab gripper blue s1l1 s1)    0.25
6.251   (move-gripper gripper s1l1 s3l3)      1
7.251   (stack gripper blue white s3l3 s3l2)  0.25
```

Abbildung 17: Ausgabe eines Plans in PlanSys2 Terminal

6 Ergebnis

6.1 Zusammenfassung

- off. Unterstützung von Foxy des OMX erst während der Bearbeitung Der Zusammenbau erfolgte ohne größere Probleme. Die beschriebenen Details, die leicht zu übersehen sind, lassen sich auch im Nachhinein ohne großen Aufwand beheben oder korrigieren. Durch die verfügbaren Installations-Skripte gab es auch Softwareseitig keine Hindernisse für einen schnellen Einstieg. Auch wenn zu Beginn dieser Arbeit noch keine offizielle Unterstützung von ROS2 Foxy vorhanden war, konnte anhand der Anleitung für ältere ROS2 Distributionen alles zum laufen gebracht werden. Durch die Kombination der ROS2 Foxy Tutorials sowie der OMX API Dokumentation ließen sich auch ohne Vorkenntnisse gute Fortschritte machen.

6.2 Ausblick

6.2.1 MoveIt

Um den in der dieser Arbeit verwendeten Ansatz der Bewegungsplanung robuster und dynamischer zu machen empfiehlt sich die Implementierung der Bibliothek MoveIt¹⁰. Zum Zeitpunkt der Bearbeitung wird diese vom OMX beim Betrieb mit ROS2 Foxy allerdings noch nicht offiziell unterstützt.

MoveIt bietet mittels virtueller Szenen die Möglichkeit effiziente und kollisionsfreie Wege für die Bewegung des OMX zu planen und auszuführen. In der virtuellen Szene befindet sich sowohl ein Modell des Roboters als auch aller relevanten Hindernisse. Wird eine Bewegung zu einer Position angefordert, kann der generierte Pfad auf potenzielle Kollisionen mit Objekten der Szene überprüft werden.

6.2.2 Vorschläge für Übungen in der Lehre

Um die Verwendung des OMX in der Lehre näher zu bringen gibt es in dem für diese Arbeit entstandenen Package mehrere Stellen, an denen angesetzt werden kann. Grundsätzlich lassen sich diese in folgende Bereiche aufteilen:

- ROS2
- Domäne und Problem in PDDL
- ...

Für jeden dieser Bereiche besteht die Möglichkeit die Aufgaben in die folgenden Richtungen zu führen:

- Fehlersuche mit sinnvoll platzierten Fehlern
- Erkennen und Korrigieren weggelassener Abschnitte, welche durch das Verstehen des Codes erkannt werden können

¹⁰MOVEIT URL

6 Ergebnis

- die Anpassung des Codes an ein ähnliches Szenario

Ein Beispiel für die Domäne ist, den Effekt der MOVE-GRIPPER Aktion, der die Startposition löscht erst am Ende der Aktion auszuführen, was dazu führt, dass zum Start des Plans alle nötigen Bewegungen gleichzeitig ausgeführt werden.

Eine Aufgabe für das Schreiben/ Anpassen eines Problems in PDDL, welche sich anbietet, ist eine vorgegebene Problemdatei, welche nur die grundlegende Struktur über Stapel und Positionen enthält so zu erweitern, dass eine von mehreren möglichen, vorgegebenen Varianten von gestapelten Blöcken beschreibt. Die Aufgabe kann dadurch erleichtert werden, dass das gegebene Problem bereits eine bestimmte Stapelung von Blöcken darstellt und nur angepasst werden muss.

Für alle Aufgaben die PDDL betreffen bietet es sich an, VS-Code mit dem PDDL-Plugin zu nutzen um schnelle Iterationen und eine Visualisierung der Pläne zu ermöglichen.

Literaturverzeichnis

- [1] J. A. Bagnell, F. Cavalcanti, L. Cui, T. Galluzzo, M. Hebert, M. Kazemi, M. Klingensmith, J. Libby, T. Y. Liu, N. Pollard, M. Pivtoraiko, J.-S. Valois und R. Zhu. An integrated system for autonomous robotics manipulation. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Seiten 2955–2962, 2012. DOI: 10.1109/IROS.2012.6385888.
- [2] T. Bylander. The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence*, 69:165–204, 1994.
- [3] M. Cashmore, M. Fox, D. Long, D. Magazzeni, B. Ridder, A. Carrera, N. Palmeras, N. Hurtós und M. Carreras. ROSPlan: Planning in the Robot Operating System. In *ICAPS*, 2015.
- [4] A. Coles, A. Coles, M. Fox und D. Long. Forward-Chaining Partial-Order Planning. In Seiten 42–49, Jan. 2010.
- [5] M. Colledanchise und P. Ögren. Behavior Trees in Robotics and AI: An Introduction. *CoRR*, abs/1709.00084, 2017. arXiv: 1709 . 00084. URL: <http://arxiv.org/abs/1709.00084>.
- [6] C. R. Dyer. Partial Order Planning. <http://pages.cs.wisc.edu/~dyer/cs540/notes/pop.html>, 2003.
- [7] P. Eyerich, R. Mattmüller und G. Röger. *Using the Context-Enhanced Additive Heuristic for Temporal and Numeric Planning*. In *Towards Service Robots for Everyday Environments: Recent Advances in Designing Service Robots for Complex Tasks in Everyday Environments*. E. Prassler, M. Zöllner, R. Bischoff, W. Burgard, R. Haschke, M. Hägele, G. Lawitzky, B. Nebel, P. Plöger und U. Reiser, Herausgeber. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, Seiten 49–64. ISBN: 978-3-642-25116-0. DOI: 10.1007/978-3-642-25116-0_6. URL: https://doi.org/10.1007/978-3-642-25116-0_6.
- [8] R. E. Fikes und N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189–208, 1971. ISSN: 0004-3702. DOI: [https://doi.org/10.1016/0004-3702\(71\)90010-5](https://doi.org/10.1016/0004-3702(71)90010-5). URL: <https://www.sciencedirect.com/science/article/pii/0004370271900105>.
- [9] A. Grastien. Planning (04) Partial-order planning. <http://www.grastien.net/ban/teaching/06-planning4.pdf>, 2006.
- [10] J. Hoffmann und B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *CoRR*, abs/1106.0675, 2011. arXiv: 1106 . 0675. URL: <http://arxiv.org/abs/1106.0675>.

-
- [11] A. Klöckner. Behavior Trees for UAV Mission Management. In M. Horbach, Herausgeber, *INFORMATIK 2013: Informatik angepasst an Mensch, Organisation und Umwelt*, Band P-220 der Reihe *GI-Edition-Lecture Notes in Informatics (LNI) - Proceedings*, Seiten 57–68, Koblenz, Germany. Gesellschaft für Informatik e.V. (GI), Köllen Druck + Verlag GmbH, Bonn, 16-20 September 2013. URL: <http://www.gi.de/service/publikationen/lni/>. ISBN 978-3-88579-614-5.
 - [12] F. Martín, J. Ginés, V. Matellán und F. J. Rodriguez. PlanSys2: A Planning System Framework for ROS2. *CoRR*, abs/2107.00376, 2021. arXiv: 2107.00376. URL: <https://arxiv.org/abs/2107.00376>.
 - [13] A. Marzinotto, M. Colledanchise, C. Smith und P. Ögren. Towards a unified behavior trees framework for robot control. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, Seiten 5420–5427, 2014. DOI: 10.1109/ICRA.2014.6907656.
 - [14] Robotis. OpenMANIPULATOR-X. URL: https://emanual.robotis.com/docs/en/platform/openmanipulator_x/quick_start_guide/#install-ros-on-pc (besucht am 03.11.2021).
 - [15] ROS. Installing ROS 2 on Ubuntu Linux. URL: <https://docs.ros.org/en/foxy/Installation/Ubuntu-Install-Binary.html> (besucht am 03.11.2021).
 - [16] G. J. Sussman. *A Computational Model of Skill Acquisition*. Dissertation, Massachusetts Institute of Technology, 1973. URL: <https://dspace.mit.edu/handle/1721.1/6894>.
 - [17] What is AI Planning? URL: <https://planning.wiki/guide/whatis/aip> (besucht am 13.11.2021).

A Quellcode

A.1 Planungsdomäne

