Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Studienarbeit Nr. 2468

Entwicklung einer Skriptsprache zur interaktiven Robotersteuerung

Robin Böhm

Studiengang: Informatik

Prüfer/in: Prof. Dr. Marc Toussaint

Betreuer/in: M. Sc. Peter Englert

Beginn am: 3. März 2015

Beendet am: 4. September 2015

CR-Nummer: I.2.9, I.2.4

Kurzfassung

In der vorliegenden Arbeit wird eine Skriptsprache zur interaktiven Robotersteuerung entwickelt. Sie ermöglicht die Steuerung des *PR2* mittels eines *Python*-Kommandointerpreters. So kann die Planung von Roboterverhalten übernommen und auf unvorhergesehene Ereignisse eingegangen werden. Die Evaluierung der Nutzbarkeit der entwickelten Skriptsprache geschieht mittels einer Benutzerstudie. Diese kommt zu dem Ergebnis, dass der Roboter durch die Skriptsprache intuitiv steuerbar ist, jedoch auch noch Potential zur weiteren Entwicklung bietet.

Danksagung

Zunächst möchte ich mich an dieser Stelle bei einigen Personen bedanken, welche diese Arbeit ermöglicht haben. Mein herzlichster Dank gilt Prof. Dr. Marc Toussaint, M. Sc. Peter Englert, M. Sc. Stefan Otte, Claudia und Martin Böhm und den Teilnehmern der Benutzerstudie.

Inhaltsverzeichnis

1. Einleitung **9**

1.1. Problemstellung und Motivation 9

1.2. Zielsetzung der Arbeit 9

1.3. Gliederung 10

2. Steuerungsvarianten von Roboterverhalten im Vergleich **11**

2.1. Regelungsarchitekturen der Robotik 11

2.2. Erzeugung von Robotervehalten 14

2.3. Mächtigkeit verschiedener Steuerungsvarianten im Vergleich 21

3. Konzeption einer Skriptsprache zur interaktiven Robotersteuerung **23**

3.1. Anforderungen an eine interaktive Robotersteuerung 23

3.2. Grobkonzeption der Skriptsprache zur interaktiven Robotersteuerung 24

3.3. Umsetzung des entwickelten Konzeptes 26

4. Anwendung und Diskussion der entwickelten Skriptsprache **29**

4.1. Anwendung der entwickelten Skriptsprache 29

4.2. Benutzerstudie 32

4.3. Diskussion und Auswertung der Anwendung der entwickelten Skriptsprache 36

5. Zusammenfassung **39**

A. Anhang 1 - Aufgabenblatt der Benutzerstudie zur Skriptsprache zur intertaktiven Robotersteuerung **41**

Literaturverzeichnis **45**

Abbildungsverzeichnis

2.1. Schema der SPA-Architektur 12
2.2. Schema der verhaltensbasierten Robotik 13
2.3. Schema Hybrider Architekturen 14
2.4. Szenario des Beispiels für *Learning from Demonstration* mit Abstand zur Wand d und Abstand b zum Ziel z. 15
2.5. Prinzip des modellbasierten (1.), modellfreien (2.) und hybriden (3.) *Reinforcement Learning*. Gestrichelte Pfeile sind zu lernende Zustandsübergänge, Fragezeichen zu lernende Trajektorien. 16
2.6. Umsetzung des Bsp. als *Finite State Machine* mit $\{Fahren, Stehenbleiben, Hinlenken, Weglenken\} \in S$ und $\{d = optimal, d > optimal, d < optimal, b = 0\} \in A$. 18
2.7. Beispiel eines *Optionsbaumes* von *XABSL*. *Optionen* sind rechteckig, Zustände des *FSM* rund und Zustandsübergänge als Pfeile dargestellt. Die gestrichelten Pfeile zeigen, wie *Optionen* durch Zustände aktiviert werden können. 19
2.8. Prinzip des Regelkreises eines PD-Reglers 21

3.1. Foto des PR2 24
3.2. Struktur der Implementierung der Skriptsprache 25

4.1. Foto des Settings der Benutzerstudie 33

Tabellenverzeichnis

4.1. Ergebnis der Benutzerstudie 35

Verzeichnis der Algorithmen

3.1. Auszug aus der Headerdatei des C++ Interfaces 27
3.2. Beispiel einer Minimalanwendung der erzeugten Bibliothek 28

4.1. Beispielquelltext für die Bewegung des Roboterautos 31
4.2. Funktion im Plan-Format 32

Abkürzungsverzeichnis

CST Constructing Skill Trees

FSM Finite State Machine

LfD Learning from Demonstration

PR2 Personal Robot 2

RL Reinforcement Learning

RM Relational Machine

SPA Sense Plan Act

SWIG Simplified Wrapper and Interface Generator

XABSL Extensible Agent Behavior Specification Language

1. Einleitung

Roboterverhalten in hochdynamischen Umgebungen zu steuern ist noch immer ein schwieriges Problem im maschinellen Lernen und der Robotik [LRJ06, Vgl. S. 5124]. Selbst einfache Aufgaben, wie das Aufheben eines Objektes von einem Tisch erfordern vielschichtige Entscheidungsvorgänge. Der Agent muss beispielsweise entscheiden, wo er steht, wo er das Objekt greift oder wie viel Kraft er aufwendet. Jedoch gibt es einen Mangel an mächtigen Softwarewerkzeugen, welche eine effiziente und effektive Implementierung von differenziertem Roboterverhalten erlauben [BMTR12, Vgl. S. 1]. Die vorliegende Arbeit versucht ein Programmierwerkzeug bereitzustellen, welches die effiziente Erzeugung komplexen Roboterverhaltens erlaubt.

Das folgende Kapitel beschreibt die Problemstellung (S. Kapitel 1.1), das Ziel (S. Kapitel 1.2) und gibt ein Überblick über die Gliederung dieser Arbeit (S. Kapitel 1.3).

1.1. Problemstellung und Motivation

Diese Arbeit beschäftigt sich mit der Erweiterung der *Relational Machine - RM* [Tou15] um eine interaktive Robotersteuerung. Sie ist ein Framework zur Erzeugung von Roboterverhalten und hat zum Ziel sequenzielles und paralleles Handeln in einer Weise zu repräsentieren, die zu den Formalismen von relationalen Lernalgorithmen passt. Zudem soll ein einfaches aber flexibles Programmiergerüst zum direkten Programmieren von Roboterverhalten zur Verfügung gestellt werden [Tou15, Vgl. S. 1]. Die *RM* ist in *C++*, einer Compilersprache implementiert. Deshalb gibt es bisher keine Möglichkeit zur interaktiven Verhaltenserzeugung, jedoch besteht Bedarf zum schnellen Prototyping und zur interaktiven Robotersteuerung auf Basis einer Skriptsprache.

1.2. Zielsetzung der Arbeit

Das Ziel dieser Arbeit ist die Konzeption und Entwicklung einer Skriptsprache zur interaktiven Robotersteuerung, welche die Funktionalitäten der *Relational Machine* verwendet. Die Arbeit untersucht Anwendungsfelder und die Mächtigkeit der entwickelten Sprache im

Rahmen einer Benutzerstudie und dient als Grundlage zur weiteren Forschung an diesem Projekt.

1.3. Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Steuerungsvarianten von Roboterverhalten im Vergleich stellt verschiedene Regelungsarchitekturen und Algorithmen zur Verhaltenserzeugung der Robotik vor, vergleicht diese und geht auf deren Mächtigkeit ein.

Kapitel 3 – Konzeption einer Skriptsprache zur interaktiven Robotersteuerung konzipiert eine Skriptsprache zur interaktiven Robotersteuerung und erläutert das Prinzip der entwickelten Sprache.

Kapitel 4 – Anwendung und Diskussion der entwickelten Skriptsprache zeigt die Anwendung der entwickelten Sprache, reflektiert diese im Rahmen einer Benutzerstudie und diskutiert die Ergebnisse.

Kapitel 5 – Zusammenfassung

2. Steuerungsvarianten von Roboterverhalten im Vergleich

Die komplexe Aufgabe der Steuerung von Servicerobotern zwingt zur Nutzung geeigneter Systemarchitektur. Diese soll steigender Hard- und Softwarekomplexität hinsichtlich der Echtzeitanforderung und Transparenz gerecht werden und dynamischen Ansprüchen genügen [Mil10, Vgl. S. 25]. Entsprechend haben sich in der Robotik verschiedene Regelungsparadigmen etabliert. Zudem bedingen die Art der Repräsentation des Roboterverhaltens und der Welt direkt die Anwendungsmöglichkeiten und Mächtigkeit des Roboters. So hat, im Gegensatz zur künstlichen Intelligenz, die Robotik ständig Umgang mit der physikalischen Welt und ihren kontinuierlichen Größen, jedoch erfordert eine Repräsentation bzw. Modellbildung eine Diskretisierung dieser. Bis heute ist es eine große, bisher ungelöste Aufgabe in der Robotik die fundamentalen Eigenschaften natürlicher Welten abstrakt zu repräsentieren [TLJ13, Vgl. S. 1f].

Das vorliegende Kapitel beschäftigt sich mit verschiedenen Steuerungsvarianten der Robotik. Zu Beginn werden verschiedene Regelungsarchitekturen und grundlegende Paradigmen der Robotik vorgestellt (S. Kapitel 2.1). Davon ausgehend werden verschiedene Varianten der Verhaltenserzeugung der Robotik und die Funktionalität der *Relational Machine* beschrieben (S. Kapitel 2.2). Schließlich wird die Mächtigkeit verschiedener Steuerungsvarianten verglichen (S. Kapitel 2.3).

2.1. Regelungsarchitekturen der Robotik

Unter einer Regelungsarchitektur, dem Regelungsparadigma wird das einer Robotersteuerung zu Grunde liegende Prinzip verstanden. Es stellt sich die Frage nach der Modellierung der Welt insofern, ob ein reaktiver, modellfreier Ansatz schlicht auf der Basis der Sensorik gewählt wird, oder ob die Sensorinformationen in ein Weltmodell einfließen sollen, auf dessen Grundlage dann die Entscheidungen getroffen werden.

Dieses Kapitel gibt einen Überblick über drei wichtige dieser Architekturen. Beginnend mit der modellbasierten Robotik (S. Kapitel 2.1.1) und der verhaltensbasierten Robotik (S. Kapitel 2.1.2) wird das Paradigma einer hybriden Architektur beschrieben, welche die Vorteile der anderen Architekturen ausnutzt (S. Kapitel 2.1.3).

