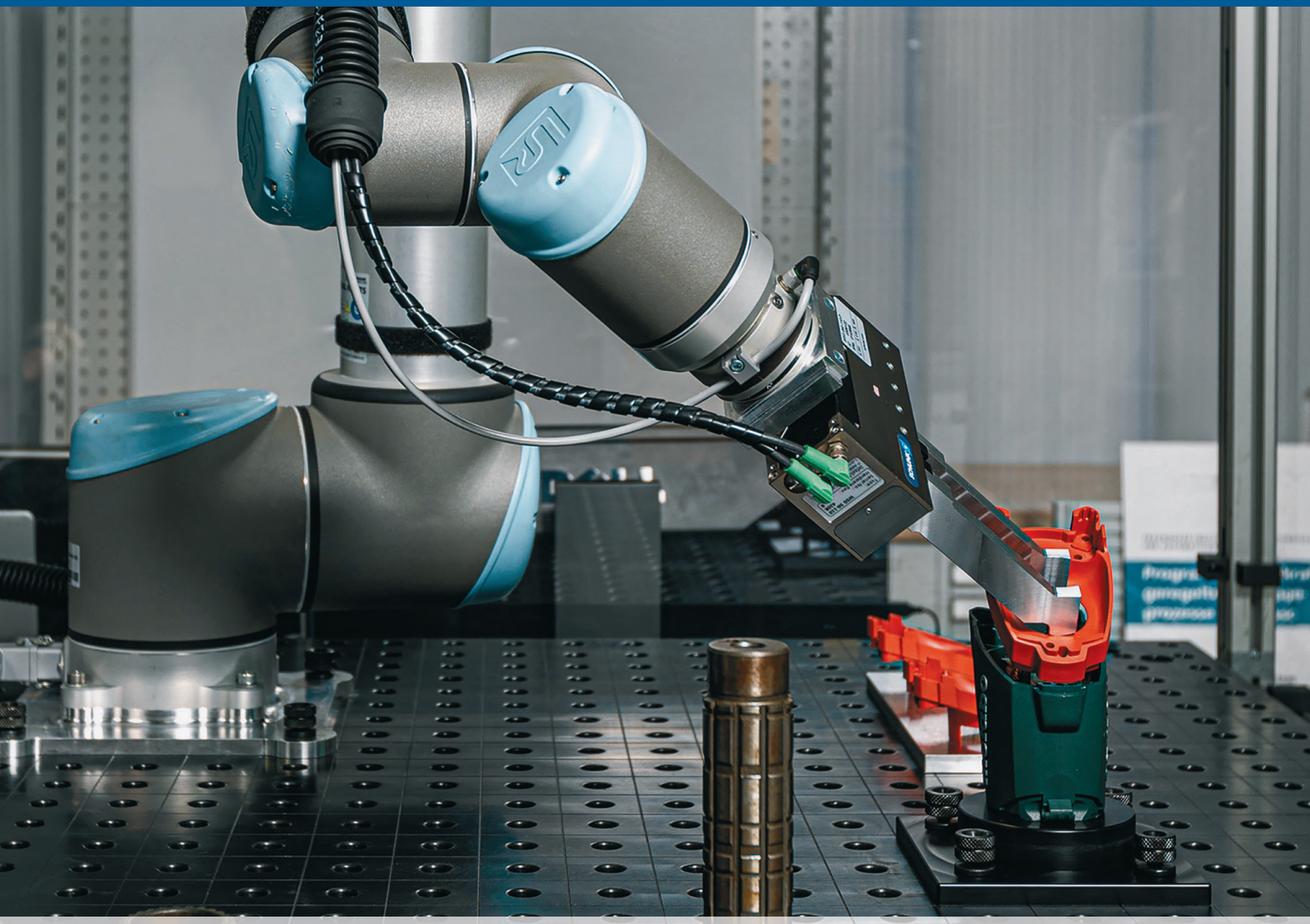
STUTTGARTER BEITRÄGE ZUR PRODUKTIONSFORSCHUNG

FRANK NÄGELE

Prototypbasiertes Skill-Modell zur Programmierung von Robotern für kraftgeregelte Montageprozesse



Universität Stuttgart

Fraunhofer IPA

STUTTGARTER BEITRÄGE ZUR PRODUKTIONSFORSCHUNG BAND 121

Frank Nägele

Prototypbasiertes Skill-Modell zur Programmierung von Robotern für kraftgeregelte Montageprozesse

FRAUNHOFER VERLAG

Kontaktadresse:
Fraunhofer-Institut für Produktionstechnik und Automatisierung IPA, Stuttgart
Nobelstraße 12, 70569 Stuttgart
Telefon 07 11/9 70-11 01
info@ipa.fraunhofer.de; www.ipa.fraunhofer.de

STUTTGARTER BEITRÄGE ZUR PRODUKTIONSFORSCHUNG

Titelbild: © Rainer Bez, Fraunhofer IPA

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.de abrufbar.

ISBN: 978-3-8396-1719-9

D 93

Zugl.: Stuttgart, Univ., Diss., 2020

Druck und Weiterverarbeitung:
Fraunhofer Verlag, Mediendienstleistungen

Für den Druck des Buches wurde chlor- und säurefreies Papier verwendet.

Nobelstraße 12
70569 Stuttgart
verlag@fraunhofer.de
www.verlag.fraunhofer.de

als rechtlich nicht selbständige Einheit der

Fraunhofer-Gesellschaft zur Förderung
der angewandten Forschung e.V.
Hansastraße 27 c
80686 München
www.fraunhofer.de

Prototypbasiertes Skill-Modell zur Programmierung von Robotern für kraftgeregelte Montageprozesse

Von der Fakultät Konstruktions-, Produktions- und Fahrzeugtechnik
der Universität Stuttgart
zur Erlangung der Würde eines Doktor-Ingenieurs (Dr.-Ing.)
genehmigte Abhandlung

Vorgelegt von
Frank Nägele
aus Biberach a.d. Riß

Hauptberichter:	PD Dr.-Ing. habil. Andreas Pott
Mitberichter:	Univ.-Prof. Dr.-Ing. habil. Marco Huber
Tag der mündlichen Prüfung:	20. Juli 2020

Institut für Steuerungstechnik der Werkzeugmaschinen und Fertigungseinrichtungen (ISW)
der Universität Stuttgart

2020

Vorwort des Autors

Die vorliegende Arbeit ist während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Produktionstechnik und Automatisierung IPA der Fraunhofer-Gesellschaft entstanden. Mein herzlicher Dank gilt PD Dr.-Ing. habil. Andreas Pott für die wissenschaftliche Betreuung meiner Arbeit sowie die wertvollen Ratschläge und Anregungen bei der Findung und Schärfung des Dissertationsthemas und bei der Erstellung meiner Veröffentlichungen. Ebenso danke ich Univ.-Prof. Dr.-Ing. habil. Marco Huber für seine hilfreichen Anmerkungen und die freundliche Übernahme des Mitberichts. Heide Kreuzburg und Luzia Schuhmacher danke ich für die stets herzliche organisatorische Unterstützung.

Mein Dank gilt Martin Naumann, Dr. Werner Kraus und Martin Hägele für das entgegengebrachte Vertrauen, mit nur dem es möglich war, die der Arbeit zugrundeliegende Software pitasc zu entwerfen, zu entwickeln und schließlich zur Motek 2015 und ICRA 2018 erstmals dem Fachpublikum und der wissenschaftlichen Gemeinschaft zu präsentieren.

Ich danke allen meinen Kollegen am Fraunhofer IPA. Ohne die enge Zusammenarbeit mit Lorenz Halt, Philipp Tenbrock und Philipp Gorczak wäre diese Arbeit nicht möglich gewesen. Ebenso danke ich meinen Kollegen Bernd Winkler und Daniel Bargmann für unsere Diskussionen und gemeinsamen Projekte. Mein Dank gilt auch unseren Studenten für ihr Mitwirken.

Meiner Familie und meinen Freunden danke ich von Herzen für ihre Unterstützung und insbesondere meiner Frau Sabine für die Ermutigungen und die Freiräume, die es mir erlaubt haben, diese Arbeit zu schreiben. Ihr und meinen Kindern danke ich für die schönen gemeinsamen Wochenenden, die zwischen arbeitsreichen Werktagen lagen.

Stuttgart, im November 2020 Frank Nägele

Kurzinhalt

Ausgehend von der Motivation, den Einsatz von Industrierobotern für Montageanwendungen zu erleichtern, wird in dieser Arbeit ein Skill-Modell zur Programmierung kraftgeregelter Montageprozesse konzipiert und demonstriert.

Klassische Positions- und Bahnsteuerungen zielen auf das sequenzielle Anfahren abgespeicherter Zielpositionen mit hoher Geschwindigkeit und Wiederholgenauigkeit ab. Sie stoßen jedoch bei vielen Montageprozessen an ihre Grenzen, die stattdessen eine definierte Regelung der Fügekräfte erfordern und eine große Anzahl von Produktvarianten abdecken müssen. Skill-basierte Methoden wie der in dieser Arbeit verwendete iTaSC Formalismus erlauben es hingegen, Positions-, Geschwindigkeits- und Kraftregelung zu vereinen und durch eine werkstückzentrierte Modellierung Programme leicht an Varianten anzupassen. Es stellt sich allerdings die Frage, wie auf Basis eines solchen Formalismus eine umfangreiche Skill-Bibliothek erstellt werden kann, mit deren Skills sich eine große Vielzahl an Montageprozessen effizient realisieren lässt.

Das vorgestellte *pitasc* Skill-Modell wird daher als Baukastensystem entworfen. Die elementaren Bausteine, aus denen sich Skills zusammensetzen, werden in die Teilaspekte Hardwareabstraktion, kinematische Modellierung, Aufgabenspezifikation mit Reglern und Stoppbedingungen sowie Koordination der einzelnen Skills mittels Statecharts aufgeteilt. Da jeder Baustein nur einen der Teilaspekte modelliert, wird seine Wiederverwendung erleichtert. Um nicht jeden Skill von Grund auf neu zusammensetzen und parametrieren zu müssen, können Skills mittels Komposition und prototypbasierter Vererbung auf bestehenden Skills aufbauen und diese wiederverwenden und erweitern. Die Modellierung erfolgt dabei in einem generischen Parameterbaum, der es erlaubt, die Methoden der Parametrierung, Komposition und Vererbung auf alle Arten von Parametern gleichermaßen anzuwenden – vom einfachen String über Bausteine und Skills bis hin zu ganzen Anwendungen. Zur einfachen Modellierung wird eine domänenspezifische Sprache vorgestellt.

Anhand von zehn exemplarischen Montageanwendungen wird gezeigt, dass bereits eine kleine Anzahl an Skills die Basis für eine Vielzahl von industriellen Anwendungsfällen bilden kann und es sehr einfach ist, weitere Skills mittels vorhandener Bausteine zu erstellen. Die erreichbare Robustheit wird ebenso untersucht wie die Abbildung von Produktvarianten über Parameter und die Unabhängigkeit der Skills von den eingesetzten Robotern, Sensoren und Greifern.

Short Summary

With the motivation to facilitate the use of industrial robots for assembly applications, this thesis designs and demonstrates a skill model for programming force-controlled assembly processes.

Traditional point-to-point and continuous path control systems aim to move through a list of target positions with high speed and accuracy. However, they reach their limitations in many assembly processes, which instead require a precise control of the joining forces and the configurability to accommodate large numbers of product variants. Skill-based methods such as the iTaSC formalism used in this thesis, make it possible to combine position, velocity, and force control and to easily adapt programs to product variants through workpiece-centered modeling. The critical question arises as to how an extensive skill library can be created on the basis of such a formalism, allowing a broad range of assembly processes to be efficiently implemented.

The presented *pitasc* skill model is therefore designed as a modular system. The elementary building blocks from which skills are composed, are separated into the aspects of hardware abstraction, kinematic modelling, task specification with controllers and stop conditions, as well as coordination of the individual skills using statecharts. Each building block models only one of the aspects, facilitating its reuse.

In order to avoid having to compose and parameterize each skill from scratch, skills can be built on top of existing ones using composition and prototype-based inheritance, reusing and extending them. The modeling is carried out in a generic parameter tree, which allows the methods of parameterization, composition, and inheritance to be applied equally to all types of parameters, from simple strings, building blocks, and skills to entire applications. A domain-specific language is introduced for easy modelling.

Using ten exemplary assembly applications, it is shown that even a small number of skills can form the basis for a variety of industrial applications and that it is very easy to create additional skills using existing building blocks. The achievable robustness is examined, as well as the mapping of product variants via parameters and the independence of the skills from the robots, sensors, and grippers in use.

Inhaltsverzeichnis

Vorwort des Autors **III**

Kurzinhalt **V**

Short Summary **VII**

Abkürzungsverzeichnis **XIII**

Abbildungsverzeichnis **XV**

Tabellenverzeichnis **XVII**

1 Einleitung **1**

1.1 Problemstellung 1
1.2 Literaturübersicht 3
1.3 Ziele und Ansatz der Arbeit 10

2 Grundlagen **13**

2.1 Grundlagen der Kinematik 13
2.1.1 Kinematik 14
2.1.2 Differentielle Kinematik 17
2.1.3 Inverse differentielle Kinematik 21
2.2 Kraftregelung 31
2.3 Programmierung sensorbasierter Robotersysteme 32
2.4 Spezifikation sensorbasierter Roboterbewegungen mit iTaSC 40

3 Modellierung und Ausführung elementarer Bausteine **45**

3.1 Softwarearchitektur 45
3.2 Hardwarekomponenten 50
3.3 Kinematische Elemente 53
3.3.1 Datenquelle 53
3.3.2 Kinematisches Kettenglied 58
3.3.3 Kinematische Kette 62

3.3.4 Kinematische Schleife . . . 63
3.4 Aufgabenspezifikation . . . 65
3.4.1 Task . . . 65
3.4.2 Stoppbedingung . . . 66
3.4.3 Skript . . . 67
3.5 Koordination . . . 68
3.6 Ausführung des Modells . . . 72
3.6.1 Scene . . . 72
3.6.2 iTaSC . . . 73
3.6.3 Solver . . . 75

4 Modell zur Komposition und Spezialisierung von Bausteinen 77
4.1 Skill-Modell . . . 78
4.1.1 Modellbasierter Ansatz und Abstraktion . . . 80
4.1.2 Komposition . . . 88
4.1.3 Prototypbasierte Vererbung . . . 94
4.2 Domänenspezifische Sprache für die Modellierung von Montageanwendungen . . . 100
4.2.1 Syntax und allgemeine Sprachelemente . . . 101
4.2.2 Domänenspezifische Sprachelemente . . . 104
4.3 Komposition und inkrementelle Spezialisierung von Skills . . . 108
4.3.1 Komposition einfacher Skills . . . 108
4.3.2 Komposition von Sequence-Skills . . . 113
4.3.3 Komposition von Statechart-Skills . . . 114
4.3.4 Komposition von Concurrency-Skills . . . 115

5 Evaluation 119
5.1 Erstellung einer umfassenden Skill-Bibliothek . . . 120
5.1.1 Skripte . . . 120
5.1.2 Stoppbedingungen . . . 125
5.1.3 Skills . . . 127
5.2 Komposition und Einsatz komplexer Montage-Skills . . . 131
5.2.1 Montage von Hutschienenelementen . . . 131
5.2.2 Montage eines Plastikbauteils durch Aufstecken . . . 136
5.2.3 Eindrehen von Schrauben . . . 138
5.3 Modellierung robuster Skills und Anwendungen . . . 140
5.4 Wiederverwendung und Anpassbarkeit von Skills . . . 145
5.4.1 Anpassbarkeit bei Produktvarianten . . . 146
5.4.2 Wiederverwendbarkeit bei neuen Prozessen und Produkten . . . 148

5.4.3 Betrachtung der Hardwareabhängigkeit 150

6 Zusammenfassung und Ausblick **159**

Literatur **161**

Abkürzungsverzeichnis

b-CAP	Binary Control Access Protocol, Roboterschnittstelle von Denso
BCM	BRICS Component Model
BRICS	Best Practice in Robotics, Forschungsprojekt im Rahmen des «European Union's Seventh Framework Programme» (FP7)
BRIDE	BRICS Integrated Development Environment
CPC	Component-Port-Connector Model
DAG	Gerichteter azyklischer Graph (von engl. *directed acyclic graph*)
DSL	Domänenspezifische Sprache (von engl. *domain-specific language*)
EMF	Eclipse Modeling Framework
eTaSL	expressiongraph-based Task Specification Language
FSM	Zustandsmaschine (von engl. *finite state machine*)
GUI	Grafische Benutzeroberfläche (von engl. *graphical user interface*)
IDL	ROS Interface Definition Language
iTaSC	instantaneous Task Specification using Constraints
KDL	Kinematics and Dynamics Library
KIF	Knowledge Integration Framework
KRL	KUKA Robot Language
KUKA	Hersteller von Robotern
LBR	Leichtbauroboter, oft synonym für KUKA LBR verwendet
LIN	Lineare kartesische Roboterbewegung
M2M	Modell-zu-Modell-Transformation

M2T	Modell-zu-Text-Transformation
MDA	Modellgetriebene Architektur (von engl. *model-driven architecture*)
MDE	Modellgetriebene (Software-) Entwicklung (von engl. *model-driven engineering*)
MOF	Meta Object Facility
OCL	OROCOS Component Library
OMG	Object Management Group
OROCOS	Open Robot Control Software
PTP	Punkt-zu-Punkt Bewegung (von engl. *point-to-point*)
RCC	Passives Ausgleichselement (von engl. *remote center of compliance*)
rFSM	restricted Finite State Machine
ROS	Robot Operating System
RPC	Remote Procedure Call
RSI	Robot Sensor Interface, Roboterschnittstelle von KUKA
RTDE	Real-Time Data Exchange, Roboterschnittstelle von Universal Robots
RTT	Real-Time Toolkit
SPA	Sense-Plan-Act Programmierparadigma
SVD	Singulärwertzerlegung (von engl. *singular value decomposition*)
TFA	Task Function Approach
TFF	Task Frame Formalism
UML/P	Unified Modeling Language / suitable for programming
UML	Unified Modeling Language
URDF	Unified Robot Description Format

Abbildungsverzeichnis

1.1 Weltweit eingesetzte Industrieroboter in Prozent nach Anwendungsfeldern 2
1.2 Kraftgeregelte Montage von Kunststoffbauteilen 4

2.1 Repräsentation der Orientierung in Form von Drehachse und Drehwinkel 14
2.2 Repräsentation der Orientierung in Form von Z-Y-X Kardan-Winkeln 15
2.3 Direkte und inverse Kinematik 16
2.4 Transformation des Geschwindigkeitsvektors 18
2.5 MDA 4-Ebenenmodell der Abstraktion 37
2.6 Modellierung einer Anwendung mit iTaSC 41
2.7 Kinematische Schleife mit Objekt- und Feature-Koordinatensystemen 42

3.1 Primäre Komponenten und Ressourcen des Robotersystems 46
3.2 Elemente des Kernprogramms und Datenfluss 47
3.3 Modell einer Roboterzelle mit zwei Robotern 49
3.4 Elementare kinematische Bausteine 53
3.5 Arten von Datenquellen 54
3.6 Modellierung der Feature-Koordinaten mithilfe von Zylinder- oder Kugelkoordinaten 57
3.7 Implementierungsarten von Kettengliedern 59
3.8 Transformation eines Kettenglieds mit Drehgelenk 60
3.9 Transformation der Jacobi-Matrix 62
3.10 Kinematische Schleife 64
3.11 Arten von Skills und Skill-Kompositionen 69
3.12 Beispiel eines Concurrency-Skills 69
3.13 Statechart einer Schraubapplikation 71
3.14 Priorisierte Liste aktiver Skills eines Statecharts vor und nach einer Transition . . . 72

4.1 Konzeptuelle Darstellung des Gesamtsystems 78
4.2 Ansatz des *pitasc* Skill-Modells 79
4.3 MDA 4-Ebenenmodell der Abstraktion 81
4.4 Deklarative Aufgabenbeschreibung und Parametrierung des Ausführungsmechanismus 82
4.5 Parameter als Datencontainer 83

4.6 UML-Diagramm der Parameter-Klassen 84
4.7 UML-Diagramm der Factory-Klassen 86
4.8 Beispiel eines Parameterbaums 89
4.9 Beispiel für eine klassenbasierte Objektorientierung 96
4.10 Beispiel für eine prototypbasierte Objektorientierung 96
4.11 Varianten prototypbasierter Vererbung 97
4.12 Vererbungshierarchie mit Kopiermechanismus auf Basis tiefer Kopien 99
4.13 Vererbungshierarchie mit einfachem Kopiermechanismus 99
4.14 Beispiel für die Vererbung eines Prototyps 100
4.15 Beispiele für die Komposition und Vererbung einfacher Skills 109
4.16 Verwendung des `slide` Skills für die Montage von Hutschienenelementen 115
4.17 Beispiele für Concurrency-Skills 116

5.1 Versuchsaufbauten, die das vorgestellte Skill-Modell verwenden 122
5.2 Hierarchie der wichtigsten Skill-Familien 127
5.3 Komposition und Vererbung kraftgeregelter Skills 128
5.4 Komposition und Vererbung von `idle` Skills 129
5.5 Beispielanwendung zur Montage von Hutschienenelementen 132
5.6 Modellierung des Hutschienenelements 132
5.7 Ablauf und Parametrierung der Hutschienenbestückung (1) 133
5.8 Ablauf und Parametrierung der Hutschienenbestückung (2) 135
5.9 Gemessene Kräfte bei der Hutschienenbestückung 136
5.10 Beispielanwendung zur Montage eines Plastikbauteils durch Aufstecken 137
5.11 Ablauf und Parametrierung des Einführvorgangs 137
5.12 Beispielanwendung zur Verschraubung eines Getriebegehäuses 138
5.13 Ablauf und Parametrierung des Schraubprozesses (1) 139
5.14 Ablauf und Parametrierung des Schraubprozesses (2) 140
5.15 Möglicher Fehlerfall bei der Hutschienenbestückung 141
5.16 Fehlerbehebung: Erneuter Versuch mit nach links verschobener Startposition 142
5.17 Ergebnisse der Montagevorgänge 143
5.18 Kräfte in z-Richtung aller erfolgreichen Montagevorgänge 144
5.19 Kräfte in z-Richtung aller beim ersten Versuch gescheiterten Montagevorgänge . . . 145
5.20 Bauteilvarianten für die Hutschienenbestückung 146

Tabellenverzeichnis

1.1 Übersicht vergleichbarer Arbeiten 5

4.1 Attribute eines Parameters 83
4.2 Suchalgorithmen des Parameterbaums 90
4.3 Operationen zur Modifikation des Parameterbaums 93

5.1 Beschreibung und Kategorisierung der exemplarischen Anwendungen 121
5.2 Kategorien und Beispiele häufig genutzter Skripte 123
5.3 Kategorien und Beispiele häufig genutzter Stoppbedingungen und ihre jeweiligen Auslösekriterien 125
5.4 Auflistung der für die exemplarischen Anwendungen verwendeten Skills 149
5.5 Unterstützte Roboter und ihre Schnittstellen 151
5.6 Unterstützte Kraft-Momenten-Sensoren und ihre Schnittstellen 153
5.7 Unterstützte Greifer und ihre Schnittstellen 155

1 Einleitung

1.1 Problemstellung

Industrieroboter zeichnen sich durch große Beweglichkeit, freie Programmierbarkeit und damit einhergehend durch eine hohe Anwendungsflexibilität aus – Eigenschaften, die gerade für die Automatisierung von Montageprozessen essenziell sind. Nach den Zahlen der International Federation of Robotics (IFR 2018) befinden sich jedoch nur knapp 11 Prozent der weltweit eingesetzten Roboter in der Montage. Handhabung und Schweißen hingegen dominieren mit zusammen über 70 Prozent der eingesetzten Roboter klar als Anwendungsfelder (Abbildung 1.1). Anschaulich wird dieser Unterschied beim Blick in die Automobilproduktion, die sich in die fast vollständig automatisierten Bereiche des Karosseriebaus und Lackierens und den größtenteils manuellen Bereich der Endmontage aufteilt (Hägele et al. 2016). Abseits der Automobilproduktion zeichnet sich ein ähnliches Bild ab, wobei der Automatisierungsgrad hier insgesamt bereits deutlich geringer ausfällt[1]. Gleichzeitig steht die verarbeitende Industrie an vielen Stellen unter dem Druck zu automatisieren, sei es aus Gründen der Produktivität, Qualität, Ergonomie oder des Fachkräftemangels (Pott et al. 2019).

Es gibt viele potenzielle Ursachen für den geringen Automatisierungsgrad der Montage, wobei in dieser Arbeit Defizite im Bereich der Programmierung von Robotern im Vordergrund stehen. Beim Fügen von Bauteilen kommt es per definitionem zum Kontakt der Bauteile. Klassische Positions- und Bahnsteuerungen, die auf das sequenzielle Anfahren abgespeicherter Zielpositionen mit hoher Wiederholgenauigkeit abzielen, stoßen hierbei an ihre Grenzen. So kann es aufgrund von Werkstücktoleranzen oder Ungenauigkeiten in der Positionierung der Werkstücke zu sehr hohen Kontaktkräften kommen, wenn der Roboter die eingespeicherte Zielposition zu erreichen versucht. Vorteilhaft ist es hier, den Montageprozess über eine auszuübende Fügekraft zu definieren und diese durch den Roboter zu regeln. Kraft-Momenten-Sensoren erlauben dafür die Erfassung der wirkenden Kräfte und Momente am Roboter. Es werden jedoch neue

[1] So liegt in Deutschland, über alle Anwendungsfelder hinweg, die Anzahl an Robotern pro Mitarbeiter in der Automobilindustrie (1.162 Roboter je 10.000 Mitarbeiter) über sechsmal höher als in den verbleibenden Industriebranchen (191 Robotern je 10.000 Mitarbeiter) (IFR 2018).

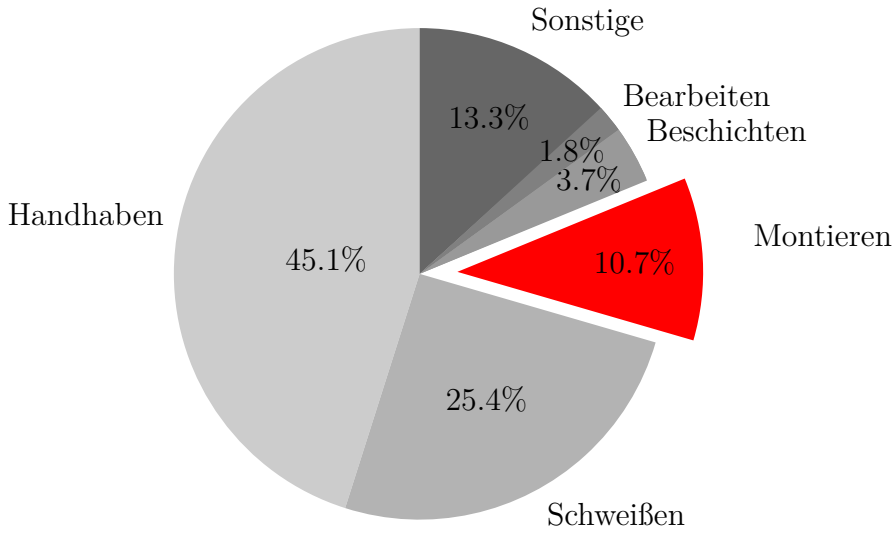


Abbildung 1.1: Weltweit eingesetzte Industrieroboter in Prozent nach Anwendungsfeldern (IFR 2018)

Programmierkonzepte zur Beschreibung und Ausführung von Roboterprogrammen nötig, die eine Integration von Kraftmesswerten berücksichtigen.

Eine weitere Herausforderung wird durch den Trend zur individualisierten Massenproduktion und Personalisierung von Produkten bedingt. Dieser führt zu einer wachsenden Variantenvielfalt und damit einhergehend zu kleineren Stückzahlen je Variante. Robotersysteme müssen daher flexibler als bisher ausgelegt werden. Auch die Aufwände für die Programmierung müssen sinken, da sie auf entsprechend weniger Exemplare je Variante umgelegt werden können. Häufigere Variantenwechsel erfordern zudem eine schnellere Inbetriebnahme, um eine Zunahme der kumulierten Umrüstzeiten zu vermeiden.

Neben der Variantenvielfalt ist auch die Prozessvielfalt in der Montage eine Herausforderung für deren wirtschaftliche Automatisierung. So ist Montage der Oberbegriff für eine Vielzahl unterschiedlicher Fügeprozesse. Allein die Gruppe des *Zusammensetzens* von Bauteilen unterteilt sich nach der Norm DIN 8593-1 *Fertigungsverfahren Fügen* in die Verfahren zum Auflegen, Einlegen, Ineinanderschieben, Einhängen, Einrenken und federnd Einspreizen. Hinzu kommen weitere Kategorien wie das An- und Einpressen (inklusive des Schraubens) oder das Kleben. Soll der Automatisierungsgrad der Montage signifikant erhöht werden, sind rein prozessspezifische Lösungen aufgrund der großen Anzahl unterschiedlicher Fügeprozesse nicht zielführend.

Neue Verfahren der Roboterprogrammierung müssen demnach die Prozessvielfalt der Montage von vornherein adressieren, beispielsweise indem verallgemeinerbare Lösungen gefunden werden, die sich auf verschiedene Prozesse anwenden lassen. Ebenso müssen sie eine Programmierung komplexer, kraftgeregelter Fügeprozesse ermöglichen und die Anpassung von Robo-

terprogrammen an neue Produktvarianten vereinfachen. Abbildung 1.2 zeigt eine Roboterzelle für die Montage von Kunststoffteilen im Automobilbau, die beispielhaft für die in dieser Arbeit betrachteten Montageprozesse sein soll.

1.2 Literaturübersicht

Die Robotikforschung widmet sich mit dem Ansatz der skill-basierten Roboterprogrammierung den Herausforderungen, die kraftgeregelte Fügeprozesse, Varianten- und Prozessvielfalt sowie die Komplexität der Roboterprogrammierung mit sich bringen. Bereits in den 1990er Jahren stellen Hasegawa et al. (1992) und Morrow et al. (1997) die wichtigsten Vorzüge modellbasierter Skills gegenüber traditionellen Programmiermethoden dar:

- *Skills* teilen eine komplexe Roboteraufgabe in handhabbare Teilprozesse und verbergen die Details ihrer jeweiligen Implementierung, wodurch weniger Expertenwissen zur Programmierung von Roboteraufgaben nötig ist. Einzelne Skills werden zu Programmen zusammengesetzt. Durch die strukturierte Programmierung lassen sich dabei selbst komplexe Programmabläufe einfach darstellen.

- Die Verwendung eines *Formalismus zur Spezifikation und Ausführung kraftgeregelter Roboteraufgaben* erlaubt das systematische Programmieren komplexer Fügeprozesse. Durch das Einbinden von Kraftregelung können Modellfehler und Werkstücktoleranzen ausgeglichen werden.

- Die geometrische *Modellierung der Umgebung und der Roboteraufgabe* erhöht die Flexibilität von Roboterprogrammen, da viele Variationspunkte über Parameter anstelle von neu programmierten Funktionen modelliert werden können. Neue Produktvarianten oder Werkstückpositionen können oft über eine einfache Anpassung der Parameter berücksichtigt werden, während bei traditionellen Methoden meist eine erneute Programmierung der Roboterbewegungen nötig ist.

- Parametrierbare Skills sind zudem ein wichtiger Schritt in Richtung *Wiederverwendbarkeit*. Eine umfangreiche Skill-Bibliothek erlaubt eine schnellere Inbetriebnahme von Robotersystemen und geringere Programmierkosten.

- Skills fügen eine *Hardwareabstraktionsschicht* ein. Die Aufgabenbeschreibung kann weitgehend unabhängig vom ausführenden Roboter oder den tatsächlichen Positionen von Werkstücken und Vorrichtungen in der Roboterzelle erfolgen. Dadurch wird die Wiederverwendbarkeit weiter erhöht.

Abbildung 1.2: Kraftgeregelte Montage von Kunststoffbauteilen

Über die Jahre wurde eine Vielzahl an Skill-Modellen zur Spezifikation von Roboteraufgaben veröffentlicht. Dieser Abschnitt gibt eine Übersicht der für die vorliegende Arbeit relevanten Modelle. Von Interesse sind vorrangig Skill-Modelle, die sämtliche der folgenden Ebenen betrachten:

- Formalismus zur Spezifikation und Ausführung kraftgeregelter Roboteraufgaben
- Mechanismus zur Koordination der zeitlichen und logischen Abfolge einzelner Skills
- Domänenspezifische Programmiersprache (DSL, von engl. *domain-specific language*) oder andere Nutzerschnittstelle zur einfachen und effizienten Aufgabenspezifikation

Tabelle 1.1 fasst die primären Publikationen zusammen, die diese Kriterien weitestgehend erfüllen.

Bei vielen Skill-Modellen basiert die Aufgabenspezifikation auf dem *Task Frame Formalism* (TFF), oft auch *Compliance Frame Formalism* genannt. Die Grundzüge des TFF gehen auf Mason (1981) zurück. Bruyninckx et al. (1996) formalisieren die Definition des TFF und zeigen eine Reihe von exemplarischen Aufgabenspezifikationen. Die Aufgabenspezifikation des TFF erfolgt bezüglich eines beweglichen kartesischen Koordinatensystems, dem sogenannten *Task Frame*. Entlang dessen drei Achsen können die zu realisierenden Geschwindigkeiten oder Kräfte vorgegeben werden. Dem entsprechend werden die Drehgeschwindigkeiten oder Drehmomente um die drei Achsen definiert. Jeder dieser insgesamt sechs Freiheitsgrade wird demnach entweder geschwindigkeits- oder kraftgeregelt.

Tabelle 1.1: Übersicht vergleichbarer Arbeiten

Primäre Publikationen	Formalismus zur Aufgabenspez.	Koordinations-mechanismus	Nutzerschnittstelle oder Metamodell
Klotzbücher et al. (2011), Klotzbücher (2013)	TFF	rFSM	TFF-DSL, rFSM-DSL
Weidauer et al. (2014), Finkemeyer (2004)	TFF	Petri-Netz	–
Thomas et al. (2013), Butting et al. (2015)	TFF	Statechart	LightRocks DSL
Kresse et al. (2012), Kresse (2017)	TFA	DAG	Movement description language
Vanthienen et al. (2013), Vanthienen (2015)	iTaSC	rFSM	iTaSC-DSL, rFSM-DSL
Stenmark (2015), Stolt (2015)	iTaSC	JGrafchart	RobotStudio Plug-In, Text-/Spracheingabe

Klotzbücher et al. (2011) und Klotzbücher (2013) stellen zwei DSLs zur Programmierung von Roboteraufgaben vor. Mithilfe der ersten DSL erfolgt die Aufgabenspezifikation auf Basis des TFF, mit einer zweiten DSL die Koordination der Skills über eine *restricted Finite State Machine* (rFSM) (Klotzbücher et al. 2010; Klotzbücher et al. 2012c). Beide DSLs sind als Erweiterung der Lua-Skriptsprache umgesetzt und basieren auf der Modellierungssprache *uMF* (Klotzbücher et al. 2012b). Skills werden über parametrisierte Templates erstellt, die vom Anwender ausgefüllt werden. Stoppbedingungen definieren, wann ein Skill beendet werden soll und werden als Lua-Funktionen programmiert. Eine Bibliothek mit einer kleinen Anzahl an Skills wird bereitgestellt, unter anderem zum Aufbauen eines Kontakts, Drücken und Schieben. Ein komplexer Skill zum Ausrichten von Werkstücken wird beispielhaft aus diesen zusammengesetzt.

Mosemann et al. (2001) stellen die ebenfalls TFF-basierten Aktionsprimitive vor (engl. auch *skill primitive*, *manipulation primitive* oder *manipulation task*). Montageprozesse aus einem Planungsprogramm werden klassifiziert und den Aktionsprimitiven zugeordnet, mit denen sie ausgeführt werden sollen. Aktionsprimitive können hier einfache Bewegungsbefehle, Werkzeugoperationen oder ein sogenannter *HybridMove*-Befehl sein, der eine sensorgeführte Bewegung nach dem TFF beschreibt. Das Konzept wird von Thomas et al. (2003) um ein Aktionsprimitivnetz (engl. *skill primitive net*) ergänzt. Aktionsprimitive werden als Knoten in einem gerichteten Graph angeordnet. Die Kanten zwischen Aktionsprimitiven definieren dabei die Bedingungen

für eine Transition. So lassen sich komplexe Montagefolgen modellieren, insbesondere auch mit Verzweigungen auf Basis aktueller Sensormesswerte. Eine adaptive Selektionsmatrix, die entscheidet, welcher Regler zu jedem Zeitpunkt aktiv ist, erweitert das System (Finkemeyer 2004; Finkemeyer et al. 2005; Kröger et al. 2010). Für jede Achse des Task Frames kann dabei eine priorisierte Liste an Reglern vorgegeben werden. Kann ein Regler zum aktuellen Zeitpunkt nicht verwendet werden, beispielsweise ein Kraftregler für eine Richtung, in der es aktuell keinen Kontakt mit der Umgebung gibt, so wird der nächste, niedriger priorisierte Regler verwendet, der aktiviert werden kann. Dieser Ansatz erlaubt es, innerhalb eines Takts den Regler zu wechseln, um somit schnell auf sich verändernde Bedingungen zu reagieren. Weidauer et al. (2014) schließlich stellen die Koordination auf ein Petri-Netz um. Dieses erlaubt eine hierarchische Anordnung von Skills, wodurch die Komplexität gegenüber eines flachen Netzes reduziert wird und das Ersetzen oder die Wiederverwendung einzelner Teile des Netzes erleichtert werden. Die gleichzeitige Ausführung von Aktionsprimitiven ist nun möglich, beispielsweise um zwei Roboter anzusteuern. Es kann zu jedem Zeitpunkt jedoch nur ein Aktionsprimitiv je Roboter aktiv sein. Eine Kombination partieller Aufgabenspezifikationen, bei denen jeweils nur einzelne Freiheitsgrade eingeschränkt werden, ist demnach nicht möglich. Die genannten Publikationen demonstrieren den Einsatz der Aktionsprimitive für verschiedene Anwendungen, unter anderem dem Einsetzen einer Glühlampe mit Bajonettverschluss, der Montage eines Handyakkus oder dem Herausziehen von Holzklötzen des Spiels Jenga (Kröger et al. 2008).

Thomas et al. (2013) übernehmen das Konzept der Aktionsprimitive (hier *elemental actions* genannt) und adaptieren es für die Impedanzregelung eines KUKA LBR. Um die Programmierung von Anwendungen zu vereinfachen und die Wiederverwendung von Skills zu verbessern, werden drei Abstraktionsebenen definiert. Auf unterster Ebene werden Aktionsprimitive zu Skills verknüpft. Auf den beiden darüber liegenden Ebenen werden auf gleiche Art Skills zu Tasks und schließlich Tasks zu Prozessen verbunden. Aktionsprimitive und Skills sollen dabei von Experten erstellt werden. Die Programmierung von Tasks und Prozessen sei dann aber ohne besondere Robotik- beziehungsweise Programmierkenntnisse möglich. Zur Programmierung wurde die *LightRocks* DSL entworfen. Sie nutzt UML/P Statecharts zur Beschreibung der Aktionsprimitiv-, Task- und Skill-Netze und die Infrastruktur des *MontiCore*-Frameworks zur Codegenerierung (Krahn et al. 2010). Ein grafischer Editor als Plug-in für Eclipse unterstützt das Arbeiten mit LightRocks. Obwohl das System auf einen KUKA LBR zugeschnitten ist, wird Codegenerierung für verschiedene Roboter vorgesehen, für die jeweils ein Adapter zur Roboterschnittstelle erstellt werden muss. Demonstriert wird das System anhand einer Hutschienenbestückung, ähnlich zu der in dieser Arbeit gezeigten. Butting et al. (2015) stellen eine überarbeitete Version von LightRocks vor. Dabei werden die UML/P Statecharts durch vier konzeptionell einfachere MontiCore DSLs ersetzt, die jeweils zur Definition von Prozessen, Tasks, Skills beziehungsweise Aktionsprimitiven genutzt werden.

Der TFF stößt bei komplexen Aufgaben an seine Grenzen. Die Beschränkung auf einen einzelnen Task Frame erschwert die Kombination partieller Aufgabenspezifikationen, die jeweils nur einen Teil der Freiheitsgrade einschränken, sowie die Integration mehrerer Sensoren. Die Ansätze der Constraint-basierten Programmierung (engl. *constraint-based programming*) adressieren dieses Problem. Dazu gehören der *Task Function Approach* (TFA) (Samson et al. 1991), iTaSC (De Schutter et al. 2007) und eTaSL (Aertbeliën et al. 2014). Bei diesen Ansätzen wird nicht die Bewegung eines Werkzeug- oder Task-Frames vorgeschrieben. Stattdessen werden Relativbewegungen zwischen Objekten des Robotersystems modelliert, hauptsächlich Werkstücke, Werkzeuge und Vorrichtungen. Die Relativbewegungen können nun durch *Constraints* (engl. für Zwangs- oder Nebenbedingungen) eingeschränkt werden, um eine Roboteraufgabe zu modellieren. Da Freiheitsgrade einzeln eingeschränkt werden können, lassen sie sich zu komplexen Aufgaben kombinieren. Hierbei ist jedoch nicht garantiert, dass die kombinierten partiellen Aufgabenspezifikationen konfliktfrei sind. Sie müssen deshalb entsprechend gewichtet oder priorisiert werden, um eventuelle Konflikte zu lösen. Ausgehend von dieser deklarativen Aufgabenspezifikation lässt sich mit dem TFA, iTaSC oder eTaSL die resultierende Roboterbewegung berechnen.

Kresse et al. (2012) und Kresse (2017) verwenden den Task Function Approach (TFA) für ihr Skill-Modell. Elementare Zwangsbedingungen lassen sich für jeweils einen einzelnen Freiheitsgrad feingranular angeben. Dazu werden ein Werkzeug- und ein Werkstück-Feature (Punkt, Linie oder Ebene) über eine Feature-Funktion verknüpft und je ein Sollwertbereich für Position und Kraft angegeben. Feature-Funktionen definieren dabei die räumliche Beziehung der Features zueinander und können über den TFA direkt in die Bewegungsausführung übersetzt werden. Darüber hinaus geben sie der räumlichen Beziehung eine semantische Bedeutung, wie beispielsweise *pointing-at*, *height*, *distance* oder *align*, wodurch die Programmierung anschaulicher wird. Die Verwendung von Sollwertbereichen gegenüber eines einzelnen Werts ermöglicht es, Zwangsbedingungen zu deaktivieren, die gerade erfüllt sind, um niedriger priorisierte Aufgaben zu realisieren. Die Zwangsbedingungen werden über die *Movement Description Language* (Bartels et al. 2013) definiert, die jedoch nicht formell als DSL eingeführt wird. Mehrere elementare Zwangsbedingungen lassen sich zu komplexeren Aufgaben wie *move-under* oder *keep-horizontal* kombinieren, wodurch eine sehr abstrakte, symbolische Aufgabenspezifikation möglich wird. Neben den Zwangsbedingungen zur Aufgabenspezifikation wird eine Liste an Zwangsbedingungen angegeben, die als Stoppbedingung agieren. Erst wenn alle Zwangsbedingungen erfüllt sind, gilt die Aufgabe als erledigt. Die Koordination der einzelnen Bewegungsphasen erfolgt über einen gerichteten azyklischen Graphen (DAG). Zwar ist die Arbeit auf häusliche Tätigkeiten ausgerichtet – so wird das Wenden eines Pfannkuchens demonstriert – die Aufgabenspezifikation und das Regelungskonzept sollten aber auf industrielle Montageaufgaben übertragbar sein. In einem ähnlichen System, das jedoch auf eTaSL beruht, werden Features aus Punktwolken

extrahiert und Skill-Parametern zugeordnet (Gajewski et al. 2019). Die Skills sind dabei sehr generisch definiert. So wird die Übertragbarkeit eines Skills zum Abschaben von Lebensmitteln in Gefäße mit verschiedensten Haushaltsgegenständen demonstriert.

Smits et al. (2008) demonstrieren, wie sich iTaSC für die Integration mehrerer Sensoren und Roboter eignet und wie einfach sich damit selbst komplexe Aufgabenspezifikationen modellieren lassen. So wird ein Robotersystem mit zwei Robotern, Kraft-Momenten-Sensor, Kamera, Laser-Distanzsensor und Laser-Scanner gezeigt, für das eine Reihe unterschiedlicher Aufgaben spezifiziert wird. Es wird zudem dargestellt, wie sich bei einer unterbestimmten Aufgabenspezifikation verbleibende Freiheitsgrade zur Erfüllung sekundärer Zwangsbedingungen nutzen lassen. Im gezeigten Beispiel werden die beiden Roboter damit nahe ihrer optimalen Arbeitskonfiguration gehalten. Smits et al. (2009) erweitern iTaSC um eine Koordinationsebene auf Basis von Statecharts. Über die Gewichtung von Zwangsbedingungen lassen sich zudem sowohl das Überblenden von einem Skill zum nächsten umsetzen als auch Zwangsbedingungen als Ungleichungen formulieren. Ein rekursiver Algorithmus wird von Smits (2010) angewendet, um eine beliebige Anzahl an priorisierten Zwangsbedingungen auszuführen.

Vanthienen et al. (2013) und Vanthienen (2015) präsentieren eine DSL für iTaSC. Diese verwendet eine überarbeitete Version (Vanthienen et al. 2014) von Smits ursprünglicher iTaSC-Implementierung. Wie bei Klotzbücher wird die DSL als Erweiterung der Lua-Skriptsprache umgesetzt, mit uMF als Meta-Metamodell. Zur Koordination der Skills wird ebenfalls eine rFSM verwendet. Anders als bei den bisher vorgestellten Systemen werden Sollwerte über Sollwertgeber (engl. *setpoint generators*) definiert. Da diese als eigene Softwarekomponenten umgesetzt werden, können nicht nur statische, sondern auch zeitlich veränderliche Werte vorgegeben werden. Als Beispielanwendung wird das Öffnen einer Schublade durch einen mobilen Manipulator demonstriert, wobei hier ebenso eine Anwendung für industrielle Montage denkbar ist.

Stolt (2015) zeigt eine Implementierung von iTaSC mit vordefinierten Skills für kraftgeregelte Suchbewegungen. Zur Programmierung von Montageanwendungen wird zunächst ein Montagevorranggraph in ABB RobotStudio definiert, einer Software zur Offline-Programmierung von Robotern. Zu jedem Montageprozess werden die involvierten Bauteile oder Baugruppen angegeben, sowie ihre gewünschten geometrischen Beziehungen zueinander. Ebenso müssen der zu verwendende Greifer und geeignete Greifpunkte angegeben werden. Aus den nun vorliegenden Daten wird automatisch eine Montagefolge generiert und es werden den einzelnen Montageprozessen Skills zugeordnet und parametriert. Die Spezifikation der Anwendung kann dann exportiert und als JGrafchart Zustandsmaschine ausgeführt werden. Weitere Publikationen demonstrieren die Montage von Notausschaltern (Stolt et al. 2011a) und Flugzeugbauteilen (Stolt et al. 2011b). Darüber hinaus wird für die Montage des Notausschalters ein sich selbst optimierender Impedanzregler vorgestellt (Stolt et al. 2012a). Mithilfe eines Recursive-Least-Squares

Algorithmus werden hier Parameter des Kontaktmodells identifiziert, wodurch der Anwender von der manuellen Optimierung der Reglerparameter befreit wird.

Stenmark (2015) stellt ein System zur einfacheren Programmierung von Robotern vor. Dazu gehört ein Plug-In für ABB RobotStudio, mit dem in natürlicher Sprache geschriebene Anweisungen zur Programmierung von Montageprozessen genutzt werden können (Stenmark et al. 2013). Anweisungen wie «Hole die Leiterplatte aus dem Werkstückträger und platziere sie in der Vorrichtung. Dann nimm das Abschirmblech und montiere es auf die Leiterplatte» werden von einem Parser interpretiert. Die Satzbestandteile werden mit den in der Software hinterlegten Werkstücken und parametrierbaren Skills abgeglichen und diesen zugeordnet. Daraus wird die Struktur des Roboterprogramms generiert, die dann vom Anwender geprüft, angepasst und ausgeführt werden kann. Stenmark et al. (2014a) erweitern das System um Mengenangaben («Nimm zwei Schrauben»), Konditionalsätze («wenn ... dann ... sonst») und Stoppbedingungen («... bis 5 N gemessen werden»). Zudem wird eine App zur Umwandlung von Sprache in Text integriert, wodurch eine direktere Programmierung möglich wird. Stenmark et al. (2014b) detaillieren den Schritt der Codegenerierung. Aus der Montagefolge wird eine JGrafchart Zustandsmaschine generiert, die bei der Ausführung unter anderem auch die sensorgeführten Skills von Stolt aufruft. Um die vielen einzelnen Schritte von der Spracheingabe bis zur Ausführung auf dem Roboter weitgehend automatisiert zu realisieren, wird eine ganze Reihe an wissensbasierten Diensten benötigt. Björkelund et al. (2011) und Stenmark et al. (2015) beschreiben das *Knowledge Integration Framework* (KIF), das die hierfür verwendeten Ontologien und die Datenhaltung sowie die damit verbundenen Dienste zusammenfasst.

Weitere Ansätze zur Vereinfachung der Roboterprogrammierung kommen aus dem Bereich maschineller Lernverfahren. Hier zeigen sich vielversprechende neue Möglichkeiten insbesondere zur (teil-)automatischen Parametrierung von Skills. Durch den Einsatz verschiedener Algorithmen wie beispielsweise DDPG (*Deep Deterministic Policy Gradient*) (Vecerik et al. 2019) und CMA-ES (*Covariance Matrix Adaptation Evolution Strategy*) (Johannsmeier et al. 2019) wird das Lernen von Einführvorgängen (sogenannte *peg-in-hole* Prozesse) demonstriert. Hierbei führt der Roboter den Montageprozess mehrfach selbstständig aus. Alternativ kann auch ein Mensch den Montageprozess vorführen und der Roboter erlernt durch Imitation beispielsweise Fügestrategien (Suomalainen et al. 2016; Scherzinger et al. 2019), Suchstrategien (Ehlers et al. 2019) oder eine kollisionsfreie Vorpositionierung (Vergara Perico et al. 2019). Die genannten Beispiele beschränken sich jedoch auf einfache *peg-in-hole* Prozesse. Auch ist die Reduzierung des Datenbedarfs für Lernverfahren ein offenes Forschungsthema. Dies betrifft insbesondere Montagevorgänge, die irreversibel sind oder kostspielige Bauteile fügen. Skill-Formalismen wie der in dieser Arbeit vorgestellte sind eine vielversprechende Möglichkeit, den Datenbedarf zu reduzieren (El-Shamouty et al. 2019).

In Summe sind skill-basierte Ansätze zur Programmierung von Robotern ein wichtiger Schritt in die Richtung der übergreifenden Motivation, Montageroboter flexibler, schneller und damit wirtschaftlicher zu programmieren. Das zentrale Argument ist dabei die hohe Effizienz der Programmierung, die durch die Wiederverwendung existierender Skills entsteht. Die in diesem Abschnitt beschriebenen Ansätze und Beispiele zeigen meist jedoch nur eine sehr kleine, beispielhafte Zahl an Skills. Soll der skill-basierte Ansatz für eine breite Masse an Montageanwendungen eingesetzt werden, so stellt sich letztlich nicht nur die Frage, wie Skills effektiv eingesetzt werden können, sondern insbesondere auch, wie sie effizient erstellt und erweitert werden können.

1.3 Ziele und Ansatz der Arbeit

Basierend auf der Problemstellung und dem Stand der Technik lautet die zentrale Fragestellung, der sich diese Arbeit widmet:

> *Wie können kraftgeregelte Montageprozesse modelliert werden, um neue Bauteilvarianten, Prozesse und Hardwarekomponenten mit einem hohen Maß an Wiederverwendbarkeit abzubilden? Wie lässt sich dabei die Wiederverwendung modellieren?*

Im Folgenden wird diese Fragestellung auf einzelne Ziele heruntergebrochen, die dann in den entsprechenden Kapiteln der Arbeit aufgegriffen werden.

Mit dem Anspruch der Wiederverwendbarkeit stellt sich zunächst die Frage, welche Bestandteile des Systems wiederverwendet werden sollen und in welcher Granularität. Für ein hohes Maß an Wiederverwendbarkeit reicht es nicht aus, sich auf die Wiederverwendung von vorgefertigten Skills zu beschränken. Jeder neu benötigte Skill müsste durch Experten erstellt werden. Die Vielzahl an existierenden Montageprozessen und deren Vielseitigkeit machen es jedoch unwirtschaftlich, eine abgeschlossene Liste aller je benötigter Skills zu erstellen. Ziel ist vielmehr die Erstellung eines Baukastensystems, das es erlaubt, neue Skills anwendungsspezifisch aus vorhandenen Skills oder Teilen dieser zusammenzusetzen. Damit wird eine Wiederverwendung der einzelnen Bestandteile von Skills ermöglicht, nicht nur der kompletten Skills. Nach Darlegung der Grundlagen in Kapitel 2 betrachtet Kapitel 3, wie dieses Ziel und dessen Teilziele umgesetzt werden können:

- *Identifikation und Klassifizierung der* ***elementaren Bausteine****, aus denen Skills modelliert werden*

- *Aufstellung von **Regeln zur Komposition** von Skills aus einzelnen Bausteinen und durch Kombination bereits existierender Skills*
- *Implementierung eines **Mechanismus zur Ausführung** der modellierten Skills und Anwendungen*

Dabei bietet sich eine Aufteilung der elementaren Bausteine in die Bereiche Hardwareabstraktion, kinematische Modellierung, Aufgabenspezifikation mit Reglern und Stoppbedingungen sowie Koordination der einzelnen Skills an. Da jeder Baustein nur einen dieser Teilaspekte modelliert, wird seine Wiederverwendung erleichtert. Zur Ausführung wird der iTaSC-Formalismus gewählt, der eine Komposition partieller Aufgabenspezifikationen erlaubt.

Die Erstellung eines Baukastensystems allein reicht nicht aus, um die eingangs beschriebene Problemstellung zufriedenstellend zu lösen. Es stellt sich auch die Frage, wie die Bausteine des Systems effektiv eingesetzt werden können, wie der Baukasten effizient um neue Bausteine erweitert werden kann und welche fachlichen Kenntnisse dazu jeweils nötig sind. Die Nutzung eines skill-basierten Modells sollte nicht ein mühseliges Ausfüllen automatisch generierter Templates oder ein intensives Duplizieren und Anpassen von Skill-Definitionen nach sich ziehen. Auch sollte kein sogenannter *Glue Code* nötig sein, um einzelne Bausteine anwendungsspezifisch zusammenzubringen. Das zweite Ziel der Arbeit ist daher die Einführung eines Skill-Modells, das die Wiederverwendung und Erweiterung der einzelnen Bausteine explizit modelliert. Ein in der objektorientierten Programmierung etabliertes Konzept hierfür ist die Vererbung. In dieser Arbeit wird vorgeschlagen, eine prototypbasierte Vererbung zu nutzen, um Skills und Applikationen mit wenigen Änderungen an neue Bauteilvarianten, Prozesse und Hardwarekomponenten anzupassen. Kapitel 4 befasst sich daher mit dem Entwurf eines Skill-Modells zur Komposition und Spezialisierung von Bausteinen:

- *Explizite Modellierung der Wiederverwendung und Erweiterung einzelner Bausteine mithilfe **prototypbasierter Vererbung** und **Komposition***
- *Gestaltung einer einfachen **domänenspezifischen Sprache**, mit der das Skill-Modell effizient und ohne Programmierkenntnisse genutzt werden kann*
- ***Inkrementelle Erstellung von Skills** mithilfe der domänenspezifischen Sprache*

Der letzte Teil der Arbeit befasst sich schließlich damit, die Nützlichkeit des Ansatzes nachzuweisen und seine Grenzen aufzuzeigen. Eine Reihe an Anwendungsbeispielen zeigt, wie sich komplexe Montageprozesse in Subprozesse zerteilen und unter Verwendung der erstellten Skills

robust ausführen lassen. Darüber hinaus wird der Grad der Wiederverwendung und Anpassbarkeit von Skills betrachtet, insbesondere inwieweit Skills anwendungsspezifisch, variantenspezifisch und hardwarespezifisch sind. Kapitel 5 evaluiert dabei das Maß, in dem die folgenden Punkte erreicht werden:

- *Das Baukastensystem erlaubt die Erstellung einer umfassenden* ***Bibliothek an Skills*** *und anderen Bausteinen, die sich zu neuen, spezialisierteren Skills kombinieren lassen.*
- *Es ist möglich, komplexe Montage-Skills auf Basis einfacher Bausteine zu erstellen und damit* ***kraftgeregelte Montageaufgaben umzusetzen*** *und* ***robust auszuführen****.*
- *Neue Bauteilvarianten und Prozesse oder der Wechsel von Hardwarekomponenten sind durch wenige Anpassungen von Parametern oder zumindest mit einem* ***hohen Grad an Wiederverwendung*** *bestehender Elemente umsetzbar.*

Sprachliche Anmerkung: Umgangssprachlich werden «parametrisieren» (ein Modell mit Parametern versehen) und «parametrieren» (den Parametern Werte zuteilen) oft synonym verwendet. Da der Unterschied in dieser Arbeit relevant ist, werden die Begriffe hier den genannten Bedeutungen entsprechend verwendet.

2 Grundlagen

Die Programmierung sensorbasierter Roboteranwendungen erfordert Kenntnisse in unterschiedlichen Disziplinen. Hierzu zählen insbesondere die Kinematik, Regelungstechnik und Softwareentwicklung. Dieses Kapitel beschreibt ausgewählte Begriffe, Methoden und Formeln aus diesen Gebieten, die dieser Arbeit zugrunde liegen.

Zunächst werden in Abschnitt 2.1 kinematische Grundlagen zusammengefasst, die es erlauben, die Gelenkgeschwindigkeiten eines seriellen Roboters so zu wählen, dass der Roboter eine gewünschte kartesische Bewegung ausführt. Abschnitt 2.2 gibt einen kurzen Einblick in die Methoden der Kraftregelung, mit denen Roboter direkt oder indirekt eine Kraft auf ihre Umgebung ausüben können. Abschnitt 2.3 führt in die Architektur und Implementierung sensorbasierter Robotersysteme ein. Abschließend wird in Abschnitt 2.4 der iTaSC-Formalismus vorgestellt, der für diese Arbeit zur Spezifikation von Roboterbewegungen ausgewählt wurde.

2.1 Grundlagen der Kinematik

Mithilfe der Kinematik lässt sich der geometrische Aufbau der Gelenke eines Roboters und deren Bewegung beschreiben. Von besonderem Interesse ist hierbei der Zusammenhang zwischen den Gelenkwinkeln und der sich daraus ergebenden Pose des Roboterwerkzeugs. Ebenso von Bedeutung ist es, auf Ebene der Geschwindigkeiten den Zusammenhang zwischen Gelenkgeschwindigkeiten und der Geschwindigkeit des Roboterwerkzeugs darzustellen.

In diesem Abschnitt werden dazu die Grundlagen für die kinematischen Berechnungen serieller Roboter beschrieben. Der Abschnitt dient gleichzeitig als Einführung in die Notation der Arbeit, die großteils Siciliano et al. (2010) folgt. Für skalare Größen und Vektoren werden dabei Kleinbuchstaben verwendet, für Matrizen Großbuchstaben. Vektoren und Matrizen werden durch Fettdruck hervorgehoben.

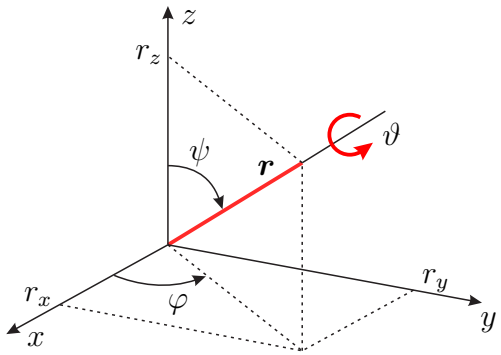


Abbildung 2.1: Repräsentation der Orientierung in Form von Drehachse und Drehwinkel

2.1.1 Kinematik

Das körperfeste Koordinatensystem eines Objekts, oft auch kurz Frame genannt (von engl. *coordinate frame* für Koordinatensystem), beschreibt die Lage des Objekts im Raum. Es kann in Form der homogenen Transformationsmatrix

$$\boldsymbol{T} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{p} \\ \boldsymbol{0}^\mathsf{T} & 1 \end{bmatrix} \tag{2.1}$$

in Relation zu einem Bezugskoordinatensystem angegeben werden. Soweit aus dem Kontext nicht ersichtlich, wird für Matrizen ebenso wie für Vektoren der Name des Bezugskoordinatensystems hochgestellt angegeben, der Name des Koordinatensystems tiefgestellt. So beschreibt beispielsweise ${}^0_1\boldsymbol{T}$ die Transformation des Koordinatensystems O_1 zum Bezugskoordinatensystem O_0.

Der Ortsvektor

$$\boldsymbol{p} = \begin{bmatrix} x & y & z \end{bmatrix}^\mathsf{T} \tag{2.2}$$

beschreibt die Position des Ursprungs des körperfesten Koordinatensystems, die Drehmatrix $\boldsymbol{R}$ die Richtung der Basisvektoren und somit dessen Orientierung. Die Inverse der homogenen Transformationsmatrix

$$\boldsymbol{T}^{-1} = \begin{bmatrix} \boldsymbol{R}^\mathsf{T} & -\boldsymbol{R}^\mathsf{T}\boldsymbol{p} \\ \boldsymbol{0}^\mathsf{T} & 1 \end{bmatrix} \tag{2.3}$$

lässt sich sehr einfach bilden, da für Drehmatrizen $\boldsymbol{R}^{-1} = \boldsymbol{R}^\mathsf{T}$ gilt.

Eine zur Drehmatrix alternative Darstellung der Orientierung geschieht über die Angabe einer Drehachse und eines Drehwinkels, wie in Abbildung 2.1 dargestellt. Sind die Drehachse $\boldsymbol{r}$ und der Drehwinkel ϑ gegeben, so lässt sich die Drehmatrix $\boldsymbol{R}(\vartheta, \boldsymbol{r})$ durch Verkettung mehrerer Elementardrehungen bestimmen (Siciliano et al. 2010). Zunächst wird dabei die z-Achse an

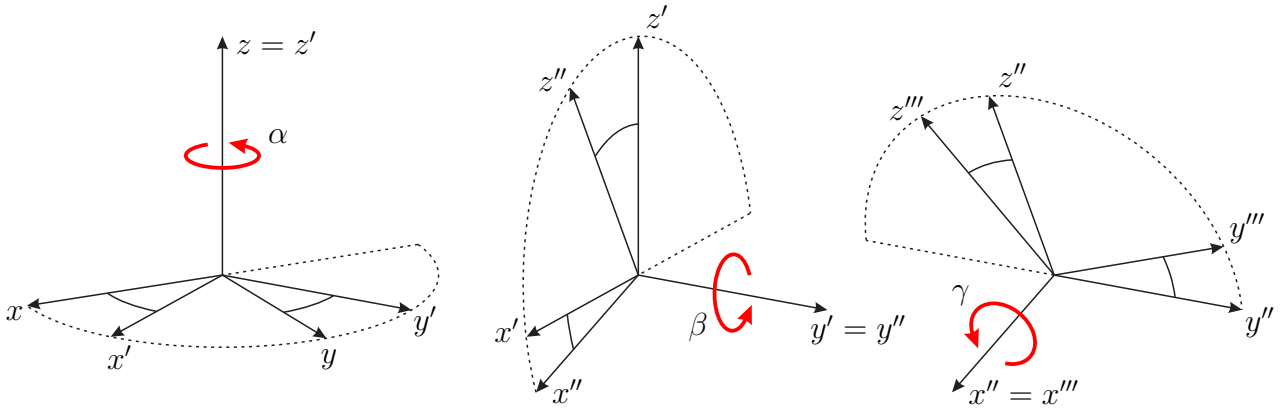


Abbildung 2.2: Repräsentation der Orientierung in Form von Z-Y-X Kardan-Winkeln

Vektor $\boldsymbol{r}$ ausgerichtet. Nach der anschließenden Drehung mit ϑ, wird die z-Achse wieder entsprechend ihrer ursprünglichen Orientierung bezüglich $\boldsymbol{r}$ ausgerichtet. Nach Elimination von φ und ψ erhält man die Drehmatrix

$$\boldsymbol{R}(\vartheta, \boldsymbol{r}) = \boldsymbol{R}_Z(\varphi)\boldsymbol{R}_Y(\psi)\boldsymbol{R}_Z(\vartheta)\boldsymbol{R}_Y(-\psi)\boldsymbol{R}_Z(-\varphi) \tag{2.4}$$

$$= \begin{bmatrix} r_x^2\,(1-c_\vartheta)+c_\vartheta & r_x r_y\,(1-c_\vartheta)-r_z s_\vartheta & r_x r_z\,(1-c_\vartheta)+r_y s_\vartheta \\ r_x r_y\,(1-c_\vartheta)+r_z s_\vartheta & r_y^2\,(1-c_\vartheta)+c_\vartheta & r_y r_z\,(1-c_\vartheta)-r_x s_\vartheta \\ r_x r_z\,(1-c_\vartheta)-r_y s_\vartheta & r_y r_z\,(1-c_\vartheta)+r_x s_\vartheta & r_z^2\,(1-c_\vartheta)+c_\vartheta \end{bmatrix}, \tag{2.5}$$

mit den Kurzschreibweisen $c_\vartheta = \cos(\vartheta)$ und $s_\vartheta = \sin(\vartheta)$.

Für Drehachse und Drehwinkel müssen in Summe vier Werte angegeben werden. In manchen Darstellungen wird die Drehachse auch normiert und mit dem Drehwinkel ϑ multipliziert. So kann die Orientierung mit nur drei Werten angegeben werden, im Gegensatz zu den neun Richtungskosinussen der Drehmatrix. Ist der Drehwinkel gleich null, so ist die Drehachse jedoch nicht eindeutig.

Ebenfalls möglich ist eine Darstellung der Orientierung durch drei Drehwinkel. Einfache Drehungen sind für Anwender damit leicht händisch angebbar. Als Drehwinkel werden in dieser Arbeit die Z-Y-X Kardan-Winkel

$$\boldsymbol{\phi} = \begin{bmatrix} \alpha & \beta & \gamma \end{bmatrix}^\mathsf{T} \tag{2.6}$$

verwendet. Diese werden in der Literatur häufig auch als Roll-Nick-Gier-Winkel (engl. *roll-pitch-yaw angles*) bezeichnet, jedoch teils unterschiedlich definiert. Zu beachten sind hier die Reihenfolge der Drehungen und das jeweilige Bezugssystem. Bei der hier verwendeten Konvention wird zunächst mit dem Winkel α um die z-Achse gedreht, anschließend mit dem Winkel β um die bereits gedrehte y-Achse und schließlich mit dem Winkel γ um die zweimal gedrehte

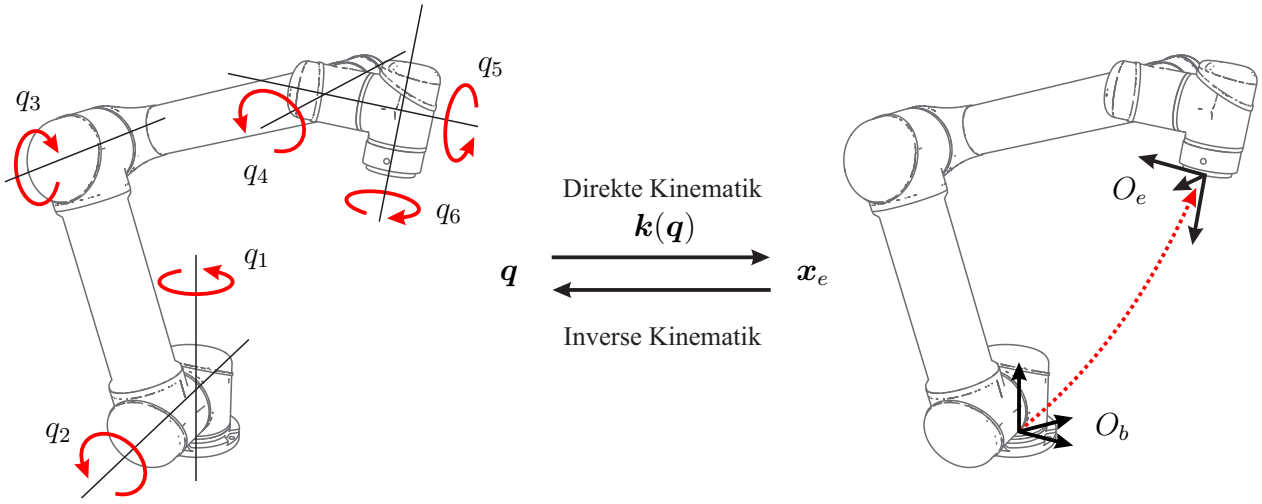


Abbildung 2.3: Direkte und inverse Kinematik

x-Achse (Craig 2005). Abbildung 2.2 stellt diese Abfolge bildlich dar. Durch Verkettung der Elementardrehungen erhält man entsprechend die Drehmatrix

$$\boldsymbol{R}(\boldsymbol{\phi}) = \boldsymbol{R}_Z(\alpha)\boldsymbol{R}_Y(\beta)\boldsymbol{R}_X(\gamma)\,, \quad \text{mit} \quad \boldsymbol{\phi} = \begin{bmatrix}\alpha & \beta & \gamma\end{bmatrix}^\mathsf{T}\,. \tag{2.7}$$

Wie jede Darstellung mit drei Werten ist auch diese nicht immer eindeutig. Für $\beta = 0$ drehen die Winkel α und γ um dieselbe Achse, es kommt zum Verlust eines Freiheitsgrades und man spricht von einer Rahmensperre (engl. *gimbal lock*). Der Problematik kann durch die Verwendung von Quaternionen entgangen werden (Siciliano et al. 2010). So stellt Stolt et al. (2012c) eine Formulierung des später in Abschnitt 2.4 eingeführten iTaSC-Algorithmus mithilfe von Quaternionen vor.