```
1  (define (domain blockworld)
2    (:requirements :strips :typing :adl :fluents :durative-actions)
3
4    ;; Types ;;;;;;;;;;;;;;;
5    (:types
6      box
7      gripper
8      location
9      stack
10   );; end Types ;;;;;;;;;;;
11
12   ;; Predicates ;;;;;;;;;;;
13   (:predicates
14
15     (gripper_at ?g - gripper ?l - location)
16     (box_at ?b - box ?l - location)
17     (box_on ?b_above ?b_below - box)
18     (gripper_open ?g - gripper)
19     (is_holding ?g - gripper ?b - box)
20     (clear ?b - box)
21
22     (stack_empty ?s - stack)
23     (location_above ?l_above ?l_below - location)
24     (is_base_loc ?l - location ?s - stack)
25
26   );; end Predicates ;;;;;;;;;;;
27
28   ;; Functions ;;;;;;;;;;;
29   (:functions
30
31   );; end Functions ;;;;;;;;;;;
32
33   ;; Actions ;;;;;;;;;;;
34
35
36   (:durative-action GRAB
37     (:parameters (
38       ?g - gripper
39       ?b - box
40       ?l - location
41       ?s - stack
42     )
43     (:duration (= ?duration 0.25))
44     (:condition (
45       and
46         (over_all(gripper_at ?g ?l))
47         (at_start(box_at ?b ?l))
48         (at_start(gripper_open ?g))
49         (at_start(clear ?b))
50         (at_start(is_base_loc ?l ?s))
51     )
52   )
```