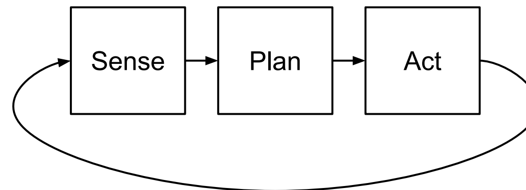


Abbildung 2.1.: Schema der SPA-Architektur

2.1.1. Modellbasierte Robotik

Die ersten für die Robotik entwickelten Architekturen basierten auf dem *SPA - sense plan act* Paradigma, der *modellbasierten Robotik*. *SPA* bedeutet, dass die Sensorinformationen (*sense*) in ein Modell der Welt als Grundlage für die Planung (*plan*) einfließen. Abhängig der Sensorinformationen wird ein Plan erstellt oder ausgewählt. Dieser Plan wird daraufhin in einem unidirektionalen Fluss an die Aktuatoren weitergeleitet (*act*) (S. Abbildung 2.1).

Schnell wurde klar, dass eine Echtzeitplanung mit dieser Architektur viel zu zeitaufwändig ist und ein offener Regelkreis in einer dynamischen Umwelt nicht zuverlässig funktionieren kann. Zudem ermöglicht sie keine Hierarchisierung von Verhaltensstrategien [Mil10, Vgl. S. 25] .

2.1.2. Verhaltensbasierte Robotik

Die *verhaltensbasierte Robotik* steht im Gegensatz zur rechenaufwändigen *SPA* Architektur. Sie ist reaktiv. Das bedeutet, es gibt keinen Bedarf ein Modell der Welt zu erstellen. Der Roboter braucht beispielsweise weder ein Modell von einem Stuhl noch von dem Untergrund auf welchem er sich bewegt um damit interagieren zu können [Bro91, Vgl. S. 141]. Vielmehr wird die Roboterbewegung ständig durch eine direkte Verbindung der Sensorik mit den Aktuatoren angepasst. Solche Regeleinheiten können kompetitiv, hierarchisch oder zustandsbasiert angeordnet sein, um komplexeres Verhalten realisieren zu können [LRJ06, Vgl. S. 5124] (S. Abbildung 2.2).

Es werden mit diesem Ansatz eindrucksvolle Verhaltensweisen erreicht, jedoch erweist es sich als problematisch langfristige Ziele zu erreichen und Roboterverhalten zu optimieren [Mil10, Vgl. S. 25f].

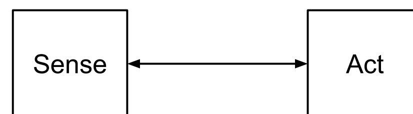


Abbildung 2.2.: Schema der verhaltensbasierten Robotik

2.1.3. Hybride Architekturen

Um dem Problem der mangelnden Möglichkeit der automatischen mittel- und langfristigen Planung der *verhaltensbasierten Robotik* zu begegnen und trotzdem reaktiv und zeiteffizient bleiben zu können, wurden, *hybride Architekturen* entwickelt. Dadurch wird die Rolle des Planers minimiert, indem automatische Selektionsmechanismen die optimale Strategie dem Zustand der Welt entsprechend auswählen [LRJ06, Vgl. S. 5124].

Auf der untersten Ebene, der Regelungsebene, sind die Regelkreise realisiert, welche direkt mit Sensoren und Aktuatoren verbunden sind (S. Kapitel 2.1.2). Diese verarbeiten die Daten der Sensorik in Echtzeit und regeln die Effektoren in einem kurzen Zeithorizont. Auf der nächsten Ebene, der Durchführungsebene, liegt die langfristige Planung in kurzen Planungseinheiten vor. Für diese werden dynamisch, abhängig von den Sensordaten der Regelungsebene, die richtigen Regelungsstrategien ausgewählt und an die Regelungsebene weitergeleitet. Die Erfüllung der Aufgabe wird ständig überwacht um auf Konflikte effizient reagieren zu können. Die oberste Ebene stellt die Planungsebene dar. Hier werden die langfristigen Ziele in Teilaufgaben unterteilt und an die Durchführungsebene übermittelt, die Beschränkungen der Aufgaben berücksichtigt und Ressourcen entsprechend verteilt.

In der Praxis werden jedoch oft die oberen Ebenen in eine hybride Ebene vereint um die Kommunikation zwischen den Abstraktionsstufen zu verbessern. So findet man neben dreischichtigen auch häufig zweischichtige Architekturen [Mil10, Vgl. S. 26f] (S. Abbildung 2.3).

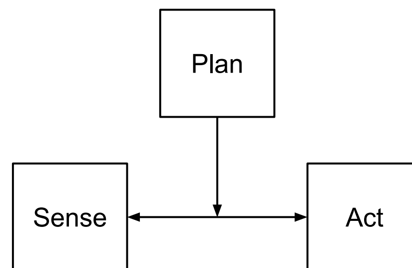


Abbildung 2.3.: Schema Hybrider Architekturen

2.2. Erzeugung von Robotervehalten

Im Gegensatz zu Industrierobotern brauchen Serviceroboter mehr als nur ein festes Repertoire an Fähigkeiten. Sie müssen in der Lage sein ihre Strategien an eine dynamische Welt anzupassen und neue Fähigkeiten zu erlernen. So haben sich neben dem direkten Programmieren von Roboterverhalten auch verschiedene Methoden des maschinellen Lernens in der Robotik etabliert.

Dieses Kapitel beschäftigt sich mit verschiedenen Ansätzen der Roboterprogrammierung und entsprechender Repräsentation des Verhaltens. Zu Beginn werden selbstlernende Algorithmen (S. Kapitel 2.2.1), dann wird direktes Programmieren von Roboterverhalten im Prinzip einer *Finite State Machine* vorgestellt (S. Kapitel 2.2.2). Schließlich wir die Funktionalität der *Relational Machine* erläutert, welche als Grundlage dieser Arbeit dient (S. Kapitel 2.2.3).

2.2.1. Verhaltenserzeugung durch lernende Algorithmen

Das direkte Programmieren von Roboterverhalten ist zeitaufwändig und wenig intuitiv. Auch sind die Anforderungen an Roboterverhalten teilweise so komplex, dass diese nicht manuell umgesetzt werden können. An dieser Stelle greift die Erzeugung von Roboterverhalten durch lernende Algorithmen.

Learning from Demonstration

Beim Lernen durch Demonstration, dem *LfD - Learning from Demonstration* werden dem Agent Bewegungen durch einen Demonstrator vorgemacht. Beispielsweise „zeigt“ der Demonstrator dem Roboter die Bewegung durch Teleoperation oder indem er die Effektoren manuell bewegt. Der Agent erzeugt aus der Aufzeichnung einer einzelnen oder mehreren

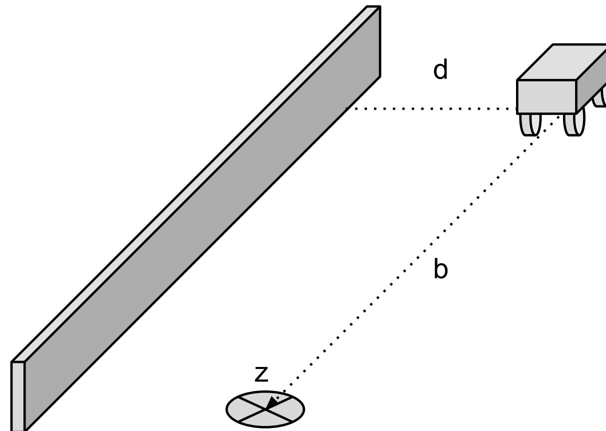


Abbildung 2.4.: Szenario des Beispiels für *Learning from Demonstration* mit Abstand zur Wand d und Abstand b zum Ziel z.

Bewegungen eine Strategie, die aus verschiedenen Ausgangszuständen s zum gleichen Zielzustand z führen soll. Ein Ansatz dies zu erreichen ist die Approximation einer Funktion, welche jeden Zustand $s \in S$ auf ein Verhalten $a \in A$ abbildet $f() : S \Rightarrow A$. Hieraus wird ein Zustandsübergangsmodell T der Form $T(z|s, a)$ erzeugt aus dem der Agent sein Verhalten ableitet [ACVB09, Vgl. S. 470f].

So erzeugte Strategien sind reaktiv (S. Kapitel 2.1.2), d.h. sie kommen ohne Modellbildung aus. Komplexes Roboterverhalten, welches aus vielen Teilbewegungen, welche nacheinander ausgeführt werden sollen, besteht, oder welches aus den dynamischen Eigenschaften der Welt abgeleitet wird, ist schwer durch *LfD* zu erzeugen. Zudem liegen trivialerweise die Grenzen innerhalb dessen, was der Demonstrator vormachen kann. Ein weiteres Problem ist die Approximation von Funktionen aus vieldimensionalen Systemen, jedoch ist der Algorithmus *Constructing Skill Trees* ein vielversprechender Lösungsansatz (S. Kapitel 2.2.1).

Um einem Roboterauto das Fahren entlang einer Wand mit konstantem Abstand d zu einem Ziel vorzumachen, würde das Auto entlang verschiedener Wände manuell entlanggefahren und immer am Zielpunkt z gestoppt werden. Die hieraus approximierte Strategie könnte eine Funktion sein, welche die Lenkbewegung direkt aus dem Abstand d zur Wand ableitet, dass diese immer gleich bleibt. Die Geschwindigkeit der Autos wäre proportional zum Abstand b zwischen Auto und Ziel z. So würde der Agent für jedes s Zielzustand z erreichen. (S. Abbildung 2.4).

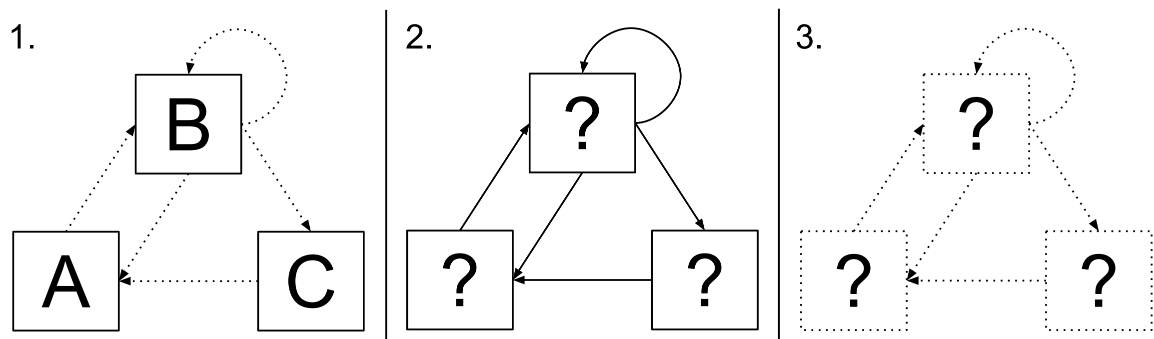


Abbildung 2.5.: Prinzip des modellbasierten (1.), modellfreien (2.) und hybriden (3.) *Reinforcement Learning*. Gestrichelte Pfeile sind zu lernende Zustandsübergänge, Fragezeichen zu lernende Trajektorien.

