Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

dipl Nr. 3735

Aktionen autonomer Systeme als Elemente relationaler nebenläufiger Markov-Entscheidungsprozesse

Robin Böhm

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Marc Toussaint
Betreuer/in:	M.Sc. Peter Englert, M.Sc. Matthew Bernstein
Beginn am:	4. Dezenber 2015
Beendet am:	3. Juni 2016
CR-Nummer:	I.2.9, I.2.4

Kurzfassung

In der vorliegenden Arbeit wird die Frage behandelt, welche Eigenschaften die Implementierung von Aktionen autonomer Agenten erfüllen muss um als Elemente relationaler Markov-Entscheidungsprozesse nutzbar zu sein. Sie formalisieren ein stochastisches Zustandsmodell als Grundlage des Entscheidungsproblems der Aktionsplanung. Neben der Untersuchung, wie dieses Modell erweiterbar ist um nebenläufige Aktionsausführung zu ermöglichen, werden auch verschiedene Modelle beschrieben die auftretende Ressourcenkonflikte lösen. Anhand der theoretischen Untersuchung dieser Frage wird an der praktischen Umsetzung einer Softwareumgebung zur Implementierung von autonomen Aktionen gearbeitet. Diese bietet dem Programmierer ein Interface, welches die stabile Umsetzung unterschiedlichster Prozesse der Robotik ermöglicht, sodass diese einer Aktionsplanung zur Verfügung stehen. Es wird um eine Ausnahmebehandlung erweitert, sodass an Stellen, die vorher zum Programmabbruch geführt haben nun eine Fehlerbehandlung greift welche eine adäquate Reaktion des Aktionsplaners ermöglicht. Neben der genauen Beschreibung der Architektur und einer Anleitung zur Implementierung neuer Aktionen, werden auch die Möglichkeiten der Software experimentell evaluiert.

Danksagung

Zunächst möchte ich mich an dieser Stelle bei einigen Personen bedanken, welche diese Arbeit ermöglicht haben. Vielen herzlichen Dank an Prof. Dr. Marc Toussaint, M.Sc. Peter Englert, M.Sc. Matthew Bernstein und Claudia und Martin Böhm.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen

Abkürzungsverzeichnis

MDP Markov Decision Process

SMDP Semi Markov Decision Process

CoMDP Concurrent Markov Decission Processes

CAM Concurrent Action Model

RAP Relational Activity Processes

STRIPS Stanford Research Institute Problem Solver

CRAM Cognitive Robot Abstract Machine

CPL CRAM Plan Language

RoLL Robot Learning Language

ROS Robot operating system

RM Relational Machine

KOMO Newton methods for k-order Markov Constrained Motion Problems

1. Einleitung

Der Umgang mit unvorhersehbarem, stochastischem Verhalten natürlicher Prozesse ist eine der vielschichtigen Aufgaben der Aktionsplanung von Roboterverhalten. Damit Manipulationsaufgaben autonom von Agenten gelöst werden können, muss ein Formalismus für die reale Umgebung gefunden werden, welcher diese Vorgänge adäquat abstrahieren kann. Verhaltenserzeugung unter unsicheren Bedingungen ist eine schwierige Aufgabe und noch immer Gegenstand vieler Forschungsarbeiten [YMS03, S. 195]. Die vorliegende Arbeit nimmt sich dieser Problematik an, gibt einen Überblick über die Forschung und entwickelt ein Konzept, welches es erlaubt, verschiedenste Prozesse autonomer Systeme der Verhaltenserzeugung zur Verfügung zu stellen, welches die Stochastik natürlicher Umgebungen berücksichtigt.

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

1.1. Problemstellung und Motivation

Manuell gesteuerte bzw. teleoperierte Roboter sind teilweise in der Lage eine beeindruckende Performance bei der Bewältigung von Alltagsaufgaben zu erzielen, jedoch haben autonome Roboter bis heute große Probleme selbst die einfachsten Manipulationsaufgaben zu lösen. Das lässt sich an zwei Videos eindrucksvoll beobachten. Das erste Video zeigt den *PR1*, wie er teleoperiert ein Zimmer mit Leichtigkeit aufräumt [BW, S.]. Auf dem zweiten ist zu sehen, wie ein *PR2* autonom Socken faltet jedoch bei dieser Aufgabe auf Hilfsobjekte und sehr umständliche Ausführung angewiesen ist [WMF$^+$, S.]. Zudem ist die Aufteilung der Aufgabe in verschieden Teilaktionen Erkennbar, ist. Dies zeigt, dass die Probleme der autonomen Robotik nicht in der Mechanik sondern in den Steuerungssystemen liegen. Eine der Herausforderungen ist die Formalisierung eines geeigneten Weltmodells, welches der abstrakten Repräsentation der Elemente auch entsprechende Bedeutung zuweist. So ist zur Verhaltenserzeugung die einfache Repräsentation der Objekte nicht ausreichend, sondern es muss auch ein *Verständnis* über die Bedeutung der Objekte vorliegen [Har90, S. 335f]. Auch wird versucht die Eigenschaften einzelner Steuerungsbefehle so zu formulieren, dass diese direkt auf der Ebene der Objektmanipulation agieren können, anstatt sich mit der Robotermechanik befassen zu müssen [TLJ13, S. 3].

Ein *MDP - Markov-Entscheidungsprozess* ist das Modell eines Entscheidungsproblems, welches auf die Verhaltenserzeugung autonomer Agenten angewendet werden kann. Es formuliert stochastische Änderungen des Zustandes durch Roboteraktionen oder durch externe Einflüsse und bildet so ein Modell, welches die natürliche Unsicherheit realer Umgebungen beinhaltet. Dadurch können Aktionen unterschiedlichster Art einer Aktionsplanung zur Verhaltenserzeugung auf höchster Ebene zur Verfügung gestellt werden. Jedoch ist die Übertragung der *MDPs* auf vorhandene Anwendungsfälle mit ihren vielschichtigen Ansprüchen, wie Parallelisierung oder der Verarbeitung von Fehlern oft schwierig und bedarf näherer Untersuchung.

1.2. Zielsetzung der Arbeit

Das Ziel dieser Arbeit ist die Untersuchung der Anwendung von *MDPs* anhand der Weiterentwicklung eines Frameworks, welches verschiedenste Aktionstypen mit ihren unterschiedlichen Eigenschaften einer Aktionsplanung zur Verfügung stellt. Dazu werden verschiedene Methoden zur Parallelisierung von Aktionen in *MDPs* diskutiert und anhand von Programmierumgebungen die Anwendung gezeigt. Um auch auftretende Fehler als mögliche Elemente von Aktionen nutzbar zu machen, wird an einer Ausnahmebehandlung von Aktionen gearbeitet und die Anwendung beschrieben.

1.3. Gliederung

Die Arbeit ist auf folgende Weise gegliedert:

Kapitel 2 – Aktionsplanung in natürlichen Umgebungen beschreibt das Entscheidungsproblem der Verhaltenserzeugung als *MDPs* und erörtert verschiedene Möglichkeiten zu Parallelisierung und Nutzung in zeit-stochastischen Szenarien.

Kapitel 3 – Frameworks zur komplexen Aktionsplanung stellt das Konzept *Kognitiver Robotik* vor und beschreibt die Funktionalität verschiedener Softwareumgebungen zur Verhaltenserzeugung nach diesem Prinzip. Es vergleicht diese und diskutiert die Grenzen der Anwendung.

Kapitel 4 – Implementierung autonomer Aktionen als Activities beschreibt die Konzeption eines Frameworks, welches verschiedene Roboteraktivitäten einer Aktionsplanung zur Verfügung stellt und gibt Beispiele der Anwendung.

Kapitel 5 – Experimentelle Evaluierung der Ausnahmebehandlung diskutiert die gewonnenen Ergebnisse kritisch anhand eines Experiments und gibt einen Überblick über Erweiterungsmöglichkeiten.

Kapitel 6 – Zusammenfassung und Ausblick

Anhang A – Anhang - Anleitung zur Implementierung einer neuen Activity ist die Anleitung zur Erstellung einer neuen *Activity*, welche die entwickelte Ausnahmebehandlung integriert.

2. Aktionsplanung in natürlichen Umgebungen

Das vorliegende Kapitel behandelt die Frage, wie natürliche Umgebungen abstrahiert werden können, damit sie den Formalismen einer Aktionsplanung zur autonomen Verhaltenserzeugung genügen. Natürliche Umgebungen, die realen Szenarien in denen ein Agent agiert, sind in ihrer Natur immer stochastisch, d.h. dass der Effekt eines Vorgangs nur ungenau vorhersehbar ist. Zudem ist die Modellbildung realer Umgebungen eine Abstraktion und somit ungenau. Wird ein Entscheidungsproblem zur Lösung einer Aufgabe erstellt, muss ein Formalismus gefunden werden, welcher reale Prozesse mit ihrer Stochastik integrieren kann. Ein geeignetes Modell ist ein *MDP - Markov-Entscheidungsprozess*. Er formalisiert das Entscheidungsproblem der Aktionsplanung und ist in der Lage die stochastischen Zustandsänderungen der Welt zu formalisieren.

Zuerst wird in Kapitel 2.1 ein *MDP* als geeignetes Modell zur Aktionsplanung unter stochastischen Bedingungen gezeigt und in Kapitel 2.2 eine geeignete Repräsentation der Umwelt zur Lösung des Entscheidungsproblems der Verhaltenserzeugung beschrieben. In Kapitel 2.3 werden darauf einige Möglichkeiten der Erweiterung von *MPDs* um Nebenläufigkeit dargestellt und in Kapitel 2.4 die Problematik von andauernden Prozessen für die Aktionsplanung gezeigt. Schließlich werden in Kapitel 2.5 verschiedene Modelle verglichen und kritisch diskutiert.

2.1. Aktionsplanung als Entscheidungsproblem

Die Aktionsplanung zur Erzeugung von Verhalten autonomer Agenten ist ein Entscheidungsproblem. Es ist beschrieben durch die Zustandsmenge S und die Menge der möglichen Aktionen A. Ein Zustand $s \in S$ beschreibt einen möglichen Zustand der Welt und des Roboters durch seine Variablen. Dagegen beschreibt eine Aktion $a \in A$ eine der möglichen Aktionen des Agenten, welche im Falle der Ausführung aus einem Zustand s einen Zustand s' durch Manipulation der Umwelt oder des Roboterzustandes erzeugt.

Das Entscheidungsproblem selbst ist das Finden einer Strategie π um eine gestellte Manipulationsaufgabe zu lösen. Eine Strategie $\pi : S \to A$ ist die Auswahl der richtigen Aktionen $a \subseteq A$ in jedem Zustand $s \in S$, sodass möglichst effizient ein Zielzustand $z \in G \subseteq S$

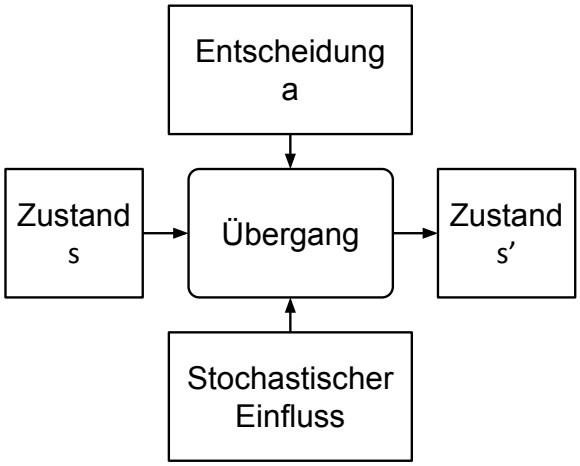


Abbildung 2.1.: Ablauf eines Zustandsübergangs im Markov-Entscheidungsprozess

aus der Menge der Zielzustände G erreicht wird. Eine Strategie π^* ist optimal, wenn die erwarteten Kosten $\Re$ einer gegebenen Kostenfunktion $R : S \times A \times S \to \Re$, welche jedem Zustandsübergang Kosten zuordnet, minimal sind.

2.1.1. Markov Entscheidungsprozesse

Ein *MDP - Markov-Entscheidungsprozess* erweitert die Formulierung des Entscheidungsproblems um die stochastischen Eigenschaften natürlicher Umgebungen [Thr00, S. 306]. Somit wird der Prozess des Zustandsübergangs stochastisch. Wird in Zustand $s \in S$ eine Aktion $a \in A$ ausgeführt wird mit Wahrscheinlichkeit p Zustand s' erreicht (S. Abbildung 2.1). Die Übergangswahrscheinlichkeit von a zu allen möglichen Zuständen s' ergibt immer 1.

Ein *MDP* wird hier definiert als ein Tupel $M = (S, A, Ap, P, R, G)$ nach [MW08, S. 3f] mit

- S, einer endlichen Menge diskreter Zustände.
- A, einer endlichen Menge der Aktionen.
- $Ap : S \to \mathcal{P}(A)$, einer Funktion der anwendbaren Aktionen je Zustand ($\mathcal{P}$ ist Potenzmenge).
- $P : S \times A \times S \to [0, 1]$, der Übergangsfunktion. $P(s'|s, a)$ ist die Wahrscheinlichkeit Zustand s' zu erreichen, wenn Aktion a in Zustand s ausgeführt wird.
- $R : S \times A \times S \to \Re$, der Kostenfunktion. Jedem Zustandsübergang durch eine Aktion a werden Kosten zugeordnet.
- $G \subseteq S$, der Menge der Zielzustände mit $z \in G$ einem Zielzustand.

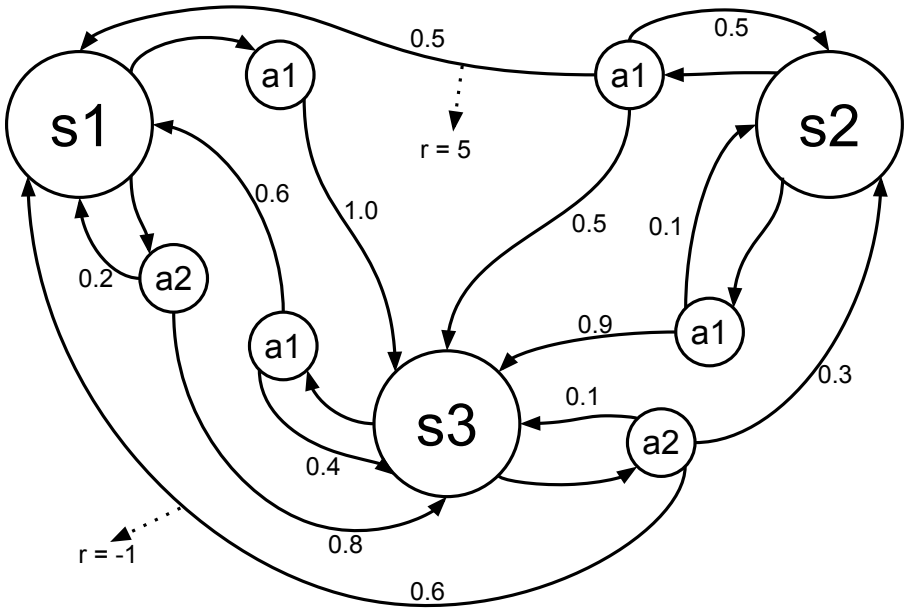


Abbildung 2.2.: Beispiel eines einfachen Markov-Entscheidungsprozesses mit drei Zuständen und zwei Aktionen

Jedem Zustandsübergang sind neben der Übergangswahrscheinlichkeit P auch Kosten $\Re$ zugeordnet. Ziel des Aktionsplaners ist es eine Strategie $\pi : S \rightarrow A$ zu finden, welche einen Zielzustand $z \in G$ erreicht und die erwarteten Kosten $\Re$ minimiert. Dadurch können Randbedingungen wie Effizienz in das Modell integriert werden. Es sei $J : S \rightarrow \Re$ die Kostenfunktion, welche jedem Zustand $s \in S$ die erwarteten Kosten zu einem Zielzustand $z \in G$ zuordnet. Eine Strategie π definiert sich wie folgt:

$$\pi_J(s) = \underset{a \in Ap(s)}{\arg\min} \sum_{s' \in S} P(s'|s,a)\Big(R(s,a,s') + J(s')\Big)$$

Somit leitet sich die optimale Strategie π^* von der optimalen Kostenfunktion J^* ab, welche sich über folgende beiden *Bellmanngleichungen* definiert:

$$J^*(s) = 0, \texttt{if } s \in G \texttt{ else}$$
$$J^*(s) \min_{a \in Ap(s)} \sum_{s' \in S} P(s'|s,a)\Big(R(s,a,s') + J^*(s')\Big)$$

Ein klassisches *MDP* ist ein geeignetes Modell für die Aktionsplanung unter stochastischen Bedingungen. Jedoch hat das Modell zwei Schwächen: Zum einen werden die Aktionen sequenziell ausgeführt und zum anderen geschehen alle Aktionen unmittelbar. In der Realität jedoch sind diese Annahmen unrealistisch, da Aktionen Zeit brauchen und z.T. eine Parallelausführung sinnvoll ist um die Effizienz zu erhöhen [WM04, S. 2f]. Beispielsweise führt ein

Roboterarm eine Aktion aus, während der andere sich gerade in einem Zustandsübergang befindet und beschäftigt ist.

2.2. Relationale Repräsentation

Neben der Auswahl eines geeigneten Modells für das Entscheidungsproblem muss auch eine geeignete Repräsentation der Umwelt gewählt werden. Es kann beispielsweise jeder Zustand explizit $S = \{z_1, z_2, ..., z_{n-1}, z_n\}$ oder aber durch Funktionen $z_x = f_x(\vec{X})$ mit ihren Zustandsvariablen $\vec{X}$ formalisiert werden. Es lässt sich erkennen, dass eine explizite Repräsentation in hinreichend großen Bereichen schnell an ihre Grenzen kommt. Objekte und andere Elemente können durch Symbole bzw. Konstanten, welche mit ihren Eigenschaften verknüpft sind repräsentiert und in Zusammenhang gebracht werden. Diese Art der impliziten Repräsentation wird unter dem Begriff der *Relationalen Repräsentation* zusammengefasst [LTK12, S. 3732].

Die Symbole, welche neben den Objekten auch fundamentale Eigenschaften der Roboterwelt repräsentieren, werden mithilfe von Prädikaten in Zusammenhang gesetzt [TMM+15, S. 1f]. Beispielsweise kann so mit dem Prädikat $\texttt{on}(objA, objB)$ formalisiert werden, dass ein Objekt $objA$ *auf* einem anderen Objekt $objB$ liegt. So ein Prädikat bedeutet mit Variablen nur die Relation *auf*. Erst durch das Einsetzen von Konstanten wie Objekten, wird daraus eine Aussage. Auf diese Weise werden Objekte in einen relationalen Zusammenhang gebracht und stehen der Prädikatenlogik zur Verfügung. So beschreibt die *Relationale Repräsentation* der Welt nicht nur die Eigenschaften der, sondern auch die Beziehungen zwischen den Objekten. Mithilfe der Prädikatenlogik können nun Aussagen über die Eigenschaften der Umgebung gemacht werden.

2.2.1. STRIPS - Stanford Research Institute Problem Solver

Eine standardisierte relationale Formalisierung eines Problems der Aktionsplanung ist *STRIPS - Stanford Research Institute Problem Solver* [FN71, S. 1ff]. Das Entscheidungsproblem wird in *STRIPS* durch den Startzustand, der Menge der verfügbaren Aktionen und deren Zustandsübergänge und dem Zielzustand definiert. Der Zustand ist als wohldefinierte Funktion erster Ordnung definiert. Beispielsweise könnte die Position $\texttt{at}(a, b)$ der Kisten $objA$ und $objB$ an $posA$ und $posB$ sowie die Position des Roboter $\texttt{atr}(a)$ an $posC$ und auf folgende Weise definiert werden:

$$\texttt{atr}(posC), \texttt{at}(objA, posA), \texttt{at}(objB, posB)$$

Es wäre zusätzlich die Bedingung, dass ein Objekt sich nicht an zwei Orten gleichzeitig befinden wie folgt beschrieben:

$$(\forall u \forall v \forall objB)\{[\texttt{at}(u, x) \wedge (x \neq objB)] \rightarrow \texttt{at}(u, objB)\}$$

Aktionen werden auf ähnliche weise definiert. Beispielsweise bewegt die Aktion $\texttt{goto}(a, b)$ den Roboter von a nach b:

$$\texttt{goto}(posA, posB) : \texttt{atr}(posA) \rightarrow \neg\texttt{atr}(posA), \texttt{atr}(posB)$$

Die Implementierung solch einer Aktion beinhaltet Vorbedingungen und die Änderungen des Zustandes, also Änderung der Menge der Prädikate, welche den Zustand definieren. Schließlich ist auch die Menge der Zielzustände wohldefiniert z.B. durch

$$\texttt{at}(objA, posA) \wedge \texttt{at}(objB, posA)$$

2.2.2. Probabilistic STRIPS

In *Probabilistic STRIPS* wird das klassische *STRIPS*-Modell durch probabilistische Regeln erweitert [ZPK05, S. 912]. Dadurch kann der unsichere Ausgang einer Aktion beschrieben werden. Somit eignet sich die Beschreibung auch für stochastische *MDPs*. Das folgende Beispiel der Aktion $\texttt{pickup}(objA, objB)$ mit den Vorbedingungen $\texttt{on}(objA, objB)$, dass $objA$ sich auf $objB$ befindet und `inhandNil`, dass die Hand der Roboters leer ist, hat drei verschiedene mögliche Ausgänge. Mit der Wahrscheinlichkeit $p = 0.8$ landet das Objekt $objA$ in der Hand, mit der Wahrscheinlichkeit $p = 0.1$ fällt das Objekt $objA$ auf den Tisch T und mit der Wahrscheinlichkeit $p = 0.1$ passiert nichts.

$$\begin{aligned}
&\texttt{ObjektA}(objA), \texttt{ObjektB}(objB), \texttt{Tisch}(T) \\
&\texttt{pickup}(objA, objB) : \\
&\texttt{on}(objA, objB), \texttt{inhandNil} \\
&\rightarrow \begin{cases} .80 : \neg\texttt{on}(objA, objB), \texttt{inhand}(objA), \neg\texttt{inhandNil}, \texttt{clear}(objB) \\ .10 : \neg\texttt{on}(objA, objB), \texttt{on}(objA, T), \texttt{clear}(objB) \\ .10 : \texttt{no change} \end{cases}
\end{aligned}$$

Allgemein definiert sich die Regel einer Aktion $a \in A$ durch

$$\forall X : \psi(\vec{X}) \wedge a(\vec{X}) \rightarrow \begin{cases} p_1 : \psi'_1(\vec{X}) \\ \dots \\ p_n : \psi'_n(\vec{X}) \end{cases}$$

mit $\vec{X}$ dem Vektor der Zustandsvariablen und dem Kontext ψ,der Formel des aktuellen Zustandes. $\psi'_1 ... \psi'_n$ sind die Ergebnisse der Aktion a. $p_1 ... p_n$ sind die Ausgangswahrscheinlichkeiten und ergeben zusammen 1.

Ein stochastisches *MDP* lässt sich als *Probabilistic STRIPS* beschreiben. Alle Aktionen von Regeln mit der gleichen Vorbedingung beschreiben die Menge Ap. Die Übergangsfunktion $P : S \times A \times S \to [0, 1]$ ergibt sich durch

$$P(s'|s, a) = \sum_{i=1}^{n} P(s'|\psi'_i, s, a)P(\psi'_i|s, a)$$

mit $P(s'|\psi'_i, s, a)$ der Wahrscheinlichkeit, dass das Ergebnis ψ'_i Zustand s' formuliert und $P(\psi'_i|s, a)$ der Wahrscheinlichkeit, dass Aktion a das Ergebnis ψ'_i erreicht. Soll eine Strategie π gefunden werden, muss noch eine Kostenfunktion hinzugefügt werden. Jedoch könnte auch eine Regel als reellwertiges Prädikat eine Belohnung $\Re$ erhalten, dann wäre die Belohnungsfunktion R aber nur über $S \times A$ und nicht über $S \times A \times S$ definiert.

2.3. Nebenläufigkeit in stochastischer Aktionsplanung

Viele Planungsvorgänge beinhalten nebenläufige Optimierung hierarchisch strukturierter Teilziele des Problems durch die dynamische Auswahl zuvor gelernter Strategien, welche die Teilziele optimieren. Die meisten Alltagsaufgaben beinhalten eine Struktur dieser Art. Beispielsweise ist beim Essen das Kauen eine Teilaufgabe, für welche jedoch das Essen zuerst zum Mund geführt werden muss. Allgemein ist die Lösung solcher Aufgaben eine Herausforderung, da Teilziele in Konflikt geraten können und mit den begrenzten Ressourcen umgegangen werden muss. Nebenläufigkeit in *MDPs* kann auf mehrere Arten verstanden werden. Zum einen können mehrere *MDPs* parallel ausgeführt werden, welche jeweils Teilziele lösen oder aber jeweils einen Effektor. Zum anderen kann ein *MDP* mehrere Aktionen parallel ausführen um die Effizienz zu erhöhen [RPMG04, S. 1137]. Selbstverständlich sind auch Kombinationen verschiedener Methoden denkbar.

2.3.1. Nebenläufigkeit in Markov-Entscheidungsprozessen

Nebenläufigkeit in *MDPs* bedeutet die Auswahl mehrerer Aktionen gleichzeitig. So muss im o.g. Beispiel des Essens nicht immer erst gewartet werden, bis das Kauen beendet ist um den nächsten Bissen auf die Gabel zu legen. Im Modell der *CoMDP - Concurrent Markov Decision Processes* wird das oben beschriebene Modell der *MPDs* erweitert, sodass mehrere Aktionen, sogenannte Multi-Aktionen $M \in \mathcal{P}(A)$, parallel ausgeführt werden können [MW08, S. 37f]. Die Eingabe eines *CoMDP* unterscheidet sich etwas von einem *MDP* $\texttt{CoMDP} = (S, A, Ap_{\|}, P_{\|}, R_{\|}, G, s0)$. Die Applikationsfunktion $Ap_{\|}$, die Übergangsfunktion

Tabelle 2.1.: Probabilistische *STRIPS*-Definition eines einfachen *MDP* mit potentieller Parallelisierung, mit den Zustandsvariablen x_1, x_2, x_3, x_4, p_{12} und dem Ziel $x_1 = 1$, $x_2 = 1$, $x_3 = 1$, $x_4 = 1$ [MW08, S. 33]

Aktionen	Vorbedingung	Effekt	Wahrscheinlichkeit
`toggle`(x_1)	$\neg p_{12}$	$x_1 \rightarrow \neg x_1$	1
`toggle`(x_2)	p_{12}	$x_2 \rightarrow \neg x_2$	1
`toggle`(x_3)	true	$x_3 \rightarrow \neg x_3$	0.9
		no change	0.1
`toggle`(x_4)	true	$x_4 \rightarrow \neg x_4$	0.9
		no change	0.1
`toggle`(p_{12})	true	$p_{12} \rightarrow \neg p_{12}$	1

.

$P_{\|}$ und die Belohnungsfunktion $R_{\|}$ beschreiben die Erweiterung von sequenzieller Ausführung einzelner Aktionen zur nebenläufiger Ausführung einer Menge von Aktionen. Eine Aktionskombination $\mathcal{P}(A) \subseteq A$ ist eine Potenzmenge der Aktionen A. Daraus folgen die neuen Eingaben:

- $Ap_{\|} : S \rightarrow \mathcal{P}(\mathcal{P}(A))$ definiert die neue Applikationsfunktion. Sie beschreibt die Menge der Aktionskombinationen, welche je Zustand $s \in S$ angewendet werden können.
- $P_{\|} : S \times \mathcal{P}(A) \times S \rightarrow [0, 1]$ ist die Übergangsfunktion. Sie beschreibt die Übergangswahscheinlichkeit P von $s \in S$ nach $s' \in S$ unter $\mathcal{P}(A)$.
- $R_{\|} : S \times \mathcal{P}(A) \times S \rightarrow \Re$ ist die Belohnungsfunktion. Sie beschreibt die Belohnung $\Re$ von $s \in S$ nach $s' \in S$ unter $\mathcal{P}(A)$.

Sich ausschließende Aktionen

Es muss berücksichtigt werden, dass bei der gleichzeitigen Ausführung von Aktionen eventuell Konflikte auftreten können. Einerseits können zwei Aktionen nicht gleichzeitig ausgeführt werden, wenn deren Vorbedingungen sich widersprechen. Zusätzlich werden im Modell der *CoMDP* zwei weitere Kriterien genannt, welche die gleichzeitige Ausführung verhindern, dass es nicht zu Konflikten kommt. Wenn der Effekt einer Aktion den Effekt einer anderen beeinflusst, können sie nicht gleichzeitig gestartet werden. Auch nicht, wenn ein Effekt die Vorbedingung der anderen beeinflusst. Beispielsweise haben in Tabelle 2.1 `toggle`(x_1) und `toggle`(x_2) einen Konflikt in der Vorbedingung und der Effekt von `toggle`(p_{12}) bedingt die Vorbedingung von `toggle`(x_1). Dagegen können `toggle`(x_1), `toggle`(x_3) und `toggle`(x_4) parallel ausgeführt werden. Am Beispiel des Essens kann der Bissen erst in den Mund geschoben werden, wenn dieser leer ist.

2.3.2. Koartikulierende Markov Entscheidungsprozesse

Um die Variabilität weiter zu steigern, kann ein Problem auf mehrere *MDPs* aufgeteilt werden. Das Konzept der koartikulierenden *MDP* ist ein Ansatz, welcher mehrere *MDPs* jeweils ein Teilziel lösen lässt um das globale Ziel zu erreichen [SPS99, S. 189]. Der Begriff der Koartikulation, nach dem Prinzip der Lautbildung in der Phonetik unterteilt eine Aufgabe in Teilziele, welche hierarchisch strukturiert abgearbeitet werden. Im o.g. Beispiel des Essens wären die Teilziele Kauen und Essen zum Mund führen. Selbstverständlich muss das Essen zuerst zum Mund geführt werden um gekaut werden zu können. Während des Kauens jedoch kann schon der nächste Bissen vorbereitet werden. So ist die koartikulierende Ausführung beider Teilziele effizienter als die parallele.

Es wird angenommen, dass der Planungsalgorithmus Zugriff auf Controller $C = \{C_1, C_2, ..., C_n\}$ hat, welche jeweils ein Teilziel $\omega_i \in \Omega$, mit Ω einer hierarchisch strukturierten Menge der Teilziele lösen [RPMG04, S. 1137ff]. Die übergeordnete Aufgabe wird gelöst, wenn konkurrierende Teilziele abgearbeitet werden. Ein Controller C modelliert eine Menge von Optionen $\langle I, \pi, \beta \rangle$ mit der Initiationsmenge $I \subseteq S$, der Strategie $\pi : S \times A \to [0, 1]$ und der Terminierungsbedingung $\beta : S^+ \to [0, 1]$ über einem *SMDP - Semi-MDP* $M_c : \{S_c, A_c, P_c, R_c\}$ mit $S_c \subseteq S$, $A_c \subseteq A$ der Übergangswahrscheinlichkeit P_c und der Kostenfunktion R_c. Eine Option kann in einem Zustand $s \in I$ gestartet werden und löst mit π ein Teilziel. β wird 1, wenn das Teilziel erfolgreich abgearbeitet wurde.

Jede Aktionsauswahl ändert den Zustand in allen *MDPs*. Die einzelnen Controller haben Zugriff auf mehrere Optionen, die jeweils eine Teilaufgabe nahezu optimal lösen. Dadurch entsteht eine gewisse Flexibilität und dem Planer stehen je Zustand s verschiedene Aktionen a zur Verfügung. Ziel ist es nun eine globale Strategie zu finden, welche die Teilziele schrittweise abarbeitet und Aktionen auswählt, welche einen Kompromiss bilden und in allen *SMDPs* den Teilzielen $\omega \subseteq \Omega$ näherkommt. Zusätzlich sind Start- und Terminierungsbedingungen der einzelnen *SMDPs* gegeben. Diese beeinflussen die Aktionsauswahl, welche entsprechend einen Controller C startet oder beendet.

2.4. Das Problem mit der Zeit

Bisher wurde angenommen, dass alle Aktionen in einem Markov-Schritt abgearbeitet werden. Das bedeutet sie haben keine eigentliche Dauer und die Aktionsausführung geschieht unmittelbar. In der Realität ist dies leider meist nicht der Fall. Folgend werden zwei Methoden vorgestellt, welche die Gleichzeitigkeit von Aktionen um eine Zeitdauer erweitern. Das *CAM - Concurrent Action Model* formuliert die Dauer von Aktionen als diskrete Zeitschritte und *RAP - Relational Activity Processes* erweitert dieses Modell um stochastische und reellwertige Zeit, indem es Aktionen um in das Starten und Beenden dieser Aufteilt. So wird eine Verbindung zwischen Markov-Schritten und verstrichener Zeit erzeugt.

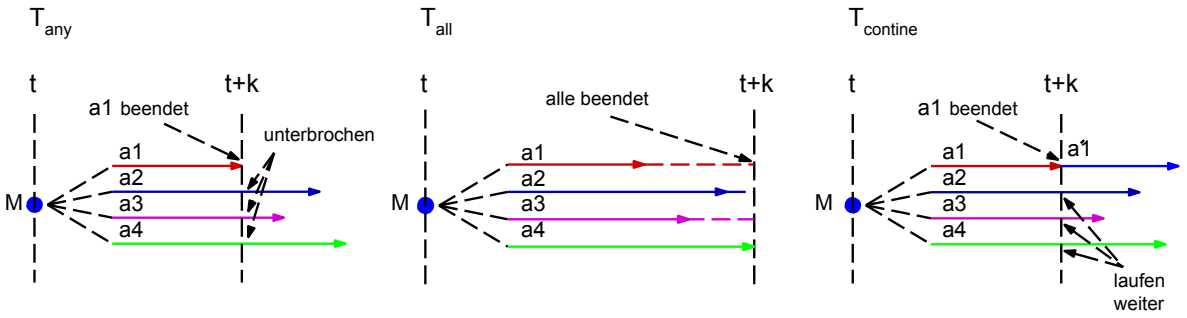


Abbildung 2.3.: Terminierungskriterien T_{any}, T_{all} und $T_{continue}$ für die Multi-Aktion $M = \{a1, a2, a3, a4\}$ [RM02, S. 1620].

2.4.1. CAM - Concurrent Aktion Modell

Im *CAM - Concurrent Action Modell* [RM02, S. 1619ff] werden, um der Problematik der unterschiedlichen Dauer verschiedener Aktionen zu begegnen, die Terminierungsschemata T_{any}, T_{all} und $T_{continue}$ eingeführt. Eine Multi-Aktion $M \in \mathcal{P}(A)$ wird mit solch einem Schema gekennzeichnet um zu spezifizieren, was passieren soll, wenn eine der Teilaktionen terminiert

T_{any} bedeutet, dass alle Aktionen beendet werden, wenn eine terminiert.

T_{all} bedeutet, dass alle Aktionen weiterlaufen, bis die letzte terminiert. Die fertigen Aktionen verweilen in einem Idle-Zustand.

$T_{continue}$ bedeutet, dass nach der Terminierung einer, eine neue Einzelaktion gestartet werden kann, welche die anderen nicht beeinflusst.

(S. Abbildung 2.3).

Dadurch können Multi-Aktionen eine diskrete Dauer haben und der Aktionsplaner kann entsprechend der Schemata eine Strategie finden, welche Aktionen mit unterschiedlicher Länge parallel ausführen kann. Am Beispiel des Essens, welches aus drei Aktionen, dem Kauen, dem Essen zum Mund führen und dem Beladen der Gabel besteht, könnte eine Strategie folgendermaßen aussehen:

```
while(!plateEmty){
        T_all(kauen(), gabelFüllen())
        essenZumMund()
      }
```

2.4.2. RAP - Relational Activity Processes

RAP - Relational Activity Processes erweitern das *CAM* um stochastische reellwertige Zeit [TMM+15, S. 2f]. Es ist ein Modell eines *SMDP* für eine Aktionsplanung mit nebenläufigen Aktionen. Der Kern dieses Modells ist die Aufteilung dauernder Aktionen in Initiations- und Terminierungsoperatoren, welche jeweils unmittelbar geschehen. Die Aktionen werden so Teil der Zustandsmenge. Nebenläufigkeit wird durch das sequentielle Starten oder Beenden von Aktionen durch diese Operatoren erreicht, anstatt der parallelen Ausführung von Multi-Aktionen. Dadurch können die diskreten Zeitschritte, wie in *CAM*, zu kontinuierlichen erweitert werden. Jede Aktion erhält eine deterministische oder stochastische reellwertige Dauer, welche durch Prädikate in der Zustandsmenge definiert wird. Zudem wird noch ein `wait`-Operator eingeführt, welcher den zeitlichen Ablauf koordiniert.

Gegeben ist eine *relationale Repräsentation* des Weltmodells mit den Prädikaten und Symbolen. Ein *RAP* wird durch ein *sMDP* $\{S, A, D, P, T, R\}$ definiert mit

S, einer endlichen Menge diskreter Zustände.

A, einer endlichen Menge der Aktionen.

$D(s)$, einer endlichen Menge der möglichen Entscheidungen d je Zustand s.

$P : S \times D \times S \to [0, 1]$, der Übergangsfunktion. $P(s'|s, d)$ ist die Wahrscheinlichkeit Zustand s' zu erreichen, wenn Entscheidung d in Zustand s ausgeführt wird.

$T : S \times D \times S \to \mathbb{R}$, dem Laufzeitmodell. $(s, d, s') \to \tau$ ist die Dauer eines Markov-Schrittes.

$R : S \times A \times T \times S \to \Re$, der Kostenfunktion. $(s, d, \tau, s') \to r$ beschreibt die Kosten für eine Entscheidung respektive der Dauer.

Die Menge der Entscheidungen D ergibt sich aus der Menge der Aktionen A. Der Aktionsplaner kann aus dieser Menge auswählen. Für jede Aktion $a \in A$ gibt es einen oder mehrere Initiierungs- und Terminierungsoperatoren $\mathtt{o}_{init}$ und $\mathtt{o}_{term}$, welche eine Aktion startet oder beendet mit

$$\mathtt{o}_{init}(a, \vec{X}) : \mathtt{pre}_{init}(a, \vec{X}) \to \mathtt{go}(a, \vec{X}) = \tau_a, \mathtt{post}_{init}(a, \vec{X})$$
$$\mathtt{o}_{term}(a, \vec{X}) : \mathtt{pre}_{term}(a, \vec{X}) \to \neg\mathtt{go}(a, \vec{X}), \mathtt{post}_{term}(a, \vec{X})$$

$\vec{X}$ ist ein Vektor aus Zustandsvariablen, $\mathtt{pre}_x(a, \vec{X})$ eine Vorbedingung wie in *STRIPS*, $\mathtt{go}(a, \vec{X}) \to \mathbb{R}$ ein spezielles reellwertiges Prädikat, welches die Ausführung der Aktion beschreibt mit $\tau_{a,\vec{X}}$ der erwarteten Dauer. $\mathtt{post}_x(a, \vec{X})$ beschreibt den Effekt auf die Zustandsmenge durch die Veränderung der Prädikatenmenge. Zu beachten ist das $\mathtt{go}(a, \vec{X})$ Prädikat, welches durch $\mathtt{o}_{init}$ erzeugt und durch $\mathtt{o}_{term}$ wieder entfernt wird.

Beispielsweise könnte ein Initiierungsoperator $\texttt{o}_{init}(\texttt{pickup}(objA, objB))$ für das Aufheben eines Objektes $objA$ von einem Objekt $objB$ durch den Agent wie folgt aussehen:

$$\begin{aligned}&\texttt{o}_{init}(\texttt{pickup}(objA, objB)) :\\ &\texttt{go}(\texttt{pickup}(objA, objB))!, \texttt{inhandNil}, \texttt{on}(objA, objB), \texttt{busy}(objA)!\\ &\rightarrow \texttt{go}(\texttt{pickup}(objA, objB)) = 2.5, \texttt{busy}(objA), \texttt{busy}(objB)\end{aligned}$$

Er hat die Vorbedingungen, dass kein go-Prädikat für diese Aktion existiert ($\texttt{go}(\texttt{pickup}(objA, objB))!$), die Roboterhand leer ist (inhandNil), das Objekt $objA$ auf dem Objekt $objB$ liegt ($\texttt{on}(objA, objB)$) und dass $objA$ nicht in Benutzung ist ($\texttt{busy}(objA)$). Die Effekte erzeugen ein go-Prädikat mit der Zeit 2.5 und erzeugen Prädikaten, welche repräsentieren, dass $objA$ und $objB$ in Verwendung sind. Der entsprechende Terminierungoperator $\texttt{o}_{term}(\texttt{pickup}(objA, objB)$ könnte wie folgt aussehen:

$$\begin{aligned}&\texttt{o}_{term}(\texttt{pickup}(objA, objB)) :\\ &\texttt{go}(\texttt{pickup}(objA, objB)) = 0, \texttt{busy}(objA)\\ &\rightarrow \neg\texttt{go}(\texttt{pickup}(objA, objB)), \neg\texttt{busy}(objB), \neg\texttt{inhandNil}, \texttt{inhand}(objA)\end{aligned}$$

Wird eine Entscheidung $d \in D(s)$ im Zustand s getroffen, wird ein Zwischenzustand $\hat{s} = \texttt{post}_{x,a}(\vec{x}) \circ s$ erzeugt, welcher die Effekte des Operators auf s anwendet. In diesem Zustand wird durch Vorwärtsverkettung aller Prädikate bis zur Konvergenz ein stabiler Zustand s' erreicht. Entsprechend müssen für alle Vorbedingungen $\texttt{pre}_x(a, \vec{X})$ die Effekte $\texttt{post}_x(a, \vec{X})$ implizit vorliegen.

Zusätzlich ist der Entscheidungsmenge $D(s)$ noch der wait-Operator hinzugefügt. Er regelt den zeitlichen Prozess und ist quasi die Schnittstelle zwischen einem Markov-Schritt und der realen Zeit. Wird er ausgeführt, sind alle Operatoren ausgeführt und es verstreicht Zeit bis zur nächsten Beendigung einer Aktion. Konkret ist er definiert durch:

1. Finde das go-prädikat mit der kleinsten Restzeit τ_{min}.
2. Verringere alle go-prädikate um τ_{min}.
3. Entferne alle go-prädikate $\texttt{go}(a, \vec{x})$ mit $\tau = 0$ aus $\hat{s}$ und füge entsprechende Terminierungsoperatoren $\texttt{post}_x(a, \vec{x})$ ein.

Das definiert den Zwischenzustand $\hat{s}$. Der stabile Zustand s' wird wieder durch Vorwärtsverkettung der vorhandenen Prädikate erreicht. Hat man anstatt der deterministischen Dauer von Aktionen eine stochastische $P(\tau_{a,\vec{x}}|s, a, \vec{x})$, wird der wait-operator etwas anders definiert:

1. Nimm $\tau_{a,\vec{x}} \sim P(\tau_{a,\vec{x}}|s, a, \vec{x})$ für alle aktiven Aktionen.
2. Wähle das kleinste τ_{min}.

3. Entferne alle go-prädikate $\texttt{go}(a, \vec{x})$ mit $\tau = \tau_{min}$ aus $\hat{s}$ und füge entsprechende Terminierungsoperatoren $\texttt{post}_x(a, \vec{x})$ ein.
4. Für Alle $\tau_{a,\vec{x}} > \tau_{min}$ verringere den Erwartungswert $E(\tau_{a,\vec{x}}|s, a, \vec{x})$ aller $\texttt{go}(a, \vec{x})$ um τ_{min}

Das Laufzeitmodell $T : (s, d, s') \rightarrow \tau$ ist entsprechend des `wait`-operators τ_{min}, da angenommen wird, dass für die Initiierungs- und Terminierungsoperatoren keine Zeit verstreicht. Somit hängt es implizit von den `go`-Prädikaten in $\texttt{o}_{init}$ ab.

2.5. Vergleich der verschiedenen Modelle

Ausgehend von einem *MDP* wurden verschiedene Erweiterungen zur Parallelisierung beschrieben, zudem wurde das Konzept der Koartikulation gezeigt (S. Abbildung 2.4). Im Folgenden sollen die Eigenschaften der verschiedenen *MDP*-Modelle verglichen werden.

Ein Grundproblem von nebenläufigen *MDPs* sind die begrenzten Ressourcen eines Systems, so kann eine Multi-Aktion M nur ausgeführt werden, wenn die einzelnen Aktionen sich nicht widersprechen bzw. nicht in einen Ressourcenkonflikt geraten. Um sich ausschließende Bedingung wie in *CoMDP* auch auf *STRIPS* oder *RAP* anzuwenden, können diese in den Vorbedingungen als negative Literale formuliert werden.

Eine weitere Schwierigkeit ist die Erweiterung von Multi-Aktionen auf mehr als einen Markov-Schritt um zu formalisieren, sodass einzelne Aktionen unterschiedliche Dauer haben können. Hierzu wurden in *CAM* die verschiedenen Terminierungsschemata eingeführt. Jedoch formulieren diese die verstrichene Zeit immer noch als diskrete Zeitschritte. Um von der diskreten Zeit zur kontinuierlichen Zeit zu kommen, wurde schließlich in *RAP* der `Wait`-Operator eingeführt, welcher durch ein Laufzeitmodell zwischen zwei Markov-Schritten eine reellwertige Zeit verstreichen lässt. Diese entspricht der minimalen erwarteten Dauer einer der aktiven Aktionen. Die verschiedenen Terminierungsschemata von *CAM* können von *RAP* simuliert werden. T_{any} entspricht der Entfernung aller `go`-Prädikate bei der Ausführung eines `wait`-Operators. $T_{continue}$ wird einfach durch das Weiterlaufen aller anderen `go`-Prädikate reproduziert. T_{all} wird durch die Einführung eines `blocked`-Prädikats erreicht, welches die Initiationsbedingungen undurchführbar macht, solange nach einem `wait` noch Aktionen vorhanden sind.

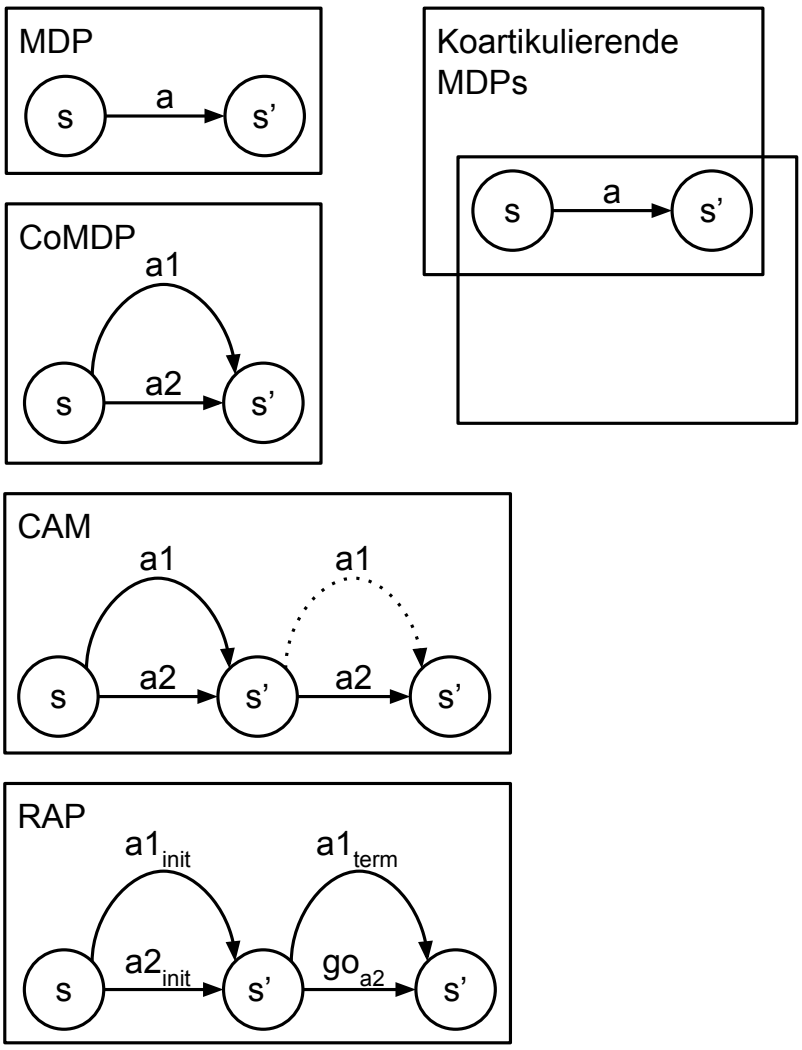


Abbildung 2.4.: Übersicht über die Funktionsweise der verschiedenen *MDP*-Modelle. *MDP*, *CoMDP*, *CAM*, *RAP* und *Koartikulierende MDP*.

3. Frameworks zur komplexen Aktionsplanung

Um vielschichtiges und komplexes Roboterverhalten zu erzeugen ist ein Paradigmenwechsel nötig um Aktionen auf einer intuitiv verständlichen Ebene zu formulieren. Jedoch erfordert diese auch flexible Softwareumgebungen, welche die Implementierung von anspruchsvollen Bewegungen auf einfache Weise ermöglicht. In Kapitel 2 wurden theoretische Modelle erörtert, welche verschiedene Probleme der Aktionsplanung geschickt formalisieren. So soll nun eine Softwareumgebung zur Aktionsplanung gezeigt werden, welche diese Probleme löst und praktisch umsetzt.

Zuerst wird in Kapitel 3.1 das Paradigma der kognitiven Robotik beschrieben. Es bedeutet die Formulierung und Lösung einer Manipulationsaufgabe auf der Ebene der Objektmanipulation. Darauf wird in Kapitel 3.2 die Softwareumgebung *CRAM* erörtert, welche eine Verhaltenserzeugung nach diesem Prinzip erlaubt. Schließlich wir in Kapitel 3.3 die Mächtigkeit von *CRAM* diskutiert und inwiefern es ein *MDP* implementiert.

3.1. Kognitive Robotik

Um in der Robotik immer schwieriger werdende Aufgaben lösen zu können, ist ein Paradigmenwechsel vom Stellen von Gelenkwinkeln oder Effektorpositionen zu der direkten Manipulation äußerer Freiheitsgrade notwendig. So kann die Aktionsplanung direkt mit Objekten interagieren ohne sich mit den mechanischen, inneren Prozessen beschäftigen zu müssen. Der Begriff der kognitiven Robotik fasst dieses Prinzip zusammen. Er unterscheidet Regelung auf motorischer Ebene von der Steuerung mittels abstrakter Aktionen [TLJ13, S. 1f]. Um noch einen Schritt weiter zu gehen hat ein kognitives System entsprechend auch kognitive Fähigkeiten und *weiß, was es tut*. Schließlich geht die Entwicklung in der Robotik in eine Richtung, welche genau diese Fähigkeit anstrebt [BSR+08, S. 1].

Es ergeben sich jedoch einige Schwierigkeiten für die Interaktion eines Agenten mit der Umwelt. Der Zustandsraum äußerer Freiheitsgrade ist komplex und nur schwer zu formalisieren. Auch können, im Gegensatz zur direkten Regelung von Stellwinkeln, externe Freiheitsgrade nur indirekt gesteuert werden. Beispielsweise kann ein Objekt nur bewegt werden werden, nachdem es gegriffen wurde. So ergibt sich die Notwendigkeit hierarchischer

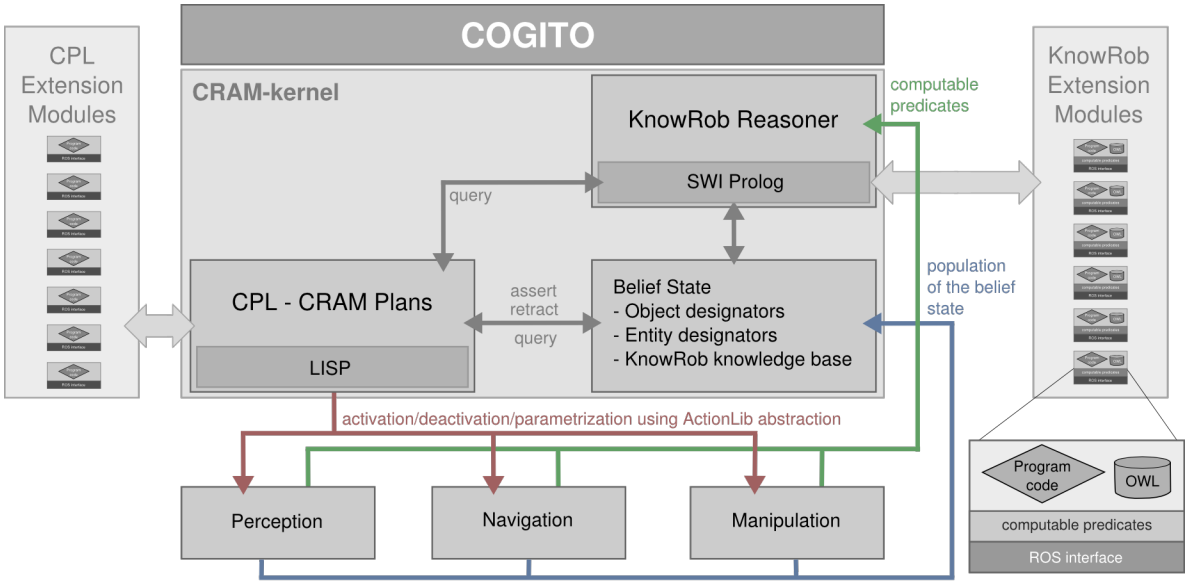


Abbildung 3.1.: Die Struktur von *CRAM*. Der Kernel besteht aus CPL und KnowRob [BMTR12, S. 3].

Steuerungsmodelle, welche sequenziell Bewegungsroutinen abarbeiten können. Zudem ist die Abbildung der fundamentalen physikalischen und geometrischen Randbedingungen in Wahrscheinlichkeitsmodellen erforderlich, jedoch immer ein Kompromiss, da nur Teilaspekte formuliert werden können. Schließlich soll ein Agent Aufgaben in Szenarien lösen können, für die er nicht explizit trainiert wurde. So sollte ein kognitives System immer von einer gewissen Flexibilität und Resilienz gekennzeichnet sein [VMS07, S. 151].

3.2. CRAM - A Cognitive Robot Abstract Machine

Die Erzeugung eines kognitiven Systems ist eine anspruchsvolle Programmieraufgabe und erfordert spezialisierte Software-Tools. *CRAM - A Cognitive Robot Abstract Machine* ist ein Framework für das Design, der Entwicklung und der Implementierung der Aktionsplanung autonomer kognitiver Roboter [BMTR12, S. 1]. Es erweitert Steuerungssysteme wie *ROS* [ros, S.] oder *Player* [G$^+$, S.], die die rudimentären Prozesse des Roboters regeln, welchen es jedoch an mächtigen Softwarewerkzeugen zur Verhaltenserzeugung fehlt. *CRAM* beinhaltet Datenstrukturen, einfache Kontrollstrukturen, Tools und Bibliotheken, welche speziell für die kognitive Robotersteuerung entwickelt wurden. Es ermöglicht die Erzeugung von komplexen Kontrollprogrammen, welche Entscheidungen auf Grund der gemachten Wahrnehmung und der gewonnenen Erkenntnisse treffen. Im Kern besteht *CRAM* aus der *CPL - CRAM Plan Language* und dem Wissensverarbeitungssystem *KnowRob* (S. Abbildung 3.1).

Algorithmus 3.1 Beispiel-*CRAM*-plan zum Aufheben des Objektes `objA`

```
(def-goal (achieve (object-in-hand objA))
  (with-designators
    (pickup-place ...)
    (grasp-type ...)
    (pickup-reaching-traj ...)
    (lift-trajectory ...)
    (when (and (holds-bel (object-in-hand currObj) now) (obj-equal currObj objA))
      (succeed (object-in-hand objA))
    )
    (at-location pickup-place
      (achieve (arm-at pickup-reaching-traj))
      (achieve (grasped grasp-type))
      (achieve (arm-at lift-trajectory))
      (succeed (object-in-hand objA))
    )
  )
)
```

3.2.1. CRAM Plans

Der Entwickler erstellt mit *CRAM* sog. Pläne, die ein Ziel verfolgen. Teilziele oder andere Befehle werden mit dem Weltzustand als Argument definiert. Der Algorithmus 3.1 zeigt einen Beispielplan zum Aufheben des Objektes `objA`.

Zu beachten ist die Formulierung des zu erreichenden Zieles durch den gewünschten Zustand `(object-in-hand objA)` anstatt der Formulierung als Funktionsbeschreibung `pick-up objA`. So kann diese Art von Zustandsbeschreibung auch für andere Routinen verwendet werden, wie der Wahrnehmung `(perceive(object-in-hand objA))`. Dadurch kann der Roboter einen Zustand überprüfen, bevor er erreicht werden soll. Ein zweiter wichtiger Aspekt ist die Formulierung verschiedenster Kontrollsysteme wie der Pfadplanung als Objekte erster Klasse, sodass Entscheidungen darüber getroffen werden können.

3.2.2. CPL - CRAM Plan Language

CPL ist eine Sprache zur Verhaltenserzeugung von autonomen Robotersystemen. Sie ermöglicht nicht nur das Ausführen von, sondern auch das Schließen über die Kontrollprogramme und die automatische Manipulation dieser. Dies wird durch die symbolische Repräsentation der Schlüsselaspekte der Kontrollprogramms ermöglicht. So kann die Steuerung automatisch erkennen, warum es gescheitert ist oder wann es falsche Erkenntnisse über die Welt hat und diese entsprechend anpassen. *CPL* beinhaltet einige Low-Level Kontrollstrukturen, welche die gleichzeitige Ausführung verschiedener Aktivitäten ermöglichen.

Tabelle 3.1.: Die CPL Kontrollstrukturen und ihre Verwendung [BMTR12, S. 4]

Kontrollstruktur	Beispielanwendung
in parallel do $p_1 ... p_n$	**in parallel do** $navigate([235, 468])$ $buildGridMap()$
try in parallel $p_1 ... p_n$	**try in parallel** $detectDoorWithLaser()$ $detectDoorWithCamera()$
with constraining plan $p\ b$	**with constraining plan** $relocalizeIfNec()$ $deliverMail()$
Plan with name $N_1\ p_1$ ... **with name** $N_n\ p_n$ **order** $n_i < n_j$	**plan with name** $S_1\ putOnTable(C)$ **with name** $S_2\ putOn(A, B)$ **with name** $S_3\ putOn(B, C)$ **order** $S_1 < S_3$ $S_3 < S_2$

in parallel do führt eine Menge von Teilplänen aus. Es schließt erfolgreich, wenn alle Teilpläne erfolgreich sind. Es schlägt fehl, wenn einer der Teilpläne scheitert.

try in parallel führt eine Menge von Teilplänen aus und schließt erfolgreich ab, wenn einer der Teilpläne erfolgreich ist.

with contraining plan führt eine Aktivität aus, welche durch eine zweite beschränkt ist.

plan führt Aktivitäten parallel aus, es sei denn diese beschränken sich gegenseitig, dann werden sie in einer gegebenen Reihenfolge abgearbeitet.

(S. Tabelle 3.1)

3.2.3. KnowRob

Dem gegenüber steht die Wissensverarbeitung *KnowRob* [TB09, S. 4261f]. Sie wurde speziell für die Sprache *CPL* entwickelt und repräsentiert Wissen erster Ordnung. *KnowRob* integriert enzyklopädisches Wissen, ein Umgebungsmodell, aktionsbasiertes Schließen und menschliche Beobachtungen und erlaubt den Zugang zu all dieser Information auf einheitliche, symbolische Weise. Sie stellt Werkzeuge zu aktionsbezogener Wissensrepräsentation zur Verfügung, welche automatisch durch Beobachtung und Erfahrung akquiriert wird. *KnowRob* ist darüber hinaus in der Lage mit der stochastischen und unsicheren Eigenschaft der Realität umzugehen und auf Abfragen effizient zu reagieren (S. Abbildung 3.2).. Sie stellt somit die Grundlage zum logischen Schließen erster Ordnung dar und kann so nach Wahrheitswerden befragt werden.

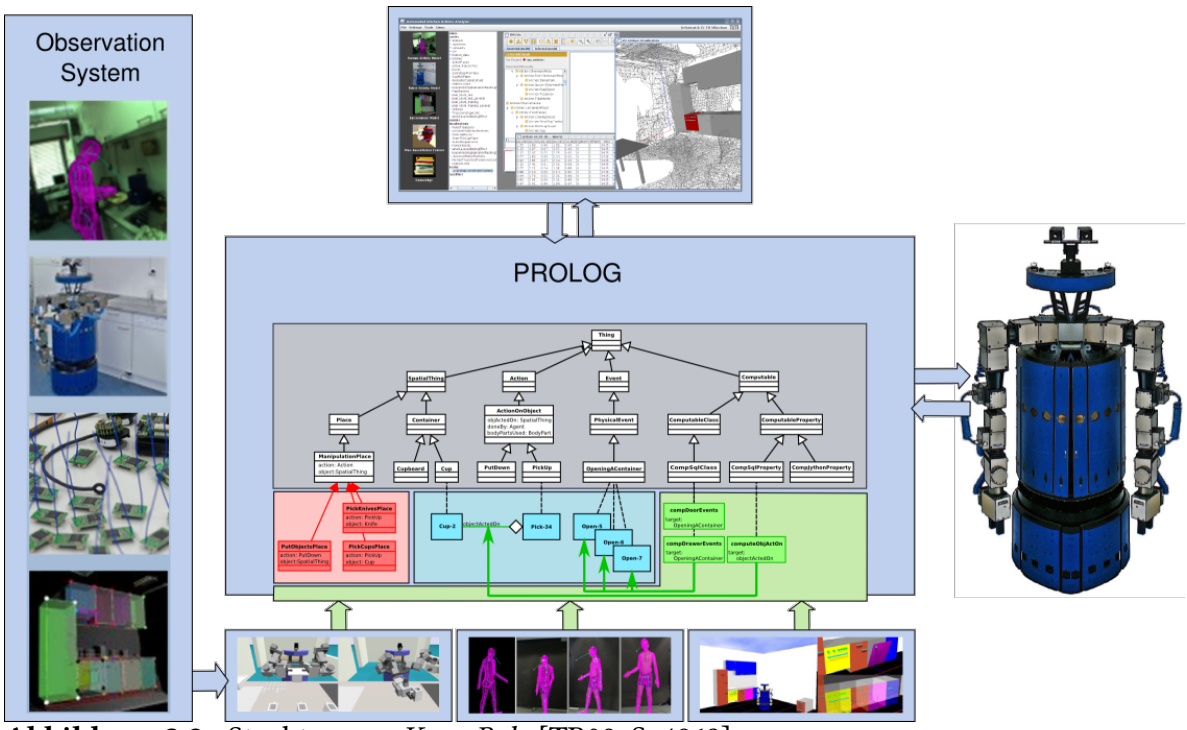


Abbildung 3.2.: Struktur von *KnowRob*. [TB09, S. 4263]

3.2.4. Cognito

Cognito erlaubt die Untersuchung der Ausführung des *CRAM-Kernel*. Der Kernel realisiert Pläne und untersucht die Wahrnehmung der Umwelt, jedoch untersucht *Cognito* die Ausführung der Pläne. So kann untersucht werden, ob diese erfolgreich ausgeführt wurden oder ob Probleme auftraten. *CPL* erzeugt für alle Pläne Objekte, welche diese repräsentieren. *Cognito* kann über diese Objekte schließen.

3.2.5. Erweiterungsmodule

Zusätzlich verfügt *CRAM* über einige Erweiterungsmodule, welche die Funktionalität der *CPL* und von *KnowRob* erweitern. Diese könnten verbesserte Wahrnehmung und Objekterkennung sein. Auch werden so verbesserte Lernalgorithmen hinzugefügt wie die Integration der *RoLL - Robot Learning Language*.

Algorithmus 3.2 Beispiel eines *RoLL*-Kontrollprogramms

```
do-continuously
  do-in-parallel acquire-experiences re
              excecute top-level plan
  learn lp
```

3.2.6. RoLL - Robot Learning Language

Die *RoLL - Robot Learning Language* basiert auf dem Konzept von hierarchischen hybriden Automaten um eine explizite Spezifizierung von Lernproblemen zu ermöglichen [Kir09, S. 2ff]. *RoLL* versucht sich kontinuierlich verbessernde Roboter zu ermöglichen indem sie das Lernen als zentrales Konzept in die Programmiersprache integriert. Ein typisches Lernproblem wird spezifiziert und gelöst in zwei Schritten. Zuerst wird die nötige Erfahrung akquiriert, dann wird diese zum Lernen und zur Verbesserung des Kontrollprogramms verwendet. Ein mögliches Programm könnte aussehen wie in Algorithmus 3.2.

Dieses Programm führt einen **top-level plan** parallel zur Akquirierung von Erfahrung aus. Nach dem Ende der Ausführung wird die gewonnene Erfahrung für einen Lernprozess verwendet und vor der nächsten Ausführung in das Programm integriert.

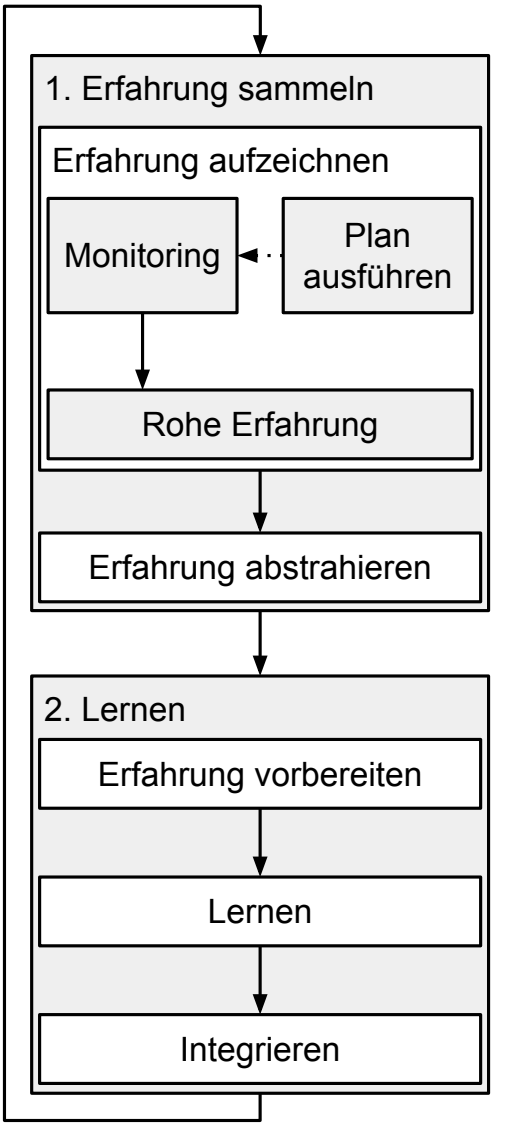
Eine **Erfahrung** ist mehr als die gewonnenen Rohdaten einer Episode. Es werden nur notwendige Parameter zum Lernen aufgezeichnet: Zustandsübergänge, Steuerungskommandos, interne Annahmen, Entscheidungen sowie Fehler und entsprechende Reaktion. Die gewonnene Erfahrung wird dann abstrahiert und in einer Datenbank für das Offline-Lernen gespeichert oder direkt verwendet. Aus der rohen Erfahrung wird durch Weiterverarbeitung der Daten eine Abstraktion der Erfahrung erreicht, welche für den nächsten Schritt, das Lernen verwendet werden kann.

Das **Lernen** erfolgt durch die Transformation der Daten in ein Format, welches auf den gewünschten Lernalgorithmus passt. Danach wird der Algorithmus ausgeführt und schließlich werden die Ergebnisse in das Kontrollprogramm integriert. Dann startet der Prozess von neuem. (S. Abbildung 3.3).

3.3. Diskussion von CRAM

3.3.1. CRAM als MDP

Ein *CRAM*-Plan hat ein durch einen Zustand definiertes Ziel. Es definiert eine Menge von Aktionen über einem *MDP*, welche in einem Zielzustand konvergieren. Eine komplexe Aufgabe hat verschiedene Zwischenschritte, welche hierarchisch gelöst werden. Die Kontrollstrukturen der *CPL* lassen Parallelisierung zu. Beispielsweise ist `in parallel do` vergleichbar mit




```
raw experience: re
   observe execution of
plan p
   recording variables
      (x,y) continuously
      (a,b,c) in case of
success

abstract experience: ae
   of class aec
   abstract from raw
experience re

learning problem: lp
   learn target function
      specified as lps
   using experience of
class lec
      abstracted from
experience ae
   applying learning system
ls
```

Abbildung 3.3.: Lernprozess von *RoLL*. Auf der rechten Seite der entsprechende Quelltext, welcher den Lernprozess definiert. [Kir09, S. 5]

dem Terminierungsschema T_{all} von *CAM* zur Mult-Aktionsausführung. `try in parallel` implementiert T_{any}. Gleichzeitig können mit `plan` wie in *CoMDP* Aktionen parallel ausgeführt werden, welche gegenseitig nicht in Konflikt geraten, sonst werden diese entsprechend sequenziell ausgeführt.

3.3.2. Mächtigkeit von CRAM

CRAM regelt nicht die unterliegenden Prozesse eines Roboters, sondern stellt nur die Verbindung zwischen dem Agenten, der Planerzeugung und weiterer Software durch die Erweiterungsmodule dar. Durch die Erweiterung durch *RoLL* beispielsweise lässt sich die Performance des Planungssystems steigern. Es ist auch denkbar, dass durch *Cognito* gemachte Beobachtungen über Fehlverhalten des Plans gelernt werden kann und dieser angepasst wird um Fehler zu vermeiden. So werden neben der Wissensverarbeitung und der Planerstellung mittels Prädikaten auch Lernmethoden integriert, dass durch *Cognito* über eigenes (Fehl-)Verhalten gelernt werden kann. Letztlich versucht *CRAM* durch flexible und einfache Formulierung komplexer Steuerungseinheiten und der Integration verschiedener Module ein Werkzeug zu sein um ein kognitives System zu entwickeln.

4. Implementierung autonomer Aktionen als Activities

Im Rahmen dieser Arbeit wird an der Implementierung eines Interfaces für Aktionen gearbeitet welches die Verbindung zwischen dem relationalen Planen und Lernen und den eigentlichen Roboteraktionen bildet der *RM - Relational Machine* [Tou15, S. 1f]. Die Implementierung der Aktionen wird folgend *Activities* genannt, um diese von der vorangegangenen abstrakten Formulierung von Aktionen zu unterscheiden. Der Kern dieser Arbeit ist die Untersuchung, welche Eigenschaften notwendig sind, um das Ausführen dieser *Activities* robust und sicher zu gestalten und wie die Formulierung allgemein gehalten werden kann um höchste Flexibilität bei der Implementierung unterschiedlichster *Activities* zu erreichen.

Das folgende Kapitel beschreibt die Implementierung eines Frameworks, welches ein Interface implementiert, das verschiedenste Aktionen eines Agenten Planungsalgorithmen zur Verfügung stellt. Zuerst wird in Kapitel 4.1 die *Relational Machine*, das der Arbeit zugrundeliegende Framework vorgestellt, dann werden in Kapitel 4.2 verschiedene mögliche Aktionstypen, welche dem Aktionsplaner zur Verfügung stehen, beschrieben. In Kapitel 4.3 wird die Implementierung von autonomen Aktionen als *Activities* dokumentiert. Darauf wird in Kapitel 4.4 die Ausnahmebehandlung als Element der Aktionsplanung als Teil der Implementierung des Frameworks erörtert. Schließlich wird in Kapitel 4.5 umfangreich die Implementierung einer Beispiel-*Activity* am Beispiel des Pfadplaners *KOMO* gezeigt.

4.1. RM - Relational Machine

Das der Arbeit zugrundeliegende Interface zwischen Planungsmethoden und den Aktionskontrolle eines Roboters, die *RM - Relational Machine* implementiert ein Interface, welches die Schnittstelle zwischen dem Aktionsplaner und der eigentlichen Robotersteuerung darstellt [Tou15, S. 1f]. In einer dieser Arbeit vorausgegangenen Studienarbeit wurde dieses Framework um ein Interface erweitert, was die interaktive Verhaltenserzeugung durch einen Benutzer ermöglicht [Bö15, Vgl.]. Die *RM* repräsentiert zum einen die sequenzielle und parallele Ausführung von Aktionen, hier *Activities* genannt in einer Weise, dass diese den Formalismen von Lern- und Planungsmethoden entsprechen. Zum anderen stellt sie ein flexibles Framework dar, welches es ermöglicht auf einfache Weise manuell Roboterverhalten zu erzeugen.

4.1.1. Struktur der RM

Formal ergibt sich die *RM* aus folgenden Mengen.

- Der **Relationale Zustand** ist eine Menge aktiver Fakten, den *Facts*. Die Konjunktion dieser bilden den aktuellen Zustand. *Facts* sind Prädikate erster Ordnung und stellen Informationen über die Um- und Roboterwelt dar. Diese können Annahmen, Sensor- oder Aktionsinformationen, Terminierungskriterien o.ä. sein.
- Die **Menge der Symbole** beschreibt die Objekte der Roboterwelt. Es ist die Menge der Konstanten, die alle Elemente im Einflussbereich des Agenten beschreibt. Neben Objekten können dies auch Aktionssymbole, interne Prozesse, Terminierungsbedingungen o.ä.sein.
- Die **Menge der Regeln** beschreibt relationale Regeln erster Ordnung. Sie beschreibt Übergangswahrscheinlichkeiten, welche durch Aktionen oder extern ausgelöst werden können. Vergleichbar mit dem Übergangsfunktion von *MDPs*.

Gesetzte *Facts* können *Activities* auslösen. Diese haben vollen Zugriff auf den *Relationalen Zustand*, welcher dadurch wie bei einem *Markov Schritt* geändert werden kann. Eine andere Möglichkeit der Veränderung ist durch die Vorwärtsverkettung der Regeln, bis ein stabiler Zustand erreicht ist, beispielsweise durch Erfüllung einer Terminierungsbedingung und entsprechender Beendigung einer *Activity*. Auf der anderen Seite hat auch der Aktionsplaner Zugriff auf den *Relationalen Zustand* und kann diesen manipulieren.

4.2. Activities autonomer Systeme

Aktionen autonomer Agenten können in mehrere Kategorien unterteilt werden. Neben der Manipulation durch Aktoren und der Wahrnehmung der Welt durch Sensoren können auch innere Prozesse als *Activity* definiert werden. Diese sind beispielsweise Planungs- oder Lernalgorithmen oder interne Prozesse der Steuerungssoftware wie das An- und Abschalten einzelner Sensoren oder Steuerungsroutinen. Diese Aktivitäten können entsprechend in einem Markov-Prozess verwendet werden (S. Abbildung 4.1).

Sensor Activities Die Sensorik des Agenten erfasst die Umwelt und versucht diese zu sinnvollen Einheiten zu verknüpfen. Das Ergebnis einer Sensoraktivität könnte das Erkennen und Verfolgen von Objekten sein, welche der Menge der Symbole hinzugefügt werden.

Effektor Activities Der Wahrnehmung steht die Manipulation gegenüber und ist der Output des Agenten. Durch Bewegung der Effektoren wir so die Umwelt verändert und Objekte werden bewegt.

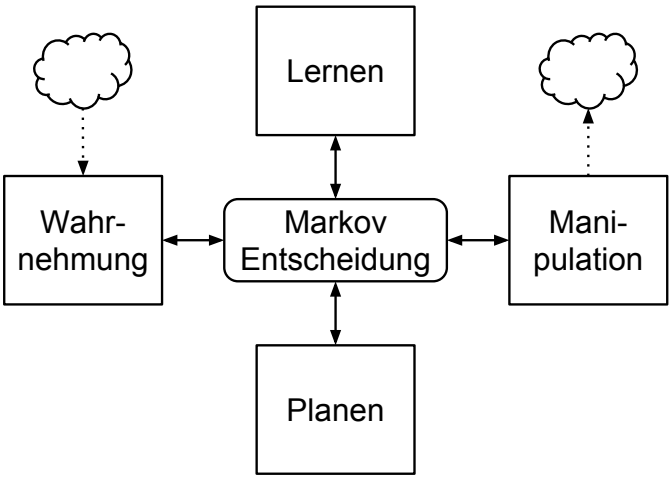


Abbildung 4.1.: Modularisierte Aktivitäten als Elemente von Markov-Entscheidungsprozessen

Planungs-Activities Die Planung von Aktionen ist im Gegensatz zur direkten Manipulation offline, also nicht unmittelbar. Sie erstellt anhand von Randbedingungen Trajektorien zur Objektmanipulation und bereitet so Bewegung vor.

Lern Activities Lernalgorithmen können auch als Aktivitäten modularisiert werden. So können verschiedene Lernalgorithmen an- oder ausgestellt werden entsprechend des Bedarfs.

Interne Steuerung Auch interne Steuerungsprozesse können als Aktionen definiert werden. Dadurch erhält der Aktionsplaner Zugriff auf Kontrollstrukturen und Schnittstellen der Software.

4.3. Implemetierung autonomer Aktionen als Activities

Im Folgenden wird die Implementierung autonomer Aktionen als *Activities* dokumentiert. Zuerst wird der Syntax zur Aktionsausführung beschrieben, dann das allgemeine Interface, welches die *Activities* beschreibt, gezeigt. Schließlich wird ein Beispielquelltext einer neuen *Activity* beschrieben und der Vergleich mit *RAP* diskutiert.

4.3.1. Syntax der Aktionsausführung

Die *Activities* sind Teilmenge der Prädikate des *Relationalen Zustandes*, der *Facts*. Ist ein entsprechender *Fact* gesetzt, wird eine *Activity* ausgelöst, wird er entfernt, die *Activity* beendet. Das entspricht dem Formalismus der *RAP*. Ein *Fact* besteht aus einem symbolischen

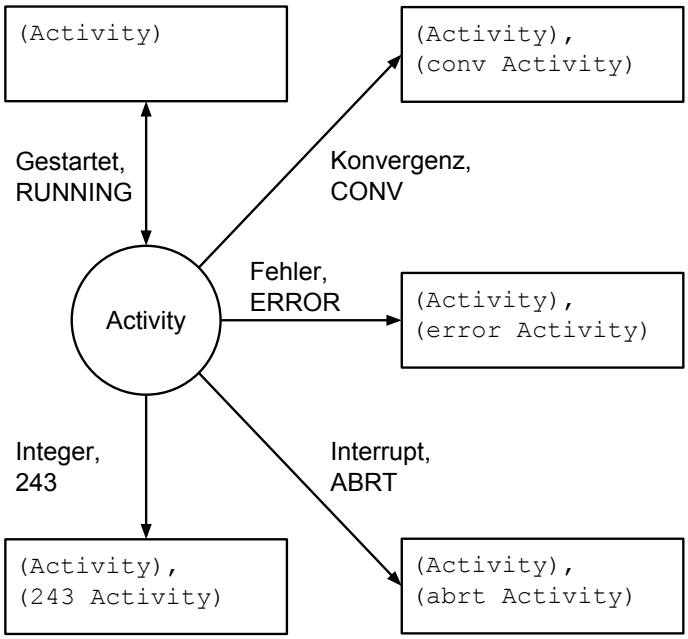


Abbildung 4.2.: Mögliche Zustandsveränderungen des *Relationalen Zustandes* (hier rechteckig) entsprechend des Zustandes einer *Activity* (hier rund).

Teil, welcher die *Activity* definiert und einem parametrischen Teil um ihr Parameter zu übermitteln. Prädikate sind hier in flacher Schreibweise. Das Prädikat `pos`(`endeff`, `objA`) wäre hier `(pos endeff objA)`. Beispielsweise löst der *Fact*

```
(Control pos endeff objA){ tol=.01 PD=[1,1,1,10] }
```

eine *Activity* vom Typ `Control`, eine Roboterbewegung aus. Diese erhält das Prädikat `pos`(`endeff`, `objA`) bzw. `(pos endeff objA)` welches entsprechend einen Positionscontroller auslöst, welcher den Endeffektor `endeff` zu der Position des Objektes `objA` bewegt. Im parametrischen Teil wird ein PD-Regler und eine Toleranz zur erfolgreichen Ausführung definiert. Da *Activities* vollen Zugriff auf den relationalen Zustand haben, kann diese bei erfolgreicher Ausführung der Bewegung folgenden *Fact* erzeugen:

```
(conv Control pos endeff objA)
```

Dieses Prädikat `conv`(`Control`(`pos`(`endeff`, `objA`))) repräsentiert die erfolgreiche Ausführung der *Activity*.

Anstatt dem Symbol `conv` können auch andere Symbole wie `ABORT` oder `error` den Zustand der *Activity* beschreiben. Zusätzlich kann jeder Integer als Zustand definiert werden. (S. Abbildung 4.2). Soll eine *Activity* bzw. ein*Fact* wieder entfernt werden, geht das mit hinten

angestellten `!`. Im oberen Beispiel müsste nach erfolgreicher Ausführung die *Activity* beendet und der *Fact* über die erfolgreiche Ausführung entfernt werden. Dies würde mit

```
(Control Control pos endeff objA)!
(conv Control pos endeff objA)!
```

geschehen. Jedoch ist das Interface der *Activities* so konzipiert, dass die Zustandsveränderungen automatisch passieren, d.h. der Zustand der *Activities* wird automatisch mit dem relationalen Zustand synchonisiert.

4.3.2. Allgemeines Interface der Activities

Die Implementierung des Frameworks der *RM* beinhaltet eine Klasse (diese ist als `struct` implementiert, welche jedoch folgend Klasse genannt wird) in *C++* zur Implementierung neuer *Activities*. Im Kern bestehen *Activities* aus der Initialisierung, Vergleichbar mit dem Initialisierungsoperator der *RAP* $\mathtt{o}_{init}$ einer Step-Funktion, welche in jedem Schritt den Zustand der *Activity* überprüft und anpasst und der Terminierung, vergleichbar mit dem Terminierungsoperator der *RAP* $\mathtt{o}_{term}$. Der Algorirthmus 4.1 zeigt den Header der Klasse der *Activities* in *C++*

Im folgenden werden einige Elemente des Headers des Interfaces beschrieben.

`Node *fact` ist der korrespondierende *Fact* zu der *Activity*.

`double activityTime` ist die verstrichene reelle Zeit seit dem Beginn der *Activity*.

`int statenum` beschreibt den Zustand der *Activity*. Dieser kann beispielsweise `CONV`, `RUNNING`, oder `ABORT` sein. Zudem kann jeder Integer-Wert angenommen und als Zustand definiert werden. Der Name `statenum` leitet sich von der Interrupt-Variable `signum` ab.

`void configure()` konfiguriert die *Activity*. Hier werden die Parameter gesetzt und Ziele definiert. Diese Funktion wird zu Beginn aufgerufen und stellt die Intitialisierung der *Activities* dar.

`void interruptHandler(int signum)` wird aufgerufen, wenn der Aktionsplaner die *Activity* außerplanmäßig unterbrechen will, um auf unerwartete Ereignisse zu reagieren.

`void activitySpinnerStep(double dt)` die Step-Funktion wird in jedem Zeitschritt aufgerufen und stellt so den richtigen Rahmen für beispielsweise einen PD-Regler zur Verfügung.

`Activity()` und `~Activity()` der Kon- und Destruktor wird nach Erzeugung bzw. direkt vor Beendigung der *Activity* ausgeführt.

Algorithmus 4.1 Header der Klasse der *Activities*

```
const int CONFIGURE = -1;
const int RUNNING = 0;
const int CONV = 1;
const int ABORT = 2;
const int ERROR = 3;
const int FILEERR = 4;
const int MOVEERR = 5;


struct Activity {
  Node *fact;     ///< pointer to the fact in the state of a KB
  double activityTime; ///< for how long is this activity running yet

  StringA symbols; ///< for convenience: copies of the fact->parent keys
  Graph params;   ///< for convenience: a copy of the fact parameters PLUS refX keys for
      all symbols

 int statenum ///< state of the ongoing Activity

  Activity():fact(NULL), activityTime(0.), statenum(CONFIGURE){}
  virtual ~Activity(){}
  void associateToExistingFact(Node *fact);
  void createFactRepresentative(Graph& state);

  /// configure yourself from the 'symbols' and 'params'
  virtual void configure(){}

  /// interrupt and error-handling
  virtual void interruptHandler(int signum){}

  /// the activity spinner runs with 100Hz and calls this for all activities -- use only
      for
  /// non-computational heavy quick updates. Computationally heavy things should be
      threaded!
  virtual void activitySpinnerStep(double dt){ activityTime += dt; }

  void write(ostream& os) const { os <<"Activity (" <<symbols <<"){" <<params <<"} (t="
      <<activityTime <<") "; if(fact) os <<*fact; else os <<"()"; }
};
```

Algorithmus 4.2 Step-Funktion einer Beispiel-*Activity*

```
void BeispielActivity::activitySpinnerStep(double dt){
 activityTime += dt;

 doSomethingQuick();

 if(activityState == ABORT)
      setFact(abort + fact);
 else if(activityState == ERROR)
      setFact(error + fact);
 else if(activityState == 243)
      setFact(s243 + fact);

 if(converged())
      setFact(conv + fact);
 else
      setFact(conv + fact + "!");
}
```

4.3.3. Funktionalität der Activities anhand eines Beispieles

Eine *Activity* besteht aus der Konfiguration, der Step-Funktion und der Terminierung. Algorithmus 4.2 ist der Quelltext der Step-Funktion einer Beispiel-*Activitiy* in Pseudocode. Die Step-Funktion führt etwas Kurzes aus, prüft den Zustand der *Activity* und setzt oder löscht *Facts*, um den *Realtionalen Zustand* entsprechend zu ändern.

In diesem Fall führt sie die Funktion `doSomethingQuick()` aus und setzt Prädikate für die Zustände `ABORT`, `ERROR` und `243`. Zudem überprüft sie in jedem Durchlauf, ob die Aktion konvergiert ist mit `converged()` und setzt einsprechend den *Fact*. Folgend werden die einzelnen Elemente der Step-Funktion genau beschrieben.

`activityTime += dt` addiert der *Activity* die verstrichene Zeit.

`doSomethingQuick()` ist eine Funktion in der schnelle Berechnungen gemacht werden, wie der schrittweisen Anpassung eines Reglers o.ä..

`converged()` wird wahr, wenn die textitActivity konvergiert, d.h. das Ziel erreicht hat.

`activityState == x` ist die Abfrage des Zustandes der *Activity* nach einem bestimmten Wert *x*.

`setFact()` ist die Funktion, welche auf den *Relationalen Zustand* zugreift und die Prädikate entsprechend der *Activity* anpasst. Beachte: *Fact* mit hinten angestelltem `!` löscht ein Prädkat

4.3.4. Implementierung von Relational Activity Processes

Das oben beschriebene Framework ist in der Lage *RAPs* zu implementieren. Um das zu zeigen, müssen die Operatoren $\texttt{o}_{init}$, $\texttt{o}_{term}$ und `wait` implementierbar sein.

$\texttt{o}_{init}$ besteht aus den Vorbedingungen $\texttt{pre}_{init}$, dem `go`-Prädikat und den Effekten auf den *Relationalen Zustand* $\texttt{post}_{init}$. Durch das Setzen eines *Facts* wird eine *Activity* gestartet. $\texttt{pre}_{init}$ kann realisiert werden, wenn in der Funktion `config()` im Fall nicht erfüllter Vorbedingungen der *Fact* direkt wieder entfernt wird. Das `go`-Prädikat entspricht der Repräsentation der *Activity* im *Relationalen Zustand* und $\texttt{post}_{init}$ dem Zugriff auf den *Relationalen Zustand* in der Step-Funktion.

$\texttt{o}_{term}$ implementiert sich auf ähnliche Weise, jedoch mit dem Unterschied, dass $\texttt{post}_{term}$ im Destruktor umgesetzt und das Entfernen des `go`-Prädikats durch Löschen der *Activity* aus dem *Relationalen Zustand* realisiert wird.

Der `wait`-Operator, kann durch die Variable `activityTime` realisiert werden, indem einer *Activity* im parametrischen Teil eine erwartete Laufzeit entsprechend eines Timeouts mitgegeben wird, oder die *Activity* sich nach erfolgreicher Ausführung selber beendet.

Die oben gezeigte Implementierung soll nur ein Beispiel darstellen um die Möglichkeiten zu zeigen. Eigentlich reagiert der Aktionsplaner auf erfüllte Vorbedingungen und startet oder beendet Aktionen, welche Teilmenge der Prädikate $\texttt{post}_{init}$ bzw. $\texttt{post}_{term}$ sind. Der Rest geschieht auf Planungsebene.

4.4. Ausnahmebehandlung als Elemente der Aktionsplanung

Leider ist Software meistens fehlerbehaftet, was im schlimmsten Fall zum Programmabbruch führen kann. Deshalb wird das oben beschriebene Framework um eine Fehlerbehandlung ergänzt welche die Möglichkeiten des Aktionsplaners erweitert. Dadurch wird unerwartetes Fehlverhalten planbar und der Aktionsplaner kann das Verhalten dahingehend optimieren, dass Fehlverhalten minimiert wird.

4.4.1. Struktur der Ausnahmebehandlung

Die Ausnahmebehandlung fängt nicht nur Fehler ab und überführt den Agenten in einen Fehlerzustand, sondern sie ermöglicht auch die Unterbrechung des normalen Programmablaufs durch den Aktionsplaner oder durch die interaktive Steuerung mittels Interruptsignale. Dadurch kommt entsteht ein neuer Signalfluss, welcher direkt mit den *Activities* kommuniziert ohne den Umweg über den *Relationalen Zustand*. Im Prinzip werden Fehler der *Activities*

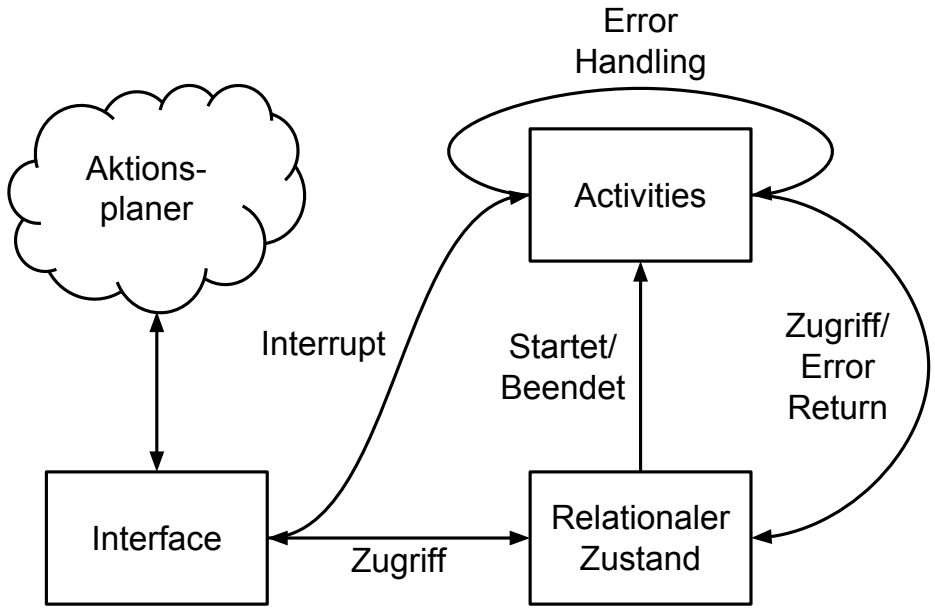


Abbildung 4.3.: Systemarchitektur der *Activities*

abgefangen und ein *Fact* bzw. ein Prädikat wird erstellt, welches den Fehler im *Relationalen Zustand* repräsentiert.

4.4.2. Systemarchitektur der Activities

Grundsätzlich startet und beendet der Aktionsplaner *Activities* durch das Setzen von entsprechenden Prädikaten bzw. *Facts*, welche die *Activities* triggern. Diese haben Zugriff auf den *Relationalen Zustand* und verändern diesen entsprechend ihres Zustandes. Neben dieser Grundfunktionen wurden im Rahmen dieser Arbeit die *Activities* um verschiedene Ausnahmebehandlungen erweitert.

Die Aktionsplanung kann durch ein Interruptsignal *Activities* ohne Rücksicht auf den *Relationalen Zustand* unterbrechen. Es wird dann für jede *Activity* eine Interruptfunktion aufgerufen, welche jeweils die entsprechenden Anweisungen ausführt und den *Relationalen Zustand* darüber informiert. Wird beispielsweise durch das Drücken von `ctrl + c` ein Interrupt ausgelöst, wird für jede aktive *Activity* eine Interruptfunktion aufgerufen. Diese könnte ein Interruptprädikat setzen, welches im nächsten Schritt die *Activity* beendet, oder diese sogar unmittelbar abbricht.

Zudem gibt es ein Exception-Handling, welches innerhalb der modularisierten *Activities* agiert. Wird ein Fehler erzeugt, führt das nicht mehr zum Programmabbruch, sondern führt lediglich eine Fehlerbehandlung aus, beendet evtl. die *Activity* und informiert wieder den

Relationalen Zustand über den Umgang mit dieser Ausnahme. D.h. die *Activity* teilt dem Aktionsplaner mit, dass beispielsweise eine Aktion außerplanmäßig abgebrochen wurde, oder der Interrupt erfolgreich war (S. Abbildung 4.3). Das Exception-Handling ist mittels der `std::exception` Klasse implementiert. Es lassen sich so eigene Fehler definieren und in einem `try-catch`-Block abfangen.

4.5. KOMO eine Beispielactivity mit Ausnahmebehandlung

Folgend wird eine Beispiel-*Activity* beschrieben, welche den Pfadplaner *KOMO* implementiert und auch mögliches Fehlverhalten abfängt. *KOMO* ist ein Framework zur Pfadoptimierung eines der Optimierungsprobleme [Tou14, S. 1f]. Es errechnet offline eine Trajektorie, die eine gewünschte Bewegung repräsentiert. Gegeben ist ein Weltmodell und Randbedingungen, welche bei der Pfadoptimierung eingehalten werden müssen. So kann beispielsweise die erforderliche Energie minimiert oder aber die Geschwindigkeit der Bewegung maximiert werden.

Der Algorithmus 4.3 zeigt die Implementierung des *KOMO*-Frameworks als *Activity*, der `komo.cpp`. In diesem Fall leitet sich die *Activity* nicht von der Klasse `ActivityS`, welche die Klasse `Activity` und einige Helfer erweitert sondern auch von der Klasse `Threat` ab, ist somit ein Thread und ermöglicht dementsprechend eine zeitintensive Offlineberechnung, ohne den Hauptprozess zu unterbrechen.

Im Kern besteht `KomoActivity` aus dem Konstruktor, welcher das *KOMO*-Objekt initialisiert, der Funktion `open()`, welche nebenläufig *KOMO* ausführt, dem Destruktor und dem Interrupthandler. Die einzelnen Elemente des Quelltextes werden folgend genau beschrieben.

Im Konstruktor wird das Weltmodell und die Randbedingungen aus der `specs.g` eingelesen und ein Komo-Objekt erstellt. Gibt es ein Problem mit der Datei wird der Zustand `FILEERR` sonst `RUNNING`.

In der Funktion `void open()`, welches die parallele Version von `void configure()` (s.o.) darstellt, wird *KOMO* mit `komo->run()` ausgeführt und im Falle der erfolgreichen Ausführung wird der Zustand der *Activity* nach `CONV` geändert. Zudem wird die erzeugte Trajektorie in der Datei `trajectory.dat` gespeichert. Wird ein Fehler erzeugt, wird der `catch`-block gestartet und der Zustand in `ERROR` geändert.

In der Funktion `void interruptHandler(int signum)` werden Interrupts abgefangen und der Zustand `statenum` entsprechend in den Interruptzustand `ABORT` geändert.

Die einzelnen Zustandsübergänge sind in Abbildung 5.1 dargestellt.

Algorithmus 4.3 Implementierung des Pfadplaners *KOMO* als *Activity*

```
#include "komo.h"
#include <Ors/ors.h>
#include <Motion/komo.h>
#include <Motion/motion.h>
#include <Core/graph.h>


KomoActivity::KomoActivity() : Thread("Komo", .1){
 try{
  komo = new KOMO(Graph("specs.g"));
  changeState(RUNNING);
  threadLoop();
 }catch(FileException &ex){
  changeState(FILEERR);
 }
}

void KomoActivity::open(){
 try{
  komo->run();
  FILE("trajectory.dat") <<komo->x;
  changeState(CONV);
 }
  catch(SIGSEGVException &ex){
   changeState(ERROR);
 }
}

void KomoActivity::interruptHandler(int signum){
 changeState(ABORT);
}

KomoActivity::~KomoActivity(){
 threadClose();
}
```

Die erstellte *KOMO*-Activity ist somit eine offline Activity, welche ein in der `specs.g` definiertes Optiermungsproblem löst und als Trajektorie speichert. Die in der `trajectory.dat` erstellte Trajektorie kann darauf in einer anderen *Activity*, beipieslweise mit `runTrajectory()`(Vgl. Anhang A) ausgeführt werden. Hierzu muss beachtet werden, dass der Ausgangszustand derselbe ist, da eine Trajektorie immer relativ zum Zustand gespeichert wird.

5. Experimentelle Evaluierung der Ausnahmebehandlung

Die im Rahmen dieser Arbeit erstellte Ausnahmebehandlung soll im folgenden Kapitel evaluiert werden. Es wird ein Szenario entwickelt, welches die hinzugekommenen Features zeigen, die Qualität bewerten und Grenzen aufzeigen soll. Dieses wird darauf umgesetzt und mit der Umsetzung ohne Fehlerbehandlung verglichen.

Zuerst wird in Kapitel 5.1 ein Anwendungsfall zur Evaluierung entwickelt und dann wird in Kapitel 5.2 das Experiment durchgeführt. Schließlich werden in Kapitel 5.3 die gewonnenen Ergebnisse ausgewertet und beurteilt.

5.1. Entwicklung eines Anwendungsfalls

Um die Funktionalität zu testen, sollen als exemplarische *Activities* der Pfadplaner `KOMO` und die *Activity* `runTrajectory` zur Ausführung von Trajektorien dienen. Daran soll gezeigt werden wie mit Fehlverhalten umgegangen werden kann. Diese werden im Folgenden beschrieben.

`KOMO` Wird der *Fact* `KOMO` gesetzt, wird die in Kapitel 4.5 beschriebene *Activity* zur Pfadoptimierung gestartet. Diese liest aus einer Datei das Weltmodell und die Parameter und erzeugt eine Trajektorie, welche in der Datei `trajectory.dat` gespeichert wird. Sie kann folgende Zustände annehmen:

- `CONFIGURE` ist der Startzustand. Er beschreibt die Konfigurationsphase mit der Dateieinlesung.
- `RUNNING` repräsentiert die asynchrone Pfadplanung.
- `CONV` ist der Endzustand der erfolgreichen Planung des Pfades.
- `FILEERR` ist ein Fehlerzustand, welcher auftritt, wenn die Datei `specs.g` einen Fehler enthält und *KOMO* nicht initialisierbar ist.
- `ERROR` ist ein Fehlerzustand, welcher auftritt, wenn *KOMO* einen nicht weiter spezifizierten Fehler macht.

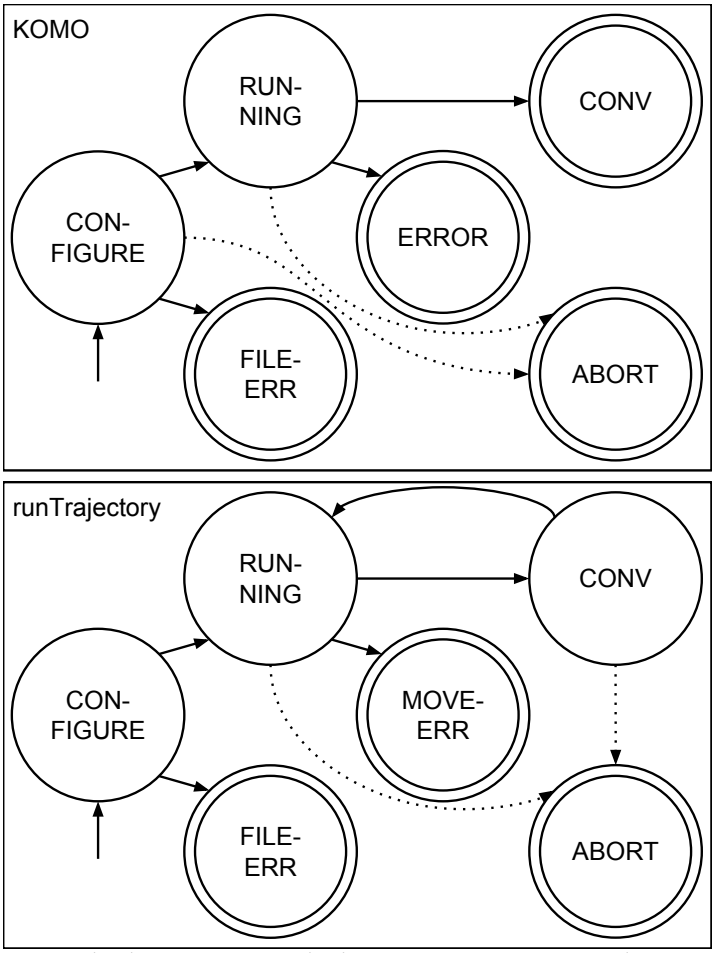


Abbildung 5.1.: Zustandsübergangsgraph der *Activities* `KOMO` und `runTrajectory`. Zielzustände sind doppelt umkreist, der Startzustand ist mit einem Pfeil markiert. Der Zustand `ABORT` wird mittels eines Interruptsignals erreicht

`runTrajectory` Wird der *Fact* `runTrajectory` gesetzt, wird eine *Activity* zur Ausführung der in der Datei `trajectory.dat` gespeicherten Trajektorie gestartet. Sie kann folgende Zustände haben:

`CONFIGURE` ist der Startzustand. Er beschreibt die Konfigurationsphase mit der Dateieinlesung.

`RUNNING` ist der Zustand während der Trajektorienausführung.

`FILEERR` ist ein Fehlerzustand. Er tritt auf, wenn die Datei `trajectory.dat` nicht vorhanden oder leer ist.

`MOVEERR` ist ein Fehlerzustand. Er entsteht, wenn während der Trajektorienausführung ein Fehler auftritt. Beispielsweise entsteht der Fehler, wenn der Zustand des

Algorithmus 5.1 *Python* Quelltext einer Strategie zur Trajektorienausführung

```
def runKOMOTrajectory():
  while True:
    fact("(KOMO)")
    waitForActivity("(KOMO)")
    if isTrue("fileerr KOMO"):
      fact("(KOMO)!")
      return("Die Datei specs.g ist fehlerhaft!")
    elif isTrue("running KOMO"):
      waitForActivity("(KOMO)")
      if isTrue("error KOMO"):
        fact("(KOMO)!")
        return("KOMO ist fehlerhaft!")
      elif isTrue("(conv KOMO"):
        fact("(KOMO)!")
        fact("(runTrajectory)")
        waitForActivity("runTrajectory")
        if isTrue("(fileerr runTrajectory)"):
          print("KOMO wird erneut ausgefuehrt")
        elif isTrue("(running runTrajectory)"):
          waitForActivity("runTrajectory")
          if isTrue("moveerr runTrajectory"):
            print("KOMO wird erneut ausgefuehrt")
          elif isTrue("conv runTrajectory"):
              fact("(runTrajectory)!"")
            return("Trajektorie wurde erfolgreich ausgefuehrt!")
```

Agenten sich verändert hat und die geplante Trajektorie nicht ausführbar ist oder ein Objekt im Weg ist.

`CONV` ist der Endzustand der erfolgreichen Ausführung des Pfades.

Abbildung 5.1 ist der Zustandsübergangsgraph der beiden beschriebenen *Activities*.

5.2. Realisierung des Experiments

Zur Umsetzung des Experiments wird eine Strategie entwickelt, welche die Berechnung und Ausführung der Trajektorie zum Ziel hat. Algorithmus 5.1 ist eine mögliche Strategie, welche das in der vorangegangenen Arbeit entwickelte *Python*-Interface nutzt [Bö15, Vgl.]. Abbildung 5.2 zeigt den Zustandsübergangsgraph des Algorithmus.

Die Funktion `runKOMOTrajectory()` führt zuerst `KOMO` und dann `runTrajectory` aus. Im Erfolgsfall gibt sie *"Trajektorie wurde erfolgreich ausgefuehrt!"* zurück, sonst *"Die Datei specs.g ist fehlerhaft!"* oder *"KOMO ist fehlerhaft!"*. Gibt es bei der Ausführung der Trajektorie ein Problem, wird `KOMO` erneut gestartet. Der Vorgang ist jederzeit mit `ctrl + c` unterbrechbar.

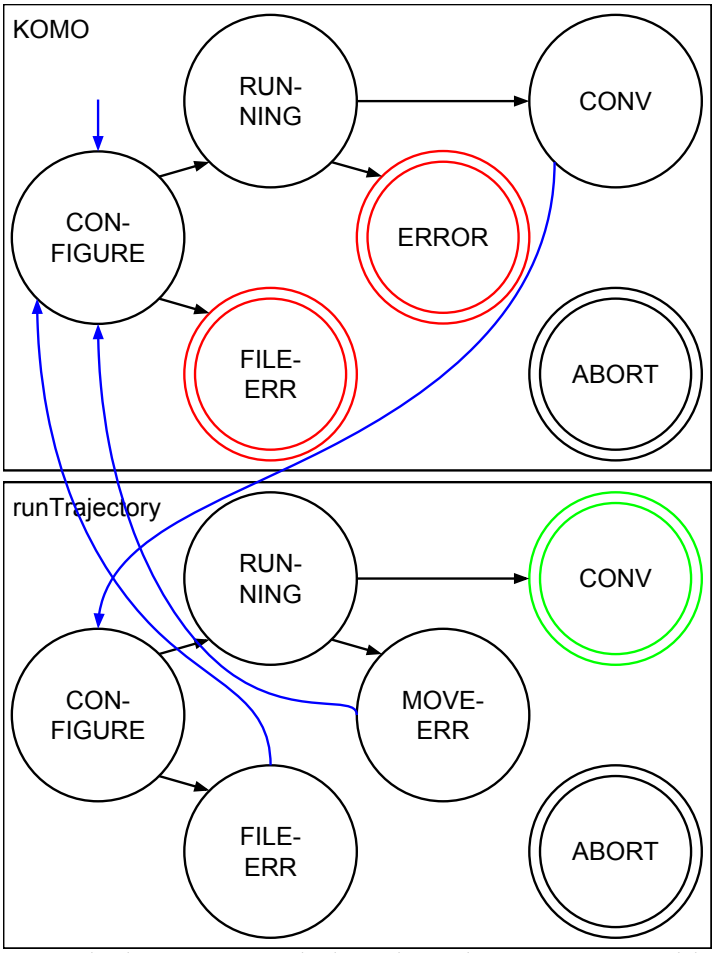


Abbildung 5.2.: Zustandsübergangsgraph des Algorithmus 5.1. Die Fehlerzustände sind rot, der Zielzustand ist grün. Blaue Pfeile sind externe Zustandsübergänge des Quelltexts.

Die Funktion `waitForActivity(X)` wartet auf einen Zustandsübergang der gegebenen *Activity* `X`. Die Funktion `isTrue(X)` überprüft, ob ein gegebener *Fact* im *Relationalen Zustand* vorhanden ist.

An der Abbildung 5.2 ist erkennbar, dass die Zustandsübergänge auch innerhalb der *Activites* realisierbar wären. Dann würde `KOMO` im Falle der Konvergenz den *Fact* `runTrajectory` setzen und `runTrajectory` würde in der Fehlerbehandlung bei allen Fehlern `KOMO` erneut starten. So würde sich Algorithms 5.1 auf Algorithmus 5.2 verkürzen.

Algorithmus 5.2 *Python* Quelltext einer Strategie zur Trajektorienausführung mit verbesserten *Activities*

```
def runKOMOTrajectoryNew():
  fact("KOMO")
  while True:
    if isTrue("fileerr KOMO"):
      return("Die Datei specs.g ist fehlerhaft!")
    elif isTrue("error KOMO"):
      print("KOMO ist fehlerhaft!")
    elif isTrue("conv Trajectory")
      print("Trajektorie wurde erfolgreich ausgefuehrt!")
```

Algorithmus 5.3 *Python* Quelltext einer Strategie zur Trajektorienausführung ohne Fehlerbehandlung

```
def runKOMOTrajectoryOld():
  fact("(KOMO)")
  while not isTrue(conv KOMO):
       pass
  fact("(KOMO)!")
  fact("runTrajectory")
  while not isTrue("(conv Trajectory)"):
       pass
  fact("(runTrajectory)!")
  return("Trajektorie wurde erfolgreich ausgefuehrt!")
```

5.3. Bewertung der Ergebnisse

Vor der Erweiterung des Frameworks um eine Fehlerbehandlung würde oben beschriebene Strategie wie in Algorithmus 5.3 implementiert werden. Die möglichen Fehler wie ein Dateifehler oder KOMO-Fehler würden zum sofortigen Programmabbruch führen. Auch kann auf mögliche falsche Trajektorienausführung nicht reagiert werden.

Durch die im Rahmen dieser Arbeit entwickelte Ausnahmebehandlung scheitert im schlimmsten Fall die *Activity* aber nicht das Programm.

6. Zusammenfassung und Ausblick

In der vorliegenden Arbeit wird die Frage behandelt, welche Eigenschaften die Implementierung von Aktionen autonomer Agenten haben muss um als Elemente relationaler, nebenläufiger Markov-Entscheidungsprozesse nutzbar zu sein. Neben der theoretischen Untersuchung dieser Frage wird auch an der praktischen Umsetzung einer Softwareumgebung zur Implementierung von Aktionen gearbeitet. Diese stellt ein Framework zur Verfügung, welches die stabile Implementierung unterschiedlichster Prozesse der Robotik ermöglicht.

Im ersten Abschnitt der Arbeit werden verschiedene Varianten verglichen das Entscheidungsproblem der Aktionsplanung als relationale, nebenläufige Markov-Entscheidungsprozesse zu formalisieren. Ausgehend von einem einfachen Markov-Entscheidungsprozess wird eine Parallelisierung erreicht, wenn der Aktionsplaner mehrere Aktionen gleichzeitig ausführen kann. Dadurch können aber Ressourcen- und Zeitkonflikte entstehen. Modelle wie *Concurrent Markov Decision Processes*, *Concurrent Action Model* oder *Relational Activity Processes* lösen diese Probleme. Darauf werden Programmiersprachen zur Aktionsplanung beschrieben, welche Verhaltenserzeugung nach dem Paradigma der *Kognitiven Robotik* ermöglichen und das maschinelle Lernen in ihrem Kern integrieren. So kann Verhalten auf der Ebene der Manipulation beschrieben und komplizierte Lernprozesse können mit einfachen Funktionen aufgerufen werden. Insbesondere wird auf die Sprachen *CRAM* und *RoLL* eingegangen.

Im darauffolgenden Abschnitt wird ein Framework erweitert, um unterschiedliche Aktionen implementieren zu können, dass die Planungs- und Lernprozeduren, welche das Entscheidungsproblem der Aktionsplanung lösen zur Verfügung zu stellen. Da in der Realität Software meistens fehlerbehaftet ist, wird dieses Interface um eine Fehler- und Signalverarbeitung erweitert. So können Fehler im System erzeugt und abgefangen werden, ohne dass diese zum Programmabbruch führen. Zudem kann der Aktionsplaner auf unvorhergesehene durch das Erzeugen von Interruptsignalen den normalen Programmablauf unterbrechen. Schließlich wird untersucht, inwieweit sich die verschiedenen *Markov-Entscheidungsmodelle* auf so erzeugte *Activities* anwenden lassen.

Schließlich wird im dritten Abschnitt durch die beispielhafte Implementierung und Anwendung einiger *Activities* die Funktionalität untersucht. Strategien über diese Aktionen können robust gestaltet werden, jedoch müssen Ausnahmen für jede *Activity* einzeln definiert werden. Erweiterbar wäre dieses Framework, wenn es eine zentrale Fehlerbehandlung enthielte, welche fest in das System integriert wäre.

A. Anhang - Anleitung zur Implementierung einer neuen Activity

Der vorliegende Anhang gibt eine Anleitung, wie mittels des neuen Interfaces eine neue *Activity* implementiert werden kann, welche die neuen Features zur integrierten Sicherheit beinhaltet. Eine so erstellte *Activity* kann mittels `ctrl + c` abgebrochen werden und hat ihre eigene Fehler- und Signalbehandlung.

Im Prinzip wird entsprechend des Ereignisses der Zustand der *Activity* geändert, welcher dann mit dem Relationalen Zustand synchronisiert wird bzw. andere Effekte auslöst. Die Funktionalität wird anhand der neuen Klassen beschrieben.

Klassen

Es gibt zwei neue Klassen

`ActivityS` abgeleitet von `Activity`. Sie erweitert `Activity` um einige Helfer und dient als Basisklasse zur Implementierung neuer *Activities*

`ActivityException` abgeleitet von `std::exception`. Sie bildet die Basisklasse für Klassen zur Fehlerbehandlung.

Die Klasse `Acitvity` wurde um neue Features erweitert.

Activity.h

Der Algorithmus A.1 zeigt alle Veränderungen der **Activity.h**. Diese werden folgend näher beschrieben.

`const int X = y` definiert Zustandskonstanten. Diese sind bisher

- `CONFIGURE` für Initialisierung der *Acitivity*.

Algorithmus A.1 Neue Elemente in der Activity.h

```
const int CONFIGURE = -1
const int RUNNING = 0;
const int CONV = 1;
const int ABORT = 2;
const int ERROR = 3;
const int FILEERR = 4;
const int MOVEERR = 5;

struct \textit{Activity} {

  int statenum; ///< states of the ongoing \textit{Activity}

  \textit{Activity}():fact(NULL), \textit{Activity}Time(0.), statenum(RUNNING){};

  /// interrupt and error-handling
  virtual void interruptHandler(int signum){}

};
```

  - `RUNNING` für die aktive *Activity*.
  - `CONV` für die konvergierte *Activity*.
  - `ABORT` für den Abbruch durch ein Interruptsignal.
  - `ERROR` für den Abbruch durch einen Fehler.
  - `FILEERR` für einen Dateifehler.
  - `MOVEERR` für einen Fehler bei der Ausführung einer Bewegung. Beispielsweise wenn die aufzuwendende Kraft zu groß wird.

  Diese sind bisher sehr allgemein gehalten und können erweitert werden.

`int statenum` beschreibt den aktuellen Zustand der *Activity*. Der Name leitet sich von Interrupt-Typ `int signum` ab.

`void interruptHandler (int signum)` ist die Interrupthandlerfunktion, welche für alle aktiven Activities bei einem Interrupt aufgerufen wird. Für `ctrl + c` gilt `sigint == 2 == ABORT`.

`Activity(): statenum(RUNNING)` der Konstruktor `Activity()` setzt für `statenum` `CONFIGURE`. D.h. die *Activity* wird konfiguriert.

Algorithmus A.2 ActivityS.h

```
#pragma once
#include <Ors/ors.h>
#include <Core/graph.h>
#include "\textit{Activity}.h"
#include <FOL/relationalMachine.h>
#include <csignal>
#include <exception>


struct \textit{Activity}S : \textit{Activity} {
  ACCESSname(RelationalMachine, RM);
  struct TaskControllerModule *taskController;
  struct RelationalMachineModule *relationalMachine;
  \textit{Activity}S();

  void changeState(int newStatenum);
  mlr::String getStateString(int statenum);

};

class \textit{Activity}Exception: public std::exception {
   private:
     const std::string msg;
   public:
     \textit{Activity}Exception() : msg("\textit{Activity}Error"){};
     \textit{Activity}Exception(const std::string& msg) : msg(msg) {};
   virtual ~\textit{Activity}Exception() throw() {};
   const char* what() const throw() {
     return msg.c_str();
   }
  };


class SIGSEGVException: public \textit{Activity}Exception {};
```

ActivityS.h

Neu dazugekommen ist die **ActivityS.h**. Sie beinhaltet die neue von `Activty` abgeleitete `struct ActivityS` und Klassen zur Fehlerbehandlung `ActivityException`. Der Algorithmus A.2 zeigt die Headerdatei **ActivityS.h**. Die Funktionalität wird wie folgt beschrieben.

`struct Activity` abgeleitet von *`Activity`* ist die Erweiterung um Sicherheitsfeatures und Hilfsfunktionen.

`void changeState(int newStatenum)` synchronisiert den *Relationalen Zustand*. D.h. sie entfernt den aktuellen Zustand und setzt den neuen `int newStatenum`.

`mlr::String getStateString(int statenum)` erzeugt den String welcher den aktuellen Zustand im *Relationalen Zustand* repräsentiert. Beispielsweise erzeugt die Eingabe `2` für die *Activity* `KOMO äbort KOMO"`.

`class ActivityException` abgeleitet von std::exception ist die Klasse zur Fehlerbehandlung der *Activities*. Neue Fehlerklassen können von dieser abgeleitet werden um Fehler in einem `try - catch` Block abzufangen.

`class SIGSEGVException` abgeleitet von `ActivityException` fängt einen erzeugten `SIGSEGV`-Fehler ab.

Implementierung einer neuen Activity

Soll eine neue *Activity* implementiert werden, ist als Basisklasse `ActivityS` zu verwenden. Zu Implementieren ist neben dem Konstruktor, `void configure()`, `void activitySpinnerStep(dt activityTime)` und dem Destruktor auch `interruptHandler(int signum)`. Zur Zustandsveränderung wird empfohlen, `void changeState()` zu verwenden, da sie den aktuellen Zustand automatisch aus dem *Relationalen Zustand* entfernt und den neuen setzt.

Literaturverzeichnis

[BMTR12] M. Beetz, L. Mosenlechner, M. Tenorth, T. Ruhr. Cram–a cognitive robot abstract machine. In *5th International Conference on Cognitive Systems (CogSys 2012)*, S. 9. 2012. (Zitiert auf den Seiten 6, 7, 28 und 30)

[BSR+08] M. Beetz, F. Stulp, B. Radig, J. Bandouch, N. Blodow, M. Dolha, A. Fedrizzi, D. Jain, U. Klank, I. Kresse, et al. The assistive kitchen—a demonstration scenario for cognitive technical systems. In *Robot and Human Interactive Communication, 2008. RO-MAN 2008. The 17th IEEE International Symposium on*, S. 1–8. IEEE, 2008. (Zitiert auf Seite 27)

[BW] E. Berger, K. Wyrobek. PR1 Robot Cleans a Room (8x Speed Up).flv. URL https://www.youtube.com/watch?v=jJ4XtyMoxIA. (Zitiert auf Seite 9)

[Bö15] R. Böhm. *Entwicklung einer Skriptsprache zur interaktiven Robotersteuerung*. Studienarbeit, Universität Stuttgart, 2015. (Zitiert auf den Seiten 35 und 49)

[FN71] R. E. Fikes, N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971. (Zitiert auf Seite 16)

[G+] B. Gerkey, et al. The Player Project. URL http://playerstage.sourceforge.net/. (Zitiert auf Seite 28)

[Har90] S. Harnad. The symbol grounding problem. *Physica D: Nonlinear Phenomena*, 42(1):335–346, 1990. (Zitiert auf Seite 9)

[Kir09] A. Kirsch. Robot learning language—integrating programming and learning for cognitive systems. *Robotics and Autonomous Systems*, 57(9):943–954, 2009. (Zitiert auf den Seiten 6, 32 und 33)

[LTK12] T. Lang, M. Toussaint, K. Kersting. Exploration in relational domains for model-based reinforcement learning. *The Journal of Machine Learning Research*, 13(1):3725–3768, 2012. (Zitiert auf Seite 16)

[MW08] Mausam, D. S. Weld. Planning with Durative Actions in Stochastic Domains. *J. Artif. Intell. Res.(JAIR)*, 31:33–82, 2008. (Zitiert auf den Seiten 7, 14, 18 und 19)

[RM02] K. Rohanimanesh, S. Mahadevan. Learning to take concurrent actions. In *Advances in neural information processing systems*, S. 1619–1626. 2002. (Zitiert auf den Seiten 6 und 21)

[ros] ROS - Robot Operating System. URL http://ros.org. (Zitiert auf Seite 28)

[RPMG04] K. Rohanimanesh, R. Platt, S. Mahadevan, R. Grupen. Coarticulation in markov decision processes. In *Advances in Neural Information Processing Systems*, S. 1137–1144. 2004. (Zitiert auf den Seiten 18 und 20)

[SPS99] R. S. Sutton, D. Precup, S. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1):181–211, 1999. (Zitiert auf Seite 20)

[TB09] M. Tenorth, M. Beetz. KnowRob—knowledge processing for autonomous personal robots. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, S. 4261–4266. IEEE, 2009. (Zitiert auf den Seiten 6, 30 und 31)

[Thr00] S. Thrun. Towards programming tools for robots that integrate probabilistic computation and learning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, Band 1, S. 306–312. IEEE, 2000. (Zitiert auf Seite 14)

[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 den Seiten 9 und 27)

[TMM+15] M. Toussaint, T. Munzer, Y. Mollard, L. Y. Wu, M. Lopes. Relational Activity Processes for Modeling Concurrent Cooperation. S. 8, 2015. (Zitiert auf den Seiten 16 und 22)

[Tou14] M. Toussaint. Newton methods for k-order Markov constrained motion problems. *arXiv preprint arXiv:1407.0414*, 2014. (Zitiert auf Seite 44)

[Tou15] M. Toussaint. Relational Machine Interfacing between robot activity control and relational learning and planning methods. 2015. (Zitiert auf Seite 35)

[VMS07] D. Vernon, G. Metta, G. Sandini. A survey of artificial cognitive systems: Implications for the autonomous development of mental capabilities in computational agents. *IEEE Transactions on Evolutionary Computation*, 11(2):151, 2007. (Zitiert auf Seite 28)

[WM04] D. S. Weld, Mausam. Solving concurrent Markov decision processes. In *Proceedings of the 19th national conference on Artifical intelligence*, S. 716–722. AAAI Press, 2004. (Zitiert auf Seite 15)

[WMF+] P. C. Wang, S. Miller, M. Fritz, T. Darrell, P. Abbee. PR2 Autonomously Pairing Socks. URL https://www.youtube.com/watch?v=uFkIHPrzS_8. (Zitiert auf Seite 9)

[YMS03] H. L. Younes, D. J. Musliner, R. G. Simmons. A Framework for Planning in Continuous-time Stochastic Domains. In *ICAPS*, S. 195–204. 2003. (Zitiert auf Seite 9)

[ZPK05] L. S. Zettlemoyer, H. Pasula, L. P. Kaelbling. Learning planning rules in noisy stochastic worlds. In *AAAI*, S. 911–918. 2005. (Zitiert auf Seite 17)

Alle URLs wurden zuletzt am 30. 05. 2016 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