```

:effect (
  and
    (at end(not (box_at ?b ?1)))
    (at end(not (gripper_open ?g)))
    (at end(is_holding ?g ?b))
    (at end(stack_empty ?s))
    (at end(not(clear ?b)))
)
)

(:durative-action PLACE
 :parameters (
  ?g - gripper
  ?b - box
  ?l - location
  ?s - stack
)
 :duration (= ?duration 0.25)
 :condition (
  and
    (at start(is_holding ?g ?b))
    (over all(gripper_at ?g ?l))
    (at start(is_base_loc ?l ?s))
    (at start(stack_empty ?s))
)
 :effect (
  and
    (at end(box_at ?b ?1))
    (at end(not (is_holding ?g ?b)))
    (at end(gripper_open ?g))
    (at end(clear ?b))
    (at end(not(stack_empty ?s)))
)
)

(:durative-action STACK
 :parameters (
  ?g - gripper
  ?b ?b2 - box
  ?l ?l2 - location
)
 :duration (= ?duration 0.25)
 :condition (
  and
    (at start(clear ?b2))
    (at start(is_holding ?g ?b))
    (over all(gripper_at ?g ?l))
    (at start(location_above ?l ?l2))
    (at start(box_at ?b2 ?l2))
)
 :effect (

```

```

        and
101      (at end(not(clear ?b2)))
102      (at end(box_on ?b ?b2))
103      (at end(clear ?b))
104      (at end(box_at ?b ?l))
105      (at end(not(is_holding ?g ?b)))
106      (at end(gripper_open ?g))
107    )
108  )

(:durative-action UNSTACK
  :parameters (
111    ?g - gripper
112    ?b ?b2 - box
113    ?l ?l2 - location
  )
  :duration (= ?duration 0.25)
  :condition (
    and
      (at start(clear ?b))
      (at start(gripper_open ?g))
      (at start(box_on ?b ?b2))
121      (over all(gripper_at ?g ?l))
      (at start(box_at ?b ?l))
      (at start(box_at ?b2 ?l2))
  )
  :effect (
    and
      (at end(not(clear ?b)))
      (at end(not(box_on ?b ?b2)))
      (at end(clear ?b2))
      (at end(is_holding ?g ?b))
      (at end(not(box_at ?b ?l)))
      (at end(not(gripper_open ?g)))
  )
)