Reinforcement Learning

Das bestärkende Lernen, das *RL - Reinforcement Learning* folgt dem „trial and error“ Prinzip. In einem dynamischen System wählt der Agent zu jedem Zustand eine Strategie und erhält eine Belohnung.

Im modellbasierten Ansatz des *RL* (S. 1. in Abbildung 2.5) ist ein Zustandsübergangsmodell $T(s'|s,a)$ und eine Belohnungsfunktion $R(s)$ mit dem Belohnungswert r gegeben, aus welchen der Agent eine Strategie $\pi : S \Rightarrow A$ ableitet. Er testet zu jedem Zustand s verschiedene Verhalten a und erhält eine Belohnung r. Schließlich wählt der Agent seine Strategie, so dass sie den erwarteten Gewinn langfristig maximiert [KBP13, Vgl. S. 1]. Die Belohnungsfunktion kann so formuliert werden, dass beispielsweise eine Kostenfunktion optimiert wird, welche die Energie minimiert oder die Ausführungsgeschwindigkeit maximiert.

Das aus dem *RL* erzeugte Verhalten kann zum einen deterministisch sein, so dass zu jedem Zustand genau eine Strategie abgeleitet wird, welche den erwarteten Gewinn maximiert. Zum anderen existieren aber auch nichtdeterministische oder gemischte Methoden, so dass die ausgewählte Aktion als Wahrscheinlichkeitsfunktion $P(a|s)$ definiert ist [Mat07, Vgl. S. 258].

Auf der anderen Seite steht der modellfreie, reaktive Ansatz (S. 2. in Abbildung 2.5). Dieser approximiert, ähnlich wie bei *Lfd* (S. Kapitel 2.2.1) Trajektorien, also direkte Sensor-Aktuator-Verbindungen in einer Weise, dass der erwartete Gewinn maximiert wird. So wird ein Zustandsübergangsmodell der Form $T(s'|s,a)$ mit $R(S)$ maximal erzeugt.

Das Lernen eines Zustandsübergangsmodells, also die automatische Unterteilung einer Aufgabe in Teilzustände, welche so differenziert gewählt sind, das sie als Grundlage zum modellbasierten *RL* dienen können, ist nach Grollman bis heute nicht möglich. Deshalb ist

es eine noch ungelöste Aufgabe eine hybride Architektur (S. 3. in Abbildung 2.5), also eine vollwertige *Finite State Machine* (S. Kapitel 2.2.2) mittels *RL* zu erzeugen. [GJ10, Vgl. S.22].

Soll ein Roboterauto mittels modellfreiem *RL* an einer Wand entlang fahren, könnte die Belohnung maximal sein, wenn der Abstand zur Wand gleich bleibt und der Abstand zum Zielpunkt sich verringert. Zum Maximieren des erwarteten Gewinns würde sich ein Verhalten entwickeln, welchen das Auto an einer Wand entlangfahren lässt und am Ziel z stehenbleibt (S. Abbildung 2.4).

Costructing Skill Trees

Beim maschinellen Lernen gibt es den sogenannten „curse of dimensionality“. So sind in einfachen, wenigdimensionalen Systemen Funktionen leicht aus Punktwolken approximierbar, jedoch haben Agent und seine diskretisierte Welt häufig sehr viele Freiheitsgrade zur Verhaltensspezifizierung. Dieses Problem der Freiheitsgrade löst der Algorithmus *CST - Constructing Skill Trees*. Er segmentiert Demonstrationstrajektorien in Ketten von Verhalten, den *Skills*, und hierarchisiert diese in einem Baum, dem *Skill Tree* [KKGB10, Vgl. S. 1].

Anstatt aus einer Trajektorie ein einzelnes Verhalten zu erzeugen, wird eine Kette von *Skills* erzeugt, so dass jeder *Skill* n das Ziel hat, dass der folgende *Skill* $n + 1$ erfolgreich ausgeführt werden kann. Eine Trajektorie kann beispielsweise durch die Erkennung von Wendepunkten segmentiert werden. Ein so erzeugter *Skill* kann optimiert und abstrahiert werden, so dass er nur von wenigen Parametern abhängt. Das ermöglicht effizientes Lernen und eine effizientere Repräsentation. Ein komplexer, vieldimensionaler Bewegungsablauf besteht so aus Einzelsegmenten mit jeweils wenig Zustandsvariablen. Schließlich werden Ketten verschiedener Trajektorien zu einem *Skill Tree* zusammengefügt. In diesem sind *Skills*, welche zum selben Zustand führen miteinander verknüpft mit dem Zielzustand der Bewegung als Wurzel. Der Agent wählt zu seinem Zustand eine Strategie, in dem er am ehesten passende *Skills* per Wahrscheinlichkeitsfunktion auswählt, um zum Zielzustand zu gelangen [KKGB11, Vgl. S. 1ff].

2.2.2. Verhaltenserzeugung durch direkte Programmierung

Ein grundlegendes Problem lernender Algorithmen ist die Skalierbarkeit. Erzielt man bei Aufgaben, welche eine Folge einfacher Bewegungsabläufe sind, gute Ergebnisse, so muss noch gezeigt werden, wie sich diese generalisieren lassen. Um jedoch die Rolle des Programmierers zu minimieren, sollten Selektionsmechanismen die Auswahl des richtigen Verhaltens übernehmen [LRJ06, Vgl. S. 1]. So erfordert die Umsetzung hybrider Architekturen (S. Kapitel 2.1.3) bis heute eine direkte Programmierung. Zudem ist es oft schlicht effizienter einfache Verhaltensweisen direkt zu implementieren.

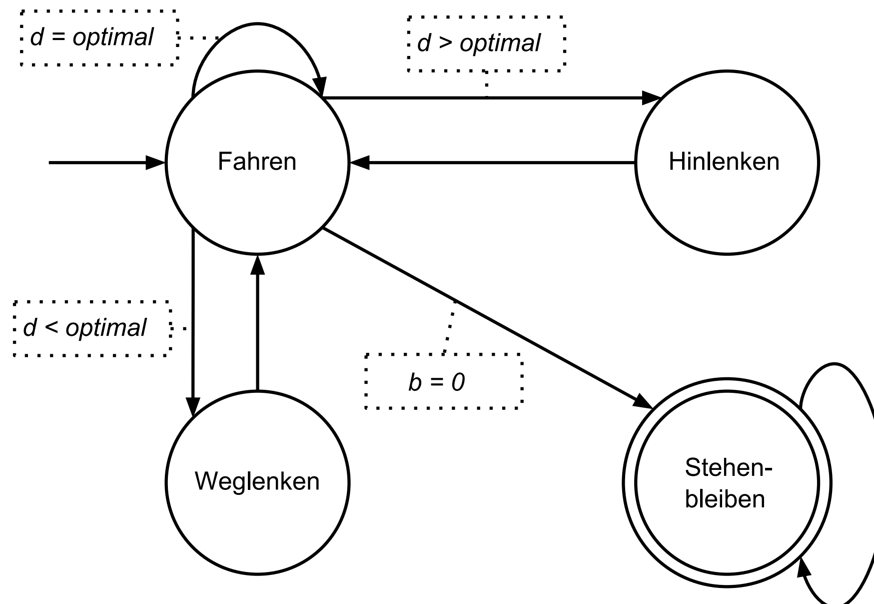


Abbildung 2.6.: Umsetzung des Bsp. als *Finite State Machine* mit $\{Fahren, Stehenbleiben, Hinlenken, Weglenken\} \in S$ und $\{d = optimal, d > optimal, d < optimal, b = 0\} \in A$.

Finite State Machine

Direktes Programmieren von Roboterverhalten kann im Prinzip des endlichen Automaten, der *FSM - final state machine* geschehen. Eine *FSM* ist ein Zustandsübergangsmodell $T(s'|s)$ bzw. $T(s'|s, a)$ hierarchisch strukturierter Zustände und besteht aus einer endlichen Anzahl von Zuständen $s \in S$, einem Startzustand $s0 \in S$, einem Endzustand $z \in S$ und Zustandsübergängen A. Beginnend mit $s0$ werden durch Aktionen $a \in A$ oder externe Trigger andere Zustände erreicht. Ist z erreicht, gilt die Aufgabe als erfolgreich ausgeführt (S. Option A in Abbildung 2.7).

Entsprechend der Sensorik werden die Zustände abgeleitet, aus welchen dann entweder deterministisch $\pi : S \Rightarrow A$ oder probabilistisch $P(a|s)$ eine Strategie, welche den Zielzustand erreicht, abgeleitet wird.

Soll das Entlangfahren eines Roboterautos an einer Wand mit Abstand $d = optimal$ per *FSM* erreicht werden, würde die Zustandsmenge S aus $Fahren$, $Stehenbleiben$, $Hinlenken$ und $Weglenken$ bestehen. Die Sensorik löst die Zustandsübergänge A entsprechend dem Abstand d zur Wand und Abstand b zum Ziel z aus (S. Abbildung 2.6 und Abbildung 2.4).

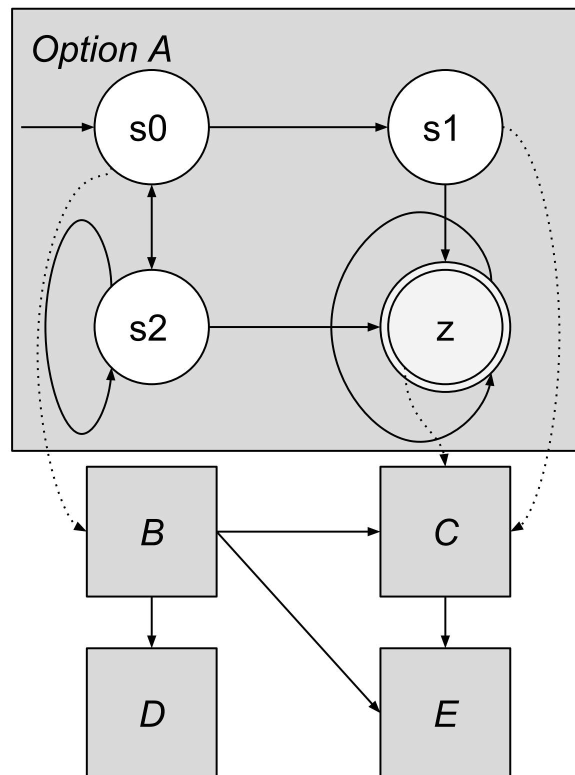


Abbildung 2.7.: Beispiel eines *Optionsbaumes* von *XABSL*. *Optionen* sind rechteckig, Zustände des *FSM* rund und Zustandsübergänge als Pfeile dargestellt. Die gestrichelten Pfeile zeigen, wie *Optionen* durch Zustände aktiviert werden können.