Als Kombination aus Positionen und Kardan-Winkeln parametrisiert

$$\boldsymbol{x} = \begin{bmatrix}\boldsymbol{p}\\ \boldsymbol{\phi}\end{bmatrix} = \begin{bmatrix}x & y & z & \alpha & \beta & \gamma\end{bmatrix}^\mathsf{T} \tag{2.8}$$

die Pose des Koordinatensystems im Raum. Sie stellt somit eine minimale Parametrisierung der homogenen Transformationsmatrix dar.

Die aktuelle Pose $\boldsymbol{x}_e$ des Endeffektors wird bei gegebenen n Gelenkkoordinaten

$$\boldsymbol{q} = \begin{bmatrix}q_1 & q_2 & \dots & q_n\end{bmatrix}^\mathsf{T} \tag{2.9}$$

durch Lösen des Problems der Vorwärtskinematik oder direkten Kinematik

$$\boldsymbol{x}_e = \boldsymbol{k}(\boldsymbol{q}) \tag{2.10}$$

bestimmt. Die Vorwärtskinematik bildet demnach den Raum der Gelenkkoordinaten (engl. *joint space*, *configuration space*) in den kartesischen Raum ab (auch Aufgabenraum, engl. *operational space*, *task space*), wie in Abbildung 2.3 dargestellt. Gelöst wird die Vorwärtskinematik meist in Matrixform, über die Verkettung der einzelnen Gelenktransformationen

$$ {}^{b}_{e}\boldsymbol{T}(\boldsymbol{q}) = {}^{b}_{0}\boldsymbol{T}\,{}^{0}_{1}\boldsymbol{T}(q_1)\,{}^{1}_{2}\boldsymbol{T}(q_2)\ldots{}^{n-1}_{n}\boldsymbol{T}(q_n)\,{}^{n}_{e}\boldsymbol{T}\,, \tag{2.11} $$

mit den konstanten Transformationen ${}^{b}_{0}\boldsymbol{T}$ von der Roboterbasis zum ersten Gelenk und ${}^{n}_{e}\boldsymbol{T}$ vom letzten Gelenk zum Endeffektor.

Die inverse Kinematik bestimmt die einzustellenden Gelenkkoordinaten für eine gegebene Endeffektorlage. Sie ist für serielle Roboter schwieriger zu lösen als die Vorwärtskinematik, da es zu Mehrdeutigkeiten und Singularitäten kommen kann. Benötigt wird sie in dieser Arbeit jedoch nur auf der im Vergleich zur Positionsebene einfacher zu lösenden Geschwindigkeitsebene. Im folgenden Abschnitt wird aus diesem Grund zunächst die differentielle Kinematik eingeführt und anschließend zur Lösung der Inverskinematik zurückgekehrt.

2.1.2 Differentielle Kinematik

Die differentielle Kinematik von Robotern beschreibt den Zusammenhang zwischen der kartesischen Geschwindigkeit des Endeffektors und den Gelenkgeschwindigkeiten des Roboters. Hierbei lassen sich für Rotationsgeschwindigkeiten unterschiedliche Darstellungen wählen, auf die zunächst eingegangen wird.

Darstellung von Geschwindigkeitsvektoren

Durch zeitliche Ableitung der Pose aus Gleichung (2.8) erhält man den Vektor der translatorischen und rotatorischen Geschwindigkeiten

$$ \dot{\boldsymbol{x}} = \begin{bmatrix} \dot{\boldsymbol{p}} \\ \dot{\boldsymbol{\phi}} \end{bmatrix} = \begin{bmatrix} \dot{x} & \dot{y} & \dot{z} & \dot{\alpha} & \dot{\beta} & \dot{\gamma} \end{bmatrix}^{\mathsf{T}} \tag{2.12} $$

des Endeffektors. Eine alternative Darstellung verwendet zur Repräsentation des rotatorischen Anteils die Winkelgeschwindigkeiten

$$ \boldsymbol{\omega} = \begin{bmatrix} \omega_x & \omega_y & \omega_z \end{bmatrix}^{\mathsf{T}}\,, \tag{2.13} $$

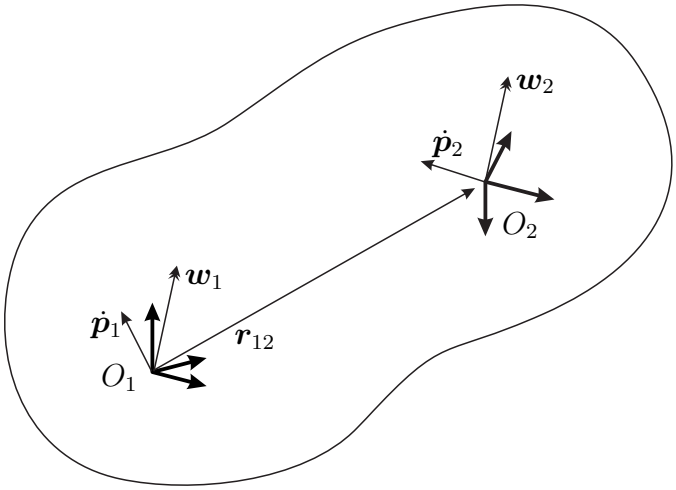


Abbildung 2.4: Transformation des Geschwindigkeitsvektors

wobei der resultierende verallgemeinerte Geschwindigkeitsvektor

$$\boldsymbol{v} = \begin{bmatrix} \dot{\boldsymbol{p}} \\ \boldsymbol{\omega} \end{bmatrix} = \begin{bmatrix} \dot{x} & \dot{y} & \dot{z} & \omega_x & \omega_y & \omega_z \end{bmatrix}^{\mathsf{T}} \tag{2.14}$$

auch unter dem Namen Geschwindigkeitswinder (engl. *twist*) verwendet wird. Die beiden Darstellungen lassen sich über die Transformation

$$\boldsymbol{\omega} = \boldsymbol{E}(\boldsymbol{\phi})\dot{\boldsymbol{\phi}} \tag{2.15}$$

mit

$$\boldsymbol{E}(\boldsymbol{\phi}) = \begin{bmatrix} 0 & -s_\alpha & c_\alpha c_\beta \\ 0 & c_\alpha & s_\alpha c_\beta \\ 1 & 0 & -s_\beta \end{bmatrix} \tag{2.16}$$

ineinander überführen, die für die Parametrisierung der Rotation mit Z-Y-X Kardan-Winkeln gilt. Die Rücktransformation mit

$$\boldsymbol{E}^{-1}(\boldsymbol{\phi}) = \begin{bmatrix} \frac{c_\alpha s_\beta}{c_\beta} & \frac{s_\alpha s_\beta}{c_\beta} & 1 \\ -s_\alpha & c_\alpha & 0 \\ \frac{c_\alpha}{c_\beta} & \frac{s_\alpha}{c_\beta} & 0 \end{bmatrix} \tag{2.17}$$

ist für $\beta = -\pi$ und $\beta = \pi$ nicht definiert.

Koordinatentransformation von Geschwindigkeitsvektoren

Zur Darstellung eines Geschwindigkeitsvektors in einem anderen Bezugskoordinatensystem

$$\begin{bmatrix} {}^2\dot{\boldsymbol{p}} \\ {}^2\boldsymbol{\omega} \end{bmatrix} = \begin{bmatrix} {}^2_1\boldsymbol{R} & \boldsymbol{0} \\ \boldsymbol{0} & {}^2_1\boldsymbol{R} \end{bmatrix} \begin{bmatrix} {}^1\dot{\boldsymbol{p}} \\ {}^1\boldsymbol{\omega} \end{bmatrix} \tag{2.18}$$

lassen sich der translatorische und rotatorische Anteil jeweils getrennt mithilfe der Drehmatrix transformieren.

Häufig ist es von Interesse, die Geschwindigkeit in einem Punkt $\boldsymbol{p}_2$ eines bewegten starren Körpers anhand der Geschwindigkeit in einem anderen Punkt $\boldsymbol{p}_1$ desselben starren Körpers zu bestimmen, wie in Abbildung 2.4 dargestellt. Dies lässt sich über die Beziehung

$$\begin{aligned} \dot{\boldsymbol{p}}_2 &= \dot{\boldsymbol{p}}_1 + \boldsymbol{\omega}_1 \times \boldsymbol{r}_{12} && (2.19)\\ &= \dot{\boldsymbol{p}}_1 - \boldsymbol{r}_{12} \times \boldsymbol{\omega}_1 && (2.20)\\ &= \dot{\boldsymbol{p}}_1 - \left[\boldsymbol{r}_{12}\right]_\times \boldsymbol{\omega}_1 \,, && (2.21)\\ \boldsymbol{\omega}_2 &= \boldsymbol{\omega}_1 && (2.22) \end{aligned}$$

lösen. In Gleichung (2.20) wird dazu die Antikommutativität des Kreuzprodukts genutzt. Anschließend wird das Kreuzprodukt in Gleichung (2.21) durch die schiefsymmetrische Kreuzproduktmatrix

$$\left[\boldsymbol{r}_{12}\right]_\times = \begin{bmatrix} 0 & -r_{12,z} & r_{12,y} \\ r_{12,z} & 0 & -r_{12,x} \\ -r_{12,y} & r_{12,x} & 0 \end{bmatrix} \tag{2.23}$$

ersetzt. Durch die Umstellung lässt sich die Transformation von translatorischer und rotatorischer Geschwindigkeit in Matrixform zu

$$\begin{bmatrix} \dot{\boldsymbol{p}}_2 \\ \boldsymbol{\omega}_2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{I} & -\left[\boldsymbol{r}_{12}\right]_\times \\ \boldsymbol{0} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \dot{\boldsymbol{p}}_1 \\ \boldsymbol{\omega}_1 \end{bmatrix} \tag{2.24}$$

zusammenfassen, mit der 3×3 Einheitsmatrix $\boldsymbol{I}$. Durch Zusammenführung mit Gleichung (2.18) ergibt sich die vollständige Transformation

$$\begin{bmatrix} {}^2\dot{\boldsymbol{p}}_2 \\ {}^2\boldsymbol{\omega}_2 \end{bmatrix} = \begin{bmatrix} {}^2_1\boldsymbol{R} & -{}^2_1\boldsymbol{R}\left[{}^1\boldsymbol{r}_{12}\right]_\times \\ \boldsymbol{0} & {}^2_1\boldsymbol{R} \end{bmatrix} \begin{bmatrix} {}^1\dot{\boldsymbol{p}}_1 \\ {}^1\boldsymbol{\omega}_1 \end{bmatrix} \tag{2.25}$$

zur Berechnung der Geschwindigkeit im Punkt $\boldsymbol{p}_2$ bezüglich Koordinatensystem O_2 bei gegebener Geschwindigkeit im Punkt $\boldsymbol{p}_1$ bezüglich Koordinatensystem O_1.

Ist $\boldsymbol{r}$ bezüglich O_2 und in entgegengesetzter Richtung gegeben, von Punkt $\boldsymbol{p}_2$ nach Punkt $\boldsymbol{p}_1$, so kann die Gleichung mit

$$\left[{}^1\boldsymbol{r}_{12}\right]_\times = -{}^1_2\boldsymbol{R}\left[{}^2\boldsymbol{r}_{21}\right]_\times {}^2_1\boldsymbol{R} \tag{2.26}$$

umgestellt werden zu

$$\begin{bmatrix} {}^2\dot{\boldsymbol{p}}_2 \\ {}^2\boldsymbol{\omega}_2 \end{bmatrix} = \begin{bmatrix} {}^2_1\boldsymbol{R} & \left[{}^2\boldsymbol{r}_{21}\right]_\times {}^2_1\boldsymbol{R} \\ \boldsymbol{0} & {}^2_1\boldsymbol{R} \end{bmatrix} \begin{bmatrix} {}^1\dot{\boldsymbol{p}}_1 \\ {}^1\boldsymbol{\omega}_1 \end{bmatrix}, \tag{2.27}$$

wobei die hierfür benötigten Größen direkt der homogenen Transformationsmatrix

$${}^2_1\boldsymbol{T} = \begin{bmatrix} {}^2_1\boldsymbol{R} & {}^2\boldsymbol{r}_{21} \\ \boldsymbol{0}^\mathsf{T} & 1 \end{bmatrix} \tag{2.28}$$

entnommen werden können.

Geometrische Jacobi-Matrix

Die geometrische Jacobi-Matrix $\boldsymbol{J}$ beschreibt den Zusammenhang

$$\boldsymbol{v}_e = \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}} \tag{2.29}$$

zwischen den Gelenkgeschwindigkeiten des Roboters $\dot{\boldsymbol{q}}$ und der kartesischen Geschwindigkeit des Endeffektors $\boldsymbol{v}_e$. Interessant ist, dass dieser Zusammenhang linear ist. Es ist jedoch zu beachten, dass die Jacobi-Matrix von der aktuellen Roboterkonfiguration $\boldsymbol{q}$ abhängig ist.

Anschaulich bezeichnen die Spaltenvektoren $\boldsymbol{J}_i$ der Jacobi-Matrix[2]

$$\boldsymbol{J} = \begin{bmatrix} \boldsymbol{J}_1 & \boldsymbol{J}_2 & \dots & \boldsymbol{J}_n \end{bmatrix} \tag{2.30}$$

den Einfluss jedes einzelnen Gelenks auf die kartesische Geschwindigkeit des Endeffektors. Wie nach Ausmultiplizieren der Spaltenvektoren

$$\begin{aligned} \boldsymbol{v}_e &= \begin{bmatrix} \boldsymbol{J}_1 & \boldsymbol{J}_2 & \dots & \boldsymbol{J}_n \end{bmatrix} \dot{\boldsymbol{q}} && (2.31) \\ &= \boldsymbol{J}_1\dot{q}_1 + \boldsymbol{J}_2\dot{q}_2 + \cdots + \boldsymbol{J}_n\dot{q}_n && (2.32) \end{aligned}$$

ersichtlich wird, trägt ein Gelenk i mit der Geschwindigkeit

$$\boldsymbol{v}_i = \boldsymbol{J}_i\dot{q}_i \tag{2.33}$$

[2] Abweichend von der eingeführten Notation werden Jacobi-Matrizen stets einheitlich mit Großbuchstaben dargestellt, selbst wenn sie nur aus einem einzelnen Spaltenvektor bestehen.

zur kartesischen Geschwindigkeit des Endeffektors $\boldsymbol{v}_e$ bei. Es wird auch ersichtlich, dass die geometrische Herleitung der Jacobi-Matrix über die Zusammenfassung der Jacobi-Matrizen der einzelnen Robotergelenke $\boldsymbol{J}_i$ (jeweils ein 6×1 Spaltenvektor) erfolgen kann.

Diese müssen jedoch zunächst in ein einheitliches Koordinatensystem transformiert werden. Die dafür benötigte Transformation lässt sich mithilfe der Gleichungen (2.27) und (2.33) herleiten. Sei der Geschwindigkeitsbeitrag von Gelenk i bezüglich des Koordinatensystems O_1 gegeben, so kann er in das Koordinatensystem O_2 mit

$$
\begin{aligned}
{}^2\boldsymbol{v}_i &= \begin{bmatrix} {}^2_1\boldsymbol{R} & \left[{}^2\boldsymbol{r}_{21}\right]_\times {}^2_1\boldsymbol{R} \\ \boldsymbol{0} & {}^2_1\boldsymbol{R} \end{bmatrix} {}^1\boldsymbol{v}_i && (2.34) \\
&= \begin{bmatrix} {}^2_1\boldsymbol{R} & \left[{}^2\boldsymbol{r}_{21}\right]_\times {}^2_1\boldsymbol{R} \\ \boldsymbol{0} & {}^2_1\boldsymbol{R} \end{bmatrix} {}^1\boldsymbol{J}_i \dot{q}_i && (2.35)
\end{aligned}
$$

transformiert werden, woraus direkt

$$
{}^2\boldsymbol{J}_i = \begin{bmatrix} {}^2_1\boldsymbol{R} & \left[{}^2\boldsymbol{r}_{21}\right]_\times {}^2_1\boldsymbol{R} \\ \boldsymbol{0} & {}^2_1\boldsymbol{R} \end{bmatrix} {}^1\boldsymbol{J}_i \tag{2.36}
$$

folgt.

Es sei abschließend angemerkt, dass sich bei der Verwendung von $\dot{\boldsymbol{x}}_e$ anstelle des Geschwindigkeitsvektors $\boldsymbol{v}_e$ eine alternative Form der Jacobi-Matrix ergibt, die sogenannte analytische Jacobi-Matrix.

Durch Inversion der Jacobi-Matrix lässt sich schließlich die Inverskinematik des Roboters auf Geschwindigkeitsebene lösen. Sie wird im folgenden Abschnitt betrachtet.

2.1.3 Inverse differentielle Kinematik

Die Inverskinematik auf Ebene der Geschwindigkeiten dient der Bestimmung der Gelenkgeschwindigkeiten $\dot{\boldsymbol{q}}$, die zur Ausführung einer gegebenen kartesischen Bewegung $\boldsymbol{v}$ des Endeffektors nötig sind. Sie erfolgt durch das Auflösen des linearen Gleichungssystems

$$
\boldsymbol{v} = \boldsymbol{J}\dot{\boldsymbol{q}} \tag{2.37}
$$

nach $\dot{\boldsymbol{q}}$. Ist die Jacobi-Matrix $\boldsymbol{J}$ quadratisch und besitzt vollen Rang, lässt sich die Gleichung

$$
\dot{\boldsymbol{q}} = \boldsymbol{J}^{-1}\boldsymbol{v} \tag{2.38}
$$

durch einfache Inversion von $\boldsymbol{J}$ bestimmen. In einer Singularität verliert die Jacobi-Matrix jedoch ihren vollen Rang und ist nicht mehr invertierbar. Anschaulich existiert mindestens eine kartesische Richtung, in die sich der Roboter nicht bewegen kann. Dies ist beispielsweise in Strecklagen der Fall oder wenn zwei Achsen des Roboters auf eine Linie fallen. Nahe den Singularitäten können zudem bereits kleine Bewegungen im kartesischen Raum sehr große Bewegungen im Raum der Gelenkkoordinaten erfordern, die die zulässigen Gelenkgeschwindigkeiten weit übersteigen.

Des Weiteren existieren Fälle, in denen die Jacobi-Matrix nicht quadratisch ist. Wenn die Anzahl der Gelenke n des Roboters größer ist als die Dimension m der spezifizierten Aufgabe, so ist das Gleichungssystem unterbestimmt. Dies ist beispielsweise bei einem Schraubprozess der Fall, bei dem die Orientierung des Werkzeugs um die Schraubachse für den Prozess nicht relevant ist. Bei redundanten Robotern (Chiaverini et al. 2016) mit sieben oder mehr Achsen ist n generell größer als m. Ebenso ist die Jacobi-Matrix nicht quadratisch, wenn die spezifizierte Aufgabe überbestimmt ist und somit nicht alle Teilaufgaben gleichzeitig erfüllt werden können. Das Gleichungssystem (2.37) ist dann im Allgemeinen nicht mehr exakt lösbar. Bevor Lösungsansätze für diese Fälle betrachtet werden, wird zunächst auf eine Problematik hingewiesen, die bei der Herleitung der Lösungsansätze auftritt.

Gewichtung der euklidischen Norm

Zur Lösung des Gleichungssystems (2.37) in Fällen, bei denen die Jacobi-Matrix keinen vollen Rang besitzt, werden euklidische Normen verwendet, um ausgewählte Kriterien zu optimieren. Wie beispielsweise von Duffy (1990) und Doty et al. (1993) gezeigt wird, sind Formulierungen wie $\|\dot{\boldsymbol{q}}\|_2$ oder $\|\boldsymbol{v}\|_2$ jedoch im Allgemeinen physikalisch inkorrekt. Dies ist leicht zu erkennen, wenn man die Berechnung der euklidischen Norm des Geschwindigkeitsvektors

$$\|\boldsymbol{v}\|_2 = \sqrt{\boldsymbol{v}^\mathsf{T}\boldsymbol{v}} \tag{2.39}$$

$$= \sqrt{\dot{\boldsymbol{p}}^\mathsf{T}\dot{\boldsymbol{p}} + \boldsymbol{\omega}^\mathsf{T}\boldsymbol{\omega}} \tag{2.40}$$

betrachtet, wo Größen der Einheiten $(\mathrm{m/s})^2$ und $(\mathrm{rad/s})^2$ addiert werden. Die gleiche Problematik tritt für $\|\dot{\boldsymbol{q}}\|_2$ bei Robotern auf, die sowohl Schub- als auch Drehgelenke besitzen.

Gelöst wird das Problem durch die Transformation der Vektoren

$$\hat{\boldsymbol{v}} = \boldsymbol{N}_v\boldsymbol{v}\,, \tag{2.41}$$

$$\hat{\dot{\boldsymbol{q}}} = \boldsymbol{N}_q\dot{\boldsymbol{q}} \tag{2.42}$$

und der Jacobi-Matrix

$$\hat{\boldsymbol{J}} = \boldsymbol{N}_v \boldsymbol{J} \boldsymbol{N}_q^{-1}\,, \tag{2.43}$$

die man durch Einsetzen der beiden Gleichungen in (2.37) erhält (Nakamura et al. 1986). $\boldsymbol{N}_v$ und $\boldsymbol{N}_q$ ergeben sich aus den Gewichtungsmatrizen

$$\boldsymbol{W}_v = \boldsymbol{N}_v^\mathsf{T} \boldsymbol{N}_v\,, \tag{2.44}$$

$$\boldsymbol{W}_q = \boldsymbol{N}_q^\mathsf{T} \boldsymbol{N}_q\,, \tag{2.45}$$

wobei $\boldsymbol{W}_v$ die Dimension $m \times m$ und $\boldsymbol{W}_q$ die Dimension $n \times n$ besitzt. Die Gewichtungsmatrizen sind so zu wählen, dass die gewichtete euklidische Norm physikalisch korrekt ist. Eine geeignete Gewichtungsmatrix für den Geschwindigkeitsvektor lässt sich beispielsweise auf Basis der kinetischen Energie des Werkzeugs bestimmen, eine Gewichtungsmatrix für den Gelenkraum auf Basis der kinetischen Energie des gesamten Roboters (Doty et al. 1993; Bruyninckx et al. 2000; Whitney 1969).

Wie in den genannten Quellen hervorgehoben wird, ist die Wahl der Gewichtungsmatrizen stets anwendungsabhängig. Werden die Gewichtungsmatrizen weggelassen, so wird implizit $\boldsymbol{W}_v = \boldsymbol{I}_m$ beziehungsweise $\boldsymbol{W}_q = \boldsymbol{I}_n$ gewählt. Dies bedeutet, dass eine Abweichung von 1 rad $\approx 57{,}3°$ gleich einer Abweichung von einem Meter gewichtet wird (sofern rad und Meter die gewählten Maßeinheiten sind). Für viele Anwendungen ist diese Gewichtung zwar willkürlich aber akzeptabel, was erklären mag, warum die Problematik in der Literatur häufig übersehen oder vernachlässigt wird.

Durch die Herleitung

$$\begin{aligned}
\|\hat{\boldsymbol{v}}\|_2 &= \sqrt{(\boldsymbol{N}_v \boldsymbol{v})^\mathsf{T} (\boldsymbol{N}_v \boldsymbol{v})} && (2.46)\\
&= \sqrt{\boldsymbol{v}^\mathsf{T} \boldsymbol{N}_v^\mathsf{T} \boldsymbol{N}_v \boldsymbol{v}} && (2.47)\\
&= \sqrt{\boldsymbol{v}^\mathsf{T} \boldsymbol{W}_v \boldsymbol{v}} && (2.48)\\
&= \|\boldsymbol{v}\|_{\boldsymbol{W}_v} && (2.49)
\end{aligned}$$

ist zu erkennen, dass anstelle der gewichteten euklidischen Norm $\|\boldsymbol{v}\|_{\boldsymbol{W}_v}$ auch stets die ungewichtete euklidische Norm $\|\hat{\boldsymbol{v}}\|_2$ des transformierten Vektors verwendet werden kann. Im weiteren Text wird das Subskript $\|\cdot\|_2$ der ungewichteten euklidischen Norm weggelassen. Bei gewichteten euklidischen Normen wird die Gewichtungsmatrix als Subskript $\|\cdot\|_{\boldsymbol{W}}$ angegeben.

Durch Einsetzen der Gleichungen (2.41) und (2.42) in die inverse Kinematik

$$\hat{\dot{\boldsymbol{q}}} = \hat{\boldsymbol{J}}^\dagger \hat{\boldsymbol{v}} \tag{2.50}$$

ergibt sich der Zusammenhang zur gewichteten Inversen $\boldsymbol{J}^{\#}$ der Jacobi-Matrix

$$\dot{\boldsymbol{q}} = \boldsymbol{N}_q^{-1}\hat{\boldsymbol{J}}^{\dagger}\boldsymbol{N}_v\boldsymbol{v} \tag{2.51}$$
$$= \boldsymbol{J}^{\#}\boldsymbol{v} \tag{2.52}$$

oder explizit

$$\boldsymbol{J}^{\#} = \boldsymbol{N}_q^{-1}\hat{\boldsymbol{J}}^{\dagger}\boldsymbol{N}_v \tag{2.53}$$

für die gewichtete Inverse der Jacobi-Matrix $\boldsymbol{J}^{\#}$ und

$$\hat{\boldsymbol{J}}^{\dagger} = \boldsymbol{N}_q\boldsymbol{J}^{\#}\boldsymbol{N}_v^{-1} \tag{2.54}$$

für die gewichtete Inverse der transformierten Jacobi-Matrix $\hat{\boldsymbol{J}}^{\dagger}$. Da die genannten Jacobi-Matrizen in vielen Fällen nicht invertierbar sind, werden die Symbole † und $^{\#}$ zur Kennzeichnung von Pseudoinversen herangezogen. Möglichkeiten zur Bildung von Pseudoinversen werden im nachfolgenden Abschnitt vorgestellt.

Inversion der Jacobi-Matrix

Im Folgenden wird die Inversion der transformierten $m \times n$ Jacobi-Matrix $\hat{\boldsymbol{J}}$ für verschiedene Fälle betrachtet. Zur besseren Vergleichbarkeit der Lösungsansätze erfolgt die Herleitung stets in gleicher Form. Zunächst wird ein Optimierungsproblem auf Basis der mit den Gleichungen (2.41) bis (2.43) transformierten Vektoren aufgestellt und gelöst. Anschließend wird die Lösung zurück in die ursprünglichen Variablen transformiert, um die gewichtete Lösung zu erhalten.

Ist die Jacobi-Matrix quadratisch und regulär, das heißt $n = m = \text{Rang}(\hat{\boldsymbol{J}})$, so existiert genau eine Lösung des linearen Gleichungssystems aus (2.37), die mit der Inversen $\hat{\boldsymbol{J}}^{-1}$ der Jacobi-Matrix bestimmt werden kann.

Bei einer unterbestimmten Aufgabenspezifikation hat der Roboter mehr Freiheitsgrade als durch die Aufgabenspezifikation eingeschränkt werden und die Jacobi-Matrix besitzt mehr Spalten als Zeilen ($n > m$). Ist der Roboter in seiner aktuellen Gelenkkonfiguration nicht durch eine Singularität eingeschränkt, besitzt die Jacobi-Matrix vollen Zeilenrang $\text{Rang}(\hat{\boldsymbol{J}}) = m$ und es existieren unendlich viele Lösungen für (2.37). Diese Redundanz kann aufgelöst werden, indem

ein zusätzliches Kriterium optimiert wird. Eine häufige Formulierung des Optimierungsproblems erfolgt über die Lagrange-Funktion

$$\mathcal{L}(\hat{\dot{\boldsymbol{q}}}, \boldsymbol{\lambda}) = \left\|\hat{\dot{\boldsymbol{q}}}\right\|^2 + \boldsymbol{\lambda}^\mathsf{T}\left(\hat{\boldsymbol{v}} - \hat{\boldsymbol{J}}\hat{\dot{\boldsymbol{q}}}\right), \tag{2.55}$$

mit den Lagrange-Multiplikatoren $\boldsymbol{\lambda}$. Durch Minimierung der Funktion wird aus allen Lösungen für $\hat{\boldsymbol{v}} - \hat{\boldsymbol{J}}\hat{\dot{\boldsymbol{q}}} = \boldsymbol{0}$ diejenige Lösung ausgewählt, die die euklidische Norm der Gelenkgeschwindigkeiten $\left\|\hat{\dot{\boldsymbol{q}}}\right\|$ minimiert. Nach dem Lösen des Optimierungsproblems (Siciliano et al. 2010) erhält man die Rechtsinverse

$$\hat{\boldsymbol{J}}^\dagger = \hat{\boldsymbol{J}}^\mathsf{T}\left(\hat{\boldsymbol{J}}\hat{\boldsymbol{J}}^\mathsf{T}\right)^{-1} \tag{2.56}$$

der Jacobi-Matrix sowie nach dem Einsetzen von Gleichung (2.56) in (2.53) und mit dem Zusammenhang aus (2.43) schließlich die gewichtete Rechtsinverse

$$\boldsymbol{J}^\# = \boldsymbol{W}_q^{-1}\boldsymbol{J}^\mathsf{T}\left(\boldsymbol{J}\boldsymbol{W}_q^{-1}\boldsymbol{J}^\mathsf{T}\right)^{-1}. \tag{2.57}$$

Im Falle einer überbestimmten Aufgabenspezifikation besitzt die Jacobi-Matrix mehr Zeilen als Spalten ($m > n$) und es existiert im Allgemeinen keine exakte Lösung für (2.37). Von Interesse ist daher oft die approximierte Lösung, die den Fehler von $\hat{\boldsymbol{v}} - \hat{\boldsymbol{J}}\hat{\dot{\boldsymbol{q}}}$ in Form der Zielfunktion

$$f(\hat{\dot{\boldsymbol{q}}}) = \left\|\hat{\boldsymbol{v}} - \hat{\boldsymbol{J}}\hat{\dot{\boldsymbol{q}}}\right\|^2 \tag{2.58}$$

minimiert. Nach dem Lösen des Optimierungsproblems erhält man die Linksinverse

$$\hat{\boldsymbol{J}}^\dagger = \left(\hat{\boldsymbol{J}}^\mathsf{T}\hat{\boldsymbol{J}}\right)^{-1}\hat{\boldsymbol{J}}^\mathsf{T} \tag{2.59}$$

der Jacobi-Matrix beziehungsweise die gewichtete Linksinverse

$$\boldsymbol{J}^\# = \left(\boldsymbol{J}^\mathsf{T}\boldsymbol{W}_v\boldsymbol{J}\right)^{-1}\boldsymbol{J}^\mathsf{T}\boldsymbol{W}_v. \tag{2.60}$$

Sollte die Jacobi-Matrix weder vollen Zeilen- noch vollen Spaltenrang haben, so sind die Rechts- beziehungsweise Linksinversen nicht anwendbar. Daher wird im folgenden Abschnitt eine allgemeine Methode zur Bildung von Pseudoinversen vorgestellt.

Singulärwertzerlegung

Die Matrix $\hat{\boldsymbol{J}}$ lässt sich mithilfe der Singulärwertzerlegung (SVD, von engl. *singular value decomposition*) in die Form

$$\hat{\boldsymbol{J}} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\mathsf{T}} = \sum_{i=1}^{r} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^{\mathsf{T}} \tag{2.61}$$

mit $r = \mathrm{Rang}(\hat{\boldsymbol{J}})$ bringen (Yoshikawa 1985; Maciejewski et al. 1988; Maciejewski et al. 1989). Für die $m \times n$-Matrix $\hat{\boldsymbol{J}}$ sind die $m \times m$-Matrix $\boldsymbol{U}$ und die $n \times n$-Matrix $\boldsymbol{V}$ orthonormal. Die Matrix

$$\boldsymbol{\Sigma} = \begin{bmatrix} \boldsymbol{S} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \tag{2.62}$$

besitzt die Dimension $m \times n$ und enthält die $r \times r$ Diagonalmatrix

$$\boldsymbol{S} = \begin{bmatrix} \sigma_1 & & 0 \\ & \ddots & \\ 0 & & \sigma_r \end{bmatrix} \tag{2.63}$$

der r positiven Singulärwerte. Die Singulärwerte werden üblicherweise der Größe nach sortiert. Dabei ist σ_1 der größte Singulärwert, σ_r der kleinste (der nicht null ist). Anschaulich stellt σ_r das Verhältnis von kartesischer Geschwindigkeit zu Gelenkgeschwindigkeit in derjenigen Richtung dar, entlang der eine Bewegung am schwierigsten ist (Maciejewski et al. 1988). Das bedeutet, dass selbst für eine kleine kartesische Geschwindigkeit in diese Richtung mitunter sehr große Gelenkgeschwindigkeiten nötig sind. Nahe einer singulären Achskonfiguration ist mindestens ein Singulärwert klein, in einer singulären Achskonfiguration mindestens ein Singulärwert null. Mithilfe von Singulärwerten lässt sind demnach die Nähe zu einer Singularität bestimmen.

Durch die Zerlegung lässt sich zudem sehr leicht die Pseudoinverse

$$\hat{\boldsymbol{J}}^{\dagger} = \boldsymbol{V}\boldsymbol{\Sigma}^{\dagger}\boldsymbol{U}^{\mathsf{T}} = \sum_{i=1}^{r} \frac{1}{\sigma_i} \boldsymbol{v}_i \boldsymbol{u}_i^{\mathsf{T}} \tag{2.64}$$

bilden, mit

$$\boldsymbol{\Sigma}^{\dagger} = \begin{bmatrix} \boldsymbol{S}^{-1} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix}, \tag{2.65}$$

wonach nur von null verschiedene Singulärwerte invertiert werden. Diese Definition der Pseudoinversen ist auch dann gültig, wenn $\hat{\boldsymbol{J}}$ ihren vollen Zeilenrang, Spaltenrang oder beides verliert. Sie verallgemeinert demnach die Linksinverse, die vollen Spaltenrang erfordert, sowie die Rechtsinverse, die vollen Zeilenrang erfordert. Dies lässt sich durch Einsetzen von Gleichung (2.61) in die Gleichungen (2.56) und (2.59) zeigen, wobei sich in beiden Fällen die Lösung

aus (2.64) ergibt. Nachteilig kann bei einer Singulärwertzerlegung der höhere Rechenaufwand sein.

Mithilfe von Singulärwerten lässt sich das Verhalten in der Region um eine Singularität genauer untersuchen. Wenn der kleinste Singulärwert σ_r nahe einer Singularität gegen null strebt, so strebt der Wert

$$\frac{1}{\sigma_r} \tag{2.66}$$

der Pseudoinversen gegen unendlich. Selbst eine sehr kleine kartesische Geschwindigkeit des Endeffektors in die zugehörige Richtung erfordert sehr große Geschwindigkeiten der Antriebe, die die maximal möglichen Stellgrößen überschreiten können.

Eine einfache Methode, diese Verstärkung zu beschränken, setzt Singulärwerte null, die ein gewähltes Limit $\sigma_{\min}$ unterschreiten. Anschaulich gesprochen wird der zugehörige Freiheitsgrad «abgeschaltet», das heißt eine kartesische Bewegung in diese Richtung wird nicht weiter angestrebt, und die Dimension des Nullraums vergrößert sich (Aertbeliën 2009). Durch die Wahl von $\sigma_{\min}$ lassen sich die maximale Gelenkgeschwindigkeit und die Genauigkeit der ausgeführten kartesischen Geschwindigkeit gegeneinander abwägen. Der Nachteil dieser Methode ist der unstetige Übergang bei $\sigma_{\min}$. Eine Alternative dazu wird daher im Folgenden vorgestellt.

Damped Least-Squares Inverse

Der Levenberg-Marquardt-Algorithmus (engl. auch *damped least-squares*) ergänzt die Methode der kleinsten Quadrate um einen Dämpfungsfaktor (Nakamura et al. 1986; Wampler 1986). Anhand der Formulierung als zu minimierende Zielfunktion

$$f(\hat{\boldsymbol{q}}) = \left\| \hat{\boldsymbol{v}} - \hat{\boldsymbol{J}}\hat{\dot{\boldsymbol{q}}} \right\|^2 + k^2 \left\| \hat{\dot{\boldsymbol{q}}} \right\|^2 \tag{2.67}$$

wird ersichtlich, wie der Dämpfungsfaktor k die genaue Einhaltung der Bewegungsvorgabe im linken Summanden mit den Gelenkgeschwindigkeiten im rechten Summanden abwägt. Je größer der Dämpfungsfaktor k ist, desto stärker werden die Gelenkgeschwindigkeiten in der Nähe von Singularitäten gedämpft, desto größer wird jedoch auch die Abweichung von der vorgegebenen kartesischen Geschwindigkeit.

Man erhält nach Lösung des Optimierungsproblems die beiden äquivalenten Pseudoinversen

$$\hat{\boldsymbol{J}}^{\dagger} = \left(\hat{\boldsymbol{J}}^{\mathsf{T}} \hat{\boldsymbol{J}} + k^2 \boldsymbol{I} \right)^{-1} \hat{\boldsymbol{J}}^{\mathsf{T}} \tag{2.68}$$

und

$$\hat{\boldsymbol{J}}^{\dagger} = \hat{\boldsymbol{J}}^{\mathsf{T}} \left(\hat{\boldsymbol{J}} \hat{\boldsymbol{J}}^{\mathsf{T}} + k^2 \boldsymbol{I} \right)^{-1} . \tag{2.69}$$

Auch diese Pseudoinversen lassen sich wieder unter Verwendung der ursprünglichen Variablen ausdrücken und man erhält

$$\boldsymbol{J}^{\#} = \left(\boldsymbol{J}^{\mathsf{T}}\boldsymbol{W}_v\boldsymbol{J} + k^2\boldsymbol{W}_q\right)^{-1}\boldsymbol{J}^{\mathsf{T}}\boldsymbol{W}_v \tag{2.70}$$

beziehungsweise

$$\boldsymbol{J}^{\#} = \boldsymbol{W}_q^{-1}\boldsymbol{J}^{\mathsf{T}}\left(\boldsymbol{J}\boldsymbol{W}_q^{-1}\boldsymbol{J}^{\mathsf{T}} + k^2\boldsymbol{W}_v^{-1}\right)^{-1}. \tag{2.71}$$

Alternativ dazu kann die Singulärwertzerlegung auch hier herangezogen werden. Setzt man $\hat{\boldsymbol{J}} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\mathsf{T}}$ in Gleichung (2.68) oder (2.69) ein, so erhält man (in beiden Fällen) die Pseudoinverse

$$\hat{\boldsymbol{J}}^{\dagger} = \boldsymbol{V}\boldsymbol{\Sigma}^{\mathsf{T}}\left(\boldsymbol{\Sigma}\boldsymbol{\Sigma}^{\mathsf{T}} + k^2\boldsymbol{I}\right)^{-1}\boldsymbol{U}^{\mathsf{T}} = \sum_{i=1}^{r}\frac{\sigma_i}{\sigma_i^2 + k^2}\boldsymbol{v}_i\boldsymbol{u}_i^{\mathsf{T}}. \tag{2.72}$$

Nahe einer Singularität, wenn der kleinste Singulärwert σ_r gegen null strebt, strebt auch

$$\frac{\sigma_r}{\sigma_r^2 + k^2} \tag{2.73}$$

gegen null anstatt unendlich. Eine Verletzung der maximalen Gelenkgeschwindigkeiten in der Nähe von Singularitäten wird verhindert, jedoch hat der Dämpfungsfaktor auch abseits von Singularitäten einen kleinen aber mitunter unerwünschten Einfluss.

Um hier die Genauigkeit der Lösung zu erhöhen, lässt sich ein Dämpfungsfaktor verwenden, der abseits von Singularitäten «deaktiviert» wird. In einem frühen Ansatz schlagen Nakamura et al. (1986) vor, das Maß der Manipulabilität (Yoshikawa 1985) zu verwenden, um die Nähe zu einer Singularität abzuschätzen. Die Bestimmung dieses Maßes erfordert weniger Rechenaufwand als eine Singulärwertzerlegung, ergibt jedoch nur eine ungenaue Annäherung für die Nähe zu einer Singularität. Berechnet man stattdessen den kleinsten Singulärwert, so kann dieser als besserer Maßstab für die Nähe zu Singularitäten verwendet werden. Chiaverini et al. (1991) definieren damit den Dämpfungsfaktor

$$k^2 = \begin{cases} 0 & \text{für } \sigma_r \geq \epsilon \\ \left(1 - \left(\frac{\sigma_r}{\epsilon}\right)^2\right)k_{\max}^2 & \text{sonst} \end{cases}, \tag{2.74}$$

wobei ϵ die Größe der zu dämpfenden Region bestimmt. Für $\sigma_r = \epsilon$ ist der Dämpfungsfaktor null, wodurch ein stetiger Übergang gewährleistet wird. Der Dämpfungsfaktor ist durch den Parameter $k_{\max}^2$ begrenzt, den er für $\sigma_r = 0$ erreicht. Von Chiaverini et al. (1991) und Chiaverini et al. (1994) wird zudem über eine Gewichtungsmatrix ein geringeres Gewicht auf Richtungen gelegt, für die keine hohe Genauigkeit erforderlich ist. Wird die Bewegung in eine dieser Rich-

tungen in der Nähe einer Singularität eingeschränkt, so hat die entsprechende Dämpfung eine kleinere Auswirkung auf die Genauigkeit der stärker gewichteten Richtungen.

Spezifikation sekundärer Aufgaben

Bei der Herleitung der Linksinversen zur Lösung einer unterbestimmten Aufgabenspezifikation wurde die Norm $\left\|\hat{\dot{\boldsymbol{q}}}\right\|$ minimiert. Dies bedeutet, dass unter den unendlich vielen möglichen Lösungen diejenige mit den geringsten (gewichteten) Gelenkgeschwindigkeiten ausgewählt wird. Alternativ dazu kann auch versucht werden, ein sekundäres Ziel zu erreichen, das den Nullraum der Lösung

$$\hat{\dot{\boldsymbol{q}}} = \hat{\boldsymbol{J}}^{\dagger}\hat{\boldsymbol{v}} \tag{2.75}$$

ausnutzt. Wird das sekundäre Ziel als Gelenkgeschwindigkeit $\hat{\dot{\boldsymbol{q}}}_{null}$ ausgedrückt, so lässt sich die zu minimierende Lagrange-Funktion

$$\mathcal{L}(\hat{\dot{\boldsymbol{q}}}, \boldsymbol{\lambda}) = \left\|\hat{\dot{\boldsymbol{q}}} - \hat{\dot{\boldsymbol{q}}}_{null}\right\|^2 + \boldsymbol{\lambda}^{\mathsf{T}}\left(\hat{\boldsymbol{v}} - \hat{\boldsymbol{J}}\hat{\dot{\boldsymbol{q}}}\right) \tag{2.76}$$

aufstellen, die die Abweichung von $\hat{\dot{\boldsymbol{q}}}$ zu $\hat{\dot{\boldsymbol{q}}}_{null}$ minimiert. Die entsprechende Lösung

$$\begin{aligned} \hat{\dot{\boldsymbol{q}}} &= \hat{\boldsymbol{J}}^{\dagger}\hat{\boldsymbol{v}} + \left(\boldsymbol{I} - \hat{\boldsymbol{J}}^{\dagger}\hat{\boldsymbol{J}}\right)\hat{\dot{\boldsymbol{q}}}_{null} && (2.77)\\ &= \hat{\boldsymbol{J}}^{\dagger}\hat{\boldsymbol{v}} + \hat{\boldsymbol{P}}\hat{\dot{\boldsymbol{q}}}_{null} && (2.78) \end{aligned}$$

enthält die Projektionsmatrix

$$\hat{\boldsymbol{P}} = \left(\boldsymbol{I} - \hat{\boldsymbol{J}}^{\dagger}\hat{\boldsymbol{J}}\right), \tag{2.79}$$

mit der die sekundäre Gelenkgeschwindigkeit $\hat{\dot{\boldsymbol{q}}}_{null}$ in den Nullraum der primären Lösung projiziert wird. Die sekundäre Bewegung wird dabei gegebenenfalls nicht exakt ausgeführt, sondern nur so weit es ohne eine Beeinträchtigung der primären Bewegung möglich ist.

Das Einsetzen der Gleichungen (2.41) bis (2.43) und (2.54) ergibt den Zusammenhang in den ursprünglichen Variablen

$$\begin{aligned} \dot{\boldsymbol{q}} &= \boldsymbol{J}^{\#}\boldsymbol{v} + \left(\boldsymbol{I} - \boldsymbol{J}^{\#}\boldsymbol{J}\right)\dot{\boldsymbol{q}}_{null} && (2.80)\\ &= \boldsymbol{J}^{\#}\boldsymbol{v} + \boldsymbol{P}\dot{\boldsymbol{q}}_{null} && (2.81) \end{aligned}$$

mit

$$\boldsymbol{P} = \left(\boldsymbol{I} - \boldsymbol{J}^{\#}\boldsymbol{J}\right). \tag{2.82}$$

Spezifikation sekundärer kartesischer Bewegungen

In vielen Fällen ist es einfacher, sekundäre Aufgaben als kartesische Geschwindigkeit anzugeben. Nakamura et al. (1987) und Maciejewski et al. (1985) formulieren die sogenannte *Task Priority Strategy*, die es erlaubt, neben der primären Aufgabe

$$\hat{\boldsymbol{v}} = \hat{\boldsymbol{J}}\hat{\dot{\boldsymbol{q}}} \tag{2.83}$$

eine sekundäre kartesische Bewegung

$$\hat{\boldsymbol{v}}_2 = \hat{\boldsymbol{J}}_2\hat{\dot{\boldsymbol{q}}} \tag{2.84}$$

vorzugeben. Das Einsetzen von (2.78) in Gleichung (2.84) und das Auflösen nach

$$\hat{\dot{\boldsymbol{q}}}_{null} = \left(\hat{\boldsymbol{J}}_2\hat{\boldsymbol{P}}\right)^{\dagger}\left(\hat{\boldsymbol{v}}_2 - \hat{\boldsymbol{J}}_2\hat{\boldsymbol{J}}^{\dagger}\hat{\boldsymbol{v}}\right), \tag{2.85}$$

wiederum eingesetzt in (2.78) ergibt schließlich die Lösung

$$\hat{\dot{\boldsymbol{q}}} = \hat{\boldsymbol{J}}^{\dagger}\hat{\boldsymbol{v}} + \hat{\boldsymbol{P}}\left(\hat{\boldsymbol{J}}_2\hat{\boldsymbol{P}}\right)^{\dagger}\left(\hat{\boldsymbol{v}}_2 - \hat{\boldsymbol{J}}_2\hat{\boldsymbol{J}}^{\dagger}\hat{\boldsymbol{v}}\right). \tag{2.86}$$

Maciejewski et al. (1985) zeigen, dass

$$\hat{\boldsymbol{P}}\left(\hat{\boldsymbol{J}}_2\hat{\boldsymbol{P}}\right)^{\dagger} = \left(\hat{\boldsymbol{J}}_2\hat{\boldsymbol{P}}\right)^{\dagger}, \tag{2.87}$$

da $\hat{\boldsymbol{P}}$ symmetrisch und idempotent ist. Vanthienen (2015) weist darauf hin, dass dies jedoch bei der Verwendung von Gewichtungen nicht der Fall sei.

Bei der Transformation zurück in die ursprünglichen Variablen wird eine eigene Gewichtung für die sekundäre kartesische Bewegung

$$\hat{\boldsymbol{v}}_2 = \boldsymbol{N}_{v2}\boldsymbol{v}_2 \tag{2.88}$$

verwendet. Die Gewichtung der Gelenkgeschwindigkeiten sei jeweils identisch. Dadurch ergibt sich als gewichtete Task Priority Strategy

$$\dot{\boldsymbol{q}} = \boldsymbol{J}^{\#}\boldsymbol{v} + \boldsymbol{P}(\boldsymbol{J}_2\boldsymbol{P})^{\#_2}(\boldsymbol{v}_2 - \boldsymbol{J}_2\boldsymbol{J}^{\#}\boldsymbol{v}) \tag{2.89}$$

wobei für die Pseudoinverse $(\boldsymbol{J}_2\boldsymbol{P})^{\#_2}$ die Gewichtung $\boldsymbol{N}_{v2}$ der sekundären Bewegung Verwendung findet.

Die vorgestellte Task Priority Strategy beschränkt sich auf zwei Prioritätsstufen. Siciliano et al. (1991) stellen darauf aufbauend eine rekursive Form des Algorithmus vor, bei der beliebig viele

Prioritätsstufen vorgegeben werden können. Startend mit der höchsten Prioritätsstufe 1, wird der Algorithmus

$$\dot{\boldsymbol{q}}_1 = \boldsymbol{J}_1^{\#_1} \boldsymbol{v}_1 \,, \tag{2.90}$$

$$\dot{\boldsymbol{q}}_i = \dot{\boldsymbol{q}}_{i-1} + \boldsymbol{P}_{i-1} (\boldsymbol{J}_i \boldsymbol{P}_{i-1})^{\#_i} (\boldsymbol{v}_i - \boldsymbol{J}_i \dot{\boldsymbol{q}}_{i-1}) \tag{2.91}$$

bis zur niedrigsten Prioritätsstufe n_p ausgeführt und ergibt schließlich die gewünschte Gelenkgeschwindigkeit $\dot{\boldsymbol{q}} = \dot{\boldsymbol{q}}_{n_p}$. Bei jeder Iteration wird dabei die Projektionsmatrix

$$\boldsymbol{P}_i = \boldsymbol{I} - \boldsymbol{J}_{1,i}^{\#_{1,i}} \boldsymbol{J}_{1,i} \tag{2.92}$$

mit den bis dahin akkumulierten Jacobi-Matrizen

$$\boldsymbol{J}_{1,i} = \begin{bmatrix} \boldsymbol{J}_1^\mathsf{T} & \boldsymbol{J}_2^\mathsf{T} & \dots & \boldsymbol{J}_i^\mathsf{T} \end{bmatrix}^\mathsf{T} \tag{2.93}$$

aller höher priorisierten Aufgabenspezifikationen verwendet.

Die hier vorgestellten kinematischen Verfahren werden in Abschnitt 2.4 zur Lösung des iTaSC-Algorithmus herangezogen. Zunächst wird jedoch in den nachfolgenden Abschnitten auf die Kraftregelung und Programmierung sensorbasierter Robotersysteme eingegangen.

2.2 Kraftregelung

Für viele Anwendungsfelder der Robotik ist es essenziell, eine definierte Prozesskraft durch den Roboter aufzubringen. Hierzu zählen beispielsweise Bearbeitungsprozesse wie Polieren, Entgraten oder Zerspanen. Auch in der Montage sind Kontaktkräfte zu berücksichtigen. Aufgrund von Lage- und Bauteiltoleranzen kann es bei klassischer Positionsregelung vorkommen, dass die programmierte Zielpose beim Fügevorgang nicht erreichbar ist und es beim Kontakt mit dem Bauteil zu einem Aufbau sehr großer, ungewollter Kräfte kommt. Je nach Roboter und Anwendung kommt es zu einem Überlastfehler und einhergehender Stillstandszeit oder gar zu Schäden an Werkstücken, Werkzeugen, Vorrichtungen oder am Roboter. In diesen Fällen ist eine Nachgiebigkeit als Toleranzausgleich notwendig.

Hierbei wird zwischen passiver und aktiver Nachgiebigkeit unterschieden (Villani et al. 2016). Zur passiven Nachgiebigkeit zählt zunächst die Nachgiebigkeit der mechanischen Struktur des Roboters. Darüber hinaus kann mithilfe eines nachgiebigen Ausgleichselements (RCC, von engl. *remote center of compliance*) eine größere Nachgiebigkeit erreicht werden. Dazu werden

Federn oder Elastomere in das Werkzeug integriert, wodurch dieses in eine oder mehrere Richtungen nachgiebig wird. Dies scheint zunächst eine einfache und kostengünstige Lösung zu sein. Die Hardwarelösung ist jedoch wenig flexibel. Sie muss spezifisch für jede Anwendung ausgelegt werden, kann nicht innerhalb einer Anwendung umkonfiguriert werden und kann insbesondere auch nicht deaktiviert werden, falls eine hohe Positioniergenauigkeit für einen anderen Prozessschritt der Anwendung nötig ist.

Eine aktive Regelung der Nachgiebigkeit wird hingegen erreicht, indem die auftretenden Kräfte gemessen oder geschätzt und in die Regelung des Roboters mit einbezogen werden. Die Kraftsensorik wird dabei meist extern als Kraftmessdose am Roboterflansch, seltener auch in der Montagevorrichtung angebracht. Daneben enthalten einzelne Roboter auch interne Gelenkmomentensensoren und berechnen daraus modellbasiert die am Werkzeug auftretenden Kräfte. Stolt et al. (2012b) zeigen zudem eine Methode zur Schätzung der Prozesskräfte ohne Kraftsensor, die auf Basis der durch den Kontakt verursachten Regelabweichungen in den Drehgelenken erfolgt.

Es kann ferner zwischen direkter und indirekter Kraftregelung unterschieden werden, abhängig davon, ob es das Regelungskonzept erlaubt, die Kontaktkraft auf ein erwünschtes Maß zu regeln oder sie sich indirekt ergibt. Bei indirekter Kraftregelung wird nicht die Kontaktkraft selbst vorgegeben, sondern das Verhältnis von Bewegung zu Kontaktkraft. Hierzu zählen die Impedanzregelung, bei der eine Bewegungsabweichung zu einer Kraft führt, und die Admittanzregelung, bei der eine gemessene Kraft zu einer Bewegung führt (Hogan 1985; Hogan 1987).

Zur direkten Kraftregelung zählen die hybride und die parallele Kraft-/Positionsregelung. Bei der hybriden Kraft-/Positionsregelung (Raibert et al. 1981) werden die sechs Freiheitsgrade entlang den Richtungen eines kartesischen Koordinatensystems aufgeteilt und jeweils entweder kraft- oder positionsgeregelt. Im Unterschied dazu wird bei der parallelen Kraft-/Positionsregelung (Chiaverini et al. 1993; De Schutter et al. 1988) Kraft- und Positionsregelung entlang derselben Richtung kombiniert. Das Regelungskonzept ist dabei so gestaltet, dass die Kraftregelung die Positionsregelung dominiert und demnach eine korrekte Kraftregelung unter der Inkaufnahme eines Positionsfehlers erreicht wird. Mithilfe dieses Ansatzes lassen sich Modellfehler besser ausgleichen, da Kräfte in bewegungsgeregelte Richtungen und Bewegungen in kraftgeregelte Richtungen explizit berücksichtigt werden.

2.3 Programmierung sensorbasierter Robotersysteme

Bei der Integration von sensorbasierten Funktionalitäten, wie Kraftregelung oder Bildverarbeitung, stößt die klassische Roboterprogrammierung an ihre Grenzen. Dieser Abschnitt gibt einen

Codebeispiel 2.1: KRL Beispielprogramm nach Pott et al. (2019)

```
1  DEF Beispielprogramm()
2
3  INI
4
5  PTP HOME Vel= 100% DEFAULT
...
10 PTP PMaschine Vel= 100% PDAT4 Tool[0] Base[0]
11 PTP PAnfahrt Vel= 80% PDAT6 Tool[0] Base[1]
12 LIN PGreif Vel= 0.1M/s CPDAT5 Tool[0] Base[1]
13
14 $OUT[16]=TRUE
...
26 PTP HOME Vel= 100% DEFAULT
27
28 END
```

Überblick, wie Kraftregelung in die Steuerungen von Industrierobotern integriert werden kann, welche Architekturparadigmen dazu in der Forschung untersucht werden und wie Middlewares oder eine modellgetriebene Softwareentwicklung bei der Programmierung komplexer Robotersysteme unterstützen können.

Programmierung von Industrierobotern

Hersteller von Industrierobotern bieten zur Programmierung ihrer Roboter eigene, proprietäre Sprachen an. Die Programmiersprachen sind meist einfach gehalten und basieren auf imperativen Sprachen wie Pascal aus den 1970er Jahren (Mühe et al. 2010). Ein typisches Roboterprogramm besteht aus einer Liste an Bewegungsbefehlen und Operationen wie das Lesen von Eingängen und Setzen von Ausgängen zur Kommunikation mit Peripheriekomponenten. Das Codebeispiel 2.1 zeigt die Programmiersprache KRL des Roboterherstellers KUKA mit Punkt-zu-Punkt-Bewegungen (PTP) sowie einer linearen Bewegung (LIN). Über Prozeduren und einfache Kontrollstrukturen wie Verzweigungen und Schleifen lassen sich Programme strukturieren. Eine Wiederverwendung einzelner Quelltextabschnitte erfolgt durch die Definition und den mehrfachen Aufruf von Prozeduren oder schlichtweg durch das Kopieren und Anpassen der Quelltextabschnitte.

Die einzelnen vom Benutzer programmierten Befehle werden durch den Interpolator der Bewegungssteuerung in kleine Teilschritte zerlegt. Diese werden anschließend als Sollwerte an die Lageregelung weitergereicht (Pott et al. 2019). Für eine Sensorintegration ist jedoch ein direkter Zugriff auf die Lageregelung nötig, um die Roboterbahn basierend auf Sensormesswerten zyklisch zu korrigieren. Technologiepakete wie RSI (KUKA Roboter GmbH 2010) erlauben es dazu, Sensordaten im Interpolationstakt (bei RSI 4 oder 12 ms) einzulesen und zu verarbeiten. Darauf aufbauend wird die Roboterbahn korrigiert oder komplett erzeugt. Vorgefertigte

Funktionsblöcke wie Filter, Koordinatentransformationen oder Regler erleichtern die Programmierung, generell ist jedoch ein weitreichendes Expertenwissen nötig.

RSI und ähnliche Schnittstellen anderer Hersteller wie b-CAP (Denso Wave Inc. 2017) oder RTDE (Universal Robots A/S 2019) erlauben es zudem, Industrieroboter von einer externen Software aus anzusteuern. Bewegungsbefehle können dabei in Form von Achs- oder kartesischen Geschwindigkeiten angegeben werden. Die Modellierung dynamischer Eigenschaften, wie die Berücksichtigung des Eigengewichts des Roboters oder das Gewicht des Werkzeugs, erfolgt weiterhin durch die Robotersteuerung. Dies erlaubt eine sehr einfache Integration der Roboter in neuartige Softwarearchitekturen.

Architekturparadigmen in der Robotikforschung

Softwarearchitekturen mit Fokus auf der Integration von Sensordaten wurden insbesondere zur Steuerung von autonomen, mobilen Robotern erforscht. Da diese in unstrukturierten Umgebungen zurechtkommen müssen, bringen sie erhöhte Anforderungen an die Sensorintegration mit sich. Sukzessive hat sich hierbei das jeweils vorherrschende Architekturparadigma gewandelt (Kortenkamp et al. 2016; Gat 1998). In den frühen Anfängen der Erforschung autonomer Roboter dominierte das *Sense-Plan-Act* (SPA) Paradigma. Charakteristisch war die Aufteilung der Robotersoftware in die drei Bereiche der Sensordatenerfassung, Planung und Ausführung, die aufeinander aufbauen aber strikt nacheinander bearbeitet werden. Bei der Sensordatenerfassung werden die beispielsweise von einer Kamera aufgenommenen Daten ausgewertet und das Umgebungsmodell aktualisiert. Daraufhin erstellt der Planer auf Basis eines vorgegebenen Ziels und des aktualisierten Umgebungsmodells einen neuen Plan, der anschließend ausgeführt wird. Sensordaten werden hierbei nicht während der Ausführung berücksichtigt, sondern erst bei der Erstellung des nächsten Plans. Dies hat den entscheidenden Nachteil, dass nur sehr langsam auf eine sich ändernde Umgebung reagiert werden kann.

In den 1980er Jahren setzten sich reaktivere Architekturen durch, allen voran die *Subsumption* Architektur von Brooks (1986). Verhaltensbausteine (engl. *behaviors*) verbinden hierbei Sensoren und Aktoren direkt miteinander, wodurch deutlich schneller auf Änderungen in der Umgebung reagiert werden kann. Die Koordination der Verhaltensbausteine erfolgt über die Deaktivierung nicht benötigter Verhalten. Es stellte sich dabei jedoch als schwierig heraus, komplexere Aufgaben zu erfüllen.

Mit Drei-Schichten-Architekturen gelang es schließlich, die Vorteile beider Ansätze zu verbinden. Auf der untersten Architekturebene wird eine reaktionsfähige Regelung in Echtzeit ausgeführt, auf der obersten Ebene ein Planer mit einer deutlich geringeren Frequenz. Die Ebene

dazwischen verbindet die beiden anderen Ebenen, indem sie die Ausführung erstellter Pläne koordiniert. Diese Form der Architektur erlaubt eine größere Abstraktion und Modularisierung von Verhaltensbausteinen und erleichtert deren Koordination und Wiederverwendung.

Das in dieser Arbeit vorgestellte System besitzt eine Drei-Schichten-Architektur, beinhaltet jedoch keinen Planer. An seiner Stelle befinden sich das Skill-Modell und die dazugehörige DSL, mit der Benutzer Anwendungen programmieren können.

Middlewares

Die zwei Implementierungen zur Ausführung des in dieser Arbeit vorgeschlagenen Skill-Modells verwenden ROS beziehungsweise OROCOS als Middlewares, die im Folgenden kurz vorgestellt werden. Middlewares sind zwischen dem Betriebssystem und den Anwendungen angesiedelt. Sie sind oft auf die Anforderungen einer bestimmten Domäne zugeschnitten und erleichtern insbesondere die Kommunikation zwischen Programmteilen sowie deren Integration.

Das *Robot Operating System* (ROS) ist ein quelloffenes Software-Framework für Robotersysteme (Quigley et al. 2009), das sich weltweit in der Robotikforschung etabliert hat und auch zunehmend in kommerziellen Produkten und Anlagen Verwendung findet. ROS funktioniert nach dem Peer-to-Peer-Prinzip, bei dem einzelne Prozesse als Knoten (engl. *node*) in einem Netz miteinander kommunizieren. ROS bietet eine Kommunikationsschicht, mit der die Knoten über Nachrichten untereinander kommunizieren können. Je nach Anforderungen kann dabei auf verschiedene Kommunikationsarten zurückgegriffen werden. Die Kommunikation über *Topics* (dt. Themen) erfolgt nach dem *Publish-Subscribe*-Modell, bei dem ein oder mehrere Knoten unter einem Topic Nachrichten veröffentlichen, die von beliebig vielen anderen Knoten empfangen werden können. Sender und Empfänger sind dabei entkoppelt. *Services* (dt. Dienste) arbeiten nach dem Prinzip eines *Remote Procedure Calls* (RPC): Nach dem Aufrufen des Services wird der aufrufende Prozess blockiert, bis die entsprechende Antwort eintrifft. Für nicht-blockierende Anfragen können *Actions* (dt. Aktionen) verwendet werden.

ROS Pakete (engl. *packages*) kapseln Daten und Dienste, modellieren explizit ihre Abhängigkeiten zu anderen Paketen und erleichtern so den Austausch von Softwarekomponenten zwischen Entwicklern sowie ihre Integration. Eine Vielzahl an Paketen mit Softwarewerkzeugen zur Visualisierung, Entwicklung und Diagnostik werden von der ROS-Community angeboten. So verwaltet *ROS tf* einen Baum aus Koordinatensystemen. Über tf können Transformationen zwischen beliebigen Koordinatensystemen abgefragt werden. *ROS Control* dient der Entkopplung von Robotertreibern und Reglern. Es definiert einheitliche Schnittstellen zur Regelung von Robotern und stellt einen Manager zum Aktivieren und Deaktivieren bestimmter Regler zur Verfü-

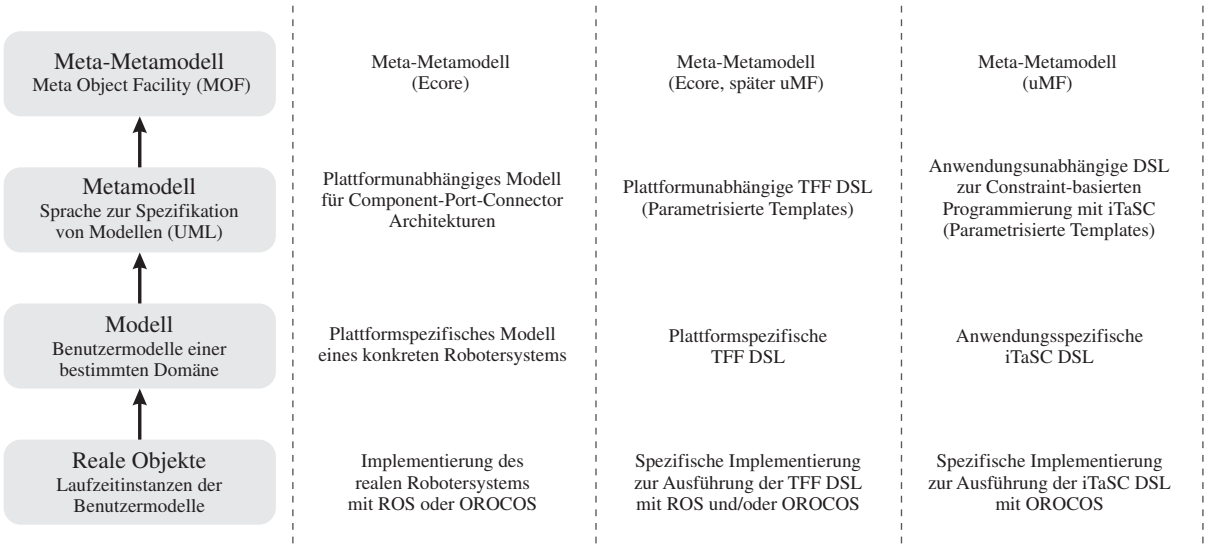
gung. Während ROS Control eine echtzeitfähige Kommunikation zwischen Robotertreibern und Reglern ermöglicht, ist die Kommunikation zwischen Knoten in ROS nicht echtzeitfähig. Das sich in Entwicklung befindende ROS 2.0 soll hingegen Echtzeit in der Kommunikationsschicht erlauben.

Die *Open Robot Control Software* (OROCOS) ist wie ROS komponentenbasiert und funktioniert ebenso nach dem Peer-to-Peer-Prinzip, setzt jedoch einen starken Fokus auf Echtzeitfähigkeit (Bruyninckx 2001). OROCOS gliedert sich in mehrere Programmbibliotheken, unter anderem das Real-Time Toolkit (RTT) zur Programmierung echtzeitfähiger Regelungssysteme, die OROCOS Component Library (OCL) mit essenziellen RTT-Komponenten sowie die Kinematics and Dynamics Library (KDL) zur Modellierung und Berechnung kinematischer Ketten. Mit *Data Flow Ports* lassen sich in OROCOS Daten in Echtzeit und thread-sicher nach dem *Publish-Subscribe*-Modell senden und empfangen. *Operations* ermöglichen blockierende und nicht-blockierende RPCs. Eine Schnittstelle zwischen ROS und OROCOS erlaubt den leichten Austausch von Daten zwischen den beiden Middlewares.

Modellgetriebene Softwareentwicklung

Das in dieser Arbeit beschriebene System verfolgt einen modellbasierten Ansatz. Modellbasierte Ansätze erleichtern den Umgang mit komplexen Systemen und erhöhen somit die Produktivität von Softwareentwicklern. Insbesondere zielen die Ansätze der modellgetriebenen Softwareentwicklung (MDE, von engl. *model-driven engineering*) darauf ab, Modelle nicht nur zu Dokumentationszwecken, sondern auch zur Generierung oder automatischen Ausführung von Softwaresystemen einzusetzen (da Silva 2015).

Der Ausgangspunkt dieser Ansätze ist die abstrakte Beschreibung von Softwaresystemen durch Modelle. Die Abstraktion eines konkreten Systems hilft unter anderem bei der Analyse des Systems oder bei der Kommunikation über seine Eigenschaften. Umgekehrt können Modelle auch herangezogen werden, um daraus konkrete Systeme zu erzeugen, beispielsweise über Codegenerierung oder Parametrierung eines Ausführungsalgorithmus. Darüber hinaus können weitere Artefakte aus Modellen generiert werden, wie Dokumentationen, Spezifikationen oder Visualisierungen. Neben der erhöhten Produktivität von Softwareentwicklern ist auch der strategische Aspekt nicht zu unterschätzen, Softwaresysteme durch die Verwendung von Modellen robuster gegenüber Veränderungen zu machen (Atkinson et al. 2003). Dies wird zum einen durch das explizite Erfassen von Wissen der Entwickler in den Modellen ermöglicht, zum anderen durch die Abstraktion von mit der Zeit wechselnden Entwicklungs- und Zielplattformen. Es steigt dadurch die Lebensdauer der Softwaresysteme und damit einhergehend ihre Rendite.



(a) OMG (b) BRICS (c) Klotzbücher et al. (d) Vanthienen et al.

Abbildung 2.5: MDA 4-Ebenenmodell der Abstraktion

Ein Modell repräsentiert einen Aspekt eines Systems für einen gewissen Nutzen. Es ist eine Vereinfachung des originalen Systems, die durch gezieltes Weglassen von für den Nutzen vernachlässigbaren Details erzeugt wird. Es kann dabei viele unterschiedliche Modelle eines Systems geben, die zu verschiedenen Zwecken und in verschiedenen Abstraktionsgraden erstellt werden. Der Grad der Abstraktion beschreibt dabei, wie viele Details entfernt werden. Ein höherer Abstraktionsgrad erlaubt es einem Modell, eine größere Menge an konkreten Systemen zu repräsentieren (beispielsweise eine Aufgabenspezifikation, die vom eingesetzten Roboter unabhängig ist) – dafür müssen jedoch wieder entsprechend mehr Details hinzufügt werden, um ein Modell in eine konkrete Implementierung umzusetzen (im genannten Beispiel spezifische Informationen über den eingesetzten Roboter).

Die Object Management Group (OMG) definiert mit der Model Driven Architecture (MDA) einen Rahmen für Standards wie die Unified Modeling Language (UML) für die modellgetriebene Softwareentwicklung (Bézivin et al. 2001; OMG 2014). Mit ihrem 4-Ebenenmodell beschreibt die OMG vier verschiedene Abstraktionsschichten, wie in Abbildung 2.5a dargestellt. Die M0-Ebene enthält die Laufzeitinstanzen einer ausgeführten Anwendung und beschreibt somit das reale System. Auf M1-Ebene befinden sich Modelle des realen Systems, beispielsweise dargestellt durch UML-Diagramme. Die Modelle dieser Ebene sind spezifisch für das reale System. Sie werden in einer Modellierungssprache ausgedrückt, die ihre Syntax und Semantik beschreibt. Diese Modellierungssprache ist von realen Systemen unabhängig und auf Ebene M2 definiert. Die UML ist beispielsweise die Modellierungssprache, nach der konkrete UML-Diagramme erzeugt werden können. Da Modellierungssprachen oft ebenfalls durch Modelle beschrieben werden, spricht man auch von Metamodellen. Die OMG definiert mit der Meta Object Facility (MOF)

zudem eine weitere Ebene M3 als Meta-Metamodell, nach der wiederum die einzelnen Metamodelle einheitlich beschrieben werden können.

Von Interesse ist es, diese Hierarchie über Modelltransformationen sowohl innerhalb einer Ebene als auch von einer Ebene zu einer anderen zu durchlaufen. Auf gleicher Ebene kann mit einer Modell-zu-Modell-Transformation (M2M) ein Modell eines Metamodells in ein Modell eines anderen Metamodells überführt werden. So kann beispielsweise ein Modell aus der Form einer menschenlesbaren DSL in eine für Berechnungen effizientere Modellierungsform überführt werden. Ebenso ist es möglich, eine Transformation in eine höhere Abstraktionsstufe oder mit einer Modell-zu-Text-Transformation (M2T) in eine niedrigere Abstraktionsstufe vorzunehmen. Letzteres ist bei Codegenerierung der Fall. Die für eine niedrigere Abstraktionsstufe benötigten Details lassen sich manuell ergänzen oder durch die Kombination von Modellen und durch Anwendung von Mustern oder Regeln automatisch erzeugen.

Das BRICS Component Model (BCM) baut auf dem Metamodellierungsansatz der OMG auf und wendet diesen auf die Robotik an (Bruyninckx et al. 2013). Das BCM stellt Richtlinien, Metamodelle und Werkzeuge bereit, um die Entwicklung von Komponenten und komponentenbasierten Architekturen für die Robotik zu strukturieren. Die Interpretation des 4-Ebenenmodells aus Sicht des BCMs ist in Abbildung 2.5b dargestellt. Als Meta-Metamodell der Ebene M3 wird Ecore des Eclipse Modeling Framework (EMF) verwendet. Die Ebene M2 wird durch ein Component-Port-Connector (CPC) Modell beschrieben. Hierbei werden die einzelnen Bestandteile des Robotersystems in Komponenten zerteilt, die über Ports miteinander interagieren. Das abstrakte CPC-Modell wird durch Modelle für konkrete Frameworks wie ROS und OROCOS ergänzt. Auf Ebene M1 werden damit Modelle für spezifische Robotersysteme erstellt. Die BRIDE-Toolchain unterstützt Entwickler dabei, diese Modelle zu entwerfen (Bubeck et al. 2014). Die Ebene M0 beschreibt schließlich die konkrete Implementierung eines Robotersystems und seiner Komponenten.

Klotzbücher et al. (2011) und Vanthienen et al. (2013) beziehen sich ebenfalls auf das 4-Ebenenmodell der OMG und wenden dieses auf die Spezifikation von Roboteranwendungen an, wie in den Abbildungen 2.5c und 2.5d dargestellt. Ebene M2 definiert dabei jeweils eine DSL zur Spezifikation der Anwendungen mithilfe des TFF beziehungsweise iTaSC, basierend auf parametrisierten Templates. Das Modell einer spezifischen Anwendung wird auf Ebene M1 erstellt und durch Konfiguration und Instanziierung zur Ausführung gebracht.

Wie durch eine Verschmelzung der Ebenen M2 und M1 im vorliegenden Skill-Modell der Einsatz und die Wiederverwendung von Skills erleichtert werden können, wird in Abschnitt 4.1.1 ausgeführt. Einen Überblick über weitere modellgetriebene Ansätze in der Robotik geben Ramaswamy et al. (2014) sowie Brugali (2015).

Domänenspezifische Sprachen

Eine domänenspezifische Sprache (DSL) ist eine Programmiersprache, die auf eine gewisse Anwendungsdomäne spezialisiert ist. DSLs sind üblicherweise kompakte Sprachen mit limitiertem Ausdrucksvermögen, teilweise ergänzen sie aber auch existierende allgemeine Programmiersprachen (Van Deursen et al. 2000). Sie stellen den Menschen und die Bedienbarkeit in den Vordergrund (Fowler et al. 2010). Dies unterscheidet eine DSL einerseits von bloßen Dateiformaten, andererseits auch von allgemeinen Programmiersprachen, die mächtiger aber dadurch bedingt auch meist schwerer les-, schreib- und erlernbar sind. Eine klare Abgrenzung ist dabei jedoch nicht immer möglich.

DSLs lassen sich in drei Kategorien einteilen (Fowler et al. 2010). Eine externe DSL ist eine eigens definierte Sprache, die sich von der eigentlichen Programmiersprache der Anwendung unterscheidet. Ein typischer Vertreter externer DSLs ist SQL, eine Sprache zur Arbeit mit Datenbanken. Interne DSLs hingegen ergänzen eine bestehende allgemeine Programmiersprache. Charakteristisch ist hierbei die Einführung von domänenspezifischen Befehlen und Datenstrukturen sowie die Restriktion auf die für die Nutzung relevanten Spracheigenschaften der allgemeinen Programmiersprache, mit dem Ziel eine leicht lesbare Sprache zu erzeugen. Viele interne DSLs basieren auf Sprachen wie Lisp, Lua oder Ruby. Als dritte Alternative bieten Language Workbenches eine Entwicklungsumgebung zur Erstellung von DSLs. Darüber hinaus ermöglichen sie auch, eine eigene Entwicklungsumgebung zu entwerfen, in der sich Anwendungen mithilfe der entwickelten DSLs programmieren lassen. Ihr Ziel ist also gleichermaßen die Unterstützung von Sprachentwicklern und Domänenexperten.

Der Einsatz einer DSL bietet Chancen wie Risiken (Van Deursen et al. 2000). Zu den Chancen zählt, dass Domänenexperten ohne weitreichende Programmierkenntnisse durch DSLs in die Lage versetzt werden können, komplexe Softwareprogramme zu schreiben. Das Domänenwissen der Experten ist in der DSL verankert und somit leichter zugänglich und wiederverwendbar. DSL-Code ist kompakt und in weitem Maße durch seine erhöhte Lesbarkeit selbstdokumentierend. Insgesamt können DSLs die Produktivität von Domänenexperten und Softwareentwicklern erhöhen und die Kommunikation über ein System zwischen den beiden Personengruppen vereinfachen.

Es bestehen demgegenüber jedoch auch Nachteile, wie die initialen Kosten für die Entwicklung einer DSL und der Bedarf an Experten, die dazu in der Lage sind. Zur Laufzeit kann es zudem zu Effizienzeinbußen gegenüber einer allgemeinen Programmiersprache kommen.

Ob die Entwicklung einer DSL in Betracht gezogen wird und wie sie zu gestalten ist, sollte mit Bedacht entschieden werden (Mernik et al. 2005). Auch sollte man sich bewusst sein, dass viele

Vorteile einer DSL bereits durch das Modell entstehen, auf dem die DSL am Ende ruht und somit auch ohne die eigentliche DSL bestehen könnten (Fowler et al. 2010).

Auch in der Robotik finden DSLs zunehmend Anwendung. In Abschnitt 1.2 wurden hierzu bereits einige Beispiele behandelt. Eine ausführliche Übersicht, Analyse und Klassifizierung geben Nordmann et al. (2014) und Nordmann et al. (2016). Beachtlich ist, dass hier von 137 identifizierten DSLs nur zwei primär dem Bereich Kraftregelung zugerechnet werden.

Abschnitt 4.2 beschreibt eine DSL, mit der das vorgestellte Skill-Modell effizient und ohne Programmierkenntnisse genutzt werden kann. Die vorgestellte DSL ist eine externe DSL und verwendet die Syntax von XML (engl. auch als *carrier-syntax* bezeichnet). Mit ihr lassen sich Skills und Anwendungen gemäß dem gewünschten Roboterverhalten zusammenstellen. Darauf aufbauend wird der auf dem iTaSC-Formalismus basierende Ausführungsalgorithmus instanziiert und parametriert.

2.4 Spezifikation sensorbasierter Roboterbewegungen mit iTaSC

Der iTaSC-Formalismus baut auf dem Grundgedanken auf, Roboteraufgaben nicht aus der Perspektive des Roboters zu programmieren, sondern aus der Sicht der zu handhabenden Objekte (De Schutter et al. 2007). Eine Roboteraufgabe wird spezifiziert, indem Relativbewegungen zwischen jeweils zwei Objekten vorgegeben werden. Objekte können dabei der Roboter selbst, ein angebrachtes Werkzeug, Werkstücke oder andere Objekte in der Umgebung des Roboters sein. Die geometrische Beziehung zweier Objekte zueinander wird durch sogenannte Feature-Koordinaten modelliert, die zwischen relevanten Merkmalen (engl. *features*) der Objekte definiert werden. Dies können Konturen, Symmetrieachsen oder andere Merkmale sein. Durch Vorgabe von Sollwerten für Feature-Koordinaten und von passenden Reglern können dann die gewünschten Relativbewegungen mithilfe des iTaSC-Algorithmus erzeugt werden. Auch für die einfache Integration von Sensorsignalen, wie gemessene Kräfte und Momente, lassen sich Feature-Koordinaten und Regler definieren.

Der iTaSC-Algorithmus wird hier auf Ebene der Geschwindigkeiten betrachtet, eine Formulierung auf Ebene der Beschleunigungen ist auch möglich (Smits 2010). Zur Ansteuerung der Roboter kommen die in Abschnitt 2.3 beschriebenen Schnittstellen wie RSI, RTDE und b-CAP zum Einsatz. Die Dynamik des Roboters wird entsprechend von den unterlagerten Regelkreisen der originalen Robotersteuerung berücksichtigt.

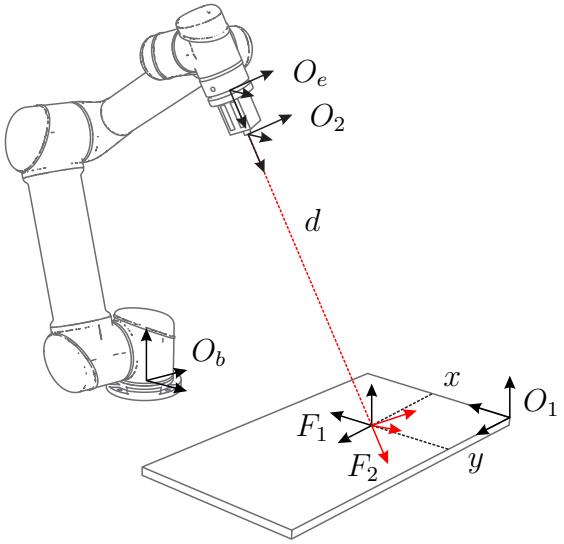


Abbildung 2.6: Modellierung einer Anwendung mit iTaSC

Um die Definition von Feature-Koordinaten zu formalisieren, führt iTaSC vier Hilfskoordinatensysteme ein. Für jedes der beiden involvierten Objekte wird zunächst ein Objekt-Koordinatensystem definiert. Zusätzlich werden zwei Feature-Koordinatensysteme in die für die Relativbewegung relevanten Merkmale gelegt. Der iTaSC-Formalismus schlägt demnach eine Aufteilung der sechs Freiheitsgrade der Relativbewegung in drei Teilbewegungen vor, die zwischen den Objekt- und Feature-Koordinatensystemen liegen. Die Koordinatensysteme werden dabei so gewählt, dass die zu lösende Roboteranwendung möglichst einfach beschrieben werden kann, indem beispielsweise jede Teilbewegung entlang einer der Achsen eines Hilfskoordinatensystems ausgeführt wird.

Abbildung 2.6 zeigt eine fiktive Anwendung in Anlehnung an De Schutter et al. (2007), bei der ein Roboter mit angebrachtem Laserabstandssensor auf einen vorgegebenen Punkt auf einer Tischplatte zeigen soll. Von Interesse sei dabei die Position des Punkts auf der Tischplatte und der Abstand zwischen diesem Punkt und dem Laser. Die für die Relativbewegung relevanten Objekte sind demnach die Tischplatte und der Laser. Das Objekt-Koordinatensystem O_1 wird in eine Ecke des Tisches gelegt, das Objekt-Koordinatensystem O_2 entspricht dem Sensor-Koordinatensystem des Lasers. Die Feature-Koordinatensysteme F_1 und F_2 liegen im Schnittpunkt des Laserstrahls und der Tischplatte. Das Feature-Koordinatensystem F_1 besitzt dabei die Orientierung des Tischkoordinatensystems O_1, das Feature-Koordinatensystem F_2 die des Sensor-Koordinatensystems O_2.

Diese Aufteilung führt zu einer sehr einfachen Modellierung der Feature-Koordinaten

$$\boldsymbol{\chi}_f = \begin{bmatrix} \boldsymbol{\chi}_{fI}^\mathsf{T} & \boldsymbol{\chi}_{fII}^\mathsf{T} & \boldsymbol{\chi}_{fIII}^\mathsf{T} \end{bmatrix}^\mathsf{T} , \tag{2.94}$$

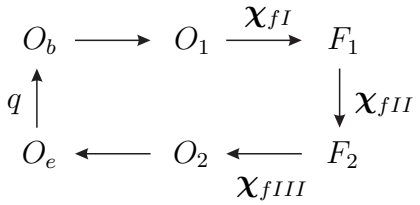


Abbildung 2.7: Kinematische Schleife mit Objekt- und Feature-Koordinatensystemen

mit

$$\boldsymbol{\chi}_{fI} = \begin{bmatrix} x & y \end{bmatrix}^\mathsf{T}, \tag{2.95}$$

$$\boldsymbol{\chi}_{fII} = \begin{bmatrix} \alpha & \beta & \gamma \end{bmatrix}^\mathsf{T}, \tag{2.96}$$

$$\boldsymbol{\chi}_{fIII} = \begin{bmatrix} d \end{bmatrix}. \tag{2.97}$$

Die Position des Punkts in der Ebene des Tisches wird relativ zum Tischkoordinatensystem O_1 über die Koordinaten x und y angegeben, der Abstand vom Laser zum Schnittpunkt über die Distanz d. Die verbleibenden drei Feature-Koordinaten sind die Z-Y-X Kardan-Winkel zwischen F_1 und F_2. Je nach zu lösender Roboteranwendung lassen sich nun Sollwerte für einzelne oder alle Feature-Koordinaten vorgeben. Die Modellierung lässt sich auch sehr einfach um weitere Beziehungen ergänzen. So zeigen De Schutter et al. (2007) ein Beispiel mit zwei Lasern sowie weitere Anwendungen.

Um die zu regelnden Feature-Koordinaten $\boldsymbol{\chi}_f$ mit den tatsächlich regelbaren Gelenkkoordinaten $\boldsymbol{q}$ des Roboters in Beziehung zu setzen, wird eine kinematische Schleife gebildet. In Abbildung 2.7 ist die Schleife für das vorherige Beispiel schematisch dargestellt. Neben Feature- und Gelenkkoordinaten beschreiben De Schutter et al. (2007) Unsicherheitskoordinaten, die es ermöglichen, geometrische Unsicherheiten explizit zu modellieren und mithilfe verschiedener Verfahren zu schätzen. Für die Anwendungsfälle dieser Arbeit sind die intrinsische Robustheit der Regelung und der Einsatz von Suchbewegungen jedoch zum Ausgleich von Unsicherheiten ausreichend.

Die Ausgangsgleichung

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{q}, \boldsymbol{\chi}_f) \tag{2.98}$$

beschreibt den Zusammenhang zwischen den Regelgrößen $\boldsymbol{y}$ und den Gelenkkoordinaten $\boldsymbol{q}$ sowie den Feature-Koordinaten $\boldsymbol{\chi}_f$. Auf Geschwindigkeitsebene lautet sie

$$\dot{\boldsymbol{y}} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{q}} \dot{\boldsymbol{q}} + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{\chi}_f} \dot{\boldsymbol{\chi}}_f \tag{2.99}$$

$$= \boldsymbol{C}_q \dot{\boldsymbol{q}} + \boldsymbol{C}_f \dot{\boldsymbol{\chi}}_f \tag{2.100}$$

mit $\boldsymbol{C}_q = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{q}}$ und $\boldsymbol{C}_f = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{\chi}_f}$. Üblicherweise sind $\boldsymbol{C}_q$ und $\boldsymbol{C}_f$ einfache Selektionsmatrizen, das heißt es werden die zu regelnden Koordinaten aus $\dot{\boldsymbol{q}}$ und $\dot{\boldsymbol{\chi}}_f$ durch entsprechende Platzierung von Einsen und Nullen in $\boldsymbol{C}_q$ und $\boldsymbol{C}_f$ ausgewählt.

Die Bindungsgleichung der Schleife

$$\boldsymbol{l}(\boldsymbol{q}, \boldsymbol{\chi}_f) = \boldsymbol{0} \tag{2.101}$$

wird ebenfalls auf Geschwindigkeitsebene formuliert

$$\begin{aligned} \boldsymbol{0} &= \frac{\partial \boldsymbol{l}}{\partial \boldsymbol{q}} \dot{\boldsymbol{q}} + \frac{\partial \boldsymbol{l}}{\partial \boldsymbol{\chi}_f} \dot{\boldsymbol{\chi}}_f & (2.102) \\ &= \boldsymbol{J}_q \dot{\boldsymbol{q}} + \boldsymbol{J}_f \dot{\boldsymbol{\chi}}_f \,, & (2.103) \end{aligned}$$

mit den Jacobi-Matrizen $\boldsymbol{J}_q = \frac{\partial \boldsymbol{l}}{\partial \boldsymbol{q}}$ und $\boldsymbol{J}_f = \frac{\partial \boldsymbol{l}}{\partial \boldsymbol{\chi}_f}$. Zu der bekannten Jacobi-Matrix $\boldsymbol{J}_q$ des Roboters kommt die Jacobi-Matrix $\boldsymbol{J}_f$ der Feature-Koordinaten hinzu. Anschaulich bedingt eine gewünschte Bewegung der Feature-Koordinaten eine entsprechende Bewegung des Roboters, um die kinematische Schleife geschlossen zu halten.

Der iTaSC-Formalismus setzt voraus, dass die Anzahl der Feature-Koordinaten der Anzahl unabhängiger Bindungsgleichungen entspricht. Somit lässt sich die Jacobi-Matrix $\boldsymbol{J}_f$ der Feature-Koordinaten invertieren. Nach der Eliminierung von $\dot{\boldsymbol{\chi}}_f$ durch Umstellen von Gleichung (2.103) und Einsetzen in Gleichung (2.100) erhält man die endgültige Ausgangsgleichung

$$\left(\boldsymbol{C}_q - \boldsymbol{C}_f \boldsymbol{J}_f^{-1} \boldsymbol{J}_q\right) \dot{\boldsymbol{q}} = \dot{\boldsymbol{y}} \tag{2.104}$$

oder kurz

$$\boldsymbol{A} \dot{\boldsymbol{q}} = \dot{\boldsymbol{y}} \tag{2.105}$$

mit

$$\boldsymbol{A} = \boldsymbol{C}_q - \boldsymbol{C}_f \boldsymbol{J}_f^{-1} \boldsymbol{J}_q \,. \tag{2.106}$$

Zwangsbedingungen (engl. *constraints*) werden durch die Vorgabe von Sollwerten für die Regelgrößen $\boldsymbol{y}$ beziehungsweise deren Geschwindigkeiten definiert. Eine einfache Variante mit Geschwindigkeitsvorsteuerung

$$\dot{\boldsymbol{y}}_d^{\circ} = \dot{\boldsymbol{y}}_d + \boldsymbol{K}_p (\boldsymbol{y}_d - \boldsymbol{y}_m) \tag{2.107}$$

beinhaltet die aktuellen Messwerte $\boldsymbol{y}_m$ und Sollwerte $\boldsymbol{y}_d$ der Regelgrößen sowie den Verstärkungsfaktor $\boldsymbol{K}_p$.

Die Lösung von Gleichung (2.105) kann mit den in Kapitel 2.1.3 vorgestellten Methoden erfolgen und wird in Abschnitt 3.6 weiter betrachtet.

Zusammenfassung

Die in diesem Kapitel vorgestellten kinematischen Grundlagen bilden die Basis für die im folgenden Kapitel behandelte Modellierung von Roboteraufgaben. Mithilfe des iTaSC-Formalismus können die Roboteraufgaben explizit und anschaulich aus Perspektive der zu handhabenden Objekte spezifiziert werden. In Konflikt tretende Roboteraufgaben können dabei in eine Rangfolge gebracht werden. Durch Verwendung der Task Priority Strategy ist sichergestellt, dass höher priorisierte Aufgaben vorrangig ausgeführt werden.

3 Modellierung und Ausführung elementarer Bausteine

Der in dieser Arbeit beschriebene Skill-Formalismus folgt dem Baukastenprinzip. Skills können aus elementaren Bausteinen zusammengesetzt werden, wobei jeder Baustein für sich nur einen Teilaspekt des gesamten Skills modelliert. Die Bausteine teilen sich auf in die Bereiche Hardwareabstraktion, kinematische Modellierung, Aufgabenspezifikation mit Reglern und Stoppbedingungen sowie Koordination der einzelnen Skills. Die Komposition und die Ausführung der Bausteine erfolgen nach einem fest definierten Schema, wodurch die Erzeugung von *Glue Code* zur manuellen Integration einzelner Bausteine entfällt.

Mit Abschnitt 3.1 wird zunächst ein Überblick über die Modellierung des Robotersystems und die Architektur der Software gegeben. Anschließend widmet sich dieses Kapitel der Beschreibung des Baukastensystems und seiner Bestandteile:

- *Identifikation und Klassifizierung der* ***elementaren Bausteine****, aus denen Skills modelliert werden (Abschnitte 3.2, 3.3 und 3.4)*
- *Aufstellung von* ***Regeln zur Komposition*** *von Skills aus einzelnen Bausteinen und durch Kombination bereits existierender Skills (Abschnitt 3.5)*
- *Implementierung eines* ***Mechanismus zur Ausführung*** *der modellierten Skills und Anwendungen (Abschnitt 3.6)*

3.1 Softwarearchitektur

Das vorgestellte Robotersystem, genannt *pitasc*, verwendet eine komponentenbasierte Architektur. Die einzelnen Programmteile sind dabei eigenständige Komponenten, die über die Middlewares ROS beziehungsweise OROCOS miteinander verbunden werden. Abbildung 3.1 zeigt eine Übersicht der primären Komponenten und Ressourcen des Robotersystems.

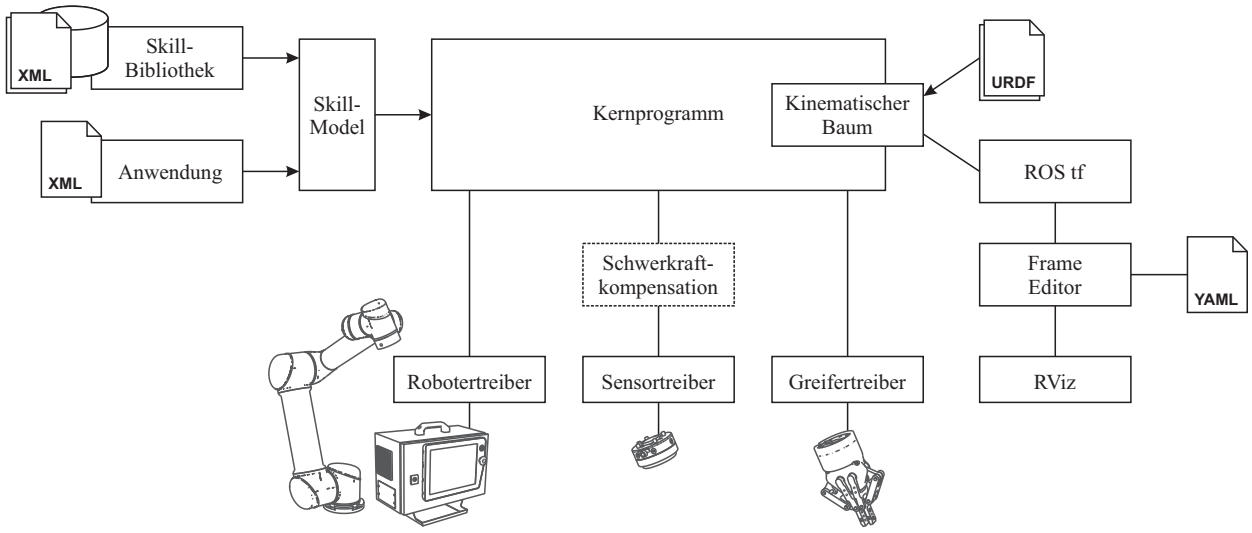


Abbildung 3.1: Primäre Komponenten und Ressourcen des Robotersystems

Das Kernprogramm enthält die elementaren Bausteine des Baukastensystems sowie den Ausführungsmechanismus. Es ist über Treiber mit Hardwaregeräten wie Roboter, Sensoren oder Greifer verbunden. Dazwischen können optional weitere Komponenten wie eine Schwerkraftkompensation geschaltet werden. Das Kernprogramm besitzt zudem einen kinematischen Baum zur Abfrage von Transformationen zwischen den Koordinatensystemen von Werkzeugen, Werkstücken und Vorrichtungen. Diese bezieht er von verschiedenen Quellen, beispielsweise ROS tf oder einem Modell der Roboterzelle im *Unified Robot Description Format* (URDF). Mithilfe des in Kapitel 4 beschriebenen Skill-Modells werden schließlich Skills und Anwendungen modelliert.

Kernprogramm

In Abgrenzung zum Skill-Modell umfasst das Kernprogramm den zur Laufzeit aktiven Programmcode in Form der elementaren Bausteine und des Ausführungsmechanismus, während das Skill-Modell deren Parametrierung und Komposition übernimmt. Der Entwurf des vorgestellten Robotersystems zielt dabei auf eine möglichst hohe Wiederverwendbarkeit der Softwarebestandteile des Robotersystems ab. Dazu ist es essenziell, nicht nur Skills als Ganzes wiederzuverwenden, sondern auch eine flexible Komposition von Skills aus Unterbausteinen zu ermöglichen. Um die Bausteine dabei möglichst vielseitig einsetzbar zu gestalten, behandelt jeder einzelne Baustein nur einen Teilaspekt der für die Realisierung eines Roboterprozesses nötigen Aufgaben. Dies folgt dem in der Softwareentwicklung verbreiteten Prinzip der Trennung der Zuständigkeiten (engl. *separation of concerns*). Zunächst erfolgt eine Aufteilung in die Kategorien Hardwareabstraktion, kinematische Modellierung des Robotersystems, Aufgabenspezifikation sowie Koordination der einzelnen Skills. Darüber hinaus existieren zu jedem

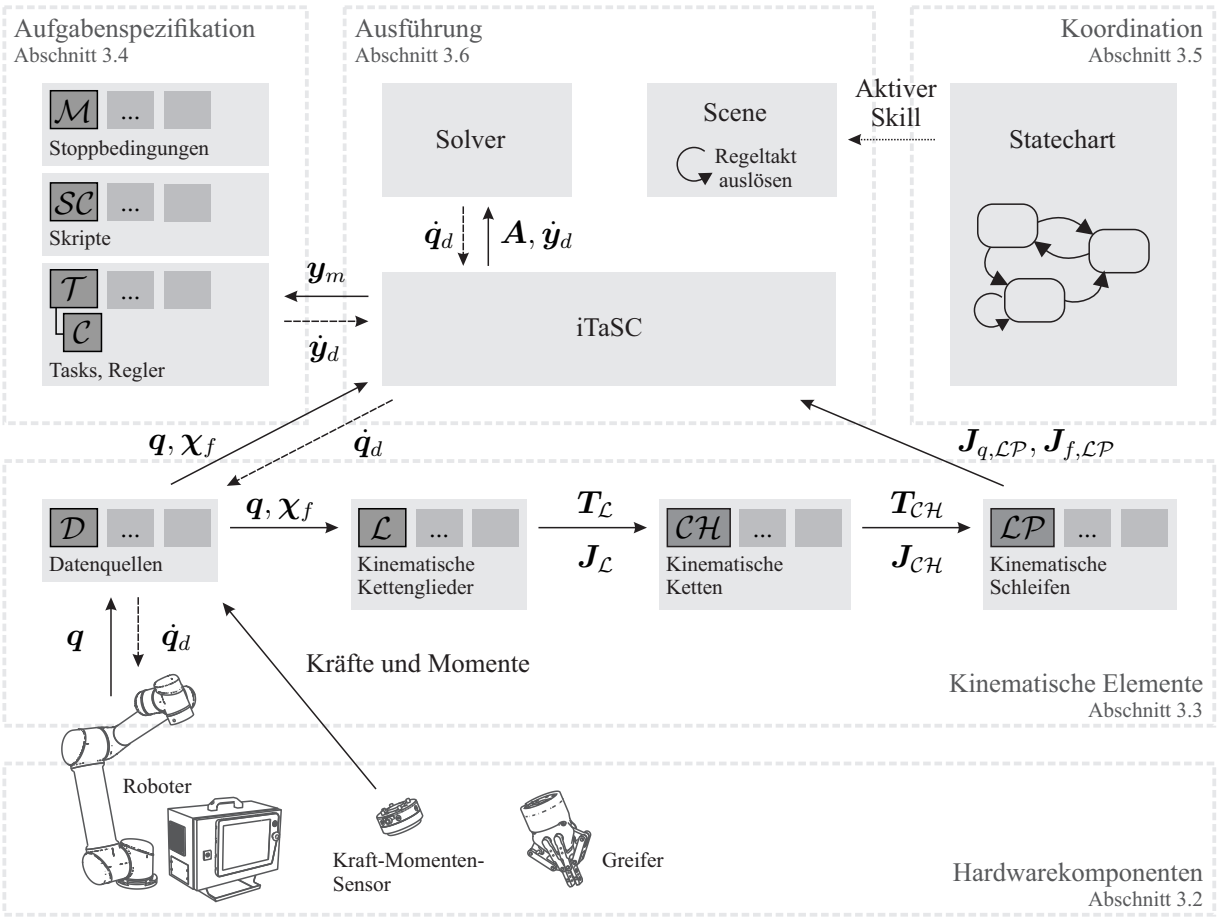


Abbildung 3.2: Elemente des Kernprogramms und Datenfluss

dieser Aspekte weitere Arten von elementaren Bausteinen. Neben der Wiederverwendbarkeit zeigt sich diese Aufteilung zudem bei der Erstellung der einzelnen Bausteine vorteilhaft, da jeweils nur Expertise in den entsprechenden Fachrichtungen wie Kinematik, Regelungstechnik oder Softwareentwicklung vorausgesetzt wird.

Im Folgenden werden die einzelnen Arten und Verantwortlichkeiten der Bausteine eingeführt, in den nachfolgenden Abschnitten des Kapitels werden diese dann vertiefend betrachtet. Die wichtigsten Elemente des Kernprogramms und der wesentliche Datenfluss zwischen diesen sind in Abbildung 3.2 aufgeführt.

Die kinematischen Elemente legen die Basis für die Aufgabenspezifikation, indem sie die Kinematik des Roboters und die geometrischen Beziehungen zwischen den Objekten des Robotersystems modellieren. Die einzelnen kinematischen Elemente bauen in Form einer Pipeline aufeinander auf und lassen sich anwendungsspezifisch zu den für die Modellierung mit iTaSC benötigten kinematischen Schleifen kombinieren. So liefern Datenquellen wie Roboter- oder Sensorschnittstellen die Koordinatenwerte für kinematische Kettenglieder, die zu kinematischen Ketten verknüpft werden, die wiederum kinematische Schleifen bilden.

Aus der kinematischen Modellierung ergeben sich die regelbaren Roboter- und Feature-Koordinaten. Die Aufgabenspezifikation definiert über Zwangsbedingungen deren gewünschten Zustand und weist ihnen Regler zu, die das Robotersystem in den Zielzustand überführen sollen. Stoppbedingungen definieren, wann der gewünschte Zustand ausreichend genau erreicht wird oder in welchen Fällen ein Fehlerfall signalisiert werden soll. Skripte ermöglichen die Kommunikation mit Peripheriekomponenten oder erfüllen weitere Hilfsfunktionen.

Die genannten Bausteine werden zu Skills kombiniert. Auch lassen sich mehrere Skills zu einem neuen Skill zusammenfassen, wobei sie sequenziell, parallel oder in Form einer Zustandsmaschine ausgeführt werden. Es entsteht dabei eine hierarchische Zustandsmaschine (Statechart), die die Koordination der einzelnen Skills und ihrer Bausteine übernimmt.

Der Ausführungsmechanismus schließlich beinhaltet die nötigen Algorithmen, um die jeweils aktiven Bausteine aufzurufen und um mithilfe ihrer Daten in jedem Regelungstakt Gelenkgeschwindigkeiten für die Ansteuerung des Roboters zu berechnen. Anschließend werden diese an den Robotertreiber versendet.

Aus Sicht einer Drei-Schichten-Architektur, wie sie in Abschnitt 2.3 vorgestellt wurde, lässt sich das Statechart der mittleren Schicht zuordnen, alle weiteren Elemente der unteren Schicht. Das Statechart koordiniert den Wechsel zwischen Skills und die Aktivierung ihrer Elemente, die dann in schneller Taktrate unter Einbezug der Sensordaten ausgeführt werden. Die obere der drei Schichten, der Planer, ist nicht vertreten, da das System nicht als autonom ausgelegt ist. An seiner Stelle befinden sich das in Kapitel 4 vorgestellte Modell und die dazugehörige DSL, mit der Benutzer Anwendungen programmieren können.

Modellierung der Roboterzelle

Die Modellierung der Roboterzelle umfasst die Beschreibung der Roboterkinematik sowie die Definition von Werkzeug- und Werkstückkoordinatensystemen. Ein zentraler Punkt beim Entwurf der hier beschriebenen Architektur ist die Trennung von Namen und Zahlenwerten der Koordinatensysteme. Bei der Aufgabenspezifikation werden in der Regel keine konkreten Koordinatenwerte angegeben (eine typische Ausnahme bilden kleine Relativbewegungen). Stattdessen werden lediglich die Namen von Koordinatensystemen angegeben, während deren konkrete Koordinatenwerte zur Laufzeit aus dem kinematischen Baum abgerufen werden. Es lassen sich damit Änderungen in der Umgebung, wie das Umplatzieren von Robotern, Werkstücken und Vorrichtungen, meist ohne eine Anpassung der Aufgabenspezifikation realisieren. Umgekehrt formuliert ist die Aufgabenspezifikation von der konkreten Anordnung von Robotern, Werkstücken oder Vorrichtungen unabhängig.

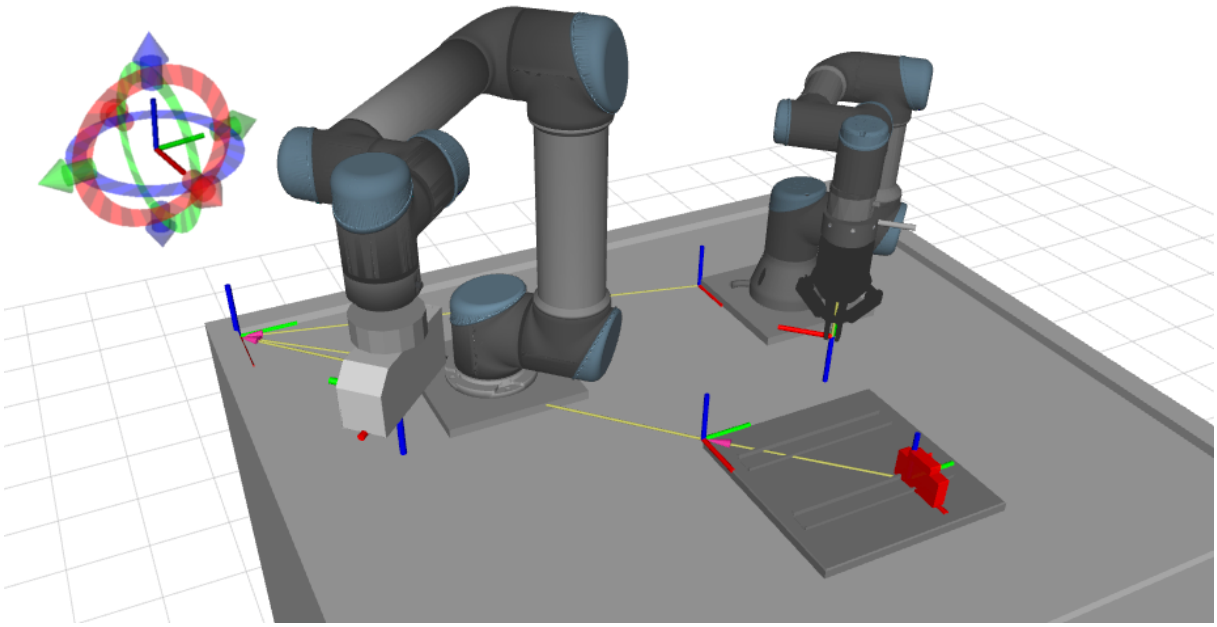


Abbildung 3.3: Modell einer Roboterzelle mit zwei Robotern

Die Softwarekomponenten der rechten Seite in Abbildung 3.1 zeigen dazu zwei mögliche Quellen für Koordinatenwerte. Zur Modellierung von Robotern, Greifern und Sensoren wird das URDF als Dateiformat gewählt. Dieses findet insbesondere bei ROS-basierten Systemen weite Verbreitung, weshalb für eine Vielzahl von Robotern und anderen Hardwarekomponenten auf existierende Modelle zurückgegriffen werden kann. Gelenke werden im URDF über eine konstante Transformation in Form einer Pose sowie eine Dreh- oder Schubachse beschrieben. Dies unterscheidet das URDF von vielen anderen Modellen, die der Denavit-Hartenberg-Konvention folgen. Die kinematischen Modelle werden geladen und in das in den nächsten Abschnitten beschriebene eigene kinematische Laufzeitmodell transformiert. Weiterhin können auch dynamische Eigenschaften, wie die Masse und Trägheit von Objekten, sowie grafische Modelle zur Visualisierung der Roboterzelle in URDF-Modellen hinterlegt werden. Diese werden für die Aufgabenspezifikation jedoch nicht verwendet. Abbildung 3.3 zeigt beispielhaft das Modell einer Roboterzelle mit zwei Robotern und der später beschriebenen Beispielanwendung zur Hutschienenbestückung.

Zur einfacheren Erstellung von Werkzeug- und Werkstückkoordinatensystemen wird die Modellierung mit dem URDF durch den *Frame Editor*[3] ergänzt, einem eigens entwickelten Plug-in für das ROS Benutzerschnittstellen-Framework *rqt*. Mit dem Frame Editor lassen sich Koordinatensysteme grafisch erstellen und verwalten. Die Koordinatensysteme werden dabei über ROS tf publiziert und lassen sich im Visualisierungswerkzeug *RViz* bearbeiten, wie in Abbildung 3.3 links oben dargestellt. Das Kernprogramm ist über den kinematischen Baum ebenfalls an ROS tf

[3] `https://github.com/ipa320/rqt_frame_editor_plugin`, quelloffen unter MIT-Lizenz

angebunden und kann von dort Transformationen zwischen beliebigen Koordinatensystemen in der Roboterzelle abrufen.

Abschließend sei hervorgehoben, dass CAD-Daten primär zur Visualisierung verwendet werden. Für eine schnelle Inbetriebnahme neuer Anwendungen ist es ausreichend, Werkzeug- und Werkstückkoordinatensysteme zu konfigurieren. Die Bereitstellung detaillierter CAD-Daten von Werkstücken oder der Umgebung ist hierbei hilfreich aber nicht zwingend notwendig. Daher wird eine Ableitung von Parametern aus CAD-Modellen nicht weiter betrachtet und auf weiterführende Arbeiten wie von Arbo et al. (2018) verwiesen.

Implementierung

Zur Evaluation des vorgestellten Skill-Modells wurde das Kernprogramm in zwei verschiedenen Varianten implementiert. Eine erste Implementierung wurde in Python vorgenommen. Während Python vergleichsweise leicht zu lernen, gut lesbar und schnell programmierbar ist, so bringt es auch den Nachteil der fehlenden Echtzeitfähigkeit mit sich. Dadurch kann nicht garantiert werden, dass zu jedem Interpolationszyklus des Roboters neue Zielwerte für die Gelenkgeschwindigkeiten berechnet werden und vorliegen. Eine zweite Implementierung des Kernprogramms wurde deshalb mit der Middleware OROCOS in C++ vorgenommen. Das in Kapitel 4 vorgestellte Skill-Modell unterstützt sowohl die Implementierung mit OROCOS als auch mit ROS.

In den folgenden Abschnitten werden relevante Modelle und Parameter formell eingeführt, die die Grundbausteine des Baukastensystems bilden (Nägele et al. 2018). Die Notation wurde dabei von Mosemann et al. (2001) und nachfolgenden Publikationen inspiriert und angepasst. Die elementaren Bausteine und ihre Parameter werden kalligrafisch dargestellt, beispielsweise der Skill $\mathcal{S}$ oder dessen Name $\mathcal{N}$. Listen von Parametern werden hierbei **fett** dargestellt. Mathematische Variablen folgen der bereits eingeführten Notation. Sollte ein Parameter auch einer mathematischen Variable entsprechen, so wird die mathematische Schreibweise bevorzugt.

3.2 Hardwarekomponenten

Über Treiber werden die Hardwarekomponenten des Robotersystems eingebunden. Dazu zählen Roboter, Kraft-Momenten-Sensoren, Greifer sowie Steuergeräte für Peripheriekomponenten wie Schrauber oder Nietpistolen. Im Folgenden wird beschrieben, welche Annahmen getroffen werden und welche Anforderungen Hardwarekomponenten erfüllen müssen, um im vorgestellten System genutzt werden zu können.

Ein wichtiger Aspekt ist hierbei zudem die Hardwareabstraktion. Im vorliegenden Kontext bedeutet dies, dass bei der Aufgabenspezifikation eine bestimmte Funktionalität unabhängig von der tatsächlich dafür eingesetzten Hardwarekomponente parametriert werden kann. Dadurch wird die Austauschbarkeit der Komponente erhöht sowie damit einhergehend die Wiederverwendbarkeit von Skills und Anwendungen. Es wird daher ebenfalls darauf eingegangen, in welchen Bereichen eine Hardwareabstraktion möglich ist und wo die Grenzen basierend auf dem aktuellen Stand der Technik liegen.

Roboter

Im Fokus der Arbeit stehen Gelenkarmroboter, insbesondere Leichtbauroboter wie der Universal Robots UR5 oder Franka Emika Panda, da diese die passenden Reichweiten, Traglasten und Erreichbarkeiten für die anvisierte Montage von Kleinbauteilen besitzen. Zudem wird die Anwendbarkeit des Systems mit kleinen Robotern klassischer Bauweise demonstriert, wie beispielsweise den Robotern KUKA KR6 und KR16.

Es wird angenommen, dass Robotersteuerungen den Empfang von Bewegungsbefehlen in Form von Gelenkgeschwindigkeiten $\dot{\boldsymbol{q}}_d$ erlauben und ihre aktuellen Gelenkwinkel $\boldsymbol{q}$ publizieren. Dies wird von den meisten Roboterherstellern unterstützt, wobei je nach Schnittstelle Taktraten zwischen 83,3 Hz und 1000 Hz möglich sind. Ebenso wird angenommen, dass dynamische Eigenschaften des Roboters und des angebrachten Werkzeugs von der Robotersteuerung weitgehend berücksichtigt werden. Der hier vorgestellte Ansatz enthält daher kein eigenes dynamisches Modell.

Aufgrund der Einfachheit der Schnittstelle ist eine Hardwareabstraktion für Roboter auf Ebene der Softwareintegration sehr einfach möglich. Anwendungen lassen sich daher prinzipiell unabhängig vom eingesetzten Roboter modellieren. Wie Abschnitt 5.4.3 beleuchtet, sind dennoch im Allgemeinen einzelne Parameteranpassungen bei den Reglern nötig oder vorteilhaft, da Roboter unterschiedliche Steifigkeiten aufweisen. Zu beachten ist außerdem, dass angesichts verschiedener Reichweiten, Traglasten und Erreichbarkeiten nicht jede Anwendung auf jeden Roboter übertragbar ist.

Kraft-Momenten-Sensor

Kraft-Momenten-Sensoren messen sowohl Kräfte als auch Momente in jeweils drei Achsrichtungen und werden üblicherweise am Roboterflansch befestigt. Einige Roboter, wie KUKA iiwa und Franka Emika Panda, messen hingegen auftretende Momente in ihren Gelenken. Diese werden

dann unter Einbezug des Robotermodells bezüglich des Werkzeugkoordinatensystems berechnet. Zu beachten ist, dass die genannten Roboter die Kraft- und Momentenwerte in invertierter Form publizieren, da sie die Kräfte darstellen, die vom Roboter auf die Umgebung ausgeübt werden. Kraft-Momenten-Sensoren hingegen geben die auf sie wirkenden Kräfte an. Teil der Hardwareabstraktion ist es entsprechend, die Vorzeichen der Werte in diesen Fällen zu invertieren, um bei der Aufgabenspezifikation eine einheitliche Verwendung zu gewährleisten. Ist dies geschehen, so ist die Modellierung der Roboteraufgabe vom eingesetzten Kraft-Momenten-Sensor unabhängig. Jedoch können auch hier Parameteranpassungen bei den Reglern nötig sein, da sich die einzelnen Sensoren hinsichtlich ihrer Steifigkeiten und Messraten unterscheiden.

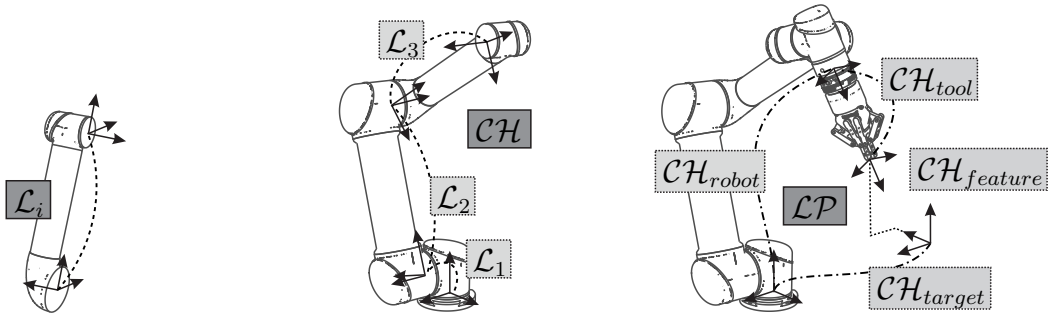
Einzelne Sensoren bieten darüber hinaus spezifische Funktionen an. So lässt sich beispielsweise mit dem Robotiq FT300 das Werkzeuggewicht bestimmen, indem der Roboter mit montiertem Werkzeug in verschiedene Orientierungen gefahren wird. Anschließend kann das Werkzeuggewicht automatisch aus den gemessenen Kräften herausgerechnet werden. Für andere Sensoren kann eine einfache Schwerkraftkompensation genutzt werden[4]. Für viele Anwendungsfälle ist jedoch die Drehung bei den Montagebewegungen so gering, dass in diesen Fällen eine Tarierung des Sensors vor der Montagebewegung ausreicht. Hierbei wird ein neuer Nullpunkt für die Sensorsignale gesetzt. Aufgabenspezifikationen, die spezifische Funktionen nutzen, müssen entsprechend des eingesetzten Sensors angepasst werden.

Greifer

Neben pneumatischen Greifern kommen vorzugsweise elektrische Greifer zum Einsatz. Viele elektrische Greifer erlauben eine genaue Einstellung und Regelung der Greifkraft. Auch die Bauteilbreite lässt sich meist angeben und der Greifer kann durch Messung von Position und Kraft prüfen, ob tatsächlich ein Bauteil mit der gewünschten Breite gegriffen wurde. Zudem lässt sich bei vielen Greifern die Geschwindigkeit der Greiferbacken beim Schließen und Öffnen definieren.

Wie in Abschnitt 5.4.3 gezeigt wird, sind die Schnittstellen elektrischer Greifer jedoch sehr unterschiedlich. Eine einheitliche Schnittstelle und damit eine Hardwareabstraktion unterstützt der Stand der Technik nicht. Die verschiedenen Schnittstellen und Funktionen einzelner Greifer erfordern daher eine individuelle Ansteuerung über Skripte und damit Anpassungen der Aufgabenspezifikation bei der Übertragung auf Robotersysteme mit anderen Greifern.

[4] `https://github.com/ipa320/g_compensator`, quelloffen unter MIT-Lizenz



(a) Kinematisches Kettenglied (b) Kinematische Kette (c) Kinematische Schleife

Abbildung 3.4: Elementare kinematische Bausteine

3.3 Kinematische Elemente

Die in diesem Abschnitt vorgestellten Elemente modellieren die Kinematik des Roboters sowie die geometrischen Beziehungen zwischen den Objekten des Robotersystems. Sie stellen die Gelenk- und Feature-Koordinaten für die spätere Aufgabenspezifikation bereit sowie Transformations- und Jacobi-Matrizen für die Lösung der Gleichungssysteme.

Die kinematischen Elemente bauen aufeinander auf und werden entsprechend dieser Reihenfolge vorgestellt. So stellen Datenquellen Koordinatenwerte für kinematische Kettenglieder bereit, die zu kinematischen Ketten verknüpft werden, die wiederum kinematische Schleifen bilden. Dieser Zusammenhang ist in Abbildung 3.4 bildlich dargestellt.

3.3.1 Datenquelle

Eine Datenquelle (`data_source`)

$$\mathcal{D} = \langle \boldsymbol{\mathcal{Y}} \rangle \tag{3.1}$$

stellt Werte für eine Liste $\boldsymbol{\mathcal{Y}}$ an Gelenk- oder Feature-Koordinaten bereit. Datenquellen befinden sich auf der untersten architektonischen Ebene des Kernprogramms und abstrahieren die Herkunft der Koordinatenwerte. Sie dienen damit einerseits als Abstraktionsschicht für Hardwarekomponenten, andererseits werden auch interne Komponenten abstrahiert, die Koordinatenwerte bereitstellen, wie beispielsweise der kinematische Baum zur Verwaltung von Koordinatensystemen. Die Aufgabe der Datenquellen besteht darin, die Koordinatenwerte entsprechend des Zwecks ihrer späteren Verwendung vorzubereiten. Sie transformieren beispielsweise einen Kraftvektor in das erforderliche Bezugskoordinatensystem oder berechnen Werte für Features in Zylinderkoordinaten.

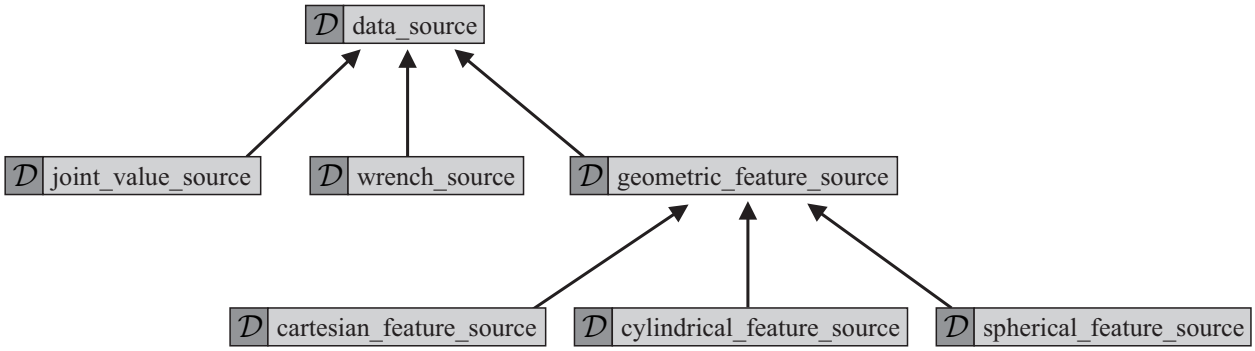


Abbildung 3.5: Arten von Datenquellen

Abbildung 3.5 stellt die vorhandenen Arten von Datenquellen dar, auf die im Folgenden jeweils eingegangen wird. Diese Auflistung kann darüber hinaus erweitert werden, um zusätzliche Quellen von Sensordaten oder Feature-Koordinaten zu modellieren. Hierzu zählen beispielsweise Bilddaten (De Laet et al. 2012), die in dieser Arbeit nicht weiter betrachtet werden.

Gelenkkoordinaten

Eine Datenquelle für Gelenkkoordinaten (`joint_value_source`) abstrahiert die Kommunikation mit dem Robotertreiber und stellt die aktuellen Werte der Gelenkkoordinaten des Roboters

$$\boldsymbol{q} = \begin{bmatrix} q_1 & q_2 & \cdots & q_n \end{bmatrix}^\mathsf{T} \tag{3.2}$$

bereit. Zudem sendet sie nach Abschluss der Berechnungen des Ausführungsalgorithmus die gewünschten Gelenkwinkelgeschwindigkeiten zurück an den Robotertreiber.

Kraft-Momenten Feature-Koordinaten

Die Datenquelle für Kraft-Momenten Feature-Koordinaten (`wrench_source`) abstrahiert auf eine ähnliche Weise die Kommunikation mit dem Sensortreiber. Für einen sechsachsigen Kraft-Momenten-Sensor werden die Feature-Koordinaten

$$\boldsymbol{\chi}_{force} = \begin{bmatrix} f_x & f_y & f_z & \mu_x & \mu_y & \mu_z \end{bmatrix}^\mathsf{T} \tag{3.3}$$

bereitgestellt.

Die Aufgabe der Datenquelle ist es, die empfangenen Messwerte für ihre jeweilige Verwendung aufzubereiten. So ist das Sensorkoordinatensystem in der Regel nicht das Bezugskoordinatensystem, in dem die Kraftwerte bei der späteren Aufgabenspezifikation definiert werden sollen.

Dies kann stattdessen ein Werkzeug-, Werkstück- oder Roboterzellenkoordinatensystem sein. Es findet daher eine Transformation des Kraftvektors

$$\begin{bmatrix} {}^2\boldsymbol{f}_2 \\ {}^2\boldsymbol{\mu}_2 \end{bmatrix} = \begin{bmatrix} {}^2_1\boldsymbol{R} & \boldsymbol{0} \\ \left[{}^2\boldsymbol{r}_{21}\right]_\times {}^2_1\boldsymbol{R} & {}^2_1\boldsymbol{R} \end{bmatrix} \begin{bmatrix} {}^1\boldsymbol{f}_1 \\ {}^1\boldsymbol{\mu}_1 \end{bmatrix} \tag{3.4}$$

vom Sensorkoordinatensystem O_1 in das Bezugskoordinatensystem O_2 statt. Dies setzt voraus, dass sich beide Koordinatensysteme auf demselben starren Körper befinden oder sich die Kräfte und Momente über Form- oder Kraftschluss übertragen lassen.

Werden die Kraft- und Momentenwerte in den in Abschnitt 3.2 beschriebenen Sonderfällen invertiert empfangen, so ist es ebenso Aufgabe der Datenquelle, die Vorzeichen anzupassen, um bei der späteren Aufgabenspezifikation eine einheitliche Verwendung unabhängig von der Herkunft der Kraftdaten zu gewährleisten.

Kartesische Feature-Koordinaten

Kartesische Feature-Koordinaten (`cartesian_feature_source`) modellieren die geometrische Beziehung zwischen zwei Koordinatensystemen als Pose

$$\boldsymbol{\chi}_{cartesian} = \begin{bmatrix} x & y & z & \alpha & \beta & \gamma \end{bmatrix}^\mathsf{T} \tag{3.5}$$

mit den in Abschnitt 2.1.1 vorgestellten Z-Y-X Kardan-Winkeln. Aus dem kinematischen Baum wird die aktuelle räumliche Lage der beiden Koordinatensysteme zueinander in Form einer homogenen Transformationsmatrix abgerufen. Die translatorischen Feature-Koordinaten können direkt aus dieser Transformationsmatrix entnommen werden. Für die Bestimmung der rotatorischen Feature-Koordinaten aus den Werten r_{ij} der Drehmatrix[5]

$$\boldsymbol{R}(\boldsymbol{\phi}) = \boldsymbol{R}_Z(\alpha)\boldsymbol{R}_Y(\beta)\boldsymbol{R}_X(\gamma) \tag{3.6}$$

$$= \begin{bmatrix} c_\alpha c_\beta & c_\alpha s_\beta s_\gamma - s_\alpha c_\gamma & c_\alpha s_\beta c_\gamma + s_\alpha s_\gamma \\ s_\alpha c_\beta & s_\alpha s_\beta s_\gamma + c_\alpha c_\gamma & s_\alpha s_\beta c_\gamma - c_\alpha s_\gamma \\ -s_\beta & c_\beta s_\gamma & c_\beta c_\gamma \end{bmatrix} \tag{3.7}$$

$$= \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{3.8}$$

[5] mit $c_\alpha = \cos(\alpha), s_\beta = \sin(\beta)$ usw.

wird die *Kinematics and Dynamics Library* (KDL) und entsprechend deren Konventionen verwendet. So liegt β im Intervall $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ und die einzelnen Koordinaten ergeben sich zu

$$\alpha = \operatorname{atan2}(r_{21}, r_{11}) , \tag{3.9}$$

$$\beta = \operatorname{atan2}\left(-r_{31}, \sqrt{r_{11}^2 + r_{21}^2}\right) , \tag{3.10}$$

$$\gamma = \operatorname{atan2}(r_{32}, r_{33}) . \tag{3.11}$$

Im Fall von $c_\beta = 0$ lässt sich lediglich die Summe aus α und γ bestimmen. Die KDL wählt hierbei

$$\alpha = \operatorname{atan2}(r_{12}, r_{22}) , \tag{3.12}$$

$$\gamma = 0 . \tag{3.13}$$

Bei den genannten Formeln findet die mathematische Funktion atan2 Verwendung[6], eine Erweiterung des Arkustangens, die alle vier Quadranten berücksichtigt.

Für einige Anwendungen, beispielsweise die Bewegung auf einer Kreisbahn, wäre eine Aufgabenspezifikation auf Basis von kartesischen Koordinaten sehr umständlich. Hierfür bietet sich eine alternative Modellierung von Feature-Koordinaten an, beispielsweise mit Zylinder- oder Kugelkoordinaten.

Zylinderkoordinaten

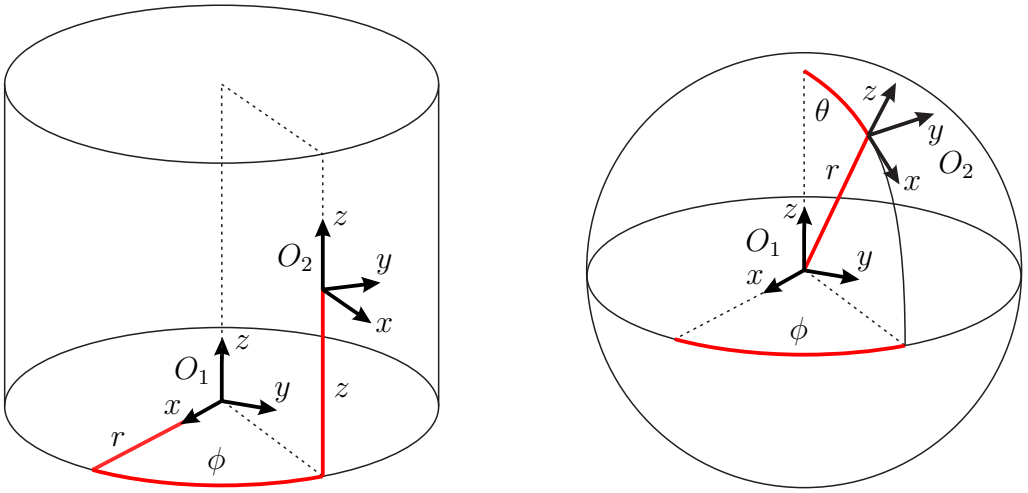
Abbildung 3.6a stellt die Modellierung mittels der Zylinderkoordinaten

$$\boldsymbol{\chi}_{cylindrical} = \begin{bmatrix} \phi & r & z & \alpha & \beta & \gamma \end{bmatrix}^\mathsf{T} \tag{3.15}$$

dar (`cylindrical_feature_source`), mit Azimutwinkel ϕ, Radius r und Höhe z. Die Orientierung des Koordinatensystems O_2 wird erneut mit Z-Y-X Kardan-Winkeln beschrieben. Dabei wurden die Achsrichtungen so gewählt, dass bei $\boldsymbol{\chi}_{cylindrical} = \mathbf{0}$ die Koordinatensysteme O_1 und O_2 gleich ausgerichtet sind.

[6] Der atan2 wird üblicherweise definiert zu

$$\operatorname{atan2}(y, x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & \text{für } x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & \text{für } x < 0,\ y \geq 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & \text{für } x < 0,\ y < 0 \\ +\frac{\pi}{2} & \text{für } x = 0,\ y > 0 \\ -\frac{\pi}{2} & \text{für } x = 0,\ y < 0 \end{cases} \tag{3.14}$$



(a) Zylinderkoordinaten (b) Kugelkoordinaten

Abbildung 3.6: Modellierung der Feature-Koordinaten mithilfe von Zylinder- oder Kugelkoordinaten

Die Bestimmung der Feature-Koordinaten aus den Werten der homogenen Transformationsmatrix erfolgt mit

$$r = \sqrt{x^2 + y^2}\,, \tag{3.16}$$

$$\phi = \operatorname{atan2}(y, x)\,, \tag{3.17}$$

$$z = z\,. \tag{3.18}$$

Die Z-Y-X Kardan-Winkel lassen sich aus der restlichen Drehung

$$\boldsymbol{R}_Z(\alpha)\boldsymbol{R}_Y(\beta)\boldsymbol{R}_X(\gamma) = \boldsymbol{R}_Z(\phi)^{-1}\,\boldsymbol{R}(\boldsymbol{\chi}_{cylindrical}) \tag{3.19}$$

mit der zuvor vorgestellten Methode bestimmen.

Kugelkoordinaten

Als dritte Möglichkeit zur Modellierung von geometrischen Feature-Koordinaten werden die Kugelkoordinaten (`spherical_feature_source`)

$$\boldsymbol{\chi}_{spherical} = \begin{bmatrix}\phi & \theta & r & \alpha & \beta & \gamma\end{bmatrix}^{\mathsf{T}}\,, \tag{3.20}$$

wie in Abbildung 3.6b gezeigt, definiert. Sie setzen sich aus dem Azimutwinkel ϕ, Polarwinkel θ und Radius r sowie den Z-Y-X Kardan-Winkeln zusammen. Analog zu den Zylinderkoordinaten wurde die Orientierung des Koordinatensystems O_2 so gewählt, dass die Koordinatensysteme

O_1 und O_2 bei $\boldsymbol{\chi}_{spherical} = \mathbf{0}$ gleich ausgerichtet sind. Ebenfalls analog dazu erfolgt auch die Bestimmung der Feature-Koordinaten

$$r = \sqrt{x^2 + y^2 + z^2}\,, \tag{3.21}$$

$$\phi = \operatorname{atan2}(y, x)\,, \tag{3.22}$$

$$\theta = \arccos\left(\frac{z}{r}\right) \tag{3.23}$$

aus den Werten der homogenen Transformationsmatrix. Da $|z| \leq r$ gilt und $\frac{z}{r}$ somit auf das Intervall $[-1, 1]$ begrenzt ist, ist $\arccos\left(\frac{z}{r}\right)$ stets definiert. Zuletzt lassen sich die Z-Y-X Kardan-Winkel aus der restlichen Drehung

$$\boldsymbol{R}_Z(\alpha)\boldsymbol{R}_Y(\beta)\boldsymbol{R}_X(\gamma) = \left(\boldsymbol{R}_Z(\phi)\boldsymbol{R}_Y(\theta)\right)^{-1} \boldsymbol{R}(\boldsymbol{\chi}_{spherical}) \tag{3.24}$$

bestimmen.

3.3.2 Kinematisches Kettenglied

Ein serieller Roboter besteht aus mehreren einzelnen Segmenten, die über Gelenke miteinander verbunden sind. Ein kinematisches Kettenglied (`link`)

$$\mathcal{L} = \langle \mathcal{TY}, \mathcal{D}, \mathcal{Y} \rangle\,, \tag{3.25}$$

mit Kettenglied-Typ $\mathcal{TY}$, Datenquelle $\mathcal{D}$ und Koordinatenname $\mathcal{Y}$, modelliert ein solches Segment bestehend aus einem Gelenk und einem starren Körper. Das Kettenglied besitzt demnach genau eine Gelenkkoordinate und damit einen Freiheitsgrad.

Neben Robotergelenken werden auch Feature-Beziehungen mit kinematischen Kettengliedern modelliert. Jede Feature-Koordinate wird dabei über ein (virtuelles) Kettenglied definiert. Als dritte Variante kann ein starrer Körper in Form eines Objekt-Kettenglieds modelliert werden, das keinen Freiheitsgrad besitzt, beispielsweise um ein Werkzeugkoordinatensystem relativ zum Roboterflansch zu beschreiben. Die drei Varianten werden über den Kettenglied-Typ $\mathcal{TY} \in \{robot, feature, object\}$ unterschieden.

Die Aufgabe eines Kettenglieds ist die Bereitstellung einer Transformationsmatrix und (mit Ausnahme von Objekt-Kettengliedern) einer Jacobi-Matrix. Objekt-Kettenglieder definieren ihre Transformationsmatrix beim Programmstart oder lesen sie zu jedem Zeitschritt aus dem kinematischen Baum aus. Ein Kettenglied mit Gelenk- oder Feature-Koordinate liest den momentanen Wert seiner Koordinate aus einer Datenquelle $\mathcal{D}$ aus und aktualisiert damit seine

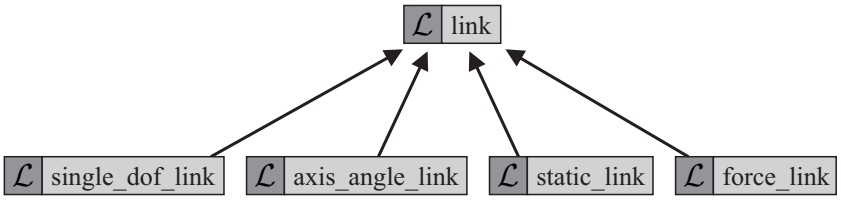


Abbildung 3.7: Implementierungsarten von Kettengliedern

Transformationsmatrix. Die Koordinate erhält darüber hinaus einen menschenlesbaren Namen $\mathcal{Y}$, um die spätere Aufgabenspezifikation zu erleichtern. Dies kann beispielsweise für ein Robotergelenk *elbow_joint* sein, für ein Feature bei Zylinderkoordinaten *radius* oder *height*.

In Abbildung 3.7 sind die verschiedenen Arten der Implementierung von Kettengliedern dargestellt. Sie unterscheiden sich unter anderem darin, ob und wie ihre Transformations- und Jacobi-Matrizen berechnet werden. Die einzelnen Arten werden im Folgenden jeweils kurz vorgestellt. Nicht zu verwechseln sind diese Implementierungsarten mit den Kettenglied-Typen $\mathcal{TY}$, die beschreiben, ob es sich um ein Roboter-, Feature- oder Objekt-Kettenglied handelt und damit den Verwendungszweck darstellen.

Kettenglied mit einachsigem Gelenk

Ein Kettenglied mit einachsigem Gelenk (`single_dof_link`, von engl. *degree of freedom*, Freiheitsgrad) kann sowohl eine Drehachse als auch eine Translationsachse sein. Zu den Parametern des Kettenglieds

$$\mathcal{L}_{single\_dof} = \langle \mathcal{TY}, \mathcal{D}, \mathcal{Y}, \boldsymbol{T}_c, \mathcal{Q} \rangle \tag{3.26}$$

gehört neben den bereits genannten allgemeinen Parametern $\mathcal{TY}$, $\mathcal{D}$ und $\mathcal{Y}$ eine konstante Transformation $\boldsymbol{T}_c$ des Segments, wie in Abbildung 3.8 bildlich dargestellt. An diese Transformation schließt sich die eigentliche Dreh- oder Translationsachse an, deren Freiheitsgrad mit $\mathcal{Q} \in \{x, y, z, \alpha, \beta, \gamma\}$ definiert wird. Daraus ergibt sich die gesamte Transformationsmatrix des Kettenglieds, beispielsweise für eine translatorische Achse entlang x zu

$$\boldsymbol{T}_x = \boldsymbol{T}_c + \begin{bmatrix} & & x \\ & \boldsymbol{I} & 0 \\ & & 0 \\ \boldsymbol{0}^\mathsf{T} & & 1 \end{bmatrix} \tag{3.27}$$

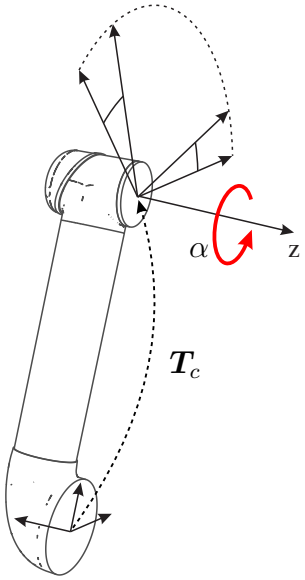


Abbildung 3.8: Transformation eines Kettenglieds mit Drehgelenk

oder für eine rotatorische Achse mit Drehung α um die z-Achse zu

$$\boldsymbol{T}_\alpha = \boldsymbol{T}_c + \begin{bmatrix} \boldsymbol{R}_Z(\alpha) & \boldsymbol{0} \\ \boldsymbol{0}^\mathsf{T} & 1 \end{bmatrix} . \tag{3.28}$$

In der hier beschriebenen Konvention für Kettenglieder erfolgt die Transformation der Achse *nach* der konstanten Transformation des Segments. Dies entspricht der Konvention des URDF, unterscheidet sich jedoch von der Denavit–Hartenberg Konvention oder Herleitungen wie in Siciliano et al. (2010) und Craig (2005), bei der der starre Körper i auf Achse i folgt.

Die Definition der Jacobi-Matrix des Kettenglieds erfolgt stets bezüglich des End-Koordinatensystems des Kettenglieds, wodurch ihre Angabe stark vereinfacht wird. Für eine translatorische Achse entlang x lautet die 6×1 Jacobi-Matrix beispielsweise

$$\boldsymbol{J}_x = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^\mathsf{T} . \tag{3.29}$$

Ein Drehgelenk, das um die z-Achse dreht, besitzt die Jacobi-Matrix

$$\boldsymbol{J}_\alpha = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}^\mathsf{T} . \tag{3.30}$$

Die Transformation der Jacobi-Matrizen aller Kettenglieder in ein einheitliches Bezugssystem ist Aufgabe der kinematischen Kette.

Kettenglied mit Drehachse und Drehwinkel

Drehgelenke des URDF verwenden zur Repräsentation der Orientierung Drehachse und Drehwinkel. Entsprechend ist die Achse $\boldsymbol{r}$ Teil der Parameter des Kettenglieds (`axis_angle_link`)

$$\mathcal{L}_{axis\_angle} = \langle \mathcal{TY}, \mathcal{D}, \mathcal{Y}, \boldsymbol{T}_c, \boldsymbol{r} \rangle \,. \tag{3.31}$$

Die Transformationsmatrix des Kettenglieds berechnet sich unter Berücksichtigung des aktuellen Koordinatenwerts ϑ zu

$$\boldsymbol{T} = \boldsymbol{T}_c + \begin{bmatrix} \boldsymbol{R}(\vartheta, \boldsymbol{r}) & \boldsymbol{0} \\ \boldsymbol{0}^\mathsf{T} & 1 \end{bmatrix} \tag{3.32}$$

mit der in Gleichung (2.4) vorgestellten Drehmatrix $\boldsymbol{R}(\vartheta, \boldsymbol{r})$. Die Jacobi-Matrix

$$\boldsymbol{J} = \begin{bmatrix} 0 & 0 & 0 & r_x & r_y & r_z \end{bmatrix}^\mathsf{T} \tag{3.33}$$

lässt sich mithilfe der normierten Achse $\boldsymbol{r}$ bestimmen.

Statisches Kettenglied

Ein statisches Kettenglied (`static_link`) besitzt keinen Freiheitsgrad und demnach auch keine Gelenk- oder Feature-Koordinate. Es hat eine reduzierte Anzahl an Parametern

$$\mathcal{L}_{static} = \langle \boldsymbol{T}_c \rangle \tag{3.34}$$

und ist stets vom Typ *object*. Die Transformationsmatrix $\boldsymbol{T}_c$ wird entweder fest vorgegeben oder zu jedem Zeitschritt aktuell aus dem kinematischen Baum ausgelesen.

Kettenglied für Kraft-Feature-Koordinaten

Kettenglieder für Kraft-Feature-Koordinaten

$$\mathcal{L}_{force} = \langle \mathcal{D}, \mathcal{Y}, \mathcal{Q} \rangle \tag{3.35}$$

sind stets vom Typ *feature*. Ähnlich zu einem einachsigen Gelenk besitzt das Kettenglied eine Variable $\mathcal{Q} \in \{x, y, z, \alpha, \beta, \gamma\}$, mit der die Achsrichtung der Kraft-Feature-Koordinate gewählt wird.

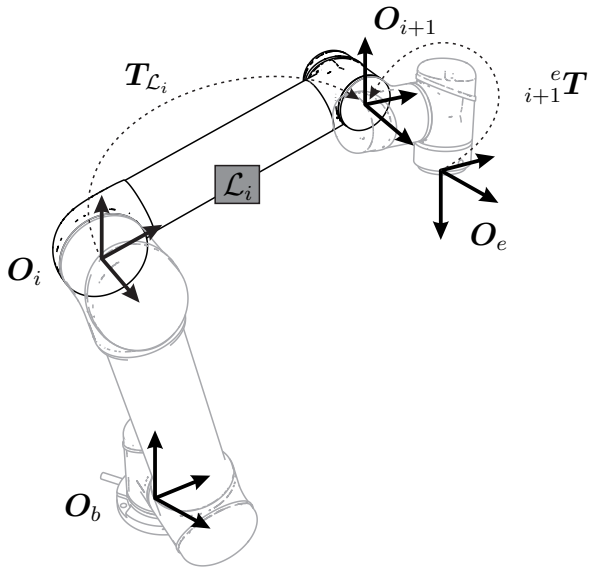


Abbildung 3.9: Transformation der Jacobi-Matrix

Das Kettenglied besitzt keine räumliche Ausdehnung, da Kraft-Momenten-Sensoren als hinreichend steif angenommen werden. Die Transformationsmatrix entspricht demnach der Einheitsmatrix $\boldsymbol{T}_c = \boldsymbol{I}$. Ist die Verformung des Sensors relevant, so ist die Modellierung eines entsprechenden Elements nötig. Dies wird hier jedoch nicht betrachtet.

3.3.3 Kinematische Kette

Eine kinematische Kette (`chain`)

$$\mathcal{CH} = \langle \mathcal{TY}, \boldsymbol{\mathcal{L}} \rangle \tag{3.36}$$

fasst mehrere zusammenhängende kinematische Kettenglieder $\boldsymbol{\mathcal{L}}$ zusammen. Wie auch die einzelnen Kettenglieder besitzt die kinematische Kette einen Typ $\mathcal{TY} \in \{robot, feature, object\}$. Roboter- und Feature-Ketten dürfen dabei nicht Kettenglieder vom jeweils anderen Typ enthalten. Sie können jedoch mit beliebig vielen Kettengliedern vom Typ *object* ergänzt werden, um beispielsweise ein Werkzeugkoordinatensystem zu modellieren. Ketten vom Typ *object* können ausschließlich Kettenglieder enthalten, die ebenfalls den Typ *object* besitzen. Sie dienen beim späteren Schließen der kinematischen Schleifen zum Verbinden von Roboter- und Feature-Ketten.

Die Aufgabe einer kinematischen Kette ist die Berechnung ihrer Vorwärtskinematik und Jacobi-Matrix. Dies geschieht schrittweise je Kettenglied und rückwärts, ausgehend vom letzten Kettenglied der kinematischen Kette. Abbildung 3.9 stellt den Iterationsschritt für Kettenglied $\mathcal{L}_i$ dar.

Wie im vorherigen Abschnitt beschrieben wird die Jacobi-Matrix $\boldsymbol{J}_i$ bezüglich des End-Koordinatensystems von Kettenglied $\mathcal{L}_i$ angegeben. Dieses entspricht dem Start-Koordinatensystem des nachfolgenden Kettenglieds $\mathcal{L}_{i+1}$. Nach Gleichung (2.36) wird die Jacobi-Matrix daher mit

$$ {}^{e}\boldsymbol{J}_i = \begin{bmatrix} {}^{e}_{i+1}\boldsymbol{R} & \left[{}^{e}\boldsymbol{r}_{e,i+1}\right]_{\times}\, {}^{e}_{i+1}\boldsymbol{R} \\ \boldsymbol{0} & {}^{e}_{i+1}\boldsymbol{R} \end{bmatrix} \boldsymbol{J}_i \tag{3.37} $$

in das End-Koordinatensystem $\boldsymbol{O}_e$ der Kette transformiert. Die Drehmatrix ${}^{e}_{i+1}\boldsymbol{R}$ und der Positionsvektor ${}^{e}\boldsymbol{r}_{e,i+1}$ können direkt der Transformationsmatrix ${}^{e}_{i+1}\boldsymbol{T}$ entnommen werden. Für den nächsten Iterationsschritt wird anschließend die neue Transformationsmatrix

$$ {}^{e}_{i}\boldsymbol{T} = {}^{e}_{i+1}\boldsymbol{T}\, {}^{i+1}_{i}\boldsymbol{T} \tag{3.38} $$

$$ = {}^{e}_{i+1}\boldsymbol{T}\, \boldsymbol{T}^{-1}_{\mathcal{L}_i} \tag{3.39} $$

bestimmt. Kettenglieder vom Typ *object* besitzen selbst keinen Freiheitsgrad und werden bei der Iteration durch die Kette lediglich für die Berechnung der jeweiligen Transformationsmatrix ${}^{e}_{i}\boldsymbol{T}$ herangezogen.

Nach dem Durchlaufen aller Kettenglieder ergibt sich die Transformation der gesamten Kette von $\boldsymbol{O}_b$ nach $\boldsymbol{O}_e$ zu

$$ \boldsymbol{T}_{\mathcal{CH}} = {}^{b}_{e}\boldsymbol{T} = {}^{e}_{b}\boldsymbol{T}^{-1} \tag{3.40} $$

und die $6 \times n_l$ Jacobi-Matrix zu

$$ {}^{e}\boldsymbol{J}_{\mathcal{CH}} = \begin{bmatrix} {}^{e}\boldsymbol{J}_{\mathcal{L},1} & {}^{e}\boldsymbol{J}_{\mathcal{L},2} & \cdots & {}^{e}\boldsymbol{J}_{\mathcal{L},n_l} \end{bmatrix}, \tag{3.41} $$

bei n_l Kettengliedern, die eine Gelenk- beziehungsweise eine Feature-Koordinate besitzen.

3.3.4 Kinematische Schleife

Eine kinematische Schleife (`loop`) fasst mehrere kinematische Ketten zu einer Schleife zusammen. Die Schleife

$$ \mathcal{LP} = \langle \boldsymbol{\mathcal{CH}}_q, \boldsymbol{\mathcal{CH}}_f \rangle \tag{3.42} $$

enthält dafür zwei Listen an Ketten. Die Liste $\boldsymbol{\mathcal{CH}}_q$ enthält eine oder mehrere Ketten vom Typ *robot* und beliebig viele Ketten vom Typ *object*. Die Liste $\boldsymbol{\mathcal{CH}}_f$ wiederum enthält eine oder mehrere Ketten vom Typ $feature$ und kann ebenfalls Ketten vom Typ *object* enthalten. Die Ketten beider Listen müssen zusammen eine geschlossene Schleife bilden.

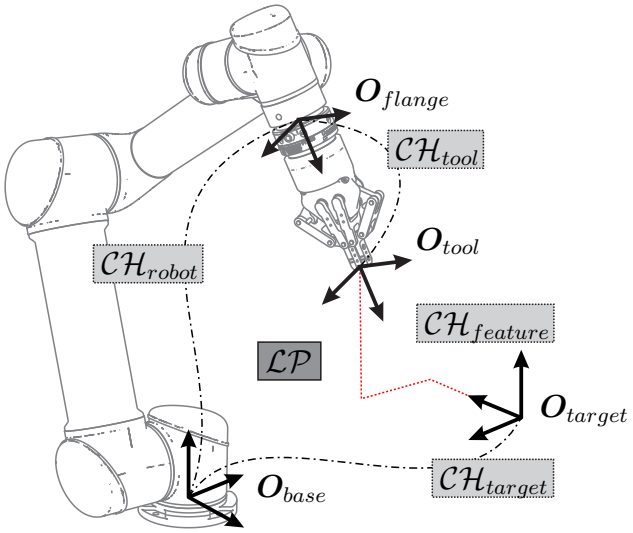


Abbildung 3.10: Kinematische Schleife $\mathcal{LP}$ mit der Roboter-Kette $\mathcal{CH}_{robot}$, Feature-Kette $\mathcal{CH}_{feature}$ und den zwei Objekt-Ketten $\mathcal{CH}_{tool}$ und $\mathcal{CH}_{target}$

Abbildung 3.10 zeigt ein Beispiel für eine kinematische Schleife mit den kinematischen Ketten

$$\boldsymbol{\mathcal{CH}}_q = \langle \mathcal{CH}_{robot}, \mathcal{CH}_{tool} \rangle \tag{3.43}$$

auf der Seite des Roboters und den kinematischen Ketten

$$\boldsymbol{\mathcal{CH}}_f = \langle \mathcal{CH}_{target}, \mathcal{CH}_{feature} \rangle \tag{3.44}$$

auf der Seite der Feature-Koordinaten. Beide Teile der Schleife gehen in diesem Fall vom Basis-Koordinatensystem O_{base} des Roboters aus und treffen sich im Koordinatensystem O_{tool}.

Im Vergleich zu der in Abbildung 2.7 gezeigten Schleife des iTaSC-Formalismus entfällt bei dieser Art der Modellierung die explizite Angabe von genau zwei Objekt- und zwei Feature-Koordinatensystemen. Ob und auf wie viele Feature-Ketten sich die Feature-Koordinaten aufteilen, ist nicht vorgegeben. Zwingend erforderlich ist nur, dass je Schleife in Summe genau sechs Feature-Koordinaten vorhanden sind.

Für die spätere Berechnung der Matrix $\boldsymbol{A} = \boldsymbol{C}_q - \boldsymbol{C}_f \boldsymbol{J}_f^{-1} \boldsymbol{J}_q$ aus Gleichung (2.105) müssen die Jacobi-Matrizen $\boldsymbol{J}_q$ und $\boldsymbol{J}_f$ bestimmt werden, die sich aus den Jacobi-Matrizen der einzelnen kinematischen Schleifen zusammensetzen. Die Jacobi-Matrix $\boldsymbol{J}_{q,\mathcal{LP}}$ einer Schleife wird aus den Ketten der Liste $\boldsymbol{\mathcal{CH}}_q$ bestimmt, $\boldsymbol{J}_{f,\mathcal{LP}}$ entsprechend aus den Ketten der Liste $\boldsymbol{\mathcal{CH}}_f$. Die Anzahl der Gelenkkoordinaten $\boldsymbol{q}_{\mathcal{LP}i}$ definiert dabei die Dimension der $6 \times n_q$ Jacobi-Matrix $\boldsymbol{J}_{q,\mathcal{LP}}$, wohingegen $\boldsymbol{J}_{f,\mathcal{LP}}$ stets sechs Feature-Koordinaten $\boldsymbol{\chi}_{f,\mathcal{LP}i}$ und damit die Dimension 6×6 besitzt. Beide Jacobi-Matrizen werden bezüglich des gemeinsamen End-Koordinatensystems ausgedrückt, in obigem Beispiel O_{tool}. Die Transformation erfolgt mit den in Abschnitt 3.3.3 vor-

gestellten Formeln zur Transformation von Jacobi-Matrizen. Es ergeben sich die feature-seitige Jacobi-Matrix

$$\boldsymbol{J}_{f,\mathcal{LP}} = \begin{bmatrix} \boldsymbol{J}_{\mathcal{CH}_f,1} & \boldsymbol{J}_{\mathcal{CH}_f,2} & \cdots & \boldsymbol{J}_{\mathcal{CH}_f,m_f} \end{bmatrix} \tag{3.45}$$

bei m_f Feature-Ketten und die roboterseitige Jacobi-Matrix

$$\boldsymbol{J}_{q,\mathcal{LP}} = \begin{bmatrix} \boldsymbol{J}_{\mathcal{CH}_q,1} & \boldsymbol{J}_{\mathcal{CH}_q,2} & \cdots & \boldsymbol{J}_{\mathcal{CH}_q,m_q} \end{bmatrix} \tag{3.46}$$

bei m_q Roboter-Ketten.

3.4 Aufgabenspezifikation

Die kinematische Modellierung aus dem vorangegangenen Abschnitt beschreibt die Beziehungen zwischen Objekten und drückt sie über Roboter- und Feature-Koordinaten aus. Diese werden bei der Aufgabenspezifikation mit Zwangsbedingungen belegt, die den erwünschten Zustand vorgeben. Auch werden sie mit Reglern verknüpft, die festlegen, wie der erwünschte Zustand erreicht werden soll. Stoppbedingungen definieren, wann die Aufgabe als erfüllt gilt oder ob ein Fehlerfall aufgetreten ist. Ergänzt wird die Aufgabenspezifikation durch Skripte, die Hilfsfunktionen erfüllen, wie beispielsweise Greifer anzusteuern oder Sensoren zu tarieren.

3.4.1 Task

Ein Task

$$\mathcal{T} = \langle \boldsymbol{y}, \boldsymbol{y}_d, \boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{A}} \rangle \tag{3.47}$$

spezifiziert Zwangsbedingungen für eine oder mehrere Roboter- oder Feature-Koordinaten und ordnet ihnen jeweils einen Regler $\mathcal{C}$ (`controller`) zu. Zu jedem Wert im Vektor der Regelgrößen $\boldsymbol{y}$ des Tasks muss daher ein entsprechender Wert im Vektor der Sollwerte $\boldsymbol{y}_d$ angegeben werden. Die Regler $\boldsymbol{\mathcal{C}}$ werden über eine Liste $\boldsymbol{\mathcal{A}}$ an Reglerzuweisungen

$$\mathcal{A} = \langle \mathcal{C}_\mathcal{A} \in \boldsymbol{\mathcal{C}}, \boldsymbol{y}_\mathcal{A} \in \boldsymbol{y} \rangle \tag{3.48}$$

den einzelnen Regelgrößen des Tasks zugeteilt. Ein Regler kann dabei beliebig vielen Regelgrößen zugeteilt werden, was bei Bedarf den Einsatz von Mehrgrößenreglern erlaubt.

Regler sind Teil der Aufgabenbeschreibung. Sie können demnach aufgabenspezifisch angegeben und bei Bedarf ersetzt werden. Es liegt in der Verantwortung des Reglers, dass ein gewünschter Sollwert einer Roboter- oder Feature-Koordinate entsprechend den Anforderungen der Aufgabe

erreicht wird und der Einfluss von Störgrößen gering gehalten wird. Die konkrete Auslegung von Reglern ist nicht Teil dieser Arbeit. Es wird davon ausgegangen, dass ein für die jeweilige Anwendung ausreichend guter und stabiler Regler verwendet wird. Von besonderem Interesse ist es dabei, Regler zu finden, die von Eigenschaften der Umgebung und des Roboters weitgehend unabhängig und damit leichter parametrierbar und wiederverwendbar sind (Halt et al. 2019).

3.4.2 Stoppbedingung

Eine Stoppbedingung überprüft im Regeltakt, ob ein Task erfolgreich abgeschlossen wurde oder, im Falle einer Anomalie, abgebrochen werden muss. Soll der Task beendet werden, sendet die Stoppbedingung (`monitor`, engl. auch oft *stop condition* oder *guard*)

$$\mathcal{M} = \langle \boldsymbol{\mathcal{E}} \rangle \tag{3.49}$$

ein Event $\mathcal{E}$, das für die spätere Definition von Transitionen verwendet werden kann. Es wird die Konvention eingeführt, als Voreinstellung für das primäre Event einer Stoppbedingung `succeeded` zu verwenden. Dies vereinfacht die spätere Programmierung und automatische Verknüpfung von Skills in Sequenzen. Die Voreinstellung des Event-Namens lässt sich jedoch ändern, sollte die Stoppbedingung beispielsweise für die Erkennung von Fehlerfällen dienen oder falls zwischen verschiedenen Erfolgskriterien unterschieden werden soll.

Stoppbedingungen kapseln die Erzeugung eines Events in ein parametrierbares Element. Dies ermöglicht die Wiederverwendung der Logik hinter der Event-Erzeugung und damit den modularen Aufbau von Skills. Ein einfaches Beispiel hierzu ist eine Stoppbedingung zur Überwachung von Zeitüberschreitungen, die Skills nach Bedarf hinzugefügt werden kann.

Jede Stoppbedingung kann drei Aktionen definieren:

- Die `on_entry` Aktion wird aufgerufen, wenn die Stoppbedingung gestartet wird. Dies geschieht, wenn der Skill aktiv wird, der die Stoppbedingung enthält. Die Aktion kann verwendet werden, um Variablen zu initialisieren und Stoppbedingungen auf ihren Einsatz vorzubereiten.

- Die `on_update` Aktion wird zyklisch aufgerufen. Hier findet die Überprüfung des oder der betrachteten Kriterien statt. Ist ein Kriterium erfüllt, so sendet die Stoppbedingung das dazugehörige Event. Im Allgemeinen führt dies zu einem Wechsel des aktiven Skills.

- Die `on_exit` Aktion wird schließlich aufgerufen, wenn der übergeordnete Skill beendet wird. Diese Aktion ermöglicht das Deaktivieren der Stoppbedingung.

Stoppbedingungen erhalten beim Aufruf der zyklischen `on_update` Aktion die aktuellen Messwerte der Regelgrößen $\boldsymbol{y}_m$. Sie reagieren beispielsweise auf ein Über- oder Unterschreiten von Schwellwerten der Kraft- oder Positionskoordinaten. Alternativ können sie auch auf externe Ereignisse reagieren, in der Regel indem sie ROS Topics folgen und auswerten. Dazu gehören beispielsweise die Signalausgänge der Robotersteuerung oder Meldungen von Peripheriekomponenten, wie dem Steuergerät eines elektrischen Schraubers, das ein Erreichen des gewünschten Anziehdrehmoments meldet. In Abschnitt 5.1.2 wird eine ausführliche Liste an Stoppbedingungen vorgestellt.

3.4.3 Skript

Skripte $\mathcal{SC}$ kapseln Programmcode zur Erfüllung von Hilfsfunktionen. Sie interagieren mit verschiedenen Hardware- oder Softwarekomponenten. Beispiele umfassen das Ansteuern von Greifern, das Tarieren von Sensoren und das Abfragen von Bauteilpositionen bei einem Bildverarbeitungssystem. In Abschnitt 5.1.1 wird eine ausführliche Liste an Skripten vorgestellt.

Skripte stellen im vorgestellten System die einzige Möglichkeit dar, solche Hilfsfunktionen auszuführen. Obwohl Skills als Zustand in einem Statechart agieren, besitzen sie selbst keine vom Nutzer implementierbare `on_entry`, `on_update` oder `on_exit` Aktion und damit keine Möglichkeit, anwendungsspezifischen Code auszuführen. Solcher Code muss zwingend in Skripten implementiert werden, die Skills hinzugefügt werden können und wie Stoppbedingungen die genannten drei Aktionen besitzen. Eine Ausnahme von dieser Regel existiert nicht. Dies ist ein wichtiger Schritt weg von der Programmierung von *Glue Code* hin zur systematischen Komposition von Bausteinen und ermöglicht so die Modularisierung und damit die Wiederverwendung von Softwarebausteinen.

Viele Skripte laufen synchron und enthalten instantan auszuführende Befehle. Ein Beispiel ist das Verwalten eines temporären Koordinatensystems, das über die `on_entry` Aktion erstellt und über die `on_exit` Aktion gelöscht wird. Da synchrone Skripte den Programmfluss blockieren, müssen sie in einer definierten und kurzen Zeit ausgeführt werden. Skripte können aber auch asynchron aufgerufen werden, wenn sie Befehle ausführen sollen, die für eine undefinierte oder längere Zeit blockieren können. Dazu gehören beispielsweise der Aufruf eines ROS Services zum Schließen eines Greifers oder die Kommunikation mit einem Bildverarbeitungssystem.

3.5 Koordination

In den vorangegangenen Abschnitten wurde eine Vielzahl an Bausteinen vorgestellt. Dieser Abschnitt befasst sich damit, wie die einzelnen Bausteine zu Skills zusammengefügt werden. Darüber hinaus wird betrachtet, wie die Ausführung mehrerer Skills koordiniert werden kann, sei es gleichzeitig, in einer einfachen Sequenz aufeinanderfolgend oder mithilfe einer Zustandsmaschine. Der vorgestellte Koordinationsmechanismus legt die Basis für die in Kapitel 4 vorgestellte Komposition von Skills und die Modellierung der Logik kompletter Anwendungen. Dem Koordinationsmechanismus liegen Statecharts zugrunde, die im Folgenden kurz eingeführt werden.

Statecharts sind eine Erweiterung klassischer Zustandsmaschinen (engl. *finite state machine*, FSM). Eine Zustandsmaschine modelliert das komplexe Verhalten eines Systems über eine endliche Anzahl an Zuständen, die das System annehmen kann. Dabei ist stets nur ein Zustand aktiv. Die einzelnen Zustände werden mithilfe von Zustandsübergängen, oft Transitionen genannt, verknüpft. Transitionen werden durch Ereignisse (engl. *events*) ausgelöst und führen einen Zustandswechsel durch. Klassische Zustandsmaschinen stoßen allerdings sehr schnell an ihre Grenzen, da die Anzahl an Zuständen und Transitionen mitunter exponentiell ansteigen kann, um jede mögliche Kombination von Eigenschaften und alle Zustandsübergänge zu modellieren.

Die von Harel (1987) eingeführten Erweiterungen adressieren dieses Problem. Hierarchische Zustandsmaschinen führen durch das Zusammenfassen von Zuständen in Unterzustandsmaschinen einen Abstraktionsmechanismus ein. Unterzustandsmaschinen werden als Ganzes eingebunden, wodurch sich die Anzahl an nötigen Transitionen stark verringert. Zudem verbergen sie die Details ihrer Implementierung und erhöhen so die Handhabbarkeit komplexer Systeme. Parallele Zustände erlauben es zudem, dass sich das Statechart gleichzeitig in mehreren Zuständen oder Unterzustandsmaschinen befindet. Voneinander unabhängige Eigenschaften des Systems lassen sich so auch getrennt voneinander modellieren. Dadurch kann die Anzahl an benötigten Zuständen verringert werden, da nicht mehr jede mögliche Kombination von Eigenschaften als eigener Zustand modelliert werden muss. Eine Variante der Harel Statecharts findet als sogenanntes Zustandsdiagramm (engl. *state diagram*) in der UML weite Verbreitung zur Modellierung von Softwaresystemen.

Den Zuständen eines Statecharts lassen sich Aktionen zuweisen, die ausgeführt werden, wenn ein Zustand aktiviert oder deaktiviert wird, beziehungsweise während er aktiv ist. Letztere Aktion wird dabei oft in einem fest vorgegebenen Takt aufgerufen. Aktionen enthalten meist eine Abfolge von Anweisungen, mit denen die Konfiguration des Programms geändert wird.

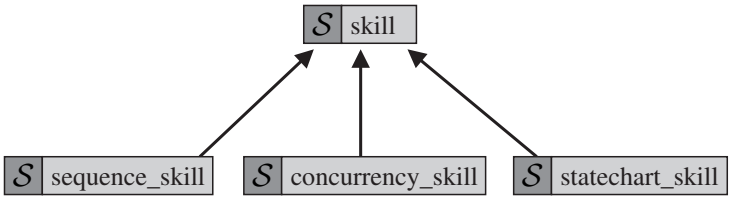


Abbildung 3.11: Arten von Skills und Skill-Kompositionen

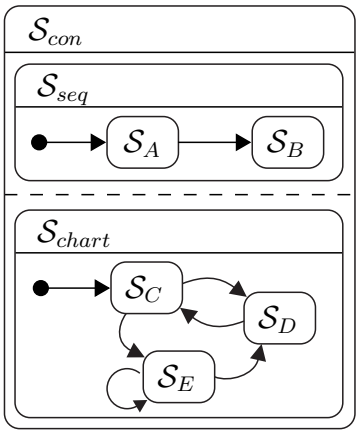


Abbildung 3.12: Beispiel eines Concurrency-Skills $\mathcal{S}_{con}$, der einen Sequence-Skill $\mathcal{S}_{seq}$ und einen Statechart-Skill $\mathcal{S}_{chart}$ enthält, sowie mit den einfachen Skills $\mathcal{S}_A$ bis $\mathcal{S}_E$

Skills

Skills teilen eine komplexe Roboteraufgabe in handhabbare Teilprozesse auf. Zur Koordination der einzelnen Teilprozesse agieren die Skills im vorliegenden Skill-Modell als Zustände in Statecharts. Da Statecharts hierarchisch sind, können Skills wiederum Sub-Skills enthalten. Insgesamt wird zwischen vier Arten von Skills unterschieden, wie in Abbildung 3.11 dargestellt. Abbildung 3.12 zeigt dazu ein Beispiel, das die verschiedenen Skill-Arten enthält. Ein Concurrency-Skill führt seine Sub-Skills gleichzeitig aus, ein Sequence-Skill in einer festen Reihenfolge. Bei einem Statechart-Skill können die Sub-Skills in einem Statechart angeordnet werden. Einfache Skills enthalten keine Sub-Skills. Die Skills $\boldsymbol{\mathcal{S}}_A$ und $\boldsymbol{\mathcal{S}}_C$ im Beispiel sind die jeweiligen Startzustände ihrer übergeordneten Skills. Sie werden gestartet, wenn ihr übergeordneter Skill gestartet wird. Verdeutlicht wird dies durch einen Pfeil, der einem Punkt entspringt.

Wie in Abschnitt 3.4.3 beschrieben, untersagt es das Konzept des vorgeschlagenen Skill-Modells, anwendungsspezifischen Code einem Skill direkt als Aktion hinzuzufügen. Skills dienen lediglich als Container für andere Elemente. Soll anwendungsspezifischer Code einem Skill hinzugefügt werden, so geschieht dies als Skript oder als Stoppbedingung. Dieses Prinzip ist verwandt mit dem von Klotzbücher et al. (2012a) vorgestellten *Configurator-Coordinator Pattern*. Hierbei wird das Reagieren auf Ereignisse und das Auslösen von Zustandswechseln von der letztlichen Ausführung der Aktionen getrennt. Die Koordination wird vom sogenannten *Pure Coordinator*

übernommen, der das Ausführen der Aktionen an einen *Configurator* delegiert. Der Vorteil eines solchen Systems ist die strikte Trennung des Koordinationsmechanismus, in Form der Statecharts, von den anwendungsspezifischen Berechnungen, hier in Form der kinematischen Elemente, Stoppbedingungen und Skripte. Daraus ergibt sich eine größere Wiederverwendbarkeit der einzelnen Bestandteile.

Jeder Skill

$$\mathcal{S} = \langle \mathcal{N}, \boldsymbol{\mathcal{KE}}, \boldsymbol{\mathcal{T}}, \boldsymbol{\mathcal{SC}}, \boldsymbol{\mathcal{M}}, \boldsymbol{\mathcal{TR}}, \boldsymbol{\mathcal{S}}_{sub} \rangle \tag{3.50}$$

besitzt einen Namen $\mathcal{N}$, der eindeutig unter den Sub-Skills seines übergeordneten Skills ist, sowie mehrere Listen von Elementen. Wie zuvor beschrieben ist ein Skill ein Container für die bisher beschriebenen Elemente. Er kann eine beliebige Menge dieser Elemente besitzen:

- Die Liste an kinematischen Elementen $\boldsymbol{\mathcal{KE}} = \langle \boldsymbol{\mathcal{D}}, \boldsymbol{\mathcal{L}}, \boldsymbol{\mathcal{CH}}, \boldsymbol{\mathcal{LP}} \rangle$ enthält die Datenquellen, Kettenglieder, Ketten und Schleifen, mit denen Roboter- und Feature-Koordinaten beschrieben werden.
- Zur Aufgabenspezifikation besitzt ein Skill eine Liste an Tasks $\boldsymbol{\mathcal{T}}$, die für Roboter- und Feature-Koordinaten Zwangsbedingungen und Regler definieren, die Skripte $\boldsymbol{\mathcal{SC}}$, die Hilfsfunktionen wie das Ansteuern von Greifern erfüllen und die Stoppbedingungen $\boldsymbol{\mathcal{M}}$, die festlegen, wann ein Skill abgeschlossen oder abgebrochen werden soll.
- Eine Liste an Transitionen $\boldsymbol{\mathcal{TR}}$ mit $\mathcal{TR} = \langle \mathcal{E}, \mathcal{N} \rangle$ gibt an, welcher Skill $\mathcal{N}$ auf ein Event $\mathcal{E}$ einer der Stoppbedingungen folgen soll.

Skills definieren die Lebensdauer dieser Elemente. Nur solange ein Skill aktiv ist, sind auch seine Unterelemente aktiv.

Über die genannten Elemente hinaus kann ein Skill auch eine Reihe an Sub-Skills $\boldsymbol{\mathcal{S}}_{sub}$ besitzen. Abhängig davon, ob der Skill ein Statechart-Skill, Sequence-Skill oder Concurrency-Skill ist, werden seine Sub-Skills dabei unterschiedlich behandelt.

Statechart-Skill

Ein Statechart-Skill enthält Sub-Skills, die über Transitionen zu einem Statechart verbunden sind. Die hierarchische Verschachtelung von Skills erlaubt es, selbst komplexe Programmlogik auf einfache und übersichtliche Art zu modellieren. Abbildung 3.13 zeigt das vereinfachte Beispiel einer Schraubapplikation. Im Startzustand wird der Schrauber aufgenommen. Einmal aufgenommen lassen sich beliebig viele Schraubprozesse durchführen. Dies wird anhand der Selbsttransition von $\mathcal{S}_{\texttt{screw}}$ deutlich. Das Statechart drückt zudem aus, dass der Schrauber erst abgelegt werden muss, bevor er erneut aufgenommen werden kann.

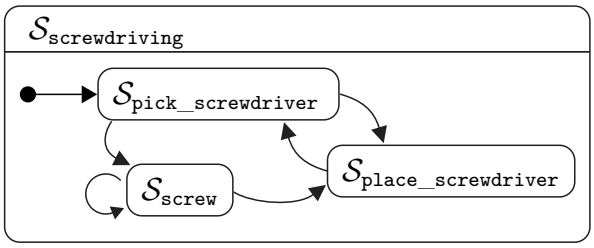


Abbildung 3.13: Statechart einer Schraubapplikation

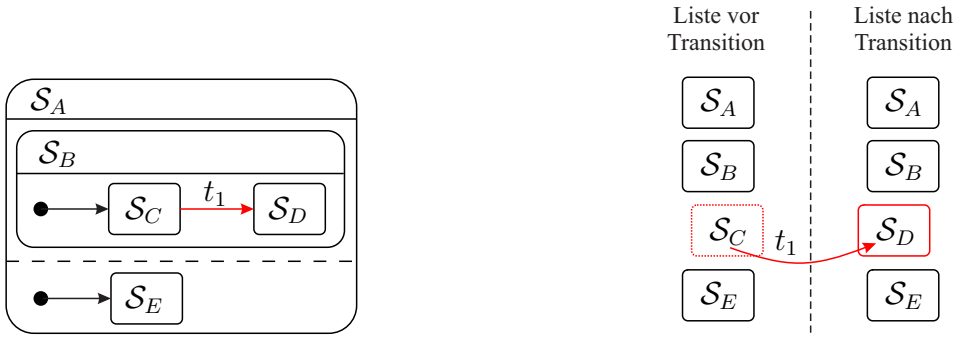
Sequence-Skill

Sequence-Skills sind eine Sonderform der Statechart-Skills, bei der vereinfachende Annahmen getroffen werden, um das Modellieren von Anwendungen zu beschleunigen. Die in der Liste der Sub-Skills aufeinanderfolgenden Skills werden in der gegebenen Reihenfolge ausgeführt. Unter Verwendung der in Abschnitt 3.4.2 eingeführten Konvention, `succeeded` als das primäre Event einer Stoppbedingung zu verwenden, erstellen Sequenzen für dieses Event automatisch Transitionen zwischen aufeinanderfolgenden Skills. Zusätzliche Transitionen, beispielsweise bei Fehlerfällen, lassen sich bei Bedarf hinzufügen. Sollte dieses vereinfachte Modell nicht ausreichen, muss ein Statechart-Skill verwendet werden, bei dem alle Transitionen manuell vorgegeben werden.

Concurrency-Skill

Concurrency-Skills führen alle Sub-Skills gleichzeitig aus. Dargestellt ist dies in Abbildung 3.12 durch eine gestrichelte Linie, mit der die beiden Sub-Skills $\mathcal{S}_{seq}$ und $\mathcal{S}_{chart}$ des Concurrency-Skills $\mathcal{S}_{con}$ getrennt werden. Aus der Reihenfolge der Sub-Skills in der Liste $\boldsymbol{\mathcal{S}}_{sub}$ ergeben sich die Prioritätslevel der einzelnen Skills. Kommt es zu Konflikten zwischen den Sub-Skills, so werden Skills mit höherer Priorität bevorzugt. Die Tasks von Skills mit niedrigerer Priorität werden dazu in den Nullraum der Tasks höher priorisierter Skills projiziert. Dies bedeutet, dass sie mitunter nicht komplett erfüllt werden.

Über Statechart-, Sequence- und Concurrency-Skills lassen sich beliebig verschachtelte Anwendungen erstellen. Zu jedem Zeitpunkt ist jedoch nur eine endliche Anzahl an Skills aktiv. Die Aktivierung und Deaktivierung von Skills und ihren Unterelementen ist Aufgabe des Ausführungsmechanismus.



(a) Beispiel eines Statecharts mit dem Concurrency-Skill $\mathcal{S}_A$, Sequence-Skill $\mathcal{S}_B$ sowie den einfachen Skills $\mathcal{S}_C$, $\mathcal{S}_D$ und $\mathcal{S}_E$

(b) Priorisierte Liste der aktiven Skills vor und nach der Transition t_1 von Skill $\mathcal{S}_C$ nach Skill $\mathcal{S}_D$

Abbildung 3.14: Priorisierte Liste aktiver Skills eines Statecharts vor und nach einer Transition

3.6 Ausführung des Modells

Die Ausführung des Modells teilt sich in drei Bereiche auf. Die Scene (Abschnitt 3.6.1) verwaltet die aktiven Skills und ihre Unterelemente. Zudem koordiniert sie den Update-Zyklus und sorgt in jedem Takt dafür, dass alle aktiven Elemente aktualisiert werden. Der iTaSC-Algorithmus (Abschnitt 3.6.2) berechnet basierend auf den bereitgestellten Werten der einzelnen Elemente in jedem Takt die Sollgeschwindigkeiten $\dot{\boldsymbol{y}}_d$ und die Matrix $\boldsymbol{A}$. Diese werden anschließend an den Solver (Abschnitt 3.6.3) gereicht, der $\boldsymbol{A}$ invertiert und die gesuchten Gelenkgeschwindigkeiten $\dot{\boldsymbol{q}}_d$ berechnet.

3.6.1 Scene

Bei einem Zustandswechsel wird die Liste aktiver Elemente von der Scene[7] aktualisiert. Wird ein Skill entfernt, so werden auch die ihm untergeordneten Elemente, einschließlich seiner Sub-Skills und deren Elemente, mit deaktiviert. Ebenso werden beim Hinzufügen eines Skills auch alle zugehörigen Elemente des Skills aktiviert. Abbildung 3.14a zeigt beispielhaft ein Statechart mit einem Concurrency-Skill und einem darin enthaltenen Sequence-Skill, bei dem die Transition t_1 ausgeführt werden soll. Die Skills $\mathcal{S}_A$, $\mathcal{S}_B$ und $\mathcal{S}_E$ bleiben weiterhin aktiv, Skill $\mathcal{S}_C$ wird beendet und dafür Skill $\mathcal{S}_D$ gestartet. Bei zu startenden Skripten und Stoppbedingungen wird von der Scene die `on_entry` Aktion aufgerufen. So kann beispielsweise eine Stoppbedingung, die nach einem gewissen Zeitintervall auslöst, ihre Startzeit setzen. Analog dazu wird bei zu

[7] Der Begriff *Scene* wird von Smits (2010) übernommen. Er leitet sich von den sogenannten Szenengraphen ab, die in der Computergrafik die darzustellenden Objekte verwalten.

beendenden Skripten und Stoppbedingungen die `on_exit` Aktion aufgerufen. Dies ermöglicht es, den Zustand eines Elements zurückzusetzen.

Darüber hinaus ist die Scene dafür zuständig, in jedem Takt die Aktualisierung aller aktiven Elemente zu veranlassen. Dafür stellen alle Elemente eine `on_update` Aktion zur Verfügung. Kinematische Elemente berechnen hier aktuelle Werte, Stoppbedingungen überprüfen, ob sie ein Event auslösen müssen. Skripte erfüllen unterschiedliche Aufgaben. So ändert ein Skript zur Sollwertgenerierung beispielsweise die Sollwerte eines Tasks, ein Skript zur Datenaufzeichnung speichert die aktuellen Messwerte.

3.6.2 iTaSC

Ein Teil der Vektoren und Matrizen, die für die Lösung des in Abschnitt 2.4 vorgestellten iTaSC-Algorithmus benötigt werden, lässt sich bereits nach einem Zustandswechsel bestimmen. Die verbleibenden Größen werden zyklisch aktualisiert.

Berechnungen bei Zustandswechseln

Bei einem Zustandswechsel und der damit einhergehenden Änderung der Liste aktiver Skills werden die neuen Gelenk- und Feature-Koordinaten sowie die daraus ausgewählten Regelgrößen bestimmt. Aus diesen Informationen leiten sich wie folgt die Selektionsmatrizen $\boldsymbol{C}_q$ und $\boldsymbol{C}_f$ ab.

Aus der Liste aktiver Skills ergibt sich zunächst die Liste aller aktiven kinematischen Elemente. Mit einem Durchlaufen aller aktiven Kettenglieder lassen sich die n_q Gelenkkoordinaten in $\boldsymbol{q}$ und n_f Feature-Koordinaten in $\boldsymbol{\chi}_f$ erfassen. Ebenso können bei einem Durchlaufen aller aktiven Tasks die n_y Regelgrößen in $\boldsymbol{y}$, die n_y Sollwerte in $\boldsymbol{y}_d$ und die dazugehörigen Regler $\boldsymbol{\mathcal{C}}$ bestimmt werden. Hierbei ist die Reihenfolge der Skills in der Liste entscheidend, da sich daraus die Priorität ihrer Tasks ergibt. In Abbildung 3.14b ist die priorisierte Liste aktiver Skills vor und nach der Transition t_1 aufgeführt. Der äußere Concurrency-Skill $\mathcal{S}_A$ besitzt dabei die höchste Priorität. Der in $\mathcal{S}_A$ enthaltene Skill $\mathcal{S}_B$ und sein jeweils aktiver Sub-Skill $\mathcal{S}_C$ oder $\mathcal{S}_D$ haben stets eine höhere Priorität als $\mathcal{S}_E$.

Aus den genannten Größen lassen sich die $n_y \times n_q$-Matrix $\boldsymbol{C}_q$ und die $n_y \times n_f$-Matrix $\boldsymbol{C}_f$ aufstellen. Diese sind einfache Selektionsmatrizen und können nach Gleichung (2.100)

$$\dot{\boldsymbol{y}} = \boldsymbol{C}_q \dot{\boldsymbol{q}} + \boldsymbol{C}_f \dot{\boldsymbol{\chi}}_f \tag{3.51}$$

mit Einsen und Nullen entsprechend der oben bestimmten Koordinaten $\boldsymbol{q}$, $\boldsymbol{\chi}_f$ und $\boldsymbol{y}$ gefüllt werden.

Ein einfaches zweidimensionales Beispiel sei mit einem vierachsigen Roboter und den Gelenkwinkeln $\boldsymbol{q} = \begin{bmatrix} q_1 & q_2 & q_3 & q_4 \end{bmatrix}^\mathsf{T}$ sowie den Feature-Koordinaten $\boldsymbol{\chi}_f = \begin{bmatrix} x & y & \alpha \end{bmatrix}^\mathsf{T}$ gegeben. Werden die geregelten Größen zu $\boldsymbol{y} = \begin{bmatrix} x & y & q_3 \end{bmatrix}^\mathsf{T}$ gewählt, so ergeben sich die Selektionsmatrizen

$$\boldsymbol{C}_q = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \text{ und } \boldsymbol{C}_f = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \tag{3.52}$$

Zyklische Berechnungen

In jedem iTaSC-Zyklus wird die Gleichung (2.105)

$$\boldsymbol{A}\dot{\boldsymbol{q}}_d = \dot{\boldsymbol{y}}_d \tag{3.53}$$

gelöst. Dafür muss die Matrix

$$\boldsymbol{A} = \boldsymbol{C}_q - \boldsymbol{C}_f \boldsymbol{J}_f^{-1} \boldsymbol{J}_q \tag{3.54}$$

bestimmt werden. Die Matrizen $\boldsymbol{J}_q$ und $\boldsymbol{J}_f$ setzen sich aus den Jacobi-Matrizen $\boldsymbol{J}_{q,\mathcal{LP}1}$ bis $\boldsymbol{J}_{q,\mathcal{LP}n_s}$ beziehungsweise $\boldsymbol{J}_{f,\mathcal{LP}1}$ bis $\boldsymbol{J}_{f,\mathcal{LP}n_s}$ der n_s aktiven kinematischen Schleifen zusammen. Sie umfassen damit alle n_q aktiven Gelenkkoordinaten sowie alle n_f aktiven Feature-Koordinaten. Es ergibt sich für $\boldsymbol{J}_q$ die Dimension $n_f \times n_q$ und für $\boldsymbol{J}_f$ die Dimension $n_f \times n_f$. Dabei ist $n_f = 6n_s$, da jede Schleife genau sechs Feature-Koordinaten enthält und somit jeweils sechs Reihen der beiden Matrizen füllt. Für $\boldsymbol{J}_f$ ergibt sich die Blockdiagonalmatrix

$$\boldsymbol{J}_f = \begin{bmatrix} \boldsymbol{J}_{f,\mathcal{LP}1} & & & \boldsymbol{0} \\ & \boldsymbol{J}_{f,\mathcal{LP}i} & & \\ & & \ddots & \\ \boldsymbol{0} & & & \boldsymbol{J}_{f,\mathcal{LP}n_s} \end{bmatrix}. \tag{3.55}$$

Für die Jacobi-Matrix $\boldsymbol{J}_q$ ist zu beachten, dass die Anzahl und Reihenfolge der Gelenkkoordinaten $\boldsymbol{q}_{\mathcal{LP}i}$ einer Schleife und der Gelenkkoordinaten $\boldsymbol{q}$ aller aktiven Schleifen nicht übereinstimmen müssen (beispielsweise, wenn eine Schleife nicht alle Robotergelenke umfasst). Entsprechend müssen die Spalten der jeweiligen Matrix $\boldsymbol{J}_{q,\mathcal{LP}i}$ in die zugehörigen Spalten der Matrix $\boldsymbol{J}_q$ eingeordnet werden.

Die einzelnen Jacobi-Matrizen werden zyklisch von den aktiven Schleifen geliefert. Die Selektionsmatrizen $\boldsymbol{C}_q$ und $\boldsymbol{C}_f$ wurden wie oben beschrieben bereits beim letzten Zustandswechsel aktualisiert. Die Matrix $\boldsymbol{A}$ wird zusammen mit den Sollwerten $\dot{\boldsymbol{y}}_d$ zur Berechnung der gewünschten Gelenkgeschwindigkeiten $\dot{\boldsymbol{q}}_d$ an den Solver gegeben. Die Sollwerte $\dot{\boldsymbol{y}}_d$ werden dafür von den Reglern auf Basis der aktuellen Messwerte $\boldsymbol{y}_m$ erzeugt. Anschließend werden die vom Solver berechneten Gelenkgeschwindigkeiten $\dot{\boldsymbol{q}}_d$ über den Robotertreiber zur Ausführung an die Robotersteuerung weitergereicht.

3.6.3 Solver

Die Aufgabe des Solvers ist die Inversion von Gleichung (2.105) zur Bestimmung der gewünschten Gelenkgeschwindigkeiten $\dot{\boldsymbol{q}}_d$. Die $n_y \times n_q$-Matrix $\boldsymbol{A}$ ist im Allgemeinen nicht quadratisch und die einzelnen Aufgabenspezifikationen können in Konflikt treten. Wie von Smits (2010) und Vanthienen (2015) gezeigt, kann die in Abschnitt 2.1.3 vorgestellte Task Priority Strategy zur Lösung der Gleichung herangezogen werden. Dazu werden die $\boldsymbol{A}$-Matrix und die Sollwerte in $\dot{\boldsymbol{y}}_d$ entsprechend ihrer Priorität gegliedert. Sind n_p Prioritätsstufen vorhanden, so erhält man

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_1^\mathsf{T} & \boldsymbol{A}_2^\mathsf{T} & \dots & \boldsymbol{A}_i^\mathsf{T} & \dots & \boldsymbol{A}_{n_p}^\mathsf{T} \end{bmatrix}^\mathsf{T} \tag{3.56}$$

und

$$\dot{\boldsymbol{y}}_d = \begin{bmatrix} \dot{\boldsymbol{y}}_{d,1}^\mathsf{T} & \dot{\boldsymbol{y}}_{d,2}^\mathsf{T} & \dots & \dot{\boldsymbol{y}}_{d,i}^\mathsf{T} & \dots & \dot{\boldsymbol{y}}_{d,n_p}^\mathsf{T} \end{bmatrix}^\mathsf{T} . \tag{3.57}$$

Mit diesen Größen kann die rekursiv formulierte Task Priority Strategy

$$\dot{\boldsymbol{q}}_{d,1} = \boldsymbol{A}_1^{\#_1} \dot{\boldsymbol{y}}_{d,1} , \tag{3.58}$$

$$\dot{\boldsymbol{q}}_{d,i} = \dot{\boldsymbol{q}}_{d,i-1} + \boldsymbol{P}_{i-1} (\boldsymbol{A}_i \boldsymbol{P}_{i-1})^{\#_i} (\dot{\boldsymbol{y}}_{d,i} - \boldsymbol{A}_i \dot{\boldsymbol{q}}_{d,i-1}) \tag{3.59}$$

ausgeführt werden. Für jeden Iterationsschritt werden dafür die partiellen $\boldsymbol{A}$-Matrizen

$$\boldsymbol{A}_{1,i} = \begin{bmatrix} \boldsymbol{A}_1^\mathsf{T} & \boldsymbol{A}_2^\mathsf{T} & \dots & \boldsymbol{A}_i^\mathsf{T} \end{bmatrix}^\mathsf{T} \tag{3.60}$$

akkumuliert. Ebenso benötigt wird die Projektionsmatrix

$$\boldsymbol{P}_i = \boldsymbol{I} - \boldsymbol{A}_{1,i}^{\#_{1,i}} \boldsymbol{A}_{1,i} . \tag{3.61}$$

Der gesuchte Vektor der Gelenkgeschwindigkeiten $\dot{\boldsymbol{q}}_d = \dot{\boldsymbol{q}}_{n_p}$ ergibt sich schließlich nach dem letzten Iterationsschritt n_p.

Zusammenfassung

Prinzipiell lassen sich mit den in diesem Kapitel vorgestellten Bausteinen komplette Roboteranwendungen programmieren. Die Bausteine werden zu Skills zusammengefasst, die sich wiederum in einem hierarchischen Statechart zu komplexen Skills verschachteln lassen. Da jeder Baustein dabei nur einen Teilaspekt einer Roboteranwendung beschreibt, lassen sich die Bausteine leicht wiederverwenden. Noch offen ist an dieser Stelle jedoch die Frage, wie diese Wiederverwendung effizient modelliert werden kann. Im nachfolgenden Kapitel 4 wird daher ein Skill-Modell vorgestellt, das die effiziente Komposition und Vererbung von Bausteinen und ihren Parametern erlaubt.

4 Modell zur Komposition und Spezialisierung von Bausteinen

Im vorangegangenen Kapitel 3 wurden die einzelnen Bausteine des Systems vorgestellt und es wurde darauf eingegangen, wie diese zu Skills zusammengesetzt werden können. Das Ergebnis ist ein hierarchisches Statechart aus Skills mit Unterbausteinen wie Stoppbedingungen, Skripte, Regler und kinematische Elemente. Zu jedem Zeitpunkt der Ausführung sind ein oder mehrere Skills aktiv, die in eine priorisierte Liste überführt und zur Parametrierung des iTaSC-Algorithmus herangezogen werden.

Um direkt mit einem solchen System Montageanwendungen zu programmieren, sind weitreichende Programmierkenntnisse erforderlich. Zudem ist das manuelle Erzeugen und Verknüpfen der einzelnen Bausteine zeitaufwendig, ermüdend und fehleranfällig. Im Folgenden wird daher eine Abstraktionsschicht eingeführt, die es erlaubt, die Parametrierung, Komposition und Wiederverwendung der genannten Bausteine zu modellieren. Das entstandene Skill-Modell wird durch eine DSL vervollständigt, mit der diese Modellierung auf einfache und konsistente Weise erfolgen kann. Abbildung 4.1 stellt das Gesamtsystem schematisch dar.

Dieses Kapitel beschreibt den Entwurf des *pitasc* Skill-Modells zur Komposition und Spezialisierung von Bausteinen und betrachtet dabei insbesondere die folgenden Aspekte:

- *Explizite Modellierung der Wiederverwendung und Erweiterung einzelner Bausteine mithilfe* ***prototypbasierter Vererbung*** *und* ***Komposition*** *(Abschnitt 4.1)*
- *Gestaltung einer einfachen* ***domänenspezifischen Sprache****, mit der das Skill-Modell effizient und ohne Programmierkenntnisse genutzt werden kann (Abschnitt 4.2)*
- ***Inkrementelle Erstellung von Skills*** *mithilfe der domänenspezifischen Sprache (Abschnitt 4.3)*

Auch andere der in Abschnitt 1.2 vorgestellten Skill-Modelle nutzen domänenspezifische Sprachen, mit dem Ziel eine einfache Programmierung von Roboteraufgaben zu ermöglichen. Sie beschreiben jedoch nur einzelne Skills und betrachten nicht, wie eine umfangreiche Skill-Bibliothek

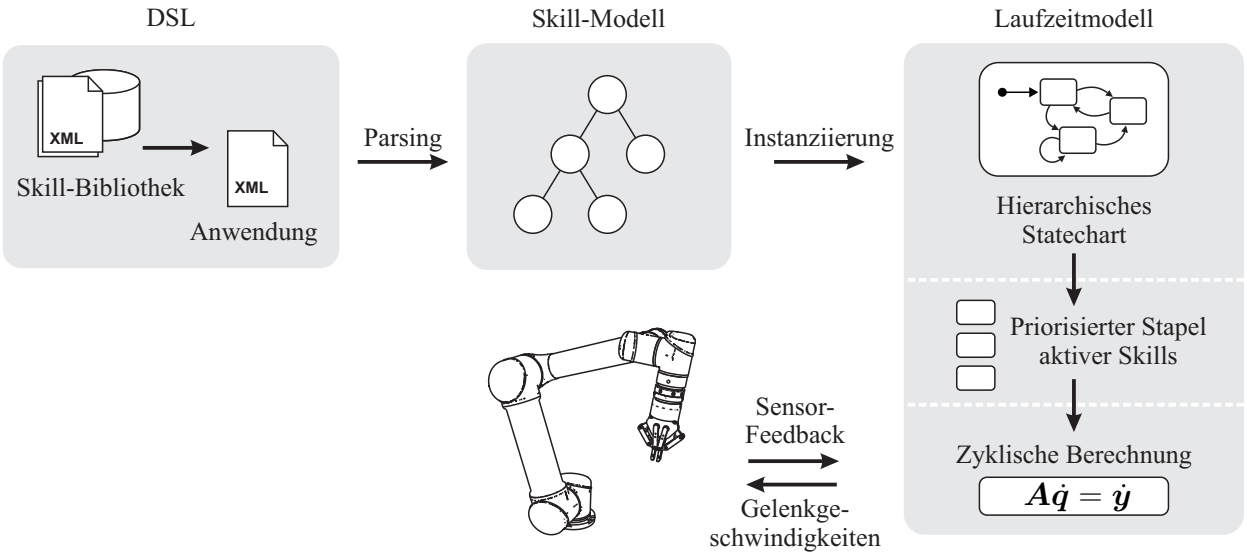


Abbildung 4.1: Konzeptuelle Darstellung des Gesamtsystems

effizient erstellt werden kann. Das hier vorgeschlagene Skill-Modell erlaubt es hingegen, Skills durch systematische Komposition und Vererbung effizient mit einem hohen Maß an Wiederverwendung zu erstellen. Dennoch sei hervorgehoben, dass der hier beschriebene Ansatz nicht zwangsläufig andere Skill-Modelle ersetzen muss. Vielmehr ließe sich das prototypbasierte Modell auch zur Parametrisierung anderer Skill-Modelle heranziehen. Diese Übertragbarkeit wird hier jedoch nicht weiter untersucht.

4.1 Skill-Modell

Der Entwurf des Skill-Modells beruht auf den drei in Abbildung 4.2 dargestellten Eckpfeilern Abstraktion, Komposition und Vererbung. Zusammen ermöglichen sie die Parametrisierung von Skills und deren Unterbausteinen, die Komposition von komplexen Parametern sowie deren Wiederverwendung auf Basis prototypbasierter Vererbung. Die wichtigsten Eigenschaften der drei Eckpfeiler werden zunächst kurz umrissen und im Folgenden dann detailliert betrachtet.

Abstraktion (Parameter)

Das primäre Ziel des Skill-Modells ist es, die Programmierung von Skills und Anwendungen durch einen modellbasierten Ansatz auf einem höheren Abstraktionsgrad zu ermöglichen. Das Skill-Modell beschreibt daher eine Abstraktionsschicht, die in der Systemarchitektur oberhalb der in Kapitel 3 dargestellten Elemente angesiedelt ist. Konzeptionell wird dabei seitens des Modells nicht zwischen Skills und anderen Elementen unterschieden. In Anlehnung an das

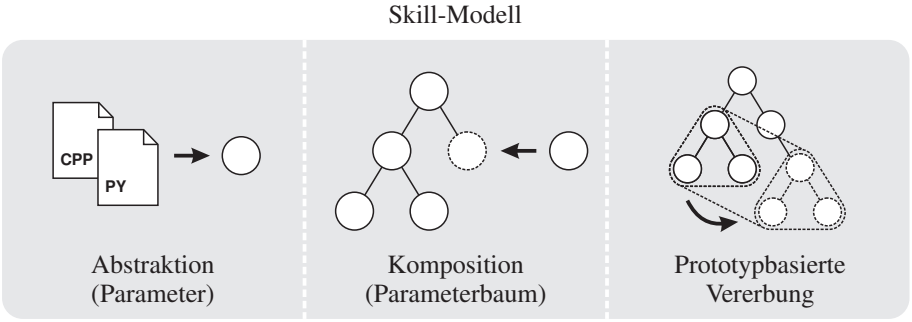


Abbildung 4.2: Ansatz des *pitasc* Skill-Modells

in der objektorientierten Programmierung verbreitete Prinzip «alles ist ein Objekt» wird im vorliegenden Modell jedes Element durch einen Parameter beschrieben. Das *pitasc* Skill-Modell folgt dabei der Maxime:

Alles ist ein Parameter.
Parameter sind wiederverwendbar.
Alles ist wiederverwendbar.

Dies gilt vom einfachen String oder Zahlenwert bis hin zu Skripten, Skills und ganzen Anwendungen, die jeweils von Parametern repräsentiert werden und zu ihrer Parametrierung Unterparameter enthalten können. Gleichwohl zeigen sich die Vorteile des Modells in besonderem Maße bei der Erstellung von Skills.

Ein wichtiger Aspekt ist darüber hinaus die Wahl eines deklarativen Ansatzes gegenüber einer klassischen Ablaufsteuerung. Mit iTaSC wird eine Aufgabenspezifikation verwendet, die dem gewünschten Roboterverhalten entsprechend parametriert werden muss. Dies geschieht hier jedoch nicht durch das Programmieren von imperativem Programmcode in einer höheren Programmiersprache wie C++ oder Python. Stattdessen wird durch die Eingabe und Strukturierung von Parametern in einer einfachen domänenspezifischen Sprache ein Modell erstellt, das die Anwendung deklarativ beschreibt. Zur Ausführung des Modells wird mittels dieser Beschreibung der eigentliche Programmcode parametriert und aufgerufen. Es erfolgt demnach eine Trennung des Ausführungsmechanismus, der für jede Anwendung derselbe ist, und dessen anwendungsspezifischer Parametrierung.

Komposition (Parameterbaum)

Die (Wieder-)Verwendung von Parametern basiert auf zwei grundsätzlichen Prinzipien, der Komposition und der Vererbung von Parametern. Parameter enthalten Unterparameter, wo-

durch eine Teil-Ganzes-Hierarchie in Form einer Baumstruktur entsteht. So enthält ein Roboterparameter beispielsweise eine Reihe an Gelenkkoordinaten oder eine Skill-Sequenz eine Reihe an Sub-Skills (die wiederum weitere Sub-Skills enthalten können). Komposition ermöglicht die Kombination einfacher Elemente zu immer komplexeren Komponenten, die als Ganzes zusammengefasst einfacher verwendbar sind, ohne ihre internen Details kennen zu müssen.

Prototypbasierte Vererbung

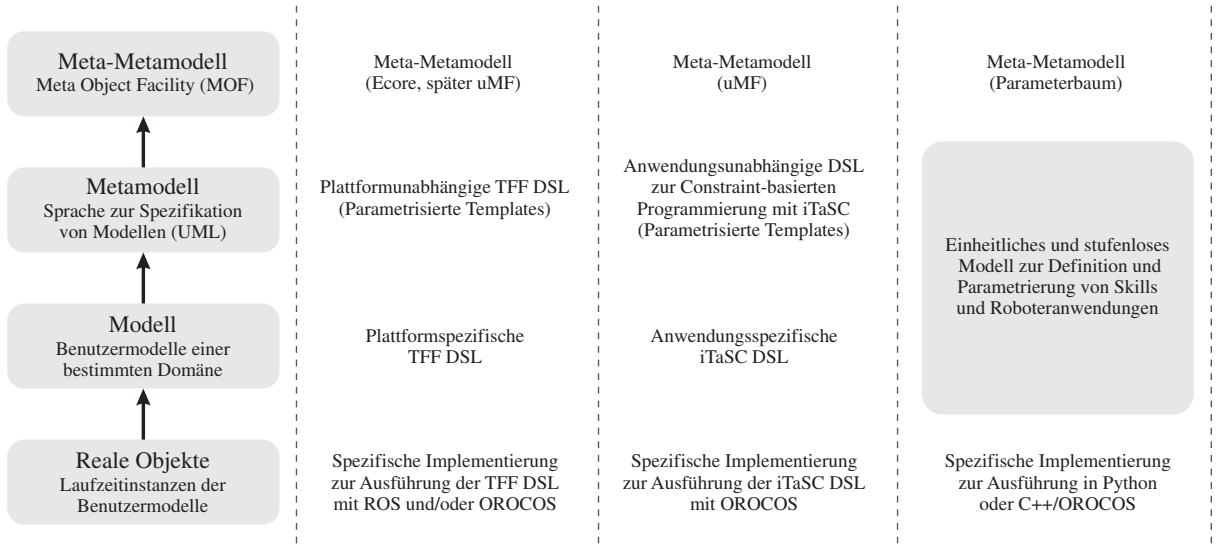
Um nicht jeden Parameter von Grund auf neu parametrieren zu müssen, ist es von Interesse, bestehende Parameter (insbesondere Skills) wiederzuverwenden und zu ergänzen. So ist es beispielsweise wünschenswert, einen Skill für eine kraftüberwachte lineare Relativbewegung auf dem Skill einer linearen Relativbewegung aufzubauen, der wiederum auf dem Skill einer einfachen linearen Bewegung aufbaut. Das zweite Prinzip der Wiederverwendung von Parametern befasst sich entsprechend mit dem in der Softwareentwicklung verbreiteten Konzept der Vererbung.

Aus Sicht des Parameterbaums sollen demnach einzelne Parameter oder ganze Zweige des Baums kopiert, angepasst und gegebenenfalls erweitert werden. Das vorgestellte Skill-Modell beruht dabei auf einer prototypbasierten Vererbung, die eine einfache Vererbung von Parametern mit geringem Aufwand erlaubt. Die Möglichkeit zu einer hohen Wiederverwendbarkeit von Parametern ist dadurch bereits tief im Modell verankert.

4.1.1 Modellbasierter Ansatz und Abstraktion

Die beiden in Abschnitt 2.3 vorgestellten modellbasierten Ansätze von Klotzbücher et al. (2011) und Vanthienen et al. (2013) beschreiben eine Adaption des MDA 4-Ebenenmodells für die Spezifikation von Roboteranwendungen. Dabei nehmen sie eine Trennung zwischen dem anwendungsspezifischen Teil und dem davon unabhängigen Teil vor. Diese Auftrennung hat den Vorteil, dass der unabhängig gehaltene Teil auf verschiedene konkrete Plattformen beziehungsweise Anwendungen übertragen werden kann. Hierdurch wird ein größeres Maß an Wiederverwendbarkeit der entwickelten Softwarebestandteile erreicht. Ein Beispiel dafür ist eine vom eingesetzten Roboter unabhängige Aufgabenbeschreibung, die aufwandsarm von einer Roboterplattform auf eine andere übertragbar ist.

Mit dem in dieser Arbeit verfolgten Ziel, die Erstellung einer umfangreichen Bibliothek an Skills und einer größeren Anzahl an Anwendungen zu erleichtern, stellt sich jedoch die Frage, wo exakt die Grenze zwischen anwendungsspezifischen und -unabhängigen Teilen des Modells verlaufen



(a) OMG (b) Klotzbücher et al. (c) Vanthienen et al. (d) pitasc

Abbildung 4.3: MDA 4-Ebenenmodell der Abstraktion

soll. Betrachtet man beispielsweise einen generischen, anwendungsunabhängig beschriebenen Schraub-Skill, so wäre es naheliegend, diesen auf Ebene M2 des MDA 4-Ebenenmodells zu platzieren, beispielsweise als Template, das anschließend auf Ebene M1 für spezifische Robotersysteme und Anwendungen parametriert wird. Dies wirft jedoch weitere Fragen auf: Auf welcher Ebene befände sich dann ein Schraub-Skill, der bereits an einen bestimmten Schrauber-Hersteller angepasst wurde? Wo würde der Skill eingeordnet werden, wenn er für eine bestimmte Art Schrauben, ein bestimmtes Produkt, eine bestimmte Produktvariante oder für eine kleine Reihe an Schraubverbindungen auf einem spezifischen Produkt optimiert wurde? In diesem Kontext ist eine scharfe Trennung zwischen anwendungsspezifischen und -unabhängigen Modellen nicht zweckdienlich.

Das *pitasc* Skill-Modell beruht entsprechend dieser Vorüberlegung auf einer stufenlosen Verschmelzung der Ebenen M1 und M2, wie in Abbildung 4.3d dargestellt. Anwendungen werden demnach auf gleiche Weise beschrieben wie die dazu verwendeten Modelle selbst. Anschaulich bedeutet dies, dass die Erstellung einer Anwendung durch Komposition und Parametrierung von Skills auf die gleiche Weise geschieht wie die Erstellung eines Skills durch Komposition und Parametrierung von Unterbausteinen, bis hin zur Definition des Basisvokabulars des Modells, wie Frames, Matrizen und Metadaten. Dies grenzt sich von template-basierten Methoden ab, bei denen die Definition der Templates und deren anwendungsspezifische Parametrierung getrennt sind. Dagegen erleichtert die konzeptionelle Gleichheit der Modelle beider Ebenen die Verallgemeinerung von anwendungsspezifischen Resultaten und somit deren Wiederverwendung für weitere Anwendungen. Umgekehrt lassen sich allgemeiner gestaltete Bausteine ohne Umwege in konkrete Anwendungen einbinden.

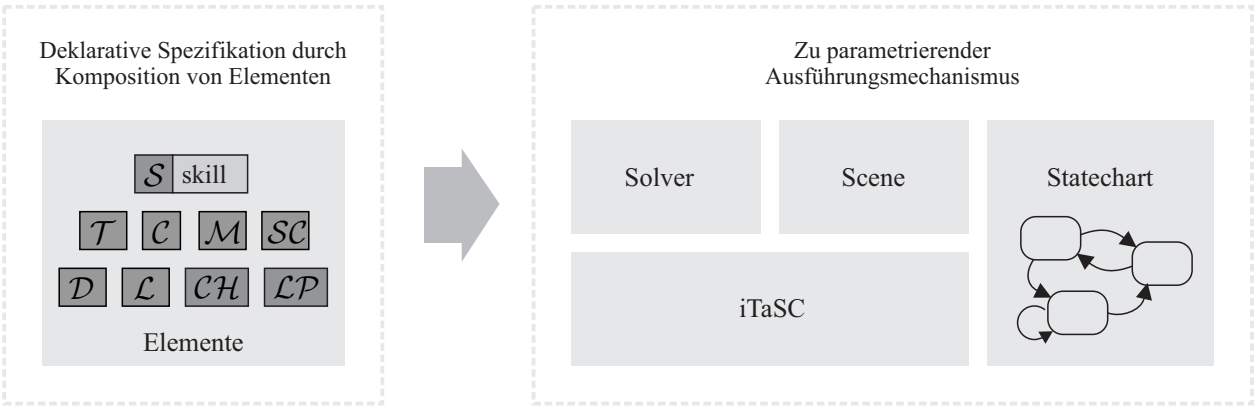


Abbildung 4.4: Deklarative Aufgabenbeschreibung und Parametrierung des Ausführungsmechanismus

Analog zu den Modellierungsansätzen von Klotzbücher et al. (2011) und Vanthienen et al. (2013) lässt sich auch beim vorliegenden Ansatz über Modell-zu-Text-Transformationen (M2T) ein Anwendungsmodell auf verschiedenen Implementierungen ausführen. In dieser Arbeit wird dies anhand von zwei Implementierungen in Python und C++ demonstriert.

Deklarative Programmierung

Eine weitere Sichtweise auf das System ist aus Perspektive des Programmierparadigmas möglich. So wird das Skill-Modell deklarativ programmiert. Diese Art der Programmierung grenzt sich durch einen Fokus auf das erwünschte *Ergebnis* gegenüber einer imperativen Programmierung ab, bei der die Implementierung des *Lösungswegs* im Fokus steht. Typische Vertreter deklarativer Programmierung sind die Datenbanksprache SQL und Reguläre Ausdrücke, mit denen beschrieben wird, welche Datenbankeinträge abgefragt beziehungsweise nach welchem Muster Wörter aus einem Text gefiltert werden sollen. Charakteristisch ist bei deklarativen Sprachen, dass aus Nutzersicht nur beschrieben wird, *was* erreicht werden soll, nicht *wie*. Aus architektonischer Sicht findet eine Auftrennung in die Beschreibung einer konkreten Aufgabe und einen allgemeinen Ausführungsmechanismus statt.

Die deklarative Programmierung bietet eine Reihe von Vorteilen. Unbeabsichtigte Nebeneffekte werden vermieden, da der Ausführungsmechanismus stets nach demselben festen Muster durchlaufen wird und die Aufgabenbeschreibungen in weiten Teilen automatisch auf Fehler geprüft werden können. Der Ausführungsmechanismus kann zudem vorab ausgiebig getestet werden und somit eine sehr stabile Form erreichen. Darüber hinaus muss der Anwender lediglich die Aufgabenbeschreibung vornehmen, eine detaillierte Kenntnis über die Funktionsweise des ausführenden Mechanismus ist in der Regel nicht mehr nötig.

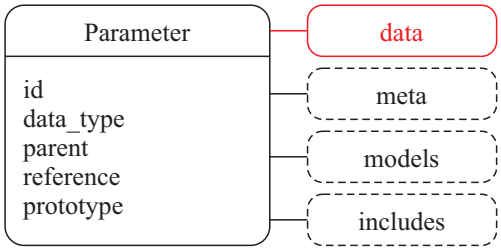


Abbildung 4.5: Parameter als Datencontainer

Tabelle 4.1: Attribute eines Parameters, wobei Referenzen auf andere Parameter mit einem * gekennzeichnet werden

Name	Typ	Beschreibung
`id`	String	Lokal eindeutiger Name zur Identifikation des Parameters
`data_type`	String	Typ des Datenfelds
`data`	(variierend)	Daten des Parameters
`parent`	Parameter*	Elternknoten des Parameters im Parameterbaum
`reference`	Parameter*	Referenz auf einen anderen Parameter im Parameterbaum
`prototype`	Parameter*	Prototyp des Parameters
`meta`	Dictionary	Metadaten des Parameters
`models`	Dictionary	Neu definierte Parameter zur Verwendung als Prototypen
`includes`	Dictionary	Aus Dateien eingebundene Prototypen

Wie in Abbildung 4.4 schematisch dargestellt wird, definieren im vorliegenden System iTaSC-Formalismus, Solver, Scene und Statechart den Ausführungsmechanismus, der aus einer Aufgabenbeschreibung die gewünschte Roboterbewegung erzeugt. Der Nutzer erstellt Anwendungen, indem er einzelne Elemente (Skills, Stoppbedingungen, Tasks usw.) durch Komposition miteinander verknüpft. Diese deklarative Beschreibung wird zur Laufzeit für die Parametrierung des Ausführungsmechanismus herangezogen.

Parameter

Parameter sind Container für beliebige Daten und ergänzen diese Daten mit Informationen für ihre Verarbeitung und Instanziierung. Wie in Abbildung 4.5 dargestellt können Parameter neben den eigentlichen Daten Metadaten des Parameters enthalten, neue Parameter als Prototypen definieren und bereitstellen sowie Parameter aus anderen Dateien einbinden (engl. als *includes* bezeichnet) und ebenfalls als Prototypen bereitstellen.

Alle Daten des Skill-Modells werden als Parameter repräsentiert. Dies vereinfacht das Skill-Modell und ermöglicht es, Methoden wie die Komposition und Vererbung auf alle Parameter

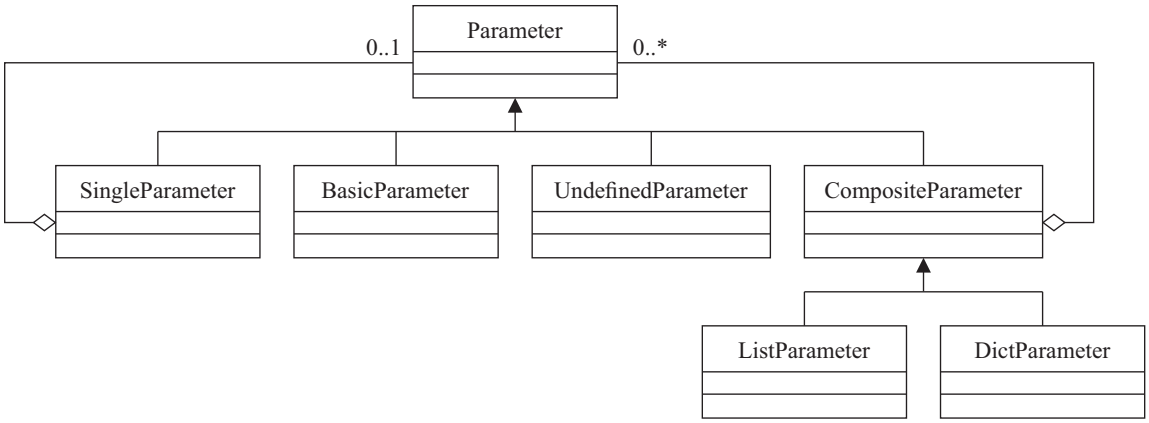


Abbildung 4.6: UML-Diagramm der Parameter-Klassen

gleichermaßen anzuwenden. Im Vergleich zu anderen Systemen können also nicht nur Skills, sondern auch beliebige andere Bausteine wiederverwendet werden. Parameter besitzen dazu die in Tabelle 4.1 aufgeführten Attribute.

- Die `id` ist das einzige Attribut, das für jeden Parameter zwingend erforderlich ist, alle weiteren Attribute sind optional. Die `id` muss lokal einzigartig sein, um eine Identifikation des Parameters zu erlauben.

- Der String `data_type` definiert den Typ des Datenfelds und beschreibt somit, welche Art von Container der Parameter darstellt. Dieser kann entweder eine Variable eines von vier Basisdatentypen besitzen (Bool, Integer, String oder Fließkommazahl), einen einzelnen anderen Parameter, eine Liste anderer Parameter oder ein assoziatives Datenfeld. Darüber hinaus kann der Datentyp noch undefiniert sein. Abbildung 4.6 zeigt die Implementierung in Form des Kompositum-Entwurfsmusters (engl. *composite pattern*) nach Gamma et al. (1994). Aus Sicht des Modells existieren in Summe die acht verschiedenen Datentypen `bool`, `int`, `float`, `string`, `parameter`, `list`, `dict` und `undefined`. Einmal gesetzt, kann der Datentyp nicht mehr geändert werden.

  Ein assoziatives Datenfeld enthält eine Reihe an Parametern, auf die man über ihre jeweilige `id` zugreifen kann. Assoziative Datenfelder sind auch unter den Bezeichnungen *Dictionary* (Python) oder *Map* (C++) geläufig. Im Folgenden werden Parameter mit assoziativen Datenfeldern als Dictionary-Parameter bezeichnet.

  Ähnlich zu Variablen beispielsweise in JavaScript, kann der Datentyp eines Parameters zunächst undefiniert bleiben. Dies erlaubt es, einen generischen Prototyp für andere Parameter zu definieren, welche dann konkrete Datentypen festlegen. Nahezu alle Parameter sind beispielsweise vom Prototyp `base` abgeleitet, der die wichtigsten Metadaten wie einen Beschreibungstext definiert.

- Das Feld `data` enthält schließlich die eigentlichen Daten des Parameters, die sich je nach Datentyp unterscheiden.

Für die Darstellung eines Parameters im Parameterbaum gibt es drei Attribute, die auf andere Parameter referenzieren können.

- Über den Elternknoten `parent` wird der Parameter dem Parameterbaum hinzugefügt. Der Elternknoten ist der Besitzer des Parameters und bestimmt seine Lebensdauer. Die Komposition von Parametern zu einem Parameterbaum wird in Abschnitt 4.1.2 behandelt.

- Ist ein Parameter eine Referenz, so besitzt er selbst keine Daten und verweist stattdessen mit seinem Feld `reference` auf einen anderen Parameter des Parameterbaums. Referenzen sind ein weiteres Werkzeug der Komposition und werden ebenfalls in Abschnitt 4.1.2 betrachtet.

- Das Feld `prototype` verweist auf den Prototyp des Parameters, dessen Eigenschaften der Parameter erbt. Dieser Prototyp kann wiederum ebenfalls einen Prototyp besitzen. Folgt man der Reihe von einem Prototyp zum nächsten, so erhält man die sogenannte *Prototypenkette* eines Parameters. Im späteren Abschnitt 4.1.3 wird dazu ausführlich auf die prototypbasierte Vererbung eingegangen.

Darüber hinaus enthält jeder Parameter drei optionale Dictionaries mit weiteren Parametern.

- Als Metadaten `meta` werden Informationen angegeben, die wichtig für die Dokumentation oder Instanziierung des jeweiligen Parameters sind.

- Die `models` enthalten neu definierte Parameter, die als Prototypen für weitere Parameter dienen können.

- Ähnlich dazu können die aus anderen Dateien über `includes` eingebundenen Parameter als Prototypen verwendet werden.

Ist ein Parameterbaum vollständig eingelesen und enthält er alle benötigten Informationen, so können die darin beschriebenen Elemente instanziiert und ausgeführt werden.

Instanziierung

Die Instanziierung beschreibt die M2T, um das Modell einer Anwendung in eine ausführbare Form zu überführen. Entgegen der allgemeinen Bezeichnung *Modell-zu-Text* wird hierbei kein

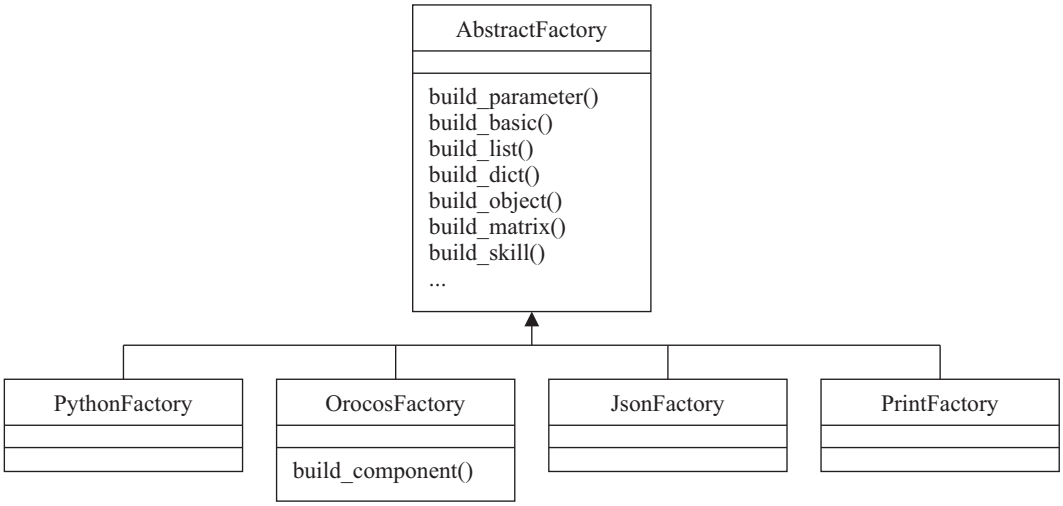


Abbildung 4.7: UML-Diagramm der Factory-Klassen

Programmcode generiert. Es werden stattdessen konkrete Objekte der Python- beziehungsweise C++/OROCOS-Implementierung instanziiert und parametriert. Alle dafür benötigten Informationen sind im Parameterbaum hinterlegt, sodass die Transformation ohne Eingriff des Nutzers geschehen kann.

Je nach Art des Parameters muss dieser bei der Instanziierung unterschiedlich behandelt werden. Manche Parameter repräsentieren einfache Strings, Zahlenwerte oder Matrizen, andere werden zur Erzeugung und Parametrierung beliebiger Objekte oder OROCOS-Komponenten herangezogen. Ausschlaggebend dafür, wie ein Parameter instanziiert wird, sind einerseits sein Typ, definiert durch seinen Prototyp beziehungsweise seine Prototypenkette, andererseits die in den Metadaten hinterlegten Informationen.

Um den Instanziierungsprozess zu strukturieren und einfach erweiterbar zu gestalten, werden etablierte Entwurfsmuster angewandt (Gamma et al. 1994). Damit einhergehend wird zudem eine weitestgehende Entkopplung des Parametermodells von den Implementierungen in Python beziehungsweise C++ erreicht. Ein Modell kann so mit beiden Implementierungen ausgeführt werden.

Der Instanziierungsprozess wird durch eine *Abstract Factory* (dt. Abstrakte Fabrik) definiert[8]. Der Parameterbaum wird dazu bei der Wurzel beginnend durchlaufen, wobei die einzelnen Parameter rekursiv instanziiert werden. Die vier in Abbildung 4.7 aufgeführten Factory-Klassen

[8] Je nach Sichtweise und Anwendungsfall kann das Design auch als das artverwandte Entwurfsmuster *Builder* (dt. Erbauer) interpretiert werden. Für komplette Instanziierungsprozesse wäre beispielsweise eine Bezeichnung als Builder zutreffender, da hierbei der Prozess im Vordergrund steht und nach dem Durchlaufen des Prozesses ein einzelnes «Produkt» in Form einer ausführbaren Zustandsmaschine übergeben wird. Für die JSON- und Print-Factories stehen hingegen einzelne Parameter im Vordergrund, die auf spezifische Art und Weise in ein JSON-Element beziehungsweise lesbaren Text transformiert werden. Hierbei ist die Bezeichnung als Abstrakte Fabrik zutreffender.

instanziieren Parameter auf unterschiedliche Weise und für unterschiedliche Zwecke. Dazu gehören einerseits jeweils eine Factory-Klasse zur Ausführung des Modells mit der Python- beziehungsweise C++/OROCOS-Implementierung. Um eine grafische Benutzerschnittstelle über ein Webinterface mit dem Modell zu verbinden, existiert zudem eine JSON-Factory. Mithilfe dieser können Informationen im JSON Datenformat über beliebige Parameter abgerufen werden. Darüber hinaus erlaubt es eine weitere Factory-Klasse, Informationen über Parameter in menschenlesbarer Form zu Debugging-Zwecken auszugeben.

Eine erweiterbare Reihe an *Factory Methods* (dt. Fabrikmethoden) definiert, auf welche Weise jeder Parameter instanziiert wird. Die wichtigsten Methoden sind ebenfalls in Abbildung 4.7 aufgeführt. Basisdatentypen können direkt in die entsprechenden Typen der jeweiligen Programmiersprache übersetzt werden. Listen- und Dictionary-Parameter werden in Python als `list` beziehungsweise `dict` erzeugt, in C++ als `vector` beziehungsweise `map`. Diese werden anschließend gefüllt, indem der Build-Prozess für ihre jeweiligen Unterparameter weitergeführt wird.

Objekt-Parameter sind spezielle Dictionary-Parameter. Sie enthalten Metadaten, die alle (implementierungsspezifischen) Informationen für die Instanziierung eines Objekts enthalten. Für Python gehören dazu ein Modul (hier eine Python-Datei in einem ROS-Package) und die zu instanziierende Klasse darin. Für OROCOS-Komponenten wird ein OROCOS Paketname und eine zu instanziierende Komponente angegeben. Darüber hinaus lassen sich für OROCOS Echtzeitparameter wie der gewählte Scheduler, die Prozesspriorität und die Periodendauer für eine zyklische Aktivierung der Komponente spezifizieren. Für einfache C++ Objekte werden der Name einer Programmbibliothek und eine darin enthaltene Funktion zur Erzeugung des Objekts angegeben.

Nach der Erzeugung des Objekts erfolgt die Parametrierung über eine (ebenfalls implementierungsspezifische) Initialisierungsfunktion. C++ Objekte beziehungsweise OROCOS-Komponenten werden mit einer Map aus Namen und Werten parametriert. Python unterstützt über das sogenannte *Keyword Argument Unpacking* ein direktes Zuweisen der Parameterwerte zu den Argumenten der Initialisierungsfunktion.

Für Parametertypen können individuelle Fabrikmethoden implementiert und registriert werden. Dies ist unter anderem für Matrizen der Fall. Ein Matrix-Parameter ist ein spezieller String-Parameter, dessen Werte in einer einfach schreibbaren Form, beispielsweise `'1 2; 3 4'`, angegeben werden können. Von der Fabrikmethode werden diese zu einer Matrixvariablen umgewandelt. Einen weiteren Sonderstatus stellen Skills dar. Spezielle Fabrikmethoden übernehmen den Aufbau der hierarchischen Zustandsmaschine, einschließlich der Verknüpfung der Zustände über Transitionen.

Das Konzept des vorgestellten Instanziierungsprozesses bietet eine Reihe an Vorteilen. Ein Hinzufügen neuer, spezifischer Parametertypen (analog zu Matrix-Parametern) ist sehr einfach durch das Erstellen und Registrieren von Fabrikmethoden möglich, ohne dabei andere Programmteile modifizieren zu müssen. Das Hinzufügen neuer Objekt-Parameter, beispielsweise für Stoppbedingungen oder Skripte, ist ganz ohne Änderung der Factory-Klasse möglich, da sämtliche Informationen für die Erzeugung der Objekte in den Metadaten hinterlegt werden.

4.1.2 Komposition

Wie im vorherigen Abschnitt bereits angedeutet wird, entsteht durch das Verschachteln von Listen- und Dictionary-Parametern eine hierarchische Struktur. Dies ist insbesondere für die Beschreibung eines hierarchischen Statecharts durch Skills und Sub-Skills der Fall. Es werden im *pitasc* Parametermodell zudem nicht nur isolierte Parameter oder Skills zusammengefügt. Vielmehr gibt es auch Referenzen von einem Parameter im Baum auf einen anderen, Suchalgorithmen, die den Parameterbaum durchlaufen, und Regeln, die bestimmen, wie der Parameterbaum verändert werden darf. Dafür müssen einige Eigenschaften wohlüberlegt definiert werden, unter anderem der Sichtbarkeitsbereich von Parametern (engl. *scope*), wie sich Parameter gleichen Namens verdecken (engl. *shadowing*) oder die Suchreihenfolge der Algorithmen.

Um die Konzepte dieses Abschnitts zu verdeutlichen, zeigt Abbildung 4.8 die vereinfachte Baumstruktur einer Applikation `my_app` mit den drei Skills `ptp_motion`, `lin_motion` und `guarded_approach`. Im Vordergrund steht dabei zunächst der abstrakte Parameterbaum als solcher, die inhaltliche Beschreibung der Struktur einer Applikation wird im späteren Abschnitt 4.2.2 gegeben.

Suchalgorithmen des Parameterbaums

Tabelle 4.2 stellt eine Reihe von Suchalgorithmen vor, um im Parameterbaum bestimmte Parameter zu finden. Im Folgenden werden die einzelnen Suchalgorithmen in aller Kürze vorgestellt. Dazu wird jeweils ein stark vereinfachter Pseudocode des Algorithmus angegeben. Die Ausgestaltung der Suchalgorithmen definiert die Sichtbarkeitsbereiche von Parametern. Diese werden im Anschluss betrachtet.

Die in Algorithmus 1 beschriebene Funktion `find_parameter` sucht den Parameter zu einer bestimmten `id`. Der Algorithmus startet beim suchenden Parameter. Ist dieser ein Dictionary-Parameter, so wird überprüft, ob er den gesuchten Parameter selbst enthält. Ist dies nicht der Fall, so wird die Suche bei seinem Elternparameter fortgesetzt. Wird der Wurzelknoten des

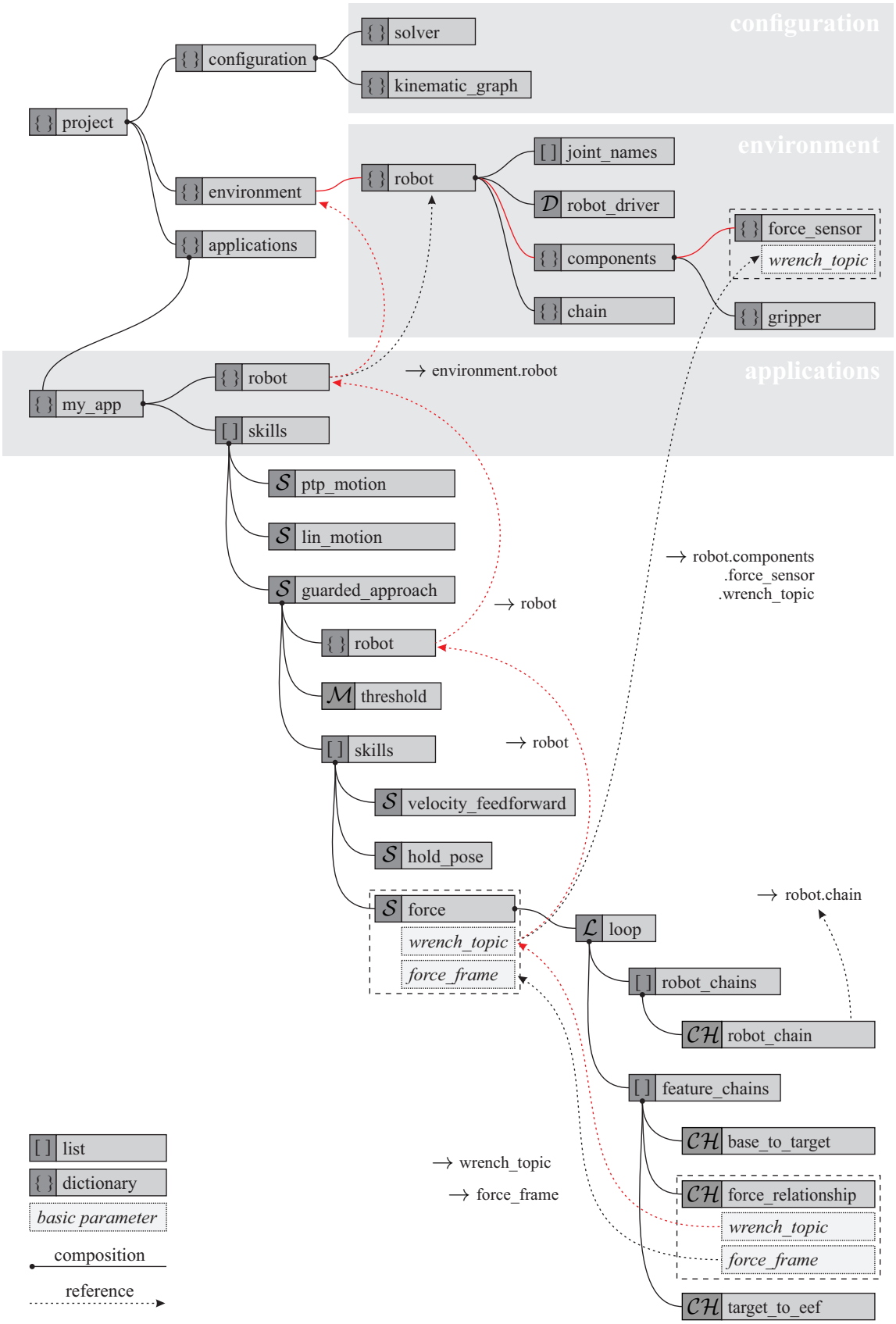


Abbildung 4.8: Beispiel eines Parameterbaums

Tabelle 4.2: Suchalgorithmen des Parameterbaums

Name	Beschreibung
`find_parameter`	Sucht den Parameter mit einer bestimmten `id`
`find_reference`	Sucht den referenzierten Parameter
`is_a`	Prüft, ob der Parameter von einem bestimmten Typ ist
`find_model`	Sucht einen bestimmten Prototyp
`find_models`	Sucht alle Prototypen eines bestimmten Typs

Baums erreicht (der keinen weiteren Elternparameter besitzt), bevor der gesuchte Parameter gefunden wurde, so wird die Suche beendet und `none` zurückgegeben.

Algorithmus 1
Suche des Parameters mit einer bestimmten `id`

```
function FIND_PARAMETER(id)
    if data_type is dict and id in data then
        return data[id]
    else if no parent then
        return none                                  ▷ Parameter not found
    else
        return parent.find_parameter(id)
```

Referenzen werden mit der Funktion `find_reference` in Algorithmus 2 aufgelöst. Diese akzeptiert dazu eine Referenz in Form eines Pfades in Punktnotation. In der gezeigten Anwendung ist dies beispielsweise für `wrench_topic` → `robot.components.force_sensor.wrench_topic` dargestellt. Zunächst wird der Pfad in den Namen des ersten Parameters sowie den restlichen Pfad getrennt. Der erste Parameter wird über die Funktion `find_parameter` bestimmt. Anschließend wird der verbleibende Pfad bis zum Zielparameter durchlaufen. Pfade können dafür bei Listen-Parametern einen Listeneintrag über die Klammernotation auswählen, wie beispielsweise `robot.joint_names[2]`.

Algorithmus 2
Suche eines referenzierten Parameters

```
function FIND_REFERENCE(id)
    parameter, path = split_first(id)
    return find_parameter(parameter)[path]
```

Um Selbstreferenzen zu vermeiden, wird derjenige Parameter ausgeschlossen, von dem die Suche ausgeht (nicht im Pseudocode dargestellt). Im Beispiel aus Abbildung 4.8 ist für den Parameter

`wrench_topic` des `force_relationship` Parameters der String `wrench_topic` als Referenz eingetragen. Durch die genannte Regel entsteht jedoch keine Selbstreferenz, stattdessen wird der gleichnamige Parameter des `force` Skills referenziert. Dieses Prinzip ermöglicht ein sehr einfaches «Durchreichen» von Referenzen in Richtung des Wurzelknotens.

Mit der Funktion `is_a` wird geprüft, ob ein Parameter von einem bestimmten Typ ist[9]. Der Algorithmus prüft zunächst, ob der Parameter eine Referenz auf einen anderen Parameter enthält und leitet die Anfrage gegebenenfalls an diesen weiter. Ist dies nicht der Fall, wird geprüft, ob der Parameter selbst der gesuchte Prototyp ist oder schließlich die Prototypenkette rekursiv durchlaufen:

Algorithmus 3
Überprüfen des Typs eines Parameters

```
function IS_A(type)
    if reference then
        return reference.is_a(type)
    else if self.is_prototype() and self.id is type then
        return true
    else if no prototype then
        return false
    else
        return prototype.is_a(type)
```

Der `find_model` Algorithmus sucht einen Prototyp mit einer bestimmten `id`. Im Gegensatz zu `find_parameter` werden hier jedoch nicht die Datenfelder durchsucht, sondern die `models` und `includes` Felder, in denen die Prototypen hinterlegt sind. Zunächst werden die eigenen Felder des Parameters durchsucht. Wird hier kein Prototyp des gesuchten Namens gefunden, so wird die Prototypenkette des Parameters nach oben durchlaufen. Sofern auch hier kein entsprechender Prototyp vorhanden ist, wird die Suche mit dem Elternparameter fortgeführt und folgt somit dem Baum in Richtung Wurzel. Sobald ein Prototyp gefunden wurde, wird die Suche beendet und der Prototyp zurückgegeben.

Auf ähnliche Weise funktioniert der `find_models` Algorithmus. Hier wird jedoch nicht ein einzelner Prototyp gesucht, sondern eine Liste aller Prototypen eines gegebenen Typs. Dies wird beispielsweise für eine grafische Benutzeroberfläche benötigt, die einem Nutzer, der einen Skill oder eine Stoppbedingung hinzufügen möchte, alle aktuell zur Verfügung stehenden Skills

[9] Zu beachten ist, dass mit *Typ* nicht der Datentyp des Datenfelds (`int`, `string`, `dict` usw.) gemeint ist. Stattdessen definiert sich der Typ durch die Prototypenkette des Parameters. Beispielsweise setzt sich die Prototypenkette der `threshold` Stoppbedingung zu `[monitor, object, dictionary, base]` zusammen. So liefert `threshold_monitor.is_a(monitor)` das Ergebnis `true`, ebenso wie `threshold_monitor.is_a(object)`. Der Datentyp der `threshold` Stoppbedingung hingegen ist `dict`.

Algorithmus 4
Suche eines Prototyps mit einer bestimmten `id`

```
function FIND_MODEL(id)                ▷ Der erste gefundene Prototyp wird zurückgegeben
    if id in models then return models[id]
    else if id in includes then return includes[id]
    else if prototype.find_model(id) then return model
    else if parent.find_model(id) then return model
    else return none                                  ▷ Kein Prototyp wurde gefunden
```

beziehungsweise Stoppbedingungen zur Auswahl stellt. Im Pseudocode nicht dargestellt ist, dass bereits durchsuchte Parameter übersprungen werden.

Algorithmus 5
Suche aller vorhandenen Prototypen eines bestimmten Typs

```
function FIND_MODELS(type)
    parameters = []
    for all models do
        parameters += model if model.is_a(type)
    for all includes do
        parameters += include.find_models(type)
    parameters += prototype.find_models(type)
    parameters += parent.find_models(type)
    return parameters
```

Sichtbarkeitsbereiche

Aus den oben beschriebenen Suchalgorithmen ergeben sich Konsequenzen für den Sichtbarkeitsbereich (engl. *scope*) eines Parameters, der beschreibt, wo der Parameter gefunden und genutzt werden kann. Modelle und einfache Parameter können über ihre `id` vom suchenden Parameter aus in Richtung Wurzel des Baums gefunden werden. Umgekehrt formuliert ist die direkte Sichtbarkeit eines Parameters demnach von seinem Elternparameter aus in Richtung Blattparameter gegeben. Darüber hinaus kann von einem Parameter aus über einen Pfad in Punktnotation auf Unterparameter zugegriffen werden, wie dies beispielsweise häufig bei Referenzen genutzt wird (`robot.components.force_sensor.wrench_topic`).

Zwar müssen Parameter innerhalb eines Dictionary-Parameters eine eindeutige `id` besitzen, diese Einschränkung gilt jedoch nicht für Parameter unterschiedlicher Elternparameter. Entsprechend erlaubt der Entwurf der Suchalgorithmen, dass ein Parameter einen gleichnamigen

Tabelle 4.3: Operationen zur Modifikation des Parameterbaums

Name	Beschreibung
`AddToList`	Fügt einen Parameter zu einem `ListParameter` hinzu
`AddToDict`	Fügt einen Parameter zu einem `DictParameter` hinzu
`SetData`	Setzt die Daten eines `BasicParameter` oder `SingleParameter`
`RemoveFromList`	Entfernt einen Parameter von einem `ListParameter`
`RemoveFromDict`	Entfernt einen Parameter von einem `DictParameter`
`SetReference`	Setzt (oder entfernt) eine Referenz
`Include`	Lädt eine Datei und importiert die enthaltenen Parameter
`CloneParameter`	Führt einen `<clone>`-Befehl aus
`CreateParameter`	Führt einen `<type>`-Befehl aus
`CreateReference`	Führt einen `<reference>`-Befehl aus

Parameter verdeckt, der im Baum über ihm steht (engl. *shadowing*). Dies geschieht ähnlich zu vielen Programmiersprachen, bei denen lokale Variablen globale Variablen verdecken.

Operationen zur Modifikation des Parameterbaums

Durch die generische Modellierung des Parameterbaums («alles ist ein Parameter») ist eine kleine Anzahl an Operationen ausreichend, um einen Parameterbaum zu erstellen und zu modifizieren. Die Operationen sind mit dem Kommando-Entwurfsmuster (engl. *command pattern*) realisiert (Gamma et al. 1994). Sie werden jeweils in einer Kommando-Klasse gekapselt, die stets sowohl eine `undo` als auch eine `redo` Funktion implementiert. Dies erlaubt es, Operationen rückgängig zu machen, was insbesondere bei der Nutzung von grafischen Benutzeroberflächen hilfreich ist. Änderungen am Parameterbaum über andere Wege sind nicht zulässig, um zu verhindern, dass die Historie ausgeführter Kommandos beeinträchtigt wird.

Tabelle 4.3 führt die vorhandenen Operationen auf. Über `AddToList`, `AddToDict` und `SetData` werden Parameter oder Datenwerte dem Baum hinzugefügt, je nachdem welchen Datentyp der Elternparameter besitzt. Über `RemoveFromList`, `RemoveFromDict` und `SetData` (mit `data: none`) können sie entsprechend wieder aus dem Baum entfernt werden. Für das Setzen oder Entfernen von Referenzen wird `SetReference` verwendet. Eine neue Datei mit weiteren Parametern wird über das `Include` Kommando eingebunden.

Darüber hinaus existieren mit `CloneParameter`, `CreateParameter` und `CreateReference` eine Reihe von Hilfsoperationen, die es erleichtern, Befehle der in Abschnitt 4.2 vorgestellten

DSL auszuführen. Sie setzen die Befehle dabei je nach Datentyp des Elternparameters durch Ausführung eines der ersten drei genannten Kommandos um.

4.1.3 Prototypbasierte Vererbung

Neben der Komposition von Skills ist die Vererbung der zweite Ansatz, um die Wiederverwendung von Skills zu erleichtern. Dem *pitasc* Skill-Modell liegt die Idee zugrunde, einen Wechsel von prozeduraler hin zu objektorientierter Programmierweise zu vollziehen. Die Aufgabenbeschreibung einer Roboteranwendung wird dafür in diskrete Stücke geteilt und jeder dieser Prozessschritte durch einzelne Skills modelliert und ausgeführt. Skills wiederum umfassen einzelne Unterbausteine wie Stoppbedingungen, Skripte oder Sub-Skills. Bei der Gestaltung des Skill-Modells, insbesondere bei der Gestaltung des Vererbungsmechanismus, liegt es daher nahe, sich von objektorientierten Programmiersprachen inspirieren zu lassen. Zwar stellen das vorliegende Modell und die dazugehörige DSL keine universelle Programmiersprache dar, dennoch übernehmen sie einige ihrer Vor- und Nachteile. Im Folgenden werden daher zunächst die klassen- und die prototypbasierte objektorientierte Programmierung und Vererbung gegenübergestellt. Anschließend wird die prototypbasierte Vererbung weiter vertieft und es wird ihre Verwendung im vorliegenden System detailliert.

Klassen- und prototypbasierte objektorientierte Programmierung

Eine Klasse beschreibt die *Struktur* seiner Objekte in Form von Variablen (Attributen) und ihr *Verhalten* in Form von Funktionen (Methoden). Während alle instanziierten Objekte einer Klasse die gleiche Struktur aufweisen und dem gleichen Verhaltensmuster folgen, werden die Unterschiede zwischen den einzelnen Objekten in den konkreten Variablenwerten festgehalten. Jedes Objekt speichert demnach seinen individuellen *Zustand* (die Werte der Attribute) selbst.

Bei einem prototypbasierten Ansatz hingegen wird nicht zwischen Klassen und Objekten unterschieden. Ein konkretes Objekt wird als prototypisch für eine Reihe an weiteren Objekten angenommen. Diese Objekte werden mit dem Prototyp verknüpft und ergänzen nur diejenigen Informationen, die das jeweilige Objekt vom Prototyp unterscheiden. Dies erlaubt es, mit der Entwicklung eines individuellen Objekts zu starten und es erst *danach* zu verallgemeinern sowie die veränderlichen Eigenschaften zu beschreiben.

Lieberman (1986) argumentiert, dass diese Vorgehensweise für Menschen intuitiver ist, als umgekehrt mit einer abstrakten Beschreibung zu starten und sie erst später auf konkrete Fälle

anzuwenden. Auch ist der prototypbasierte Ansatz konzeptionell einfacher, da nicht zwischen Klassen und Objekten unterschieden werden muss und sich die damit einhergehende Komplexität verringert. Dahingegen ist in der objektorientierten Programmierung der klassenbasierte Ansatz deutlich weiter verbreitet und entsprechendes Vorwissen vorhanden. Somit mag der prototypbasierte Ansatz vielen Entwicklern ungewohnt erscheinen, wodurch sich eine Verwendung erschweren kann.

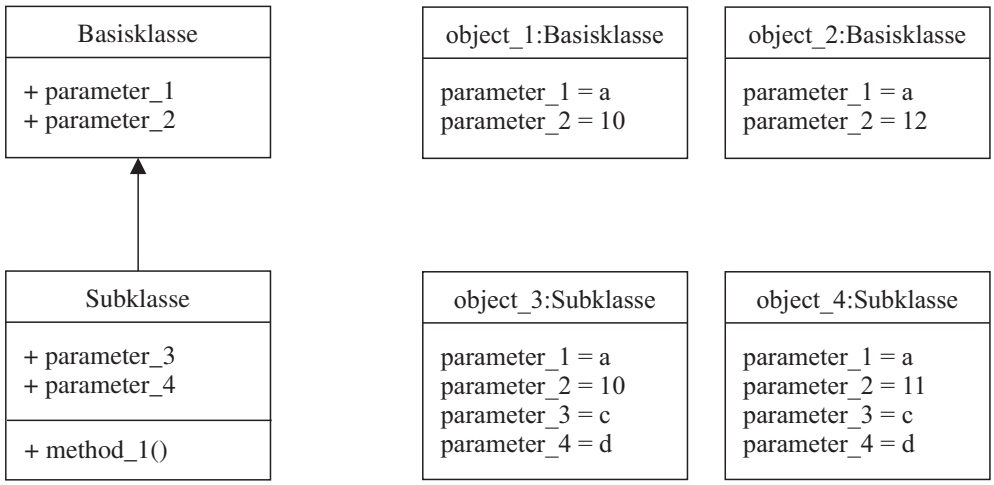
Dies zeigt sich am Beispiel von JavaScript, eine der am meisten verbreiteten Programmiersprachen weltweit. JavaScript ist prototypbasiert und nutzt den später eingeführten Delegationsansatz als Vererbungsmechanismus. Mit dem JavaScript zugrundeliegenden ECMAScript 6 Standard wurden jedoch weitere Sprachelemente eingeführt, die sich an klassenbasierten Sprachen orientieren, obwohl der vorhandene Delegationsansatz der direktere Weg wäre, Vererbung zu modellieren (Simpson 2014). Insgesamt scheinen derzeit beide Ansätze in JavaScript wenig Anwendung zu finden. In Silva et al. (2015) und Silva et al. (2017) werden 918 der populärsten quelloffenen JavaScript Anwendungen untersucht. Nach den Kriterien der Untersuchung werden lediglich 34 % der betrachteten Anwendungen als klassenfreundlich oder klassenorientiert kategorisiert. Nur 8 % der Anwendungen nutzen den Vererbungsmechanismus von JavaScript.

Klassen- und prototypbasierte Vererbung

Durch Vererbung lassen sich Klassen in objektorientierten Programmiersprachen um neue Eigenschaften erweitern, ohne dass die ursprünglichen Klassen zu groß und überladen werden. Sind gewisse Eigenschaften nämlich nur für einen Teil der Objekte einer Klasse relevant, können diese in eine Subklasse ausgegliedert werden. Umgekehrt formuliert lassen sich bei der Erstellung einer neuen Klasse existierende Eigenschaften einer bestehenden Klasse wiederverwenden und sich damit Programmieraufwand einsparen. Abbildung 4.9a zeigt das Klassendiagramm einer Basisklasse mit zwei Attributen. Diese werden von der erbenden Subklasse übernommen und um zwei weitere Attribute und eine Methode ergänzt. Das Objektdiagramm in Abbildung 4.9b zeigt zwei instanziierte Objekte je Klasse, deren Attributen Werte zugewiesen wurden.

Abbildung 4.10 zeigt die entsprechende Modellierung bei einer prototypbasierten Objektorientierung. Die Klassen entfallen hier. Das Objekt `object_1` wird stattdessen als prototypisch für die anderen Objekte angesehen. Nur die Unterschiede zwischen den Objekten müssen modelliert werden. Die Reihenfolge im Beispiel ist dabei willkürlich gewählt. Genauso könnte `object_2` als Prototyp für `object_1` dienen.

Taivalsaari (1996) betrachtet im Detail, welche der Eigenschaften Struktur, Verhalten und Zustand jeweils vererbt werden. Bei einer klassenbasierten Vererbung können Struktur und Ver-



(a) Klassendiagramm (b) Objektdiagramm

Abbildung 4.9: Beispiel für eine klassenbasierte Objektorientierung

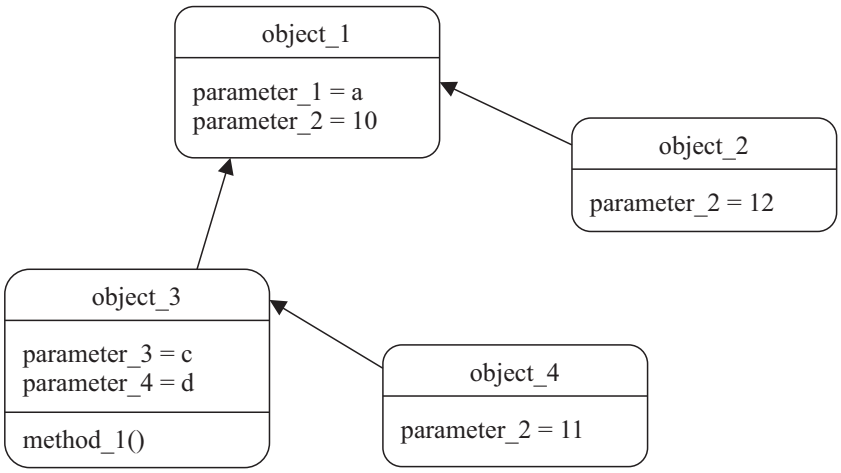
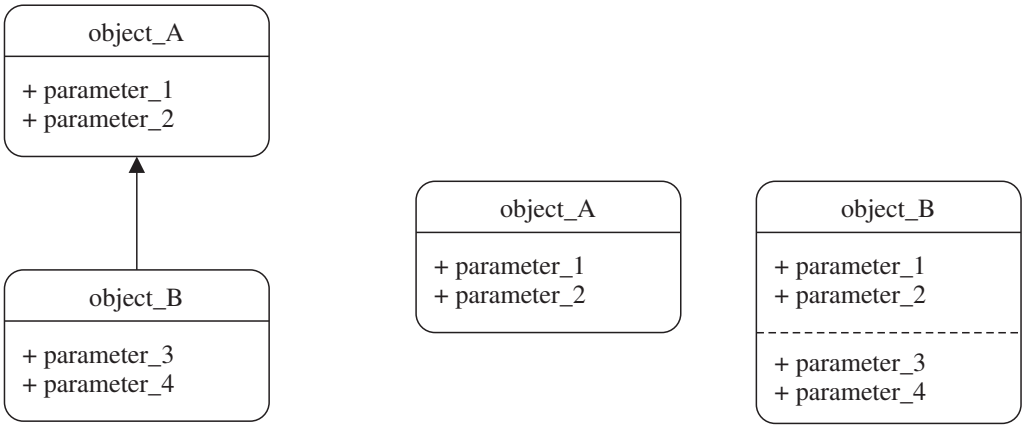


Abbildung 4.10: Beispiel für eine prototypbasierte Objektorientierung

halten vererbt werden, in Form der Deklaration von Attributen beziehungsweise der Definition von Methoden. Die Vererbung eines Zustands ist jedoch nicht möglich, da dieser von Objekten repräsentiert wird, die Vererbung aber auf Klassen beschränkt ist. Hier zeigt sich sehr deutlich der Vorteil, den ein prototypbasierter Vererbungsmechanismus für den anvisierten Anwendungsbereich haben kann. Er erlaubt eine fein abgestufte Vererbungshierarchie, die insbesondere auch Parameterwerte mit einbezieht. Wie in Abschnitt 4.1.1 motiviert, lassen sich Skills damit sehr einfach auf bestimmte Prozesse, Produkte, Varianten oder Werkzeuge spezialisieren.

Es sei angemerkt, dass bei klassenbasierten Programmiersprachen das Prototyp-Entwurfsmuster (Gamma et al. 1994) angewendet werden kann, um eine gewisse Wiederverwendung des Zustands zu erreichen. Dies ist auch der Weg, mit dem der Parameterbaum des vorgestellten



(a) Delegation (b) Verkettung

Abbildung 4.11: Varianten prototypbasierter Vererbung (Taivalsaari 1996)

Systems in der klassenbasierten Programmiersprache Python implementiert wurde. Für das Modell wird jedoch explizit auf das Konzept von Klassen verzichtet.

Ansätze der prototypbasierten Vererbung

Die beiden primären Arten prototypbasierter Vererbung, Delegation und Verkettung (engl. *concatenation*), gehen unter anderem auf Lieberman (1986) beziehungsweise Borning (1986) zurück. Bei einem delegationsbasierten Ansatz wird ein neu erzeugtes Objekt mit seinem Prototyp (oder auch mehreren Prototypen) verknüpft. Wenn das Objekt eine Anfrage durch einen Zugriff auf eine Variable oder das Aufrufen einer Funktion erhält, die es nicht selbst beantworten kann, so leitet das Objekt diese an seinen Prototyp weiter. Die Beziehung zwischen dem Objekt und seinem Prototyp wird dazu zur Programmlaufzeit weiter aufrecht erhalten (engl. *life-time sharing*).

Bei einem verkettungsbasierten Ansatz wird beim Erzeugen eines neuen Objekts der Prototyp nicht verknüpft, sondern vollständig kopiert. Eine weitere Beziehung zwischen dem Objekt und seinem Prototyp besteht danach nicht mehr. Das kopierte Objekt enthält sämtliche Informationen des Prototyps, dies jedoch nur vom Zeitpunkt des Kopiervorgangs (engl. *creation-time sharing*). Sollen nach einer Änderung des Prototyps auch kopierte Objekte aktualisiert werden, so ist die Implementierung eines Aktualisierungsmechanismus wie dem Beobachter-Entwurfsmuster (engl. *observer pattern*) nötig (Gamma et al. 1994).

Ein delegationsbasierter Ansatz hat den Vorteil einer geringeren Speicherbelegung, wohingegen bei einer Verkettung kürzere Zugriffszeiten anfallen, da hier auf Daten direkt zugegriffen wird und nicht erst eine Delegationskette durchlaufen werden muss. Diese Abwägung lässt sich jedoch auch feiner abstufen, indem beim delegationsbasierten Ansatz durch Zwischenspeicherung

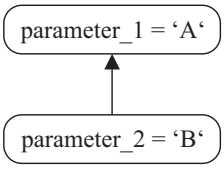
(engl. *caching*) eine größere Speicherbelegung akzeptiert wird, um Zugriffszeiten bei sich häufig wiederholenden Anfragen zu reduzieren. Umgekehrt kann beim verkettungsbasierten Ansatz erst bei einem auftretenden Schreibvorgang eine vollständige Kopie erstellt werden, wodurch eine Einsparung von Speicherplatz in vielen Fällen möglich wird.

Da für das Laden und Ausführen einer Roboteranwendung keine dynamischen Änderungen nötig sind, wurde für die Implementierung des vorliegenden Modells der Verkettungsansatz gewählt. Für die zukünftige Nutzung einer grafischen Benutzeroberfläche muss jedoch ein Mechanismus vorhanden sein, um etwaige Änderungen eines Prototyps an alle Kopien (und deren Kopien) weiterzureichen. Hierfür wäre die Umstellung auf einen Delegationsansatz vorteilhaft.

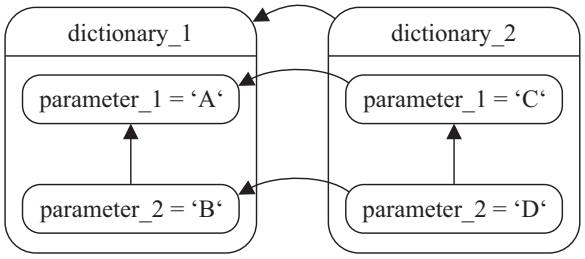
Auslegung des Vererbungsmechanismus

Bei der Gestaltung des verkettungsbasierten Vererbungsmechanismus wird die Fragestellung aufgeworfen, welche Daten des Prototyps auf welche Weise kopiert werden sollen. Dies muss auf einfache und konsistente Art geschehen, sodass für Nutzer nachvollziehbar ist, welche Auswirkungen das Anpassen oder Vererben von Parametern hat. Grundsätzlich besteht die Möglichkeit der Erzeugung einer flachen Kopie (engl. *shallow copy*) oder einer tiefen Kopie (engl. *deep copy*). Bei einer flachen Kopie werden die Attribute des Prototyps lediglich referenziert, anstatt sie komplett zu kopieren, wie es bei einer tiefen Kopie der Fall ist. Damit kommen flache Kopien jedoch nicht in Frage, da beispielsweise die Änderung der Parameter eines Skills Auswirkungen auf alle anderen Kopien dieses Skills haben würde.

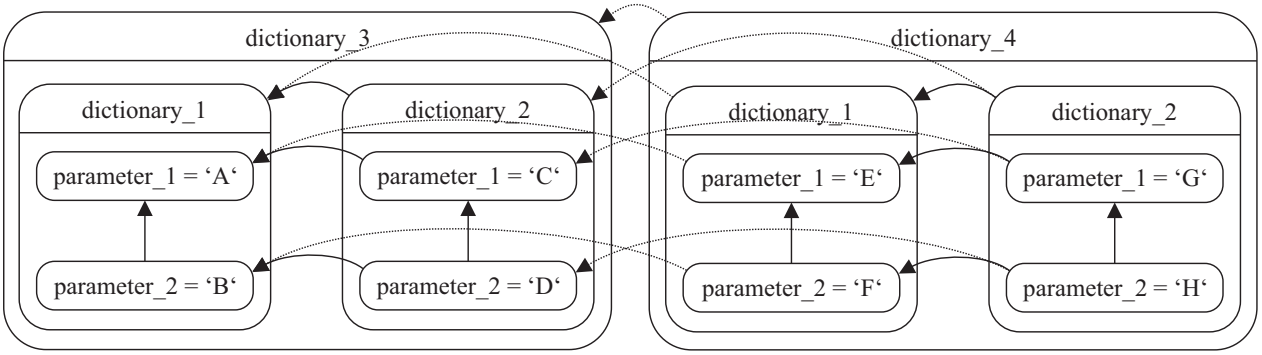
Doch auch eine tiefe Kopie wirft zunächst ein zu lösendes Problem auf. Abbildung 4.12a zeigt den String-Parameter `parameter_1` mit dem Wert `'A'`. Von diesem Parameter erbt `parameter_2`, setzt seinen Wert auf `'B'` und verdeckt somit den geerbten Wert seines Prototyps. Abbildung 4.12b erweitert das Szenario. Die beiden Parameter seien nun Teil eines Dictionary-Parameters `dictionary_1`, von dem die Kopie `dictionary_2` erzeugt wird. Durch die tiefe Kopie enthält `dictionary_2` eigene Kopien der Parameter `parameter_1` und `parameter_2` und kann diesen eigene Werte zuweisen, ohne `dictionary_1` zu verändern. Durch die tiefe Kopie entsteht aber auch ein gerichteter azyklischer Graph (DAG) der Kopierbeziehungen. So erbt der Parameter `dictionary_2.parameter_2` sowohl von `dictionary_2.parameter_1` als auch von `dictionary_1.parameter_2`. Eine Reihenfolge zu definieren, in der die Vererbung von Parametern stattfinden soll, ist nun nicht mehr trivial. Diese Problematik verstärkt sich bei jedem weiteren Vererbungsvorgang, wie Abbildung 4.12c darstellt. Zwar ließe sich durchaus eine allgemeine Reihenfolge definieren, für den Nutzer des Skill-Modells wäre aber eventuell nicht immer nachvollziehbar, welcher Parameterwert kopiert werden würde.



(a) Kopierter Parameter



(b) Kopierter Dictionary-Parameter



(c) Doppelt kopierter Dictionary-Parameter

Abbildung 4.12: Vererbungshierarchie mit Kopiermechanismus auf Basis tiefer Kopien

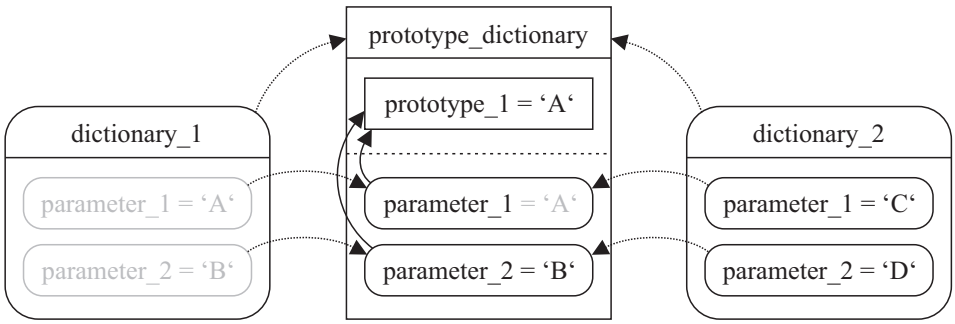


Abbildung 4.13: Vererbungshierarchie mit einfachem Kopiermechanismus

Um die Problematik einfach und robust zu lösen, werden zwei Rollen für Parameter eingeführt. Es wird explizit definiert, welche Parameter Prototypen sein können und welche nicht. Um Prototypen von normalen Parametern zu trennen, werden erstere in der Liste der `models` eines Parameters hinterlegt, wie in Abbildung 4.5 und Tabelle 4.1 bereits vorweggenommen. Prototypen werden bei einer Vererbung nicht mit kopiert, normale Parameter hingegen schon, wodurch die Vererbungshierarchie die Form eines Baums behält. Abbildung 4.13 zeigt das Szenario in dieser Variante.

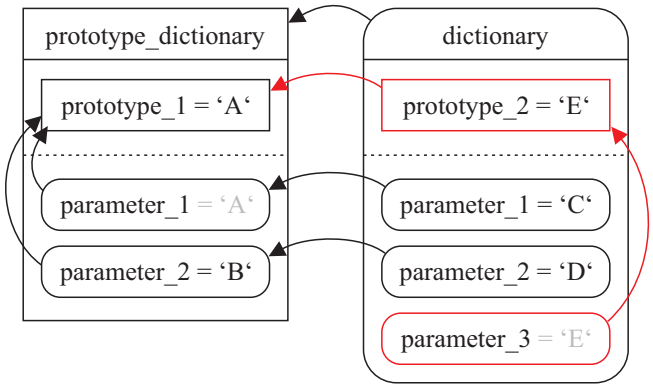


Abbildung 4.14: Beispiel für die Vererbung eines Prototyps

Prototypen lassen sich wiederum zur Erstellung neuer Protoypen verwenden. Abbildung 4.14 zeigt den Prototyp `prototype_2`, der von `prototype_1` erbt und wiederum als Prototyp von `parameter_3` dient.

Es ist wichtig zu verstehen, dass dies dennoch nicht einem klassenbasierten Ansatz entspricht. Prototypen sind kein abstraktes Modell von Objekten, wie dies bei Klassen der Fall ist. Prototypen sind konzeptionell gleich modelliert wie Parameter und nehmen lediglich eine andere Rolle bei Kopier- und Suchalgorithmen ein.

Der vorgestellte Fall bezieht sich auf einen verkettungsbasierten Ansatz. Jedoch entstünde das Problem auch bei einem delegationsbasierten Ansatz, da die Frage geklärt werden müsste, an welchen Parameter zuerst delegiert werden soll.

4.2 Domänenspezifische Sprache für die Modellierung von Montageanwendungen

Der im vorangegangenen Abschnitt vorgestellte Parameterbaum erlaubt es, eine Roboteranwendung aus den in Kapitel 3 beschriebenen Elementen zusammenzustellen und zu parametrieren. Alle Elemente und ihre Parametrierung, vom einfachen String oder Zahlenwert bis hin zu Skills und vollständigen Anwendungen, werden einheitlich durch Parameter in einer Baumstruktur dargestellt. Durch die einheitliche Modellierung ist bereits die kleine Anzahl der in Tabelle 4.3 aufgeführten generischen Operationen zur Erstellung des Parameterbaums ausreichend. Die abstrakte Darstellungsweise über Parameter erlaubt es zudem, den vorgestellten Vererbungsmechanismus auf alle Elemente gleichermaßen anzuwenden. Somit können ebenfalls vom einfachen String bis zur vollständigen Anwendung alle Arten von Elementen vererbt, angepasst und erweitert werden.

Die abstrakte Darstellung macht eine Programmierung für Menschen jedoch unhandlich. Aus diesem Grund wird im Folgenden eine domänenspezifische Sprache für *pitasc* vorgestellt, die es Nutzern erlaubt, auf einfache Art Skills und Anwendungen zu programmieren. Fowler et al. (2010) beschreiben eine DSL als «dünne Fassade» über einem Modell. Das Modell – im vorliegenden Fall der genannte Parameterbaum – sei der Motor, der das Verhalten zur Verfügung stelle, die DSL sei lediglich ein Mittel, das Modell auf eine lesbare Art mit Inhalt zu füllen. Viele Vorteile seien also auch allein durch das Modell bereits gegeben. Dennoch ist eine durchdachte DSL wichtig, da sich durch sie entscheidet, wie gut das Modell vom Benutzer verwendbar ist. Alternativ zu einer DSL kann eine grafische Benutzeroberfläche genutzt werden. Dies wird hier jedoch nicht betrachtet.

4.2.1 Syntax und allgemeine Sprachelemente

In diesem Abschnitt wird die vorgeschlagene DSL und ihre allgemeinen Sprachelemente eingeführt. Diese beschreiben das domänenunabhängige Meta-Metamodell (M3) nach dem MDA 4-Ebenenmodell und stellen die Sprache dar, in der später domänenspezifische Sprachelemente (wie Skills zur Programmierung von Roboteranwendungen) modelliert werden. Das domänenunabhängige Modell befasst sich dabei mit dem Parameterbaum, den Parametern sowie mit deren in Tabelle 4.1 aufgeführten Eigenschaften. Die Parameter werden durch Komposition und Vererbung im Parameterbaum hierarchisch angeordnet. Dafür werden die Operationen aus Tabelle 4.3 verwendet, die wiederum auf die Suchalgorithmen aus Tabelle 4.2 zurückgreifen.

Nach den in Abschnitt 2.3 aufgeführten Kategorien lässt sich die hier vorgestellte DSL als externe DSL einordnen, die sich der Syntax von XML bedient (engl. *carrier-syntax*). XML erscheint durch die Schreibweise mit *tags* (dt. Auszeichnungen, `<name>...</name>`) zunächst schwerer lesbar und aufwendiger zu schreiben als andere Auszeichnungssprachen, wie beispielsweise YAML. Jedoch bietet XML über Attribute (`<name attribut='.../>`) die Möglichkeit, Eigenschaften eines Parameters im Baum (`id`, `data_type`, `prototype`, `reference_id`) visuell deutlich getrennt von seinen Unterparametern (in den `data`, `meta`, `models` und `includes` Listen) darzustellen. Die hierarchische Struktur einer in XML formatierten DSL-Datei entspricht damit der Struktur des sich daraus ergebenden Parameterbaums.

Erzeugen von Parametern

Folgende Syntax dient zur Erstellung eines Parameters mit den in Tabelle 4.1 aufgelisteten Eigenschaften, beispielhaft dargestellt für einen String-Parameter:

```
<type id="mein_parameter" prototype="base" data_type="string">
  <meta>
    <!-- Definition von Metadaten -->
  </meta>
  <includes>
    <!-- Laden vorhandener Prototypen aus der Bibliothek -->
  </includes>
  <models>
    <!-- Definition neuer Prototypen -->
  </models>
  <data>Ein String</data>
</type>
```

Der neue Parameter kopiert den mit dem `prototype`-Attribut angegebenen Prototyp und erhält dabei den Namen, der mit dem `id`-Attribut gesetzt wird. Der Prototyp muss mit dem in Abschnitt 4.1.2 beschriebenen Algorithmus 4 vom Elternparameter ausgehend gefunden werden können. Ist kein `id`-Attribut gesetzt, so wird automatisch der Name des Prototyps verwendet. Ist dieser bereits vergeben, so wird als Suffix eine Zahl angehängt. Der Elternparameter (`parent`) ergibt sich automatisch aus der Stelle, an der der neue Parameter dem Parameterbaum hinzugefügt wird.

Wie in Abschnitt 4.1.1 beschrieben, kann ein Parameter zusätzlich ein `data_type`-Attribut enthalten, wenn sein Prototyp noch keinen Datentyp vorgibt. Der Datentyp kann einem Basisdatentyp entsprechen (`bool`, `int`, `float` und `string`), einem einzelnen Unterparameter (`parameter`), einer Liste (`list`) oder einem Dictionary (`dict`). Im gezeigten Beispiel wird ein Parameter vom Datentyp `string` erstellt, entsprechend wird der Inhalt des `data` Felds als einzelner String interpretiert. Darüber hinaus wird die Möglichkeit geschaffen, mehrere Werte eines Basisdatentyps mit Kommata getrennt anzugeben (auch *csv*-Format genannt, von engl. *comma-separated values*), um beispielsweise mehrere Achsen auszuwählen:

```
<type id="coordinates" prototype="base" data_type="csv:string">
  <data>x, y, z</data>
</type>
```

In der zugehörigen Fabrikmethode werden die angegebenen Werte getrennt und als Liste an die entsprechende Initialisierungsfunktion der parametrierten Instanz weitergegeben.

Abgesehen von der Erstellung neuer Prototypen ist es in den meisten Fällen ausreichend, das `data` Feld zu bearbeitet. Aus diesem Grund wird eine Kurzform der oben aufgeführten Syntax für die Erstellung eines Parameters eingeführt, die nur eine Änderung des Datenfelds erlaubt:

```
<clone id="mein_parameter" prototype="base" data_type="string">Ein String</clone>
```

Ebenso wird eine Kurzform für die Erstellung eines Parameters eingeführt, der eine Referenz auf einen anderen Parameter definiert:

```
<reference id="mein_parameter" reference_id="verknuepfter_parameter"/>
```

Der verknüpfte Parameter wird mit Algorithmus 2 im Parameterbaum gesucht. Der neu erstellte Parameter erhält automatisch dessen Prototyp als eigenen Prototyp.

Bearbeiten von Parametern

Um bereits existierende (geerbte) Parameter zu ändern beziehungsweise zu erweitern, wird das `member`-Element eingeführt (von engl. *member variable*, Attribut einer Klasse beziehungsweise eines Objekts in der objektorientierten Programmierung):

```
<member id="description">Beschreibungstext des Parameters</member>
```

Für die Änderung der Werte von Basisdatentyp-Parametern wird die `SetData`-Operation aus Tabelle 4.3 herangezogen. Bei Listen- und Dictionary-Parametern hingegen werden Unterparameter über die entsprechenden Operationen `AddToList` und `AddToDict` hinzugefügt. Ein Beispiel ist das Eintragen einer Stoppbedingung in die Liste der Stoppbedingungen eines Skills:

```
<member id="monitors">
  <clone id="pose_monitor" prototype="control_error_monitor">
    <!-- Parameter -->
  </clone>
</member>
```

Neben dem Setzen eines Basisdatentyp-Werts und dem Hinzufügen von Unterparametern ist das Setzen einer Referenz die dritte Art, bestehende Parameter zu ändern:

```
<member id="chain" reference_id="robot.base"/>
```

Im Gegensatz zum oben eingeführten `<reference>` Element wird hier ein bestehender Parameter modifiziert. Zum Setzen der Referenz wird die Operation `SetReference` verwendet, die auch hier den verknüpften Parameter mit Algorithmus 2 im Parameterbaum sucht.

Vererbung von Parametern

Jeder neu erstellte Parameter, der einen Prototyp besitzt, erbt dessen Eigenschaften. Um der in Abschnitt 4.1.3 beschriebenen Problematik bei der Erstellung tiefer Kopien zu entgehen, werden Prototypen in der Liste der `models` eines Parameters hinterlegt und nicht mit vererbt. Dort können sie jedoch von Algorithmus 4 gefunden werden und somit anderen Parametern als Prototyp dienen.

Um diese Vorgehensweise zu verdeutlichen, zeigt DSL-Codebeispiel 4.1 die Erstellung eines Prototyps. Zunächst wird per `include` die Definition eines LIN-Skills (`skill_lin_motion`) geladen. Anschließend wird der Skill-Prototyp `lin_tool0` definiert, für den ein Werkzeugkoordinatensystem vorausgewählt wird. Der neu erstellte Prototyp wird im Beispiel zweifach kopiert, um

Codebeispiel 4.1: Prototypen und Vererbung

```
<type id="meine_sequenz" prototype="skill_sequence">

  <!-- Laden eines vorhandenen Skill-Prototyps aus der Bibliothek -->
  <includes>
    <include package="pitasc_library" file="skills/skill_lin_motion.xml"/>
  </includes>

  <!-- Definition eines neuen Prototyps -->
  <models>
    <clone id="lin_tool0" prototype="skill_lin_motion">
      <member id="tool">tool0</member>
    </clone>
  </models>

  <!-- Verwendung des Prototyps -->
  <data>
    <member id="skills">
      <clone prototype="lin_tool0">
        <member id="target">target1</member>
      </clone>
      <clone prototype="lin_tool0">
        <member id="target">target2</member>
      </clone>
    </member>
  </data>

</type>
```

in einer Sequenz zwei Zielposen anzufahren. Der Sequenz-Skill, dem diese Parameter untergeordnet werden, wird dazu später im Detail eingeführt.

Zu beachten ist, dass sich Listen- und Dictionary-Parameter darin unterscheiden, wie neue Parameter hinzugefügt werden. Erbt ein Dictionary-Parameter (`dict`) bereits Einträge seines Prototyps, so werden zusätzlich definierte Parameter ergänzt oder sie überschreiben die vorhandenen gleichnamigen Parameter, falls diese bereits existieren. Erbt hingegen ein Listen-Parameter (`list`) Einträge seines Prototyps, so werden weitere Parameter stets am Ende der Liste angefügt.

4.2.2 Domänenspezifische Sprachelemente

Neben den allgemeinen Sprachelementen wird eine Reihe an domänenspezifischen Parametern definiert. Im MDA 4-Ebenenmodell sind diese Teil des anwendungsunabhängigen Metamodells (M2). Wie eingangs begründet, trennt das vorliegende Skill-Modell nicht zwischen der Modellierung anwendungsspezifischer und anwendungsunabhängiger Sprachelemente der Ebenen M2 beziehungsweise M1. Anwendungen werden daher später auf die gleiche Weise beschrieben, wie die hier aufgeführten Sprachelemente.

Codebeispiel 4.2: Definition einer Stoppbedingung und ihrer Metadaten

```
<type id="control_error_monitor" prototype="monitor">

 <!-- Metadaten zur Instanziierung  -->
 <meta>
   <member id="description">Checks control errors</member>
   <member id="implementation">
     <clone prototype="python">
       <member id="module">pitasc_library.monitors.basic</member>
       <member id="class">ControlErrorMonitor</member>
     </clone>
     <clone prototype="orocos">
       <member id="package">monitors</member>
       <member id="component">ControlErrorMonitor</member>
     </clone>
   </member>
 </meta>

 <!-- Argumente zur Initialisierung -->
 <data>
   <clone id="prefix" prototype="string"/>
   <clone id="coordinates" prototype="string_csv"/>
   <clone id="errors" prototype="float_csv"/>
   <clone id="samples" prototype="int">10</clone>
   <clone id="event" prototype="event">succeeded</clone>
 </data>

</type>
```

Elementare Bausteine und Skills

Zu den domänenspezifischen Parametern zählen zunächst die Repräsentationen der in Kapitel 3 vorgestellten elementaren Bausteine, wie kinematische Elemente, Stoppbedingungen und Skripte. Wie in Abschnitt 4.1.1 beschrieben, sind diese Objekt-Parameter und besitzen demnach Implementierungen, die instanziiert und parametriert werden müssen. Dazu beschreiben Metadaten, wie die Instanziierung zu erfolgen hat und es werden Unterparameter als Argumente für die Initialisierung der erzeugten Objekte definiert. In DSL-Codebeispiel 4.2 ist zur Veranschaulichung die Definition der `control_error` Stoppbedingung aufgeführt. Diese enthält Metadaten für die jeweiligen Implementierungen in C++/OROCOS und Python sowie fünf Unterparameter, die der Initialisierungsfunktion der Stoppbedingung übergeben werden.

Die elementaren Bausteine können nun zur Definition von Skills herangezogen werden. Skills sind Container für elementare Bausteine. Entsprechend ihrer Definition aus Abschnitt 3.5

$$\mathcal{S} = \langle \mathcal{N}, \boldsymbol{\mathcal{KE}}, \boldsymbol{\mathcal{T}}, \boldsymbol{\mathcal{SC}}, \boldsymbol{\mathcal{M}}, \boldsymbol{\mathcal{TR}}, \boldsymbol{\mathcal{S}}_{sub} \rangle \tag{4.1}$$

enthält das Modell eines Skills Listen für diese. DSL-Codebeispiel 4.3 zeigt, wie die Definition von Skills mit der DSL umgesetzt wird. Abschnitt 4.3 demonstriert, wie diese Listen schrittweise mit Elementen gefüllt werden.

Codebeispiel 4.3: Prototyp aller Skills

```
<type id="skill" prototype="object">
  <data>
    <!-- Name des Skills -->
    <type id="skill_name" prototype="string">
      <meta>
          <member id="description">Name of this skill. Must be locally unique.</member>
      </meta>
    </type>

    <!-- Listen fuer elementare Bausteine des Skills -->
    <type id="kinematic_elements" data_type="list:kinematic_element"/>
    <type id="tasks" data_type="list:task"/>
    <type id="scripts" data_type="list:script"/>
    <type id="monitors" data_type="list:monitor"/>

    <!-- Liste an Transitionen -->
    <type id="transitions" data_type="list:transition"/>

    <!-- Liste an Sub-Skills -->
    <type id="skills" data_type="list:skill"/>
  </data>
</type>
```

Die einzelnen Unterarten von Skills (Statechart-, Sequence- und Concurrency-Skill) unterscheiden sich lediglich darin, wie ihre Sub-Skills bei der Instanziierung des Statecharts interpretiert werden. Für ihre Modellierung ist es entsprechend ausreichend, neue Prototypen anzulegen, wobei der Statechart-Skill um einen optionalen Parameter für den Namen des Start-Skills ergänzt wird:

```
<clone id="skill_statechart" prototype="skill">
  <clone id="initial_skill" prototype="string"/>
</clone>
<clone id="skill_sequence" prototype="skill"/>
<clone id="skill_concurrency" prototype="skill"/>
```

Einfache domänenspezifische Parameter

An vielen Stellen des Modells treten domänenspezifische Parameter mit Basisdatentypen auf, beispielsweise Frame- und Event-Namen, bei denen eine Definition als Prototyp hilfreich ist[10]:

```
<clone id="frame" prototype="string"/>
<clone id="event" prototype="string"/>
```

Die Modellierung als Prototyp gegenüber der direkten Verwendung eines Strings erlaubt es, dem String einen definierten Typ zu geben. Dadurch wird eine spezielle Verarbeitung der Parameter bei der Instanziierung oder bei der Nutzung einer grafischen Benutzeroberfläche (GUI) möglich. Referenzen und Werte können automatisch überprüft werden, beispielsweise um die Typen

[10] Der Prototyp `string` definiert einen Parameter mit Prototyp `base` und Datentyp `string`, enthält also bereits die für `base` definierten Metadaten wie einen Beschreibungstext. Ebenso existieren beispielsweise der Prototyp `string_csv` mit Prototyp `base` und Datentyp `csv:string` und analog dazu Prototypen für die verbleibenden Basisdatentypen.

referenzierter Parameter auf Richtigkeit zu prüfen oder um Sonderzeichen in Event-Namen zu unterbinden. Soll mithilfe der GUI eine Referenz oder ein Parameter eines bestimmten Typs erstellt werden, so lassen sich dem Nutzer passende Vorschläge zur Auswahl stellen, die vom bereits vorgestellten `find_models` Algorithmus gefunden werden.

Des Weiteren lassen sich über Metadaten domänenspezifische Eigenschaften modellieren, wie Einschränkungen für die erlaubten Wertebereiche von Parametern (engl. *restrictions*). Für Zahlenwerte können dies obere und untere Grenzen sein, jeweils einschließlich oder ausschließlich des angegebenen Grenzwerts. Ebenso sind Aufzählungstypen möglich (`enum`, von engl. *enumeration*), die nur Werte aus einer im Vorhinein definierten endlichen Menge an Werten erlauben. Ein Beispiel hierfür ist der häufig verwendete Parameter `coordinates`, der die Auswahl einer oder mehrerer Raumachsen für Translationen oder Rotationen erlaubt:

```
<type id="coordinates" prototype="string_csv">
    <meta>
        <clone prototype="restrictions">
            <clone prototype="enum">x, y, z, a, b, c</clone>
        </clone>
    </meta>
</type>
```

Ebenfalls als Aufzählungstyp wird der Parameter `operator` definiert. Der Operator kann Werte wie `less`, `greater`, `greater_equal`, `absolute_greater` usw. annehmen und findet beispielsweise Verwendung bei einem Schwellwertmonitor, für den angegeben wird, ob der Schwellwert zur Auslösung des Events über- oder unterschritten werden soll.

Struktur und Aufbau einer Projektdatei

Neben einzelnen Parametern gehört auch die Modellierung eines gesamten Projekts zu den domänenspezifischen Elementen. Abbildung 4.8 in Abschnitt 4.1.2 stellt dazu das Beispiel eines Parameterbaums dar. Die Wurzel dieses Baums ist der Projektparameter `project`, der drei Untergruppen mit weiteren Parametern besitzt:

```
<clone id="project" prototype="object">
  <clone id="configuration" prototype="dictionary"/>
  <clone id="environment" prototype="dictionary"/>
  <clone id="applications" data_type="dict:skill" prototype="base"/>
</clone>
```

Der Konfigurationsparameter `configuration` enthält sämtliche Komponenten, die zur Ausführungsumgebung gehören sowie deren Parametrierung. Dazu zählen der kinematische Baum, die Scene und der Solver. Die Modellierung der Roboterzelle erfolgt über den `environment` Parameter. Hier sind die eingesetzten Roboter und die zugehörigen Komponenten wie Sensoren und Greifer aufgeführt. Der Dictionary-Parameter `applications` enthält schließlich die Anwendungen des Projekts. Neben der primären Anwendung für den Produktionseinsatz werden hier

häufig Hilfsanwendungen definiert, wie das Zurücksetzen der Roboterzelle auf einen bestimmten Startzustand oder Testroutinen.

Diese Dreiteilung erlaubt bei der Modellierung die Trennung der Ausführungsumgebung, der Roboterzelle sowie der umgesetzten Anwendungen – und vereinfacht damit die Wiederverwendung der einzelnen Teile.

4.3 Komposition und inkrementelle Spezialisierung von Skills

Die in den vorangegangenen Abschnitten vorgestellte DSL und das zugrundeliegende Modell des Parameterbaums sind bewusst einfach und generisch gehalten. Die geringe Anzahl der eingeführten Konzepte und Sprachelemente reicht dennoch dazu aus, die Erstellung, Komposition und Vererbung beliebiger Parameter vorzunehmen. Am deutlichsten werden die Vorteile des Modells bei der Definition von Skills. Um dies zu veranschaulichen, wird im Folgenden eine Reihe an prägnanten Beispielen aufgeführt. Eine Verallgemeinerung hin zu einer umfangreichen Bibliothek von Skills wird im anschließenden Kapitel 5 vorgenommen. Die Skills der *pitasc* Bibliothek bauen dabei analog zum hier vorgestellten Schema aufeinander auf.

Die aufgeführten Beispiele des DSL-Codes veranschaulichen, wie kompakt und einfach die deklarative Definition von Skills ist. Insbesondere ist keine Programmierung von *Glue Code* nötig, um die einzelnen Elemente miteinander zu verknüpfen. Die Komposition der Skills wird komplett mithilfe der DSL vorgenommen. Dafür sind keine Programmierkenntnisse in Sprachen wie C++ oder Python erforderlich. Ebenfalls wird anhand der Beispiele hervorgehoben, wie die einzelnen Skills schrittweise aufeinander aufbauen. Neue Skills werden erstellt, indem vorhandene Skills kopiert und erweitert werden. Vorhandene Skills können so effizient wiederverwendet werden, wobei die Wiederverwendung explizit modelliert wird.[11]

4.3.1 Komposition einfacher Skills

Nachfolgend wird eine Reihe an Skills erstellt, die auf *kartesischen* Feature-Koordinaten basieren. Beginnend mit der Modellierung der Kinematik, werden schrittweise Elemente hinzugefügt, darunter Tasks und Regler, Stoppbedingungen sowie Skripte. Es werden jeweils Beispiele für die Modellierung der einzelnen Teilaspekte genannt. Die entstandene Vererbungshierarchie der hier vorgestellten Skills ist in Abbildung 4.15 aufgeführt. Vererbungsbeziehungen werden dabei

[11] Dieser Abschnitt basiert auf der Publikation von Nägele et al. (2019)

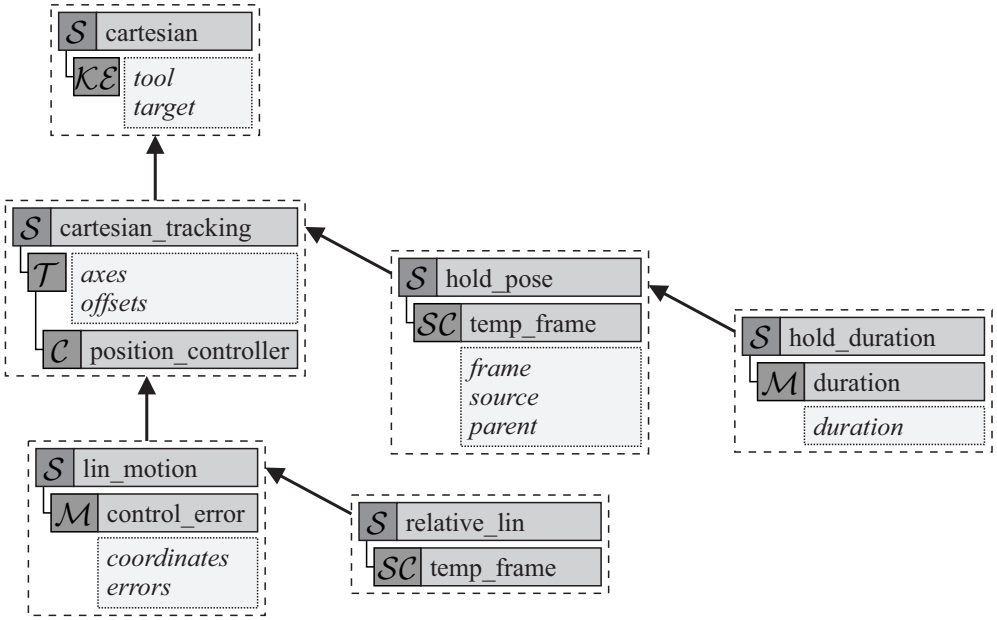


Abbildung 4.15: Beispiele für die Komposition und Vererbung einfacher Skills, die auf einer Modellierung mit kartesischen Feature-Koordinaten aufbauen

mit Pfeilen dargestellt, die Komposition von Elementen mit Linien und einfache Parameter in mit Punktlinien umrahmten Boxen.

Modellierung der Kinematik

Der `cartesian` Skill aus DSL-Codebeispiel 4.4 modelliert die bereits in Abbildung 3.10 dargestellte kinematische Schleife. Er stellt Roboter- und Feature-Koordinaten für die spätere Aufgabenspezifikation bereit.

Die kinematische Schleife $\mathcal{LP}$ besteht dabei aus der Roboter-Kette $\mathcal{CH}_{robot}$, der Feature-Kette $\mathcal{CH}_{feature}$ sowie den zwei Objekt-Ketten $\mathcal{CH}_{tool}$ und $\mathcal{CH}_{target}$. Beide Seiten der Schleife gehen vom Basis-Koordinatensystem O_{base} des Roboters aus und treffen sich im Koordinatensystem O_{tool}. Die Roboter-Kette referenziert die bereits existierende Kette des für diesen Skill gewählten Roboters. Die drei verbleibenden Ketten werden für den Skill neu erstellt. Parametriert werden sie mit in Summe vier Koordinatensystemen. Zwei dieser Koordinatensysteme, `robot.flange` und `robot.base`, werden automatisch vom gewählten Roboter übernommen. Die beiden weiteren Koordinatensysteme `tool` und `target` sind anwendungsspezifische Parameter des Skills, die vom Nutzer angepasst werden können.

Der `cartesian` Skill führt selbst keine Aufgabe aus, ist aber der Ausgangspunkt für viele verschiedene Skills, die auf der Verwendung kartesischer Koordinaten beruhen.

Codebeispiel 4.4: Definition des `cartesian` Skills: Modellierung einer kinematischen Schleife

```
<clone id="cartesian" prototype="skill">

 <!-- Definition der Parameter des Skills -->
 <clone id="tool" prototype="frame"/>
 <clone id="target" prototype="frame"/>

 <member id="kinematic_elements">

  <!-- Hinzufuegen einer kinematischen Schleife -->
  <clone prototype="kinematic_loop">
    <member id="robot_chains">
      <reference reference_id="robot.chain"/>
      <clone id="ch_tool" prototype="cartesian_chain">
        <member id="coordinate_type">object</member>
        <member id="base" reference_id="robot.flange"/>
        <member id="tip" reference_id="tool"/>
      </clone>
    </member>
    <member id="feature_chains">
      <clone id="ch_target" prototype="cartesian_chain">
        <member id="coordinate_type">object</member>
        <member id="base" reference_id="robot.base"/>
        <member id="tip" reference_id="target"/>
      </clone>
      <clone id="ch_feature" prototype="cartesian_chain">
        <member id="coordinate_type">feature</member>
        <member id="base" reference_id="target"/>
        <member id="tip" reference_id="tool"/>
      </clone>
    </member>
  </clone>

 </member>
</clone>
```

Tasks und Regler

Der `cartesian_tracking` Skill aus DSL-Codebeispiel 4.5 erweitert den `cartesian` Skill um eine Aufgabenspezifikation. Dafür werden Roboter- und Feature-Koordinaten mit Zwangsbedingungen belegt. Regler werden zudem mit diesen Zwangsbedingungen verknüpft und definieren so, wie die Zwangsbedingungen erfüllt werden sollen.

Beim `cartesian_tracking` Skill soll das `tool` Koordinatensystem dem `target` Koordinatensystem folgen. Dies ist auch für einzelne Achsen möglich, welche für den jeweiligen Anwendungsfall über den Parameter `axes` angegeben werden. Optional kann ein konstanter relativer Versatz `offset` definiert werden, der einen Eintrag je ausgewählter Achse enthält. Dieser Versatz ist beispielsweise für die Vorpositionierung eines Werkzeugs hilfreich. Über den `prefix` Parameter wird die kinematische Kette ausgewählt, für deren Roboter- oder Feature-Koordinaten die Zwangsbedingungen spezifiziert werden sollen. Dieser Parameter ist für jede kinematische Kette einzigartig. Zuletzt wird den einzelnen zu regelnden Koordinaten jeweils ein Regler zugewiesen. Ein Regler kann dabei für mehrere Koordinaten gleichzeitig zuständig sein. Im Beispiel wird jeweils ein Regler für Positionskoordinaten und Orientierungskoordinaten angegeben und konfiguriert.

Codebeispiel 4.5: Definition des `cartesian_tracking` Skills: Modellierung von Tasks und Reglern

```
<clone id="cartesian_tracking" prototype="cartesian">

 <!-- Definition weiterer Parameter des Skills -->
 <clone id="axes" prototype="string_csv"/>
 <clone id="offsets" prototype="float_csv"/>

 <member id="tasks">

  <!-- Hinzufuegen einer Aufgabenspezifikation -->
  <clone id="tracking_task" prototype="task">
    <member id="prefix" reference_id="kinematic_elements.kinematic_loop.feature_chains.
        ch_feature.prefix"/>
    <member id="coordinates" reference_id="axes"/>
    <member id="desired" reference_id="offsets"/>
    <member id="controller_assignments">
      <clone prototype="controller_assignment">
        <member id="coordinates">x, y, z</member>
        <member id="controller">
          <clone prototype="position_controller">
            <!-- ... Parameter fuer Positionsregler -->
          </clone>
        </member>
      </clone>
      <!-- ... Regler fuer Orientierung -->
    </member>
  </clone>

 </member>
</clone>
```

Stoppbedingungen

Der `cartesian_tracking` Skill besitzt keine Stoppbedingungen und würde demnach für unbegrenzte Zeit laufen. Eine Stoppbedingung definiert, wann ein Skill als erfüllt gilt oder aber ein Fehler aufgetreten ist. In beiden Fällen wird der Skill beendet und eine Transition zu demjenigen Skill ausgeführt, der mit dem ausgelösten Event verknüpft wurde.

Der in DSL-Codebeispiel 4.6 vorgestellte `lin_motion` Skill erweitert den `cartesian_tracking` Skill um eine Stoppbedingung. Die hier verwendete `control_error` Stoppbedingung aus DSL-Codebeispiel 4.2 überwacht Regelgrößen, die mit dem Parameter `coordinates` selektiert werden. Der Skill wird beendet, sobald die Abweichungen der Regelgrößen von den Sollwerten für eine gewisse Anzahl an Regelungszyklen `samples` unterhalb der spezifizierten Schwellwerte `errors` liegen. Durch die Wahl der Zyklenanzahl und Schwellwerte kann demnach eingestellt werden, wie exakt der Skill die Regelgrößen zu erfüllen hat und wie sensitiv er auf Messungenauigkeiten reagiert.

Das DSL-Codebeispiel zeigt zudem einen beispielhaften Einsatz des Skills. Der Greifer wird 10 cm über der zu greifenden Komponente platziert. Die z-Achse sei dabei senkrecht nach oben gerichtet, die Orientierung des Werkzeugs sei in diesem Beispiel nicht eingeschränkt.

Codebeispiel 4.6: Definition des `lin_motion` Skills: Modellierung von Stoppbedingungen

```
<clone id="lin_motion" prototype="cartesian_tracking">

 <member id="monitors">
   <!-- Hinzufuegen einer Stoppbedingung -->
   <clone id="pose_monitor" prototype="control_error_monitor">
     <member id="prefix" reference_id="kinematic_elements.kinematic_loop.feature_chains.
         ch_feature.prefix"/>
     <member id="coordinates" reference_id="axes"/>
     <member id="errors">0.001, 0.001, 0.001, 0.005, 0.005, 0.005</member>
   </clone>
 </member>

</clone>

<!-- Einsatz des lin_motion Skills -->
<clone id="my_lin_motion" prototype="lin_motion">
  <member id="tool">grip_point</member>
  <member id="target">component_1</member>
  <member id="axes">x, y, z</member>
  <member id="offsets">0, 0, 0.1</member>
</clone>
```

Codebeispiel 4.7: Definition des `hold_pose` Skills: Modellierung von Skripten

```
<clone id="hold_pose" prototype="cartesian_tracking">
  <member id="target">hold_temp_frame</member>

  <member id="scripts">
    <!-- Hinzufuegen eines Skriptes -->
    <clone prototype="temp_frame_script">
      <member id="frame" reference_id="target"/>
      <member id="source" reference_id="tool"/>
      <member id="parent" reference_id="robot.base_link"/>
    </clone>
  </member>

</clone>
```

Skripte

Skripte erfüllen Hilfsfunktionen, wie die Kommunikation mit Greifern oder anderen Peripheriegeräten. Aber auch programminterne Funktionen werden von Skripten übernommen, wie das Erzeugen eines temporären Koordinatensystems.

Im Folgenden werden beispielhaft zwei Skills beschrieben, die auf bisher vorgestellten Skills aufbauen und sie jeweils um ein `temp_frame` Skript erweitern. Dieses Skript kopiert ein Koordinatensystem, benennt es und weist gegebenenfalls ein neues Referenzkoordinatensystem zu. Das `temp_frame` Skript bietet eine Vielzahl an Einsatzmöglichkeiten und veranschaulicht so das hohe Maß an Wiederverwendbarkeit innerhalb des Systembaukastens.

Das DSL-Codebeispiel 4.7 zeigt den `hold_pose` Skill. Dieser Skill dient dazu, die Lage des Werkzeugkoordinatensystems beizubehalten, die es beim Starten des Skills besitzt. Dazu kopiert das `temp_frame` Skript das Werkzeugkoordinatensystem `tool` beim Starten des Skills, benennt die Kopie `hold_temp_frame` und weist dieser die Roboterbasis als Referenzkoordinatensystem zu.

Das kopierte Koordinatensystem bewegt sich so nicht mehr mit dem Roboter mit. Der `target` Parameter, den der Skill vom `cartesian_tracking` Skill erbt, wird auf das kopierte Koordinatensystem gesetzt, wodurch der Roboter entsprechend versucht, diese ursprüngliche Lage des Werkzeugkoordinatensystems zu erhalten. Wird der Skill beendet, so wird das temporäre Koordinatensystem automatisch gelöscht.

Das `temp_frame` Skript wird auf gleiche Art verwendet, um den in Abbildung 4.15 gezeigten `relative_lin` Skill zu erstellen, der eine Relativbewegung ausführt. Die vom `lin_motion` Skill geerbten `offsets` beschreiben einen konstanten Versatz, relativ zur Lage des Werkzeugkoordinatensystems beim Starten des Skills. Diese Pose wird mit einer linearen Bewegung angefahren.

Der hier gezeigte Aufbau von Skills in mehreren Schritten kann prinzipiell auch in einem einzigen Schritt erfolgen. Es zeigt sich jedoch, dass bei einem schrittweisen Aufbau viele Abzweigungen innerhalb der Vererbungskette möglich sind. Dadurch werden Dopplungen in der Spezifikation reduziert, wodurch das Modell leichter zu ändern und einfacher zu warten wird. In der Softwareentwicklung ist dies auch als DRY-Prinzip («*don't repeat yourself*», engl. für «wiederhole dich nicht») bekannt. Andererseits beeinflusst die Änderung eines Skills auch die komplette von ihm ausgehende Vererbungshierarchie. Dies mag nicht immer erwünscht sein und macht wiederum eine Änderung oder Wartung derjenigen Skills schwierig, die sich weit oben in der Vererbungshierarchie befinden. Die Tiefe der Vererbungshierarchie sollte demnach stets mit Bedacht gewählt werden.

4.3.2 Komposition von Sequence-Skills

Die bisher beschriebenen Skills stellen einfache Skills mit einer einzigen Aufgabenspezifikation dar. In diesem und den folgenden Abschnitten wird betrachtet, wie mehrere Skills kombiniert werden können.

Die konzeptionell einfachste Variante ist dabei die Skill-Sequenz, deren Sub-Skills nacheinander ausgeführt werden. Das DSL-Codebeispiel 4.8 zeigt den `approach` Skill und seine Nutzung. Dieser beinhaltet als Sub-Skills zwei lineare Bewegungen, die jeweils dieselben Werkzeug- und Zielkoordinatensysteme besitzen. Durch die Angabe eines Versatzes für den ersten Sub-Skill wird es möglich, das Zielkoordinatensystem in zwei Schritten anzufahren. Dies ist beispielsweise dann hilfreich, wenn ein zu greifendes Werkstück mit einer definierten Bewegung von oben her angefahren werden soll, um Kollisionen zu vermeiden.

Im Beispiel wird das Werkzeug zunächst 10 cm über dem Zielkoordinatensystem positioniert und fährt anschließend senkrecht nach unten auf die eigentliche Zielposition. Gegenüber einer

Codebeispiel 4.8: Definition des `approach` Skills: Modellierung einer Skill-Sequenz

```
<clone id="approach" prototype="skill_sequence">
  <clone id="tool" prototype="frame"/>
  <clone id="target" prototype="frame"/>
  <clone id="offsets" prototype="float_csv"/>

  <member id="skills">
    <!-- Hinzufuegen von Sub-Skills -->
    <clone prototype="lin_motion">
      <member id="tool" reference_id="tool"/>
      <member id="target" reference_id="target"/>
      <member id="offsets" reference_id="offsets"/>
    </clone>
    <clone prototype="lin_motion">
      <member id="tool" reference_id="tool"/>
      <member id="target" reference_id="target"/>
    </clone>
  </member>
</clone>

<!-- Nutzung des approach Skills -->
<clone id="my_approach" prototype="approach">
  <member id="tool">grip_point</member>
  <member id="target">component_1</member>
  <member id="offsets">0, 0, 0.1, 0, 0, 0</member>
</clone>
```

Codebeispiel 4.9: Verwendung des `screwdriving` Skills: Modellierung von Transitionen für Statechart-Skills

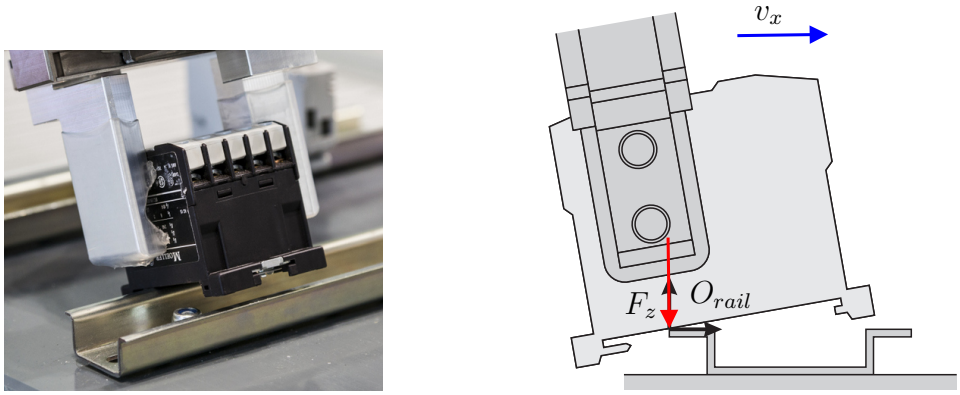
```
<clone id="screw" prototype="screwdriving">
  <!-- Parameter -->
  <!-- ... -->

  <!-- Hinzufuegen von Transitionen -->
  <member id="transitions">
      <clone prototype="transition">
          <member id="event">next</member>
          <member id="target">screw</member>
      </clone>
      <clone prototype="transition">
          <member id="event">finished</member>
          <member id="target">place_screwdriver</member>
      </clone>
  </member>
</clone>
```

direkten Verwendung von zwei `lin_motion` Skills wird die Anzahl an anzugebenden Parametern und in gleichem Maße die Anzahl an zu schreibenden Zeilen DSL-Code fast halbiert. Es tritt keine Dopplung bei der Angabe von Werkzeug- und Zielkoordinatensystemen auf, wodurch die Fehleranfälligkeit reduziert und die Wartbarkeit verbessert wird.

4.3.3 Komposition von Statechart-Skills

Nicht immer lässt sich ein Montageprozess über eine sequenzielle Verkettung von Skills darstellen, insbesondere wenn man die Behandlung von Fehlerfällen mit einbezieht. Statechart-Skills bieten deshalb die Möglichkeit, über eine Zustandsmaschine auch komplexere Programmlogiken



(a) Hutschienenbestückung

(b) Richtungen für Kraft und Geschwindigkeit des `slide` Skills

Abbildung 4.16: Verwendung des `slide` Skills für die Montage von Hutschienenelementen

darzustellen. In Abgrenzung zu Skill-Sequenzen, bei denen Sub-Skills in einer einfachen Abfolge durchlaufen werden, definieren bei Statecharts Transitionen beliebige Übergänge von Skills auf andere Skills.

In Abbildung 3.13 wurde bereits das Statechart einer Schraubapplikation vorgestellt. Das DSL-Codebeispiel 4.9 zeigt die dazugehörige Angabe der Transitionen für den Schraub-Skill. Das Event `next` tritt auf, wenn der Schraub-Skill eine Verschraubung abgeschlossen hat, aber noch weitere Verschraubungen ausgeführt werden sollen. Entsprechend verweist die für dieses Event definierte Transition wieder zurück auf den Schraub-Skill. Ein Skript setzt bei jedem Durchlauf das neue Zielkoordinatensystem des Schraub-Skills. Wurden schließlich alle Verschraubungen erfolgreich ausgeführt, so tritt das Event `finished` auf und die Transition zum Skill `place_screwdriver` wird ausgeführt. Dieser bringt den Schrauber zurück in die Aufnahmevorrichtung.

4.3.4 Komposition von Concurrency-Skills

Für die Programmierung von komplexen Montageprozessen ist häufig eine Kombination aus verschiedenen Reglern für Positions-, Geschwindigkeits- und Kraftregelung nötig. Abbildung 4.16 zeigt beispielsweise eine automatisierte Hutschienenbestückung. Der Roboter drückt dabei das gegriffene Hutschienenelement mit einer geringen Kraft F_z auf die Hutschiene und bewegt sich mit der Geschwindigkeit v_x vorwärts, um das Hutschienenelement in die Krempe der Hutschiene einzuhängen. Für diese Bewegung ist eine Kombination aus Kraftregelung (in z-Richtung), Geschwindigkeitsregelung (in x-Richtung) und Positionsregelung (verbleibende Freiheitsgrade)

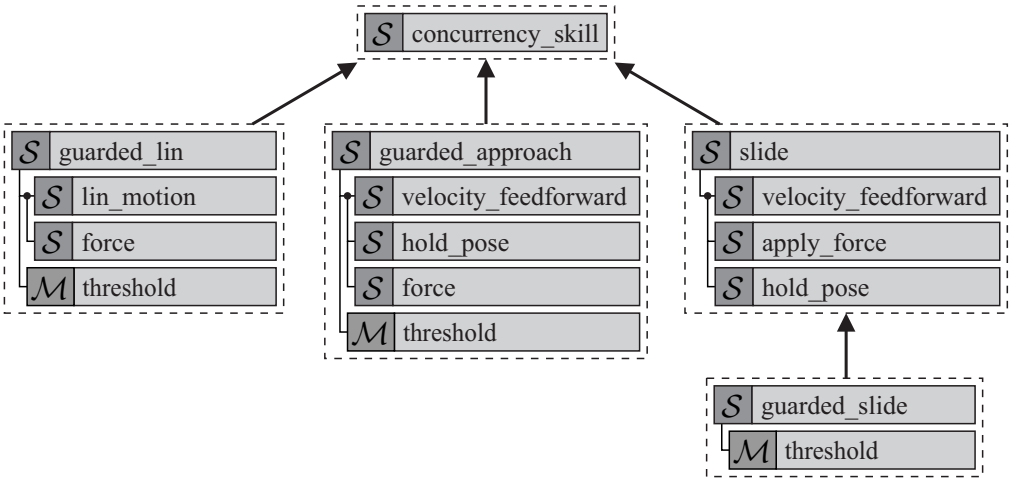


Abbildung 4.17: Beispiele für Concurrency-Skills

nötig. Concurrency-Skills ermöglichen eine Modellierung solcher Montage-Skills, durch Kombination einfacher Skills.

Ein Concurrency-Skill führt alle seine Sub-Skills gleichzeitig aus. Dies geschieht mit einer festen Priorisierung der Sub-Skills, um mögliche Konflikte aufzulösen, wobei die Skills entsprechend ihrer Reihenfolge in der Liste der Sub-Skills priorisiert werden. Im Folgenden werden die vier Concurrency-Skills aus Abbildung 4.17 vorgestellt. Dabei werden einige Skills zur Geschwindigkeits- und Kraftregelung vorweggenommen, die in Kapitel 5 hergeleitet werden.

Nicht jeder Sub-Skill muss eine Aufgabe spezifizieren. Eine rein kinematische Modellierung kann auch sinnvoll sein, um Feature-Werte für Stoppbedingungen oder Skripte zur Verfügung zu stellen, ohne die zugehörigen Features aktiv zu regeln. Ein Beispiel hierfür ist der `guarded_lin` Skill. Dieser Skill kann bei einer Handhabungsaufgabe genutzt werden, wenn positionsgeregelt eine Zielpose angefahren und dabei auf eine möglicherweise auftretende Kollision reagiert werden soll. Dazu wird ein `lin_motion` Skill um einen `force` Skill ergänzt. Dieser enthält selbst keine Aufgabenspezifikation, liefert aber Kraftmesswerte, die in ein Koordinatensystem von Interesse transformiert werden. Eine `threshold` Stoppbedingung beobachtet einen dieser Kraftmesswerte und löst ein Event aus, sobald ein gewisser Schwellwert über- oder unterschritten wurde.

Ähnlich dazu ist der `guarded_approach` Skill strukturiert. Dieser führt eine Suchbewegung aus, mit der ein Kontakt zwischen zwei Werkstücken (oder Werkzeug und Werkstück) hergestellt werden kann. Kombiniert wird ein geschwindigkeitsgeregelter `velocity_feedforward` Skill, der den Roboter in die Vorschubrichtung der Suchbewegung antreibt, und ein `hold_pose` Skill, der die verbleibenden Freiheitsgrade auf ihren ursprünglichen Werten hält. Ein `force` Skill und

eine mit diesem verbundene `threshold` Stoppbedingung führen zu einem Beenden des Skills bei Auftreten des Kontakts.

Ist ein erster Kontakt zwischen den Werkstücken in eine Achsrichtung hergestellt, so kann der `slide` Skill aus Abbildung 4.16b zum Aufbau von Kontakten in weitere Richtungen herangezogen werden. Auch hier wird mithilfe eines `velocity_feedforward` Skills eine Bewegung ausgeführt. Gleichzeitig übt ein `apply_force` Skill eine Kraft aus, um den zuvor hergestellten Kontakt der Werkstücke aufrecht zu erhalten. Dieser Skill besitzt noch keine Stoppbedingung. Der darauf aufbauende `guarded_slide` Skill fügt deshalb beispielsweise eine `threshold` Stoppbedingung hinzu, die auf eine Kraft in Bewegungsrichtung reagiert.

Zusammenfassung

Das in diesem Kapitel vorgestellte Skill-Modell erlaubt die Parametrierung, Komposition und Wiederverwendung sowohl einzelner Programmbausteine, Skills als auch ganzer Anwendungen. Da all diese Elemente einheitlich als Parameter modelliert werden, kann die Komposition und Vererbung in einem generischen Parameterbaum mit sehr wenigen, einfachen Operationen erfolgen. Für die Modellierung der Wiederverwendung wurde ein prototypbasierter Vererbungsmechanismus ausgewählt, der es zudem einfach macht, auf bestehenden Elementen aufzubauen.

Eine domänenspezifische Sprache wird eingeführt, mithilfe derer Skills und Anwendungen ohne Programmierung von Quelltext in einer Universalsprache wie Python oder C++ erstellt und parametriert werden können. Durch die generische Art des Parameterbaums kann die DSL mit wenigen allgemeinen Sprachelementen definiert werden. Darüber hinaus erfolgen sowohl die Definition domänenspezifischer Sprachelemente, die Modellierung von Skills als auch deren Parametrierung und Einsatz auf gleiche Weise. So werden Grenzen zwischen anwendungsspezifischen und -unabhängigen Teilen vermieden, wodurch die Verallgemeinerung anwendungsspezifischer Modelle und umgekehrt der Einsatz von allgemein gestalteten Bausteinen in konkreten Anwendungen erleichtert werden.

Die im letzten Abschnitt vorgestellte inkrementelle Herleitung von Skills zeigt bereits einen kleinen Ausschnitt der realisierten Skill-Bibliothek. Sie führt beispielhaft das Schema ein, nach dem im folgenden Kapitel die Erstellung einer umfassenden Bibliothek an Skills vorgenommen wird.

5 Evaluation

In diesem Kapitel wird das vorgestellte Skill-Modell auf industrielle Montageprozesse angewendet und anhand dieser evaluiert. Gerade bei Montageprozessen stoßen klassische Positions- und Bahnsteuerungen, die auf eine hohe Wiederholgenauigkeit abzielen, an Grenzen, da sie die dort auftretenden Kontaktkräfte nicht explizit berücksichtigen. Verfahren wie der iTaSC Formalismus erlauben es hingegen, Kraft-, Geschwindigkeits- und Positionsregelung in einer Aufgabenspezifikation zu vereinen und damit selbst komplexe Prozesse zu modellieren. Es stellt sich hier jedoch die Frage, wie man Aufgabenspezifikationen möglichst *effizient* definieren kann, insbesondere im Hinblick auf die in der Montage vorherrschende Varianten- und Prozessvielfalt.

Diese Arbeit geht daher der Fragestellung nach, wie kraftgeregelte Montageprozesse modelliert werden können, um neue Bauteilvarianten, Prozesse und Hardwarekomponenten mit einem hohen Maß an Wiederverwendbarkeit abzubilden und insbesondere, wie sich dabei die Wiederverwendung modellieren lässt. In den vorangegangenen Kapiteln wird dazu ein Skill-Modell vorgeschlagen, das dem Baukastenprinzip folgt. Um Aufgabenspezifikationen nach dem iTaSC Formalismus effizient beschreibbar zu machen, werden Skills und deren Unterbausteine wie Stoppbedingungen, Skripte, Regler usw. als deklarative, modulare Programmierelemente eingeführt. Durch Komposition dieser Grundbausteine können Skills flexibel und effizient zusammengestellt werden. Ein Vererbungsmechanismus ermöglicht zudem die inkrementelle Erstellung von Skills mit einem hohen Maß an Wiederverwendung.

Dieses Kapitel evaluiert, inwieweit das vorgeschlagene *pitasc* Skill-Modell die folgenden Teilziele erreicht:

- *Das Baukastensystem erlaubt die Erstellung einer umfassenden* ***Bibliothek an Skills*** *und anderen Bausteinen, die sich zu neuen, spezialisierteren Skills kombinieren lassen (Abschnitt 5.1).*
- *Es ist möglich, komplexe Montage-Skills auf Basis einfacher Bausteine zu erstellen und damit* ***kraftgeregelte Montageaufgaben umzusetzen*** *(Abschnitt 5.2) und* ***robust auszuführen*** *(Abschnitt 5.3).*

- *Neue Bauteilvarianten und Prozesse oder der Wechsel von Hardwarekomponenten sind durch wenige Anpassungen von Parametern oder zumindest mit einem* ***hohen Grad an Wiederverwendung*** *bestehender Elemente umsetzbar (Abschnitt 5.4).*

Die nachfolgenden Abschnitte beschreiben Evaluationskriterien zu den genannten Teilzielen und nehmen eine Bewertung dieser vor. Zur experimentellen Überprüfung werden zehn Anwendungen betrachtet, die in Tabelle 5.1 aufgeführt sind. Alle Anwendungen spiegeln Anfragen von Industrieunternehmen wider, repräsentieren also einen realen Bedarf nach neuartigen Lösungen. Sie wurden in über neun Aufbauten mit verschiedenen Hardwarekomponenten umgesetzt. Abbildung 5.1 zeigt sechs der Aufbauten, drei der Anwendungen werden in Abschnitt 5.2 im Detail vorgestellt.

5.1 Erstellung einer umfassenden Skill-Bibliothek

Mithilfe des vorgeschlagenen Skill-Modells wird in diesem Abschnitt eine Bibliothek an Skills, Skripten und Stoppbedingungen erstellt, die als Basis für die Anwendungen der folgenden Abschnitte dient. Es soll dabei evaluiert werden, inwieweit die Skills der Bibliothek aufeinander aufgebaut werden können und inwieweit die Skills erweiterbar sind. Alle beschriebenen Skills werden in mindestens einer der zehn exemplarischen Anwendungen eingesetzt. Eine Ausnahme davon bildet der `cylindrical`-Skill, der zur Vollständigkeit ergänzt wird.

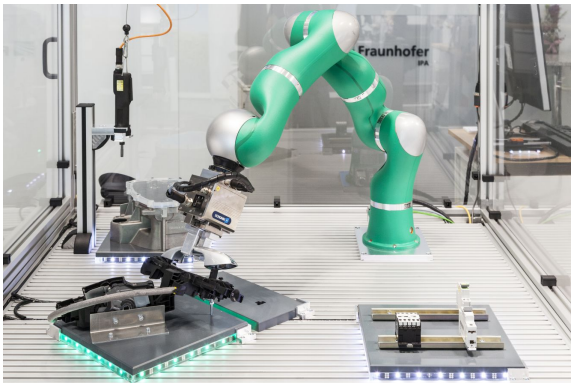
Als Maß dafür, wie sehr die Skills aufeinander aufbauen, ist die Tiefe der entstandenen Skill-Hierarchie ausschlaggebend. Diese entsteht zum einen durch die inkrementelle Vererbung und Erweiterung von Skills, andererseits auch durch die Komposition von Skills aus Sub-Skills. Zu beachten ist, dass die Hierarchie dabei nicht künstlich vertieft wird, indem überflüssige Zwischenschritte eingeführt werden. Daher ist auch die Breite der Skill-Hierarchie ausschlaggebend, die sich dadurch zeigt, dass Skills jeder Hierarchiestufe Ausgangspunkte für weitere Skills bilden, sofern sie nicht bereits einen eigenen Verwendungszweck besitzen. Zur Erstellung der Skills werden verschiedene Skripte und Stoppbedingungen benötigt. Diese werden zunächst kurz aufgeführt.

5.1.1 Skripte

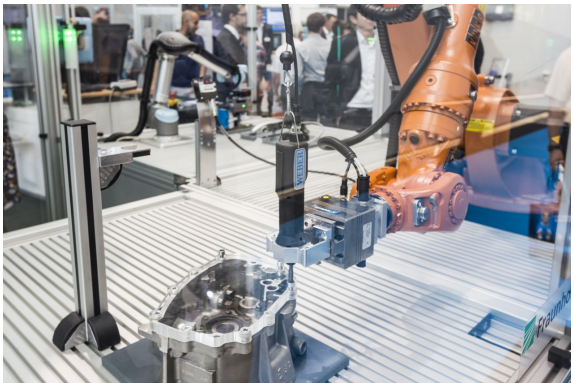
Tabelle 5.2 führt die wichtigsten Kategorien von Skripten ein und nennt jeweils eine Auswahl an Beispielen, die für den späteren Aufbau der Skill-Bibliothek relevant sind.

Tabelle 5.1: Beschreibung und Kategorisierung der exemplarischen Anwendungen

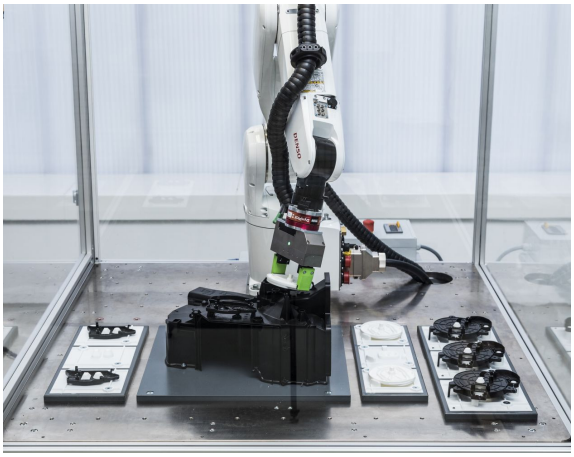
Anwendung	Beschreibung (Materialpaarung)	Prozesskategorie
Hutschienenmontage	Befestigen von Hutschienenelementen auf einer Hutschiene (Kunststoff + Metall)	Schnappverbindung
Plastikteil aufstecken	Aufschieben eines Plastikbauteils auf ein anderes Plastikbauteil (Kunststoff + Kunststoff)	Ineinanderschieben
Verschrauben	Einführen des Bits in den Schraubenkopf und Nachführen des Schraubers (Metall + Metall)	Schrauben
Autotürgriff montieren	Einführen eines Autotürgriffs in das türseitige Kunststoffbauteil (Kunststoff + Kunststoff)	Einsetzen
Blech ausrichten	Ausrichten eines Blechs am Anschlag einer Werkzeugmaschine (Metall + Metall)	(keine Montage i. e. S.)
Gehäuseteil montieren	Zusammenstecken zweier Gehäuseteile aus Kunststoff (Kunststoff + Kunststoff)	Einsetzen
Kabel verlegen	Stecken von konfektionierten Schaltdrähten (Metall/Kunststoff + Metall/Kunststoff)	Ineinanderschieben
Kabelbaum stecken	Montage eines kleinen Kabelbaums mit fünf Steckern (Kunststoff + Kunststoff)	Ineinanderschieben + Schnappverbindung
Stirnrad aufschieben	Aufschieben von Stirnrädern auf eine Welle (Metall + Metall)	Ineinanderschieben
USB-Stecker einführen	Einführen eines USB-Speichersticks oder -Kabels in einen Computer (Metall + Metall/Kunststoff)	Ineinanderschieben



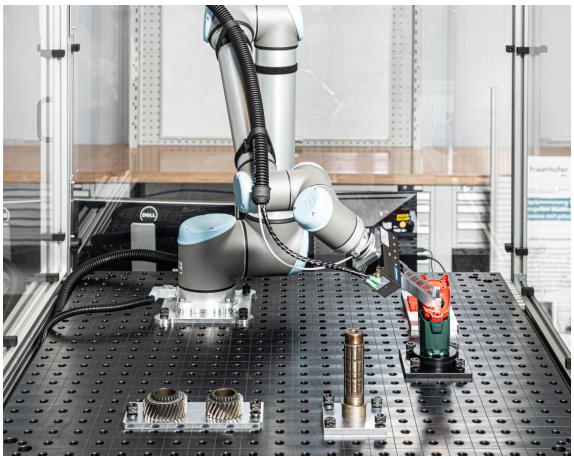
(a) Hutschienenbestückung, Montage eines Autotürgriffs und Verschrauben (KUKA LBR4+)



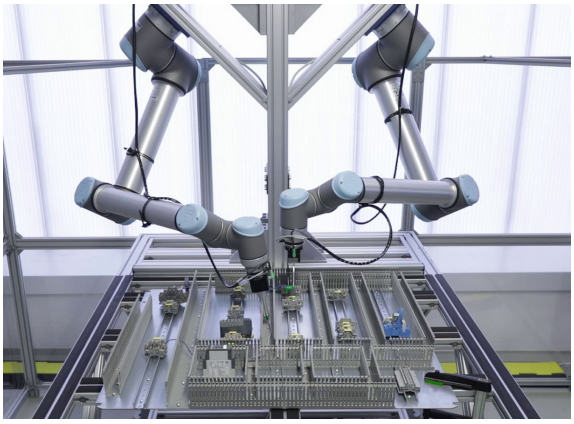
(b) Verschrauben (KUKA KR16)



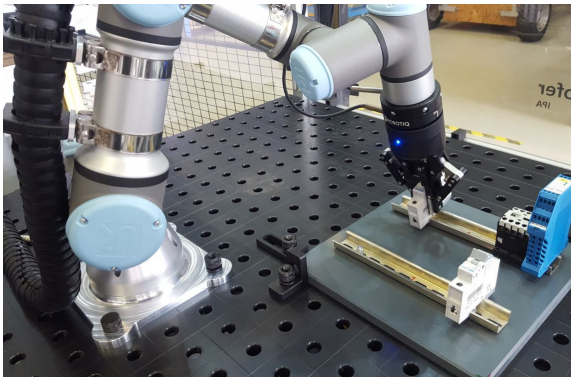
(c) Aufstecken eines Plastikteils (Denso VS-087)



(d) Montage eines Gehäuseteils und Aufschieben eines Stirnrads (Universal Robots UR10)



(e) Stecken von Schaltdrähten (zwei UR10)



(f) Hutschienenbestückung (UR3)

Abbildung 5.1: Versuchsaufbauten, die das vorgestellte Skill-Modell verwenden

Tabelle 5.2: Kategorien und Beispiele häufig genutzter Skripte

Sollwertgenerierung	
`sine_modifier`	Modifiziert Sollwerte basierend auf einer Sinuskurve
`step_modifier`	Modifiziert Sollwerte schrittweise nach jeder Ausführung
`random_modifier`	Modifiziert Sollwerte basierend auf Zufallswerten
`relative_modifier`	Addiert konstanten Versatz auf die vorhandenen Sollwerte
Kinematischer Baum	
`set_frame`	Erstellt ein Koordinatensystem und weist eine Pose zu
`copy_frame`	Kopiert ein existierendes Koordinatensystem
`remove_frame`	Löscht ein existierendes Koordinatensystem
`temp_frame`	Kopiert temporär ein existierendes Koordinatensystem
ROS Services und Actions	
`action_caller`	Prototyp für Skripte, die ROS Actions aufrufen
`service_caller`	Prototyp für Skripte, die ROS Services aufrufen
`call_empty_service`	Ruft einen ROS Service vom Typ `std_msgs/Empty` auf
`call_trigger_service`	Ruft einen ROS Service vom Typ `std_msgs/Trigger` auf
Datenaufzeichnung	
`control_error_publisher`	Publiziert Regelabweichungen über ein ROS Topic
`measurement_publisher`	Publiziert Messwerte über ein ROS Topic
`measurement_logger`	Schreibt Messwerte in eine Datei
Hardwarekomponenten	
`set_digital_output`	Setzt einen digitalen Ausgang der Robotersteuerung
`wsg50_homing`	Führt eine Referenzfahrt des Greifers durch
`wsg50_position`	Setzt den Greifer auf eine bestimmte Weite
`wsg50_grip`	Führt eine Greifbewegung aus
`wsg50_release`	Öffnet den Greifer bis auf eine bestimmte Weite
`wsg50_set_force`	Setzt die Greifkraft
`tare_kms40`	Tariert den Kraft-Momenten-Sensor

Sollwertgenerierung: Skripte können auf verschiedene Weise Sollwerte modifizieren. Dies kann in Form einer Sinuskurve geschehen (beispielsweise um in wechselnde Richtungen Kräfte aufzubringen), in festen Schritten (um ein Raster abzufahren), durch Zufallswerte (um zu Testzwecken Lagetoleranzen zu simulieren) oder durch einen konstanten Versatz (um eine Relativbewegung im Achsraum auszuführen).

Kinematischer Baum: Um kartesische Relativbewegungen auszuführen oder um beispielsweise eine Pose zu halten, muss beim Start der entsprechenden Skills die Pose des Werkzeugkoordinatensystems gespeichert werden. Dies geschieht im kinematischen Baum. Über Skripte lassen sich dort Koordinatensysteme erstellen, kopieren und löschen. In vielen Fällen werden die Koordinatensysteme nach dem Beenden des Skills nicht weiter benötigt. Sie lassen sich daher auch temporär erzeugen und automatisch löschen.

ROS Services und Actions: Für die Kommunikation mit einer übergeordneten Zellsteuerung oder beispielsweise mit einer Software zur Bildverarbeitung werden spezifische Skripte geschrieben. Existieren bereits Schnittstellen für ROS, so kann eine sehr einfache Integration mithilfe von ROS Services und Actions erfolgen, für die entsprechende Prototypen für Skripte vorhanden sind. Diese übernehmen unter anderem die Überprüfung, ob der aufzurufende Service bereits angeboten wird und ermöglichen es, fehlgeschlagene Aufrufe automatisch bis zu einer maximalen Anzahl an Versuchen zu wiederholen. Für häufige ROS Nachrichtentypen gibt es fertige, direkt parametrierbare Skripte.

Datenaufzeichnung: Häufig kann es von Interesse sein, bei der Montage gemessene Kräfte zu untersuchen und zur Qualitätssicherung zu nutzen. Zur Datenaufzeichnung lassen sich mit Skripten Messwerte (inklusive transformierter Kraftwerte) und Regelabweichungen in eine Log-Datei abspeichern. Mit diesen Skripten wurden die Daten für die Evaluierung in Abschnitt 5.3 aufgezeichnet.

Hardwarekomponenten: Für die Ansteuerung von Hardwarekomponenten kann im einfachsten Fall ein Ausgang der Robotersteuerung gesetzt werden. Für umfangreiche Schnittstellen werden spezifische Skripte benötigt. Gerade bei elektrischen Greifern sind die Schnittstellen und Funktionalitäten je Hersteller und Produkt sehr unterschiedlich. In Tabelle 5.2 ist beispielhaft der Greifer WSG50 aufgeführt. Ebenfalls aufgeführt ist ein Skript zum Tarieren eines KMS40 Kraft-Momenten-Sensors. Nach gleichem Schema existieren Skripte zur Ansteuerung weiterer Hardwarekomponenten, die in den untersuchten Anwendungsbeispielen eingesetzt werden.

Tabelle 5.3: Kategorien und Beispiele häufig genutzter Stoppbedingungen und ihre jeweiligen Auslösekriterien

Programmlogik	
`duration`	Nach einer gewissen Zeit
`cycle_counter`	Nach einer gewissen Anzahl Regelungszyklen
`execution_counter`	Nach einer gewissen Anzahl Skill-Ausführungen
`script_result`	Wenn ein Skript ein gewisses Ergebnis liefert
`sync`	Wenn alle enthaltenen Stoppbedingungen gleichzeitig auslösen
Messwerte	
`control_error`	Regelabweichungen kleiner als ein Schwellwert
`threshold`	Messwert verletzt (über- oder unterschreitet) einen Schwellwert
`topic_threshold`	Wert in einer ROS Message verletzt einen Schwellwert
Ein- und Ausgänge	
`digital_io`	Digitaler Ein- oder Ausgang der Robotersteuerung wird gesetzt
`user_confirm`	Benutzer bestätigt Meldung
`rosparam`	ROS Parameter entspricht einem gewünschten Wert
`empty_service`	Erstellter ROS Service vom Typ `std_msgs/Empty` wird aufgerufen
`trigger_service`	Erstellter ROS Service vom Typ `std_msgs/Trigger` wird aufgerufen
Geometrie	
`distance`	Abstand zweier Koordinatensysteme verletzt einen Schwellwert
`relative_distance`	Koordinatensystem bewegt sich um eine gewisse Distanz
`frame_in_box`	Koordinatensystem befindet sich in einer gewissen Region

5.1.2 Stoppbedingungen

Tabelle 5.3 führt die wichtigsten Stoppbedingungen der Skill-Bibliothek auf und beschreibt, wann diese ein Event und damit gegebenenfalls eine Transition auslösen.

Programmlogik: Die `duration` Stoppbedingung löst nach einer gegebenen Zeit aus und kann beispielsweise verwendet werden, um für diese Zeit eine Kraft auszuüben oder die Position zu halten. Alternativ kann mit der `cycle_counter` Stoppbedingung die Anzahl an durchlaufenen Regelungszyklen herangezogen werden. Der `execution_counter` registriert, wie oft der Skill, dem die Stoppbedingung zugewiesen ist, ausgeführt wird. Damit lassen sich einfach Schleifen erstellen. Die `script_result` Stoppbedingung überprüft das Ergebnis eines Skripts

nach dessen Ausführung und kann so beispielsweise auf Fehlerfälle reagieren. Eine `sync` Stoppbedingung enthält weitere Stoppbedingungen und löst erst aus, wenn diese gemeinsam auslösen. Auf diese Weise lassen sich mehrere Kriterien miteinander verknüpfen.

Messwerte: Weitere Stoppbedingungen überwachen Regelabweichungen (`control_error`), Feature-Koordinaten (`threshold`) und Messwerte aus ROS Topics (`topic_threshold`). Diese Stoppbedingungen stellen Schwellwerte dar, bei deren Über- oder Unterschreitung ein Event ausgelöst wird. Über Regelabweichungen kann betrachtet werden, ob eine Feature-Koordinate ihre Sollwerte ausreichend genau erreicht hat oder einhält. Kraftschwellwerte werden insbesondere für Suchbewegungen verwendet, die beim Überschreiten des eingestellten Schwellwerts beendet werden.

Ein- und Ausgänge: Die digitalen Ein- und Ausgänge der Robotersteuerung können mit der `digital_io` Stoppbedingung überwacht werden. So kann auf Signale von Peripheriekomponenten reagiert werden, wie beispielsweise dem Steuergerät eines elektrischen Schraubers, das ein Erreichen des gewünschten Anziehdrehmoments meldet. Soll hingegen der Nutzer um Bestätigung gebeten werden, kann über die `user_confirm` Stoppbedingung ein Hinweisfenster geöffnet werden, das vom Nutzer quittiert werden muss. Die Stoppbedingungen `empty_service` und `trigger_service` erstellen jeweils einen entsprechenden ROS Service. Diese können beispielsweise von einer Zellsteuerung aufgerufen werden, die so Einfluss auf die Programmausführung nehmen kann. Die `rosparam` Stoppbedingung liest einen ROS Parameter ein und überprüft, ob er dem gewünschten Wert entspricht. Auch hiermit lässt sich auf einfachem Wege die Programmausführung von externen Komponenten beeinflussen.

Geometrie: Darüber hinaus gibt es Stoppbedingungen, die geometrische Beziehungen überwachen. Mit `distance` wird der Abstand zweier Koordinatensysteme zueinander überprüft. Es können eine Achse und ein zugehöriger Schwellwert angegeben werden. Soll überprüft werden, wann ein Koordinatensystem sich gegenüber seiner ursprünglichen Pose um einen gewissen Versatz bewegt hat, kann die `relative_distance` Stoppbedingung herangezogen werden. Diese Stoppbedingung wird beispielsweise dazu verwendet, geschwindigkeitsgeregelte Relativbewegungen auszuführen. Zuletzt kann mit `frame_in_box` geprüft werden, ob sich beispielsweise ein Werkzeugkoordinatensystem innerhalb einer gewissen Region um ein anderes Koordinatensystem befindet.

Auch diese Liste ist nicht abschließend. Der Systembaukasten lässt sich sehr einfach um weitere allgemeine oder anwendungsspezifische Stoppbedingungen erweitern. Auch lässt sich auf den vorhandenen Stoppbedingungen aufbauen.

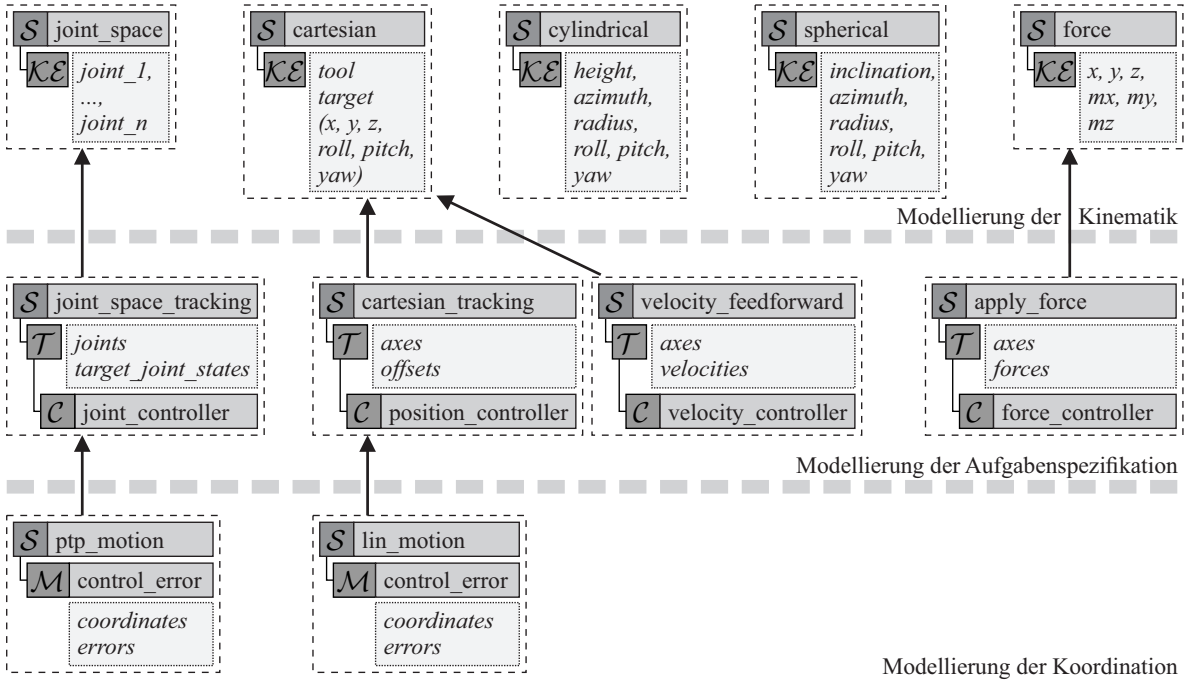


Abbildung 5.2: Hierarchie der wichtigsten Skill-Familien

5.1.3 Skills

Während im vorangegangenen Kapitel 4 unter Verwendung der DSL bereits einige aufeinander aufbauende Skills erstellt wurden, erfolgt im Folgenden die Verallgemeinerung hin zu einer Bibliothek. In Abschnitt 5.2 wird anschließend die Anwendbarkeit der Skill-Bibliothek zur Programmierung von Montageprozessen demonstriert.

Abbildung 5.2 zeigt die wichtigsten Skill-Familien, die als Basis der Skill-Bibliothek dienen. Mit Pfeilen werden dabei Vererbungsbeziehungen dargestellt, mit einfachen Linien die Komposition von Elementen. Die wichtigsten Parameter werden in mit Punktlinien umrahmten Boxen aufgeführt. Der Aufbau der Skills folgt den in Abschnitt 4.3.1 beschriebenen Schritten. Die Skills der ersten Ebene modellieren die Kinematik und damit die Feature-Koordinaten. Dazu zählen Gelenkkoordinaten, kartesische Koordinaten, Zylinder- und Kugelkoordinaten sowie eine virtuelle Feder für kraftgeregelte Skills. Anschließend wird die Aufgabenspezifikation durch das Hinzufügen von Tasks und Reglern modelliert sowie die Koordination durch die Definition von Stoppbedingungen. Dies ist beispielhaft für Gelenkraum- und kartesische Skills dargestellt. Analog dazu existieren gleichartige Skills für Zylinder- und Kugelkoordinaten. Angemerkt sei, dass die dargestellte Struktur der Bibliothek in drei Ebenen als Gliederung für Anwender dient. Aus Sicht des Modells befinden sich alle Skills in einer Baumstruktur und es wird konzeptionell nicht zwischen Ebenen unterschieden.

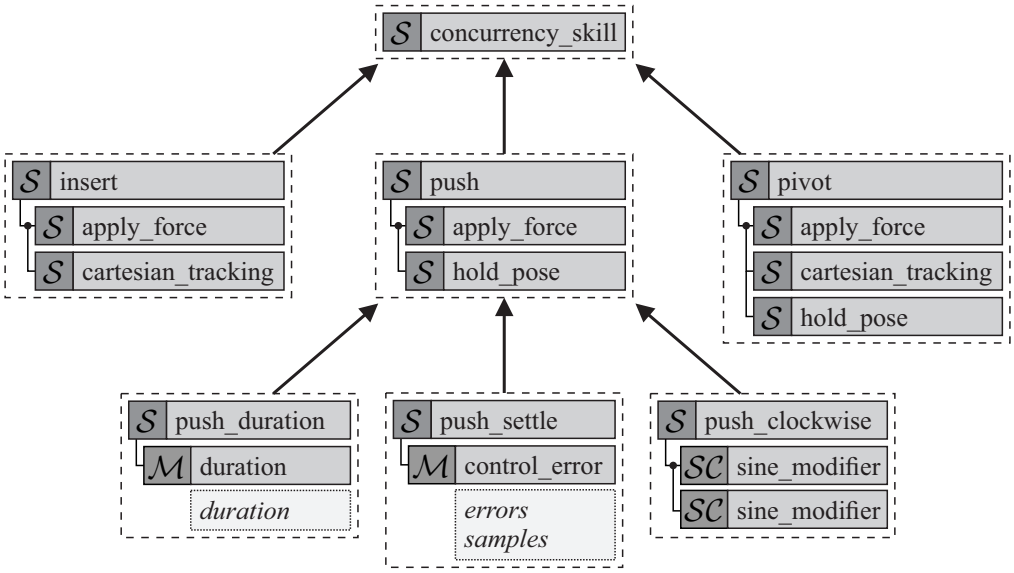


Abbildung 5.3: Komposition und Vererbung kraftgeregelter Skills

Mit den `ptp_motion` und `lin_motion` Skills finden sich in der Bibliothek Bewegungsbefehle, die auch bei der klassischen Roboterprogrammierung eingesetzt werden. Gleichzeitig wird aber ermöglicht, auf Skills mit Feature-Koordinaten weiterer Koordinatendarstellungen zurückzugreifen oder neue zu erstellen. Hinzu kommt die Flexibilität, den Feature-Koordinaten verschiedene Regler zuzuordnen, über Stoppbedingungen unterschiedliche Ziel- oder Fehlerkriterien zu definieren oder die Skills mit Skripten um weitere Funktionen zu ergänzen. In Abbildung 4.15 wurden dazu bereits weitere Skills vorgestellt, die auf kartesischen Feature-Koordinaten basieren. Zuletzt besteht die Möglichkeit, die genannten Skills als Basis für Sequence-, Statechart- und Concurrency-Skills zu verwenden. Während erstere häufig bei der Erstellung von Anwendungen zur Modellierung des Programmablaufs herangezogen werden, eignen sich Concurrency-Skills insbesondere dazu, Positions-, Geschwindigkeits- und Kraftregelung in einem Skill zu vereinen.

Einige häufig verwendete Concurrency-Skills wurden bereits in Abbildung 4.17 aufgeführt. Abbildung 5.3 enthält eine weitere Anzahl an kraftgeregelten Concurrency-Skills der Skill-Bibliothek. Mit dem `insert` Skill lassen sich Einführvorgänge durchführen. Typischerweise wird eine Kraft in Einführrichtung aufgebracht, gleichzeitig werden die Querkräfte minimiert. Die Orientierung wird über ein Zielkoordinatensystem vorgegeben. Der Skill definiert noch keine Stoppbedingung, da hier große Unterschiede in der Verwendung auftreten. Die typischerweise eingesetzten Stoppbedingungen betrachten die Distanz zwischen Werkstückmerkmalen oder die relative Distanz, die das Werkstück beim Einführvorgang zurückgelegt hat. Dazu wird das Überschreiten eines Kraftschwellwerts in Einführrichtung häufig als Indiz dafür gewertet, dass das Werkstück vollständig eingeführt wurde.

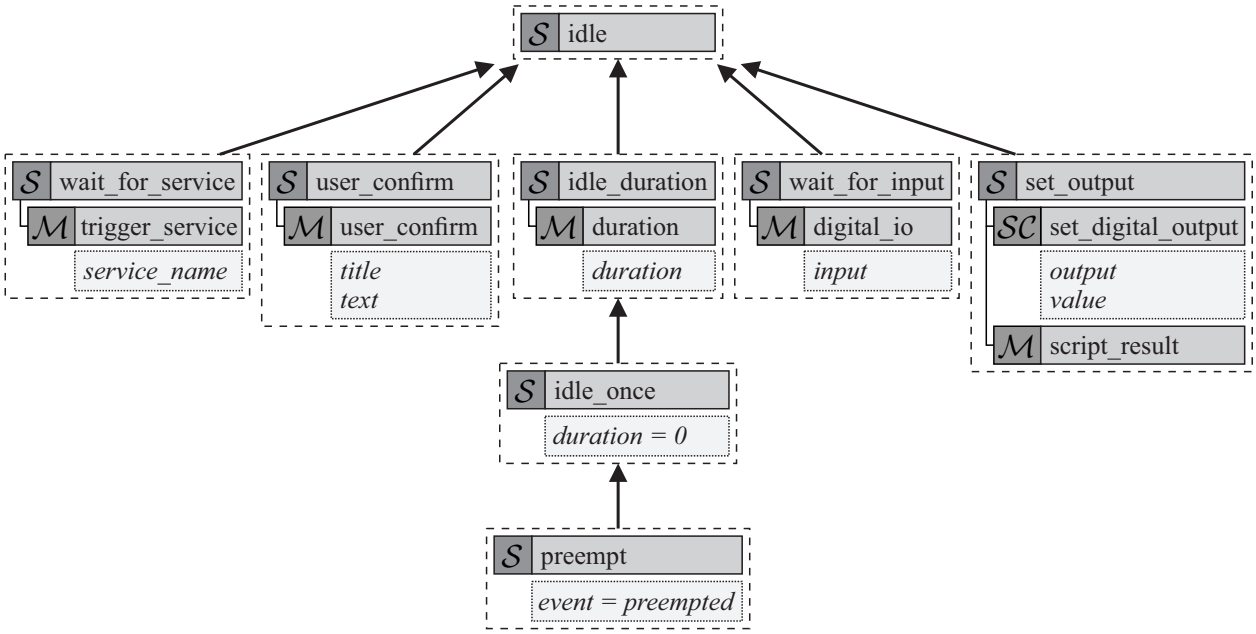


Abbildung 5.4: Komposition und Vererbung von `idle` Skills

Der `pivot` Skill eignet sich für einschwenkende Bewegungen, wie sie beispielsweise bei der Hutschienenbestückung auftreten. Ein `cartesian_tracking` Skill gibt dafür eine Drehbewegung vor. Der Kontakt im Drehpunkt wird durch das Aufbringen von Kräften mithilfe des `apply_force` Skills sichergestellt. Auch der `pivot` Skill definiert keine Stoppbedingungen und muss anwendungsspezifisch ergänzt werden.

Ausgangspunkt für eine weitere Reihe an Skills ist der `push` Skill, der aus einem `hold_pose` und einem `apply_force` Skill gebildet wird. Dieser Skill erlaubt es, in eine oder mehrere Achsrichtungen Kräfte aufzubringen, während die verbleibenden Lagekoordinaten konstant gehalten werden. Durch eine `duration` Stoppbedingung lässt sich mit dem abgeleiteten `push_duration` Skill für eine definierte Zeit drücken. Der `push_settle` Skill baut eine gewisse Kraft auf und wird beendet, sobald diese für eine eingestellte Anzahl an Regelungszyklen mit einer gegebenen Genauigkeit eingehalten wurde. Dies ist auf einfache Weise mit einer `control_error` Stoppbedingung modelliert. Zuletzt ist der `push_clockwise` Skill aufgeführt, der sich insbesondere für Einführvorgänge eignet. Zwei `sine_modifier` Skripte werden mit gleicher Frequenz, 90° Phasenverschiebung und zwei unterschiedlichen Achsrichtungen konfiguriert. Dadurch wird eine radiale Kraft aufgebracht, deren Richtung sich kontinuierlich in der von den Achsen aufgespannten Ebene dreht. Frequenz und Kraft lassen sich als Parameter vorgeben.

Abbildung 5.4 zeigt verschiedene Skills, die den Roboter nicht bewegen und stattdessen wichtige Hilfsaufgaben realisieren. Die Skills `idle_duration`, `idle_once` und `preempt` illustrieren sehr deutlich, wie schnell neue Skills durch Vererbung und einfaches Umparametrieren von Skills erzeugt werden können. Es wurde jeweils nur ein Parameter verändert, den Skills wurde dabei

jedoch aus Sicht der Programmlogik eine neue Bedeutung gegeben. Der `idle_duration` Skill besitzt eine einfache `duration` Stoppbedingung und wartet somit für eine eingestellte Zeit. Diese wird beim `idle_once` Skill auf null gesetzt, der dadurch für nur einen Regelungszyklus aktiv ist. Dieser Skill findet häufig Verwendung zur Ausführung von Skripten. Der `preempt` Skill schließlich setzt sein Event auf `preempted` und führt dadurch zum unmittelbaren Beenden des Roboterprogramms.

Weitere Skills ohne Roboterbewegung dienen der Kommunikation mit Nutzern und Peripheriekomponenten. Die Skills `wait_for_service` und `user_confirm` warten auf eine Eingabe durch den Nutzer, beispielsweise in Form eines ROS Service-Aufrufs oder der Quittierung eines Mitteilungsfensters. Die Skills `wait_for_input` und `set_ouput` dienen der Kommunikation mit Robotersteuerung und Peripheriekomponenten. Mit ihnen kann auf ein Signal eines digitalen Ein- oder Ausgangs gewartet beziehungsweise ein solches gesetzt werden.

Die hier vorgestellte Bibliothek an Skills, Skripten und Stoppbedingungen stellt die Basis für die exemplarischen Anwendungen des folgenden Abschnitts dar. Die Vielzahl und Vielseitigkeit an denkbaren Montageprozessen macht es aus praktischen Gründen unmöglich, eine vollständige Liste aller je benötigter Skills, Skripte und Stoppbedingungen zu erstellen. Ebenso ist es unpraktikabel, wenn für neuartige Montageanwendungen stets von Grund auf neue, von Experten vorgefertigte Skills zur einmaligen Verwendung erstellt werden müssen. Stattdessen soll mit dieser Arbeit gezeigt werden, dass einerseits bereits eine kleine Anzahl an Skills die Basis für eine Vielzahl von Anwendungsfällen bilden kann, es darüber hinaus aber auch sehr einfach ist, weitere Skills mittels vorhandener Bausteine zu erstellen, um komplexere Anwendungsfälle umzusetzen. Die Bibliothek erhebt somit nicht den Anspruch auf Vollständigkeit, sie erhebt vielmehr den Anspruch auf Erweiterbarkeit und damit auf die Möglichkeit, schnell und einfach anwendungsspezifische Skills auf Basis bereits existierender Skills und Elemente zu erstellen.

Wie zu Beginn des Abschnitts beschrieben, dienen Tiefe und Breite der Skill-Hierarchie als Maß dafür, inwieweit die Skills der Bibliothek aufeinander aufbauen und inwieweit sich bestehende Skills erweitern lassen. Die in den Abbildungen 4.15, 5.2 und 5.4 dargestellten einfachen Skills demonstrieren, wie mittels Vererbung und Erweiterung eine mehrstufige Hierarchie an Skills aufgebaut werden kann. Die einfachen Skills lassen sich zudem zur Komposition von Skills aus Sub-Skills heranziehen, wie es bei den in den Abbildungen 4.17 und 5.3 dargestellten Concurrency-Skills der Bibliothek geschieht. Weitere Hierarchiestufen entstehen bei der Erstellung von Skills für bestimmte Montageprozesse, beispielsweise dem Skill zur Hutschienenbestückung, der im nachfolgenden Abschnitt vorgestellt wird. Typischerweise ergeben sich Tiefen von sieben oder mehr Stufen. Für den bei der Hutschienenbestückung verwendeten `guarded_slide` Skill sind dies beispielsweise: (1) Kinematik: Modellierung der einfachen `cartesian` und `force` Skills; (2) Aufgabenspezifikation: Tasks und Regler der einfachen Skills;

(3) Koordination: Komposition zum Concurrency-Skill (`slide`); (4) Koordination: Definition der Stoppbedingung (`guarded_slide` Skill); (5) Koordination: Komposition des Teilprozess-Skills zum Aufrasten von Hutschienenelementen (Abbildungen 5.7 und 5.8); (6) Koordination: Komposition des Prozess-Skills zur Hutschienenbestückung; (7) Anwendung: Parametrierung für bestimmte Hutschienenelemente.

Jede Ebene modelliert dabei nur einen Aspekt der Aufgabenspezifikation. Dadurch bilden sich wiederum in jeder Ebene Anknüpfungspunkte für Skills der jeweils nächsten Ebene. Die Breite der Skill-Hierarchie zeigt sich beispielsweise deutlich beim `push` Skill, der in Abbildung 4.15 vorgestellten Familie der kartesischen Skills sowie den Concurrency-Skills der Abbildungen 4.17 und 5.3, die großteils auf den gleichen einfachen Skills aufbauen. Darüber hinaus sei betont, dass jeder einzelne Skill der Bibliothek als Ausgangspunkt für weitere Skills dienen kann, indem er durch Regler, Stoppbedingungen oder Skripte erweitert wird.

5.2 Komposition und Einsatz komplexer Montage-Skills

Die Anwendbarkeit der im vorangegangenen Abschnitt erstellten Skill-Bibliothek wird im Folgenden durch die Programmierung kraftgeregelter Montageprozesse demonstriert. Dieser Abschnitt stellt dazu drei der zehn in Tabelle 5.1 aufgeführten Anwendungsbeispiele im Detail vor: die Montage von Hutschienenelementen, das Aufstecken eines Plastikbauteils sowie das Verschrauben eines Getriebegehäuses. Es werden jeweils die für den Fügeprozess ausgewählten Skills und ihre Parametrierung vorgestellt. Eine Untersuchung zur Robustheit wird anschließend in Abschnitt 5.3 anhand der Hutschienenbestückung vorgenommen, eine Evaluierung der Wiederverwendbarkeit der eingesetzten Skills in Abschnitt 5.4.

5.2.1 Montage von Hutschienenelementen

Dieser Abschnitt beschreibt die kraftgeregelte Montage von Hutschienenelementen. Die vorgestellte Anwendung wurde mit unterschiedlichen Versuchsaufbauten getestet. Abbildung 5.1f zeigt den Aufbau mit einem Universal Robots UR3, Abbildung 5.1a mit einem KUKA LBR4+. Zudem wurde die Anwendung mit einem UR5 und einem Franka Emika Panda umgesetzt.

Das zu montierende Hutschienenelement wird vom Roboter gegriffen, schräg auf eine Hutschiene aufgesetzt und über eine Drehbewegung eingerastet, wie in Abbildung 5.5 dargestellt. Dafür besitzt es einen elastischen Schnapphaken aus Kunststoff, die sogenannte bewegliche Nase des Hutschienenelements. Teilweise wird diese auch über einen Blechschieber mit Feder realisiert.

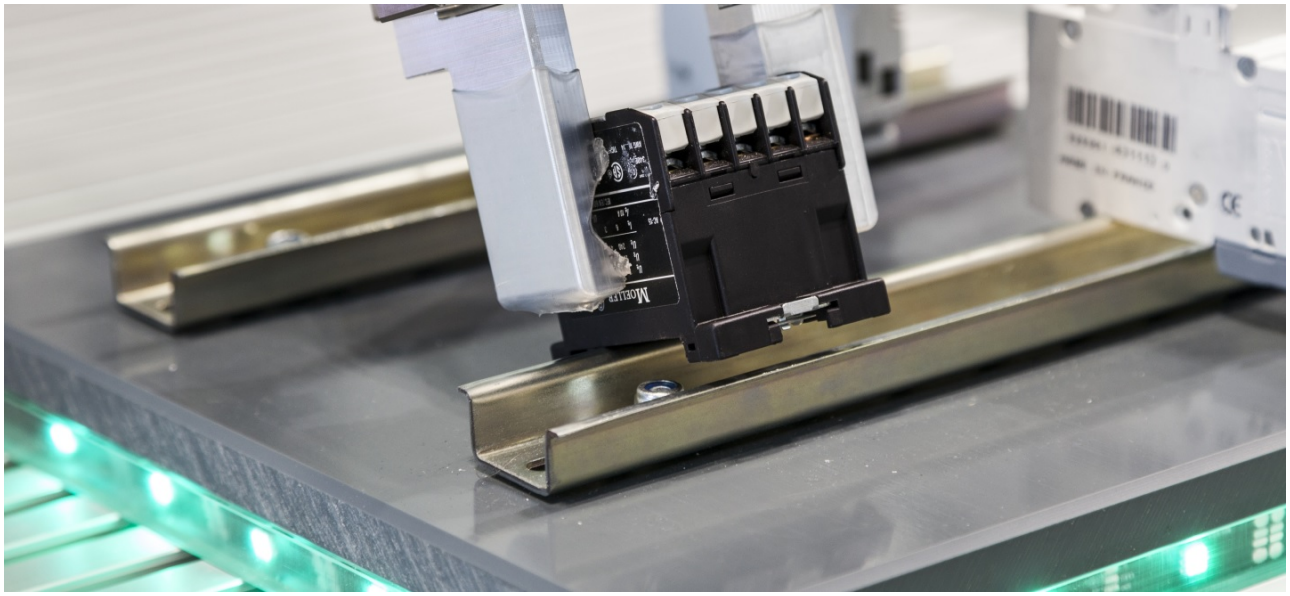
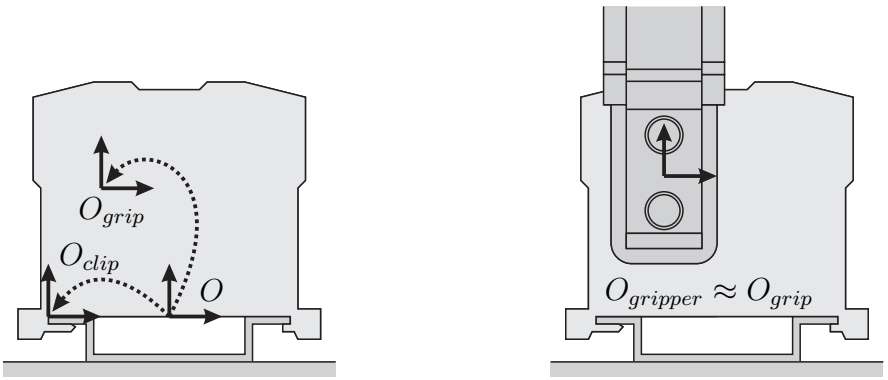


Abbildung 5.5: Beispielanwendung zur Montage von Hutschienenelementen

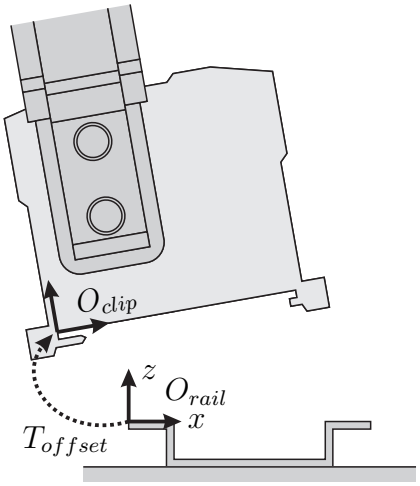


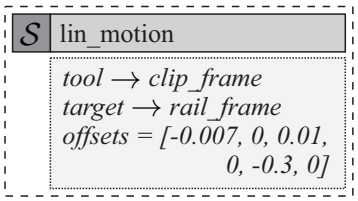
(a) Bauteilkoordinatensysteme (b) Gegriffenes Bauteil

Abbildung 5.6: Modellierung des Hutschienenelements

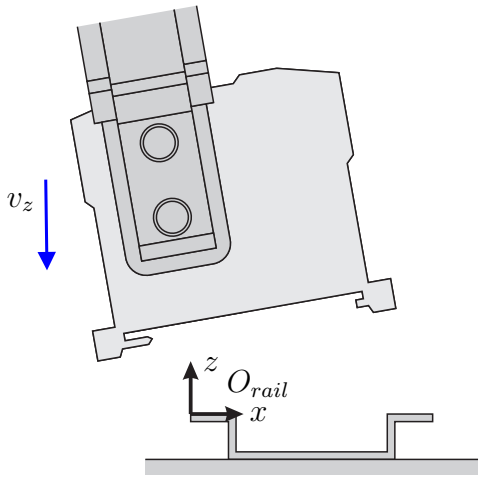
Die Modellierung eines aufgesetzten Hutschienenelements wird in Abbildung 5.6a in der Seitenansicht dargestellt. Das Koordinatensystem O_{clip} befindet sich an der Innenseite der nicht federnden Nase, die in die Krempe der Hutschiene eingehängt wird. Das Koordinatensystem O_{grip} beschreibt den Greifpunkt des Hutschienenelements. Abbildung 5.6b stellt den gegriffenen Zustand dar, bei dem das Koordinatensystem $O_{gripper}$, das sich in der Mitte der Greiferbacken befindet, zum Greifpunkt verfahren wurde.

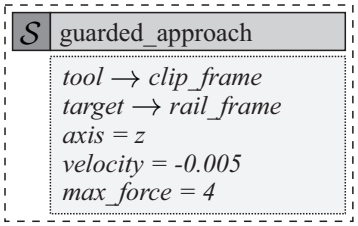
Der Ablauf der Montage in sechs Schritten und die dafür verwendeten Skills und Parameter sind in den Abbildungen 5.7 und 5.8 dargestellt. Zunächst wird das Hutschienenelement mit einem `lin_motion` Skill vorpositioniert. Dabei wird ein Versatz T_{offset} relativ zum Koordinatensystem O_{rail} eingestellt, der dem Ausgleich von Fertigungs- und Lagetoleranzen dient. Das



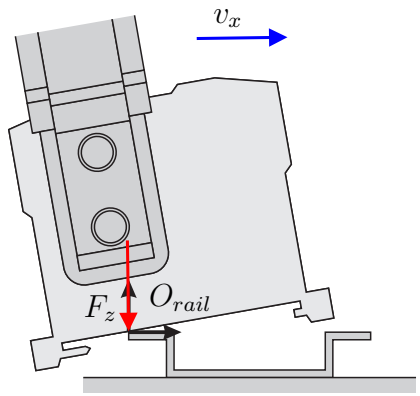


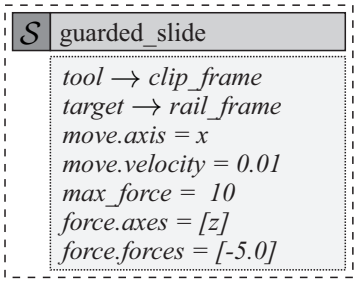
(a) Vorpositionierung mit `lin_motion` Skill





(b) Vertikaler Kontaktaufbau mit `guarded_approach` Skill





(c) Horizontaler Kontaktaufbau mit `guarded_slide` Skill

Abbildung 5.7: Ablauf und Parametrierung der Hutschienenbestückung (1)

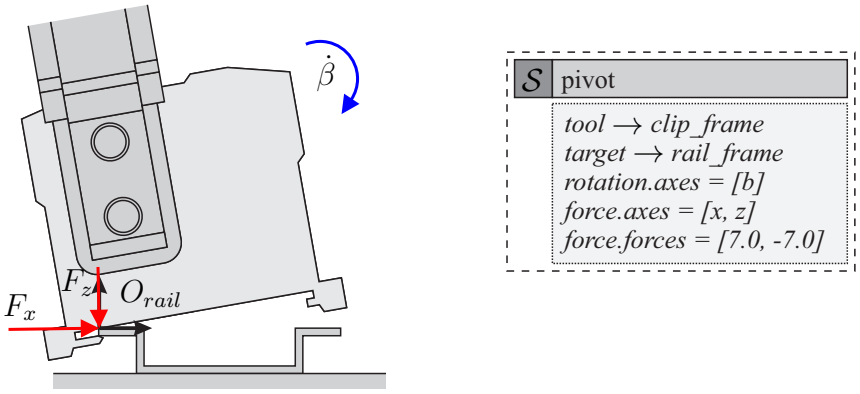
Werkzeugkoordinatensystem `tool` und das Zielkoordinatensystem `target` des Skills referenzieren die von einem übergeordneten Skill definierten Koordinatensysteme `clip_frame` (O_{clip}) und `rail_frame` (O_{rail}).

Der Vorpositionierung schließt sich eine Suchbewegung für den vertikalen Kontaktaufbau an. Dafür wird ein `guarded_approach` Skill eingesetzt, der eine geschwindigkeitsgeregelte Bewegung entlang der z-Achse ausführt, bis eine gegebene Kontaktkraft erreicht wird. Der Skill wird mit einer negativen Geschwindigkeit von 5 mm/s parametriert, da die Suchbewegung entgegen der Achsrichtung ausgeführt werden soll. Zur Parametrierung der Stoppbedingung wird die Maximalkraft von 4 N eingestellt, bei deren Erreichen die Suchbewegung beendet wird. Diese Kraft muss groß genug sein, um das Rauschen des Sensorsignals, Ungenauigkeiten bei der Modellierung der Gravitationskompensation, ein Driften des Sensors über die Zeit und nicht modellierte dynamische Kräfte ausgleichen zu können.

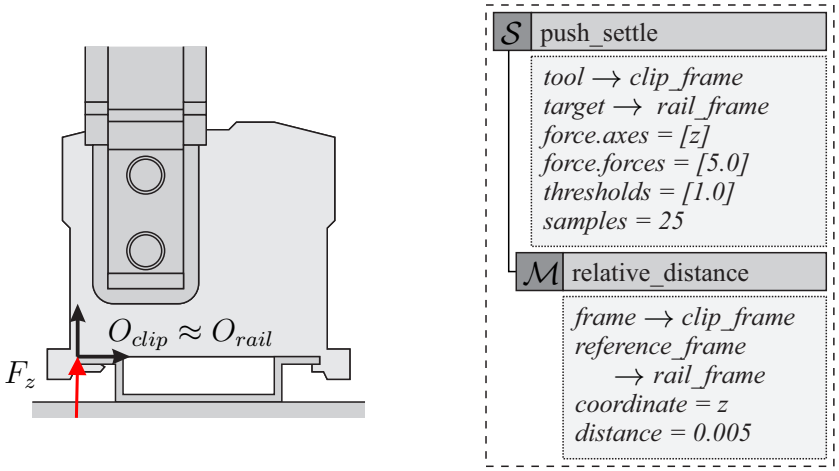
Der horizontale Kontaktaufbau erfolgt mit einem `guarded_slide` Skill. Da der eben erreichte vertikale Kontakt mit der Hutschiene gehalten werden soll, wird durch den Skill eine Kraft von 5 N auf die Hutschiene aufgebracht. Währenddessen bewegt der Skill den Roboter entlang der x-Achse des Koordinatensystems O_{rail}, bis auch in horizontaler Richtung eine Kontaktkraft auftritt. Diese ist mit 10 N etwas größer parametriert, da Reibkräfte nicht vom Modell berücksichtigt werden.

Die Einrastbewegung wird mithilfe eines `pivot` Skills durchgeführt. In Richtung der x-Achse und entgegen der z-Achse werden Kräfte aufgebracht, um die linke Nase an der Krempe der Hutschiene zu halten. Gleichzeitig wird das Hutschienenelement um die y-Achse gedreht, bis es an der Hutschiene ausgerichtet ist. Beim Aufsetzen auf der Hutschiene wird die federnde Nase dabei leicht weggedrückt und es kommt zum Einrasten des Schnapphakens.

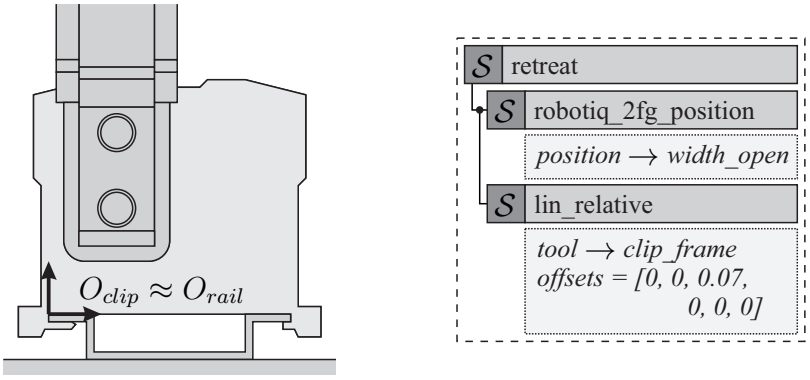
Die Drehung des hier verwendeten Skills ist rein positionsgeregelt. Um ein erfolgreiches Einrasten sicher zu stellen, kann beispielsweise eine Auswertung der aufgezeichneten Kraftkurve vorgenommen oder per Mikrofon der (Körper-)Schall auf ein Klickgeräusch hin geprüft werden. Im vorliegenden Beispiel wird hingegen ein `push_settle` Skill verwendet, der das Hutschienenelement nach oben zieht, um den in Abschnitt 5.3 beschriebenen Fehlerfall auszuschließen. Kann eine Kraft von 5 N aufgebaut und für 25 Regelungszyklen in einem Toleranzbereich von 1 N gehalten werden, so gilt die Montage als erfolgreich. Ist das Hutschienenelement nicht eingerastet, so kann die Kraft nicht aufgebaut werden und der Roboter fährt in Richtung der aufzubringenden Kraft nach oben. Mit einer `relative_distance` Stoppbedingung wird diese Bewegung nach 5 mm abgebrochen. Abschnitt 5.3 betrachtet, wie in diesem Fall eine Strategie zur Fehlerbehebung umgesetzt werden kann.



(a) Einrasten mit `pivot` Skill



(b) Überprüfen der Verbindung



(c) Öffnen des Greifers und Rückzugsbewegung

Abbildung 5.8: Ablauf und Parametrierung der Hutschienenbestückung (2)

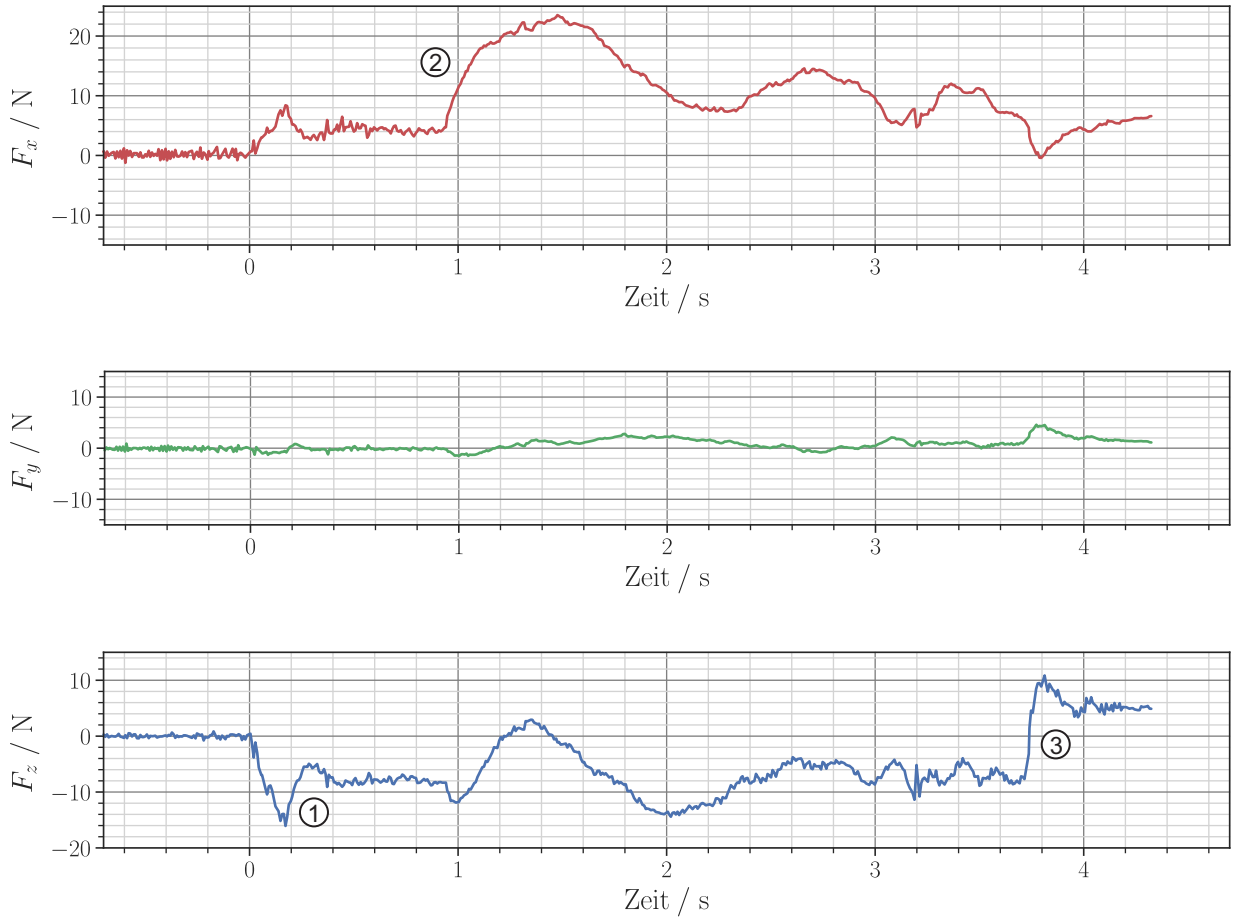


Abbildung 5.9: Gemessene Kräfte bei der Hutschienenbestückung

Im Fall einer erfolgreichen Montage wird der Greifer geöffnet und der Roboter führt eine relative Rückzugsbewegung nach oben hin aus. Diese erfolgt mittels eines `lin_relative` Skills.

Abbildung 5.9 zeigt einen typischen Verlauf der während der Montage aufgezeichneten Kräfte in x-, y- und z-Richtung bezüglich des Koordinatensystems O_{rail}. Eingesetzt wurde dazu der in Abbildung 5.1f gezeigte Aufbau mit Universal Robots UR3 und Robotiq 2F-85 Greifer. Zur Kraftmessung wurde jedoch der genauere ATI Axia 80 verwendet. Deutlich in den Kraftkurven zu erkennen sind der initiale Kontakt mit der Hutschiene in negativer z-Richtung (1), der Kontakt in positiver x-Richtung (2) sowie die Überprüfung der erfolgreichen Aufrastung durch Ziehen am Bauteil mit 5 N in positiver z-Richtung (3).

5.2.2 Montage eines Plastikbauteils durch Aufstecken

Abbildung 5.10 stellt die zweite Montageaufgabe dar, das Aufstecken eines Plastikbauteils. Der entsprechende Versuchsaufbau mit einem Denso VS-087 Roboter, DynPick WEF-6A200-4 Sensor und WSG 50-110 Greifer ist in Abbildung 5.1c dargestellt.

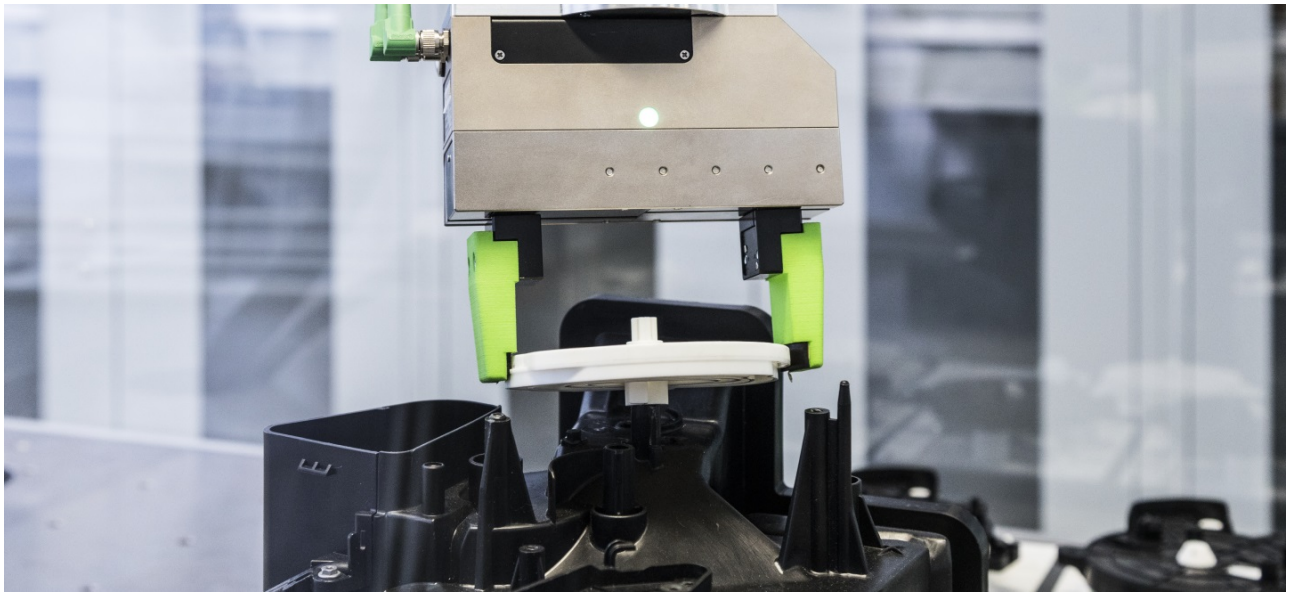
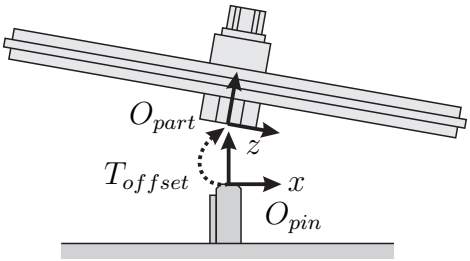
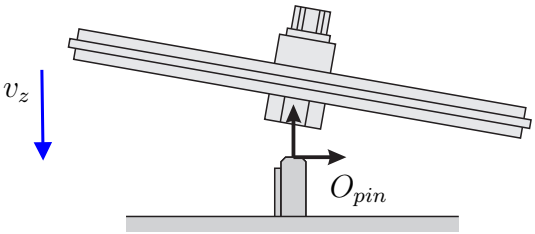


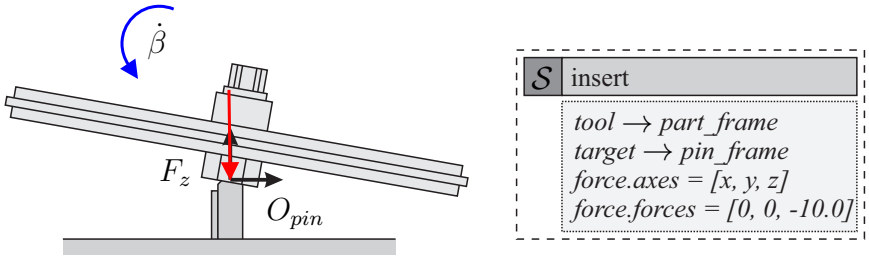
Abbildung 5.10: Beispielanwendung zur Montage eines Plastikbauteils durch Aufstecken



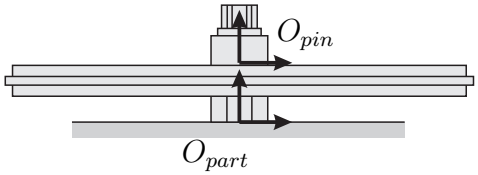
(a) Vorpositionierung (`lin_motion`)



(b) Kontaktaufbau (`guarded_approach`)



(c) Aufstecken des Plastikbauteils mit `insert` Skill



(d) Fertig montiertes Bauteil

Abbildung 5.11: Ablauf und Parametrierung des Einführvorgangs

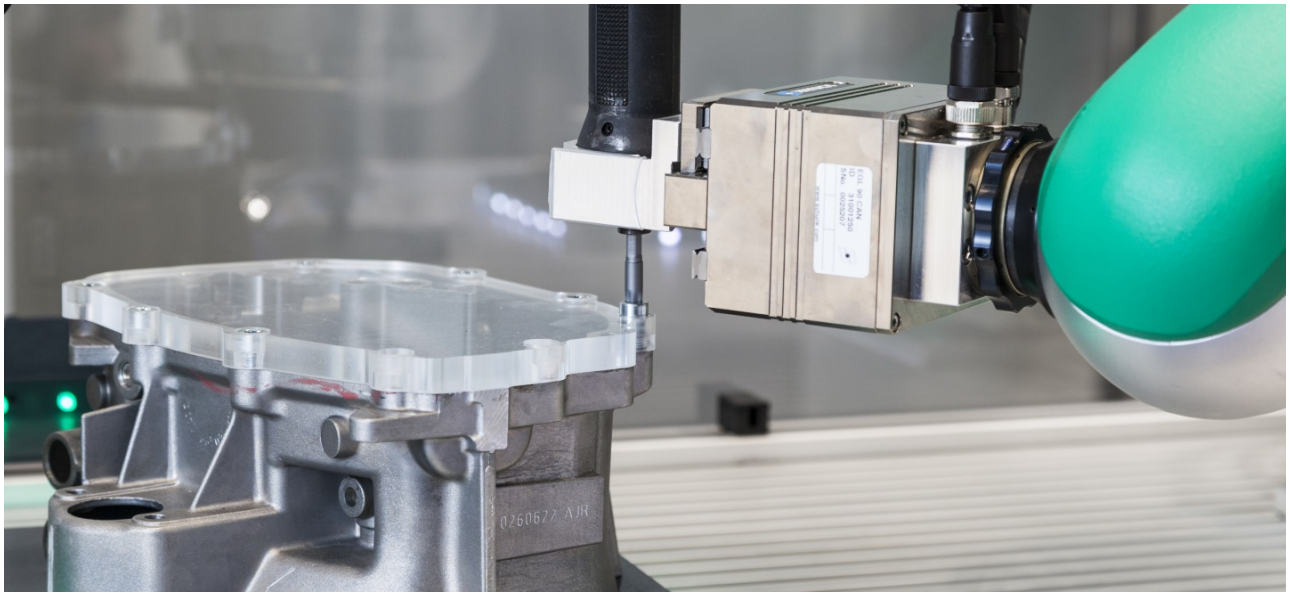


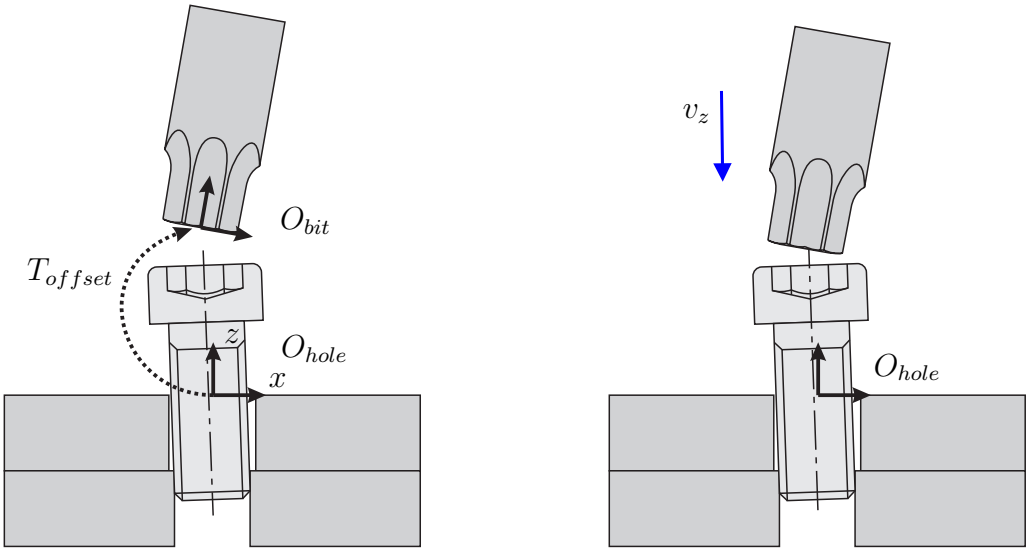
Abbildung 5.12: Beispielanwendung zur Verschraubung eines Getriebegehäuses

Der Ablauf der Montage ist zunächst ähnlich zur Hutschienenbestückung und in Abbildung 5.11 aufgeführt. Wie zuvor wird das zu montierende Bauteil in eine Vorposition gebracht und anschließend der Kontakt beider Bauteile hergestellt. Um Lagetoleranzen auszugleichen, wird das Bauteil mit einer leichten Verkippung aufgesetzt. Dies erhöht die Chance, dass der Pin auch bei kleinen Abweichungen in das Loch des Plastikbauteils eingeführt wird.

Der `insert` Skill wird verwendet, um das Bauteil schließlich in seine Bestimmungslage zu bringen. Die Querkräfte in x- und y-Richtung werden durch den Skill auf null geregelt. In z-Richtung wird hingegen eine Kraft aufgebracht, die zum Einführen des Pins in das Plastikbauteil führt. Der `insert` Skill richtet zudem die verbleibenden Orientierungskoordinaten des Werkzeugkoordinatensystems O_{part} am Zielkoordinatensystem O_{pin} aus und richtet das Bauteil somit auf.

5.2.3 Eindrehen von Schrauben

Als drittes Anwendungsbeispiel wird die Verschraubung eines Getriebegehäuses betrachtet, wie in Abbildung 5.12 dargestellt. Die Anwendung wurde mit einem KUKA LBR4+ (Abbildung 5.1a) sowie einem KUKA KR16 (Abbildung 5.1b) erprobt. Die Schraube sei dabei bereits in das Schraubloch eingesetzt, kann aber leicht verkippt sein. Der Vorschub beim Eindrehen der Schraube wird vom Roboter kraftgeregelt ausgeführt. Daher kann ein einfacher industrieller Handschrauber eingesetzt werden und es wird keine pneumatische oder elektrische Einheit für den Vorschub benötigt.



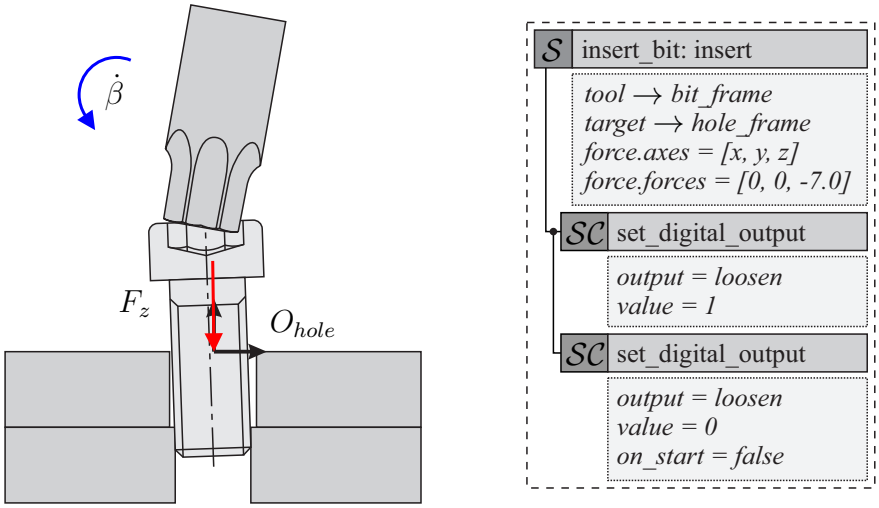
(a) Vorpositionierung (`lin_motion`) (b) Kontaktaufbau (`guarded_approach`)

Abbildung 5.13: Ablauf und Parametrierung des Schraubprozesses (1)

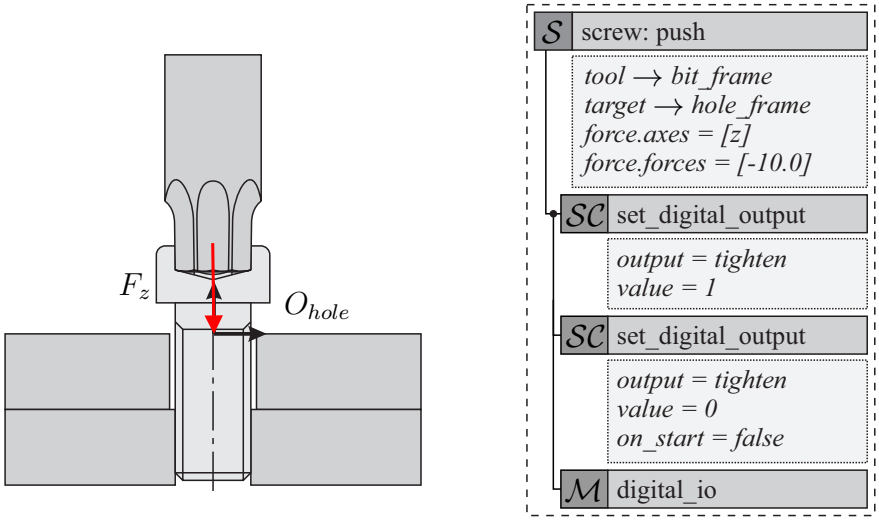
Am Schrauber angebracht ist ein Schraubendrehereinsatz (Bit) für Innensechsrundschrauben (auch unter dem Markennamen Torx geläufig). Abbildung 5.13 zeigt die ersten beiden Schritte des Schraubprozesses. Auch dieser beginnt wie bei den vorherigen Anwendungsbeispielen mit einer Vorpositionierung und dem Kontaktaufbau. Ebenso ist der Schrauber leicht gekippt, sodass der Bit bei kleinen Positionstoleranzen nicht an der Oberseite des Schraubenkopfs aufsetzt.

In Abbildung 5.14 sind die weiteren Schritte des Schraubprozesses dargestellt. Zunächst wird mit einem adaptierten `insert` Skill der Bit eingeführt. Der `insert` Skill drückt dazu mit einer definierten Kraft auf die Schraube, während er den Schrauber senkrecht aufrichtet. Hinzugefügt werden zwei `set_digital_output` Skripte, die den mit dem digitalen Ausgang verbundenen Schrauber anschalten beziehungsweise nach dem Beenden des Skills wieder ausschalten. Der Schrauber führt eine kurze Aufschraubbewegung aus, um den Bit in den Schraubenkopf einzuführen. Anschließend wird ein modifizierter `push` Skill eingesetzt. Erneut werden zwei Skripte zum An- und Abschalten des Schraubers verwendet. Durch das Aufbringen der Kraft wird der Schrauber vom Roboter während des Eindrehvorgangs nachgeführt. Beendet wird der Schraub-Skill durch ein Signal des Schraubers beim Erreichen des eingestellten Drehmoments, das durch eine `digital_io` Stoppbedingung überwacht wird.

Anhand der drei vorgestellten exemplarischen Anwendungsbeispiele wird demonstriert, dass sich Montageprozesse skill-basiert umsetzen lassen. So verschiedenartig die Anwendungsbeispiele zunächst erscheinen, zeigt sich jedoch, dass eine Implementierung in hohem Maße mit den gleichen Bausteinen der Skill-Bibliothek realisierbar ist. Eine Analyse dazu wird in Ab-



(a) Einführen des Bits mit `insert_bit` Skill



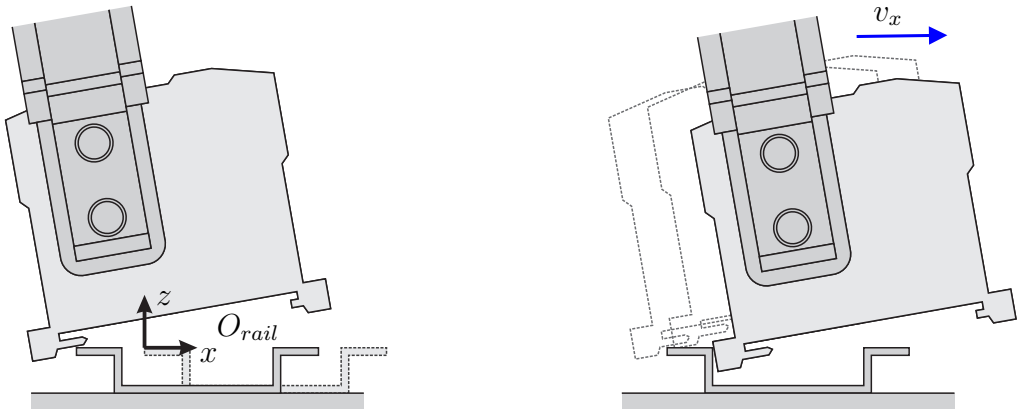
(b) Eindrehen der Schraube mit `screw` Skill

Abbildung 5.14: Ablauf und Parametrierung des Schraubprozesses (2)

schnitt 5.4.2 vorgenommen. Davor wird im nachfolgenden Abschnitt betrachtet, wie die Robustheit skill-basierter Anwendungen gewährleistet werden kann.

5.3 Modellierung robuster Skills und Anwendungen

In diesem Abschnitt wird die Robustheit skill-basierter Anwendungen gegenüber Fertigungs- und Lagetoleranzen betrachtet. Es wird evaluiert, wie sich mithilfe der Bausteine des Baukastens, insbesondere Stoppbedingungen, Fehlerfälle erkennen lassen und wie auf erkannte Fehler-



(a) Die reale Hutschiene befindet sich deutlich weiter links der modellierten (O_{rail}): Die Komponente setzt falsch auf

(b) Die Komponente rutscht bei der Ausführung des `guarded_slide` Skills über die Hutschiene hinweg und hakt nicht ein

Abbildung 5.15: Möglicher Fehlerfall bei der Hutschienenbestückung

fälle reagiert werden kann. Die Betrachtung erfolgt beispielhaft anhand der bereits vorgestellten Hutschienenbestückung und wird mittels einer Versuchsreihe demonstriert.

Die vorliegende Arbeit beschreibt einen modellbasierten Ansatz zur Programmierung von Montageprozessen. Zwischen Modell und Realität kommt es jedoch zu Abweichungen. Dies erfordert eine gewisse Robustheit des Systems gegenüber Unsicherheiten, wie beispielsweise Fertigungs- und Lagetoleranzen. Dazu wird eine zweistufige Strategie verfolgt, die sich auf die Ebene der Regelung und die Ebene der Koordination aufteilt. Die beiden Ansätze unterscheiden sich meist sowohl bezüglich ihrer Zeithorizonte als auch der geometrischen Größenordnung. So gleicht die Kraftregelung kleine Fertigungstoleranzen der Bauteile im Bereich von Millisekunden aus. Dahingegen werden auf Ebene der Koordination größere Unsicherheiten beispielsweise über Suchstrategien oder ein Statechart mit Logik zur Fehlerbehebung abgefangen. Diese Strategien bewegen sich meist im Bereich von Sekunden. In der Literatur (Stolt et al. 2011a; Villani et al. 2016) wird bereits gezeigt, wie mithilfe von Kraftregelung die Robustheit von Roboteranwendungen erhöht werden kann. Der Fokus soll hier daher auf die Ebene der Koordination gelegt werden und insbesondere darauf, wie Stoppbedingungen zur effektiven Fehlererkennung genutzt werden können.

Abbildung 5.15 zeigt einen beispielhaften Fehlerfall. Bei der Hutschienenbestückung sei die Hutschiene so weit links positioniert, dass der zum Toleranzausgleich definierte Versatz T_{offset} nicht ausreicht und der Kontaktaufbau mithilfe des `guarded_approach` Skills entsprechend zu weit rechts stattfindet. Die Nase des Hutschienenelements setzt auf der Krempe der Hutschiene auf und der `guarded_slide` Skill ist nicht in der Lage, die Nase in die Krempe einzuhaken. Es stellt sich daher die Frage, wie dieser Fehler erkannt werden kann.

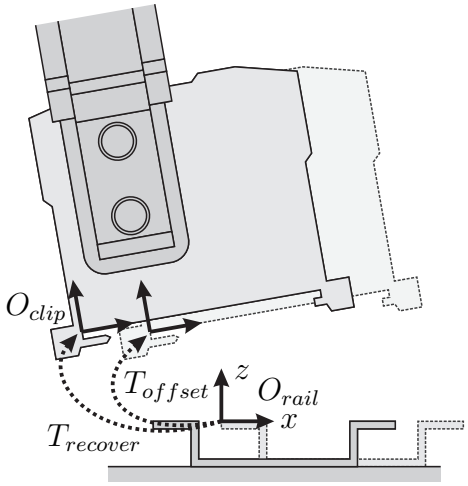


Abbildung 5.16: Fehlerbehebung: Erneuter Versuch mit nach links verschobener Startposition

Die folgenden Beispiele beschreiben zunächst theoretisch verschiedene Möglichkeiten, den genannten Fehlerfall zu erkennen und den Montagevorgang daraufhin zu stoppen. Anschließend wird eine Versuchsreihe mit 101 Durchläufen zur praktischen Evaluierung der Robustheit vorgestellt, bei der die zuerst beschriebene Möglichkeit der Fehlererkennung umgesetzt wird.

- Um zu gewährleisten, dass die Suchbewegung des `guarded_slide` Skills entlang der x-Richtung des O_{rail} Koordinatensystems nicht über das typische Maß hinaus fortgeführt wird, lässt sich diese mit einer `relative_distance` Stoppbedingung begrenzen.

- Mithilfe einer `duration` Stoppbedingung lässt sich die Dauer der Suchbewegung einschränken.

- Ein plötzliches Absenken des O_{clip} Koordinatensystems in z-Richtung von O_{rail} deutet darauf hin, dass die Nase über die Krempe der Hutschiene hinaus gerutscht ist, wie in Abbildung 5.15b bildlich dargestellt wird. Zur Erkennung lässt sich eine `relative_distance` Stoppbedingung verwenden.

- Auch ein plötzliches Abfallen der Kraft in z-Richtung von O_{rail} deutet darauf hin, dass das Bauteil von der Hutschiene gerutscht ist. Hierfür kann eine `threshold` Stoppbedingung zum Einsatz kommen.

- Falls die Unsicherheit der Lage von O_{rail} in z-Richtung geringer sein sollte als in x-Richtung, beispielsweise wenn die Lage der Montageplatte hinreichend bekannt ist, nicht aber die exakte Position der Hutschiene auf der Montageplatte, so kann die vertikale Distanz des O_{clip} Koordinatensystems gegenüber O_{rail} mit einer `distance` Stoppbedingung überprüft werden.

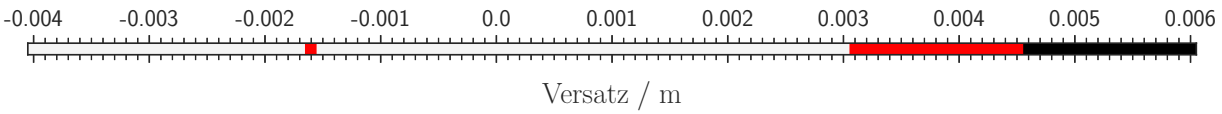


Abbildung 5.17: Ergebnisse der 101 Montagevorgänge abhängig vom Versatz in x-Richtung. Schwarz: Fehlerfall wird bereits beim `guarded_slide` Skill erkannt. Rot: Fehlerfall wird am Ende erkannt

Nach der Erkennung des Fehlerfalls ist es eine einfache Form der Fehlerbehebung, wie in Abbildung 5.16 gezeigt, einen erneuten Kontaktaufbau mithilfe des `guarded_approach` Skills zu probieren, wobei der größere Versatz $T_{recover}$ verwendet wird. Sollte auch dies nicht zu einer erfolgreichen Montage führen, ist zumindest ein kontrolliertes Aussortieren des Bauteils und damit eine unterbrechungsfreie Fortführung des Roboterprogramms möglich.

Zur praktischen Demonstration und Evaluierung der Robustheit wird der Versuchsaufbau aus Abschnitt 5.2.1 herangezogen. Der Prozess der Hutschienenbestückung wird in Summe 101 Mal ausgeführt, wobei die Startposition entlang der x-Achse variiert wird. Dies geschieht im Intervall von -4 bis 6 mm um die ursprünglich programmierte Position mit Schrittweite 0,1 mm. Die Reihenfolge der 101 Durchläufe wird dabei zufällig gewählt.

Zur Überwachung des `guarded_slide` Skills wird die zuerst beschriebene `relative_distance` Stoppbedingung eingesetzt. Löst diese aus, so wird die Hutschiene 7 mm weiter links angefahren und einmalig ein zweiter Aufrastversuch gestartet. Sollte auch dieser scheitern, wird der Versuch beendet und der Nutzer informiert.

Von den 101 Durchläufen wurden alle Hutschienenelemente spätestens beim zweiten Versuch erfolgreich montiert. Bei 70 Durchläufen war dies bereits beim ersten Versuch möglich. Bei den 15 in Abbildung 5.17 schwarz dargestellten Durchläufen, die einen Versatz in x-Richtung über 4,5 mm besitzen, tritt der zu erwartende Fehlerfall auf. Der `guarded_slide` Skill wird durch die `relative_distance` Stoppbedingung abgebrochen und das Hutschienenelement beim zweiten Aufrastversuch weiter links erfolgreich montiert.

Gegenüber der theoretischen Vorüberlegung zeigt sich im Übergangsbereich zwischen dem beschriebenen Fehlerfall und dem normalen Fall eine weitere Fehlermöglichkeit. Setzt die Nase des Hutschienenelements direkt auf der Kante der Hutschienenkrempe auf (hier bei einem Versatz in x-Richtung von 3,1 bis 4,5 mm), so verhakt sie sich dort. Die weiteren Montageschritte werden zwar ausgeführt, in diesem Fall scheitert der Aufrastvorgang jedoch, da die Nase nicht unter die Krempe rutscht. Durch die in Abbildung 5.8b beschriebene Prüfung wird auch dieser Fehlerfall erkannt und durch einen zweiten Aufrastversuch mit größerem Versatz korrigiert. Ebenso korrigiert wird der beim ersten Durchlauf gescheiterte Bestückungsvorgang bei $-1{,}6$ mm, bei dem

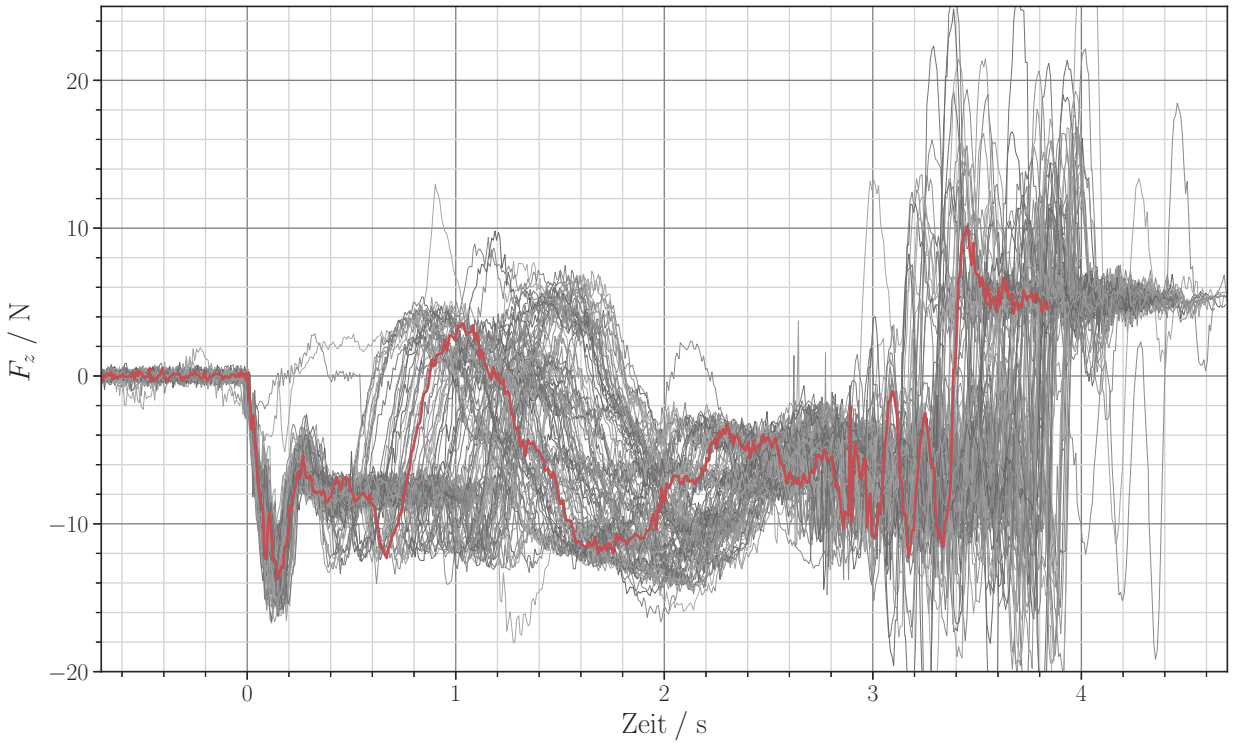


Abbildung 5.18: Kräfte in z-Richtung aller erfolgreichen Montagevorgänge, typischer Kraftverlauf in rot hervorgehoben

der `guarded_slide` Skill (vermutlich Aufgrund eines Verhakens des Bauteils) zu früh beendet wurde.

Abbildung 5.18 zeigt die Kräfte in z-Richtung des O_{rail} Koordinatensystems aller erfolgreichen Montagedurchläufe. Die Zeit $t = 0$ beschreibt dabei den Zeitpunkt, an dem das Hutschienenelement auf der Hutschiene aufsetzt. Die Streuung der Kraftkurven im weiteren Verlauf ist durch die unterschiedliche Dauer der `guarded_slide` Suchbewegung bedingt, abhängig von der Startposition des jeweiligen Durchlaufs. Deutlich zu erkennen ist bei den Kraftkurven die Überprüfung am Ende des Bestückungsvorgangs, bei der eine Zugkraft von 5 N aufgebaut wird. Bei den in Abbildung 5.19 dargestellten Fehlerfällen kann diese Kraft hingegen nicht aufgebaut werden.

Die Versuchsreihe demonstriert, wie mit Stoppbedingungen Fehlerfälle effektiv modelliert und damit Montageprozesse robust gestaltet werden können. Hervorgehoben sei, dass alle in der theoretischen Vorüberlegung genannten Beispiele für die Erkennung des Fehlerfalls mit Standardbausteinen der Bibliothek umgesetzt werden können. Keinerlei anwendungsspezifischer Programmcode muss dafür geschrieben werden. Für sehr neuartige Anwendungen ist dies jedoch mit der Programmierung weiterer Stoppbedingungen jederzeit möglich. Gegebenenfalls sind auch diese soweit verallgemeinerbar, dass sie der Bibliothek hinzugefügt werden können. Ebenso sei darauf hingewiesen, auf welch vielseitige Art Fehlerfälle wie der vorliegende gelöst

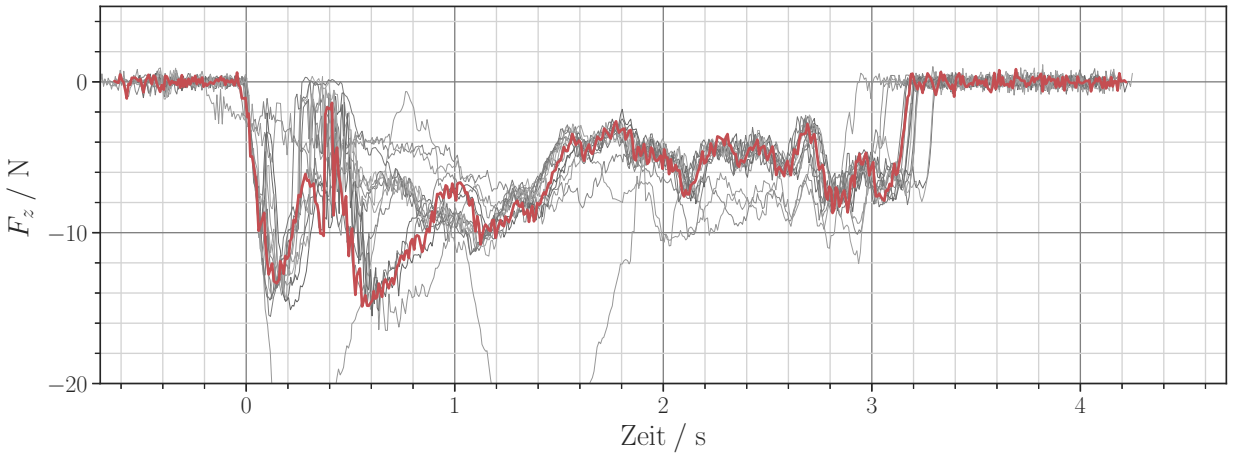


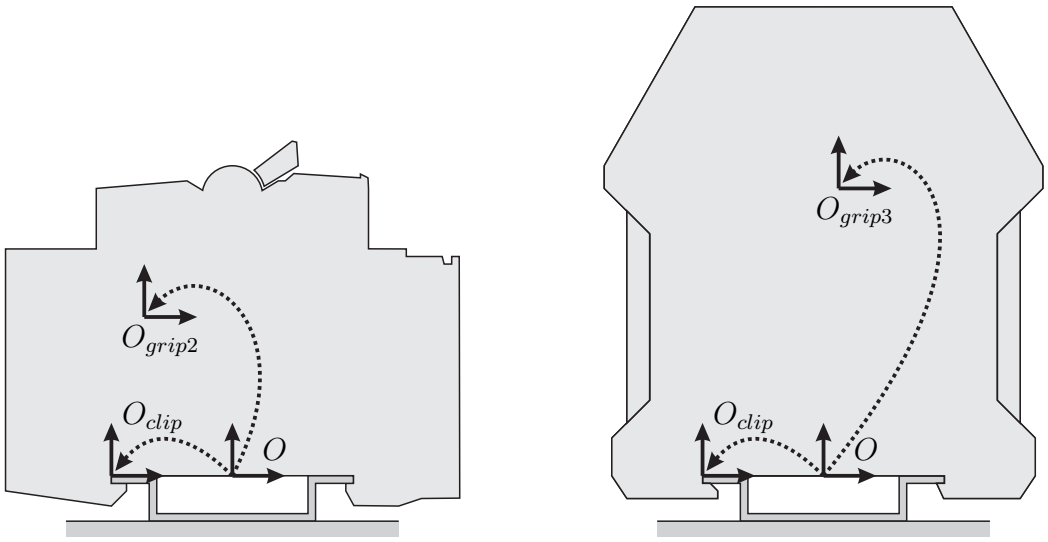
Abbildung 5.19: Kräfte in z-Richtung aller beim ersten Versuch gescheiterten Montagevorgänge, typischer Kraftverlauf in rot hervorgehoben

werden können, sei es über die Betrachtung von absoluten oder relativen Positionen, Kraftverläufen oder der Zeit.

5.4 Wiederverwendung und Anpassbarkeit von Skills

In Abschnitt 5.1 wird bereits ein hohes Maß an Wiederverwendung bei der Erstellung von Skills belegt. Im Folgenden wird darüber hinaus das Maß der Wiederverwendung bei der Parametrierung und dem Einsatz der Skills untersucht. Es wird damit die zentrale Fragestellung der Arbeit betrachtet, wie kraftgeregelte Montageprozesse modelliert werden können, um neue Bauteilvarianten, Prozesse und Hardwarekomponenten mit einem hohen Maß an Wiederverwendbarkeit abzubilden.

Die Betrachtung wird zunächst für Produktvarianten durchgeführt, bei denen sich meist nur einzelne geometrische Eigenschaften der zu montierenden Bauteile ändern, die über Parameteranpassungen abgedeckt werden können. Bei komplett neuartigen Produkten beziehungsweise Montageprozessen reichen einzelne Parameteranpassungen hingegen nicht aus. Die Wiederverwendbarkeit wird hier auf Basis der eingesetzten Skills betrachtet. Dabei ist von Interesse zu prüfen, inwieweit mit vorhandenen Skills auch sehr unterschiedliche Montageanwendungen umgesetzt werden können und an welchen Stellen neue Skills erstellt werden müssen. Abschließend wird untersucht, wie abhängig Skills und Anwendungen von den eingesetzten Hardwarekomponenten sind.



(a) Bauteilvariante 2 (b) Bauteilvariante 3

Abbildung 5.20: Bauteilvarianten für die Hutschienenbestückung

5.4.1 Anpassbarkeit bei Produktvarianten

Zu den typischen Arten von Variationspunkten zählen geometrische Eigenschaften wie Längen und Formen, die sich in den Zielposen widerspiegeln. Dies betrifft sowohl die Montagepositionen als auch die Greifpunkte. Durch die Trennung der einprogrammierten Namen der Koordinatensysteme in der Aufgabenspezifikation von deren tatsächlichen Werten lassen sich neue Geometrien leicht anpassen. Zu den weiteren Variationsmerkmalen gehören Materialeigenschaften, Gewicht und Passungen. Diese können Einfluss auf die benötigte Greifkraft haben oder sich in den auszuübenden Prozesskräften und deren Richtungen niederschlagen. Die folgenden Beispiele sollen verdeutlichen, wie Variationspunkte mit dem vorgestellten Skill-Modell abgebildet werden können, um Aufgabenspezifikationen variantenübergreifend zu definieren.

Abbildung 5.20 zeigt beispielhaft zwei neue Komponenten zur Montage auf Hutschienen. Lediglich der Greifpunkt `grip_frame` muss hier je Komponente angepasst werden sowie die Greiferpositionen `width_open` und `width_closed`, die aufgrund unterschiedlicher Bauteilbreiten variieren können. Die verbleibenden Prozessschritte der Hutschienenbestückung sind von der Bauteilgeometrie unabhängig. So ist hier primär die Transformation von O nach O_{clip} entscheidend, die für alle Komponenten dieselbe ist. DSL-Codebeispiel 5.1 zeigt die vereinfachte Definition des Skills `mount_component` zur Hutschienenbestückung. Dieser besitzt die genannten variantenspezifischen Parameter sowie einen Auszug der Sub-Skills zum Montieren von Hutschienenelementen aus den Abbildungen 5.7 und 5.8. Bei der Nutzung des Skills werden die variantenspezifischen Parameter entsprechend gesetzt. Konkrete Werte der Koordinatensysteme hingegen werden,

Codebeispiel 5.1: Skill zur Hutschienenbestückung

```
<!-- Definition des Skills zum Aufrasten -->
<clone id="mount_component" prototype="skill_sequence">

  <!-- Definition der variantenspezifischen Parameter -->
  <clone id="grip_frame" prototype="frame"/>
  <clone id="width_open" prototype="float"/>
  <clone id="width_grip" prototype="float"/>

  <!-- Definition der allgemeinen Parameter -->
  <clone id="tool_frame" prototype="frame">0_gripper</clone>
  <clone id="clip_frame" prototype="frame">0_clip</clone>
  <clone id="rail_frame" prototype="frame">0_rail</clone>

  <!-- Definition des Ablaufs -->
  <member id="skills">

    <!-- Positionierung zum Greifen -->
    <clone prototype="lin_motion">
      <member id="tool" reference_id="tool_frame"/> <!-- Verweis auf Parameter -->
      <member id="target" reference_id="grip_frame"/>
    </clone>

    <!-- Greifen des Bauteils -->
    <clone prototype="robotiq_2fg_grip">
      <member id="position" reference_id="width_grip"/>
      <member id="speed">255</member>
      <member id="force">255</member>
    </clone>

    <!-- ... -->

    <!-- Vorpositionierung -->
    <clone prototype="lin_motion">
      <member id="tool" reference_id="clip_frame"/>
      <member id="target" reference_id="rail_frame"/>
      <member id="offsets" reference_id="offsets">-0.007, 0, 0.01, 0, -0.30, 0</member>
    </clone>

    <!-- Vertikaler Kontaktaufbau -->
    <clone prototype="skill_guarded_approach">
      <member id="tool" reference_id="clip_frame"/>
      <member id="target" reference_id="rail_frame"/>
      <member id="axis">z</member>
      <member id="velocity">-0.005</member>
      <member id="max_force">4</member>
    </clone>

    <!-- ... -->

    <!-- Oeffnen des Greifers -->
    <clone prototype="robotiq_2fg_position">
      <member id="position" reference_id="width_open"/>
      <member id="speed">255</member>
    </clone>

  </member>
</clone>


<!-- Variantenspezifische Parametrierung des Skills -->
<clone id="my_component_1" prototype="mount_component">
  <member id="grip_frame">0_grip_2</member>
  <member id="width_open">50</member>
  <member id="width_grip">200</member>
</clone>
```

wie in Abschnitt 3.1 beschrieben, nicht in der Aufgabenspezifikation gesetzt, sondern von externen Quellen vorgegeben, sei es durch eine Zellsteuerung, Bildverarbeitungskomponente oder die Modellierung der Roboterzelle.

In einem weiteren Anwendungsbeispiel wurde die Montage eines kleinen Kabelbaums mit fünf Steckern durch zwei Roboterarme vorgenommen. Charakteristisch für diese Anwendung sind sehr unterschiedliche Steckertypen, bei denen sich Geometrie und Größe sowie damit einhergehend die Fügekraft des Steckvorgangs unterscheiden. Daher ist es nötig, neben den Greifpunkten, der Greiferöffnung und der Position des Steckers auch den Versatz bei der Vorpositionierung (analog zu Abbildung 5.7a) und die Kräfte beim Einführvorgang an die Steckertypen anzupassen. Die vertikalen Kräfte, die für ein Einrasten des Schnappmechanismus benötigt werden, variieren bei den gegebenen fünf Steckertypen zwischen 18 und 50 N. Die horizontalen Kräfte zum Führen des Steckers können einheitlich bei 4 N festgesetzt werden, jedoch ist deren Richtung aufgrund der unterschiedlichen Steckergeometrien (insbesondere der Einführschrägen) individuell zu wählen. Alle anderen Parameter und insbesondere der Ablauf des Montageprozesses können direkt wiederverwendet werden.

Wie die aufgeführten Beispiele nahelegen, lassen sich neue Produktvarianten mit dem vorgestellten Skill-Modell meist durch eine Anpassung weniger, einfacher Parameter eines bestehenden Prozessmodells abbilden. Art und Umfang der vorzunehmenden Anpassungen sind dabei jedoch stark von der jeweiligen Anwendung abhängig. Neben den exemplarisch gezeigten Parametern lassen sich auch beliebige weitere Parameter auf gleiche Weise variantenspezifisch formulieren, wenn sie Variationspunkte darstellen. Auch auf Ebene der Programmlogik lassen sich Unterschiede zwischen Varianten oft mit wenigen Parametern abbilden. Hierzu zählen beispielsweise die Anzahl an Schrauben oder zu montierender Bauteile sowie deren Reihenfolge. Sind die Unterschiede zwischen Varianten nicht durch einfache Parameter abbildbar, so müssen gegebenenfalls Skills neu arrangiert oder neue Skills definiert werden. Hiermit befasst sich der nachfolgende Abschnitt 5.4.2.

5.4.2 Wiederverwendbarkeit bei neuen Prozessen und Produkten

Die Montage von Bauteilen ist durch eine Vielfalt verschiedener Prozesse geprägt. So unterteilt die Norm DIN 8593-1 die Kategorie des *Zusammensetzens* von Bauteilen in die Verfahren zum Auflegen, Einlegen, Ineinanderschieben, Einhängen, Einrenken und federnd Einspreizen. Diese Kategorien lassen sich weiter in Unterkategorien einteilen. So zählen zu den Spreizverbindungen beispielsweise Sprengringe, Sicherungsringe und -scheiben sowie Schnappverbindungen. Letztere umfassen wiederum Biege-, Torsions- und Ringschnappverbindungen, ringartige Schnappverbindungen sowie Kugelgelenkverbindungen (Feldmann et al. 2014). Erschwerend kommt zu

Tabelle 5.4: Auflistung der für die exemplarischen Anwendungen verwendeten Skills

Anwendung	`guarded approach`	`guarded slide`	`guarded lin`	`pivot`	`insert`	`push`	Eigener Skill
Hutschienenmontage	x	x		x		x	
Plastikteil aufstecken	x				x		
Verschrauben	x				x	x	
Autotürgriff montieren	x	x					x
Blech ausrichten		x		x		x	
Gehäuseteil montieren	x				x	x	
Kabel verlegen	x						x
Kabelbaum stecken	x		x		x		
Stirnrad aufschieben	x					x	x
USB-Stecker einführen			x		x		

dieser Prozessvielfalt hinzu, dass die noch größere Form- und Materialvielfalt verschiedener zu montierender Produkte die auszuführenden Montagebewegungen beeinflussen.

Es ist folglich nicht davon auszugehen, dass eine kleine Anzahl an fest definierten Fügestrategien diese Vielfalt abbilden kann. Dennoch ist eine effiziente Programmierung prozess- und produktspezifischer Fügestrategien wünschenswert. Hier ist es demnach von großer Wichtigkeit, dass neue Montageprozesse leicht durch Wiederverwendung bestehender Skills und durch Kombination der Skills mit vorhandenen Bausteinen, wie Skripten und Stoppbedingungen, modelliert werden können.

Um zu evaluieren, wie übertragbar die Skills der vorgestellten Bibliothek sind, wird ihre Verwendung anhand von zehn exemplarisch umgesetzten Montageprozessen untersucht. Neben den drei in Abschnitt 5.2 detailliert vorgestellten Anwendungsfällen werden sieben weitere industrielle Anwendungsfälle mit in die Betrachtung einbezogen. Tabelle 5.1 zu Anfang des Kapitels führt alle Anwendungsfälle auf und ordnet sie nach den primären Fügebewegungen jeweils einer Kategorie zu. Trotz der begrenzten Anzahl an Anwendungsfällen mag die bereits große Vielfalt an Prozessen, Materialpaarungen und Anwendungsbereichen ein Indiz dafür sein, dass das vorgestellte Skill-Modell auf eine große Anzahl weiterer Montageprozesse anwendbar ist.

Tabelle 5.4 zeigt die Auswertung der umgesetzten Anwendungen bezüglich verwendeter Skills. Dabei werden ausschließlich Skills mit Kraftregelung aufgeführt, da diese charakteristisch für die Montageprozesse sind. Es zeigt sich, dass das aufgeführte Set an Skills größtenteils ausreichend ist. Anwendungsspezifische Anpassungen der Skills werden durch Hinzufügen von Skripten und Stoppbedingungen vorgenommen (wie beispielsweise in Abbildung 5.14 dargestellt). Die Komposition von neuen Skills ist nur in drei Fällen nötig, wobei sich die erstellten Skills an `pivot`,

`push` beziehungsweise `guarded_approach` orientieren. Die meisten Skills werden dreimal oder öfter verwendet. Es ist davon auszugehen, dass für Anwendungen im Produktiveinsatz auch der `guarded_lin` Skill häufiger Verwendung findet, da er sich für die Absicherung gegenüber Fehlersituationen eignet, bei denen beispielsweise durch falsch positionierte oder defekte Bauteile unerwünschte Kontakte auftreten. Auch sei angemerkt, dass sich Montageprozesse meist mit verschiedenen Fügestrategien und damit unterschiedlichen Skills umsetzen lassen.

Für die einzelnen Anwendungen lassen sich die jeweils dafür verwendeten Skills zu Sequenzen oder Statecharts zusammenfassen, beispielsweise ein Skill für das Aufrasten von Hutschienenelementen (DSL-Codebeispiel 5.1) oder gar die gesamte Hutschienenbestückung. Diese Skills, oft Macro-Skills genannt, lassen sich sehr leicht einsetzen, da sie die komplexen Zusammenhänge ihrer Sub-Skills verbergen und nach außen hin eine kompakte Schnittstelle bieten. Dahingegen sind sie aufgrund ihrer sehr anwendungsspezifischen Art nicht oder nur eingeschränkt auf andere Anwendungsgebiete übertragbar. Eine Ausnahme bilden beispielsweise `pick` Skills, die aus einer Bewegung zu einer Zielposition, dem Schließen des Greifers und einer relativen Bewegung zum Zurückweichen bestehen. Durch die großen Unterschiede der Greiferschnittstellen sind diese `pick` Skills jedoch weitestgehend abhängig vom eingesetzten Greifer. Der nachfolgende Abschnitt 5.4.3 betrachtet dazu die Hardwareabhängigkeit von Skills und Anwendungen detaillierter.

5.4.3 Betrachtung der Hardwareabhängigkeit

Viele Ansätze zur Aufgabenspezifikation, wie beispielsweise von Klotzbücher et al. (2011) und weitere der in Kapitel 1 vorgestellten Arbeiten, streben das Ziel an, Aufgabenspezifikationen unabhängig von der eingesetzten Hardware zu gestalten. Hierdurch werden einmal modellierte Anwendungen oder Skills leichter auf andere Systeme übertragbar. Da viele Hardwarekomponenten einzigartige Funktionalitäten anbieten, deren Nutzung je nach Anwendung notwendig oder hilfreich sein kann, mag eine vollständige Hardwareabstraktion jedoch nicht immer zweckdienlich sein.

Das Ziel der Arbeit ist demnach nicht die Konzeption einer umfangreichen Hardwareabstraktionsschicht, die eine komplette Unabhängigkeit der Skills vom Hardwareaufbau erlaubt. Ziel ist es vielmehr, den Transfer einer Anwendung auf einen anderen Hardwareaufbau mit einem hohen Maß an Wiederverwendbarkeit zu ermöglichen und Änderungen in der Aufgabenbeschreibung auf die Stellen zu begrenzen, an denen sie einen Mehrwert bieten. Von Halt et al. (2018) wird bereits eine erste Untersuchung der Übertragbarkeit von Anwendungen vorgenommen. Dieser Abschnitt soll im Detail evaluieren, inwieweit Skills und Anwendungen von der eingesetzten Hardware abhängig sind. Um dies zu erörtern, werden die wichtigsten Hardwarekomponenten

Tabelle 5.5: Unterstützte Roboter und ihre Schnittstellen

Roboter	Eingesetzte Modelle	Schnittstellen	Mögliche Taktraten
Denso	VP-6242 VS-087	b-CAP	125 Hz
Franka Emika	Panda	libfranka, franka_ros	1000 Hz
KUKA	KR6, KR16 iiwa 7 R800 LBR4/4+	RSI Sunrise.FRI FRI	83,3 Hz, 250 Hz 1000 Hz 1000 Hz
Universal Robots	UR3/5/10 UR3/5/10 e-Series	RTDE	125 Hz 500 Hz

des Systems – Roboter, Greifer und Sensoren – exemplarisch ausgetauscht. Es wird dabei aufgezeigt, welche Änderungen bei einem Hardwarewechsel nötig sind und wo die Grenzen der Übertragbarkeit liegen. In Abschnitt 3.2 wurden einige Eigenschaften der genannten Komponenten bereits grob umrissen.

Betrachtet werden hier nur Änderungen, die an der Aufgabenspezifikation vorgenommen werden müssen. Zusätzlich müssen bei der erstmaligen Integration neuer Hardwarekomponenten zunächst Modelle für diese erstellt werden. Der Aufwand hierfür ist aufseiten des Skill-Modells jedoch gering. So müssen für die Modellierung eines Roboters lediglich die Namen der Gelenkkoordinaten, des Basis- und des Flanschkoordinatensystems angegeben werden. Die dazugehörigen kinematischen Daten werden automatisch dem im URDF beschriebenen Robotermodell entnommen. Besitzt der Robotertreiber bereits eine ROS Control Schnittstelle, so kann diese direkt verwendet werden und es wird kein zusätzlicher Aufwand aufseiten der Implementierung nötig.

Wechsel des Roboters

Die Ausführung von Anwendungen, die mit dem beschriebenen Skill-Modell erstellt wurden, ist mit Robotern verschiedener Hersteller möglich. Tabelle 5.5 führt dazu die bereits integrierten und erprobten Robotermodelle sowie ihre Schnittstellen auf. Zur Beurteilung der Roboterunabhängigkeit sollen im Folgenden die vorzunehmenden Änderungen der Aufgabenspezifikation beschrieben werden, die sowohl bei der Modellierung der Roboterzelle, der Anwendung als auch der einzelnen Skills nötig sind. Nicht Teil der Aufgabenspezifikation sind hingegen spezifische Einstellungen der jeweiligen Robotertreiber.

Codebeispiel 5.2: Definition des eingesetzten Roboters

```
<!-- Laden des Robotermodells -->
<include package="pitasc_library" file="universal_robots/ur.xml"/>

<!-- ... -->

<member id="environment">
    <!-- Parametrierung des Robotermodells -->
    <clone id="robot_ur5" prototype="robot_ur">
        <member id="robot_driver">
            <member id="topic_in">ur5/joint_vel_request</member>
            <member id="topic_out">ur5/joint_vel</member>
            <member id="urdf_key">ur5/robot_description</member>
        </member>
        <member id="urdf_prefix">ur5_</member>
        <member id="namespace">/ur5</member>
    </clone>
</member>
```

Codebeispiel 5.3: Verknüpfung des eingesetzten Roboters

```
<member id="applications">
    <clone id="main_app" prototype="skill_sequence">
        <!-- Zuordnung des Robotermodells zum Skill (sowie allen Sub-Skills) -->
        <member id="robot" reference_id="environment.robot_ur5"/>
        <member id="skills">
            <!-- Sub-Skills -->
        </member>
    </clone>
</member>
```

DSL-Codebeispiel 5.2 beschreibt die Parametrierung des eingesetzten Roboters. Diese enthält implementierungsspezifische Parameter (ROS Namensräume, Topics und Prefixe), die nötig sind, um mehrere Robotertreiber desselben Typs gleichzeitig anbinden zu können. Ist nur ein Roboter im System vorgesehen, so ist im Allgemeinen keine Parameteränderung für einen Roboterwechsel nötig und es muss lediglich das entsprechende Robotermodell geladen werden.

Um Skills auf einem bestimmten Roboter auszuführen, besitzt jeder Skill den Parameter `robot`, der ihm den Roboter zuweist. Als Grundeinstellung referenziert dieser den gleichnamigen Parameter des jeweiligen Eltern-Skills. Wie in Abschnitt 4.1.2 beschrieben wird, ist dadurch ein einfaches «Durchreichen» von Referenzen in Richtung des Wurzelknotens möglich. Lediglich an oberster Stelle, in DSL-Codebeispiel 5.3 die Anwendung `main_app`, muss der zu verwendende Roboter verknüpft werden, was hier über die Referenz `environment.robot_ur5` auf den bereits in DSL-Codebeispiel 5.2 definierten Roboter geschieht. Darüber hinaus ist es auch möglich, einem einzelnen Skill direkt einen Roboter zuzuweisen. Diese Zuweisung gilt entsprechend für diesen Skill wie für alle seine Sub-Skills. Über Concurrency-Skills lassen sich schließlich Skills mehrerer Roboter gleichzeitig ausführen. Auf diese Weise ist eine flexible Aufgabenspezifikation auch für zwei oder mehr Roboter möglich, wie für die in Abbildung 5.1e dargestellte Anwendung zum Stecken von Schaltdrähten.

Tabelle 5.6: Unterstützte Kraft-Momenten-Sensoren und ihre Schnittstellen

Sensor	Eingesetzte Modelle	Maximale Abtastfrequenz	Messbereich Fxy, Fz Mxy, Mz
ATI	Axia80-M20	7812 Hz	200 N, 360 N 8 Nm, 8 Nm
Optoforce (OnRobot)	HEX-E	500 Hz	200 N, 200 N 10 Nm, 6,5 Nm
Robotiq	FT300	100 Hz	300 N, 300 N 30 Nm, 30 Nm
WACOH-TECH	DynPick WEF-6A200-4	1000 Hz	200 N, 200 N 4 Nm, 4 Nm
Weiss Robotics	KMS40	500 Hz	120 N, 120 N 3 Nm, 3 Nm

Je nach verwendetem Skill kann ein Roboterwechsel Parameteränderungen bedingen. Dies ist beispielsweise bei PTP-Bewegungen der Fall, da die Achswinkelwerte von der Robotergeometrie abhängig sind. Aufgabenbeschreibungen auf Basis von Feature-Koordinaten sind vom eingesetzten Roboter hingegen unabhängig und müssen dahingehend nicht angepasst werden. Trotzdem können Parameteränderungen insbesondere bei kraftgeregelten Skills sinnvoll sein, da das Reglerverhalten von der Steifigkeit des Roboters abhängig ist. Um diese Abhängigkeit zu vermeiden, wird von Halt et al. (2019) ein Kraftregler vorgestellt und experimentell validiert, der einen Kontakt zwischen Roboter und Werkstücken herstellt und unabhängig vom eingesetzten Roboter und den vorliegenden Materialien ein vergleichbares Verhalten aufweist.

Trotz dieser Ansätze lässt sich nicht jede Aufgabe auf beliebige Roboter übertragen. Jede Roboterkinematik besitzt andere mechanische Eigenschaften wie Reichweite, Traglast, Steifigkeiten und Lage der Singularitäten, die eine Ausführung bestimmter Aufgaben mitunter verhindern können. Vielmehr soll hier hervorgehoben werden, dass derjenige Teil der Aufgabenspezifikation übertragen werden kann, der von den genannten Eigenschaften unabhängig ist. Dies ist bereits ein wichtiger Schritt hin zu einer größeren Wiederverwendbarkeit von Aufgabenspezifikationen.

Wechsel des Sensors

Der Wechsel eines Sensors erfolgt weitgehend analog zum Wechsel eines Roboters. Tabelle 5.6 stellt dazu die untersuchten Sensoren und ihre wichtigsten Eigenschaften gegenüber. Wie in

Codebeispiel 5.4: Definition des eingesetzten Sensors

```
<clone id="robot_ur5" prototype="robot_ur">
    <!-- ... -->
    <member id="components">
        <!-- Parametrierung des Sensors -->
        <clone id="force_sensor" prototype="robotiq_ft">
            <member id="wrench_topic">/robotiq/wrench</member>
            <member id="namespace">/robotiq</member>
        </clone>
    </member>
</clone>
```

DSL-Codebeispiel 5.4 dargestellt, besitzen auch Kraftsensoren implementierungsspezifische Parameter für den ROS Namensraum des Sensortreibers sowie für das Topic, über das Kraftwerte empfangen werden sollen. Der `force` Skill, als Prototyp für kraftgeregelte Skills, benötigt dieses Topic und referenziert den entsprechenden Parameter standardmäßig:

```
<reference id="wrench_topic" reference_id="robot.components.force_sensor.wrench_topic"/>
```

Abbildung 4.8 stellt diese Verknüpfung bildlich dar. Für einzelne Skills lässt sich die Einstellung überschreiben, sollen Kraftwerte einer anderen Quelle zum Einsatz kommen.

Die Spezifikation kraftgeregelter Aufgaben ist vom eingesetzten Sensor unabhängig, da bereits die Datenquellenelemente die Herkunft der Kraftwerte abstrahieren, wie in Abschnitt 3.3.1 beschrieben wird. Es können jedoch auch hier Parameteränderungen aufgrund unterschiedlicher Steifigkeiten der Sensoren erforderlich sein. So ist beispielsweise der eingesetzte Optoforce Sensor durch sein optisches Messsystem um zwei Größenordnungen nachgiebiger ($1{,}18 \cdot 10^5\,\mathrm{N/m}$ in F_{xy}) als klassische Sensoren mit Dehnmessstreifen ($2{,}7 \cdot 10^7\,\mathrm{N/m}$ in F_{xy} bei ATI Axia80).

Darüber hinaus unterscheiden sich die einzelnen Sensoren auch hinsichtlich Messrauschen, Messrate und Störeinflüssen. So kann es beispielsweise bei einem KMS40 über die Dauer einer Stunde zu einem Drift von 3 N kommen. Diese Eigenschaften können, müssen aber nicht zwangsläufig Parameteränderungen erfordern. Um Störeinflüsse zu reduzieren, reicht in vielen Fällen eine Tarierung des Sensors vor kraftgeregelten Prozessen aus. Bietet ein Sensor eine integrierte Tarierfunktion, so ist dafür ein Skill definiert, der diese aufruft:

```
<clone prototype="robotiq_ft_tare">
    <member id="duration">0.5</member>
</clone>
```

Eine Wartezeit vor dem Tarieren lässt sich dabei angeben, um gegebenenfalls ein Abbremsen und Nachschwingen des Roboters beziehungsweise des Werkzeugs aus der vorherigen Bewegung zu berücksichtigen. Für andere Sensoren wurde eine Schwerkraftkompensation implementiert[12], die ebenso einfach tariert werden kann:

```
<clone prototype="tare_g_compensator">
    <member id="duration">0.5</member>
</clone>
```

[12] `https://github.com/ipa320/g_compensator`, quelloffen unter MIT-Lizenz

Tabelle 5.7: Unterstützte Greifer und ihre Schnittstellen (die mit * markierten Werte lassen sich nicht explizit vorgeben)

Greifer	Eingesetzte Modelle	Hub	Geschwindigkeit	Kraft
Franka Emika	Panda Gripper	80 mm	30 mm/s	70 N
Robotiq	2F-85	(85 mm)*	(20-150 mm/s)*	(20-235 N)*
Schunk	EGL 90-CN	85 mm	150 mm/s	(50-600 N)*
Weiss Robotics	WSG 50-110	110 mm	5-420 mm/s	5-80 N
	IEG 55-020	20 mm	300 mm/s	10-30 N
	IEG 76-030	30 mm	200 mm/s	75-200 N

Analog zur Betrachtung der Roboterunabhängigkeit ist die Spezifikation einfacher Skills vom eingesetzten Sensor unabhängig, jedoch muss die Parametrierung aufgrund unterschiedlicher Steifigkeiten, Messraten und Störeinflüsse gegebenenfalls angepasst werden. Ohne eine verallgemeinerte Tarierfunktion sind zudem Anwendungen oder Macro-Skills (wie beispielsweise ein Skill zur Hutschienenbestückung) vom eingesetzten Sensor abhängig, da ein sensorspezifischer Tarier-Skill hinzugefügt werden muss. Eine Abstraktion der Tarierfunktion ließe sich allerdings aufseiten der Implementierung durchaus vornehmen.

Wechsel des Greifers

Zuletzt wird die Hardwareabhängigkeit der Aufgabenspezifikation anhand eines Greiferwechsels betrachtet. Tabelle 5.7 führt dazu die untersuchten Greifer und ihre Eigenschaften auf. Der angegebene Hub bezieht sich dabei stets auf den kompletten Hub mit beiden Backen.

Das DSL-Codebeispiel 5.5 stellt beispielhafte Parametrierungen verschiedener Greif-Skills dar. Wie hierbei ersichtlich ist, fallen bei elektrischen Greifern die Unterschiede der Schnittstellen sehr groß aus. Während Zielposition und Greifgeschwindigkeit für die meisten Greifer direkt angegeben werden können, ist dies für den Robotiq 2-Finger-Greifer nur indirekt möglich. Beim Initialisieren des Greifers ist eine Referenzfahrt auszuführen, wobei die minimale und die maximale Greiferöffnung bestimmt werden. Anschließend erfolgt die Positionierung mit Werten zwischen 0 und 255 linear zwischen den bestimmten Grenzwerten. Die Geschwindigkeit wird ebenfalls mit einem Wert zwischen 0 und 255 definiert, wobei 0 der minimal und 255 der maximal möglichen Geschwindigkeit entsprechen.

Bei der Greifkraft zeichnet sich ein ähnliches Bild ab. Während die Schnittstellen von Franka Emika und Weiss Robotics eine direkte Angabe der Greifkraft ermöglichen, ist dies über

Codebeispiel 5.5: Parametrierung von Greif-Skills unterschiedlicher Greifertypen

```
<!-- Weiss Robotics WSG 50-110 -->
<clone prototype="weiss_wsg50_grip">
    <member id="position">10.0</member>
    <member id="speed">420</member>
</clone>

<!-- Franka Emika Panda Gripper -->
<clone prototype="panda_gripper_grip">
    <member id="position">10.0</member>
    <member id="epsilon_inner">20.0</member>
    <member id="epsilon_outer">20.0</member>
    <member id="speed">100.0</member>
    <member id="force">100.0</member>
</clone>

<!-- Robotiq 2F-85 -->
<clone prototype="robotiq_2fg_grip">
    <member id="position">200</member>
    <member id="speed">255</member>
    <member id="force">255</member>
</clone>

<!-- Schunk EGL 90-CN -->
<clone prototype="schunk_egl90_grip">
    <member id="current">1</member>
    <member id="speed">-100</member>
</clone>
```

die Schnittstellen anderer Hersteller nicht explizit möglich. So kann die Greifkraft des EGL 90-CN nur indirekt über den Motorstrom beeinflusst werden. Der Zusammenhang zwischen Greifkraft und Motorstrom ist bei gleichbleibender Geschwindigkeit zwar linear, eine höhere Geschwindigkeit führt jedoch zu leicht höheren Greifkräften. Beim Robotiq 2-Finger-Greifer ist die Greifkraft sowohl von der Geschwindigkeits- als auch von der Krafteinstellung abhängig. Hier ist der Zusammenhang zudem nichtlinear und vom gegriffenen Material abhängig. Eine weitere Besonderheit ergibt sich bei der Schnittstelle des Weiss Robotics WSG 50-110. Hier ist die Einstellung der Greifkraft durch einen separaten Funktionsaufruf vorzunehmen. Ist die Greifkraft über die gesamte Anwendung konstant, so genügt ein einzelner Aufruf zu Beginn der Anwendung. Der Aufruf ist über einen eigens definierten Skill möglich:

```
<clone prototype="weiss_wsg50_set_force">
    <member id="force">80</member>
</clone>
```

Darüber hinaus bieten einzelne Greiferschnittstellen Zusatzfunktionen an, die genutzt werden können, um Anwendungen fehlertoleranter zu gestalten. So kann beim Franka Emika Panda und Weiss Robotics WSG 50-110 ein Toleranzbereich für das zu greifende Bauteil angegeben werden. Tritt eine Greifkraft bereits vor diesem Bereich auf, beispielsweise aufgrund eines falschen Bauteils oder eines sich im Weg befindlichen Fremdkörpers, so lässt sich ein Fehler signalisieren und mittels einer Stoppbedingung darauf reagieren. Tritt hingegen keine Kraft bis zum Ende des Toleranzbereichs auf, so lässt sich signalisieren, dass kein Bauteil gegriffen werden konnte. Bei Verwendung des EGL 90-CN kann nach dem Greifen des Bauteils eine elektrische

Haltebremse aktiviert werden, um einer Überhitzung vorzubeugen, die insbesondere bei großen Greifkräften und langer Greifdauer durch die hohen Motorströme auftreten kann.

Aus den hier aufgeführten Gründen ist ein Wechsel des Greifers nur mit den entsprechenden Änderungen der Aufgabenspezifikation möglich. Eine Abstraktion der Hardware wäre zwar im Hinblick auf die Übertragbarkeit der Parametrierung von Greif-Skills hilfreich, jedoch aufgrund der stark unterschiedlichen Funktionsumfänge vorhandener Greiferschnittstellen mit dem aktuellen Stand der Technik schwer erreichbar und mitunter einschränkend. Da Greif-Skills getrennt von anderen Skills definiert und parametriert werden, ermöglicht es die skill-basierte Struktur des Systems allerdings, nötige Änderungen der Aufgabenspezifikation lokal zu halten und somit leichter vornehmen zu können.

Zusammenfassung

Die Untersuchungen der vorangegangenen Abschnitte zeigen, dass mit dem vorgestellten Skill-Modell eine Skill-Bibliothek auf effiziente Weise aufgebaut werden kann und sich diese auf verschiedenartige Montageprozesse anwenden lässt. Die große Tiefe und Breite der Skill-Hierarchie belegt, dass Skills dabei in weitem Maße aufeinander aufbauen können. Zehn betrachtete Anwendungsfälle demonstrieren, wie sich Skills flexibel durch ihre Parametrierung sowie das Hinzufügen von Stoppbedingungen und Skripten an spezifische Produktvarianten und Prozesse anpassen lassen. Bei der Umsetzung von Anwendungen sind zudem ein rechtzeitiges Erkennen von Abweichungen und die Robustheit gegenüber Fertigungs- und Lagetoleranzen essenziell. Es wird demonstriert, wie die Erkennung von Fehlerfällen flexibel über das Hinzufügen von Stoppbedingungen modelliert werden kann.

Durch den exemplarischen Wechsel von Roboter, Sensor und Greifer zeigt sich, dass die Definition einfacher Skills zwar von den eingesetzten Hardwarekomponenten unabhängig ist, deren Parametrierung aber gegebenenfalls angepasst werden muss. Auch gibt es aufgrund unterschiedlicher Schnittstellen verschiedene Skills zur Ansteuerung von Sensoren und insbesondere Greifern, die eine hardwareunabhängige Definition von Macro-Skills einschränken. Eine weitere Grenze des vorgestellten Systems ist die Notwendigkeit, Skills und Stoppbedingung händisch parametrieren zu müssen. Auch wenn dies durch das Skill-Modell effizient geschehen mag, wäre es wünschenswert, Parameter automatisiert in einer Simulation oder basierend auf realen Daten zu lernen. Die sich daraus ergebenden Fragestellungen sind die Basis aktueller Untersuchungen (El-Shamouty et al. 2019). Darüber hinaus sind auch Regler vorzuziehen, die unabhängig von der eingesetzten Hardware ein vergleichbares Verhalten aufweisen (Halt et al. 2019) und somit keine hardwarespezifische Parametrierung benötigen.

6 Zusammenfassung und Ausblick

Die Motivation dieser Arbeit beruht darauf, den Einsatz von Robotern für Montageanwendungen durch eine effiziente Programmierung zu vereinfachen. Klassische Positions- und Bahnsteuerungen stoßen hier an ihre Grenzen, sei es im Hinblick auf den Einbezug von Sensordaten zur Programmierung robuster kraftgeregelter Montageprozesse oder die effiziente Handhabung von Prozess- und Variantenvielfalt. Um die genannten Herausforderungen zu adressieren, wird in dieser Arbeit ein skill-basierter Ansatz gewählt. Im Vordergrund steht dabei insbesondere, ein hohes Maß an Wiederverwendbarkeit zu ermöglichen, um neue Bauteilvarianten, Prozesse oder Hardwarekomponenten effizient abbilden zu können. Es zeigt sich, dass hierfür nicht nur Skills als Ganzes, sondern auch ihre Bestandteile und Parametrierung wiederverwendbar sein müssen. Entsprechend wird das Skill-Modell als Baukastensystem entworfen, das die Komposition und inkrementelle Erweiterung von Skills erlaubt.

Die elementaren Bausteine, aus denen sich Skills zusammensetzen, werden in verschiedene Teilaspekte getrennt: die Hardwareabstraktion zur Einbindung verschiedener Datenquellen wie Roboter und Sensoren, die Modellierung der geometrischen Beziehungen zwischen zu manipulierenden Werkstücken, die Aufgabenspezifikation mit Sollwerten, Reglern, Stoppbedingungen und Skripten sowie die Koordination der erstellten Skills. Als Ausführungsmechanismus wird der iTaSC Formalismus gewählt, der die Kombination von Skills mit partiellen Aufgabenbeschreibungen erlaubt, die jeweils nur einzelne Freiheitsgrade einschränken. Es zeigt sich, dass auf diesem Weg Skills mit Positions-, Geschwindigkeits- und Kraftregelung einfach kombiniert werden können. Zur Koordination der Skills werden Statecharts verwendet. Das Skill-Modell und der Ausführungsmechanismus erlauben es damit, Skills parallel, sequenziell oder als hierarchische Zustandsmaschine auszuführen und so auch Montageprozesse mit komplexen Bewegungsabfolgen abzubilden.

Um nicht jeden Skill von Grund auf neu zusammensetzen und parametrieren zu müssen, können Skills mittels Komposition und prototypbasierter Vererbung auf bestehenden Skills aufbauen und diese wiederverwenden und erweitern. Parametrierung, Komposition und prototypbasierte Vererbung der Bausteine erfolgen dabei in einem generischen Parameterbaum. Vom einfachen String oder Zahlenwert bis hin zu Skills und vollständigen Anwendungen werden darin alle Daten des Skill-Modells einheitlich als Parameter repräsentiert. Dies erlaubt es, die genannten

Methoden auf einfache und konsistente Weise auf alle Arten von Parametern gleichermaßen anzuwenden. Um Skills und Anwendungen ohne Programmierung von Quelltext definieren zu können, wird eine domänenspezifische Sprache zur Aufgabenspezifikation vorgestellt. Von Vorteil erweist sich dabei die Verwendung eines einheitlichen Modells sowohl für die Definition der domänenspezifischen Sprachelemente, die Modellierung der Skills als auch deren Parametrierung und Einsatz. Hierdurch werden Grenzen zwischen anwendungsspezifischen und -unabhängigen Teilen vermieden. Es vereinfacht sich die Verallgemeinerung anwendungsspezifischer Modelle und somit deren Wiederverwendung für weitere Anwendungen. Zudem lassen sich allgemein gestaltete Bausteine ohne Umwege in konkrete Anwendungen einbinden.

Alle Skills und Bausteine werden in einer auf Erweiterbarkeit ausgelegten Bibliothek zusammengefasst. Anhand von zehn exemplarischen Montageanwendungen wird gezeigt, dass bereits eine kleine Anzahl an Skills die Basis für eine Vielzahl von Anwendungsfällen bilden kann und es darüber hinaus sehr einfach ist, weitere Skills mittels vorhandener Bausteine zu erstellen, um komplexere Anwendungsfälle umzusetzen. Durch die explizite Modellierung geometrischer Merkmale der Werkstücke und die Parametrierbarkeit der Skills vereinfacht das Skill-Modell zudem die Abbildung von Produktvarianten. Anhand der Bestückung von Hutschienen wird die Verwendung von Stoppbedingungen zur Programmierung robuster Montageprozesse demonstriert.

Die Aufgabenspezifikation ist in vielen Bereichen bereits von den eingesetzten Hardwarekomponenten unabhängig, wodurch eine einfache Übertragung einmal erstellter Skills und Anwendungen auf neue Hardwarezusammensetzungen möglich wird. Forschungsbedarf besteht hingegen insbesondere bei der Parametrierung von Kraftreglern, die noch weitestgehend abhängig von Roboter- und Umgebungseigenschaften ist. Von großem Interesse ist zudem, die fortschreitenden Erkenntnisse des maschinellen Lernens für die Parametrierung und Komposition von Skills zu nutzen. Dies kann sowohl durch das Beobachten und Nachahmen des Menschen, Lernen in der Simulation als auch Lernen in realen Umgebungen geschehen. Das vorgestellte Skill-Modell bietet hier die Chance, durch seine Struktur den Bedarf an Daten zu reduzieren.

Zur Verfeinerung des Systems lassen sich Konzepte anderer Arbeiten integrieren. Die Angabe eines Sollwertbereichs anstelle eines einzelnen Sollwerts ermöglicht es, gerade erfüllte Zwangsbedingungen zeitweise zu deaktivieren, um sekundäre Kriterien zu optimieren. Eine adaptive Selektionsmatrix erlaubt den schnellen Wechsel zwischen Kraft- und Positionsregelung und damit eine größere Reaktionsfähigkeit auf sich verändernde Umgebungsbedingungen. Zudem lassen sich gegebenenfalls unter Inkaufnahme einer höheren konzeptionellen Komplexität über Mehrfachvererbung oder den Einsatz von Mixins der Grad der Wiederverwendung innerhalb des Skill-Modells weiter erhöhen.

Literatur

Aertbeliën et al. 2014
Aertbeliën, Erwin; De Schutter, Joris, 2014.
eTaSL/eTC: A constraint-based task specification language and robot controller using expression graphs.
In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.
Chicago, IL, USA, 14. 09. 2014–18. 09. 2014, S. 1540–1546.
DOI: 10.1109/IROS.2014.6942760

Aertbeliën 2009
Aertbeliën, Erwin, 2009.
Development and acquisition of skills for deburring with kinematically redundant robots.
ISBN 978-94-6018-090-3.
Leuven, Katholieke Universiteit Leuven, Netherlands, Diss., 2009

Arbo et al. 2018
Arbo, Mathias Hauan; Pane, Yudha; Aertbeliën, Erwin; Decré, Wilm, 2018.
A system architecture for constraint-based robotic assembly with CAD information.
In: *IEEE International Conference on Automation Science and Engineering (CASE)*.
Muenchen, 20. 08. 2018–24. 08. 2018, S. 690–696.
DOI: 10.1109/COASE.2018.8560450

Atkinson et al. 2003
Atkinson, Colin; Kühne, Thomas, 2003.
Model-driven development: A metamodeling foundation.
IEEE Software, **20** (5), S. 36–41.
DOI: 10.1109/MS.2003.1231149

Bartels et al. 2013
Bartels, Georg; Kresse, Ingo; Beetz, Michael, 2013.
Constraint-based movement representation grounded in geometric features.
In: *IEEE-RAS International Conference on Humanoid Robots (Humanoids)*.
Atlanta, GA, USA, 15. 10. 2013–17. 10. 2013, S. 547–554.
DOI: 10.1109/HUMANOIDS.2013.7030027

Bézivin et al. 2001
Bézivin, Jean; Gerbé, Olivier, 2001.
Towards a precise definition of the OMG/MDA framework.
In: *16th Annual International Conference on Automated Software Engineering (ASE)*.
San Diego, CA, USA, 26. 11. 2001–29. 11. 2001, S. 273–280.
DOI: 10.1109/ASE.2001.989813

Björkelund et al. 2011
Björkelund, Anders; Malec, Jacek; Nilsson, Klas; Nugues, Pierre, 2011.
Knowledge and skill representations for robotized production.
IFAC Proceedings Volumes, **44** (1), S. 8999–9004.
DOI: 10.3182/20110828-6-IT-1002.01053

Borning 1986
Borning, Alan, 1986.
Classes versus prototypes in object-oriented languages.
In: *ACM Fall joint computer conference*.
Dallas, USA, 02. 11. 1986–06. 11. 1986, S. 36–40

Brooks 1986
Brooks, Rodney A., 1986.
A robust layered control system for a mobile robot.
IEEE Journal on Robotics and Automation, **2** (1), S. 14–23.
DOI: 10.1109/JRA.1986.1087032

Bruyninckx et al. 2000
Bruyninckx, Herman; Khatib, Oussama, 2000.
Gauss' principle and the dynamics of redundant and constrained manipulators.
In: *IEEE International Conference on Robotics and Automation (ICRA)*.
San Francisco, CA, USA, 24. 04. 2000–28. 04. 2000, S. 2563–2568.
DOI: 10.1109/ROBOT.2000.846414

Bruyninckx et al. 2013
Bruyninckx, Herman; Klotzbücher, Markus; Hochgeschwender, Nico; Kraetzschmar, Gerhard; Gherardi, Luca; Brugali, Davide, 2013.
The BRICS component model.
In: *28th Annual ACM Symposium on Applied Computing*.
Coimbra, Portugal, 18. 03. 2013–22. 03. 2013, S. 1758–1764.
DOI: 10.1145/2480362.2480693

Bruyninckx et al. 1996
Bruyninckx, Herman; De Schutter, Joris, 1996.
Specification of force-controlled actions in the task frame formalism - a synthesis.
IEEE Transactions on Robotics and Automation, **12** (4), S. 581–589.
DOI: 10.1109/70.508440

Bruyninckx 2001
Bruyninckx, Herman, 2001.
Open robot control software: the OROCOS project.
In: *IEEE International Conference on Robotics and Automation (ICRA).*
Seoul, South Korea, 21. 05. 2001–26. 05. 2001, S. 2523–2528.
DOI: 10.1109/ROBOT.2001.933002

Brugali 2015
Brugali, Davide, 2015.
Model-driven software engineering in robotics.
IEEE Robotics & Automation Magazine, **22** (3), S. 155–166.
DOI: 10.1109/MRA.2015.2452201

Bubeck et al. 2014
Bubeck, Alexander; Weißhardt, Florian; Verl, Alexander, 2014.
BRIDE – A toolchain for framework-independent development of industrial service robot applications.
In: *45th International Symposium on Robotics (ISR).*
Muenchen, 02. 06. 2014–03. 06. 2014, S. 1–6.
Berlin: VDE-Verlag.
ISBN 978-3-8007-3601-0

Butting et al. 2015
Butting, Arvid; Rumpe, Bernhard; Schulze, Christoph; Thomas, Ulrike; Wortmann, Andreas, 2015.
Modeling reusable, platform-independent robot assembly processes.
International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob)

Chiaverini et al. 2016
Chiaverini, Stefano; Oriolo, Giuseppe; Maciejewski, Anthony A., 2016.
Redundant robots.
In: Siciliano, Bruno; Khatib, Oussama (Hrsg.): *Springer Handbook of Robotics*,
S. 221–242.
Berlin, Heidelberg: Springer.
ISBN 978-3-319-32552-1.
DOI: 10.1007/978-3-319-32552-1_10

Chiaverini et al. 1991
Chiaverini, Stefano; Egeland, Olav; Kanestrom, Rakel K., 1991.
Achieving user-defined accuracy with damped least-squares inverse kinematics.
In: *IEEE International Conference on Advanced Robotics (ICAR)*.
Pisa, Italy, 19. 06. 1991–22. 06. 1991, S. 672–677.
DOI: 10.1109/ICAR.1991.240676

Chiaverini et al. 1993
Chiaverini, Stefano; Sciavicco, Lorenzo, 1993.
The parallel approach to force/position control of robotic manipulators.
IEEE Transactions on Robotics and Automation, **9** (4), S. 361–373.
DOI: 10.1109/70.246048

Chiaverini et al. 1994
Chiaverini, Stefano; Siciliano, Bruno; Egeland, Olav, 1994.
Review of the damped least-squares inverse kinematics with experiments on an industrial robot manipulator.
IEEE Transactions on Control Systems Technology, **2** (2), S. 123–134.
DOI: 10.1109/87.294335

Craig 2005
Craig, John J., 2005.
Introduction to robotics: Mechanics and control.
3. Auflage.
Upper Saddle River, N.J.: Pearson/Prentice Hall.
ISBN 978-0201543612

De Schutter et al. 2007
De Schutter, Joris; De Laet, Tinne; Rutgeerts, Johan; Decré, Wilm; Smits, Ruben; Aertbeliën, Erwin; Claes, Kasper; Bruyninckx, Herman, 2007.
Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty.
International Journal of Robotics Research, **26** (5), S. 433–455.
DOI: 10.1177/0278364907078091O7

De Laet et al. 2012
De Laet, Tinne; Schaekers, Wouter; de Greef, Jonas; Bruyninckx, Herman, 2012.
Domain specific language for geometric relations between rigid bodies targeted to robotic applications.
International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob)

De Schutter et al. 1988
De Schutter, Joris; Van Brussel, Hendrik, 1988.
Compliant robot motion II. A control approach based on external control loops.
The International Journal of Robotics Research, **7** (4), S. 18–33.
DOI: 10.1177/027836498800700402

Denso Wave Inc. 2017
Denso Wave Inc., 2017.
DENSO b-CAP Communication Specifications for RC8 – Version 1.0.4.
Verfügbar unter: https://www.densorobotics.com/user-manuals/img/001511/b-CAP_Guide_RC8_en.pdf
Zugriff am: 08. 10. 2020

Doty et al. 1993
Doty, Keith L.; Melchiorri, Claudio; Bonivento, Claudio, 1993.
A theory of generalized inverses applied to robotics.
The International Journal of Robotics Research, **12** (1), S. 1–19.
DOI: 10.1177/027836499301200101

da Silva 2015
Da Silva, Alberto Rodrigues, 2015.
Model-driven engineering: A survey supported by the unified conceptual model.
Computer Languages, Systems & Structures, **43**, S. 139–155.
DOI: 10.1016/j.cl.2015.06.001

Duffy 1990
Duffy, Joseph, 1990.
The fallacy of modern hybrid control theory that is based on orthogonal complements of twist and wrench spaces.
Journal of robotic systems, **7** (2), S. 139–144.
DOI: 10.1002/rob.4620070202

Ehlers et al. 2019
Ehlers, Dennis; Suomalainen, Markku; Lundell, Jens; Kyrki, Ville, 2019.
Imitating human search strategies for assembly.
In: *IEEE International Conference on Robotics and Automation (ICRA)*.
Montreal, QC, Canada, 20. 05. 2019–24. 05. 2019, S. 7821–7827.
DOI: 10.1109/ICRA.2019.8793780

El-Shamouty et al. 2019
El-Shamouty, Mohamed; Kleeberger, Kilian; Lämmle, Arik; Huber, Marco, 2019.
Simulation-driven machine learning for robotics and automation.
tm-Technisches Messen, **86** (11), S. 673–684.
DOI: 10.1515/teme-2019-0072

Feldmann et al. 2014
Feldmann, Klaus; Schöppner, Volker; Spur, Günter (Hrsg.), 2014.
Handbuch Fügen, Handhaben, Montieren.
2., vollständig neu bearbeitete Auflage.
München: Carl Hanser Verlag.
ISBN 978-3-446-42827-0

Finkemeyer et al. 2005
Finkemeyer, Bernd; Kröger, Torsten; Wahl, Friedrich M., 2005.
Executing assembly tasks specified by manipulation primitive nets.
Advanced Robotics, **19** (5), S. 591–611.
DOI: 10.1163/156855305323383811

Finkemeyer 2004
Finkemeyer, Bernd, 2004.
Robotersteuerungsarchitektur auf der Basis von Aktionsprimitiven.
Aachen: Shaker.
ISBN 978-3832228934.
Technische Universität Braunschweig, Diss

Fowler et al. 2010
Fowler, Martin; Parsons, Rebecca, 2010.
Domain-specific languages.
Upper Saddle River, N.J.: Addison-Wesley.
ISBN 978-0321712943

Gajewski et al. 2019
Gajewski, Paweł; Ferreira, Paulo; Bartels, Georg; Wang, Chaozheng; Guerin, Frank; Indurkhya, Bipin; Beetz, Michael; Śnieżyński, Bartłomiej, 2019.
Adapting everyday manipulation skills to varied scenarios.
In: *IEEE International Conference on Robotics and Automation (ICRA).*
Montreal, QC, Canada, 20. 05. 2019–24. 05. 2019, S. 1345–1351.
DOI: 10.1109/ICRA.2019.8793590

Gamma et al. 1994
Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John, 1994.
Design patterns: Elements of reusable object-oriented software.
Boston, Mass.: Addison-Wesley.
ISBN 978-0201633610

Gat 1998
Gat, Erann, 1998.
Three-layer architectures.
In: Kortenkamp, David; Bonasso, R. Peter; Murphy, Robin (Hrsg.): *Artificial Intelligence and Mobile Robots*,
S. 195–210.
Cambridge, MA, USA: MIT Press.
ISBN 978-0-262-61137-4

Hägele et al. 2016
Hägele, Martin; Nilsson, Klas; Pires, J. Norberto; Bischoff, Rainer, 2016.
Industrial robotics.
In: Siciliano, Bruno; Khatib, Oussama (Hrsg.): *Springer Handbook of Robotics*,
S. 1385–1422.
Berlin, Heidelberg: Springer.
ISBN 978-3-319-32552-1.
DOI: 10.1007/978-3-540-30301-5_43

Halt et al. 2018
Halt, Lorenz; Tenbrock, Philipp; Nägele, Frank; Pott, Andreas, 2018.
On the implementation of transferable assembly applications for industrial robots.
In: *50th International Symposium on Robotics (ISR).*
Munich, Germany, 20. 06. 2018–21. 06. 2018.
Berlin: VDE-Verlag.
ISBN 978-3-8007-4699-6

Halt et al. 2019
Halt, Lorenz; Pan, Fengjunjie; Tenbrock, Philipp; Pott, Andreas; Seel, Thomas, 2019.
A transferable force controller based on prescribed performance for contact establishment in robotic assembly tasks.
In: *IEEE International Conference on Automation Science and Engineering (CASE).*
Vancouver, BC, Canada, 22. 08. 2019–26. 08. 2019, S. 830–835.
DOI: 10.1109/COASE.2019.8843020

Harel 1987
Harel, David, 1987.
Statecharts: A visual formalism for complex systems.
Science of Computer Programming, **8** (3), S. 231–274.
DOI: 10.1016/0167-6423(87)90035-9

Hasegawa et al. 1992
Hasegawa, Tsutomu; Suehiro, Takashi; Takase, Kunikatsu, 1992.
A model-based manipulation system with skill-based execution.
IEEE Transactions on Robotics and Automation, **8** (5), S. 535–544.
DOI: 10.1109/70.163779

Hogan 1985
Hogan, Neville, 1985.
Impedance Control: An Approach to Manipulation: Part I-III.
Journal of Dynamic Systems, Measurement, and Control, **107** (1), S. 1–24.
DOI: 10.1115/1.3140702

Hogan 1987
Hogan, Neville, 1987.
Stable execution of contact tasks using impedance control.
In: *1987 IEEE International Conference on Robotics and Automation*.
Raleigh, NC, USA, 31. 03. 1987–03. 04. 1987, S. 1047–1054.
DOI: 10.1109/ROBOT.1987.1087854

IFR 2018
International Federation of Robotics (IFR), 2018.
World Robotics 2018 Industrial Robots.
Frankfurt: VDMA Verlag

Johannsmeier et al. 2019
Johannsmeier, Lars; Gerchow, Malkin; Haddadin, Sami, 2019.
A framework for robot manipulation: Skill formalism, meta learning and adaptive control.
In: *IEEE International Conference on Robotics and Automation (ICRA)*.
Montreal, QC, Canada, 20. 05. 2019–24. 05. 2019, S. 5844–5850.
DOI: 10.1109/ICRA.2019.8793542

Klotzbücher et al. 2010
Klotzbücher, Markus; Soetens, Peter; Bruyninckx, Herman, 2010.
OROCOS RTT-Lua: An execution environment for building real-time robotic domain specific languages.
International Workshop on Dynamic languages for RObotic and Sensors (DYROS)

Klotzbücher et al. 2011
Klotzbücher, Markus; Smits, Ruben; Bruyninckx, Herman; De Schutter, Joris, 2011.
Reusable hybrid force-velocity controlled motion specifications with executable domain specific languages.
In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.
San Francisco, CA, USA, 25. 09. 2011–30. 09. 2011, S. 4684–4689.
DOI: 10.1109/IROS.2011.6094782

Klotzbücher et al. 2012
Klotzbücher, Markus; Biggs, Geoffrey; Bruyninckx, Herman, 2012a.
Pure coordination using the coordinator–configurator pattern.
International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob)

Klotzbücher et al. 2012 Klotzbücher, Markus; Bruyninckx, Herman, 2012b.
A lightweight, composable metamodelling language for specification and validation of internal domain specific languages.
In: *8th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES)*, S. 58–68.
DOI: 10.1007/978-3-642-38209-3_4

Klotzbücher et al. 2012 Klotzbücher, Markus; Bruyninckx, Herman, 2012c.
Coordinating robotic tasks and systems with rFSM statecharts.
Journal of Software Engineering for Robotics (JOSER), **3** (1), S. 28–56

Klotzbücher 2013 Klotzbücher, Markus, 2013.
Domain specific languages for hard real-time safe coordination of robot and machine tool systems.
ISBN 978-94-6018-645-5.
Leuven, Katholieke Universiteit Leuven, Netherlands, Diss., 2013

Kortenkamp et al. 2016 Kortenkamp, David; Simmons, Reid; Brugali, Davide, 2016.
Robotic systems architectures and programming.
In: Siciliano, Bruno; Khatib, Oussama (Hrsg.): *Springer Handbook of Robotics*, S. 283–306.
Berlin, Heidelberg: Springer.
ISBN 978-3-319-32550-7.
DOI: 10.1007/978-3-319-32552-1_12

Krahn et al. 2010 Krahn, Holger; Rumpe, Bernhard; Völkel, Steven, 2010.
MontiCore: A framework for compositional development of domain specific languages.
International journal on software tools for technology transfer, **12** (5), S. 353–372.
DOI: 10.1007/s10009-010-0142-1

Kresse et al. 2012
Kresse, Ingo; Beetz, Michael, 2012.
Movement-aware action control — Integrating symbolic and control-theoretic action execution.
In: *IEEE International Conference on Robotics and Automation (ICRA)*.
Saint Paul, MN, USA, 14. 05. 2012–18. 05. 2012, S. 3245–3251.
DOI: 10.1109/ICRA.2012.6225119

Kresse 2017
Kresse, Ingo, 2017.
A semantic constraint-based robot motion control for generalizing everyday manipulation actions.
München, Technische Universität München, Diss., 2017

Kröger et al. 2008
Kröger, Torsten; Finkemeyer, Bernd; Winkelbach, Simon; Eble, Lars-Oliver; Molkenstruck, Sven; Wahl, Friedrich, 2008.
A manipulator plays Jenga.
IEEE Robotics & Automation Magazine, **15** (3), S. 79–84.
DOI: 10.1109/MRA.2008.921547

Kröger et al. 2010
Kröger, Torsten; Finkemeyer, Bernd; Wahl, Friedrich M., 2010.
Manipulation Primitives — A universal interface between sensor-based motion control and robot programming.
In: Siciliano, Bruno; Khatib, Oussama; Groen, Frans; Schütz, Daniel; Wahl, Friedrich M. (Hrsg.): *Robotic Systems for Handling and Assembly*,
S. 293–313.
Springer Tracts in Advanced Robotics.
DOI: 10.1007/978-3-642-16785-0_17

KUKA Roboter GmbH 2010
KUKA Roboter GmbH, 2010.
KUKA.RobotSensorInterface – Version RSI 3.1 V1 de

Lieberman 1986
Lieberman, Henry, 1986.
Using prototypical objects to implement shared behavior in object-oriented systems.
ACM SIGPLAN Notices, **21** (11), S. 214–223.
DOI: 10.1145/28697.28718

Maciejewski et al. 1985 Maciejewski, Anthony A.; Klein, Charles A., 1985. Obstacle avoidance for kinematically redundant manipulators in dynamically varying environments. *The International Journal of Robotics Research*, **4** (3), S. 109–117. DOI: 10.1177/027836498500400308

Maciejewski et al. 1988 Maciejewski, Anthony A.; Klein, Charles A., 1988. Numerical filtering for the operation of robotic manipulators through kinematically singular configurations. *Journal of Robotic Systems*, **5** (6), S. 527–552. DOI: 10.1002/rob.4620050603

Maciejewski et al. 1989 Maciejewski, Anthony A.; Klein, Charles A., 1989. The singular value decomposition: Computation and applications to robotics. *The International Journal of Robotics Research*, **8** (6), S. 63–79. DOI: 10.1177/027836498900800605

Mason 1981 Mason, Matthew T., 1981. Compliance and force control for computer controlled manipulators. *IEEE Transactions on Systems, Man, and Cybernetics*, **11** (6), S. 418–432. DOI: 10.1109/TSMC.1981.4308708

Mernik et al. 2005 Mernik, Marjan; Heering, Jan; Sloane, Anthony M., 2005. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, **37** (4), S. 316–344. DOI: 10.1145/1118890.1118892

Morrow et al. 1997 Morrow, J. Daniel; Khosla, Pradeep K., 1997. Manipulation task primitives for composing robot skills. In: *IEEE International Conference on Robotics and Automation (ICRA)*. Albuquerque, NM, USA, 25. 04. 1997, S. 3354–3359. DOI: 10.1109/ROBOT.1997.606800

Mosemann et al. 2001 Mosemann, Heiko; Wahl, Friedrich M., 2001.
Automatic decomposition of planned assembly sequences into skill primitives.
IEEE Transactions on Robotics and Automation, **17** (5), S. 709–718.
DOI: 10.1109/70.964670

Mühe et al. 2010 Mühe, Henrik; Angerer, Andreas; Hoffmann, Alwin; Reif, Wolfgang, 2010.
On reverse-engineering the KUKA Robot Language.
International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob)

Nägele et al. 2018 Nägele, Frank; Halt, Lorenz; Tenbrock, Philipp; Pott, Andreas, 2018.
A prototype-based skill model for specifying robotic assembly tasks.
In: *IEEE International Conference on Robotics and Automation (ICRA)*.
Brisbane, QLD, Australia, 21. 05. 2018–25. 05. 2018, S. 558–565.
DOI: 10.1109/ICRA.2018.8462885

Nägele et al. 2019 Nägele, Frank; Halt, Lorenz; Tenbrock, Philipp; Pott, Andreas, 2019.
Composition and incremental refinement of skill models for robotic assembly tasks.
In: *IEEE International Conference on Robotic Computing (IRC)*.
Naples, Italy, 25. 02. 2019–27. 02. 2019, S. 177–182.
DOI: 10.1109/IRC.2019.00034

Nakamura et al. 1986 Nakamura, Yoshihiko; Hanafusa, Hideo, 1986.
Inverse kinematic solutions with singularity robustness for robot manipulator control.
Journal of Dynamic Systems, Measurement, and Control, **108** (3), S. 163–171.
DOI: 10.1115/1.3143764

Nakamura et al. 1987
Nakamura, Yoshihiko; Hanafusa, Hideo; Yoshikawa, Tsuneo, 1987.
Task-priority based redundancy control of robot manipulators.
The International Journal of Robotics Research, **6** (2), S. 3–15.
DOI: 10.1177/027836498700600201

Nordmann et al. 2014
Nordmann, Arne; Hochgeschwender, Nico; Wrede, Sebastian, 2014.
A survey on domain-specific languages in robotics.
In: *International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*,
S. 195–206.
DOI: 10.1007/978-3-319-11900-7_17

Nordmann et al. 2016
Nordmann, Arne; Hochgeschwender, Nico; Wigand, Dennis; Wrede, Sebastian, 2016.
A survey on domain-specific modeling and languages in robotics.
Journal of Software Engineering for Robotics (JOSER), **7** (1), S. 75–99

Norm DIN 8593-1
DIN 8593-1:2003-09.
Fertigungsverfahren Fügen - Teil 1: Zusammensetzen; Einordnung, Unterteilung, Begriffe

OMG 2014
Object Management Group (OMG), 2014.
Model driven architecture (MDA) - MDA Guide rev. 2.0.
Verfügbar unter: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01
Zugriff am: 08. 10. 2020

Pott et al. 2019
Pott, Andreas; Dietz, Thomas, 2019.
Industrielle Robotersysteme: Entscheiderwissen für die Planung und Umsetzung wirtschaftlicher Roboterlösungen.
Springer-Verlag, Wiesbaden.
ISBN 978-3-658-25344-8.
DOI: 10.1007/978-3-658-25345-5

Quigley et al. 2009
Quigley, Morgan; Gerkey, Brian; Conley, Ken; Faust, Josh; Foote, Tully; Leibs, Jeremy; Berger, Eric; Wheeler, Rob; Ng, Andrew, 2009.
ROS: An open-source robot operating system.
In: *ICRA Workshop on Open Source Software*

Raibert et al. 1981
Raibert, Marc H.; Craig, John J., 1981.
Hybrid position/force control of manipulators.
Journal of Dynamic Systems, Measurement, and Control, **103** (2), S. 126–133.
DOI: 10.1115/1.3139652

Ramaswamy et al. 2014
Ramaswamy, Arunkumar; Monsuez, Bruno; Tapus, Adriana, 2014.
Model-driven software development approaches in robotics research.
In: *6th International Workshop on Modeling in Software Engineering*,
S. 43–48.
DOI: 10.1145/2593770.2593781

Samson et al. 1991
Samson, Claude; Le Borgne, Michel; Espiau, Bernard, 1991.
Robot control: The task function approach.
Oxford University Press, Oxford.
ISBN 978-0-19-853805-9.
DOI: 10.1017/S0263574700000643

Scherzinger et al. 2019
Scherzinger, Stefan; Roennau, Arne; Dillmann, Rüdiger, 2019.
Contact skill imitation learning for robot-independent assembly programming.
In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.
Macau, China, 03. 11. 2019–08. 11. 2019, S. 4309–4316.
DOI: 10.1109/IROS40897.2019.8967523

Siciliano et al. 2010
Siciliano, Bruno; Sciavicco, Lorenzo; Villani, Luigi; Oriolo, Giuseppe, 2010.
Robotics: Modelling, planning and control.
Springer-Verlag London.
ISBN 978-1-84628-641-4.
DOI: 10.1007/978-1-84628-642-1

Siciliano et al. 1991
Siciliano, Bruno; Slotine, Jean-Jacques E., 1991.
A general framework for managing multiple tasks in highly redundant robotic systems.
In: *IEEE International Conference on Advanced Robotics (ICAR)*.
Pisa, Italy, 19. 06. 1991–22. 06. 1991, S. 1211–1216.
DOI: 10.1109/ICAR.1991.240390

Silva et al. 2015
Silva, Leonardo Humberto; Ramos, Miguel; Valente, Marco Tulio; Bergel, Alexandre; Anquetil, Nicolas, 2015.
Does JavaScript software embrace classes?
In: *IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*.
Montreal, QC, Canada, 02. 03. 2015–06. 03. 2015, S. 73–82.
DOI: 10.1109/SANER.2015.7081817

Silva et al. 2017
Silva, Leonardo Humberto; Valente, Marco Tulio; Bergel, Alexandre; Anquetil, Nicolas; Etien, Anne, 2017.
Identifying classes in legacy JavaScript code.
Journal of Software: Evolution and Process, **29** (8).
DOI: 10.1002/smr.1864

Simpson 2014
Simpson, Kyle, 2014.
You don't know JS: this & object prototypes.
O'Reilly Media, Sebastopol, CA, USA.
ISBN 978-1491904152

Smits et al. 2008
Smits, Ruben; De Laet, Tinne; Claes, Kasper; Bruyninckx, Herman; De Schutter, Joris, 2008.
iTaSC: A tool for multi-sensor integration in robot manipulation.
In: *IEEE Multisensor Fusion and Integration for Intelligent Systems (MFI)*.
Seoul, South Korea, 20. 08. 2008–22. 08. 2008, S. 426–433.
DOI: 10.1109/MFI.2008.4648032

Smits et al. 2009
Smits, Ruben; Bruyninckx, Herman; De Schutter, Joris, 2009.
Software support for high-level specification, execution and estimation of event-driven, constraint-based multi-sensor robot tasks.
In: *International Conference on Advanced Robotics (ICAR)*.
Munich, Germany, 22. 06. 2009–26. 06. 2009, S. 1–6

Smits 2010
Smits, Ruben, 2010.
Robot skills: Design of a constraint-based methodology and software support.
ISBN 978-94-6018-204-4.
Leuven, Katholieke Universiteit Leuven, Netherlands, Diss., 2010

Stenmark et al. 2013
Stenmark, Maj; Nugues, Pierre, 2013.
Natural language programming of industrial robots.
In: *44th International Symposium on Robotics (ISR).*
Seoul, South Korea, 24. 10. 2013–26. 10. 2013, S. 1–5.
DOI: 10.1109/ISR.2013.6695630

Stenmark et al. 2014
Stenmark, Maj; Malec, Jacek, 2014a.
Describing constraint-based assembly tasks in unstructured natural language.
19th IFAC World Congress, **47** (3), S. 3056–3061.
DOI: 10.3182/20140824-6-ZA-1003.02062

Stenmark et al. 2014
Stenmark, Maj; Malec, Jacek; Stolt, Andreas, 2014b.
From high-level task descriptions to executable robot code.
In: *IEEE International Conference Intelligent Systems.*
Warsaw, Poland, 24. 09. 2014–26. 09. 2014, S. 189–202.
DOI: 10.1007/978-3-319-11310-4_17

Stenmark et al. 2015
Stenmark, Maj; Malec, Jacek, 2015.
Knowledge-based instruction of manipulation tasks for industrial robotics.
Robotics and Computer-Integrated Manufacturing, **33**, S. 56–67.
DOI: 10.1016/j.rcim.2014.07.004

Stenmark 2015
Stenmark, Maj, 2015.
Instructing industrial robots using high-level task descriptions.
Lund University, Lund, Sweden, Diss.

Stolt et al. 2011
Stolt, Andreas; Linderoth, Magnus; Robertsson, Anders; Johansson, Rolf, 2011a.
Force controlled assembly of emergency stop button.
In: *IEEE International Conference on Robotics and Automation (ICRA).*
Shanghai, China, 09. 05. 2011–13. 05. 2011, S. 3751–3756.
DOI: 10.1109/ICRA.2011.5979745

Stolt et al. 2011
Stolt, Andreas; Linderoth, Magnus; Robertsson, Anders; Jonsson, Marie; Murray, Thomas, 2011b.
Force controlled assembly of flexible aircraft structure.
In: *IEEE International Conference on Robotics and Automation (ICRA)*.
Shanghai, China, 09. 05. 2011–13. 05. 2011, S. 6027–6032.
DOI: 10.1109/ICRA.2011.5979962

Stolt et al. 2012
Stolt, Andreas; Linderoth, Magnus; Robertsson, Anders; Johansson, Rolf, 2012a.
Adaptation of force control parameters in robotic assembly.
10th IFAC Symposium on Robot Control, **45** (22), S. 561–566.
DOI: 10.3182/20120905-3-HR-2030.00033

Stolt et al. 2012
Stolt, Andreas; Linderoth, Magnus; Robertsson, Anders; Johansson, Rolf, 2012b.
Force controlled robotic assembly without a force sensor.
In: *IEEE International Conference on Robotics and Automation (ICRA)*.
Saint Paul, MN, USA, 14. 05. 2012–18. 05. 2012, S. 1538–1543.
DOI: 10.1109/ICRA.2012.6224837

Stolt et al. 2012
Stolt, Andreas; Linderoth, Magnus; Robertsson, Anders; Johansson, Rolf, 2012c.
Robotic Assembly Using a Singularity-Free Orientation Representation Based on Quaternions.
10th IFAC Symposium on Robot Control, **45** (22), S. 549–554.
DOI: 10.3182/20120905-3-HR-2030.00074

Stolt 2015
Stolt, Andreas, 2015.
On robotic assembly using contact force control and estimation.
ISBN 978-91-7623-457-0.
Lund University, Lund, Sweden, Diss.

Suomalainen et al. 2016
Suomalainen, Markku; Kyrki, Ville, 2016.
Learning compliant assembly motions from demonstration.
In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.
Daejeon, South Korea, 09. 10. 2016–14. 10. 2016, S. 871–876.
DOI: 10.1109/IROS.2016.7759153

Taivalsaari 1996
Taivalsaari, Antero, 1996.
On the notion of inheritance.
ACM Computing Surveys (CSUR), **28** (3), S. 438–479.
DOI: 10.1145/243439.243441

Thomas et al. 2003
Thomas, Ulrike; Finkemeyer, Bernd; Kröger, Torsten; Wahl, Friedrich M., 2003.
Error-tolerant execution of complex robot tasks based on skill primitives.
In: *IEEE International Conference on Robotics and Automation (ICRA)*.
Taipei, Taiwan, 14. 09. 2003–19. 09. 2003, S. 3069–3075.
DOI: 10.1109/ROBOT.2003.1242062

Thomas et al. 2013
Thomas, Ulrike; Hirzinger, Gerd; Rumpe, Bernhard; Schulze, Christoph; Wortmann, Andreas, 2013.
A new skill based robot programming language using UML/P statecharts.
In: *IEEE International Conference on Robotics and Automation (ICRA)*.
Karlsruhe, 06. 05. 2013–10. 05. 2013, S. 461–466.
DOI: 10.1109/ICRA.2013.6630615

Universal Robots A/S 2019
Universal Robots A/S, 2019.
Real-Time Data Exchange (RTDE) Guide – CB3 Software version: 3.4.
Verfügbar unter: https://www.universal-robots.com/articles/ur/real-time-data-exchange-rtde-guide/
Zugriff am: 08. 10. 2020

Van Deursen et al. 2000
Van Deursen, Arie; Klint, Paul; Visser, Joost, 2000.
Domain-specific languages: An annotated bibliography.
ACM SIGPLAN Notices, **35** (6), S. 26–36.
DOI: 10.1145/352029.352035

Vanthienen et al. 2013
Vanthienen, Dominick; Klotzbücher, Markus; De Schutter, Joris; De Laet, Tinne; Bruyninckx, Herman, 2013.
Rapid application development of constrained-based task modelling and execution using domain specific languages.
In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.
Tokyo, Japan, 03. 11. 2013–07. 11. 2013, S. 1860–1866.
DOI: 10.1109/IROS.2013.6696602

Vanthienen et al. 2014 Vanthienen, Dominick; Klotzbücher, Markus; Bruyninckx, Herman, 2014.
The 5C-based architectural composition pattern: Lessons learned from re-developing the iTaSC framework for constraint-based robot programming.
Journal of Software Engineering for Robotics (JOSER), **5** (1), S. 17–35

Vanthienen 2015 Vanthienen, Dominick, 2015.
Composition pattern for constraint-based programming with application to force-sensorless robot tasks.
ISBN 978-94-6018-931-9.
Leuven, Katholieke Universiteit Leuven, Netherlands, Diss., 2015

Vecerik et al. 2019 Vecerik, Mel; Sushkov, Oleg; Barker, David; Rothörl, Thomas; Hester, Todd; Scholz, Jon, 2019.
A practical approach to insertion with variable socket position using deep reinforcement learning.
In: *IEEE International Conference on Robotics and Automation (ICRA).*
Montreal, QC, Canada, 20. 05. 2019–24. 05. 2019, S. 754–760.
DOI: 10.1109/ICRA.2019.8794074

Vergara Perico et al. 2019 Vergara Perico, Cristian Alejandro; De Schutter, Joris; Aertbeliën, Erwin, 2019.
Combining imitation learning with constraint-based task specification and control.
IEEE Robotics and Automation Letters, **4** (2), S. 1892–1899.
DOI: 10.1109/LRA.2019.2898035

Villani et al. 2016 Villani, Luigi; De Schutter, Joris, 2016.
Force control.
In: Siciliano, Bruno; Khatib, Oussama (Hrsg.): *Springer Handbook of Robotics*,
S. 195–220.
Berlin, Heidelberg: Springer.
ISBN 978-3-319-32550-7.
DOI: 10.1007/978-3-319-32552-1_9

Wampler 1986 Wampler, Charles W., 1986.
Manipulator inverse kinematic solutions based on vector formulations and damped least-squares methods.
IEEE Transactions on Systems, Man, and Cybernetics, **16** (1), S. 93–101.
DOI: 10.1109/TSMC.1986.289285

Weidauer et al. 2014 Weidauer, Ingo; Kubus, Daniel; Wahl, Friedrich M., 2014.
A hierarchical extension of manipulation primitives and its integration into a robot control architecture.
In: *IEEE International Conference on Robotics and Automation (ICRA)*.
Hong Kong, China, 31. 05. 2014–07. 06. 2014, S. 5401–5407.
DOI: 10.1109/ICRA.2014.6907653

Whitney 1969 Whitney, Daniel E., 1969.
Resolved motion rate control of manipulators and human prostheses.
IEEE Transactions on Man-Machine Systems, **10** (2), S. 47–53.
DOI: 10.1109/TMMS.1969.299896

Yoshikawa 1985 Yoshikawa, Tsuneo, 1985.
Manipulability of robotic mechanisms.
The International Journal of Robotics Research, **4** (2), S. 3–9.
DOI: 10.1177/027836498500400201

Ausgehend von der Motivation, den Einsatz von Industrierobotern für Montageanwendungen zu erleichtern, wird in dieser Arbeit ein Skill-Modell zur Programmierung kraftgeregelter Montageprozesse konzipiert und demonstriert. Im Vordergrund steht dabei insbesondere ein hohes Maß an Wiederverwendbarkeit, um neue Bauteilvarianten, Prozesse oder Hardwarekomponenten effizient abbilden zu können.

Es zeigt sich, dass hierfür nicht nur Skills als Ganzes, sondern auch ihre Bestandteile und Parametrierung wiederverwendbar sein müssen. Entsprechend wird das Skill-Modell als Baukastensystem entworfen, das die Komposition und inkrementelle Erweiterung von Skills erlaubt. Anhand von zehn exemplarischen Montageanwendungen wird gezeigt, dass bereits eine kleine Anzahl an Skills die Basis für eine Vielzahl von industriellen Anwendungsfällen bilden kann und es sehr einfach ist, weitere Skills mittels vorhandener Bausteine zu erstellen.

ISBN 978-3-8396-1719-9

9 783839 617199

FRAUNHOFER VERLAG