(:durative-action MOVE-GRIPPER
  :parameters (
    ?g - gripper
    ?l_from ?l_to - location
  )
  :duration (= ?duration 1)
  :condition (
    and
      (at start(gripper_at ?g ?l_from))
  )
  :effect (
    and
      (at start(not (gripper_at ?g ?l_from))))
      (at end(gripper_at ?g ?l_to))
  )
)

```

```

151     )
)
); end Domain ;;;;;;;;;;;;;;;;
```

A.2 Gripper Package Launch Datei

```

# Copyright 2019 Intelligent Robotics Lab
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
# limitations under the License.

import os

17 from ament_index_python.packages import get_package_share_directory

from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, IncludeLaunchDescription,
    SetEnvironmentVariable
from launch.launch_description_sources import PythonLaunchDescriptionSource
22 from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node

def generate_launch_description():
    # Get the launch directory
27 example_dir = get_package_share_directory('blockworld_gripper')
namespace = LaunchConfiguration('namespace')

declare_namespace_cmd = DeclareLaunchArgument(
    'namespace',
    default_value='',
    description='Namespace')

32 stdout_linebuf_envvar = SetEnvironmentVariable(
    'RCUTILS_LOGGING_BUFFERED_STREAM', '1')

37 plansys2_cmd = IncludeLaunchDescription(
    PythonLaunchDescriptionSource(os.path.join(
        get_package_share_directory('plansys2_bringup'),
        'launch',
```

```

    'plansys2_bringup_launch_monolithic.py')),
47 launch_arguments={
    'model_file': example_dir + '/pddl/test-domain.pddl',
    'namespace': namespace
}.items()))

# Specify the actions
move_gripper_cmd = Node(
52     package='blockworld_gripper',
     executable='move_gripper_action_node',
     name='move_gripper_action_node',
     namespace=namespace,
     output='screen',
     parameters=[])

57 grab_cmd = Node(
     package='blockworld_gripper',
     executable='grab_action_node',
     name='grab_action_node',
     namespace=namespace,
     output='screen',
     parameters=[])

62 place_cmd = Node(
67     package='blockworld_gripper',
     executable='place_action_node',
     name='place_action_node',
     namespace=namespace,
     output='screen',
     parameters=[])

72 stack_cmd = Node(
77     package='blockworld_gripper',
     executable='stack_action_node',
     name='stack_action_node',
     namespace=namespace,
     output='screen',
     parameters=[])

82 unstack_cmd = Node(
87     package='blockworld_gripper',
     executable='unstack_action_node',
     name='unstack_action_node',
     namespace=namespace,
     output='screen',
     parameters=[])

# Create the launch description and populate
ld = LaunchDescription()

# Set environment variables
ld.add_action(stdout_linebuf_envvar)

```

```

ld.add_action(declare_namespace_cmd)

# Declare the launch options
97 ld.add_action(plansys2_cmd)

ld.add_action(move_gripper_cmd)
ld.add_action(grab_cmd)
ld.add_action(place_cmd)
102 ld.add_action(stack_cmd)
ld.add_action(unstack_cmd)

return ld

```

A.3 Move Gripper Action Node

```

#include <memory>
#include <algorithm>
#include <queue>

5 #include "plansys2_executor/ActionExecutorClient.hpp"

#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"