XABSL als Implemetierung hierarchischer FSM

XABSL - Extensible Agent Behavior Language ist eine Programmiersprache zur Implementierung hierarchischer *FSM*. *XABSL* hierarchisiert *FSM* wie folgt:

- Verhaltenselemente heißen in *XABSL Optionen*. Diese sind in einem gerichteten, kreisfreien Graphen, dem *Optionsbaum* angeordnet.
- Jede *Option* ist selbst eine *FSM* und enthält eine endliche Menge an Zuständen mit einem Start- und mindestens einem Zielzustand.
- Jeder Zustand ist durch zwei Teile definiert. Der erste ist die *Aktion*, die in dem Zustand ausgeführt wird. In ihr können andere Optionen aktiviert und auf Symbole zugegriffen werden. Der zweite Teil ist die Entscheidung. Sie definiert die Übergänge zu anderen Zuständen mit Hilfe eines Entscheidungsbaums.

- Die Kommunikation mit anderer, auf dem Roboter laufender Software geschieht mittels Symbolen (S. Abbildung 2.7) [LRJ06, Vgl. S. 1f].

2.2.3. Verhaltenserzeugung durch die Relational Machine

Die sich unter der Leitung von Toussaint in Entwicklung befindende *RM - Relational Machine* ist ein Framework zur Robotersteuerung. Sie hat zum Ziel, einerseits mit den Formalismen von relationalen Lernalgorithmen zusammenzupassen und andererseits ein einfaches aber flexibles Programmiergerüst zum manuellen Programmieren zu sein. Sie ist zentral organisiert und wird durch drei Dinge spezifiziert:

Relationaler Zustand
: Er ist eine Menge an Literalen, welche die laufenden Ereignisse repräsentiert. Diese sind beispielsweise Aktivitäten, Sensorinformationen oder Terminierungskriterien.

Menge an Symbolen
: Sie beschreibt, welche Objekte sich momentan in der Roboterwelt befinden. Sie beschreibt neben aktivitätsabhängigen Prädikaten auch Zustandssymbole oder Terminierungskriterien.

Menge an Regeln
: Sie beschreibt einen Disjunktionsterm erster Ordnung oder probabilistische relationale Regeln für Zustandsübergänge. Beispielsweise beschreibt eine Regel $P(a|s)$ die Ausführungswahrscheinlichkeit der Aktivität a unter dem Zustand s. Genauso können Regeln Zustandsübergänge im Prinzip von $P(s'|s, a)$ oder $P(s'|s)$ für externe Trigger definieren [Tou15, Vgl. S. 1f].

Soll das Entlangfahren eines Autos mittels *RM* realisiert werden, wären Ziel, Wand und Auto in der Menge der Symbole. Der *relationale Zustand* bestünde aus einem Positionscontroller zwischen Ziel und Auto, welcher mit einer geringen Toleranz terminiert und einem Positionscontroller zwischen Wand und Auto mit einem Offset, dem Abstand, welche beide einem PD-Regler folgen.

Der PD-Regler

Die eigentliche Roboterbewegung wird mittels eines *PD-Reglers* erzeugt. Unter einem *PD-Regler*, dem *proportional-derivative-controller* wird ein Regelkreis verstanden, welcher aus Soll- und Istwert der Regelabweichung den neuen Stellwert berechnet und besteht aus einem Proportional- und einem Differentialanteil. Er regelt die Bewegung des Roboters. Der P-Anteil regelt die Bewegung proportional der Abweichung vom Ziel, jedoch reagiert der unmittelbar. Der D-Anteil bewertet die Änderung der Regelabweichung, er differenziert diese

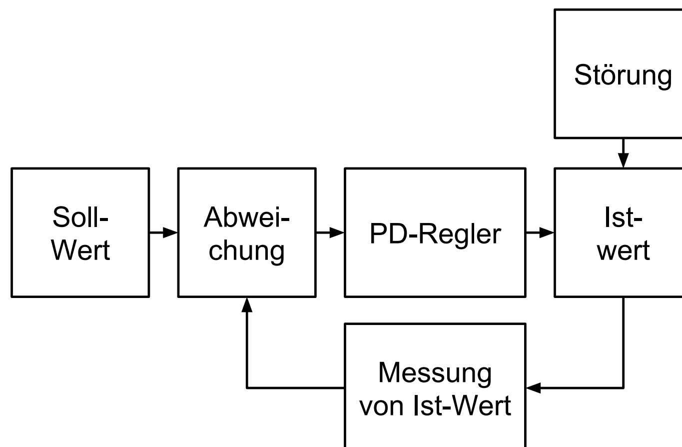


Abbildung 2.8.: Prinzip des Regelkreises eines PD-Reglers

und berechnet so die Änderungsgeschwindigkeit. Hierdurch wird der P-Anteil geglättet und schwingendes Verhalten verhindert (S. Abbildung 2.8) [LL09, Vgl. S. 763f]).

2.3. Mächtigkeit verschiedener Steuerungsvarianten im Vergleich

Ein einfaches Roboterverhalten lässt sich mit verschiedenen Methoden realisieren, wie sich an dem Beispiel des Roboterauto sehen lässt. Jedoch spielt sie Auswahl der Steuerungsarchitektur bei der Implementierung komplexen Verhaltens eine wichtige Rolle.

2.3.1. Vergleich von modellfreier und modellbasierter Verhaltenserzeugung

Auf der einen Seite steht die Erzeugung von Verhalten, welches sich aus Bewegungscontrollern ergibt, der modellfreie Ansatz. Komplexe Bewegungen sind schwer durch direktes Programmieren realisierbar, da diese von vielen Parametern abhängen. Solche Bewegungen können häufig nur durch lernende Algorithmen erreicht werden, jedoch ist das Ergebnis häufig nur schwer vorhersehbar.

Auf der anderen Seite steht die Erzeugung von planendem Verhalten, den Agent selber entscheiden lassen, was zu tun ist. Dies geschieht auf Basis eines Modells. Dieses eigenständig zu erlernen ist für Agent und Programmierer eine anspruchsvolle Aufgabe. Das *RL* ist ein vielversprechender Ansatz, jedoch fordert es ein Modell der Welt, welches vorher manuell

implementiert werden muss. So ist es bis heute notwendig, dass manuelle Programmierarbeit geleistet wird um planendes Verhalten zu erzeugen. Meist geschieht das im Pinzip einer *FSM*.

2.3.2. Einordnung der Relational Machine

Die *Relational Machine* stellt im Kern eine *FSM* dar. Zudem können einzelne Controller kompetitiv, also parallel laufen. Deshalb lässt sich einfach vielschichtiges Verhalten erzeugen, was jedoch mit steigender Komplexität zunehmend schwer vorhersehbar wird. Durch ein flexibles und einheitliches Interface (S. Kapitel 4.1.1) zur Manipulation des *relationalen Zustandes* können lernende Algorithmen und andere Frameworks die Funktionalität der *RM* nutzen und diese erweitern. Durch die Menge an Regeln der *RM* lässt sich ähnlich wie bei *XABSL* eine Hierarchisierung erreichen. Bei Vorhandensein eines Zustandsübergangsmodells kann die *RM* Verhalten simulieren und so eine Strategie planen. Gleichzeitig kann sie in Echtzeit auf Sensorinformationen reagieren und implementiert so im Prinzip eine *hybride Architektur* (S. Kapitel 2.1.3).

3. Konzeption einer Skriptsprache zur interaktiven Robotersteuerung

Auf der Grundlage der *Relational Machine* wird im Rahmen dieser Arbeit eine Skriptsprache implementiert. Sie soll die Möglichkeit einer einfachen aber flexiblen interaktiven Robotersteuerung mittels einem Kommandointerpreter bieten. Das folgende Kapitel stellt die Konzeption und Umsetzung der Skriptsprache dar. Zu Beginn (S. Kapitel 3.1) werden die Anforderungen an solch eine Steuerung herausgearbeitet. Dann wird aus den Anforderungen ein Konzept erstellt (S. Kapitel 3.2), welches daraufhin umgesetzt wird (S. Kapitel 3.3).

3.1. Anforderungen an eine interaktive Robotersteuerung

Dieses Kapitel gibt einen kurzen Überblick über die Anwendung der *RM* auf den Roboter *PR2*, stellt den Handlungsbedarf einer interaktiven Skriptsprache heraus (S. Kapitel 3.1.1) und erörtert schließlich die Anforderungen an solch eine Sprache (S. Kapitel 3.1.2).

3.1.1. Motivation

Als Forschungsgrundlage dieser Arbeit wird die *RM* auf den Serviceroboter *PR2 - Personal Robot 2* von *Willow Garage* angewendet. Er ist etwa menschengroß, verfügt über zwei Arme mit jeweils sieben Freiheitsgraden und bewegt sich auf Rädern. Neben Laserscannern verfügt er auch über Stereokameras, eine Kinect und verschiedene Sensoren um seine Umwelt wahrzunehmen (S. Abbildung 3.1) (Vgl. [Gar]).

Da die *RM* in *C++* implementiert ist und da *C++* eine Compilersprache ist, gibt es keine Möglichkeit zur interaktiven Robotersteuerung. Jedoch existieren einige Anwendungsgebiete, welche schnelles, interaktives Skripten von Roboterverhalten erfordern. Neben Prototyping kann mittels einer interaktiven Skriptsprache spontan auf eine dynamische Umwelt reagiert werden. Mit einer interaktiven Robotersteuerung kann der Entscheidungsprozess des Roboters übernommen und auf unvorhersehbare Ereignisse individuell reagiert werden.