10 #include "open_manipulator_msgs/msg/open_manipulator_state.hpp"
#include "open_manipulator_msgs/msg/kinematics_pose.hpp"
#include "open_manipulator_msgs/srv/set_kinematics_pose.hpp"

using namespace std::chrono_literals;
15 using std::placeholders::_1;

class MoveGripperAction : public plansys2::ActionExecutorClient {
public:
    MoveGripperAction()
20        : plansys2::ActionExecutorClient("move_gripper", 100ms) {
        isStarted = false;
        isCurrentMovementFinished = false;
        manipulator_state_subscription_ = this->create_subscription<
open_manipulator_msgs::msg::OpenManipulatorState>(
            "states", 10, std::bind(&MoveGripperAction::
manipulator_state_callback, this, _1));
        kinematics_pose_subscription_ = this->create_subscription<
open_manipulator_msgs::msg::KinematicsPose>(
            "kinematics_pose", 10, std::bind(&MoveGripperAction::
kinematics_pose_callback, this, _1));
        kinematicsPoseClient = this->create_client<open_manipulator_msgs::
srv::SetKinematicsPose>(
            "goal_task_space_path_position_only");
        while (!kinematicsPoseClient->wait_for_service(1s)) {
30            if (!rclcpp::ok()) {

```

```

                RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Interrupted
while waiting for the service. Exiting.");
                break;
            }
            RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "KinematicsPose
service not available, waiting again... ");
        }
    }

private:
    void manipulator_state_callback(const open_manipulator_msgs::msg::
OpenManipulatorState::SharedPtr msg) {
        //std :: cout << msg->open_manipulator_moving_state << std :: endl;
        last_moving_state = current_moving_state;
        current_moving_state = msg->open_manipulator_moving_state;
        if(last_moving_state == STATE_MOVING && current_moving_state ==
STATE_STOPPED){
            isCurrentMovementFinished = true;
            queueTasksDone++;
        }
    }
    void kinematics_pose_callback(const open_manipulator_msgs::msg::
KinematicsPose::SharedPtr msg) {
        //std :: cout << &msg << std :: endl;
        kinematicsPose = msg;
    }

void do_work() {

    if (!isStarted) {
        isStarted = true;
        isCurrentMovementFinished = true; //we want to immediately start
the next movement
        auto args = get_arguments();
        int level = -1;

        std :: map<std :: string , int >::const_iterator iterLevel = levelMap .
find(args[2]);
        if (iterLevel != levelMap.end()) {
            std :: cout << iterLevel->first << "->" << iterLevel->second
<< " ";
            level = iterLevel->second;
        }
        int stack = -1;
        std :: map<std :: string , int >::const_iterator iterStack = stackMap .
find(args[2]);
        if (iterStack != levelMap.end()) {
            std :: cout << iterStack->first << "->" << iterStack->second;
            stack = iterStack->second;
        }
    }
}

```

```

    create_movement(stack, level);
    queueLength = requestQueue.size();
    queueTasksDone = 0;
    send_feedback(float(queueTasksDone) / queueLength, "Move started");
}
if(isCurrentMovementFinished){
    if(!requestQueue.empty()){
        auto nextRequest = requestQueue.front();
        requestQueue.pop();
        isCurrentMovementFinished = false;
        auto result = kinematicsPoseClient->async_send_request(
nextRequest);
        send_feedback(float(queueTasksDone) / queueLength, "Move
started");
    }
    else if(isStarted){
        //we did start and no more movement enqueued
        finish(true, 1.0, "Move completed");

        isStarted = false;
        std::cout << std::endl;
    }
}

std::cout << "\r\ue[K" << std::flush;
std::cout << "Moving ... [" << float(queueTasksDone) / queueLength
<< "] " << std::flush;
}

void create_movement(int stack, int level){
    //clear the queue by swapping with an empty one
    std::queue<std::shared_ptr<open_manipulator_msgs::srv::SetKinematicsPose::Request>> emptyQueue;
    std::swap(requestQueue, emptyQueue);
    if(stack == -1 || level == -1 || kinematicsPose == nullptr){
        std::cout << "Error detected when creating movement stack: " <<
stack << " level: " << level << " Pose null: " << (kinematicsPose ==
nullptr);
        return;
    }
    //a position above the current that is clear from collisions
    auto currentPosition = kinematicsPose->pose.position;
    std::cout << "Current (" << currentPosition.x << ", " <<
currentPosition.y << ", " << currentPosition.z << ")" << std::flush;
    std::cout << "Target (" << STACK_POS << ", " << stackPosMap[stack]
<< ", " << heightMap[level] << ")" << std::flush;
    if(roundTo2DecimalPlaces(currentPosition.x) != STACK_POS ||
roundTo2DecimalPlaces(currentPosition.y) != stackPosMap[stack]){

```