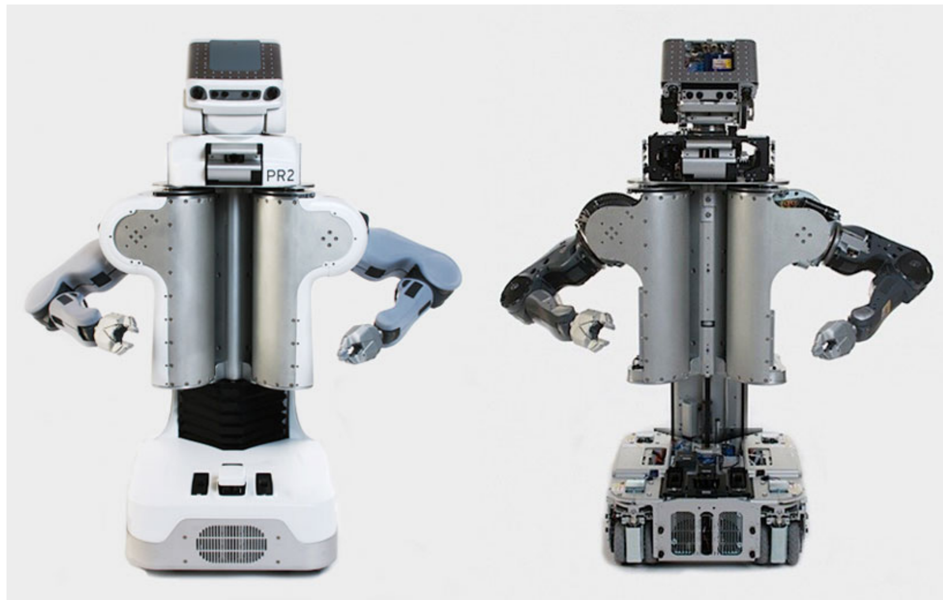


Abbildung 3.1.: Foto des PR2

3.1.2. Anforderungen

Aus dem Handlungsbedarf erschließt sich die Implementierung einer Skriptsprache zur interaktiven Robotersteuerung. Da sich die *RM* noch in Entwicklung befindet, sollte die zu entwickelnde Skriptsprache in der Lage sein Änderungen der *RM* zu übernehmen, ohne dass Anpassungen vorgenommen werden müssen, d.h., dass die Schnittstelle zwischen der *RM* und der Skriptsprache gehalten werden muss. Zudem sollte sie leicht verständlich sein und eine konsistente Syntax bereitstellen. Sie sollte neben der Möglichkeit zum interaktiven Steuern auch eine Funktionalität zur Skripterstellung für das Prototyping bieten.

3.2. Grobkonzeption der Skriptsprache zur interaktiven Robotersteuerung

In diesem Kapitel wird eine Skriptsprache zur interaktiven Robotersteuerung konzipiert. Zuerst werden die Kommunikationsebenen zwischen der *RM* und dem Kommandointerpreter, (S. Kapitel 3.2.1) dann die Struktur der verwendeten Schnittstelle der *RM* (S. Kapitel 3.2.2) beschrieben.

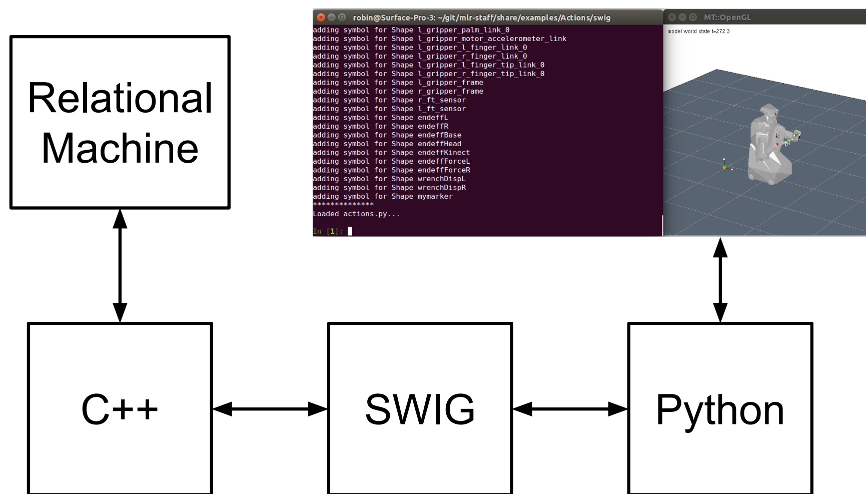


Abbildung 3.2.: Struktur der Implementierung der Skriptsprache

3.2.1. Kommunikationsebenen zwischen Kommandointerpreter und der RM

Die Skriptsprache wird in *Python* implementiert und kommuniziert bidirektional mit einem in *C++* implementierten Interface der *RM* mittels formatierten Strings. So ist der Datenaustausch mit einem einheitlichen Datentyp möglich. Um den *relationalen Zustand* der *RM* mit einem *Python* Kommandointerpreter kommunizieren zu lassen, sind folgende Schritte nötig (S. Abbildung 3.2).

1. Es wird ein einfaches Interface in *C++* erzeugt, welches die Schnittstelle der *RM* nach außen darstellt. Es stellt die Kommunikation mit einigen Grundfunktionen der *RM* mittels Strings zur Verfügung. So wird der Datenaustausch einheitlich gehalten und Veränderungen im Syntax werden einfach weitergegeben.
2. Mit Hilfe von *SWIG - Simplified Wrapper and Interface Generator* wird aus den *C++*-Funktionen ein *Python*-Erweiterungsmodul erstellt. Das bedeutet, dass die Funktionen und Datentypen *Python* zur Verfügung gestellt werden.
3. In *Python* wird das durch *SWIG* erzeugte Modul importiert, angepasst und Makros werden erstellt. Eingaben werden zu korrekten, für die *RM* verständliche Strings formatiert.
4. Da *Python* eine Skriptsprache ist, steht nun die gewünschte Funktionalität der *RM* einem Kommandointerpreter zur Verfügung.

3.2.2. Schnittstelle der Relational Machine

Die in *Python* implementierte Skriptsprache ändert den *relationalen Zustand* der *RM* durch das Starten von Aktivitäten, den *Activities*. Zudem kann sie „true-false"-Werte über Tupel von vorhandenen Symbolen, wie Terminierungssymbolen, Objekten oder *Activities* sowie Informationen über den Zustand der Effektoren und Sensoren des Roboters abfragen. Das Erstellen von Regeln, bzw. der Zugriff auf die Menge der Regeln der *RM* wird nicht implementiert!

Activities

Um eine *Activity*, eine Aktivität zu starten wird ein String übergeben, welcher ein *Literal*, der Name einer Aktion, und die zugehörigen Parameter enthält.

Um eine *Activity* zu stoppen wird ein String übergeben, welcher das zu stoppende *Literal* gefolgt von einem „`!`" enthält.

Symbole

Die *RM* kann nach „true-false"-Werten von Symbolen befragt werden. So kann abgefragt werden, ob eine *Activity* terminiert ist. Hierfür wird ein String mit einem *Literal* und dem „Converged"-Symbol übergeben und als Rückgabewert ein Boolean erhalten. Ist der Rückgabewert `true`, gilt die *Activity* als erfolgreich ausgeführt.

Effektoren und Sensoren

Zusätzlich können zu Effektoren, Gelenken und Sensoren des *PR2* einige Informationen abgefragt werden. Beispielsweise Gelenkwinkel und Position oder Orientierung der Effektoren [Tou15, Vgl. S. 1f].

3.3. Umsetzung des entwickelten Konzeptes

Im folgenden Kapitel werden die einzelnen Schritte der Implementierung der Skriptsprache beschrieben. Zuerst wird das *C++* Interface der *RM* beschrieben (S. Kapitel 3.3.1), dann wird das Wrappen mittels *SWIG* und die Verwendung des Erweiterungsmoduls in *Python* gezeigt.

3.3.1. Implemetierung eines Interfaces in C++

Um gewünschte Funktionalitäten der *RM* als Erweiterungsmodul für *Python* bereitstellen zu können, werden diese in einem kompakten *C++*-Interface zur Verfügung gestellt. Dieses Interface enthält zusätzlich zu den Funktionen einen *Interrupt Eventhandler*, welcher *Activities* durch ein in *Python* erzeugtes Interrupt, im diesem Fall `strg + c`, *Activities* stoppt. Auch enthält das Interface eine *Struct*, welche die *RM* instantisiert und erforderliche Datentypen zur Verfügung stellt (S. Algorithmus 3.1).

Algorithmus 3.1 Auszug aus der Headerdatei des C++ Interfaces

```
#include <string>
#include <vector>
#include <map>

//Die Typendefinition der erforderlichen Datentypen
typedef std::vector<double> doubleV;
typedef std::vector<int> intV;
typedef std::vector<std::string> stringV;
typedef std::map<std::string, std::string> dict;
using std::string;

//Die erforderlichen Funktionen
struct ActionSwigInterface{
  struct SwigSystem *S;

  //Instantisiert die RM und startet sie
  ActionSwigInterface();
  ~ActionSwigInterface();

  //Aktivitaeten Starten, Stoppen und Liste aktiver Aktivitaeten erhalten
  void setFact(const char* fact);
  void stopFact(const char* fact);
  stringV getFacts();


  //Funktionen zur Abfrage der Parameter der Effektoren
  stringV getShapeList();
  stringV getBodyList();
  stringV getJointList();

  dict getBodyByName (string bodyName);
  dict getShapeByName (string shapeName);
  dict getJointByName (string jointName);

  //Auf Bedingungen warten und Wahrheitswerte abfragen
  void waitForCondition(const stringV& literals);
  int waitForOrCondition(const stringV& literals);
  bool isTrue(const stringV& literals);

};
```

Algorithmus 3.2 Beispiel einer Minimalanwendung der erzeugten Bibliothek

```
import swig
interface = swig.ActionSwigInterface()

# Gibt Liste vorhandener Bodies zurueck
interface.getBodyList()

# Bewegt Roboter in Ausgangsstellung
interface.setFact("(HomingActivity){type=''homing'' tol=.01 PD=[1,1,1,10] }")

# Wartet auf erfolgreiche Ausfuehrung
interface.waitForCondition("HomingActivity conv")
```

3.3.2. Wrapping von C++ Quelltext mit SWIG

SWIG - Simplified Wrapper and Interface Generator ist ein Programmierwerkzeug, welches in *C++* implementierte Funktionen vielen anderen Programmiersprachen zur Verfügung stellen kann. Da die im Rahmen dieser Arbeit erstellte Skriptsprache in *Python* implementiert ist, bietet es sich als nützliches Werkzeug an.

Es wird eine Schnittstellenspezifikation mit benötigten Includes und Typendefinitionen angelegt. Der Interface-Generator *SWIG* erzeugt daraus ein *Python*-Erweiterungsmodul, welches die *C++*-Funktionen zur Verfügung stellt und benötigte Datentypen mappt [Haj08, Vgl. S. 11].