```

        auto currentClearRequest = create_request(currentPosition.x,
currentPosition.y, CLEAR_HEIGHT);

115    //a position above the target position that is clear from
collisions
        auto targetClearRequest = create_request(STACK_POS, stackPosMap[
stack], CLEAR_HEIGHT);
        requestQueue.push(currentClearRequest);
        requestQueue.push(targetClearRequest);
    }

120    //the position we want to end up at
    auto targetPointRequest = create_request(STACK_POS, stackPosMap[
stack], heightMap[level]);
    requestQueue.push(targetPointRequest);
}

125    std :: shared_ptr<open_manipulator_msgs :: srv :: SetKinematicsPose :: Request>
create_request(float x, float y, float z) {
    auto request = std :: make_shared<open_manipulator_msgs :: srv :: SetKinematicsPose :: Request>();
    request->end_effector_name = "gripper";
    request->kinematics_pose.pose.position.x = x;
    request->kinematics_pose.pose.position.y = y;
    request->kinematics_pose.pose.position.z = z;
    request->path_time = 2.5;

    return request;
}

135    float roundTo2DecimalPlaces(float input){
        return roundf(input * 100) / 100;
    }

140    rclcpp :: Subscription<open_manipulator_msgs :: msg :: OpenManipulatorState>::
SharedPtr manipulator_state_subscription_;
    rclcpp :: Subscription<open_manipulator_msgs :: msg :: KinematicsPose>::
SharedPtr kinematics_pose_subscription_;
    rclcpp :: Client<open_manipulator_msgs :: srv :: SetKinematicsPose >:: SharedPtr
kinematicsPoseClient;
    std :: shared_ptr<open_manipulator_msgs :: msg :: KinematicsPose>
kinematicsPose;
    std :: queue<std :: shared_ptr<open_manipulator_msgs :: srv :: SetKinematicsPose
:: Request>> requestQueue;
    bool isStarted;
    bool isCurrentMovementFinished;
    int queueLength;
    int queueTasksDone;
    //int level = -1;
    std :: string current_moving_state;
    std :: string last_moving_state;
}

```

```

155     const std::string STATE_MOVING = "IS_MOVING";
156     const std::string STATE_STOPPED = "STOPPED";
157     const float STACK_POS = 0.25;
158     const float CLEAR_HEIGHT = 0.25;
159     std::map<std::string, int> levelMap = {
160         {"s111", 1},
161         {"s211", 1},
162         {"s311", 1},
163         {"s112", 2},
164         {"s212", 2},
165         {"s312", 2},
166         {"s113", 3},
167         {"s213", 3},
168         {"s313", 3},
169     };
170     std::map<std::string, int> stackMap = {
171         {"s111", 1},
172         {"s113", 1},
173         {"s112", 1},
174         {"s211", 2},
175         {"s213", 2},
176         {"s212", 2},
177         {"s312", 3},
178         {"s311", 3},
179         {"s313", 3},
180     };
181     std::map<int, float> heightMap = {
182         {1, 0.04},
183         {2, 0.085},
184         {3, 0.13},
185     };
186     std::map<int, float> stackPosMap = {
187         {1, -0.1},
188         {2, 0.0},
189         {3, 0.1},
190     };
191 };
192
193 int main(int argc, char **argv) {
194     rclcpp::init(argc, argv);
195     auto node = std::make_shared<MoveGripperAction>();
196
197     node->set_parameter(rclcpp::Parameter("action_name", "move-gripper"));
198     node->trigger_transition(lifecycle_msgs::msg::Transition::TRANSITION_CONFIGURE);
199
200     rclcpp::spin(node->get_node_base_interface());
201
202     rclcpp::shutdown();

```

```
    return 0;  
}
```