Nach dem Importieren des in *SWIG* erzeugten Modules und dem Instantisieren der *Struct* `ActionSwigInterface` stehen nun die in *C++* implementierten Funktionen und Datentypen in *Python* zur Verfügung (S. Algorithmus 3.2).

4. Anwendung und Diskussion der entwickelten Skriptsprache

Im folgenden Kapitel wird die Anwendung der entwickelten Skriptsprache beschrieben (S. Kapitel 4.1), über eine Benutzerstudie evaluiert (S. Kapitel 4.2), die Ergebnisse diskutiert und Verbesserungsbedarf herausgearbeitet (S. Kapitel 4.3).

4.1. Anwendung der entwickelten Skriptsprache

In diesem Kapitel werden die Anwendungsmöglichkeiten der entwickelten Skriptsprache erläutert. Zuerst wird die Syntax (S. Kapitel 3.2.2) und die implementierten Bewegungsconroller (S. Kapitel 4.1.1) der *RM* beschrieben. Daraufhin wird die Verwendung dieser Controller in *Python* gezeigt (S. Kapitel 4.1.2)

4.1.1. Syntax der Relational Machine

Der *relationale Zustand* beschreibt aktive Vorgänge, den *Activities* der *RM*. Jede *Activity* wird über genau einen Fakt identifiziert, an welchem seine Parameter angehängt sind. Ein Fakt besteht aus einem Tupel, die Parameter aus einer Dictionary.

Beispielsweise aktiviert
`(FollowReferenceActivity endeffR pos){ref1=”endeffR” target=[1, 1, 1] tol=.01 PD=[.5, .09, .1, 10.]}`
einen Positionscontroller (S. Kapitel 4.1.1), welcher den Effektor `ref1=endeffR`, den rechtern Gripper des *PR2*, zu der Position `target=[1,1,1]` in Weltkoordinaten bewegt. Die Bewegung folgt einem PD-Regler `PD=[.5, .09, .1, 10.]` und terminiert mit der Toleranz `tol=.01`.

Implementierte Controller der Relational Machine

Die *RM* erzeugt Roboterbewegung mittels Bewegungscontrollern. Es wird zwischen folgenden Arten der Roboterbewegung unterschieden.

pos bewegt einen Effektor zu einer Position in Weltkoordinaten. Um eine relative Bewegung zu erreichen, muss zuerst die Ausgangsposition abgefragt und die relative Bewegung aufaddiert werden.

vec richtet einen Vektor eines Effektors entlang eines Vektors in Weltkoordinaten aus.

vecDiff richtet einen Vektor eines Effektors entlang eines Vektors eines Objektes aus.

gazeAt lässt einen Vektor eines Effektors auf ein Objekt zeigen.

qItself bewegt ein Gelenk in eine Position.

wheels bewegt und dreht den Roboter zu einer Position in Weltkoordinaten. Um eine relative Bewegung zu erreichen, muss zuerst die Ausgangsposition abgefragt und die relative Bewegung aufaddiert werden.

homing bewegt den Roboter in Ausgangsstellung.

Jeder dieser Bewegungscontroller folgt einem PD-Regler und terminiert mit einer gegebenen Toleranz. Das Spezifizieren der Bewegungen erfolgt über folgende Parameter, welche zum Teil nur teilweise benötigt werden.

ref1 ist der Name des zu positionierenden Effektors des Roboters.

ref2 ist der Name des Referenzobjektes. Alle Positionierungen erfolgen in dessen Koordinatensystem. Ist im Fall von `qItself` nicht in Verwendung.

vec1 ist der normierte Vektor von `ref1`, welcher auszurichten ist. Wird nur im Fall von `gazeAt`, `vec` und `vecDiff` benötigt.

vec2 ist der normierte Referenzvektor von `ref2` in dessen Verhältnis die Ausrichtung geschieht. Ist `ref2` nicht gesetzt, ist `vec2` in Weltkoordinaten.

target ist der Vektor des Positionierungszieles in Meter im Koordinatensystem von `ref2`. Ist `ref2` nicht gesetzt, ist `target` in Weltkoordinaten. Im Fall von `qItself` ist `target` der Positionswinkel.

tol ist die Toleranz im Meter, ab welcher eine Bewegung als erfolgreich Ausgeführt gilt.

PD sind die Parameter des PD-Reglers. Er besteht aus *time scale*, *damping ratio*, *max vel* und *max acc*.

Algorithmus 4.1 Beispielquelltext für die Bewegung des Roboterautos

```
import swig
interface = swig.ActionSwigInterface()

# Haelt Roboterauto im Abstand d = 0.5m zur Wand
d = 0.5
interface.setFact("(FollowReferenceActivity Wand){ref1='auto' ref2='wand' type='pos'
    target=[d,0,0] PD=[1,1,1,10] }")

# Bewegt Roboteruto zu Ziel z
interface.setFact("(FollowReferenceActivity Ziel){ref1='auto' ref2='z' type='pos'
    PD=[1,1,1,10] }")

# Wartet auf erfolgreiche Ankunft des Roboterautos an Ziel z
interface.waitForCondition("FollowReferenceActivity ziel conv")

# Beendet Positionscontroller
interface.stopFact("(FollowReferenceActivity Wand pos)")
interface.stopFact("(conv FollowReferenceActivity Wand pos)")
interface.stopFact("(FollowReferenceActivity Ziel pos)")
interface.stopFact("(conv FollowReferenceActivity Ziel pos)")
```

4.1.2. Steuerung in Python

Die Steuerung des Roboters erfolgt über den *Python*-Kommandointerpreter. nach der Einbindung der Bibliothek und der Instantisierung des `ActionSwigInterface` mit `interface = swig.ActionSwigInterface()` kann der *Relationale Zustand* der *RM* beispielsweise mit

```
interface.setFact("(HomingActivity){type='homing' tol=.01 PD=[1,1,1,10] }")
```

verändert werden.

Ein *Python*-Skript, welcher das Roboterauto an einer Wand zu einem Ziel entlangfahren lässt, könnte wie in Algorithmus 4.1 implementiert sein (Vgl. Kapitel 2.2.3).

Um jedoch nicht immer den ganzen String selbst schreiben zu müssen, sind *Python*-Klassen implementiert, welchen die Formatierung der Strings übernehmen, die *Activity* starten und auf die Terminierung warten. Beispielsweise führt der Befehl

```
run(PosActivity('endeffR',[1,1,1]))
```

folgendes aus:

```
interface.setFact("(FollowReferenceActivity endeffR pos){type='pos' ref1='endeffR'
    target=[1., 1., 1.] tol=.01 PD=[.5, .9, .1, 10.]})
interface.waitForCondition("(conv FollowReferenceActivity endeffR pos)")
interface.stopFact("(FollowReferenceActivity endeffR pos)")
interface.stopFact("(conv FollowReferenceActivity endeffR pos)")
```

Algorithmus 4.2 Funktion im Plan-Format

```
def grabObject (shape)
  plan = [{"with":[alignGripperWithShape(shape), lookAtShape(shape)],
          "plan":[moveGripperToShape(shape, [0.1,0,0]),
                 openGripper(), moveGripperToPos([0.15,0,0]), closeGripper() }]
  return plan
```

Während der Ausführung kann jederzeit mit `strg + c` auf einen Interrupt-Eventhandler zugegriffen werden und die *Activity* unterbrochen werden (S. Kapitel 3.3.1).

Zudem lassen sich auch alle Befehle in eine Funktion packen, welche parallel und sequenziell aufgeführt werden können. So lassen sich alle Befehle auch im *Plan*-Format formulieren. Dieses ist ein Dictionary und besteht aus zwei Blöcken, dem *With*- und dem *Plan*-Block. Sie lassen sich beliebig verschachteln

Plan-Block Eine Liste von *Avtivities* im *Plan*-Block wird sequenziell ausgeführt. Jede muss terminiert sein, damit die nächste startet.

With-Block
Eine Liste von *Activities* im *With*-Block wird parallel ausgeführt. Diese werden erst beendet, wenn der Plan-Block terminiert.

Der Algorithmus 4.2 stellt eine Beispielfunktion dar, welche eine *Activity* im *Plan*-Format implementiert. Sie ist ein Makro zum Greifen eines Objektes („`shape`“ im Quelltext von Algorithmus 4.2). Die Ausrichtung des Grippers, (`alignGripperWithShape(shape)`), sowie die Fixierung des Objektes mit den Kameras des *PR2*, lookAtShape(shape), befindet sich im *With*-Block, die eigentliche Bewegung, (`moveGripperToShape(shape, [0.1,0,0])`, etc.), im *Plan*-Block.

4.2. Benutzerstudie

Um die Qualität der entwickelten Skriptsprache zu evaluieren, wird eine Benutzerstudie durchgeführt. Hierzu wird ein Aufgabenblatt ausgearbeitet, welches Benutzer einige Grundfunktionen der Skriptsprache anwenden lässt (S. Kapitel 4.2.1). Im Rahmen der Benutzerstudie werden einige Personen mit unterschiedlichen Vorkenntnissen gebeten dieses zu lösen und die Ergebnisse in einer Tabelle zusammengefasst (S. Kapitel 4.2.2).

4.2.1. Ausarbeitung eines Arbeitsblattes

Zur Durchführung der Benutzerstudie wird ein Arbeitsblatt ausgearbeitet. Das Aufgabenblatt besteht aus vier aufeinander aufbauenden Aufgaben. Zu jeder Aufgabe sind Hinweise

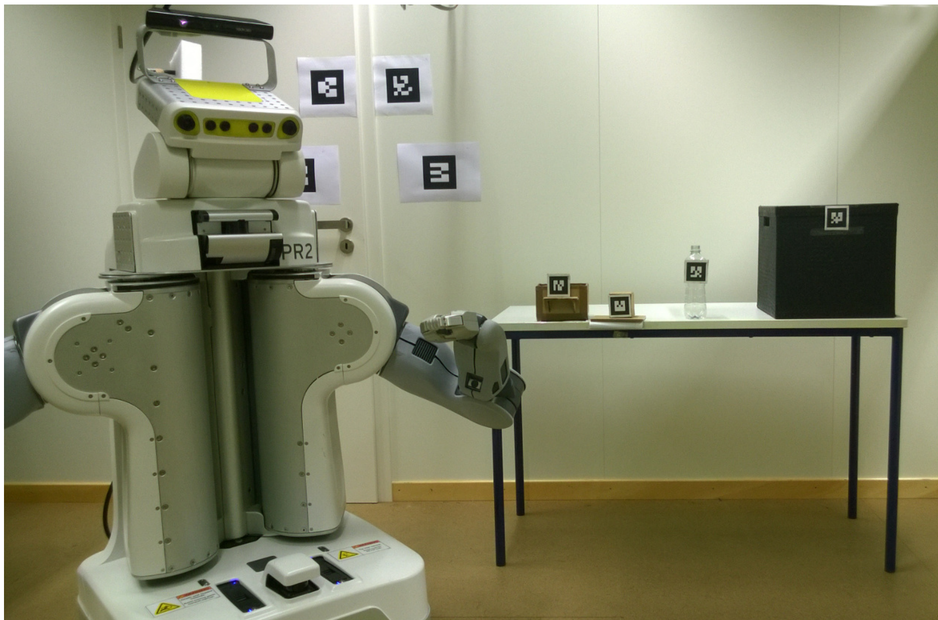


Abbildung 4.1.: Foto des Settings der Benutzerstudie

gegeben, welche zum erfolgreichen Ausführen notwendig sind. Als Setting gibt es den *PR2*-Roboter und einen Tisch mit mehreren Objekten und einen Mülleimer, welche über Marker identifiziert werden (S. Abbildung 4.1). Auf dem Monitor befindet sich der *Python*-Kommandointerpreter und ein 3D-Modell des *PR2*, welches den Soll- und den Istzustand abbildet. Das Ziel des Aufgabenblattes ist es den *PR2* den Tisch aufräumen zu lassen (S. Kapitel A – Anhang 1 - Aufgabenblatt der Benutzerstudie zur Skriptsprache zur intertaktiven Robotersteuerung).

1. Die erste Aufgabe soll mit dem Interface vertraut machen indem sie auffordert den *PR2* zu bewegen.
2. Die zweite Aufgabe fordert den Benutzer auf ein auf dem Tisch liegendes Objekt in die Hand zu nehmen. Es ist jedoch eine Beispiellösung angegeben. In der Aufgabe soll dem Benutzer ein Überblick über die Funktionalitäten der Skriptsprache verschafft werden.
3. Die dritte Aufgabe fordert nun den Benutzer auf die gelernten Fähigkeiten anzuwenden. Er soll selbständig den Tisch aufräumen.
4. Die vierte Aufgabe stellt den Benutzer vor die Aufgabe selbständig ein Makro im *Plan*-Format zu Programmieren und diese Auszuführen.

4.2.2. Durchführung der Benutzerstudie

Im Rahmen der Benutzerstudie werden sechs Personen gebeten das Aufgabenblatt zu lösen. Diese haben unterschiedlich viel Vorwissen. Es wird untersucht, ob die Aufgaben erfolgreich gelöst werden konnten und wie viel Hilfe benötigt wurde. Zudem wird die Zeit gestoppt, welche zur Lösung der Aufgaben benötigt wurde. Schließlich werden die Benutzer nach Verbesserungsvorschlägen gefragt. Die Ergebnisse sind in Tabelle 4.1 zusammengefasst.

Tabelle 4.1.: Ergebnis der Benutzerstudie

	Benutzer 1	**Benutzer 2**	**Benutzer 3**	**Benutzer 4**	**Benutzer 5**	**Referenz**
Vorerfahrung	Keine Vorerfahrung	Keine Vorerfahrung	Erfahrung im Programmieren	Schon am PR2 gearbeitet	Schon am PR2 gearbeitet	Erfahrung mit der entwickelten Skriptsprache
Aufgabe 1 Zeit	10 min	12 min	10 min	10 min	10 min	5 min
Aufgabe 1 Hilfe	Bei Klammerung	Bei Klammerung und Vektor	Kaum	Bei Klammerung	Keine	-
Aufgabe 2 Zeit	18 min	20 min	10 min	15 min	10 min	5 min
Aufgabe 2 Hilfe	Ab und zu bei Syntax und einige Tipps	Häufig bei Befehlen	Ein paar Hinweise	Etwas bei Syntax	Ab und zu bei Syntax und einige Tipps	-
Aufgabe 3 Zeit	10 min	10 min	10 min	10 min	10 min	20 min
Aufgabe 3 Hilfe	Objekte zu Gripper geführt und Tipps	Objekte zu Gripper geführt und Tipps	Objekte zu Gripper geführt	Objekte zu Gripper geführt	Objekte zu Gripper geführt	Kaum lösbar ohne Hilfe
Aufgabe 4 Zeit	5 min	5 min	5 min	5 min	5 min	5 min
Aufgabe 4 Hinweise	Nicht gelöst	PR2 fahren lassen	20 mal Gripper geöffnet	Makro zu Objekt in Mülleimer werfen	ein paar Bewegungen	Makro zum Objekt Greifen
Zeit gesamt	43 min	47 min	35 min	40 min	35 min	35 min
Kommentare	Alles i.O.	Mehr Makros, genauere Bewegungen	Bewegungsangaben ungenau, verliert Orientierung	3D-Modell verwirrend, Syntax gut	Syntax gut, intuitiv	-

4.3. Diskussion und Auswertung der Anwendung der entwickelten Skriptsprache

Das folgende Kapitel reflektiert die gewonnenen Erkenntnisse und betrachtet sie kritisch. Zu Beginn werden die Ergebnisse der Benutzerstudie ausgewertet (S. Kapitel 4.3.1) und die Sicherheit der Steuerung betrachtet (S. Kapitel 4.3.2). Daraufhin wird erörtert, was die entwickelte Sprache kann (S. Kapitel 4.3.3) und was nicht (S. Kapitel 4.3.4).

4.3.1. Auswertung der Benutzerstudie

Es lässt sich feststellen, dass die interaktive Skriptsprache von jeder Person erfolgreich angewendet werden konnte. Personen ohne Programmierkenntnisse, haben länger gebraucht und mehr Hilfestellung benötigt als solche mit mehr Erfahrung. Die Syntax und das Prinzip der Steuerung war allgemein verständlich. Es kam der Wunsch auf, dass mehr Makros vorhanden sein sollen, welche das Steuern erleichtern. Das 3D-Modell des Roboters war keine Hilfe und hat eher Verwirrung gestiftet. Auffallend war, dass die verschiedenen Personen sehr unterschiedlich an Aufgabe 2 herangegangen sind. Einige haben versucht den Gripper manuell zu positionieren, wohingegen andere die relationalen Bewegungscontroller genutzt haben. Die Bewegungen des Roboters sind schwer vorhersehbar, wenn mehrere kompetitive Bewegungscontroller aktiv sind. Auch gibt es im Kommadointerpreter keine lesbare Information über aktive Controller.

Die Erkennung der Marker ist fehlerbehaftet. Im Moment der Überdeckung eines Markers, verändert dieser seine Ausrichtung und Position unvorhersehbar, was zur Folge hat, dass der *PR2* seinen Gripper oft falsch ausrichtet und positioniert. Zudem fällt auf, dass die Bewegungen des *PR2*, insbesondere die Bewegung des Roboters im Raum z.T. nur sehr ungenau umgesetzt werden. Häufig erreicht der Gripper nicht präzise die gewünschte Position und hängt etwas tief

4.3.2. Sicherheit

Als Sicherheitsmechanismen der Robotersteuerung gibt es zwei Ebenen. Zum einen lassen sich sehr zuverlässig per `strg + c` *Activities* beenden, was den *PR2* sofort stoppen lässt und zum anderen gibt es eine Fernbedienung mit einem Not-Aus-Knopf. Der Roboter verfügt bisher über keine Kollisionskontrolle. Deshalb muss der Benutzer immer die Bewegungen des Roboters im Auge behalten um Schäden zu vermeiden. Während des Entwicklungsprozesses sind jedoch keine entstanden.

4.3.3. Was kann die entwickelte Skriptsprache

Die entwickelte Skriptsprache implementiert die *RM*, jedoch nutzt sie nur die Menge des Symbole und den *relationalen Zustand*. Sie nutzt nicht die Möglichkeit des Planens, die Menge der Regeln der *RM*, aus. Deshalb implementiert die Skriptsprache im Prinzip keine *FSM* und ist nicht mehr als eine imperative Robotersteuerung. Jedoch lässt sich planendes Verhalten mit den vorhanden Funktionen, welche den *relationalen Zustand* abfragen in *Python* implementieren, was jedoch nicht Teil dieser Arbeit ist. Sie stellt ein einfaches Werkzeug dar, auf welches andere Mechanismen oder Konzepte schnell angewendet werden können, ohne tiefes Verständnis der zugrundeliegenden Prinzipien haben zu müssen.

4.3.4. Verbesserungsbedarf

Leider startet die entwickelte Skriptsprache häufig nicht richtig, jedoch einmal gestartet läuft sie stabil. Sinnvoll wäre es, um die Möglichkeit der Planung von Roboterverhalten zu ermöglichen, zusätzlich die Menge der Regeln der *RM* zu integrieren. Zudem sollte die Möglichkeit des Nutzens konditionaler Bedingungen auch in *Python* implementiert sein. Beispielsweise könnte das *Plan*-Format dahingehend erweitert werden. Um die entwickelte Skriptsprache sinnvoll nutzen zu können, ist es unumgänglich, dass die Erkennung der Marker und die Positionierung der Effektoren zuverlässig funktioniert. Wünschenswert wäre auch ein physikalisches Modell der Umwelt, was eine Kollisionsvermeidung ermöglichen würde.

5. Zusammenfassung

In der vorliegenden Studienarbeit wurde eine Skriptsprache zur interaktiven Robotersteuerung konzipiert und umgesetzt. Diese ermöglicht schnelles Prototyping und die interaktive Steuerung des Roboters *PR2*. Im ersten Abschnitt der Arbeit werden einige Konzepte der Erzeugung von Roboterverhalten beschrieben. Neben modellbasierten Ansätzen, wie der diekten Programmierung oder dem modellbasierten *Reinforcement Learning* werden modellfreie und hybride Ansätze, wie dem *Learning from Demonstration* gezeigt und verglichen. Schließlich wird die *Relational Machine*, welche das der entwickelten Skriptsprache zugrundeliegende Framework darstellt in Vergleich mit anderen Steuerungsprinzipien gesetzt.

Im darauffolgenden Abschnitt wird das Konzept der interaktiven Skriptsprache erstellt. Sie ändert den *relationalen Zustand* der *Relational Machine* und erzeugt so Roboterverhalten. Aus vorhandenem *C++*-Quelltext wird mittels *SWIG* ein *Python*-Erweiterungsmodul erstellt, welches die Funktionen zur Manipulation des *relationalen Zustandes* beinhaltet. So steht die Funktionalität der *RM* einem Kommandointerpreter zur Verfügung.

Im dritten Abschnitt wird die Anwendung der Skriptsprache gezeigt und die Nutzbarkeit in einer Benutzerstudie evaluiert. Neben der Robotersteuerung im Kommandozeileninterpreter lassen sich auch komplexere Makros in *Python* erstellen, welche das parallele und sequenzielle Steuern von Positionscontrollern erlauben. Die Benutzerstudie zeigt, dass die Anwendung der Skriptsprache intuitiv möglich ist, jedoch zur sinnvollen Nutzbarkeit noch einige Schritte notwendig sind. Neben ungenauer Wahrnehmung der Umwelt reagiert der *PR2* oft unvorhersehbar auf Eingaben.

A. Anhang 1 - Aufgabenblatt der Benutzerstudie zur Skriptsprache zur intertaktiven Robotersteuerung

Folgend das Aufgabenblatt der Benutzerstudie, um die Nutzbarkeit der entwickelten Skriptsprache zu evaluieren. Es besteht aus vier aufeinander aufbauenden Aufgaben (Seite 1 & 2) und einem Anhang mit weiteren Informationen (Seite 3). Diese haben sechs Benutzer erfolgreich gelöst (S. Kapitel 4.2.2)

Aufgabenblatt der Benutzerstudie zur Skriptsprache zur interaktiven Robotersteuerung

Aufgabe 1

Machen Sie sich mit dem Interface vertraut, fahren Sie den PR2 etwas herum und beenden Sie die Fahrt vor dem Tisch.

- `run(`***a***`)` (***a*** = Ein beliebiger Befehl) führt Befehle aus und wartet auf deren Terminierung. Bsp.: `run(moveRobot([0.2,0.4,50]))`.
  Achten Sie auf korrekte Klammerung.
- **strg + c** bricht jeder Zeit alle Befehle ab.
- `moveRobot([`***a*, *b*, *c***`])` (***a*** = Bew. nach vorne in m; ***b*** = Bew. nach links in m; ***c*** = Drehung in ° gegen den Uhrzeigersinn) bewegt den Roboter.
- `moveBaseToShape("`***a***`")` (***a*** = Name des Markers) positioniert den Roboter vor einem Marker. Marker muss im Sichtfeld liegen.
- `homing()` bewegt Roboterarme in Ausgangsstellung.

Aufgabe 2

Bewegen Sie die Roboterhand und versuchen Sie mit den Objekten zu interagieren.
Nehmen Sie eines in die Hand.
Hinweise siehe "Anhang 1 - Koordiantensystem und Objekte".

- `run_in_bg(`***a***`)` (***a*** = Ein beliebiger Befehl) führt Befehle im Hintergrund aus. Bsp.: `run_in_bg(alignGripperHorizontal())`.
  Beenden der aktiven Befehle mit **strg + c**.
- `moveGripperToPos([`***x*, *y*, *z***`])` (***x***,***y***,***z*** = Bewegungsvektor in m) bewegt den Gripper relativ zu sich im Raum.
- `moveGripperToShape("`***s***`", [`***x*, *y*, *z***`])` (***s*** = Name des Markers; ***x***,***y***,***z*** = Offsetvektor im Koordinatensystem des Markers in m) bewegt Gripper zu Marker mit gegebenem Offset. **[***x*,*y*,*z***]** ist optional.
- `alignGripperWithShape("`***s***`")` (***s*** = Name des Markers) richtet Gripper zu Marker aus.

- `alignGripperHorizontal()` richtet Gripper horizontal aus.
- `alignGripperVertical()` richtet Gripper vertikall aus.
- `openGripper()` öffnet Gripper.
- `closeGripper()` schließt Gripper.
- `lookAtShape(”`***s***`”)` (***s*** = Name des Markers) schaut Marker an.

Bsp. Lösung:

1. `run_in_bg(alignGripperWithShape(”marker0”)`
2. `run_in_bg(lookAtShape(”marker0”)`
3. `run(moveGripperToShape(”marker0”, [0.1,0,0]))`
4. `run(openGripper())`
5. `run(moveGripperToPos([0.15,0,0]))`
6. `run(closeGripper())`
7. **strg + c**

Aufgabe 3

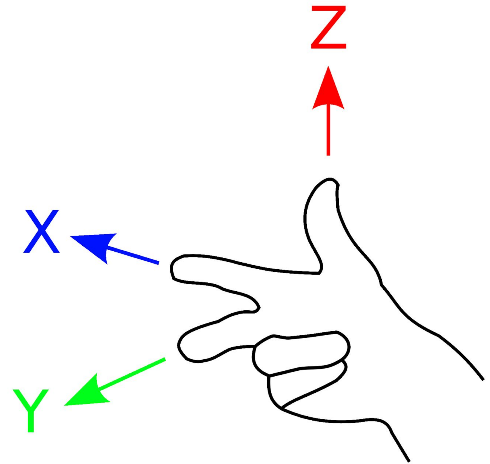
Räumen Sie den Tisch auf.

- `grabMarker(”`***s***`”)` (***s*** = Name des Markers) greift Marker
  (Der Quelltext aus dem Bsp. von Aufgabe 2).
- `throwToBin()` probieren Sie es doch einfach aus.
- `SIDE.DEFAULT =` ***a*** (***a*** = `RIGHT` oder `LEFT`) wechselt den zu verwendeten Arm.

Aufgabe 4

Erstellen Sie Ihr eigenes Makro im Plan-Format ausgehend von der Funktion `grabMarker(”`***s***`”)` und führen Sie es aus.
Nehmen Sie doch z.B. eine Bewegung aus Aufgabe 3 oder lassen Sie den Roboter tanzen.
Hinweise siehe “Anhang 2 - Auszug aus der Bedienungsanleitung”.

Anhang 1 - Koordinatensystem und Objekte



Das Koordinatensystem folgt der Drei-Finger-Regel der rechten Hand. Die X-Achse schaut aus Marker raus.

Die Namen der Objekte:

- Der Würfel heißt `marker0`
- Die Flasche heißt `marker1`
- Die Schachtel heißt `marker2`
- Die Kiste heißt `marker3`

Anhang 2 - Auszug aus der Bedienungsanleitung - The Plan Format

Soon you will be in the need to move such complex behaviors or plans around. We have a format to achieve that.

A plan is a lightwight datastructure to store sequential and simultaneous activities. A single Activity is already a plan. To create more complex plans, the following construxts are possible:

- A list of activities or plans is run sequentially. Each one must be converged for the next one to start
- A tuple of activities or plans is run simultaneously. Everything is flatten, i.e. `(a, [b, c], d)` is equally treated as `(a, b, c, d)`. All activities have to be converged for the tuple to be considered converged
- A dict with two entries. First `"with"` contains a list of activities, second `"plan"` contains a plan. The list of activities in the “with” list are run simultaneous to the plan. However, when the last activity of the plan is converged they are stopped regardless of the convergence status.

For example a plan could look like:

```
plan = [{"with": [alignGripperWithShape(shape), lookAtShape(shape)],
"plan": [moveGripperToShape(shape, [0.1,0,0]),
openGripper(), moveGripperToPos([0.15,0,0]), closeGripper() }]
```

Literaturverzeichnis

[ACVB09] B. D. Argall, S. Chernova, M. Veloso, B. Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009. (Zitiert auf Seite 15)

[BMTR12] M. Beetz, L. Mosenlechner, M. Tenorth, T. Ruhr. Cram–a cognitive robot abstract machine. In *5th International Conference on Cognitive Systems (CogSys 2012)*. 2012. (Zitiert auf Seite 9)

[Bro91] R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991. (Zitiert auf Seite 12)

[Gar] W. Garage. PR"Hardware Specs. URL `http://www.willowgarage.com/pages/pr2/specs`. (Zitiert auf Seite 23)

[GJ10] D. H. Grollman, O. C. Jenkins. Can We Learn Finite State Machine Robot Controllers from Interactive Demonstration? In *From Motor Learning to Interaction Learning in Robots*, S. 407–430. Springer, 2010. (Zitiert auf Seite 17)

[Haj08] F. Hajji. *Das Python-Praxisbuch: Der große Profi-Leitfaden für Programmierer*. Pearson Deutschland GmbH, 2008. (Zitiert auf Seite 28)

[KBP13] J. Kober, J. A. Bagnell, J. Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, S. 0278364913495721, 2013. (Zitiert auf Seite 16)

[KKGB10] G. Konidaris, S. Kuindersma, R. Grupen, A. S. Barreto. Constructing skill trees for reinforcement learning agents from demonstration trajectories. In *Advances in neural information processing systems*, S. 1162–1170. 2010. (Zitiert auf Seite 17)

[KKGB11] G. Konidaris, S. Kuindersma, R. Grupen, A. Barto. CST: Constructing Skill Trees by Demonstration. *Proceedingsof the ICML Workshop on New Developments in Imitation Learning*, 2011. (Zitiert auf Seite 17)

[LL09] B.-s. Liu, F.-C. Lin. A Semiglobally Stable PD-I (PD) Regulator for Robot Manipulators. In *Measuring Technology and Mechatronics Automation, 2009. ICMTMA'09. International Conference on*, Band 1, S. 763–766. IEEE, 2009. (Zitiert auf Seite 21)

[LRJ06] M. Loetzsch, M. Risler, M. Jungel. XABSL-a pragmatic approach to behavior engineering. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, S. 5124–5129. IEEE, 2006. (Zitiert auf den Seiten 9, 12, 13, 17 und 20)

[Mat07] M. J. Matarić. *The robotics primer.* Mit Press, 2007. (Zitiert auf Seite 16)

[Mil10] G. Milighetti. *Multisensorielle diskret-kontinuierliche Überwachung und Regelung humanoider Roboter*, Band 6. KIT Scientific Publishing, 2010. (Zitiert auf den Seiten 11, 12 und 13)

[TLJ13] M. Toussaint, T. Lang, N. Jetchev. Kognitive Robotik—Herausforderungen an unser Verständnis natürlicher Umgebungen. *at-Automatisierungstechnik Methoden und Anwendungen der Steuerungs-, Regelungs-und Informationstechnik*, 61(4):259–268, 2013. (Zitiert auf Seite 11)

[Tou15] M. Toussaint. Relational Machine Interfacing between robot activity control and relational learning and planning methods. 2015. (Zitiert auf den Seiten 9, 20 und 26)

Alle URLs wurden zuletzt am 30. 08. 2015 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift