ALWIN HOFFMANN

SERVICEORIENTIERTE AUTOMATISIERUNG VON ROBOTERZELLEN

SERVICEORIENTIERTE AUTOMATISIERUNG VON ROBOTERZELLEN

Modularität und Wiederverwendbarkeit
von Software in der Robotik

ALWIN HOFFMANN

Dissertation zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Angewandte Informatik
der Universität Augsburg

April 2015

Alwin Hoffmann: *Serviceorientierte Automatisierung von Roboterzellen,* Modularität und Wiederverwendbarkeit von Software in der Robotik, Dissertation, April 2015

GUTACHTER:
Prof. Dr. Wolfgang Reif
Prof. Dr. Bernhard Bauer

TAG DER MÜNDLICHEN PRÜFUNG:
27. April 2015

You were there in the winter, moonlight on the snow

— Bob Dylan, *Sara*

Für Stefanie und unsere Tochter Sophia.

ABSTRACT

Given an appropriate tool, industrial robots are able to perform flexible manufacturing tasks. However, robots are rarely used, if it comes to small batch production. One major reason is the high cost of programming that contribute the largest share of total cost of a robotic system. Furthermore, these costs recur for each adaptation to a high extent [91]. The reuse of software for programming a robot cell is the key requirement to reduce development costs. Accordingly, the aim must be that software for robot cells can be put together from modular and reusable components. A comprehensive approach is necessary for both the modelling and the implementation of modular software to create flexible robot cells.

The main result of this thesis is a novel and innovative approach to service-oriented automation of robot cells. In particular, a comprehensive modelling approach using SysML is introduced in order to structure the automation software and to identify relevant services. The services focus on the handling and manufacturing tasks. Subsequently, these services can be orchestrated to form the automation process. According to the ideas of Industry 4.0, the automation software uses an "intelligent product" which carries all the knowledge of its manufacturing process. The automation services rely on an object-oriented model that describes the products, robots and tools as well as their geometric and semantic relationships. Using this knowledge makes it possible to create generic programs that are applicable universally. Accordingly, the automation software can be assembled from reusable services. The approach was evaluated successfully using a cooperative assembly application.

ZUSAMMENFASSUNG

Industrieroboter sind mechanisch flexible Maschinen, die mit dem richtigen Endeffektor ausgestattet flexible Fertigungsaufgaben übernehmen können. Allerdings werden Roboter nur selten eingesetzt, wenn es sich um die Fertigung kleiner Stückzahlen handelt. Ein Hauptgrund dafür sind die hohen Kosten der Programmierung, die den größten Anteil der Gesamtkosten an einem Robotersystem darstellen und für jede Adaption in hohem Umfang wieder anfallen [91]. Die Wiederverwendung von Software bei der Programmierung von Roboterzellen ist eine zwingende Voraussetzung, um die Entwicklungskosten zu senken. Ziel muss es dementsprechend sein, dass sich Software für Roboterzellen modular aus wiederverwendbaren Komponenten zusammenstellen lässt. Eine hierfür zwingende Grundlage stellt die Entwicklung eines ganzheitlichen Ansatzes zur Modellierung und Implementierung modularer und flexibler Software für die Automatisierung von Roboterzellen dar.

Das zentrale Ergebnis der vorliegenden Dissertation ist ein umfassendes Konzept zur serviceorientierten Automatisierung von Roboterzellen. Dazu wurde ein Modellierungsansatz mithilfe von SysML entwickelt, um die Software zu strukturieren und die relevanten Services zu identifizieren. Dabei werden die Services an den möglichen Handhabungs- und Fertigungsaufgaben der Roboterzelle ausgerichtet, um anschließend den Automatisierungsprozess aus diesen Diensten zu orchestrieren. Gemäß den Ideen von Industrie 4.0 steht das intelligente Bauteil im Mittelpunkt der Automatisierungssoftware. Es trägt das Wissen, wie es gefertigt werden muss, mit sich und beeinflusst so den Produktionsprozess maßgeblich. Durch ein objektorientiertes Modell der Bauteile, Roboter und Werkzeuge sowie ihrer geometrischen und semantischen Zusammenhänge kann die Roboterzelle vollständig in Software abgebildet werden. Die direkte Verwendung dieses Wissens in der Programmierung ermöglicht es, generische Programmfragmente zu erstellen, die universell einsetzbar sind. Dementsprechend kann die Automatisierungssoftware über Services aus wiederverwendbaren Komponenten nach dem Baukastenprinzip zusammengesetzt werden. Der hier vorliegende, neue Ansatz wurde anhand einer kooperativen Montageanwendung evaluiert.

Es ist die traditionelle Aufgabe der naturwissenschaftlichen Disziplinen,
die natürlichen Dinge zu lehren:
wie sie sind und wie sie funktionieren.

Die Aufgabe der Ingenieurschulen ist es gewesen,
die künstlichen Dinge zu lehren:
wie man Artefakte mit erwünschten Eigenschaften herstellt.

— Herbert A. Simon, *Die Wissenschaft vom Künstlichen* [248, S. 95]

DANKSAGUNG

Diese Arbeit entstand im Rahmen meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Software & Systems Engineering der Universität Augsburg. Sie wäre ohne die großartige Unterstützung vieler Menschen nicht möglich gewesen. Daher möchte ich mich an dieser Stelle bei allen herzlich bedanken, die mich unterstützt haben.

Allen voran möchte ich meinem Doktorvater Prof. Dr. Wolfgang Reif für seine Unterstützung, das mir entgegenbrachte Vertrauen und seine Anregungen herzlich danken. Ebenso danke ich Prof. Dr. Bernhard Bauer für die Übernahme der Zweitbegutachtung und einer Fachprüfung. Prof. Dr. Jörg Hähner danke ich für die Übernahme der zweiten Fachprüfung.

Bei meinen Kollegen am Institut möchte ich mich für die tolle Arbeitsatmosphäre und die immer interessanten, oft inspirierenden, manchmal skurrilen Gespräche und Diskussion danken. Mein ganz besonderer Dank, gilt meinem *Büroehemann* Andreas Angerer, der mir mit Rat und Tat zur Seite stand und mich, wenn nötig, auch wieder aufgebaut hat. Bei Michael Vistein bedanke ich mich für seine Hilfe im Echtzeitteil. Ich möchte mich auch bei Andreas Schierl für seine Unterstützung und die anregenden Diskussionen bedanken. Seine Ideen waren immer wieder wertvolle Anregungen. Mein Dank gilt auch Ludwig Nägele, Miroslav Macho und Constantin Wanninger, die das Robotik-Team vervollständigen. Bei Hella Seebach bedanke ich mich für ihre aufmunternden Worte und das Korrekturlesen der Arbeit.

Zudem möchte ich mich bei allen Studenten und wissenschaftlichen Hilfskräften bedanken, besonders bei Wladislaw Stawrow, Sebastian Pieniack, Andreas Osipov, Matthias Frey, Matthias Stüben und Sascha Jocham. Herzlichen Dank an Veronika Bernhardt und Cornelius Madlener, die das Korrekturlesen übernommen haben.

Meinen Eltern möchte ich für ihre unbedingte Unterstützung und ihr Vertrauen in mich danken.

Für ihre liebevolle Unterstützung möchte ich meiner Frau Stefanie von ganzem Herzen danken. Ohne dich und deine Geduld hätte ich diese Arbeit nie vollenden können. Zuletzt danke ich meiner Tochter Sophia, dass sie ihrem Papa in den letzten Monaten mit einem Lachen immer wieder neue Kraft gegeben hat.

Augsburg im Februar 2015

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGS

AIS	Artificial Intelligence Subsystem
AML	A Manufacturing Language
API	Application Programming Interface
aRD	agile Robot Development
aRDx	agile Robot Development – next generation
ARLA	ASEA programming Robot LAnguage
BDD	Blockdefinitionsdiagramm
BPMN	Business Process Modeling Notation 2.0
BMBF	Bundesministerium für Bildung und Forschung
BMWi	Bundesministerium für Wirtschaft und Energie
CAN	Controller Area Network
CFK	carbonfaserverstärkter Kunststoff
CIRCA	Cooperative Intelligent Real-time Control Architecture
CLARAty	Coupled Layered Architecture for Robot Autonomy
CLEaR	Closed-loop execution and recovery
COLLADA	COLLAborative Design Activity
CPD	Cyber-Physical Device
CPP	Cyber-Physical Product
CPS	Cyber-Physical System
CPPS	Cyber-Physical Production System
DLR	Deutsches Zentrum für Luft- und Raumfahrt
DPWS	Devices Profile for Web Services
DSSP	Decentralized Software Services Protocol
E/A	Ein-/Ausgabe
EDDL	Electronic Device Description Language
EtherCAT	Ethernet for Control Automation Technology
FRI	Fast Research Interface
GMF	Graphical Modeling Framework
I/O	Input/Output
IBD	Internes Blockdiagramm

JAR	Java ARchive
Java ME	Java Micro Edition
Java SE	Java Standard Edition
JRE	Java Runtime Environment
JVM	Java Virtual Machine
KDL	Kinematics and Dynamics Library
KIF	Knowledge Integration Framework
KRC	KUKA Robot Control
KRL	KUKA Robot Language
LBR	Leichtbauroboter
LWA	Lightweight Arm
LWR	Light-Weight Robot
MCP	Motion Center Point
MHI	Mechanical Hand Interpreter
MiRPA	Middleware for Robotic and Process control Applications
MIT	Massachusetts Institute of Technology
NASA	National Aeronautics and Space Administration
OMG	Object Management Group
ORCCAD	Open Robot Controller Computer Aided Design
OROCOS	Open RObot COntrol Software
OSGi	Open Services Gateway initiative
PasRo	Pascal for Robots
PLC	Programmable Logic Controller
PROFIBUS	PROcess FIeld BUS
PROFINET	PROcess FIeld NETwork
PTP	Point-To-Point
PUMA	Programmable Universal Machine for Assembly
RAP	Reactive Action Packages
RCC	Robot Control Core
RCCL	Robot Control C Library
RCP	Rich Client Platform
RIPE	Robot Independent Programming Environment
RMI	Remote Method Invocation

ROS	Robot Operating System
RPI	Realtime Primitive Interface
RRT	Rapidly-exploring Random Tree
RSI	Remote Sensor Interface
RTM	Robotics Technology Middleware
RTS	Real-Time Subsystem
RTT	Real-Time Toolkit
SCA	Service Component Architecture
SCR	Service Component Runtime
SCXML	State Chart XML
SIRENA	Service Infrastructure for Real time Embedded Networked Applications
SOA	serviceorientierte Architektur
SOCRADES	Service-Oriented Cross-layer infRAstructure for Distributed smart Embedded devices
SPS	speicherprogrammierbare Steuerung
SysML	Systems Modeling Language
TCP	Tool Center Point
UDP	User Datagram Protocol
UML	Unified Modeling Language
UPnP	Universal Plug and Play
URDF	Unified Robot Description Format
VAL	Variable Assembly Language
VDI	Verein Deutscher Ingenieure
VPS	verbindungsprogrammierbare Steuerung
W3C	World Wide Web Consortium
WCF	Windows Communication Foundation
XIRP	XML-based Interface for Robots and Peripherals
XML	eXtensible Markup Language
YARP	Yet Another Robot Platform

Teil I

EINFÜHRUNG

Der erste Teil zeigt zum einen die Motivation der vorliegenden Arbeit auf und legt zum anderen die Ziele dar, die mit einer serviceorientierten Automatisierung auf Roboterzellen verbunden sind. Darüber hinaus wird ein Überblick über die wichtigsten Forschungsergebnisse, die im Rahmen dieser Dissertation entstanden sind, gegeben und der weitere Aufbau und Inhalt der Arbeit vorgestellt.

Da diese Dissertation eine interdisziplinäre Fragestellung behandelt, werden sowohl die relevanten Grundlagen der Softwaretechnik als auch der Robotik und Automatisierung erläutert. Daher werden die notwendigen Grundlagen beider Disziplinen detailliert vorgestellt. Hier sei dem Leser empfohlen, die Kapitel abhängig vom eigenen Hintergrund und nach Bedarf zu wählen.

1 EINLEITUNG

Das Ziel der vorliegenden Arbeit ist die Entwicklung eines Konzepts zur serviceorientierten Automatisierung von Roboterzellen. Dabei setzt sich eine industrielle Roboterzelle aus einer Menge hierarchisch gegliederter Dienste zusammen, die orchestriert werden können, um die Fertigungsprozesse flexibel abzubilden.

Im ersten Kapitel werden zunächst Motivation und Ziele dieses Ansatzes vorgestellt und anschließend die Forschungsergebnisse der Dissertation in Abschnitt 1.2 dargelegt. Das Kapitel schließt mit Abschnitt 1.3, in dem Aufbau und Inhalt der gesamten vorliegenden Arbeit umrissen werden.

1.1 MOTIVATION UND ZIELE

Industrieroboter sind mechanisch flexible Maschinen, die zudem frei programmierbar sind. Ein Industrieroboter kann mit beliebigen Endeffektoren, d. h. Werkzeugen oder Sensoren an seinem Flansch, ausgerüstet werden. Diese kann er flexibel in seinem Arbeitsraum bewegen und mit jeder Orientierung ausrichten. Ein Industrieroboter kann folglich für ein breites Spektrum an Anwendungen eingesetzt werden. Durch Sensoren kann er zudem seine Umgebung wahrnehmen und sich dort zurechtfinden. Insbesondere nachgiebige Roboter können durch integrierte Sensorik mittlerweile Aufgaben durchführen, die vor Jahren nicht zu automatisieren waren. Durch eine geeignete Auswahl (kooperativer) Roboter, entsprechender Endeffektoren und notwendiger Sensoren können mechanisch flexible Roboterzellen aufgebaut werden.

Trotzdem zeigt ein Blick in die Fabriken, das Roboter oft nur bei der Fertigung großer Stückzahlen wie z. B. in der Automobilproduktion eingesetzt werden. Es existiert damit eine Diskrepanz zwischen Theorie und praktischem Einsatz. Eine Erklärung dafür können die aktuell noch sehr hohen Kosten von Integration und Softwareentwicklung sein. In einer Studie [91] der Europäischen Kommission wurde die Wertschöpfungskette von Robotersystemen betrachtet. Dabei wurde festgestellt, dass nur 25 % der gesamten Kosten auf den Roboter entfallen. Dagegen sind 75 % der Kosten für die Integration nötig, wovon die Programmierungskosten mit 40 % den größten Anteil ausmachen.

Folglich stellt die Programmierung oder im Allgemeinen die Softwareentwicklung für Robotersysteme bzw. für Roboterzellen das größte Hemmnis für den breiten Einsatz dar. Die Studie zeigt zudem, dass bei einer Variation der Aufgabe ca. 75 % der ursprünglichen Entwicklungskosten erneut anfallen. Bei diesen Programmen handelt es sich also um Unikate, die nicht wiederverwendet werden können. Aber gerade die Wiederverwendung von Software oder zumindest von Programmfragmenten ist eine zwingende Voraussetzung, um die Entwicklungskosten zu senken. Ziel muss es dementsprechend sein, dass sich Software für Roboterzellen modular aus wiederverwendbaren Komponenten zusammensetzen lässt.

Gleichzeitig läutet nach Kagermann et al. [144] das Internet der Dinge und Dienste die 4. industrielle Revolution ein. In Zukunft besteht eine Fabrik demnach aus einer Menge von vernetzen und intelligenten Fertigungsma-

schinen, sogenannten cyber-physischen Produktionssystemen. Aber nicht nur die Maschinen sind intelligent, darüber hinaus auch die Produkte. Sie tragen das Wissen, wie sie gefertigt werden müssen, in sich und agieren eigenständig in der Fabrik. Sie handeln mit den vorhandenen Produktionsressourcen und den notwendigen Betriebsstoffen den Fertigungsprozess aus und steuern sich gegenseitig. Das bedeutet, dass es keinen festen Ablauf in der Fertigung gibt.

Jammes und Smit [141] haben zum ersten Mal die Vorteile serviceorientierter Architekturen (SOA) für die Automatisierung hervorgehoben. Dabei bietet jedes Gerät seine Funktionalität über Dienste an. In Analogie zu Unternehmensanwendungen werden die Dienste und ihre Granularität an den Geschäftsprozessen ausgerichtet. In einer Fabrik oder einer Roboterzelle entspricht dies den Fertigungsprozessen. Dadurch können die Dienste mehrerer Geräte orchestriert, d. h. in einer geeigneten Reihenfolge verwendet werden, um eine Automatisierungsaufgabe abzubilden. Dies ermöglicht eine flexible Abbildung der Automatisierungsprozesse auf Basis vorhandener Dienste.

In dieser Dissertation wird aufgezeigt, wie man die Vorteile serviceorientierter Architekturen auf Roboterzellen übertragen kann. Die Dienste werden an den möglichen Handhabungs- und Fertigungsaufgaben der Roboterzelle ausgerichtet, um anschließend den Automatisierungsprozess aus diesen Diensten zu orchestrieren. Der Ansatz dieser Dissertation verfolgt das oben formulierte Ziel, indem die Software von Roboterzellen modular aus vorhandenen Bausteinen und Services zusammengesetzt wird. Durch die Modularität der Software und die Wiederverwendung vorhandener Bausteine kann eine signifikante Reduktion der Entwicklungskosten – sowohl bei der erstmaligen Inbetriebnahme als auch bei der späteren Adaption – realisiert werden.

1.2 FORSCHUNGSERGEBNISSE

Vorliegende Forschungsarbeit beschäftigt sich mit der Frage, ob und wie sich Roboterzellen mit serviceorientierter Softwareentwicklung modellieren und automatisieren lassen. Dabei liegt der Fokus auf Flexibilität und Wiederverwendbarkeit. Daher lautet die zugrunde liegende Forschungsfrage:

> „Wie können Roboterzellen durch serviceorientierte Architekturen flexibel und modular, d. h. als Komposition verschiedener Bausteine, automatisiert werden?“

Diese Frage wird im Rahmen der Dissertation aus unterschiedlichen Blickwinkeln und auf unterschiedlichen Abstraktionsebenen betrachtet.

Im Laufe der Arbeit an dieser Dissertation sind verschiedene Forschungsergebnisse entstanden, die sich mit der oben gestellten Frage, aber auch allgemein mit dem Einsatz moderner Softwaretechnik in der Robotik und Automatisierung, beschäftigen. Zentrale und wichtige Ergebnisse sind die *Harmonisierung von Applikationsprogrammierung und echtzeitkritischer Robotersteuerung* [123] und die daraus entwickelte *echtzeitfähige Softwarearchitektur* [118] für Industrieroboter. Dabei werden Roboterprogramme in atomare Aktionen zerlegt und unter einem Echtzeitbetriebssystem ausgeführt. Die Applikation und die Ablaufsteuerung kann in einer modernen objektorientierten Programmiersprache entwickelt und auf einem Standardbetriebssystem ausgeführt werden. Dieses Ergebnis ist im Forschungsprojekt *SoftRobot* ge-

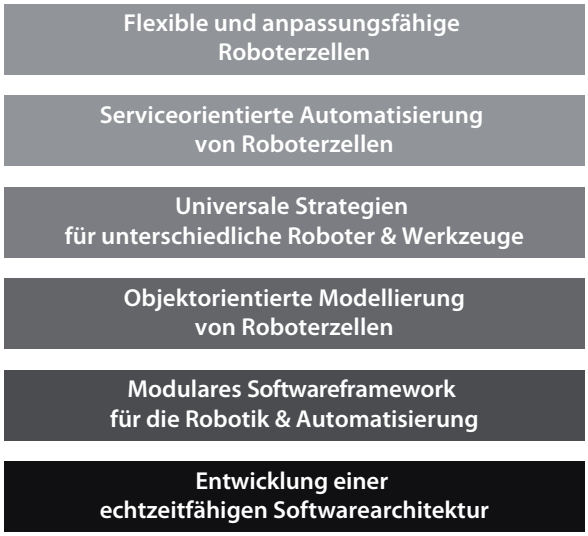


Abbildung 1.1: Übersicht der einzelnen Forschungsergebnisse und wie diese aufeinander aufbauen.

meinschaftlich entstanden und auch Teil der Dissertationen von Andreas Angerer [10] und Michael Vistein [271].

Darüber hinaus sind weitere, originäre Forschungsergebnisse entstanden. Die Übersicht in Abbildung 1.1 illustriert, wie diese aufeinander aufbauen:

- Die entwickelte Softwarearchitektur wurde in ein erweiterbares und *modulares Framework* [119] zur Entwicklung von Roboter- und Automatisierungsapplikationen transformiert. Durch die Einbettung in die *OSGi Service Platform* ergibt sich ein dynamisches und (zur Laufzeit) erweiterbares Framework. Die Erweiterbarkeit ermöglicht es, dass neue Aktuatoren und Sensoren einfach implementiert und integriert werden können. Zusätzlich sind auch orthogonale Erweiterungen, wie z. B. eine dreidimensionale Visualisierung, eine Kollisionserkennung oder eine kollisionsfreie Bahnplanung, möglich.

- Das Framework wurde – im Vergleich zu Angerer [10] – ergänzt, um ein *objektorientiertes Modell von Roboterzellen* erstellen zu können. Dazu kann eine Roboterzelle vollständig als eine Menge von Objekten in Software abgebildet werden, d. h. mit allen Bestandteilen, deren Gestalt und den geometrischen und semantischen Beziehungen untereinander [19, 120]. Die Besonderheit dieses Ansatzes ist es, dass alles Wissen über die Roboterzelle in der Programmierschnittstelle integriert ist. Das ermöglicht es, dieses Wissen direkt für die Planung und Parametrisierung von Aufgaben zu verwenden. Durch die Ausführung der Aufgaben wird das Modell aktualisiert und ist wiederum Grundlage für die weiteren Aufgaben.

- Die zuvor beschriebene objektorientierte Modellierung erlaubt es, die Gestalt und die Merkmale von Robotern, Werkzeugen sowie die in einer Roboterzelle vorhandenen Gegenstände und Handhabungsobjekte als Software abzubilden, was eine direkte Verwendung in der Programmierung möglich macht. Somit können generische Programmfragmente oder Strategien erstellt werden, die – richtig parametrisiert – universell einsetzbar sind [120]. Durch Polymorphie können diese *Strategien für verschiedene Roboter und Werkzeuge* verwendet werden.

- Genannte Strategien sind elementare Bausteine einer serviceorientierten Roboterzelle und ermöglichen eine Trennung von Aufgaben- und Bewegungsprogrammierung [121]. Es wird in dieser Dissertation ein neues Konzept zur *serviceorientierten Automatisierung von Roboterzellen* vorgestellt, mithilfe dessen die Automatisierungssoftware nach dem Baukastenprinzip zusammengestellt werden kann [121]. Für die Modellierung der unterschiedlichen Abstraktionsebenen einer serviceorientierten Roboterzelle wird die Systems Modeling Language (SysML) eingesetzt [254].

Durch das Baukastenprinzip und die roboterunabhängigen Strategien weisen serviceorientierte Roboterzellen eine hohe *Flexibilität und Anpassungsfähigkeit* auf [120]. Die in der Roboterzelle implementierten Fertigungsprozesse können einfach geändert werden. Dies betrifft sowohl den generellen Ablauf der Fertigungsprozesse als auch die dort bearbeiteten Werkstücke. Die Strategien, welche die Programmierung der Roboter und der Werkzeuge beinhalten, können ebenfalls adaptiert werden. Auch die Aktuatoren können ohne grundlegende Änderungen der Software ausgetauscht werden. Während einige dieser Änderungen nur zur Entwicklungszeit möglich sind, können andere sogar während der Laufzeit erfolgen.

1.3 STRUKTUR DER ARBEIT

Die vorliegende Arbeit ist in drei große Teile untergliedert. Teil i enthält neben der Einleitung jeweils ein Kapitel über die Grundlagen der notwendigen softwaretechnischen Konzepte und die heutige Programmierung von Roboterzellen. Kapitel 2 enthält daher eine kurze Einführung in die komponentenbasierte Softwareentwicklung. Anschließend werden die Grundlagen serviceorientierter Architekturen (SOA) erläutert. Das Kapitel schließt mit einer Einführung in die *OSGi Service Platform*, einem dynamischen Modulsystem für Java.

Nach den softwaretechnischen Konzepten wird der aktuelle Stand der Programmierung von Industrierobotern bzw. von Roboterzellen in Kapitel 3 dargestellt. Neben dem Aufbau von Robotersteuerungen und der Roboterprogrammierung ist auch eine kurze Einführung in speicherprogrammierbare Steuerungen Gegenstand dieses Kapitels.

Der zweite Teil beschäftigt sich mit der Entwicklung eines modularen Frameworks zur ganzheitlichen Programmierung von Roboterzellen. In Kapitel 4 wird zu Beginn des zweiten Teils das Forschungsprojekt *SoftRobot*, dessen Ziele und Anforderungen sowie die dort entwickelte Softwarearchitektur, vorgestellt. Die Softwarearchitektur besteht aus dem *Robotics Application Framework* sowie einer echtzeitfähigen Robotersteuerung und wurde gemeinschaftlich im Forschungsprojekt *SoftRobot* entwickelt (vgl. Angerer [10] bzw. Vistein [271]). Sie ist die Grundlage der in dieser Dissertation vorgestellten neuen Konzepte und Ergebnisse.

Kapitel 5 beschreibt die objektorientierte Modellierung einer Roboterzelle mit allen darin enthaltenen Elementen (d. h. Robotern, Werkzeugen und Bauteilen). Diese Modellierung erweitert die *Robotics API*, einer in *SoftRobot* entstandenen, objektorientierten Programmierschnittstelle für Industrieroboter [10]. Basierend darauf wird in Kapitel 6 die Modularisierung des *Robotics Application Frameworks* mithilfe der *OSGi Service Platform* vorgestellt. Die Modularisierung erlaubt, dass das *Robotics Application Framework* erweiterbar ist und alle Aspekte der Programmierung einer Roboterzelle abdecken

kann. Folglich kann die Automatisierungssoftware einer Roboterzelle aus einer Menge einzelner Module applikationsspezifisch zusammengestellt werden. Ausgesuchte Beispiele für Erweiterungen werden ebenfalls in Kapitel 6 vorgestellt.

Der dritte und letzte Teil dieser Arbeit beschäftigt sich mit der serviceorientierten Modellierung und Implementierung von Roboterzellen. Basierend auf dem modularen *Robotics Application Framework* können die Softwarekomponenten einer Roboterzelle individuell zusammengestellt werden. Zudem lässt sich durch die objektorientierte *Robotics API* die topologische und geometrische Struktur einer Roboterzelle vollständig in Software abbilden. Über dieses objektorientierte Modell einer Roboterzelle lässt sich eine serviceorientierte Struktur legen, die eine flexible und wiederverwendbare Automatisierung ermöglicht. Das generelle Konzept serviceorientierter Roboterzellen wird in Kapitel 7 vorgestellt. Die in dieser Arbeit untersuchte Fallstudie einer kooperativen Montageanwendung – die Factory 2020 – wird in Kapitel 8 beschrieben.

Bevor eine serviceorientierte Roboterzelle implementiert werden kann, muss die Zelle zuerst umfassend modelliert werden. Dazu wird in Kapitel 9 ein generelles Vorgehen erläutert und anhand der Fallstudie motiviert. Dabei ist die Modellierung, für die SysML verwendet wird, weitgehend unabhängig von den vorhandenen Robotern und Werkzeugen. Anschließend wird in Kapitel 10 vorgestellt, wie Fähigkeiten von Robotern bzw. ihrer Werkzeuge in einer serviceorientierten Architektur flexibel zur Verfügung gestellt werden können. In Kapitel 11 wird der Ansatz anhand der in Kapitel 7 eingeführten Fallstudie auf Änderungsflexibilität evaluiert. Die Arbeit endet mit einer Zusammenfassung und einem Fazit in Kapitel 12.

Ergänzende Informationen erhält der Leser im Anhang. So enthält Anhang A eine Übersicht aller Anwendungen und Technologiepakete der KUKA Roboter GmbH und der MRK Systeme GmbH, die im Rahmen von *SoftRobot* untersucht wurden. Die für die Fallstudie und die Evaluierung verwendeten Roboter und Werkzeuge sind in Anhang B aufgeführt. Eine Übersicht aller im Rahmen dieser Dissertation betreuten Abschlussarbeiten wird in chronologischer Reihenfolge in Anhang C gegeben. Ebenfalls in chronologischer Reihenfolge sind alle eigenen Publikationen in Anhang D angeführt, die im Rahmen von *SoftRobot* und dieser Dissertation entstanden sind. Abschließend folgt eine Liste aller referenzierten Quellen und Literaturangaben.

2 ÜBER SOFTWAREKOMPONENTEN UND SERVICES

Diese Arbeit beschäftigt sich mit der Modularisierung von Roboterprogrammen bzw. von robotergestützten Automatisierungssystemen. Das Ziel der Modularisierung ist die Entwicklung von Automatisierungssoftware, die flexibel, wiederverwendbar und erweiterbar ist. Ein Programm darf nicht mehr nur als einzigartiges Unikat angesehen werden, das nur für eine bestimmte Aufgabe erstellt wurde. Vielmehr muss es darauf ausgerichtet werden, dass es (oder zumindest Teile davon) wiederverwendet werden können. Daher werden in diesem Kapitel einige softwaretechnische Methoden, Architekturen und Frameworks vorgestellt, die helfen, dieses Ziel zu erreichen. Das Kapitel wendet sich infolgedessen vor allem an Leser mit einem Hintergrund in der Robotik bzw. der Automatisierung.

Die neu entwickelte Softwarearchitektur (vgl. Kap. 4) ist durch die Verwendung einer objektorientierten Programmiersprache (d. h. Java) inhärent objektorientiert. Allerdings ist dies noch kein hinreichendes Kriterium, um modulare Softwaresysteme damit zu erstellen. Auch muss die Softwarearchitektur selbst in so heterogenen Domänen wie der Robotik bzw. der Automatisierung modular und erweiterbar sein. Daher wird in Abschnitt 2.1 die komponentenbasierte Softwareentwicklung vorgestellt, die einen essentiellen Beitrag leistet, um Softwaresysteme modular und erweiterbar zu gestalten. Sie kann als Ergänzung der objektorientierten Programmierung angesehen werden [45] und fügt sich daher nahtlos in eine objektorientierte Softwarearchitektur ein (vgl. Kap. 6).

Anschließend werden in Abschnitt 2.2 die Grundlagen und Konzepte serviceorientierter Architekturen (SOA) erklärt. Sie haben ihren Ursprung in Unternehmensanwendungen und sind an den Geschäftsprozessen eines Unternehmens ausgerichtet. Das bedeutet, dass die Struktur der Software sich auf eine flexible Abbildung dieser Prozesse ausrichtet. Zum Abschluss wird in Abschnitt 2.3 die *OSGi Service Platform* beschrieben, die sowohl eine komponentenbasierte und serviceorientierte Entwicklung von Anwendungen in Java ermöglicht. OSGi wird in dieser Arbeit sowohl für die Realisierung des modularen Frameworks (vgl. Kap. 6) als auch für die Implementierung serviceorientierter Roboterzellen (vgl. Kap. 9) verwendet. Es vereint als Plattform viele Aspekte, die den Anforderungen moderner Automatisierungssystemen im Kontext von *Industrie 4.0* und dem *Internet of Things* mehr als Genüge tun [229].

2.1 KOMPONENTENBASIERTE SOFTWAREENTWICKLUNG

Um die Flexibilität und die Verständlichkeit von Softwaresystemen zu erhöhen, ist eine wichtige Voraussetzung, diese Systeme zu modularisieren und damit in einzelne Komponenten zu zerlegen [212]. Gleichzeitig kann dadurch die Entwicklungszeit und damit die Kosten der Softwareentwicklung abnehmen. Das Softwaresystem wird nicht als monolithische Einheit erstellt, sondern in einzelne Komponenten aufgeteilt. Diese sollten idealerweise wiederverwendbar entworfen sein und – wie in Abbildung 2.1 dargestellt – immer wieder neu zu einem Softwaresystem zusammengestellt werden [256].

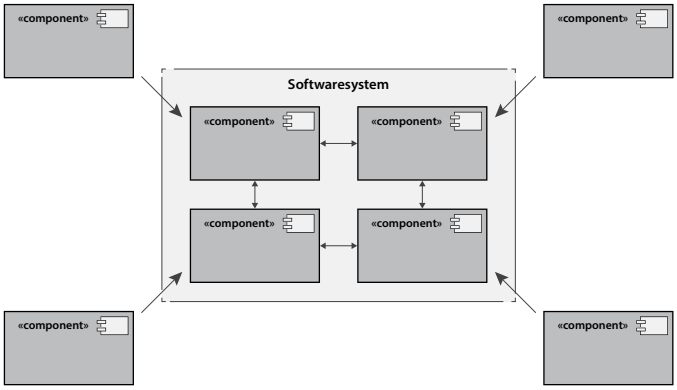


Abbildung 2.1: In der komponentenbasierte Softwareentwicklung werden Softwaresysteme aus einzelnen Komponenten zusammengestellt. Dabei müssen die Schnittstellen der Komponenten passen.

Die Gründe für den Einsatz von Softwarekomponenten können vielfältig sein. Sie ermöglichen eine strategische *make or buy* Entscheidung, was zu geringen Kosten führen kann [255]. Das heißt ein Softwaresystem kann in extern entwickelte und eingekaufte bzw. in interne entwickelte Komponenten unterteilt werden. Zudem ermöglicht eine komponentenbasierte Softwareentwicklung, ein Softwaresystem durch eine geeignete Zusammenstellung der Komponenten besser auf die spezifischen Bedürfnisse eines Kunden anzupassen. Letztendlich kann dies zur Entwicklung von *Produktlinien* führen, bei denen die einzelnen Produkte durch die Struktur und Konfiguration der Komponenten entsteht [255]. Durch eine Rekonfiguration und ggf. durch Hinzufügen neuer Komponenten lässt sich ein solches System dynamisch auf geänderte Anforderungen und Randbedingungen anpassen [170].

Eigenschaften von Komponenten

Gemäß Szyperski et al. [256] sind die drei charakteristischen Merkmale einer Komponente, dass sie

- als Einheit unabhängig verteilt werden kann,
- zu größeren Systemen zusammengestellt werden kann und
- keinen von außen beobachtbaren Zustand hat.

Damit eine Komponente unabhängig verteilt werden kann, muss sie von ihrer Umgebung und anderen Komponenten abgetrennt sein. Das bedeutet, dass sie ihren Inhalt kapselt und den internen Aufbau verbirgt. Sie kann nur als ganze Einheit verteilt werden. Während die Implementierung verborgen ist, muss eine Komponente jedoch spezifizieren, was sie zum einen für Voraussetzungen hat und zum anderen was sie für Funktionalität anbietet. Die Interaktion mit anderen Komponenten muss durch entsprechende Schnittstellen wohlbestimmt sein. Dies ist essentiell, damit Komponenten als eigenständige Einheiten zu größeren Softwaresystemen zusammengestellt werden können.

Dementsprechend leitet sich die Abgrenzung zur objektorientierten Softwareentwicklung her, die von Szyperski et al. [256] wie folgt skizziert wird:

> "A software component is what is actually deployed – as an isolated part of a system – in a component-based approach. [...] objects are almost never sold, bought, or deployed. The unit deployment is something rather more static, such as a class, or, more likely, a set or framework of classes, compiled and linked into some package." [256, S. 10]

Folgt man dieser Abgrenzung, ist eine Komponente an sich statisch und kann eher mit einer Sammlung von Klassen verglichen werden. Dagegen haben Objekte einen dynamischen Aspekt, da sie instanziiert werden und jedes Objekt eine eigene Identität hat.

Eigenschaften von Objekten

Im Gegensatz zu einer Komponente definiert Szyperski et al. [256] als charakteristischen Merkmale eines Objekts, dass es eine instanziierbare Einheit mit einer Identität ist, es einen Zustand hat, der von außen beobachtet werden kann und es seinen Zustand und sein Verhalten kapselt. Da jedes Objekt einen eigenen Zustand hat, hat es auch eine eigene Identität und kann von anderen Objekten, auch von solchen mit gleichem Zustand, unterschieden werden. Um instanziierbar zu sein, benötigt ein Objekt in Form einer Klasse einen Bauplan, der die Attribute und Methoden eines Objekts spezifiziert.

Die komponentenbasierte Softwareentwicklung kann als natürliche Ergänzung der objektorientierten Programmierung angesehen werden [45], um erweiterbare und flexible Softwaresysteme zu entwerfen. Beides kann nicht isoliert betrachtet werden, sondern hängt eng miteinander zusammen. Szyperski [255] schreibt, dass eine Komponente in der Regel erst durch Objekte *zu Leben erwacht* und daher eine Menge von Klassen beinhaltet, die den Bauplan von Objekten repräsentieren. Vereinfacht kann eine Komponente als Container für Klassen betrachtet werden.

Obwohl eine Komponente ihre Implementierung verbergen soll, gibt es dafür unterschiedliche Abstufungen [256]. Während bei einer *Blackbox*-Komponente nur die Schnittstellen der Komponente verfügbar sind, offenbart eine *Whitebox*-Komponente auch Teile ihrer Implementierung. Dies ist bspw. notwendig, um durch Vererbung die von einer Komponente bereitgestellten Klassen zu erweitern und Funktionalität wiederzuverwenden. Bei *Greybox*-Komponenten wird ein Teil der Implementierung kontrolliert offengelegt. Während sich Wiederverwendung bei einer *Blackbox*-Komponente auf die Schnittstellen der Komponente reduziert, stützt sich Wiederverwendung bei einer *Whitebox*-Komponente auf die Implementierung ab. Dies kann auf der einen Seite das Maß an Wiederverwendung erhöhen, erschwert auf der anderen Seite die Austauschbarkeit einer Komponente [256].

Definition einer Komponente

Zusammenfassend kann die Definition einer Softwarekomponente nach Bosch et al. [44] wie folgt formuliert werden:

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." [44]

Die Schnittstellen einer Komponente stellen die Zugriffspunkte auf die Komponente dar. Dadurch können Nutzer der Komponente, die in der Regel wiederum Komponenten sind, auf die Funktionalität der Komponente zugreifen. Oft hat eine Komponente mehrere Schnittstellen, die unterschiedliche Zugriffspunkte darstellen. Da Komponenten unabhängig voneinander implementiert werden, haben die Schnittstellen nicht nur einen technischen Aspekt, sondern stellen auch einen Kontrakt über die Form der Interaktion dar [255]. Neben den bereitgestellten Schnittstellen, muss eine Komponente explizit ihre Abhängigkeiten zu anderen Komponenten und weitere Voraussetzungen spezifizieren. Die Abhängigkeiten zu anderen Komponenten sind in der Regel Schnittstellen, die eine Komponente zur Funktion benötigt. Nur wenn diese Abhängigkeiten erfüllt sind, kann eine Komponente fehlerfrei funktionieren. Die weiteren Voraussetzungen beziehen sich bspw. auf eine Laufzeitumgebung oder ein Framework, das benötigt wird.

2.2 SERVICEORIENTIERTE ARCHITEKTUREN

Eine serviceorientierte Architektur (SOA) ist ein Architekturmuster, um Softwaresysteme mithilfe von Diensten zu strukturieren. In der Regel wird eine SOA an den Geschäftsprozessen eines Unternehmens ausgerichtet, da deren Abstraktionsebenen die Grundlage für die Wahl der Dienste sind. Durch eine Orchestrierung bzw. ein Zusammensetzen von Services niedriger Abstraktionsebenen können Services höherer Abstraktionsebenen geschaffen werden. Maßgeblich sind dabei nicht technische Aufgaben, sondern die zugrundeliegenden Geschäftsprozesse [150]. Dadurch soll eine unternehmensweite Integration von Geschäftsprozessen entlang der Wertschöpfungskette ermöglicht werden. Es soll eine möglichst hohe Flexibilität der abgebildeten Geschäftsprozesse durch eine hohe Wiederverwendung bestehender Services erreicht werden. Das langfristige Ziel ist hierbei eine Reduktion von Kosten in der Softwareentwicklung, da ab einem bestimmten Zeitpunkt alle notwendigen Services vorhanden sind und diese nur noch geeignet orchestriert werden müssen.

Eine SOA beschreibt die Strukturierung eines Softwaresystems mit Diensten und ist damit eine konkrete Ausprägung einer Softwarearchitektur. Obwohl es keine einheitliche Definition dafür gibt (vgl. [41, 54, 93]), besteht Konsens darin, dass eine Softwarearchitektur ein Softwaresystem in einzelne Teile zerlegt und deren Struktur bzw. Organisation beschreibt. Bei Krafzig et al. [150] findet sich dazu folgende Definition:

> "A software architecture is a set of statements that describe software components and assigns the functionality of the system to these components. It describes the technical structure, constraints, and characteristics of the components and the interfaces between them. The architecture is the blueprint for the system and [the] high-level plan for its construction." [150, S. 56]

Folglich beschreibt eine Softwarearchitektur die Zerlegung eines Softwaresystems in einzelne Bestandteile bzw. Komponenten und teilt jedem Bestandteile seine Funktionalität oder Aufgabe zu. Daraus ergibt sich eine technische Struktur, welche die Schnittstellen zwischen den Bestandteilen festlegt. Eine Softwarearchitektur kann als Bauplan des Softwaresystems verstanden werden, der die richtige Umsetzung vorgibt.

Definition einer SOA

Eine SOA ist nach Krafzig et al. [150] folgendermaßen definiert:

> "A service-oriented architecture is a software architecture that is based on the key concepts of an application frontend, service, service repository, and service bus. A service consists of a contract, one or more interfaces, and an implementation." [150, S. 57]

Nach dieser Definition besteht eine serviceorientierte Architektur mit *Application Frontends*, *Services*, einem *Service Repository* und einem *Service Bus* aus vier wesentlichen Bestandteilen, die in in Abbildung 2.2 dargestellt sind. Trotz dieser technischen Bestandteile sollte eine SOA immer an der Infrastruktur der Geschäftsprozesse ausgerichtet sein [150]. Darüber hinaus kann es jedoch Dienste mit rein technischen Aufgaben geben, die Querschnittfunktionalität bereitstellen.

Ein *Service* besteht wiederum aus einem Kontrakt, einer Implementierung und seinen Schnittstellen (vgl. Abb. 2.2). Ein Kontrakt ist eine (informelle)

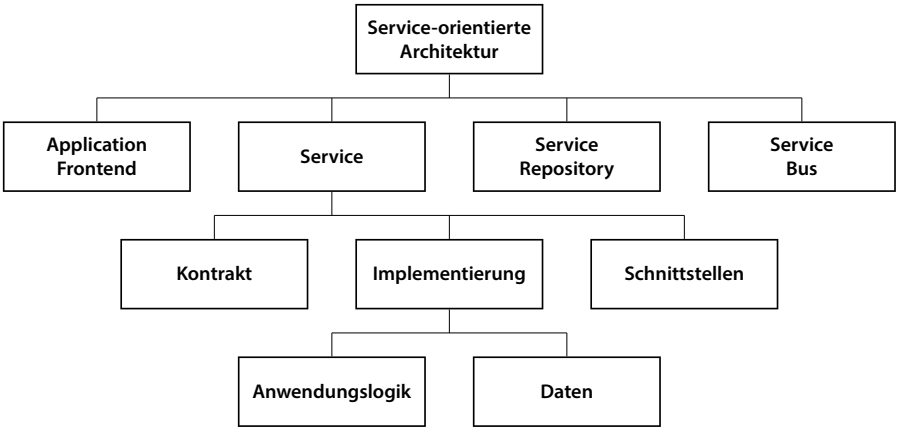


Abbildung 2.2: Services und Application Frontends sind die Hauptbestandteile einer serviceorientierten Architektur. Daneben gibt es noch ein Service Repository zum Auffinden von Diensten und ein Service Bus zur Kommunikation (vgl. Krafzig et al. [150]).

Spezifikation eines Dienstes und beschreibt Zweck, Funktionalität, Randbedingungen des Dienstes [150]. Daneben wird die Funktionalität eines Dienstes über seine Schnittstellen bereitgestellt und kann so von Dritten genutzt werden. Während der Kontrakt eine Beschreibung der Funktionalität eines Dienstes ist, ermöglichen die Schnittstellen technisch den Zugriff auf diese Funktionalität. Verborgen hinter dem Kontrakt und den Schnittstellen ist die eigentliche Implementierung eines Dienstes. Sie stellt die Anwendungslogik bzw. den Zugriff auf Daten bereit und muss der Spezifikation des Kontraktes entsprechen.

Getrieben wird eine SOA von den *Application Frontends*, die der aktive Teil der Architektur sind [150] und die Geschäftsprozesse anstoßen. Die Ergebnisse eines Geschäftsprozesses werden ebenfalls an das *Application Frontend* zurückgegeben. Dabei kann es verschiedene *Application Frontends* geben. Zum einen sind dies grafische Benutzeroberflächen (z. B. Webanwendungen), die eine Interaktion mit Kunden oder Mitarbeitern ermöglichen. Zum anderen müssen *Application Frontends* nicht immer mit einem Benutzer interagieren, sondern können auch andere Systeme oder langlaufende periodische Prozesse sein.

Ein *Service Repository* wird benötigt, um einen Dienst aufzufinden und alle Informationen zu erhalten, um den Dienst zu nutzen. Neben dem Kontrakt und den Schnittstellen können auch weitere Informationen wie bspw. eine Nutzungsgebühr, technische Randbedingungen oder Sicherheitsfragen in dem *Service Repository* enthalten sein. Nachdem ein Service mit seinem Kontrakt in einem *Service Repository* registriert ist, kann er von einem Client gefunden und verwendet werden. Abhängig von der Abstraktionsebene kann ein Client hierbei entweder ein anderer Dienst oder ein *Application Frontend* sein. Die Nutzung eines Dienstes kann während der Entwicklung vom Programmierer definiert werden. Dadurch ist der Client sehr stark an den jeweiligen Service gebunden. Alternativ kann dies auch dynamisch zur Laufzeit erfolgen [150].

Ein *Service Bus* hat zur Aufgabe, die unterschiedlichen Teilnehmer einer serviceorientierten Architektur (d. h. Dienste und *Application Frontends*) miteinander zu verbinden. Er stellt für die Teilnehmer eine Kommunikationsplattform bereit und übernimmt damit die Aufgabe einer Middleware. Angelehnt ist das Konzept an die Datenübertragung, wo ein Nachrichtenbus

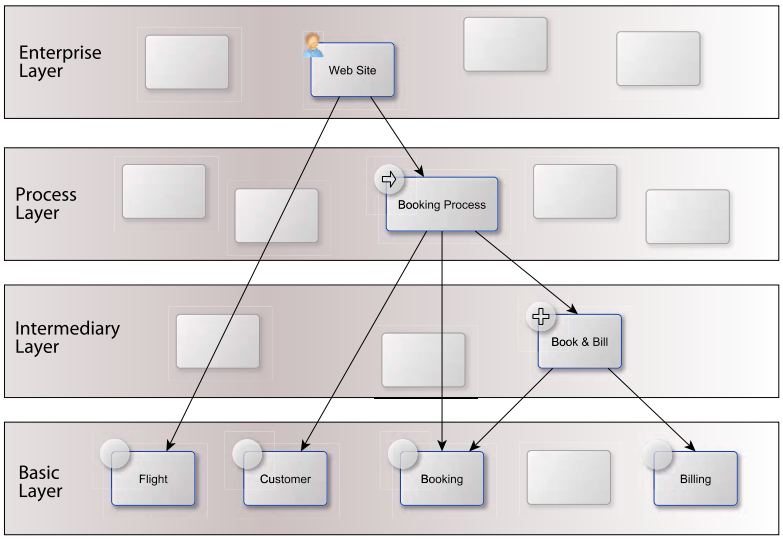


Abbildung 2.3: Eine voll ausgebildete serviceorientierte Architektur besteht aus verschiedenen Ebenen mit Services unterschiedlicher Aufgabe und Granularität (vgl. Krafzig et al. [150]).

ein einheitliches Kommunikationsmedium zwischen mehreren Teilnehmern ist. Folglich ersetzt der *Service Bus* ein Netz aus vielen direkten Kommunikationskanälen zwischen Services durch eine gemeinsame Kommunikationsinfrastruktur. Dienste verbinden ihre Schnittstellen über Endpunkte mit dem Bus und Clients kommunizieren mit einem Service, indem sie über den Bus Nachrichten austauschen.

Klassifikation von Services

Krafzig et al. [150] bezeichnen *Services* als flexible Bausteine, aus denen eine Anwendung bzw. ein Geschäftsprozess zusammengesetzt wird. Dazu stellen sie eine Klassifikation mit verschiedenen Kategorien für Services auf:

- *Application Frontends* initiieren Geschäftsprozesse und überwachen diese. Obwohl sie strenggenommen keine Dienste sind, haben sie als aktives Element einer SOA eine exponierte Stellung.
- *Basic Services* stellen grundlegende Anwendungslogik bereit und sind die Grundlage einer SOA.
- *Intermediary Services* übernehmen zum Einen als Adapter [97] oder Façade [97] zu weiteren Systemen technische Aufgaben. Zum anderen kombinieren sie *Basic Services* um funktionalen Mehrwert zu schaffen.
- *Process-centric Services* kapseln einen konkreten Geschäftsprozess und verwalten dessen Zustand.

Während ein *Process-centric Service* zustandsbehaftet ist, sollten sowohl *Basic Services* als auch *Intermediary Services* über keinen internen Zustand verfügen. *Basic Services* werden nur von anderen Diensten verwendet, wohingegen sich *Intermediary Services* und *Process-centric Services* selbst auf andere Dienste abstützen und wiederum von anderen Diensten verwendet werden.

Basierend auf der Klassifikation definieren Krafzig et al. [150] unterschiedliche Reifegrade serviceorientierter Architekturen. Jede Kategorie spiegelt eine mögliche Ebene einer SOA wider (vgl. Abb. 2.3). Umso mehr Ebenen von einer SOA abgedeckt werden, umso höher ist der Reifegrad der Architektur

und die Hebelwirkung für ein Unternehmen bei der flexiblen Umsetzung seiner Geschäftsprozesse. Folglich wird eine *Process enabled SOA* mit allen in Abbildung 2.3 dargestellten Ebenen als Optimum angesehen.

2.3 OSGI: EIN DYNAMISCHES MODULSYSTEM FÜR JAVA

Die OSGi Alliance [208][1] spezifiziert mit der *OSGi Service Platform* ein dynamisches Modulsystem für Java [176], das eine komponentenbasierte und serviceorientierte Entwicklung von Anwendungen ermöglicht [278]. Durch die Einführung eines Komponentenmodells in Java [142] können mit der *OSGi Service Platform* Anwendungen modularisiert und verwaltet werden. Die entsprechende Spezifikation wird seit 1999 von der OSGi Alliance verabschiedet und existiert mittlerweile in der sechsten Generationen. Die OSGi Alliance besteht als Industriekonsortium neben großen Unternehmen wie bspw. IBM, der Deutschen Telekom, Hitachi, NTT oder Oracle auch aus einer Vielzahl kleinerer Unternehmen. Ursprünglich zielte die Spezifikation auf die Entwicklung einer dynamischen Architektur für vernetzte, eingebettete Systeme [207]. Inzwischen wird die Technologie verstärkt bei Unternehmensanwendungen, zur intelligenten Fernsteuerung von Hausgeräten, im Open-Source-Bereich oder für Telematik-Anwendungen eingesetzt. Im Automobil wird OSGi mittlerweile von z. B. Audi, BMW, Daimler, Ford oder Volvo verwendet [127].

Bundles

Den Kern der *OSGi Service Platform* bildet das *OSGi Framework* [206], das eine Laufzeitumgebung für Bundles und Services oberhalb der Java Virtual Machine (JVM) darstellt. Ein Bundle ist dabei eine fachlich oder technisch zusammenhängende Einheit, die sowohl Klassen als auch Ressourcen (z. B. Bilder oder Textdateien) enthält. Damit andere Bundles darauf zugreifen können, müssen die *Packages*, in denen diese Klassen und Ressourcen enthalten sind, explizit exportiert werden. Das Exportieren erfolgt deklarativ durch eine Manifestdatei. Darüber hinaus müssen auch *Packages* oder Bundles, die verwendet werden sollen, deklarativ in der Manifestdatei importiert werden. Ein Bundle kann eigenständig in der Laufzeitumgebung installiert und auch wieder deinstalliert werden. Dabei werden die in der Manifestdatei beschriebenen Abhängigkeiten zwischen den Bundles automatisch durch das *OSGi Framework* aufgelöst.

Dementsprechend besteht eine Applikation – wie in Abbildung 2.4 dargestellt – aus einer Menge von Bundles. Obwohl jedes Bundle eigenständig installiert werden kann, bestehen zwischen den Bundles jedoch Abhängigkeiten. Werden Klassen oder Ressourcen aus einem anderen Bundle importiert, ist dies in Abbildung 2.4 durch einen gestrichelten Pfeil mit dem Sterotyp *required* gekennzeichnet. Zwar können dadurch Schnittstellen und (abstrakte) Klassen geteilt werden, die in dem importierenden Bundle verwendet oder sogar erweitert werden. Allerdings entsteht dadurch eine enge Kopplung zwischen den beiden Bundles. Da während der Laufzeit Bundles sowohl neu installiert als auch deinstalliert werden können, löst das *OSGi Framework* diese in der Manifestdatei beschriebenen Abhängigkeiten automatisch auf und berücksichtigt dabei eine Versionierung. Dementsprechend entlädt es auch Bundles, deren Abhängigkeiten vorläufig nicht mehr aufgelöst werden können. In Abbildung 2.4 wird bspw. *Bundle D* entladen, falls *Bundle A* oder *Bundle C* deinstalliert werden.

Services in OSGi

Während Bundles Klassen und Ressourcen in eigenständige Einheiten unterteilen und es erlauben, deren Abhängigkeiten explizit zu beschreiben,

1 ursprünglich als Open Services Gateway initiative (OSGi) gegründet

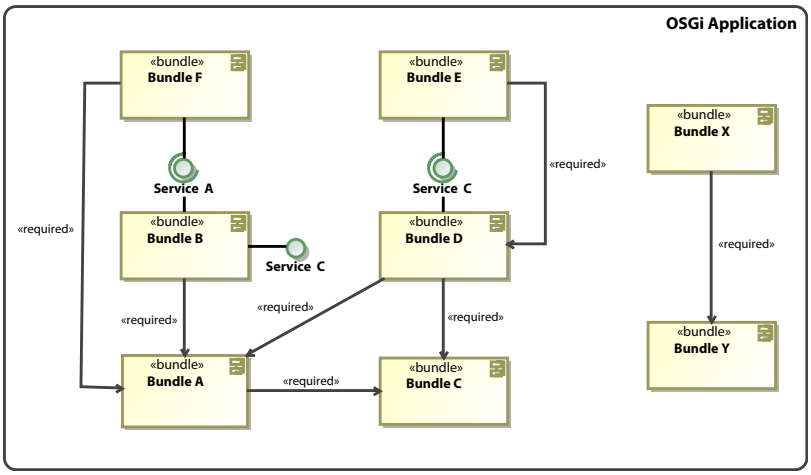


Abbildung 2.4: Eine OSGi-Applikation besteht aus einer Menge an Bundles, zwischen denen es klar definierte Abhängigkeiten gibt.

ermöglichen Services eine entkoppelte Kollaboration zwischen verschiedenen Bundles. Dazu kann Funktionalität von einem Bundle über einen Service im *OSGi Framework* bereitgestellt und von einem anderen Bundle abgerufen werden (vgl. Abb. 2.4). Um einen Service bereitzustellen, kann ein Bundle ein POJO[2] beim *OSGi Framework* als Service unter einer oder mehreren Schnittstellen registrieren. Ein veröffentlichter Dienst kann von anderen Bundles entdeckt und über die entsprechende Schnittstellen verwendet werden. In Abbildung 2.4 veröffentlicht bspw. *Bundle B* zwei Services unter den Schnittstellen A bzw. C. *Bundle F* nutzt einen der beiden von *Bundle B* bereitgestellten Dienste.

Durch die Dynamik von OSGi können Bundles zur Laufzeit installiert und ebenso deinstalliert werden. Folglich können auch Services zur Laufzeit neu registriert bzw. entfernt werden. Dies hat zur Folge, dass bei der Verwendung von Services zuerst sichergestellt werden muss, ob der zu verwendende Dienst noch zur Verfügung steht [151]. Falls *Bundle D* in Abbildung 2.4 entladen wird, wird auch dessen Service entfernt. Bundles, die diesen Dienst nutzen, müssen darauf reagieren und bspw. den von *Bundle B* bereitgestellten Dienst nutzen. Das Modell von Services in OSGi ermöglicht einen Zugriff auf Dienste, d. h. implementierte Funktionalität, der nur über eine Schnittstellenspezifikation erfolgt. Die Auswahl einer konkreten Implementierung erfolgt während der Laufzeit und kann ggf. geändert werden.

OSGi Spezifikation

Der Aufbau eines *OSGi Frameworks* wird durch eine Spezifikation [206] beschrieben. Die OSGi Alliance spezifiziert darin lediglich Programmierschnittstellen und Testfälle für eine Implementierung. Als Referenzimplementierung dient das von der Eclipse Foundation entwickelte Equinox [78]. Daneben existieren weitere Open-Source-Entwicklungen (z. B. Apache Felix [21], Knopflerfish [148]) und kommerzielle Produkte (z. B. ProSyst [222]). Das *OSGi Framework* ist in mehrere logischen Schichten gegliedert, die in Abbildung 2.5 dargestellt sind. Grundlage für das *OSGi Framework* ist immer eine JVM. Da es allerdings auf unterschiedlichen Java-Plattformen (z. B. Java SE oder Java ME) lauffähig sein soll, definiert die Spezifikation *Execution Environments*. Ein *Execution Environment* repräsentiert ein oder mehrere Java Runtime Environments (JREs) und legt fest, welche Klassen, Interfaces und

2 POJO steht im Englischen für *Plain Old Java Object* und bezeichnet einfaches Java-Objekt ohne besondere Merkmale.

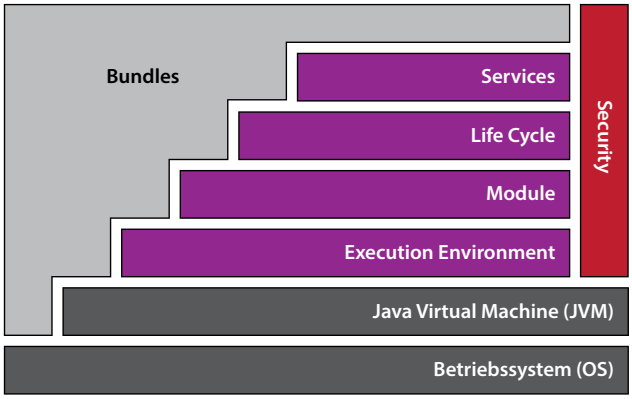


Abbildung 2.5: Das OSGi-Framework erweitert Java bzw. die JVM um ein Komponentenmodell und besteht aus mehreren logischen Schichten, die aufeinander aufbauen (vgl. [206]).

Methoden in der zugrunde liegenden JRE vorhanden sein müssen. Somit können einzelne Bundles ein *Execution Environment* definieren, auf das sie aufbauen, und das *OSGi Framework* kann vor der Installation des Bundles entscheiden, ob es in der aktuellen Laufzeitumgebung ausführbar ist.

Die darauf aufbauende *Module*-Schicht definiert ein Komponentenmodell für Java und spezifiziert ein Bundle als eine Einheit, die eigenständig installiert und deinstalliert werden kann. Ein Bundle wird technisch als ein Java ARchive (JAR) realisiert, das zum Einen die zur Implementierung der Funktionalität notwendigen Klassen und Ressourcen und zum Anderen eine Manifestdatei enthält. Diese enthält zusätzliche Informationen über das Bundle wie bspw. dessen Namen, Version oder Hersteller. Auch werden dort die exportierten und importierten *Packages* beschrieben. Das Manifest wird vom *OSGi Framework* für ein korrektes Ausführen des Bundles zwingend benötigt. Während die *Module*-Schicht den statischen Teil behandelt, wird in der *Lifecycle*-Schicht das dynamische Verhalten von Bundles definiert. Darin werden vor allem die Zustände festgelegt, in denen sich ein Bundle während seines Lebenszyklus im *OSGi Framework* befinden kann. Während die *Module*- und *Lifecycle*-Schicht die Organisation von Bundles spezifizieren, bietet die *Service*-Schicht ein dynamisches und konsistentes Programmiermodell zwischen Bundles an [206]. Die *Security*-Schicht ist Bestandteil des Sicherheitskonzepts der *OSGi Service Platform* und ermöglicht es, die Ausführungsrechte einzelner Bundles gezielt einzuschränken.

Neben dem *OSGi Frameworks* und dessen Schichten definiert die Spezifikation eine Reihe von Diensten. Der *Conditional Permission Admin Service* definiert bspw. ein Application Programming Interface (API) zum Erstellen und Verwalten von Sicherheitsberechtigungen. Der *URL Handlers Service* dagegen ermöglicht es, eigene URL Handler dynamisch im System an- und abzumelden. Zusätzlich spezifiziert das Compendium [205] weitere Dienste, die als Lösungen für unterschiedliche Problemstellungen in eigenen Anwendungen verwendet werden können. Dazu zählen die *Remote Services*, die eine Verteilung von Diensten über mehrere *OSGi Frameworks* ermöglichen, und der *Event Admin Service*, über den Nachrichten bzw. Events einerseits veröffentlicht und andererseits abonniert werden können. Über den *UPnP™ Device Service* kann die *OSGi Service Platform* mit Geräten zusammenarbeiten, die Universal Plug and Play (UPnP) unterstützen.

Declarative Services

Ein weiterer Bestandteil des Compendiums [205] ist die *Declarative Ser-*

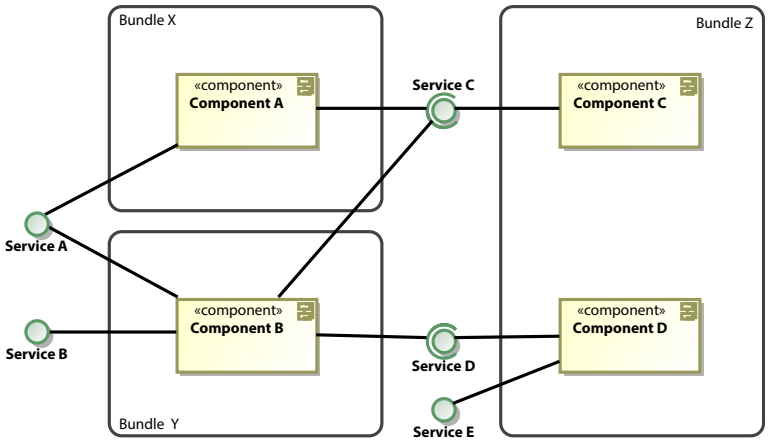


Abbildung 2.6: Die Kollaboration von Bundles wird in OSGi über Services realisiert. Durch *Declarative Services* können bereitgestellte und benötigte Dienste einzelner Komponenten eines Bundles spezifiziert werden. Die Laufzeitumgebung übernimmt die Instanziierung und Verknüpfung der Dienste.

vices Spezifikation. Darin wird ein Programmiermodell beschrieben, das den Umgang mit dynamischen Diensten vereinfacht. Durch *Declarative Services* kann ein Bundle sogenannte *Components* definieren, deren Abhängigkeiten zu anderen Diensten deklarativ beschrieben werden (vgl. Abb. 2.6). Dabei werden nicht nur die benötigten Dienste beschrieben, sondern auch die bereitgestellten Dienste. Zur Laufzeit werden die *Components* und deren Abhängigkeiten durch die Service Component Runtime (SCR) verwaltet. Ein Bundle muss sich dadurch nicht mehr selbst um die Verwaltung benötigter Dienste kümmern.

Abschließend lässt sich festhalten, dass es sich bei einem Bundle um eine *Greybox*-Komponente handelt (vgl. Abschn. 2.1), die durch das Manifest Teile der Implementierung kontrolliert offenlegt. Dieser Teil der Implementierung kann direkt (wieder)verwendet und durch Vererbung erweitert werden. Durch eine Versionierung von Bundles kann das Problem der Austauschbarkeit reduziert werden. Die expliziten Abhängigkeiten eines Bundle (vgl. Abschn. 2.1) werden ebenfalls im Manifest beschrieben und umfassen die notwendige Laufzeitumgebung und benötigte *Packages* bzw. Bundles. Durch Services können Bundles Funktionalität über wohldefinierte Schnittstellen bereitstellen. Zudem bieten die *Declarative Services* eine einfache Möglichkeit, bereitgestellte und notwendige Dienste zu spezifizieren und diese Abhängigkeiten automatisch aufzulösen.

Das *OSGi Framework* kann als „SOA innerhalb einer JVM“ [278] bezeichnet werden und bietet eine einfache und komfortable Möglichkeit, Java-Objekte als Dienste zu registrieren und über die *Service Registry* aufzufinden. Ein Kontrakt lässt sich teilweise über Eigenschaften realisieren, die zu jedem Service definiert werden können. Eine Verteilung kann entweder über *Remote Services* oder externe Technologien wie z. B. die Service Component Architecture (SCA) [194] oder Webservices und dem Devices Profile for Web Services (DPWS) [76] realisiert werden. Ein übergeordneter Aspekt ist jedoch, wie man Dienste innerhalb einer Roboterzelle auswählt und strukturiert, um die in Abschnitt 2.2 vorgestellte Orientierung von Services an Geschäftsprozessen geeignet auf Automatisierungsprozesse zu übertragen.

3 ÜBER DIE PROGRAMMIERUNG VON ROBOTERZELLEN

Bevor man der Frage nachgehen kann, wie man Software für Roboterzellen modularisieren und auf Basis einer serviceorientierten Architektur strukturieren kann, ist es wichtig, sich mit deren Aufbau und deren aktueller Programmierung auseinanderzusetzen. Zentrales Element jeder Roboterzelle ist ein Industrieroboter, der gemäß der ISO 8373 [73] definiert wird als ein „automatisch gesteuerter, frei programmierbarer Mehrzweck-Manipulator, der in drei oder mehr Achsen programmierbar ist und zur Verwendung in der Automatisierungstechnik“ [73] bestimmt ist. Die Norm versteht unter *frei programmierbar*, dass die „programmierten Bewegungen oder Hilfsfunktionen ohne physischen Eingriff“ [73] verändert werden können, d. h. ohne eine „Veränderung der mechanischen Struktur oder der Steuerung“ [73]. Eine Modifikation des Steuerprogramms ist jedoch möglich.

Als Manipulator definiert die Norm eine „Maschine, deren Mechanismus aus einer Folge von Komponenten besteht, [die] durch Gelenke oder gegeneinander verschieblich verbunden“ [73] sind. Ein Manipulator dient „dem Zweck, Gegenstände (Werkstücke oder Werkzeuge) zu greifen und/oder zu bewegen“ [73]. Ein Robotersystem besteht nicht nur aus dem Manipulator, sondern umfasst auch dessen Steuerung, die Endeffektoren und alle weiteren Einrichtungen (z. B. Sensoren), die zur Ausführung der Aufgabe notwendig sind. Ein Endeffektor ist dabei eine „Vorrichtung, die speziell zum Anbringen an die mechanische Schnittstelle [des Manipulators] konzipiert ist [und] mit der der Roboter seine Aufgabe erfüllt“ [73]. Typische Beispiele sind Greifer oder Schweißbrenner. Die mechanische Schnittstelle befindet sich am Roboterflansch, d. h. am Ende der Gelenkstruktur, und ermöglicht eine flexible Ausrüstung mit Endeffektoren.

Die Norm zeigt, dass Industrieroboter mechanisch flexible Maschinen sind, die erstens frei programmierbar sind und die zweitens durch Endeffektoren an unterschiedliche Aufgaben angepasst werden können. Die hohen Anforderungen hinsichtlich Genauigkeit, Zuverlässigkeit und Geschwindigkeit haben dazu geführt, dass bereits früh eigene Robotersteuerungen entwickelt wurden, die mit speziellen Roboterprogrammiersprachen programmiert werden. Diese Sprachen erlauben neben der Bewegungsprogrammierung des Roboters eine Interaktion mit dem Endeffektor. Komplexe Roboterzellen oder ganze Fertigungsverbünde werden zusätzlich von einer übergeordneten Steuerung kontrolliert. Dazu werden üblicherweise speicherprogrammierbare Steuerungen verwendet.

Da es sich hier um eine interdisziplinäre Arbeit handelt, werden in diesem Kapitel die Grundlagen der Industrierobotik und Automatisierung erläutert. Daher richtet sich das Kapitel an Leser mit softwaretechnischem Hintergrund. Dementsprechend wird in Abschnitt 3.1 zuerst der Aufbau einer Robotersteuerung erläutert. Anschließend werden die Besonderheiten von Roboterprogrammiersprachen in Abschnitt 3.2 beschrieben. Dazu wird exemplarisch die KUKA Robot Language (KRL) detaillierter betrachtet. Danach wird die Funktionsweise speicherprogrammierbarer Steuerungen in Abschnitt 3.3 erklärt. Abschließend folgt in Anlehnung an [122] eine kurze Bewertung des aktuellen Stands der Technik in Abschnitt 3.4.

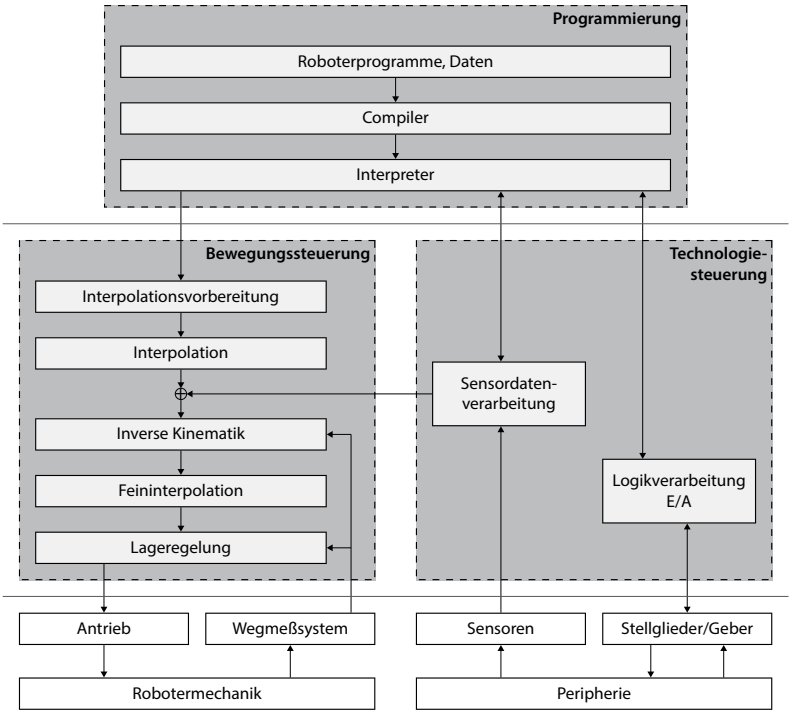


Abbildung 3.1: Schematischer Aufbau einer Robotersteuerung nach [284]: Die Bewegungssteuerung verarbeitet Fahrbefehle und erzeugt Führungsgrößen für die Achsantriebe des Roboters. Über die Technologiesteuerung werden externe Peripheriegeräte angesteuert.

3.1 AUFBAU VON ROBOTERSTEUERUNGEN

Um der mechanischen Flexibilität eines Industrieroboters gerecht zu werden, übernehmen Robotersteuerungen eine Vielzahl von Aufgaben. Dazu zählt zuerst die Bewegungssteuerung des Roboters, an die hinsichtlich Genauigkeit und Geschwindigkeit hohe Anforderungen gestellt werden. Daneben übernehmen Robotersteuerungen aber auch die Aufgabe, die Funktionen des Endeffektors anzusteuern, Zuführsysteme (z. B. Förderanlagen) und externe Achsen (z. B. Linearachsen) zu kontrollieren, sowie Sensorsignale auszuwerten und zu verarbeiten. Eine besondere Herausforderung ist dabei, diese externen System mit der Roboterbewegung zu synchronisieren, um die Aufgaben des Roboters sowohl effizient als auch mit der notwendigen Qualität zu erfüllen. Neben der Kommunikation mit externen Fertigungsrechnern und übergeordneten Steuerungen, muss eine Robotersteuerung auch über eine graphische Benutzerschnittstelle verfügen. Diese dient zum Einen während der Inbetriebnahme zur Entwicklung, Überprüfung und ggf. Korrektur von Roboterprogrammen und zum Anderen während des Betriebs zur Fehlererkennung und -behebung.

Schematisch ist der Aufbau einer Robotersteuerung in Abbildung 3.1 dargestellt (vgl. [284]). Eine Robotersteuerung lässt sich in drei große Blöcke - *Programmierung*, *Bewegungssteuerung* und *Technologiesteuerung* – unterteilen. Der Block *Programmierung* ist für die Erstellung und Ausführung von Roboterprogrammen zuständig und umfasst die dafür erforderlichen Werkzeuge (z. B. einen Editor) für den Benutzer. Durch spezielle, in der Regel proprietäre Programmiersprachen kann so die Aufgabe des Robotersystems spezifiziert werden (vgl. Abschn. 3.2). Das Programm wird durch einen entspre-

chenden *Compiler* übersetzt und kann anschließend mit einem speziellen *Interpreter* Befehl für Befehl ausgeführt werden. Fahrbefehle für den Roboter werden in der *Bewegungssteuerung* weiterverarbeitet, um die jeweiligen Führungsgrößen für die Regelung der Roboterantriebe zu erzeugen. In der *Technologiesteuerung* werden zum Einen Sensordaten erfasst und verarbeitet, die als Bahnkorrektur in die Bewegungssteuerung fließen können. Zum Anderen ist die *Technologiesteuerung* für die Kommunikation mit externen Peripheriegeräten (z. B. den Endeffektoren) über verschiedene Ein-/Ausgabeschnittstellen (E/A) verantwortlich. Mithilfe einer logischen Informationsverarbeitung können Peripheriegeräten überwacht, gesteuert und mit dem Roboter synchronisiert werden.

Bewegungsarten

In der Bewegungssteuerung unterscheidet man zwischen der Punkt-zu-Punkt-Steuerung (engl.: *point to point*) und der Bahnsteuerung (engl.: *continuous path*). Bei Point-To-Point- oder kurz PTP-Bewegungen wird jede Roboterachse einzelnen mit einer vorgegebenen Geschwindigkeit und Beschleunigung an die jeweilige Zielposition verfahren. Beeinflusst durch die mechanische Struktur des Roboters und seiner in der Regel rotatorischen Achsen, ergibt sich eine ungleichförmige, geschwungene Bahn für den Tool Center Point (TCP). Als „Werkzeugarbeitspunkt“ [73] ist dieser ein für eine gegebene Anwendung definierter Punkt des Endeffektor mit Bezug zum Roboterflansch (z. B. die Spitze eines Schweißbrenners oder der Mittelpunkt zwischen zwei Greiferbacken). Bei einer PTP-Bewegung wird der TCP auf der schnellsten, jedoch nicht auf der kürzesten und damit geradlinigen Bahn zum Zielpunkt geführt.

Bei Bahnbewegungen wird dagegen die Bewegung des TCPs im kartesischen Raum durch eine mathematische Funktion beschrieben. Diese Funktion beschreibt die Bewegungsbahn und gibt an, wie sich der TCP in Abhängigkeit der Zeit gegenüber der Roboterbasis bewegt. Die Bewegungsbahn besteht aus einer geordneten Reihe von Posen in der Zeit [73]. Eine Pose beinhaltet sowohl die Position als auch die Orientierung des TCPs im Raum. Typische Bahnbewegungen sind lineare und zirkulare Bewegungen sowie Splines [31]. Bei einer linearen oder zirkularen Bewegung wird der TCP entlang einer Geraden bzw. eines Kreises im kartesischen Raum geführt. Dementsprechend wird der TCP bei einer Spline-Bewegungen entlang der Bahn geführt, die durch den Spline definiert ist.

Interpolation und Ausführung von Bewegungen

In Abbildung 3.1 ist ebenfalls der Ablauf der Bewegungssteuerung dargestellt. Der Interpreter des Roboterprogramms leitet einen Fahrbefehl an die Bewegungssteuerung weiter. Dort durchläuft der Bewegungsbefehl zuerst die *Interpolationsvorbereitung*, die alle Daten ermittelt, die vor dem eigentlichen Abfahren der Bahn berechenbar sind. Dazu zählt die Transformation von Posen in das Basiskoordinatensystem des Roboters und die Berechnung der Bahnlänge. Robotersteuerungen haben zudem oft eine *Look-Ahead*-Funktionalität. Das bedeutet, dass Bewegungen bereits im Voraus geplant und vorbereitet werden. Damit kann bei hoher Geschwindigkeit des Roboters zwischen einer laufenden und einer neuen Bewegung umgeschaltet werden. Dies ist – insbesondere bei kurzen Bewegungen – wichtig, um den Zielpunkt einer Bewegung nicht exakt anzufahren, sondern nur zu *überschleifen* [183]. Da das Bremsen und Wiederbeschleunigen beim Erreichen dieses Punktes entfällt, werden dadurch zum Einen kürzere Taktzeiten erreicht und zum Anderen die Mechanik geschont.

Nach der Vorbereitung folgt die eigentliche *Interpolation* der Bewegung, die den kontinuierlichen Bewegungsverlauf diskretisiert. In einem Interpolationstakt, der typischerweise zwischen 1 ms und 20 ms liegt, wird die Be-

wegung für eine vorgegebene Geschwindigkeit und Beschleunigung diskretisiert. Abhängig vom verwendeten Fahrprofil, z. B. mit konstanter Beschleunigung oder konstantem Ruck (vgl. [31]), ergeben sich unterschiedliche Zwischenpunkte. Für Bahnbewegungen werden Sollwerte für die Pose des TCPs erzeugt, die durch die Funktion der *inversen Kinematik* (vgl. Abschn. 5.5 bzw. [63, S. 101 ff.]) in Winkelvorgaben für die Roboterachsen transformiert werden. Für PTP-Bewegung werden durch die Interpolation direkt Winkelvorgaben generiert. Durch den festen Interpolationstakt ergibt sich anhand der Sollwerte implizit die einzustellende Geschwindigkeit. In der *Feininterpolation* werden die Winkelvorgaben in einem höheren Takt ggf. nochmals interpoliert und an die Lageregelung der Roboterachsen weitergeben. Das Regelsystem hat die Aufgabe, die Vorgaben exakt einzustellen.

Robotersteuerungen und Echtzeit

Die oben genannten Zeitvorgaben für den Interpolationstakt zeigen, dass Robotersteuerungen hohe Anforderungen an die Einhaltung von Zeitschranken haben. Das betrifft insbesondere die Bewegungs- und Technologiesteuerung. Folglich ist eine Robotersteuerung ein Rechnersystem, „bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind“ [74]. Man spricht von Echtzeitbetrieb. Nach Benveniste und Berry [29] wird ein Echtzeitsystem (engl.: *real-time systems*) wie folgt charakterisiert:

> “It is commonly accepted to call real-time a program or system that receives external interrupts or reads sensors connected to the physical world and outputs commands to it.” [29]

Wie Abbildung 3.1 zeigt, ist eine Robotersteuerung in Interaktion mit der *physischen Welt*: entweder über Sensoren, wie z. B. das Wegemesssystem einer Roboterachse, oder über Aktuatoren, wie bspw. die Roboterachsen oder Peripheriegeräte. Bei Buttazzo [55] wird die Definition eines Echtzeitsystems enger gefasst und insbesondere mit den Auswirkungen verknüpft:

> “Real-time systems are computing systems that must react within precised time constraints to events in the environment. As a consequence, the correct behaviour of these systems depends not only on the value of the computation but also on the time at which the results are produced.” [55, S. 1]

Das bedeutet, dass nur bei Einhaltung der Zeitschranken ein korrekter Betrieb gewährleistet werden kann. Eine Robotersteuerung kann die Antriebe nur präzise steuern, wenn der feste aber sehr hohe Interpolationstakt ausreichend genau eingehalten wird. Ist dies nicht der Fall ist, ergibt sich auch keine ausreichend genaue Bewegung des TCPs auf der definierten Bahn.

Da die Einhaltung der Zeitschranken entscheidend für den Betrieb des Systems ist, spricht man in diesem Zusammenhang von harten Echtzeitbedingungen [284]. Folglich bezeichnet man ein solches System, in diesem Fall die Robotersteuerung, als *hartes Echtzeitsystem*, das von [34] wie folgt beschrieben wird:

> “In [. . .] hard real-time systems it is crucial for the task in the system to meet their specified *deadlines*, otherwise the tasks are worthless, or worse, cause catastrophic results.” [34]

Da bei einem Robotersystem, insbesondere bei einem industriellen Fertigungssystem, unter Umständen große Massen bewegt und gefährliche Prozesse ausgeführt werden, können nicht eingehaltene Zeitschranken schwerwiegende Auswirkungen haben.

Ein weiteres, vor allem für die Industrierobotik entscheidendes, Kriterium von Echtzeitsystemen ist deterministisches Verhalten. Für Robotersteuerungen bedeutet dies, dass ein Programm sich unter identischen Rahmenbedingungen identisch verhalten muss. Eine Bewegung muss dementsprechend immer exakt gleich von der Robotersteuerung abgefahren werden. Auch die Kommunikation mit externen Systemen (z. B. einer Schweiß- oder Greifersteuerung) muss vollkommen gleich sein. Dies betrifft v. a. Reaktionsverzögerungen, da diese einmalig während der Inbetriebnahme eingestellt werden und anschließend reproduzierbar sein müssen.

Die meisten Robotersteuerungen sind PC-basiert und verfügen über ein Echtzeitbetriebssystem. Dieses sorgt dafür, dass Prozesse und Threads mit der entsprechenden „Rechtzeitigkeit und Gleichzeitigkeit“ [284, S. 344] ausgeführt werden können. Ein Beispiel für eine PC-basierte Robotersteuerung ist die KUKA Robot Control (KRC). Sie unterteilt sich in zwei getrennte Betriebssysteme: Die grafische Benutzerschnittstelle läuft vollständig unter Windows XP Embedded von Microsoft. Dagegen übernimmt das Echtzeitbetriebssystem VxWorks [282] von WindRiver alle echtzeitkritischen Aufgaben wie bspw. die Bewegungs- und Technologiesteuerung (vgl. Abb. 3.1). Die Ausführung der in KRL (vgl. Abschn. 3.2) geschriebenen Roboterprogramme erfolgt daher in VxWorks. Dazu wird der Programmtext vollständig auf das Echtzeitbetriebssystem übertragen, kompiliert und kann anschließend kann mittels eines Interpreters ausgeführt werden. Die Verwaltung der beiden parallel laufenden Betriebssysteme wird von VxWin [96] übernommen. Nachdem Windows geladen wurde, startet VxWin das Echtzeitbetriebssystem, das daraufhin die Kontrolle über das System übernimmt. Dementsprechend bekommt Windows vom Echtzeitbetriebssystem immer nur Rechenzeit zugewiesen, wenn diese nicht anderweitig benötigt wird. Die Kommunikation zwischen den Betriebssystemen findet über gemeinsam genutzte Speicherbereiche statt.

Beispiel: KUKA Robot Control

3.2 ROBOTERPROGRAMMIERUNG

Die ersten industriellen Roboter wurden durch einfaches Vormachen programmiert [167]. Dazu wurde der Roboter manuell über ein Bediengerät an die gewünschte Position verfahren. Gleichzeitig wurden die Positionswerte der Roboterachsen gespeichert und an passenden Positionen wurden die Aktionen des Endeffektors festgelegt (z. B. das Öffnen oder Schließen eines Greifers). Das resultierende Programm besteht aus einer Sequenz von Positionswerten, die der Roboter reproduzieren kann, und den Signalen des Endeffektors. Diese Methode wird *teaching by showing* genannt und eignet sich für sehr einfache Abläufe, die keine Schleifen, Verzweigungen und Berechnungen benötigen. Anspruchsvolle Anwendungen mit komplexen Abläufen benötigen allerdings eine Programmiersprache, um die Aufgaben adäquat zu beschreiben. Dazu wurden spezielle Programmiersprachen, so genannte *Roboterprogrammiersprachen*, entwickelt, die die Bedürfnisse und Besonderheiten der Robotik berücksichtigen (z. B. für die Spezifikation von Bewegungen oder die Werkzeugansteuerung). Eine Übersicht ausgewählter Roboterprogrammiersprachen, sowohl aus dem akademischen Umfeld als auch von kommerziellen Roboterherstellern, ist in Tabelle 3.1 zu finden.

Roboterprogrammiersprachen

Die erste spezielle Programmiersprache für Roboter wurde bereits in den 1960ern am Massachusetts Institute of Technology (MIT) entwickelt und war für die Programmierung der mechanischen Hand MH-1 konzipiert [81]. In den 1970er Jahren wurden an der Stanford University mit WAVE [213] und

NAME	ENTWICKLER	ANMERKUNGEN
AL [84]	Stanford University	entwickelt 1974, basiert auf WAVE, ähnlich zu ALGOL
AML [257]	IBM	entwickelt 1977, A Manufacturing Language
ARLA	ASEA/ABB	von 1981 bis 1992, ASEA programming Robot LAnguage
Inform III	YASKAWA	
KAREL	Fanuc	ähnlich zu Pascal
KRL [156]	KUKA	ähnlich zu Pascal, KUKA Robot Language
MHI [81]	MIT	entwickelt 1960, Mechanical Hand Interpreter
RAPID	ABB	ab 1992 Nachfolger von ARLA
URScript	Universal Robots	Scriptsprache
V+	Adept	basiert auf VAL
VAL [244]	Unimation	entwickelt 1975, basiert auf WAVE, Variable Assembly Language
VAL3	Stäubli	basiert auf VAL
WAVE [213]	Stanford University	entwickelt 1970

Tabelle 3.1: Übersicht ausgewählter Roboterprogrammiersprachen: Die kursiv gedruckten Programmiersprachen finden sich auf industriellen Robotersteuerungen aktuell im Einsatz.

AL [84] weitere Roboterprogrammiersprachen entwickelt. Während WAVE noch einer maschinennahen Assemblersprache ähnelt, ist AL bereits als Hochsprache mit Ähnlichkeiten zu ALGOL und Pascal konzipiert. Die erste kommerzielle Roboterprogrammiersprache war die Variable Assembly Language (VAL) [244] ab Mitte der 1970er Jahre. VAL wurde von der Firma *Unimation Inc.* entwickelt und wurde für die Programmierung der PUMA-Roboter eingesetzt. Wie Tabelle 3.1 zeigt, verwenden im Moment fast durchgehend alle kommerziellen Hersteller eine eigene proprietäre Programmiersprache für ihre Industrieroboter. Gemäß [217] sind diese Sprachen in ihrer Syntax noch immer sehr ähnlich zu PASCAL. Während einige Sprachen, wie z. B. V+ oder VAL3, Weiterentwicklungen von VAL sind, stellen andere Sprachen, wie z. B. KAREL oder RAPID, Eigenentwicklungen dar.

Online- und Offline-Programmierung

Bei der Entwicklung von Roboterprogrammen wird zwischen der Online- und der Offline-Programmierung unterschieden [108]. Während bei Onlineverfahren das Programm, z. B. durch einen Editor auf dem Bediengerät, direkt am Roboter erstellt wird (vgl. *teaching by showing*), wird bei der Offline-Programmierung eine Simulationsumgebung verwendet. Die Roboteranlage wird dadurch nicht belegt bzw. sie muss noch nicht einmal physisch existieren. Stattdessen wird ein virtuelles Abbild des Robotersystems verwendet, um das Anwendungsprogramm zu erstellen. Eine Offline-Programmierumgebung kann entweder nur einen Texteditor zur Erstellung des Programms bieten oder unterstützt den Entwickler idealerweise durch die Verwendung von CAD-Daten oder die automatische Berechnung kollisionsfreier Roboterbahnen [109]. In der Regel stellt jeder Hersteller für seine Robotersysteme ein eigenes System zur Offline-Programmierung zur Verfügung (z. B. das ABB RobotStudio [1] oder KUKA.Sim Pro [161]). Darüber hinaus existieren auch herstellerunabhängige Systeme, wie z. B. Robcad [245] von Siemens oder DELMIA [64] von Dassault Systèmes.

Das Ergebnis der Offline-Programmierung ist immer ein textuelles Programm in der Roboterprogrammiersprache des Zielsystems [123]. Anschließend muss dieses Programm jedoch in der Roboterzelle getestet und gegebenenfalls händisch angepasst werden. Insbesondere dadurch, dass die virtuelle Roboterzelle in der Regel nicht mit der realen Zelle exakt übereinstimmt, müssen Positionen manuell eingelernt und Bahnen evtl. angepasst werden. Da die Anpassungen im Vergleich zu der Entwicklung des gesam-

ten Roboterprogramms nur einen kleineren Anteil ausmachen, bleiben die Vorteile der Offline-Programmierung bestehen, v. a. die kurze Inbetriebnahme an dem physischen Robotersystem. Applikationen, die verstärkt auf Sensorintegration basieren, lassen sich jedoch nur schwer oder gar nicht mit Offline-Programmierung entwickelt, da der Zugang zur Anlage und den Messungen der Sensoren unabdingbar ist.

Exkurs: KRL

Der Aufbau und die Funktionsweise einer Roboterprogrammiersprache wird exemplarisch an der KUKA Robot Language (KRL) [156] erläutert, der Standardprogrammiersprache der KRC. Da jede KRC genau einen Roboter steuern kann, kann man mit KRL ebenfalls einen Roboter programmieren. Die Syntax der imperativen Programmiersprache KRL ist ähnlich zu PASCAL und beinhaltet neben den typischen Anweisungen wie bspw. Zuweisungen, bedingten Anweisungen, Verzweigungen und Schleifen auch robotikspezifische Elemente. Eine Besonderheit ist die Ausführungssemantik, da sie zwei separate Programmzeiger verwendet (vgl. *Look-Ahead* in Abschn. 3.1). Der Zeiger des *Hauptlaufes* gibt die Programmzeile mit der aktuell ausgeführten Bewegung an. Der *Vorlauf* zeigt hingegen an, an welcher Stelle sich die Programminterpretation bereits befindet. Dies ist notwendig, um Bewegungen im Voraus zu planen und ein *Überschleifen* von Zwischenpunkten zu ermöglichen [183]. Hierbei wird der Zwischenpunkt nicht exakt angefahren, sondern nur annäherungsweise erreicht. Dazu wird im Programm bereits vorzeitig auf den nächsten Bewegungsbefehl umgeschaltet und die Bewegung wird kontinuierlich zum nächsten Punkt übergeleitet.

Ein Beispielprogramm mit dem wichtigsten Sprachelementen von KRL ist in Listing 3.1 dargestellt[1]. Das Programm beginnt mit der Deklaration aller Variablen, anschließend folgen Programmanweisungen. Jede Variable muss am Anfang eines Programm bzw. einer Funktion zuerst deklariert werden, bevor eine Zuweisung stattfinden kann. Neben den üblichen Datentypen (d. h. `INT`, `REAL`, `BOOL` und `CHAR`), erlaubt KRL die Definition von Feldern (engl.: *arrays*) und benutzerspezifischen Strukturen (engl.: *structs*). Auf den Datentypen sind die klassischen Operatoren (z. B. logische Operatoren auf boolschen Werten) definiert. Darüber hinaus existieren bereits vordefinierte Strukturen wie `AXIS` für die Achsstellung eines Roboters und `POS` für die Pose (d. h. die Position und Orientierung) eines Roboters. Für diese Datentypen sind geometrische Operatoren verfügbar.

In KRL gibt es eine große Anzahl globaler Variablen. Sie bestimmen zum Einen die Konfiguration der Steuerung und des Roboters und zum Anderen ermöglichen sie eine Kommunikation mit externen Systemen über einen Feldbus. Ein Beispiel für die Konfiguration des Roboters ist die Systemvariable `$VEL_AXIS`, die die maximale Geschwindigkeit der Roboterachsen bestimmt. Die als Felder definierten Variablen `$IN` und `$OUT` repräsentieren jeweils bis zu 4096 digitale Ein- und Ausgänge, die über einen Feldbus gelesen bzw. geschrieben werden können. Zudem können über die Felder `$ANIN` und `$ANOUT` jeweils bis zu 32 analoge Ein- und Ausgänge adressiert werden. Über diese Variablen findet in der Regel die Kommunikation mit dem Endeffektor bzw. dessen Steuerung statt.

Darüber hinaus kennt KRL die klassischen Kontrollstrukturen imperativer Programmiersprachen. Bedingte Anweisungen bzw. Verzweigungen können durch `IF` mit optionalem `ELSE` ausgedrückt werden. Für die Definition von Schleifen stehen mit `FOR`, `WHILE`, `REPEAT ... UNTIL` und `LOOP` mehrere Möglichkeiten zur Auswahl. Selbst unbedingte Sprünge sind mit `GOTO` möglich. Spezifischer sind `WAIT`-Anweisungen. Mit `WAIT FOR` wird das Pro-

1 Eine ausführliche Befehlsreferenz ist in [156] zu finden.

```
DEF example()
  DECL INT i
  DECL POS cpos
  DECL AXIS jpos

  FOR i = 1 TO 6
    $VEL_AXIS[i] = 60
  ENDFOR

  jpos = {AXIS: A1 0,A2 -90,A3 90,A4 0,A5 0,A6 0}
  PTP jpos

  IF $IN[1] == TRUE THEN
    cpos = {POS: X 300,Y -100,Z 1500,A 0,B 90,C 0}
  ELSE
    cpos = {POS: X 250,Y -200,Z 1300,A 0,B 90,C 0}
  ENDIF

  INTERRUPT DECL 3 WHEN $IN[2] == FALSE DO BRAKE
  INTERRUPT ON 3

  TRIGGER WHEN DISTANCE = 0 DELAY = 20 DO $OUT[2] = TRUE
  LIN cpos
  LIN {POS: X 250,Y -100,Z 1400, A 0,B 90,C 0} C_DIS
  PTP jpos

  INTERRUPT OFF 3
END
```

Listing 3.1: Beispiel für ein KRL-Programm aus [183]

gramm angehalten, bis eine bestimmte Bedingung erfüllt ist. `WAIT SEC` hält den Programmablauf ebenfalls an und setzt ihn nach einer Wartezeit fort. Außerdem kann mit `HALT` das Programm angehalten werden, wobei die letzte Bewegungsanweisung noch vollständig ausgeführt wird. Das Programm kann anschließend nur mit dem Bediengerät fortgesetzt werden.

Für die Bewegungsprogrammierung des Roboters verfügt KRL über verschiedene Befehle, die unterschiedliche Bewegungsarten repräsentieren. Beispielsweise können durch das Schlüsselwort `PTP` Punkt-zu-Punkt-Bewegungen ausgeführt werden. Lineare und kreisförmige Bahnbewegungen können mit dem Schlüsselwort `LIN` bzw. `CIRC` definiert werden. Während hier die geometrische Bahn des Endeffektor im kartesischen Raum beschrieben wird, ist bei einer PTP-Bewegung keine Aussage über die kartesische Bahn möglich (vgl. Abschn. 3.1). Als Startpunkt einer Bewegung dient in KRL immer die aktuelle Position des Roboters. Das bedeutet, dass bei einer Bewegung nur der Zielpunkt spezifiziert werden muss.

Das Beispielprogramm in Listing 3.1 beinhaltet mit `PTP` und `LIN` zwei Bewegungsarten. Die Angabe eines Zielpunktes erfolgt für Bahnbewegungen wie `LIN` in kartesischen Koordinaten (z. B. durch den Datentyp `POS`) und beschreibt die Pose des TCPs zu einem Bezugskoordinatensystem (z. B. der Roboterbasis). Da ein Roboter mit seinem TCP einen Punkt in der Regel mit mehr als einer Achskonfiguration erreichen kann, ist die Angabe einer Pose nicht ausreichend, um die Position des Roboters eindeutig festzulegen. Daher kennt KRL mit *Status* und *Turn* zusätzliche Angaben, um aus mehreren möglichen Achsstellungen eine eindeutige Konfiguration zu bestimmen.

Für PTP-Bewegungen kann der Zielpunkt sowohl in kartesischen Koordinaten als auch direkt in Achswinkeln (z. B. durch den Datentyp `AXIS`) angegeben werden. Die Angabe `C_DIS` hinter dem letzten `LIN`-Befehl in Listing 3.1 bedeutet, dass der Zielpunkt dieser Bewegung überschliffen wird.

Bahnbezogene Schaltaktionen können in KRL durch `TRIGGER` realisiert werden. Damit kann während einer Roboterbewegung beim Erreichen eines definierten Punktes eine Anweisung ausgelöst werden. Mögliche Anweisungen sind eine Wertzuweisung an eine Variable, das Schreiben eines Ausgangs oder der Aufruf eines Unterprogramms. Mit `TRIGGER WHEN DISTANCE` kann eine Anweisung relativ zum Start- oder Zielpunkt einer Bewegung definiert werden (vgl. Lst. 3.1). Ein `TRIGGER WHEN PATH` kann nur für Bahnbewegungen definiert werden und ermöglicht es, die Schaltaktion nicht nur zeitlich sondern auch räumlich auf der Bahn zu verschieben.

In KRL ermöglicht ein `INTERRUPT`, ein Unterprogramm beim Eintreten eines vorher definierten Ereignisses auszuführen. Bei Eintreten des Ereignisses wird das aktuelle Programm unterbrochen und das Unterprogramm wird abgearbeitet. Anschließend wird das Hauptprogramm bei der unterbrochenen Anweisung fortgesetzt. Jedem `INTERRUPT` kann eine Priorität zugewiesen werden. Treten mehrere Interrupts gleichzeitig auf, wird zuerst der Interrupt mit der höchsten Priorität aufgerufen, anschließend folgen die mit niedrigerer Priorität. Listing 3.1 enthält ebenfalls einen `INTERRUPT`, der während der letzten drei Bewegungen aktiv ist und beim Eintreten des Ereignisses (Eingang 2 wird nicht wahr) den Roboter abbremst.

Exkurs: Roboterteams in KRL

Durch das nachladbare Technologiepaket *KUKA.RoboTeam* [159] kann KRL für die Programmierung von bis zu vier kooperierenden Robotern eingesetzt werden. Das bedeutet, dass „die Bahnbewegungen zeitlich und geometrisch aufeinander abgestimmt“ [159] werden. Dementsprechend werden zwei Formen der Kooperation unterstützt, die als *zeitliche Kopplung* bzw. *geometrische Kopplung* bezeichnet werden. Obwohl die Steuerungen der Roboter über eine *daisy chain* miteinander verbunden sind, wird jeder Roboter weiterhin auf seiner KRC programmiert. Die zeitliche Kopplung wird über den Befehl `SYNCCMD` ermöglicht, der auf allen beteiligten Steuerungen einen benannten Synchronisationspunkt definiert. Dabei kann zwischen einer Programm- und einer Bewegungssynchronisation gewählt werden.

Bei der Programmsynchronisation erzwingt KRL das zeitgleiche Durchlaufen des Synchronisationspunktes und ermöglicht den synchronen Start von Bewegungen. Bei der Bewegungssynchronisation wird ebenfalls das zeitgleiche Durchlaufen eines Synchronisationspunktes erzwungen. Zusätzlich wird jedoch die Bewegungszeit der nachfolgenden Fahrbefehle aller Roboter angeglichen. Über die Funktion `LK` (*Linked Kinematic*) erfolgt eine geometrische Kopplung zwischen einem Roboter, dem *Slave*, und dem Flansch eines anderen Roboters, dem *Master*. Dadurch kann der *Slave* dem Master folgen und z. B. mit seinem Endeffektor ein vom ersten Roboter gehaltenes Werkstück bearbeiten, während letzterer das Werkstück bewegt.

Exkurs: Sensoren in KRL

Das Remote Sensor Interface (RSI) [160] ist ein nachladbares Technologiepaket, um Roboterbewegungen oder den Programmablauf mithilfe von Sensordaten zu beeinflussen. RSI erlaubt es, durch Funktionsbausteine eine zyklische Signalverarbeitung zu konfigurieren, die im Interpolationstakt ausgewertet wird. Ein Sensorsignal kann unter Zuhilfenahme von RSI kontinuierlich eine Roboterbewegung beeinflussen. Dadurch können sowohl vorgeplante Bewegungen mit überlagerter Sensorkorrektur als auch sensorgeführte Bewegungen realisiert werden. Die Bewegung kann dabei im kartesischen Raum und auf Ebene der Achswinkel verändert werden.

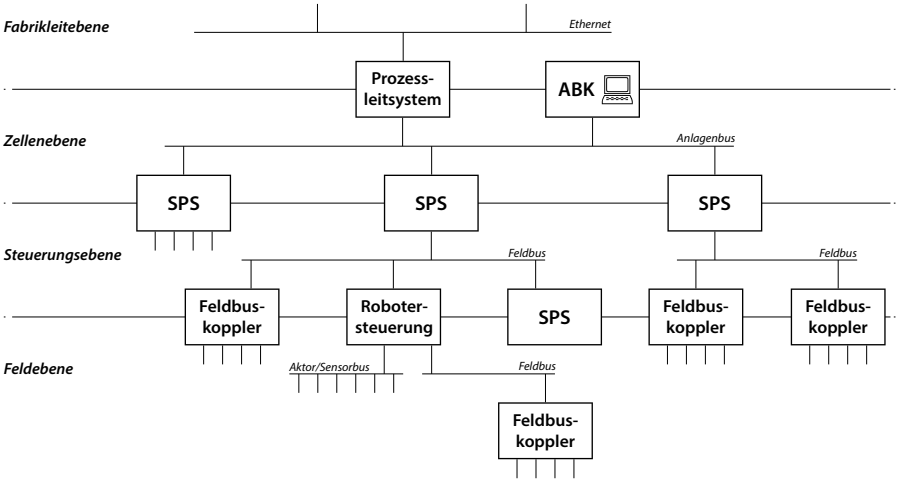


Abbildung 3.2: Dezentrale und hierarchische Automatisierungsstruktur: Auf unterschiedlichen Ebenen koordinieren und überwachen speicherprogrammierbare Steuerungen das Automatisierungssystem.

3.3 SPEICHERPROGRAMMIERBARE STEUERUNGEN

Die Automatisierungstechnik hat die Aufgabe, fertigungstechnische bzw. verfahrenstechnische Systeme selbsttätig zu überwachen und zu steuern. Nach DIN 19 266 [75] versteht man unter *Steuern* den Vorgang, „bei dem Eingangsgrößen andere Größen als Ausgangsgrößen aufgrund der dem System eigentümlichen Gesetzmäßigkeiten beeinflussen“ [75]. Im Gegensatz zur Regelung ist „ein offener Wirkungsweg“ [75] für eine Steuerung charakteristisch. Das bedeutet, dass eine Steuerung ein Steuersignal ohne Berücksichtigung des aktuellen Werts der Ausgangsgröße berechnet. Somit werden auch die Auswirkungen einer Störgröße bei einer Steuerung nicht kompensiert.

Bis Anfang der 1970er Jahre wurde zur Steuerung eines technischen Prozesses eine verbindungsprogrammierbare Steuerung (VPS) verwendet. Dabei wurde die Schaltlogik zur Steuerung des Prozesses durch eine feste Verdrahtung von Bauelementen (z. B. Relais, Schütze oder Schalter) definiert. Bei einer Veränderung der Steuerung musste die Verdrahtung und oft auch die Bestückung der Bauelemente angepasst werden. Dadurch waren diese Steuerungen unflexibel und durch den hohen Verdrahtungsaufwand fehleranfällig. In den 1970er Jahren wurde die speicherprogrammierbare Steuerung (SPS) entwickelt. Dabei wird die Schaltlogik durch Software realisiert und kann so jederzeit verändert werden. Dementsprechend wird eine SPS im Englischen als *Programmable Logic Controller* (PLC) bezeichnet.

Die Struktur von Automatisierungssystemen ist gewöhnlich hierarchisch gegliedert und unterteilt sich gemäß der Automatisierungspyramide [239] in verschiedene Ebenen. Die Automatisierungspyramide beginnt bei der Fabrikleitebene und wird über die Zellebene und Steuerungsebene bis hin zur Feldebene immer breiter. Auf den Ebenen werden unterschiedliche Kommunikationstechnologien bzw. Bussysteme eingesetzt, die sich z. B. bei Echtzeitfähigkeit oder Übertragungsgeschwindigkeit unterscheiden [239]. In Abbildung 3.2 ist exemplarisch die Struktur eines Fertigungssystem dargestellt. Hier zeigt sich, dass speicherprogrammierbare Steuerungen eine zentrale Rolle in einem Fertigungssystem einnehmen. Sie sind zwischen Steuerungs- und Zellebene angesiedelt und kommunizieren über einen Anlagenbus mit einem übergeordneten Prozessleitsystem bzw. mit Anzeige- und Bedien-

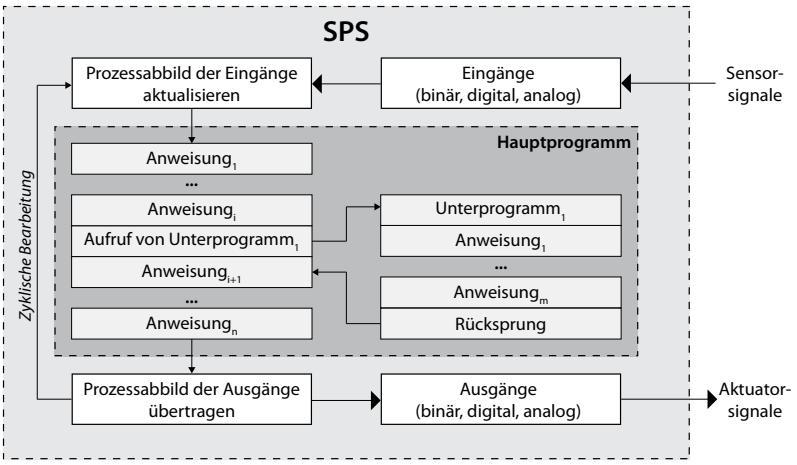


Abbildung 3.3: Funktionsprinzip einer Speicherprogrammierbaren Steuerung: Nach dem Einlesen der Eingänge wird das Programm schrittweise abgearbeitet und abschließend werden die Ausgänge geschrieben. Dieser Ablauf wird zyklisch in einem konstanten, aber einstellbaren Takt wiederholt.

komponenten (ABK). Zusätzlich koordinieren sie über einen Feldbus die Geräte einer Fertigungszelle. Dazu gehören u. a. Robotersysteme, Antriebe, Fördereinrichtungen oder Sensoren. Über einen Feldbuskoppler kann eine dezentrale Anbindung von Ein- und Ausgängen erreicht werden. Die Robotersteuerung ist über einen internen Bus mit den Antrieben des Manipulators und in der Regel über einen Feldbus mit dem Endeffektor verbunden. Aufgrund der hierarchischen Struktur kann auch eine weitere SPS Bestandteil der Fertigungszelle sein und ihrerseits Sensoren und Aktoren über einen Feldbus steuern.

Eine klassische speicherprogrammierbare Steuerung ist eine eigenständige Hardware, die aus mehreren Baugruppen modular aufgebaut ist. Dazu gehört eine Stromversorgungseinheit, eine Zentraleinheit mit CPU und mehrere Signalbaugruppen, die über ein internes Bussystem mit der Zentraleinheit verbunden sind. Als Signalbaugruppen stehen typischerweise digitale bzw. analoge Ein- und Ausgabebaugruppen zur Verfügung. Daneben gibt es noch intelligente Funktionsbaugruppen, die spezielle Aufgaben (z. B. Messung, Reglung und Motorsteuerung) selbständig ausführen und die Zentraleinheit entlasten. Neben diesem Aufbau gibt es speicherprogrammierbare Steuerungen zusätzlich als Slot-SPS und Soft-SPS. Während es sich bei einer Slot-SPS um eine Einsteckkarte für einen PC handelt, ist eine Soft-SPS als Software auf einem Industrie-PC mit Echtzeitbetriebssystem realisiert.

Zyklisches Programmiermodell

Auf einer SPS erfolgt die Programmausführung in der Regel zyklisch, d. h. das Hauptprogramm wird in einem konstanten Takt immer wieder abgearbeitet. Die Taktdauer ist einstellbar und richtet sich nach der Programmlänge [284]. Das Funktionsprinzip einer SPS ist vereinfacht in Abschnitt 3.3 dargestellt. Sensorsignale werden über Eingänge erfasst. Zu Beginn eines Zyklus werden die Signalzustände der Eingänge in das *Prozessabbild der Eingänge* übertragen. Eine weitere Signaländerung eines Einganges bleibt bis zum nächsten Zyklus für das Prozessabbild ohne Folgen. Danach wird das Hauptprogramm aufgerufen und abgearbeitet. Durch Funktionen und Funktionsbausteine kann die Steuerungslogik strukturiert und in Unterprogramme aufgeteilt werden, die wie in Abbildung 3.3 vom Hauptprogramm aufgerufen werden. Zusätzlich unterstützen speicherprogrammier-

ABK.	BEZEICHNUNG	ART	ANMERKUNGEN
LD	Kontaktplan, *Ladder Diagram*	Graphisch	ähnlich zu Stromablaufplänen
IL	Anweisungsliste, *Instruction List*	Textuell	ähnlich zu Assembler
FBD	Funktionsplan, *Function Block Diagram*	Graphisch	ähnlich zu Logikschaltplänen
ST	Strukturierter Text, *Structured Text*	Textuell	ähnlich zu Hochsprachen
SFC	Ablaufsprache, *Sequential Function Chart*	Beides	ähnlich zu Petrinetzen

Tabelle 3.2: Übersicht der einzelnen Programmiersprachen von speicherprogrammierbaren Steuerungen nach IEC 61131 [138].

bare Steuerungen eine ereignisgesteuerte Programmierung, d. h. durch das Auftreten eines Ereignisses (z. B. eine Fehlermeldung) wird die Ausführung des Hauptprogramms kurzzeitig unterbrochen und ein Programm zur Behandlung des Ereignisses aufgerufen.

Üblicherweise werden die Signalwerte der Eingänge aus dem Prozessbild auf Basis des internen Zustands der SPS verarbeitet und resultieren in Änderungen des *Prozessabbilds der Ausgänge*. Die Zustände der Ausgänge ändern sich während der Programmbearbeitung nicht. Erst am Ende eines Zyklus wird das Prozessabbild an die Ausgänge und von dort an die Aktuatoren übertragen. Die für die einmalige Abarbeitung des Hauptprogramms benötigte Zeit wird als *Zykluszeit* bezeichnet und muss ausreichend klein sein, um Signaländerungen an den Eingängen sicher zu registrieren und ggf. durch das Setzen von Ausgängen auf den steuernden Prozess einzuwirken. Folglich muss eine SPS – analog zu einer Robotersteuerung – Echtzeitbedingungen einhalten.

IEC 61131-3

Durch die internationale Norm IEC 61131-3 [138] ist die Programmierung herstellerunabhängig. Die Norm sieht fünf verschiedene Programmiersprachen vor, die in Tabelle 3.2 aufgelistet sind. Der Kontaktplan ist eine graphische Sprache, die stark an die Stromablaufpläne einer VPS erinnert. Daher ist diese Sprache aktuell noch sehr weit verbreitet und eignet sich für die Realisierung einer diskreten Steuerung [106]. Die Anweisungsliste ist eine Assembler-ähnliche textuelle Sprache und kann analog zum Kontaktplan für diskrete Steuerungen eingesetzt werden [106]. Zur Realisierung kontinuierlicher Steuerungen eignet sich laut [106] ein Funktionsplan. Dabei handelt es sich um eine graphische Datenflussprogrammiersprache, die aus Funktionsblöcken mit Ein- und Ausgängen besteht.

Mit der imperativen Programmiersprache Strukturierter Text lassen sich komplexe Algorithmen und mathematische Funktionen geeignet beschreiben [280]. Aufgrund des zyklischen Ausführungsmodells unterscheidet sich jedoch die Verwendung von z. B. Schleifen und Warteoperationen im Vergleich zu imperativen oder objektorientierten Programmiersprachen stark. Eine übergeordnete Position hat die Ablaufsprache, die graphische und textuelle Element vereint und ähnlich zu Petrinetzen ist [3]. Folglich lässt sich mithilfe der Ablaufsprache eine prozessorientierte Programmierung von Steuerungsaufgaben realisieren. Dazu besteht die Ablaufsprache aus Schritten, in den Aktionen ausgeführt werden, und Transitionen, die Übergangsbedingungen zwischen einzelnen Schritten definieren.

Neben den Programmiersprachen schreibt die IEC 61131-3 außerdem eine einheitliche Definition für Datentypen, Konstanten und Variablen vor und verlangt eine explizit auszuführende Variablendeklaration. Darüber hinaus definiert die Norm ein hierarchisch gegliedertes Konzept zur Organisation von SPS-Programmen. Demnach bildet ein *Programm* (PRG) die oberste Hierarchieebene der Programmorganisation. In einigen Systemen ist nur ein

```
FUNCTION BANG_BANG_CONTROLLER : VOID
VAR_INPUT
  MEA:REAL;
  SP:REAL;
  HYS:REAL;
END_VAR
VAR_IN_OUT
  OUT:BOOL;
END_VAR

  IF MEA<(SP-HYS/2) THEN
    OUT:=TRUE
  ELSIF MEA>(SP+HYS/2) THEN
    OUT:=FALSE
  ELSE
    OUT:=OUT
  END_IF
END_FUNCTION
```

Listing 3.2: Beispiel für ein SPS-Programm in der Programmiersprache *Structured Text* gemäß IEC 61131-3 [138]

einziges Programm, das zyklische Hauptprogramm, erlaubt, von dem aus *Funktionen* (FC) und *Funktionsbausteine* (FB) aufgerufen werden. Eine Funktion wird verwendet, wenn das Ergebnis nur aus den Eingabeparametern zu berechnen ist. Dagegen besitzt ein Funktionsbaustein interne Zustandsvariablen, deren Belegung zwischen Bearbeitungszyklen erhalten bleiben. Um einen Funktionsbaustein mehrmals benutzen zu können, muss er – analog zu einem Objekt – instantiiert werden.

Listing 3.2 zeigt ein Beispiel für eine Funktion (vgl. [280, S. 363]), die einen einfachen Zweipunktregler (engl.: *bang bang controller*) in Strukturiertem Text realisiert . Abhängig von einem sich ändernden Messwert wird ein Ausgang beim Überschreiten eines Schaltwertes aus- und beim Unterschreiten eingeschaltet. Um ein schnelles Schwingen des Ausganges zu vermeiden, wird eine Hysterese berücksichtigt. Die Funktion in Listing 3.2 beginnt mit der Deklaration der Eingangs- und Ausgangsvariablen, anschließend folgt der Funktionsrumpf mit der Implementierung. Die Funktion wird zyklisch aufgerufen und vergleicht die Messwertvariable `MEA` mit dem Schaltpunkt `SP` unter Berücksichtigung der Hysterese `HYS`. Falls der Messwert den Schaltwert entsprechend übersteigt bzw. unterschreitet, wird der Ausgangswert `OUT` verändert. Andernfalls wird er gleich belassen.

IEC 61499

Die IEC 61499 stellt als internationale Norm [136] eine Erweiterung der IEC 61131 dar und beschreibt Richtlinien zur Entwicklung von verteilten Steuerungs- und Regelungssystemen [260]. Während konventionelle Automatisierungssysteme durch eine feste Hierarchie auf mehrere Steuerungen verteilt sind, spezifiziert die IEC 61499 ein flexibles Verteilungsmodell, das auf einer ereignisgesteuerten Programmausführung basiert [221]. Die Norm erweitert zudem das Konzept der Funktionsbausteine um objektorientierte Konzepte [260]. In der IEC 61499 ist der Funktionsbaustein die Basis eines Automatisierungssystem und weist im Vergleich zur IEC 61131 zusätzlich Ein- und Ausgänge für Ereignisse bzw. *Events* auf. Anders als bei einer klassischen SPS rufen sich Funktionsbausteine gegenseitig durch Events auf. Diese ereignisgesteuerte Architektur wird von Preuße et al. [221] als leichtgewichtige Komponentenlösung für Automatisierungssysteme beschrieben.

Im Gegensatz zur hierarchischen Organisation eines klassischen SPS-Programms, werden netzartige Programmstrukturen aufgebaut. Dadurch werden Funktionsbaustein ausschließlich bei Bedarf durch ein externes Ereignis aufgerufen. Dies verringert die Kommunikation zwischen verteilten Steuerungen, da nur im Ereignisfall kommuniziert werden muss. Jeder Funktionsbaustein besitzt einen Zustandsautomaten zur Beschreibung seines internen Verhaltens [260]. Innerhalb von Zuständen können Aktionen ausgeführt und Events erzeugt werden. Ein *Composite Function Block* fasst mehrere einzelne Funktionsbaustein zusammen, erscheint jedoch nach außen wie ein einfacher Funktionsbaustein. Eine weitere Gruppe von Funktionsbausteinen sind *Service Interface Function Blocks*, die im Wesentlichen zur Kommunikation mit Geräten, d. h. Aktuatoren und Sensoren, dienen. Trotz der augenscheinlichen Vorteile der IEC 61499 ist die industrielle Akzeptanz sehr gering [94] und der Nutzen wird kontrovers diskutiert (vgl. [259, 289]).

Kommerzielle Hersteller von Robotersystemen sind dazu übergegangen, eine stärkere Integration zwischen Industrierobotern und Speicherprogrammierbaren Steuerungen zu schaffen. So bieten mehrere Roboterhersteller bereits eine zusätzliche Soft-SPS für die Robotersteuerung an. Beispielsweise steht mit KUKA.PLC ProConOS [158] eine Laufzeitumgebung für die KRC zur Verfügung, die zum Ausführen von IEC 61131-3 kompatiblen Anwendungen verwendet werden kann. Dazu wird die Laufzeitumgebung unter dem Echtzeitbetriebssystem VxWorks neben der eigentlichen Robotersteuerung ausgeführt. Die Programmierung von Steuerungsaufgaben erfolgt in den Sprachen der IEC 61131-3 auf der KRC.

PLCopen

Ein weiterer Ansatz ist, Roboter direkt aus einer SPS anzusteuern. Durch die PLCopen [200], einem Zusammenschluss von Herstellern und Universitäten im Bereich der Steuerungstechnik, wurden verschiedene Spezifikationen verabschiedet, um die Entwicklung von Steuerungsapplikationen zu verbessern und stärker zu standardisieren. Diese Spezifikationen setzen alle auf der IEC 61131 auf und befassen sich u. a. mit Bewegungssteuerung. Insbesondere werden in einer Spezifikation [105] Funktionsbausteine definiert, um eine oder mehrere Achsen zu steuern. Eine weitere Spezifikation [211] definiert Funktionsbausteine, um komplexere kinematische Strukturen (z. B. einen Roboter) koordiniert im kartesischen Raum zu verfahren.

In Anlehnung an diese Spezifikationen, hat bspw. KUKA eine Bibliothek mit Funktionsbausteinen entwickelt, die elementare Bewegungsbefehle für einen Industrieroboter bereitstellen. Durch das Funktionspaket mxAutomation [157] können Roboter mithilfe dieser Funktionsbausteine vollständig von einer externen SPS gesteuert werden. Dazu sind keine Kenntnisse der Roboterprogrammierung in KRL notwendig. Neben achsspezifischen und kartesischen Bewegungen können – analog zu KRL – bahnbezogene Schaltaktionen und Interrupts definiert werden. Auch ein Überschleifen zwischen Bewegungen ist möglich.

3.4 BEWERTUNG

Der immer häufigere Einsatz eines Robotersystems als flexibler Manipulator in unterschiedlichen industriellen oder kommerziellen Prozessen führt dazu, dass ein Robotersystem nicht mehr als isoliertes System in einer abgetrennten Roboterzelle mit starren Prozessabläufen betrachtet werden kann. Durch den verstärkten Einsatz in der Kleinserienproduktion [218] und Servicerobotik [107] stoßen existierende Lösungen an ihre Grenzen. Das grundlegende Modell eines Manipulators mit einem Endeffektor, für das viele

Steuerungen ursprünglich entwickelt wurden, ist nicht mehr ausreichend. Vielmehr sind die Interaktion bzw. Kooperation mit anderen Systemen, eine flexible Handhabung und Bearbeitung verschiedenster Bauteile sowie eine intuitive Benutzerführung essentiell.

Die klassische Automatisierung von Roboterzellen basiert dagegen auf einer starren Vernetzung der Systeme gemäß der Automatisierungspyramide. Jeder Roboter wird über eine dedizierte Steuerung (vgl. Abschn. 3.1) programmiert. Die jeweiligen Aufgaben des Roboters liegen als (imperative) Programme einer proprietären Roboterprogrammiersprache vor (vgl. Abschn. 3.2). In der Regel werden Endeffektoren über einen Feldbus und digitale bzw. analoge Ein- und Ausgänge synchronisiert zum Roboterprogramme angesteuert. Über eine SPS werden die verschiedenen Robotersteuerungen koordiniert (vgl. Abschn. 3.3). Zugleich stellt die SPS über einen weiteren Feldbus die Schnittstelle zu einer übergeordneten Steuerung dar. Der große Nachteil dieses klassischen Ansatzes ist die fehlende Flexibilität, die zum Einen einer starren Vernetzung der einzelnen Systeme und zum Anderen der Art der Programmierung geschuldet ist. Vor allem die Verteilung des Wissens über die Bauteile und den Fertigungsprozess sind problematisch. In einem Roboterprogramm ist bspw. die Geometrie eines Werkstückes implizit durch die Bewegungen abgebildet. Daher ist es schwierig, diese Programme wiederzuverwenden bzw. für andere Werkstücke zu generalisieren.

Im Allgemeinen wird die Funktionalität von Robotersteuerungen den heutigen Anforderungen – insbesondere im Hinblick auf die *Servicerobotik* – nicht mehr gerecht. So sind bspw. nur wenige Bewegungsanweisungen vorhanden und eine Integration von Sensoren in die Bewegungssteuerung ist nur eingeschränkt möglich (z. B. mit dem in Abschn. 3.2 vorgestellten RSI von KUKA). Wie das Beispiel KRL zeigt, ist der Funktionsumfang von imperativen Roboterprogrammiersprachen und die Verfügbarkeit von Bibliotheken nicht vergleichbar mit objektorientierten Standardprogrammiersprachen wie Java. Dieses Defizit hat zu einer Entwicklung von *dual programmierten Anwendungen* geführt [123]. Die bedienerfreundliche Interaktion, die Kommunikation mit externen Systemen oder die Entwicklung domänenspezifischer Anwendungslogik wird vermehrt in Standardprogrammiersprachen realisiert. Dagegen werden die Programme auf der Robotersteuerung ausschließlich für die Bewegungsprogrammierung und die echtzeitfähige Kommunikation mit Werkzeugen und ggf. Sensoren eingesetzt.

Ein Beispiel für diesen Ansatz ist bei Ge und Yin [99] zu finden und beschreibt eine Anwendung mit einer graphischen Benutzeroberfläche zur Bedienung einer Plastikspritzgussanlage. Der Anlagenaufbau erfordert eine Kooperation zwischen einem Robotersystem von ABB, einem Förderband und der Spritzgussmaschine. Der überwiegende Teil der domänenspezifischen Anwendungslogik inklusive der graphischen Benutzeroberfläche wurde mit der objektorientierten Programmiersprache C# entwickelt (ca. 50.000 Zeilen Programmcode). Die Roboterprogrammiersprache RAPID wird in einem Umfang von ca. 5.000 Codezeilen zur Bewegungssteuerung verwendet. Eine weiteres Beispiel, ebenfalls in C# realisiert, wird bei Pires et al. [219] beschrieben. Hierbei wird der Roboter über eine Sprachsteuerung befehligt.

Der Ansatz dual programmierter Anwendungen weist jedoch Probleme bei Wartbarkeit bzw. Erweiterbarkeit auf, da beide Programmteile getrennt entwickelt und durch geeignete Mechanismen synchronisiert werden müssen [122]. Eine anschließende Anpassung ist nur mit hohem Aufwand möglich, da Änderungen meist in beiden Programmteilen nachgezogen werden

müssen. Aufgrund mangelnder Skalierbarkeit ist der Aufwand für Erweiterungen auf neue Aufgaben und Geräte kaum geringer als bei einer Neuentwicklung. Die potentiell echtzeitkritische Erweiterung einer Robotersteuerung ist ebenfalls aufwendig und bedingt immer auch eine Anpassung der dazugehörigen Roboterprogrammiersprache (vgl. Blomdell et al. [35]).

Ebenso ist die Portabilität von bestehenden Programmen auf andere Robotersysteme schwierig, da die Sprachen üblicherweise nur von einem einzigen Hersteller unterstützt werden und sich herstellerübergreifende Sprachen (vgl. [72]) nicht durchsetzen konnten. Die bei Offline-Programmierumgebungen oft eingesetzte Generierung von Roboterprogrammen verbessert die Portabilität von Roboterprogrammen. Durch die zusätzliche Indirektion ist es in der Regel einfacher, Teile des Programms oder zumindest Programmbeschreibungen wiederzuverwenden und andere Robotersprachen zu verwenden [210]. Nachteilig ist, dass bei Änderungen meist das ganze Programm erneut generiert und übertragen werden muss. Das bedeutet auch, dass das Programm anschließend neu getestet und eventuell neu an die reale Roboterzelle angepasst werden muss [210]. Eine systematische Rückübertragung dieser Änderungen in das, dem Programm zugrunde liegende Modell, ist meist nicht möglich.

Zusammenfassend lässt sich feststellen, dass bei der Entwicklung von Roboterprogrammen aus Sicht der Softwaretechnik Defizite sowohl bei der funktionalen Realisierung als auch im Hinblick auf qualitative Anforderungen auftreten. Diese Probleme sind vorwiegend auf die Verwendung spezieller Roboterprogrammiersprachen mit ihren spezifischen Anforderungen zur echtzeitfähigen Steuerung der Robotermechanik zurückzuführen. Allerdings können diese Defizite durch die Einführung einer geeigneten Softwarearchitektur behoben werden. Bei der Realisierung von Roboterzellen mithilfe speicherprogrammierbarer Steuerungen in Kombination mit Robotersteuerungen kann eine Verteilung des fertigungsrelevanten Wissens beobachtet werden. Dieses Wissen ist nicht gesammelt vorhanden und kann so durchgängig verwendet werden. Stattdessen sind an vielen Stellen relevante Informationen codiert, teilweise, wie die Geometrie der Bauteile, direkt in einem Programm. Dadurch ist die Wartbarkeit und Erweiterbarkeit dieser Roboterzellen stark eingeschränkt.

Teil II

ENTWICKLUNG EINES MODULAREN FRAMEWORKS FÜR ROBOTERZELLEN

Der zweite Teil der Arbeit beschreibt die Entwicklung einer neuen, echtzeitfähigen Softwarearchitektur für Industrieroboter, deren Besonderheit eine objektorientierte Programmierschnittstelle darstellt. Es wird darüber hinaus aufgezeigt, wie diese Architektur zu einem modularen Framework weiterentwickelt wird. Dadurch kann die benötigte Software, um die Roboter und Werkzeuge einer Roboterzelle zu steuern, individuell zusammengestellt und angepasst werden. Die damit einhergehende Erweiterbarkeit wird anschließend mit Beispielen illustriert.

Die Softwarearchitektur ist essentiell für die Forschungsergebnisse. Durch die Trennung von echtzeitkritischen Roboteraktionen und der Programmlogik wird der Einsatz serviceorientierter Architekturen erst möglich. Die objektorientierte Programmierschnittstelle wurde im Rahmen dieser Arbeit erweitert, um sowohl die Topologie als auch die geometrischen und logischen Zusammenhänge einer Roboterzelle abzubilden. Dieses Wissen ist die Grundlage für die serviceorientierte Strukturierung einer Roboterzelle.

4

ENTWICKLUNG EINER NEUEN SOFTWAREARCHITEKTUR FÜR INDUSTRIEROBOTER

Dieses Kapitel beschreibt das Vorgehen im Forschungsprojekt *SoftRobot*[1] eine neue Softwarearchitektur für Industrieroboter zu entwickeln. Das Projekt wurde von November 2007 bis April 2012 gemeinsam vom Institut für Software & Systems Engineering (ISSE) und den Firmen KUKA Roboter GmbH[2] und MRK-Systeme GmbH bearbeitet. Das Projekt wurde im Rahmen des Förderprogramms „Informations- und Kommunikationstechnik" von der Bayerischen Staatsregierung und der Europäischen Union unterstützt. Ziel des Forschungsprojekts war es, die Programmierung von Industrierobotern auf den aktuellen Stand der Softwaretechnik zu bringen, d. h. objektorientierte Programmiersprachen und serviceorientierte Architekturen auf die Domäne anzuwenden. Gleichzeitig sollten damit die Einschränkungen heutiger Roboterprogrammiersprachen überwunden werden (vgl. Kap. 3).

Zu Beginn des Projekts wurden gemeinsam von allen Projektpartnern die Ziele und Anforderungen für die Entwicklung einer neuen Softwarearchitektur für Industrieroboter erarbeitet. Dazu fanden eine Reihe von Treffen statt, in denen verschiedene aktuelle Anwendungen und Aufgabenbereiche von Industrierobotern diskutiert wurden. Die Ausgangslage für die Anforderungsanalyse waren die bisher verfügbaren Anwendungen und Technologiepakete der Fa. KUKA Roboter GmbH und der Fa. MRK-Systeme GmbH (vgl. Tab. 4.1 und Anhang A). Neben den bestehenden Anwendungen wurden auch aktuelle Trends der Industrierobotik bzw. der Robotik im Allgemeinen diskutiert, um die Softwarearchitektur nicht nur an aktuellen Anwendungen sondern auch an evtl. zukünftigen Anwendungen auszurichten. Dazu zählt z. B. die Realisierung sensorgeführter Roboteranwendungen bzw. die gemeinsame Programmierung kooperativer Roboter. Ein weiterer Schwerpunkt der gemeinsamen Diskussionen und Gespräche waren ebenso nicht-funktionale Anforderungen, die bei der Gestaltung der Softwarearchitektur einen zentralen Stellenwert einnehmen sollten.

Als Ergebnis dieses Austausches wurden gemeinsame Ziele benannt, die in Abschnitt 4.1 beschrieben werden. Aufbauend auf den Zielen wurden die Anforderungen an eine neue Softwarearchitektur entwickelt. Zentrale Frage war dabei, wie sich der notwendige Echtzeitbetrieb einer Robotersteuerung mit den Paradigmen moderner Programmiersprachen harmonisieren lässt. Der Lösungsansatz, d. h. die Trennung von echtzeitkritischen Roboteraktionen und dem eigentlichen Programmablauf, wird in Abschnitt 4.2 zusammen mit den entwickelten Anforderungen vorgestellt. Diese Idee ist die Grundlage für die *SoftRobot*-Architektur, die ausführlich in Abschnitt 4.3 erklärt wird. In Abschnitt 4.4 werden verwandte Arbeiten vorgestellt und von der entwickelten Architektur abgegrenzt. Zum Abschluss wird in Abschnitt 4.5 eine Zusammenfassung und kurze Bewertung der entwickelten Softwarearchitektur in Bezug auf die Ziele und Anforderungen gegeben.

1 Der vollständige Titel lautet: *SoftRobot – Eine neue Softwaregeneration für die Steuerung von Industrierobotern*. Im Weiteren wird ausschließlich die Kurzschreibweise *SoftRobot* verwendet.

2 Ab Anfang 2011 wurde das Projekt aufgrund interner Umstrukturierungen der KUKA AG von der KUKA Laboratories GmbH bearbeitet.

TECHNOLOGIEPAKET	ANWENDUNG	BESCHREIBUNG
ArchTech Digital	Schweißen	Bahnschweißen mit bis zu zwei Robotern
BendTech	Handhabung	Bearbeitung von Blechen in Abkantmaschinen
ConveyorTech	—	Synchronisation mit Fördereinrichtungen
Cooperating Robots	—	Synchronisation mehrerer Roboter
Ethernet KRL XML	—	Datenübertragung mittels TCP/IP
Ethernet RSI XML	—	Korrektur von Bewegungen, zyklische Datenübertragung
ForceControl	—	Kraftregelung eines Roboters
GlueTech	Kleben	Erstellen von Klebeapplikationen
Gripper&SpotTech	Handhabung	Programmierung von Greifern
Gripper&SpotTech	Schweißen	Erstellen von Punktschweißapplikationen
Laser RPFO	Schweißen	Bahnschweißen mit Laser der Fa. Trumpf
LaserTech	Schweißen	Erstellen von Laserschweiß-Applikationen
LaserTech	Schneiden	Erstellen von Laserschneid-Applikationen
Occubot	Testen	System zum Testen von Sitzen
PalletTech	Handhabung	Palletieren
PlastTech	Handhabung	Be- und Entladen einer Spritzgießmaschine
PLC Multiprog	—	Software-SPS nach IEC 61131-3
SeamTech SRT	Schweißen	Bauteilsuche mit einem Lichtschnittsensor
ServoGun TC	Schweißen	Programmierung von Schweißzangen
TouchSense	Schweißen	Bauteilsuche mit einem Schweißdraht

Tabelle 4.1: Alphabetische Übersicht der untersuchten Technologiepakete der Fa. KUKA Roboter GmbH

4.1 ZIELE EINER NEUEN SOFTWAREARCHITEKTUR

Einfache Realisierung von Anwendungen

Bei einer *traditionellen Roboteranwendung* wird ein Roboter durch eine feste Abfolge von Bewegungsbefehlen programmiert [63, S. 342 ff.]. Zusätzlich wird das am Roboterflansch montierte Werkzeug im Roboterprogramm an definierten Positionen angesteuert. Die Bewegungen des Roboters sind fest vorgegeben und werden nicht durch zusätzliche Sensoren nachkorrigiert. Diese Programme stellen aktuell den Großteil der vorhandenen Anwendungen dar und werden höchstwahrscheinlich auch in Zukunft noch eine entscheidende Rolle spielen. Daher darf die Erstellung solcher Anwendungen durch ein neues Steuerungsparadigma nicht erschwert werden. Idealerweise werden die bisher gebräuchlichsten Befehle in einer syntaktisch ähnlichen Form vorhanden sein, sodass eine bereits vertraute Art der Programmierung möglich bleibt.

Eine typische Anwendung dieser Kategorie von Roboterprogrammen ist das Schutzgasschweißen. Dabei ist der Roboter mit einem Schweißbrenner am seinem Flansch ausgestattet. Die Steuerung des Schweißbrenners ist über einen Feldbus mit der Robotersteuerung verbunden. Der Roboter ist so programmiert, dass er, sobald ein neues Bauteil eingelegt wurde, eine Reihe von Schweißbahnen abfährt. Die Bahnen sind fest vorgegeben und mithilfe von vorgeplanten Bewegungen programmiert. Der Startpunkt einer Bahn wird für gewöhnlich mit einer Punkt-zu-Punkt-Bewegung (PTP), die keine feste Bahn sondern nur einen Zielpunkt vorschreibt, angefahren. Die Bahn selbst ist im kartesischen Raum exakt spezifiziert und wird über Bahnbewegungen der Robotersteuerung realisiert. Am Startpunkt einer Bahn wird der Schweißbrenner aktiviert, indem über die Robotersteuerung ein Signal an die Schweißsteuerung gegeben wird. Bereits zuvor muss der Schweißvorgang zeitgesteuert vorbereitet werden, damit Schutzgas vorströmen kann. Analog wird der Schweißbrenner am Ende der Bahn deaktiviert.

Kooperative Anwendungen

Daneben stehen in Zukunft kooperative Anwendungen stärker im Fokus [109]. Dabei handelt es sich um Aufgaben, die ein einzelner Roboter nicht oder weniger effizient ausführen kann. Auch hier kann das robotergestützte Schweißen als Beispiel angeführt werden. Aktuell werden bspw. in der Automobilindustrie mehrere Roboter eingesetzt, um gleichzeitig ein Bauteil an unterschiedlichen Schweißpunkten bzw.-bahnen zu schweißen. Die Kooperation ist im Voraus genau geplant, um Kollisionen zu vermeiden und einen effizienten Ablauf zu gewährleisten. Aus Gründen der Erreichbarkeit wird teilweise ein Roboter zu Handhabung des Bauteils eingesetzt, während ein weiterer Roboter das Bauteil schweißt [108]. Dabei bewegen sich beide Roboter exakt synchron, um ein gezielte relative Bewegung des Schweißbrenners zum Bauteil zu erreichen. Eine weitere kooperative Anwendung ist der gemeinsame Transport schwerer oder sperriger Bauteile.

Bislang stellen kooperative Anwendungen trotz ihres bereits vorhandenen industriellen Einsatzes eine Ausnahme dar. Grund dafür ist die hohe Komplexität der Programmierung, die verteilt auf allen beteiligten Robotersteuerungen stattfindet [123]. Kooperative Anwendungen sind jedoch nicht nur auf mehrere Roboterarme beschränkt. Vielmehr umfassen sie eine beliebige Kombination von Roboterarmen, Lineareinheiten oder mobilen Plattformen, die zusammen programmiert werden müssen [109]. Die Kooperation dieser Geräte beschränkt sich dabei nicht nur auf Koordination, sondern beinhaltet auch immer die Synchronisation von Bewegungen. Die neue Steuerungsarchitektur muss auf kooperative Anwendungen vorbereitet sein und Programmierkonstrukte für deren Realisierung bieten.

Stärkere Integration von Sensoren

Des weiteren ist zukünftig die stärkere Integration von Sensoren ein wichtiger Aspekt [152]. Mit der neuen Softwarearchitektur muss es möglich sein, beliebige Sensoren zu integrieren und deren Datenwerte in der Echtzeitsteuerung verfügbar zu machen. Um Sensorwerte weiterzuverarbeiten und ggf. mit anderen Daten zu fusionieren, müssen eine Reihe von arithmetischen und logischen Operationen zur Verfügung stehen. Hierfür müssen weiterverarbeitete Sensorwerte in exakt definierten Zeitintervallen überwacht werden, um bei Abweichungen oder spezifizierten Ereignissen reagieren zu können. Sensoren sollen zukünftig bei der Spezifikation und Korrektur von Roboterbewegungen stärker eingebunden werden. Dabei werden Sensordaten zyklisch eingelesen und ausgewertet. Die Ergebnisse werden anschließend im Interpolationstakt zur Modifikation einer vorgeplanten Bahn verwendet.

Basierend darauf wird zwischen überwachten und geführten Bewegungen unterschieden (vgl. [85]). Bei geführten Bewegungen (engl.: *guided motion*) wird aus dem Sensorwert und einzuhaltender Randbedingungen (z.B. einer Kontaktkraft oder einem Abstand) ein Korrekturterm ermittelt und mit der vorgeplanten Bahn verrechnet. Anschließend wird die neue Positionsvorgabe an den Manipulator weitergegeben. Bei einer überwachten Bewegung (engl.: *guarded motion*) wird der Sensor nicht zur Positionsvorgabe verwendet, sondern eine (vorgeplante) Bewegung wird aufgrund eines Sensorereignisses verlangsamt oder gestoppt. Zudem kann ein Roboter, ohne dass eine geplante Bahn vorgegeben wird, über einen Sensor geführt werden. Eine besondere Rolle spielt dabei die Kraftregelung. So ist es für einige Anwendungen (z. B. dem Fügen von Bauteilen) wichtig, dass der Roboter Kontakt zu seiner Umgebung aufbaut und evtl. sogar eine definierte Kraft auf die Umgebung ausübt. Entsprechende Regelungskonzepte sind bspw. die Impedanzregelung [6], die Admittanzregelung [65] oder hybride Ansätze (vgl. [88, 174]).

Abschließend lässt sich festhalten, dass die stärkere Integration von Sensoren hohe Anforderungen an eine neue Softwarearchitektur stellen. Neben der reinen Integration des Sensors in den echtzeitfähigen Teil der Robotersteuerung, ist auch eine Weiterverarbeitung der Daten notwendig. Zudem sollte ein breites Spektrum von sensorüberwachten und -geführten Bewegungen realisiert werden können. Zusätzlich sind neue Regelungsansätze, die auf Sensoren basieren, wünschenswert.

Standardhard- und software

Die Industrierobotik ist vornehmlich davon geprägt, dass Aktuatoren, insbesondere Roboterachsen, mit harten Echtzeitanforderungen gesteuert werden müssen. Deshalb haben alle Hersteller zunächst Spezialhardware eingesetzt, um die erforderlichen Taktraten für die Interpolation der Roboterachsen zu erzielen [281]. Mitte der 1990er Jahre wagte die KUKA als erster Hersteller den Umstieg auf eine leistungsfähige PC-basierte Steuerung. Dieser Schritt wurde inzwischen von den meisten großen Herstellern nachgezogen [108]. Folglich muss sicher gestellt werden, dass auch eine neue Softwarearchitektur zur Steuerung von Industrierobotern auf PC-basierter Computerhardware läuft.

Für viele Programmiersprachen stellt die Notwendigkeit, Echtzeitanforderungen zu gewährleisten, eine große Herausforderung dar. Trotzdem ist die Verwendung einer objektorientierten Programmiersprache, ihrer Entwicklungsumgebungen und Bibliotheken ein wichtiges Ziel, das es für die neue Softwarearchitektur zu Erreichen gilt. Da insbesondere die automatische Speicherbereinigung von Programmiersprachen wie z. B. Java sich per se nicht mit Echtzeitanforderungen vereinbaren lassen [215], stellt dies eine besondere Herausforderung dar. Hinzu kommt, dass Applikationsentwickler sich – ähnlich wie bei Roboterprogrammiersprachen (vgl. Abschn. 3.2) – nicht mit der Echtzeitfähigkeit ihrer Programme beschäftigen sollten. Demnach muss eine geeignete Abstraktion geschaffen werden.

Erweiterbarkeit

Neben der reinen Anwendungsprogrammierung sollte eine neue Softwarearchitektur in Zukunft erweiterbar gestaltet werden [109]. Dazu müssen Erweiterungspunkte und Schnittstellen vorhanden sein, damit einerseits neue Geräte (d. h. Roboter, Werkzeuge und Sensoren) integriert und andererseits Lösungen für spezielle Anwendungsbereiche (z. B. Schweißen oder Palettieren) entwickelt werden können. Dazu ist es nicht ausreichend, nur die Programmierschnittstelle zu erweitern. Erweiterungen müssen auch in den echtzeitkritischen Steuerungskern integrierbar sein.

Insbesondere für die Erweiterung um neue Aktuatoren und Sensoren wird in der Regel ein Treiber auf Ebene der Echtzeitsteuerung erforderlich sein. Die Funktionalität soll gleichzeitig durch elementare Steuerungsbefehle in der Anwendung nutzbar sein. Die Änderungen sollen sich aber auf die Einbindung des Treibers und einer Programmierschnittstelle auf Anwendungsebene beschränken. Dazwischen muss ein generischer Kommunikationsweg existieren, der für neue Komponenten genutzt werden kann. So will bspw. ein Hersteller von Schweißbrennern ein neues Modell integrieren. Da das Modell spezielle Funktionen unterstützt, ist ein neuer Treiber notwendig. Daneben entwickelt der Hersteller eine eigene Programmierschnittstelle, um die Steuerungsbefehle für Applikationsentwickler nutzbar zu machen. Solche Erweiterungen müssen ohne einschneidende Änderungen an der Gesamtarchitektur möglich sein.

Wiederverwendung

Für eine zeit- und kosteneffiziente Entwicklung von Technologiepaketen und Roboteranwendungen ist ein hohes Maß an Wiederverwendung notwendig [122]. Dies betrifft zum einen die Integration neuer Roboter, Werkzeuge und Sensoren, die mit wenig Aufwand und einem hohen Grad an

Wiederverwendung erfolgen soll. Zum anderen sollten auch neue Technologiepakete, d. h. spezifische Lösungen für bestimmte Anwendungsbereiche, allgemein und wiederverwendbar gestaltet werden.

Programmierer sollen durch die neue Softwarearchitektur die Möglichkeit haben, Anwendungen losgelöst von einer spezifischen Instanz eines Roboters oder Werkzeugs zu entwickeln. Dadurch können Anwendungen offline, d. h. ohne Zugang zu einer konkreten Steuerung, und wiederverwendbar entwickelt werden. Für den Anwendungsentwickler soll es möglich sein, die Roboter und Werkzeuge nur anhand von bestimmten Eigenschaften oder Klassifikationen auszuwählen, z. B. 6-Achs-Roboter mit bestimmter Reichweite, Parallelgreifer mit definierter Öffnungsweite oder einen entsprechenden Schweißbrenner.

Für Schutzgasschweißen kann bspw. ein abstrakter Schweißbrenner modelliert werden. Der abstrakte Schweißbrenner besitzt die kleinste gemeinsame Menge an Attributen und geometrischen Merkmalen. Gleichzeitig können abstrakte Operationen des Schweißvorgangs mit den entsprechenden Prozessparametern definiert werden (z. B. Vorströmen des Schutzgases, Aktivieren des Lichtbogens). Konkrete Schweißbrenner müssen die abstrakte Definition eines Schweißbrenners implementieren und ggf. erweitern. Zusätzlich müssen sie die spezifizierten Operationen implementieren.

Ein Anwendungsentwickler kann nun ein Programm zur Steuerung einer Schweißzelle erstellen, deren finale Ausstattung bzgl. Roboter und Werkzeug bspw. nicht endgültig feststeht. Er entwickelt die Anwendung daher nur für einen generischen bzw. abstrakten Roboter und Schweißbrenner. Erst bei der Inbetriebnahme werden die abstrakten Geräte an die vorhandenen Instanzen gebunden. Diese als Polymorphie [97] bezeichnete Austauschbarkeit kann zu einer höheren Wiederverwendbarkeit von Roboterprogrammen führen.

4.2 ENTWICKLUNG DER ANFORDERUNGEN

Um die Domäne der Industrierobotik und speziell die vielfältigen Applikationen zu verstehen, wurden ca. 20 der momentan vertriebenen Technologiepakete der KUKA Roboter GmbH (vgl. Tab. 4.1) mit Methoden der Softwaretechnik analysiert [12, 122]. Ein Technologiepaket ist ein wiederverwendbares Softwareprodukt, das für eine bestimmte Anwendung Funktionalität auf der Robotersteuerung bereitstellt. Die untersuchten Technologiepakete lassen sich in zwei Kategorien unterteilen. Durch Technologiepakete der ersten Kategorie werden konkrete industrielle Applikationen realisiert. Dazu zählen bspw. Bahn- und Punktschweißen oder Kleben. Außerdem gibt es Technologiepakete, die Querschnittsfunktionen bereitstellen, d. h. Funktionen die in unterschiedlichen Applikationen benötigt werden (z. B. die Synchronisation mit Förderbändern oder die externe Korrektur von Bewegungen). Neben den bereits bestehenden Technologiepaketen wurden außerdem zukünftige Anwendungsszenarien und deren technologische Herausforderungen berücksichtigt (vgl. [32, 109]).

Bei der softwaretechnischen Analyse wurden sowohl funktionale als auch nicht-funktionale Anforderungen betrachtet. Außerdem wurden mehrere Akteure identifiziert und deren Interaktion mit dem Robotersystem untersucht. Ein besonderes Augenmerk wurde auf die Echtzeitbedingungen und einen deterministischen Programmablauf gelegt. Wie in Kapitel 3 beschrieben wurde, sind Robotersteuerungen zwingend Echtzeitsysteme mit sehr kurzen Berechnungszyklen von bis zu 1 ms. Da diese Zeitschranken ein-

TECHNOLOGIEPAKET	ANWENDUNG	MOTION	SENSOR	SYNC	EVENT	TOOL
ArchTech Digital	Schweißen	X		X	X	X
BendTech	Handhabung	X		X	X	X
ConveyorTech	—	X		X	X	
Cooperating Robots	—	X		X	X	
Ethernet KRL XML	—				X	
Ethernet RSI XML	—		X	X		
ForceControl	—		X		X	
GlueTech	Kleben	X			X	X
Gripper&SpotTech	Handhabung	X			X	X
Gripper&SpotTech	Schweißen	X			X	X
Laser RPFO	Schweißen	X			X	X
LaserTech	Schweißen	X			X	X
LaserTech	Schneiden	X			X	X
Occubot	Testen	X	X		X	
PalletTech	Handhabung	X	X	X	X	X
PlastTech	Handhabung	X		X	X	
PLC Multiprog	—					
SeamTech	Schweißen	X	X		X	
ServoGun TC	Schweißen	X		X	X	X
TouchSense	Schweißen	X	X		X	

Tabelle 4.2: Übersicht der identifizierten Echtzeitmuster bei den untersuchten Technologiepaketen aus Tabelle 4.1. Technologiepakete ohne Anwendungsbereich beschreiben Querschnittsfunktionen.

gehalten werden müssen und das Gegenteil katastrophale Auswirkungen haben kann, zählen Robotersteuerungen zu den *harten Echtzeitsystemen*. Für die industrielle Robotik ist *Determinismus* von Roboteroperationen ein wichtiger Aspekt, d. h. ein Roboter führt eine gegebene Bewegung unter den gleichen äußeren Randbedingungen identisch aus.

Real-time Patterns

Um die funktionalen Anforderungen zu identifizieren, wurden die Technologiepakete nach wiederkehrenden Grundfunktionen und gemeinsamen Mustern untersucht. Typische Grundfunktionen sind bspw. „Bewegung mit Werkzeugschaltaktionen“ oder „Bewegung mit Abbruchbedingung“. Viele Grundfunktionen weisen eine ähnliche Charakteristik auf und konnten auf wenige Muster (engl.: *pattern*) reduziert werden. Ein *Pattern* beschreibt ein in einem Kontext „immer wieder auftretendes Problem, beschreibt dann den Kern der Lösung dieses Problems, und zwar so, daß man diese Lösung millionenfach anwenden kann“ [7, S. X]. Da der hier vorliegende Kontext die echtzeitfähige Steuerung von Robotern ist, konnten folgende vier Echtzeitmuster identifiziert werden, die in Tabelle 4.2 aufgelistet sind:

1. Das *MOTION-Pattern* besagt, dass sowohl im Achsraum als auch kartesisch spezifizierte und geplante Bewegungen präzise und reproduzierbar ausgeführt werden. Zielpunkte von Bewegungen müssen mit hoher Genauigkeit immer wieder erreicht werden. Dabei müssen die gegebenen Randbedingungen der Bewegungen, d. h. die Bahn, das Fahrprofil sowie die maximale Geschwindigkeit und Beschleunigung des Roboters, flexibel definierbar sein.

2. Das *SENSOR-Pattern* besagt, dass der Roboter durch einen oder mehrere Sensoren geführt wird. Das setzt voraus, dass die Datenwerte von Sensoren unter Echtzeitbedingungen bearbeitet, gefiltert und ggf. fusioniert werden können. Die Führung durch den Sensor kann sowohl

die Korrektur einer vorgeplanten Bewegungsbahn bedeuten als auch die *reine* Steuerung durch den Sensor. Hierzu müssen neben der Positionssteuerung weitere Arten der Bewegungssteuerung möglich sein.

3. Das *SYNC-Pattern* besagt, dass räumliche Synchronisationsbeziehungen zwischen unterschiedlichen Entitäten (d. h. Roboter, Werkzeuge, Sensoren und Bauteile) beschrieben werden, die während der Ausführung von Bewegungen aufrechterhalten werden. Dabei können beliebige Koordinatensysteme auf diesen Entitäten definiert werden. Sobald eine der verbundenen Entitäten bewegt wird, folgen die anderen Entitäten automatisch. Eine Konsistenzprüfung sollte nach Möglichkeit vor der Ausführung einer synchronisierten Bewegung erfolgen.

4. Das *EVENT-Pattern* besagt, dass Ereignisse spezifiziert werden, deren Eintreten periodisch unter Echtzeitbedingungen geprüft wird und die bestimmte Reaktionen auslösen können. Ereignisse können dabei zeitlich, räumlich, auf Sensoren bzw. Aktuatoren oder spezifisch für eine Operation definiert werden. Mögliche Reaktionen sind das Starten einer neuen bzw. das Abbrechen einer laufenden Aktion.

Daneben existiert noch ein weiteres wiederkehrendes Muster. Dabei handelt es sich um die Definition neuer Werkzeugaktionen, gekennzeichnet als *TOOL-Pattern* (vgl. Tab. 4.2). Dies ist bei anwendungsspezifischen Technologiepaketen von großer Bedeutung, da sich diese hauptsächlich über ein Werkzeug und dessen Aktionen identifizieren. Die Definition einer neuen Werkzeugaktion kann jedoch mit einem der vier anderen Muster – meistens mit dem *EVENT-Pattern* – abgebildet werden.

Kernidee der Architektur

Die Identifikation der Echtzeitmuster zeigt, dass ein Roboterprogramm nicht vollständig unter Echtzeitbedingungen ausgeführt werden muss. Zwar hat jede Operation, die unter eines der Echtzeitmuster fällt, hohe Echtzeitanforderungen, jedoch beinhalten Roboterprogramme einen großen Teil nichtechtzeitkritischer Ablauflogik. Diese Feststellung ist das zentrale Ergebnis der Anforderungsanalyse und hat die gesamte Softwarearchitektur von *SoftRobot* maßgeblich beeinflusst [118]. Eine echtzeitfähige Steuerung wird nur benötigt, um jede einzelne Bewegung präzise und deterministisch zu steuern und Werkzeugaktionen exakt während Bewegungen zu schalten. Bewegungen und Werkzeugaktionen können jedoch in eine *echtzeitkritische Transaktion* zusammengefasst und auf einem entsprechenden Betriebssystem als Einheit ausgeführt werden. Eventuell auftretende Verzögerungen zwischen Transaktionen sind unproblematisch, da sie die Semantik des Roboterprogramms nicht verändern [118]. Die Anwendungslogik kann unter einem Standardbetriebssystem ohne spezielle Anforderungen ausgeführt werden.

Abbildung 4.1 zeigt schematisch die Aufteilung eines Roboterprogramms zwischen der Ablauf- und Echtzeitsteuerung anhand eines (vereinfachten) Beispiels als Sequenzdiagramm. Die Applikation repräsentiert dabei die Ablaufsteuerung und schickt den Befehl, eine Schweißnaht (engl.: *welding seam*) abzufahren und zu schweißen, an die echtzeitfähige Robotersteuerung. Dieser Befehl muss vollständig an die Robotersteuerung übertragen werden, damit diese ihn auszuführen kann. Die Applikation ist an der weiteren Ausführung nicht beteiligt und kann, wie in Abb. 4.1, bis zum Ende des Roboterbefehls blockieren (*synchrone Ausführung*) oder den Ablauf fortsetzen (*asynchrone Ausführung*). Die Robotersteuerung dagegen führt die Bewegung aus, d. h. sie generiert in jedem Interpolationstakt neue Stützpunkte (engl.: *set points*), die der Roboter erreichen muss. Dies entspricht dem *MOTION-Pattern*. Neben der Interpolation der Bewegung muss die Robotersteuerung

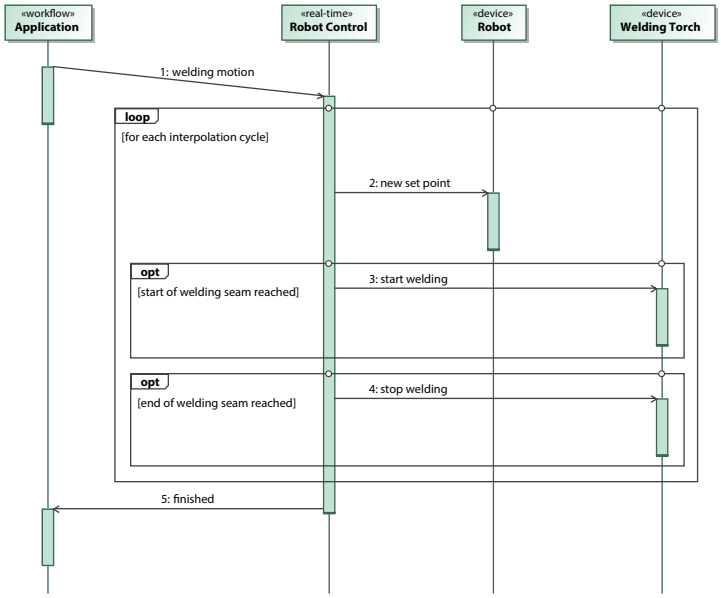


Abbildung 4.1: Aufteilung einer Roboteraufgabe zwischen Ablaufsteuerung (*workflow*) und Echtzeitsteuerung (*real-time*).

in jedem Takt prüfen, ob die Schweißnaht bereits erreicht wurde. Falls dies zutrifft, wird der Schweißbrenner aktiviert[3]. Analog erfolgt die Deaktivierung des Schweißbrenners beim Erreichen des Nahtendes. Das Aktivieren und Deaktivieren des Schweißbrenners entspricht dem *EVENT-Pattern*.

Die Trennung zwischen der echtzeitkritischen Steuerung von Aktuatoren und der Programmierung der Anwendungslogik war die Grundlage für den Entwurf der Softwarearchitektur in *SoftRobot*. Durch diese Trennung ergeben sich gemäß [118] folgende Konsequenzen für die Softwarearchitektur:

- Roboteranwendungen können in einer universellen Programmiersprache wie Java entwickelt werden, die durch Merkmale, wie z. B. eine automatische Speicherverwaltung, nur eingeschränkt echtzeitfähig sind. Außerdem können Anwendungen auf Standardbetriebssystemen (z. B. Windows, Linux) und unter Verwendung integrierter Entwicklungsumgebungen (z. B. Eclipse) implementiert werden.
- Durch das Konzept *echtzeitkritischer Transaktionen* kann von der Steuerungsebene, die eine echtzeitfähige Programmierung voraussetzt, abstrahiert werden. Stattdessen kann grundlegende Funktionalität eines industriellen Robotersystems über eine wohldefinierte Programmierschnittstelle zur Verfügung gestellt werden, die von einem Anwendungsentwickler benutzt werden kann.

Dies ermöglicht es, die Bewegungssteuerung von Industrierobotern direkt in die Anwendungsentwicklung zu integrieren. Eine Programmierschnittstelle, d. h. ein Application Programming Interface, abstrahiert von der Komplexität der darunterliegenden Robotersteuerung und ermöglicht es dem Applikationsentwickler, sich auf die Implementierung der eigentlichen Aufgabe zu fokussieren. Folglich muss der Entwickler nicht Sorge dafür tragen, dass sein Programm (harte) Echtzeitbedingungen einhält. Er muss sich nur

3 In Realität ist dieser Vorgang z. B. aufgrund des Vorströmens eines Schutzgases komplexer. Um die Klarheit des Beispiels nicht zu beeinträchtigen, wurde darauf verzichtet.

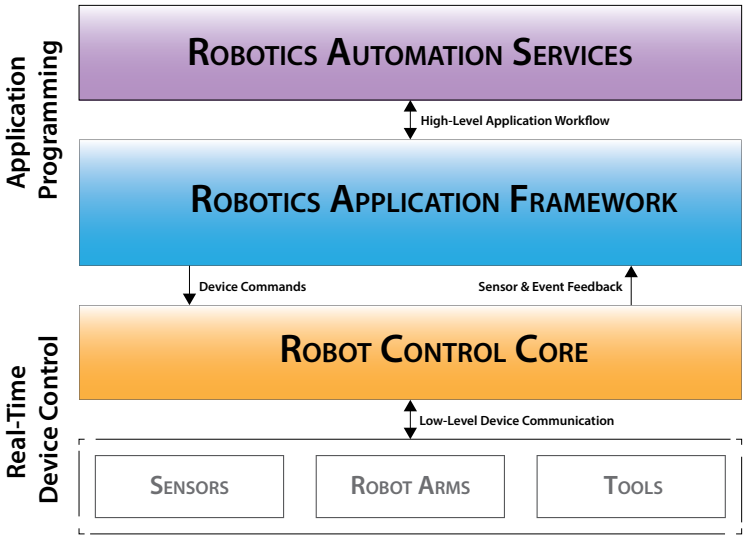


Abbildung 4.2: Überblick über die *SoftRobot*-Architektur, die sich in Applikationsprogrammierung und Echtzeitsteuerung untergliedert.

bewusst sein, welche Operationen zeitkritisch sind und diese gemeinsam als *echtzeitkritische Transaktion* ausführen lassen. Damit findet eine Verschiebung von der Echtzeitprogrammierung hin zu einem *echtzeitbewussten Programmieren* (engl.: *real-time aware programming*) statt.

Die Programmierschnittstelle wurde objektorientiert konzipiert, um die Domäne der Industrierobotik und Automatisierung durch ein (abstraktes) Modell abzubilden [122]. Dessen Objekte (z. B. Roboter, Werkzeuge) können verwendet werden, um eine Aufgabe zuerst zu beschreiben und anschließend zu implementieren. Das objektorientiert Modell wird ausführlich in Kapitel 5 vorgestellt. Für den Entwurf der Programmierschnittstelle wurden auf Basis der bisherigen Analyse Szenarien erarbeitet. Ein Szenario beschreibt eine Interaktion mit dem System unter Berücksichtigung der bereits identifizierten Muster, um die zentrale Konzepte der Industrierobotik herauszuarbeiten. Die daraus abgeleiteten funktionalen Anforderungen an die Programmierschnittstelle sind in [10] ausführlich beschrieben. Die durch die Szenarien identifizierten Konzepte wurden kategorisiert und anschließend iterativ in ein Konzeptmodell überführt wurden. Bei jeder Iteration wurden weitere Konzepte aufgenommen und bereits bestehende Konzepte verfeinert. Zur Modellierung wurden dabei die Unified Modeling Language (UML) [197] verwendet. Anschließend wurde das Konzeptmodell schrittweise in ein Entwurfsmodell transferiert und implementiert.

4.3 DIE *softrobot*-ARCHITEKTUR ALS ERGEBNIS

Um die gewünschte Abstraktion von der Echtzeitprogrammierung zu gewährleisten, besteht die *SoftRobot*-Architektur aus unterschiedlichen Ebenen, die in Abbildung 4.2 dargestellt und erstmals in [118] vorgestellt wurden. Zentrales Element der Architektur ist das *Robotics Application Framework*, das ein Application Programming Interface (API), d. h. eine Programmierschnittstelle, für die Entwicklung von Applikationen bereitstellt. Darauf aufbauend können unterschiedliche Arten von Applikationen realisiert werden [18]. Jedoch liegt in dieser Arbeit der Schwerpunkt hauptsächlich auf serviceorientierten Applikationen für Roboterzellen (vgl. Kap. 7).

Die echtzeitfähige Steuerung der Roboter bzw. Aktuatoren erfolgt über den *Robot Control Core*. Dieser Bestandteil der Architektur muss zwingend auf einem Echtzeitbetriebssystem [284] implementiert sein, um eine stabile und zuverlässige Kommunikation mit den Sensoren und Aktuatoren bzw. deren Steuerungen zu gewährleisten. Die Echtzeitanforderungen sind, wie in Abschnitt 4.2 erklärt, wichtig, um in einem definierten Takt deterministisch neue Sollwerte an einen Aktuator zu kommunizieren. Daher delegiert das *Robotics Application Framework* alle Kommandos zum Auslesen von Sensoren bzw. zur Steuerung von Aktuatoren an den *Robot Control Core*. Dieser wiederum benachrichtigt die Applikationsebene über Ereignisse während der Kommandoausführung und kann Sensorwerte dorthin propagieren.

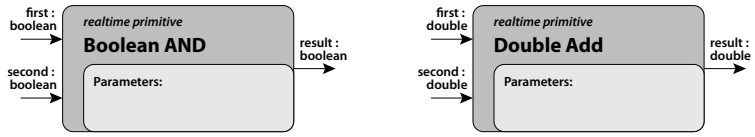
Im folgenden werden die unteren beiden Ebenen der Architektur beschrieben und deren Zusammenspiel wird abstrakt erklärt. Der *Robot Control Core* und seine Schnittstelle zur Applikationsebene wird zuerst in Abschnitt 4.3.1 beschrieben. Anschließend wird in Abschnitt 4.3.2 näher auf das *Robotics Application Framework* und dessen internen Aufbau eingegangen. Dort wird das Zusammenspiel zwischen beiden Ebenen erklärt. Der modulare Aufbau des *Robotics Application Frameworks* mittels OSGi wird später in Kapitel 6 behandelt. Dort wird nochmals genauer auf den Zusammenhang der beiden Ebenen und ihrer Abhängigkeiten eingegangen. Der Aufbau der *Robotics Automation Services* wird im dritten Teil dieser Arbeit ab Kapitel 7 schwerpunktmäßig vorgestellt.

4.3.1 *Robot Control Core*

Für die echtzeitkritische Steuerung ist der Robot Control Core (RCC) zuständig, der über eine eigens entwickelte Schnittstelle – das Realtime Primitive Interface (RPI) – angesprochen wird [118]. RPI erlaubt eine generische Spezifikation von echtzeitkritischen Aufgaben für die Industrierobotik mithilfe einer Datenflusssprache [274], die ähnlich zur synchronen Datenflusssprache LUSTRE [110] ist. Zudem ist eine Semantik definiert, wie mit RPI spezifizierte Aufgaben als Kommando periodisch in einem festen Zeitraster ausgeführt werden [272]. Die Ausführungssemantik kann von einer kompatiblen Robotersteuerung, d. h. einem RCC, unter Berücksichtigung von Echtzeitanforderungen implementiert werden. Somit können in RPI beschriebene Kommandos an einen RCC übertragen und dort unter harten Echtzeitbedingungen ausgeführt werden. Im Rahmen von *SoftRobot* wurde eine Referenzimplementierung [272] entwickelt, die ebenfalls kurz vorgestellt wird.

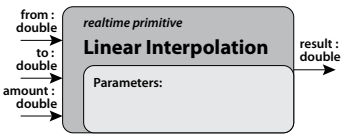
Die Idee echtzeitkritische Programme mithilfe einer synchronen Datenflusssprache zu beschreiben ist in der Literatur oft zu finden (vgl. z. B. [29]). Die Besonderheit an RPI jedoch ist, dass es einen flexiblen Ablauf ermöglicht. Das bedeutet, dass sich zur Laufzeit dynamisch neue Datenflussprogramme bilden, die unter Echtzeitbedingungen ausgeführt werden. Zudem können diese Programme ohne Unterbrechung, d. h. von einem Ausführungstakt auf den nächsten, von neuen Programmen abgelöst werden [273]. Folglich steht mit RPI eine sehr flexible und mächtige Beschreibungssprache zur Verfügung, die die Anforderungen der Industrierobotik erfüllen kann.

Der elementare Baustein in RPI ist ein *Realtime Primitive* bzw. Echtzeitprimitiv, das einen atomaren Funktionsblock mit Ein- und Ausgängen repräsentiert. Es stellt formal eine Abbildung einer Menge von n typisierten Eingabewerten auf eine Menge von m typisierten Ausgabewerten mit $n, m \in \mathbb{N}_0$ dar [274]. Zudem kann ein *Realtime Primitive* über Parameter konfiguriert werden. Durch ein *Realtime Primitive* kann unterschiedliche Funk-

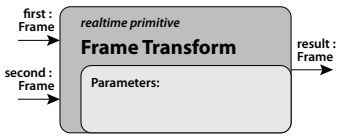


(a) Boolesches AND

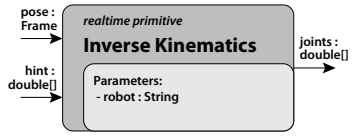
(b) Addition von zwei Gleitkommazahlen



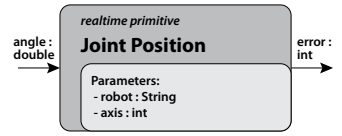
(c) Lin. Interpolation zwischen zwei Gleitkommazahlen



(d) Transformation von kartesischen Koordinaten



(e) Berechnung der inversen Kinematik eines Roboters



(f) Steuerung einer Roboterachse

Abbildung 4.3: Beispiele für *Realtime Primitives*: Die Pfeile auf der linken Seite repräsentieren die Eingänge, auf der rechen Seite die Ausgänge.

Beispiel: Realtime Primitives

tionalität gekapselt werden (vgl. Abb. 4.3). Zum einen können Primitive logische Operatoren oder arithmetische Funktionen repräsentieren. Zum anderen sind komplexe Primitive möglich, die z. B. Koordinatensysteme transformieren oder Trajektorien berechnen. Aktuatoren oder Sensoren werden ebenfalls durch *Realtime Primitives* repräsentiert, wobei Echtzeitprimitive für Sensoren nur Ausgänge und für Aktuatoren nur Eingänge[4] besitzen. Typisches Beispiel für einen Aktuator ist die achsspezifische Steuerung eines Robotergelenks (vgl. Abb. 4.3 (f)). Eine mit RPI kompatible Robotersteuerung stellt eine Menge an implementierten *Realtime Primitives* bereit, die instanziiert und verwendet werden können.

Um die Ein- und Ausgänge mehrerer Instanzen von *Realtime Primitives* miteinander zu verbinden, existiert in RPI das Konzept eines *Links*, der einen Ausgang einer *Realtime Primitive* Instanz mit einem Eingang einer anderen Instanz verbindet. Ein Ausgang kann über mehrere *Links* mit mehreren Eingängen verbunden werden, während ein Eingang nur mit einem Link verbunden werden kann. Da Ein- und Ausgänge statisch typisiert sind, können nur *Links* zwischen Ein- und Ausgängen gleichen Typs definiert werden. Neben typischen Datentypen (d. h. Boolean, Integer, Double und String) und Arrays existieren auch komplexe Robotik-spezifische Datentypen wie z.B. dreidimensionale Koordinaten.

Instanzen von *Realtime Primitives* und *Links* zwischen diesen Instanzen bilden einen Graph, der als *Fragment* bezeichnet wird. Formal besteht ein *Fragment* aus einer Menge P von *Realtime Primitives* und einer Menge L von *Links*, die zwischen Ein- und Ausgängen der Elemente von P verbunden sind. Nach außen verhält sich ein *Fragment* selbst wie ein *Realtime Primitive*, wobei die nicht verbundenen Ein- und Ausgänge aller Elemente von P die Ein- und Ausgänge des *Fragment* definieren. Ein *Fragment* kann wie

4 Eine Ausnahme stellen spezielle Ausgänge dar, die Fehlerzustände eines Aktuator anzeigen (vgl. [272]).

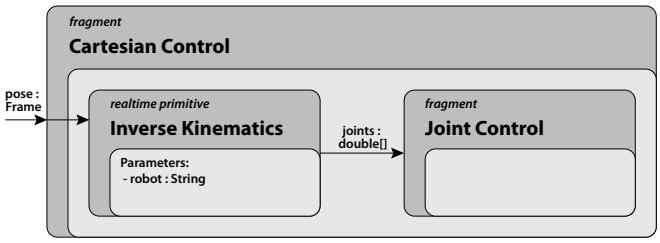


Abbildung 4.4: Die kartesische Steuerung eines Roboter ist ein *Fragment*, das auf dem *Fragment* zur achsspezifische Steuerung basiert.

ein *Realtime Primitive* verwendet werden und dementsprechend mit anderen Echtzeitprimitiven oder Fragmenten verbunden werden.

Beispiel: Fragments

Wie in Abbildung 4.4 in einem reduzierten Beispiel zu erkennen ist, ermöglichen Fragmente eine hierarchische Gliederung in RPI. Die achsspezifische Steuerung eines Roboters ist als Fragment (*Joint Control*) realisiert, das intern aus mehreren Echtzeitprimitiven zur Steuerung der einzelnen Achsen aufgebaut ist (vgl. Abb. 4.3 (f)). Die kartesische Steuerung des Roboters ist ebenfalls als Fragment (*Cartesian Control*) realisiert und besteht aus dem oben genannten Fragment und einem Echtzeitprimitiv zur Berechnung der inversen Kinematik des Roboters (vgl. Abb. 4.3 (e)). Die inverse Kinematik ermöglicht die Bestimmung der Gelenkwinkel eines Roboters anhand einer gegebenen Roboterpose, d. h. der Position und Orientierung des Flansches.

In RPI wird ein ausführbares Kommando als *Realtime Primitives Net* bezeichnet und ist als spezielles Fragment (*root fragment*) spezifiziert, das mit einem offenen booleschen Ausgang das Ende der Kommandoausführung anzeigt. Daneben darf dieses Fragment keine weiteren offenen Ein- bzw. Ausgänge aufweisen. Die Ausführung erfolgt periodisch, d. h. in jedem Ausführungstakt wird der ganze Graph einmal ausgewertet. Da der Graph per Definition immer azyklisch sein muss, können die Echtzeitprimitive topologisch sortiert und die Auswertung linearisiert werden, sodass es eine optimale sequentielle Ausführungsordnung der Echtzeitprimitive gibt [274].

Beispiel: Realtime Primitives Net

Ein vereinfacht dargestelltes Beispiel für ein *Realtime Primitives Net* ist in Abbildung 4.5 gegeben. Dabei wird ein Roboter entlang einer Trajektorie, die im kartesischen Raum definiert ist, geführt. Das Kommando besteht aus drei Fragmenten. Im ersten Fragment wird die Trajektorie schrittweise als dreidimensionale Koordinate des Endeffektors berechnet und durch das zweite Fragment in eine entsprechende Roboterpose transformiert. Schließlich wird das Fragment aus Abbildung 4.4 zur kartesischen Steuerung des Roboter verwendet. Der offene Ausgang zeigt die Terminierung des Kommandos an, sobald die Trajektorie vollständig abgefahren wurde.

Ein *Realtime Primitives Net* kann, sofern alle notwendigen Echtzeitprimitive vorhanden sind, von einer kompatiblen Robotersteuerung, einem RCC, ausgeführt werden. Während der Ausführung hat jedes *Realtime Primitives Net* einen eigenen Lebenszyklus [272]. Nachdem ein *Realtime Primitives Net* an einen RCC übertragen wurde, wird das Kommando geladen. Falls die benötigten Echtzeitprimitive vorhanden sind, werden diese für das *Realtime Primitive* instantiiert und das Kommando wird zur Ausführung eingeplant. Es kann gestartet werden, sobald die benötigten Aktuatoren erfolgreich allokiert wurden. Das *Realtime Primitives Net* und alle darin enthaltenen Echtzeitprimitive werden zyklisch (d. h. in einem Takt von bis zu 1 kHz) immer wieder ausgeführt.

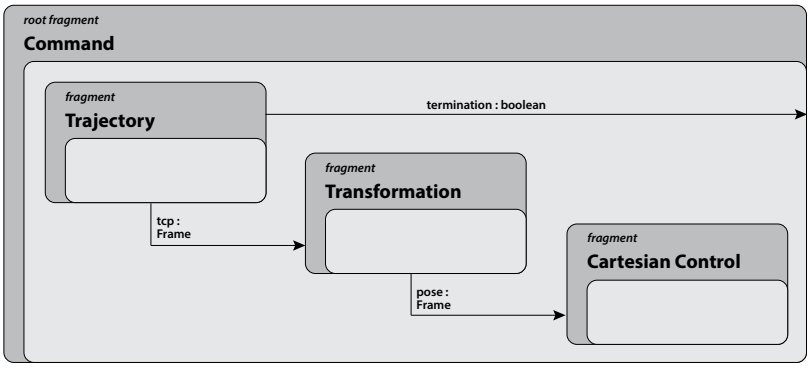


Abbildung 4.5: Der Endeffektor des Roboters wird in diesem *Realtime Primitives Net* entlang einer kartesischen Trajektorie geführt.

Für die Ausführung eines einzelnen Echtzeitprimitivs werden zuerst die Eingänge gelesen, dann die internen Aktionen durchgeführt und abschließend die Ausgänge geschrieben. Wenn die Ausführung terminiert, werden die verwendeten Aktuatoren freigegeben und können in folgenden Kommandos wieder verwendet werden. Die Terminierung wird, wie in Abbildung 4.5 dargestellt, über einen speziellen Ausgang des Kommandofragments angezeigt. Zudem existiert ein spezielles Echtzeitprimitiv (*Cancel*), das signalisiert, ob ein Kommando von Extern abgebrochen werden soll. Ein Kommando muss darauf entsprechend reagieren, z. B. eine laufende Bewegung abbrechen und den Roboter definiert abbremsen.

In der Regel muss ein *Realtime Primitives Net* so definiert sein, das es in sich abgeschlossen ist [274]. Das bedeutet, dass sich das System vor und nach dessen Ausführung in einem stabilen und konsistenten Zustand befindet; ein Roboterarm darf zum Beispiel nicht mehr in Bewegung sein. Allerdings erlaubt RPI die instantane Übernahme von Kommandos [273]. Damit kann zwischen zwei Takten von einem Kommando auf ein nachfolgendes Kommando umgeschaltet werden. Da das System beim Beenden des ersten Kommandos noch nicht in einem stabilen Zustand sein muss, muss das zweite Kommando das System in diesem Zustand übernehmen können. Durch diesem Mechanismus können Bewegungen nahtlos über mehrere Kommandos verteilt werden, d. h. der Aktuator muss zwischen den Bewegungen nicht anhalten.

Details zur Implementierung

Innerhalb des Projekts *SoftRobot* wurde eine Referenzimplementierung für einen Robot Control Core in C/C++ umgesetzt. Dieser kann mit RPI spezifizierte Kommandos laden und wie oben beschrieben unter harten Echtzeitbedingungen ausführen. Er stellt die echtzeitkritische Ebene der Architektur dar und ist dementsprechend performant und deterministisch auf einem Echtzeitbetriebssystem [284] implementiert. Dazu basiert die Referenzimplementierung auf Komponenten des OROCOS Frameworks [50]. So wird das Real-Time Toolkit (RTT) [201] als Abstraktionsschicht für ein konkretes Echtzeitbetriebssystem verwendet. Aus der Kinematics and Dynamics Library (KDL) [202] werden Robotik-spezifische Datentypen und Algorithmen verwendet. Aktuell läuft die Referenzimplementierung unter Xenomai [286], einer echtzeitfähigen Erweiterung für Linux. Daneben kann der RCC für Test- und Simulationszwecke auch unter Windows laufen.

Allerdings ist die Implementierung eines RCC nicht an OROCOS gebunden, sondern allgemein definiert als Ausführungsumgebung von echtzeitkritischen Aufgaben, die mit RPI spezifiziert sind. Darüber hinaus bestehen

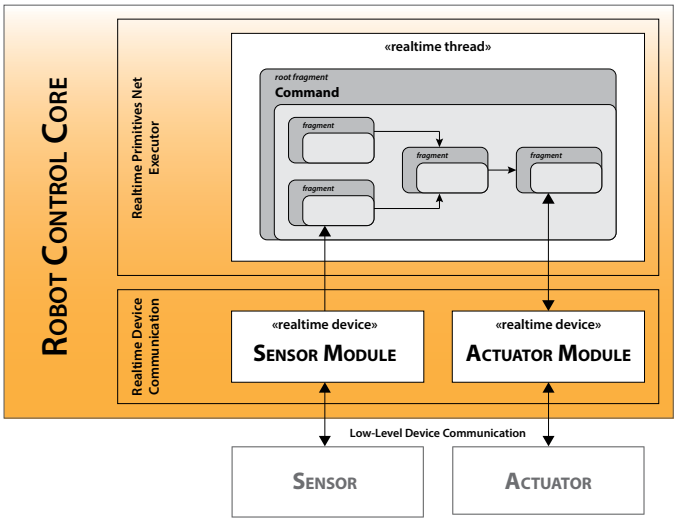


Abbildung 4.6: Über ein Echtzeitgerätemodule wird zwischen dem eigentlichen Gerät und einem *Realtime Primitives Net* kommuniziert.

keine Vorgaben bzgl. des Kommunikationsprotokolls und der vorhandenen *Realtime Primitives* (vgl. Kap. 6). Um neue Aktuatoren, Sensoren bzw. Steuerungskonzepte zu integrieren, ist die Menge an verfügbaren *Realtime Primitives* erweiterbar. Standardmäßig stehen nur Echtzeitprimitive zur Verfügung, die den Lebenszyklus eines Kommandos beeinflussen. Weitere Echtzeitprimitive sind als Erweiterungen konzipiert (vgl. Kap. 6). Die Referenzimplementierung, die im weiteren Verlauf gleichbedeutend als RCC bezeichnet wird, ist ausführlich in [271] beschrieben.

Dessen interne Struktur ist schematisch in Abbildung 4.6 dargestellt. Darin ist eine Laufzeitumgebung für *Realtime Primitives Nets* enthalten, die eine simultane Ausführung mehrerer Kommandos erlaubt [272]. Dazu wird jedes Kommando in einem eigenen Echtzeitthread [284] ausgeführt. Voraussetzung für parallel laufende Kommandos ist, dass disjunkte Mengen von Aktuatoren gesteuert werden. Für die echtzeitfähige Kommunikation mit Sensoren und Aktuatoren sieht die Referenzimplementierung spezielle Module vor (vgl. Abb. 4.6), die als *Realtime Device* bezeichnet werden. Ein solches Modul wird applikationsspezifisch konfiguriert und geladen und ist für die Kommunikation mit dem eigentlichen Gerät bzw. dessen Steuerung zuständig. Diese Kommunikation kann z. B. über einen industriellen Feldbus (z. B. EtherCAT [137], CAN [140]), Ethernet oder sogar USB erfolgen. Das Kommunikationsprotokoll muss innerhalb des Moduls implementiert werden und kann entweder standardisiert (z. B. CANopen [135]) oder proprietär (z. B. FRI [240]) sein.

Ein *Realtime Device* kommuniziert über spezielle *Realtime Primitives* mit einem laufenden Kommando (vgl. Abb. 4.6). Die Werte eines Sensors werden über ein entsprechendes *Realtime Device* Modul eingelesen und bei Bedarf an ein oder mehrere Echtzeitprimitive weitergegeben. Die Sensorwerte können dadurch in mehreren *Realtime Primitives Nets* verwendet und ggf. weiterverarbeitet werden. Auch Sensordaten von Aktuatoren (z. B. Positionswerte von Encodern) können folglich in mehreren Kommandos gelesen werden. Der steuernde Zugriff auf einen Aktuator ist nur von einem Kommando gleichzeitig möglich und erfolgt ebenfalls über ein Echtzeitprimitiv auf das entsprechende *Realtime Device*.

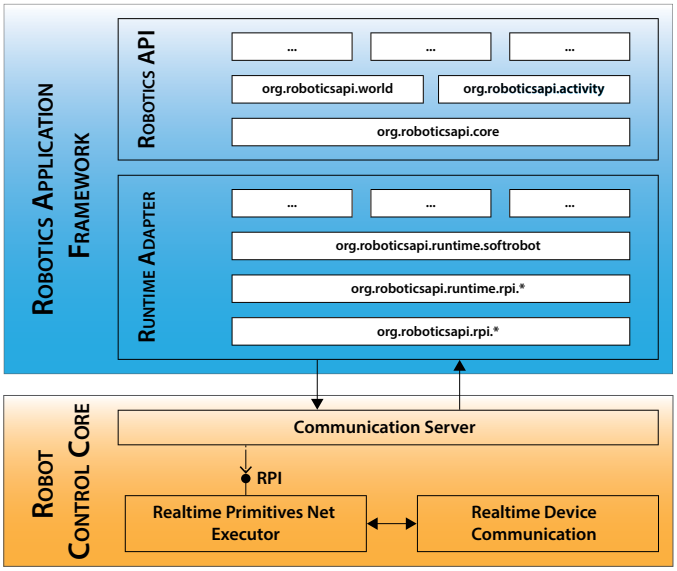


Abbildung 4.7: Das *Robotics Application Framework* besteht aus der *Robotics API* und einem Adapter, der für die Kommunikation der Echtzeitrobotersteuerung zuständig ist. Intern sind beide in Komponenten unterteilt.

4.3.2 *Robotics Application Framework*

Oberhalb des echtzeitfähigen RCC ist in der *SoftRobot*-Architektur das *Robotics Application Framework* angeordnet. Das Framework besteht insgesamt aus zwei Bestandteilen, die beide in Abbildung 4.7 dargestellt sind. Für die Anwendungsentwicklung ist die *Robotics API* der zentrale Bestandteil und stellt eine objektorientierte Programmierschnittstelle zur Verfügung. Diese enthält Klassen zur Beschreibung von mechatronischen Geräten und deren Operationen. Diese Operationen können über einen flexiblen Mechanismus miteinander kombiniert werden [18]. Dadurch können mehrere Roboter, Werkzeuge oder andere Geräte miteinander kombiniert werden. Durch den RCC als Ausführungsumgebung dieser kombinierten Operationen ist die Synchronisation der Geräte echtzeitfähig. Das bedeutet, dass zeitlich exakt synchronisierte Operationen möglich sind, z. B. das Aktivieren eines Werkzeugs auf einem bestimmten Punkt einer Roboterbahn. Darüber hinaus können sogar gemeinsame, synchronisierte Bewegungen mehrerer Roboter bzw. Aktuatoren mit der *Robotics API* umgesetzt werden. Für die Robotik unterstützt die *Robotics API* eine Vielzahl unterschiedlicher Bewegungsarten. Neben vorgeplanten Bewegungen können ebenfalls von Sensoren überwachte und geführte Bewegungen spezifiziert werden.

Robotics API

Neben der *Robotics API* ist ein spezieller *Runtime Adapter* notwendig, der die *Robotics API* mit der Echtzeitrobotersteuerung verbindet. Dieser Adapter ist notwendig, da mit RPI nur die Semantik von echtzeitkritischen Aufgaben und deren Lebenszyklus spezifiziert wird. Die Syntax zur Beschreibung dieser Aufgaben und das verwendete Kommunikationsprotokoll ist spezifisch für eine Implementierung. Wie in Abbildung 4.7 dargestellt erfolgt die Kommunikation mit dem RCC ausschließlich über den Adapter. Die Menge der verfügbaren Echtzeitprimitive ist in RPI ebenfalls nicht festgelegt, sondern ist erweiterbar für zukünftige Geräte, Bewegungen und Operationen. Daher ist der Adapter dafür zuständig, die Operationen der *Robotics API* auf *Realtime Primitives*, *Fragments* bzw. *Realtime Primitives Nets* zu übersetzen.

Adapter für den RCC

Der Adapter entspricht dabei dem in [97] vorgestellten, gleichnamigen Entwurfsmuster und entkoppelt die *Robotics API* von der konkreten Implementierung eines RCC. Durch diese Trennung können folglich unterschiedliche Implementierungen einer RPI-kompatiblen Echtzeitsteuerung mit der *Robotics API* verwendet werden. Im Rahmen von *SoftRobot* wurde ein Adapter für die Referenzimplementierung des RCC entwickelt.

Abbildung 4.7 zeigt, dass sowohl die *Robotics API* als auch der dazugehörige *Runtime Adapter* intern aus einer Menge von Paketen (engl.: *packages*) bestehen[5]. Dadurch sind die *Robotics API* und der Adapter ebenso flexibel und erweiterbar wie der RCC mit seiner offenen Menge von Echtzeitprimitiven (vgl. Kap. 6). Die zentralen Pakete der *Robotics API* und des Adapters sind in Abbildung 4.7 dargestellt. Sie stellen das Grundgerüst dar, auf dem weitere Pakete aufgebaut und implementiert werden. Dazu stellen sie Schnittstellen und (abstrakte) Klassen bereit, die einerseits zur Programmierung verwendet werden können und von denen andererseits abgeleitet werden kann, um neue Konzepte, Geräte oder Operationen zu implementieren.

Aktuatoren in der Robotics API

Zentrales Element der *Robotics API* ist das Paket *org.roboticsapi.core*, das das grundlegende objektorientierte Modell definiert. Das Modell basiert auf der Betrachtungsweise, dass es auf der einen Seite steuerbare Geräte mit einer Menge von Eigenschaften gibt und auf der anderen Seite Aktionen, die von diesen Geräten ausgeführt werden können. Eine steuerbares Gerät, das mit seiner Umgebung interagieren und diese so verändern kann, wird durch die Klasse ACTUATOR repräsentiert. Von der Klasse kann weiter abgeleitet werden, um konkrete Ausprägungen eines Aktuators zu definieren. Damit können durch das objektorientierte Modell alle in der Robotersteuerung verfügbaren und dadurch steuerbaren Geräte repräsentiert werden. Dazu zählen vor allem Roboter und Werkzeuge, aber auch Lineareinheiten oder mobile Plattformen. Detailliert wird die Modellierung in Abschnitt 5.2 vorgestellt.

Sensorwerte in der Robotics API

Für den Zugriff auf echtzeitfähige Datenwerte, insbesondere Sensordaten, existiert die Klasse REALTIMEVALUE. Sie repräsentiert eine typisierte Datenquelle, die in der echtzeitfähigen Robotersteuerung zur Verfügung steht. Bei Datenquellen handelt es sich in der Regel um Sensorwerte, die sich über die Zeit verändern. Ein Beispiel ist die aktuelle Winkelposition einer Roboterachse, die von Encodern gemessen wird und über die Robotersteuerung zur Verfügung steht. Jedoch können damit auch andere Datenwerte abgebildet werden. Beispiele sind Daten eines laufenden Kommandos, echtzeitfähige Berechnungen oder auch konstante Werte. Durch Unterklassen stehen entsprechende Typisierungen (boolesche Werte, Fließkommazahlen) und abgeleitete REALTIMEVALUES zur Verfügung, um Datenwerte, z. B. durch arithmetische Operationen, weiterzuverarbeiten. Weiterhin bieten REALTIMEVALUES die Möglichkeit, sich gemäß dem Observer-Pattern [97] über Änderungen des Datenwerts informieren zu lassen.

In der objektorientierten Programmierung werden die Fähigkeiten eines Objekts gewöhnlich durch seine Methoden repräsentiert. In der *Robotics API* sind Fähigkeiten jedoch als eigene Objekte modelliert. Für die Spezifikation einer solchen Fähigkeit sieht die *Robotics API* die Klasse ACTION vor. Spezialisierungen dieser Klasse beschreiben durch ihren Typ eine spezielle Operation (z. B. eine bestimmte Bewegungsart) und kapseln alle Informationen, die nötig sind, um eine solche Operation mithilfe eines Aktuators auszuführen (z. B. den Start- und Zielpunkt sowie die max. Bahngeschwindigkeit). Als Spezifikation einer Fähigkeit ist eine ACTION nicht ausführbar.

5 In Kapitel 6 wird daraus eine Aufteilung in einzelne Komponenten bzw. Bundles abgeleitet.

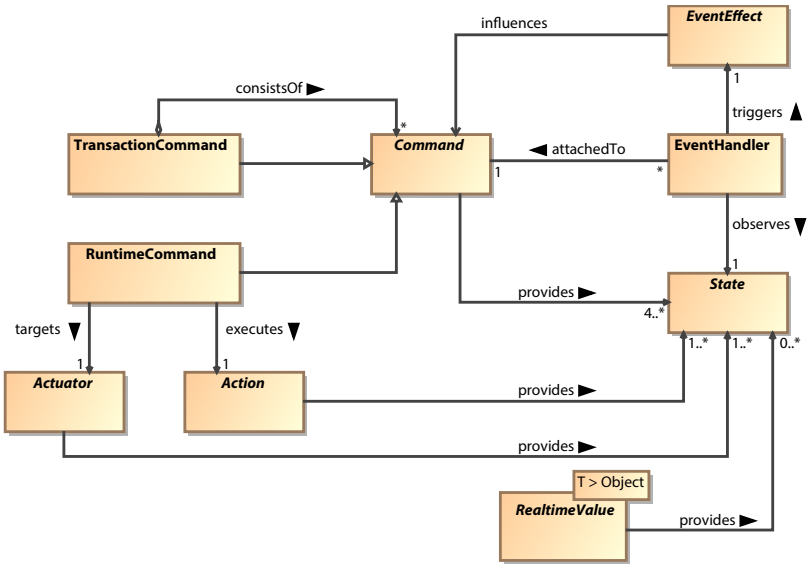


Abbildung 4.8: Die Klasse COMMAND repräsentiert eine Operation, die auf einer Echtzeitrobotersteuerung ausgeführt werden kann.

Kommandos in der Robotics API

Erst die Klasse COMMAND repräsentiert eine ausführbare Einheit (vgl. Abb. 4.8). Ein COMMAND ist definiert als eine Operation für einen oder mehrere Aktuatoren, die vollständig an einen RCC übertragen und dort atomar unter Echtzeitbedingungen ausgeführt wird. Die Klasse kapselt analog zu dem *Command Pattern* (vgl. [97]) eine ausführbare Operation als eigenes Objekt. Dies hat den Vorteil, dass erstens die Menge an verfügbaren Operationen eines Aktuator beliebig erweiterbar ist und zweitens das einzelne Fähigkeiten zu größeren und komplexeren Fähigkeiten zusammengefasst werden können. Ein RUNTIMECOMMAND ist ein elementares COMMAND, das für genau einen ACTUATOR über eine ACTION definiert ist. Folglich wird dadurch eine ACTION an ein konkretes Gerät, d. h. einen ACTUATOR, gebunden. Während ein RUNTIMECOMMAND die elementare Form eines Kommandos darstellt, können in einem TRANSACTIONCOMMAND mehrere Kommandos miteinander zu einer atomar ausgeführten Einheit verknüpft werden. Dementsprechend besteht ein TRANSACTIONCOMMAND aus mehreren inneren COMMANDS und ist selbst ebenfalls ein COMMAND. Alle inneren Kommandos werden gemeinsam an einen RCC übertragen und dort auf Basis von zuvor spezifizierten Bedingungen gestartet und gestoppt.

Wie in Abbildung 4.8 dargestellt, erfolgt die Verknüpfung der inneren COMMANDS durch Ereignisse. Dazu sind STATES notwendig, die spezifisch für ein Kommando definiert werden und während der Ausführung aktiv werden können [18]. Jeder Aktuator und jedes COMMAND hat eine feste Menge an STATES (z. B. ob ein Kommando bereits terminiert ist). Zusätzlich können für jede ACTION und für jeden REALTIMEVALUE spezifische STATES definiert werden, die im Kontext der aktuellen Operation von Bedeutung sind (z. B. ob ein bestimmter Bahnpunkt erreicht wurde). Das Aktivieren bzw. Deaktivieren eines STATES stellt ein Ereignis dar, auf den ein EVENTHANDLER registriert werden kann und der einen entsprechenden EVENTEFFECT auslösen kann. Typische Effekte sind bspw. das Starten oder Stoppen eines Kommandos. Die Auswertung der STATES und das Auslösen eines Effekts wird mithilfe der *Robotics API* spezifiziert, jedoch während der Ausführung eines Kommandos auf dem RCC ausgewertet.

Basierend auf dem oben vorgestellten Konzept der COMMANDS führt das Paket *org.roboticsapi.activity* ein zusätzliches Modell für Operationen mit einer speziellen Ausführungssemantik ein (vgl. [10]). Diese neuen Operationen werden ACTIVITY genannt. Der Fokus liegt dabei sehr stark auf einer einfachen Benutzbarkeit für Applikationsentwickler. Während eine ACTION vollständig spezifiziert sein muss, versucht eine ACTIVITY einen Teil der notwendigen Informationen aus *Meta-Daten* der vorherigen ACTIVITIES zu extrahieren; zum Beispiel den Startpunkt einer Bewegung, der gleichzeitig Endpunkt der vorherigen Bewegung ist. Da sich eine ACTIVITY weiterhin auf ein COMMAND abstützt, wird dies ausschließlich durch das Hinzufügen von expliziten Meta-Informationen zu einem COMMAND und einem speziellen Ausführungsplaner (engl.: *scheduler*) erreicht. Auf das Konzept wird nochmals in Abschnitt 5.2 eingegangen.

Um Objekte der physischen Welt und deren geometrische Beziehungen zu modellieren, führt das Paket *org.roboticsapi.world* die entsprechenden Konzepte und Klassen ein. Dabei werden kartesische Koordinatensysteme verwendet, um die Translationen und Rotationen von Objekten und Punkten zueinander und allgemein im Raum zu beschreiben. Dementsprechend enthält dieses Paket der *Robotics API* Klassen, um die mathematischen Beschreibungen für Translationen und Rotationen zu repräsentieren. Diese Klassen werden zusammen mit den mathematischen Grundlagen in Abschnitt 5.3 erklärt. Daneben werden über das Paket *org.roboticsapi.world* Klassen eingeführt, um die physischen Eigenschaften von Geräten und den Elementen der Arbeitsumgebung zu modellieren. Dadurch ist es möglich, real vorhandene Bestandteile der Arbeitsumgebung (z. B. Werkstücke) explizit als Softwareobjekte darzustellen und die, für die Aufgabe wichtigen Informationen zu modellieren (vgl. Kap. 5).

Die drei bisher vorgestellten Pakete bilden das Herzstück der *Robotics API*. Darauf aufbauend können neue Klassen entwickelt und in die *Robotics API* integriert werden (vgl. Kap. 6). Analog zur *Robotics API* besteht auch der *Runtime Adapter* aus einer Vielzahl von Paketen. Diese teilen sich einerseits in RPI-spezifische und andererseits in RCC-spezifische Pakete auf. Während die RPI-spezifischen Pakete von mehreren Adaptern wiederverwendet werden können, sind die RCC-spezifischen Pakete auf eine konkrete Implementierung zugeschnitten und können nicht oder nur sehr eingeschränkt wiederverwendet werden.

Übersetzung nach RPI

Das in Abbildung 4.7 als *org.roboticsapi.rpi.** dargestellte Paket ist in drei einzelne Pakete unterteilt, die folgenden Inhalt haben:

- Im Paket *org.roboticsapi.rpi.common* sind Klassen definiert, um die Konzepte von RPI im *Robotics Application Framework* verfügbar zu machen. Dazu existiert u. a. ein objektorientiertes Modell für *Realtime Primitives*, *Fragments* und *Realtime Primitives Nets*.
- Darauf aufbauend sind im Paket *org.roboticsapi.rpi.core* konkrete Typen von *Realtime Primitives* definiert. Diese Definitionen stellen eine sehr generische Menge von Echtzeitprimitiven dar, die ein RCC anbieten sollte. Darunter fallen bspw. die in Abb. 4.3 (a) bis Abb. 4.3 (c) dargestellten Echtzeitprimitive. Instanzen davon können als Proxy [97] für die Echtzeitprimitive eines RCC gesehen werden.
- Im Paket *org.roboticsapi.rpi.world* sind ebenfalls konkrete Typen von *Realtime Primitives* definiert, welche die mathematischen Berechnungen der notwendigen geometrischen Konzepte abbilden. Darunter fällt bspw. das in Abb. 4.3 (d) dargestellte Echtzeitprimitiv.

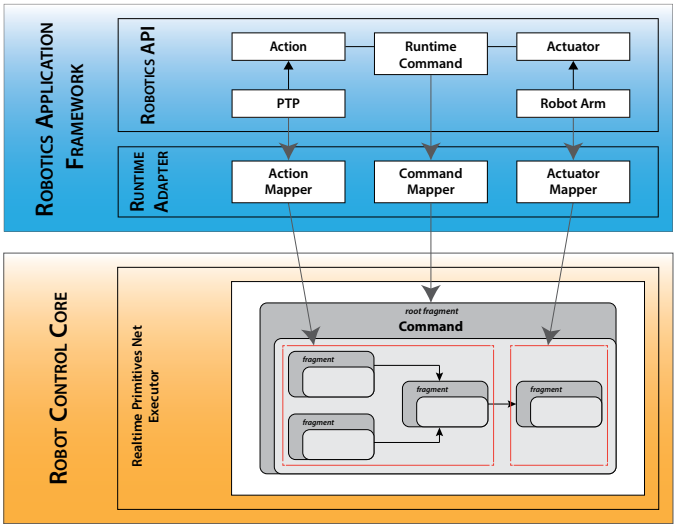


Abbildung 4.9: Die Programmierkonzepte der *Robotics API* werden in Fragmente übersetzt und zu *Realtime Primitives Nets* kombiniert, die auf der Echtzeitrobotersteuerung ausgeführt werden können.

Das objektorientierte Modell eines *Realtime Primitives* ist immer nur ein Stellvertreter für dessen Implementierung auf einem RCC. Jedoch existieren dadurch auf objektorientierter Ebene syntaktische Regeln um Echtzeitprimitive über *Links* miteinander zu verbinden und zu Fragmenten zu kombinieren. Ebenfalls vereinfacht dieses Modell die Serialisierung eines *Realtime Primitives Nets* für die Kommunikation mit einem RCC. Die Klassen aus *org.roboticsapi.rpi.common* können zudem verwendet werden, um eigene Erweiterungen für den *Runtime Adapter* zu entwickeln. Im Besonderen, falls diese Erweiterungen neue Echtzeitprimitive erfordern, kann ein objektorientiertes Modell dieser Echtzeitprimitive abgeleitet werden.

Das in Abbildung 4.7 als *org.roboticsapi.runtime.rpi.** dargestellte Paket ist ebenfalls in drei einzelne Pakete unterteilt. Während die bereits vorgestellten Pakete mit den verfügbaren Echtzeitprimitiven die statischen Aspekte von RPI, beinhalten, enthalten die *org.roboticsapi.runtime.rpi.** Pakete den dynamischen Aspekt. Sie bieten Funktionalität an, die zur Laufzeit aus Konzepten der *Robotics API* ein *Fragment* bzw. ein *Realtime Primitives Net* erzeugen. Man spricht in diesem Zusammenhang von einem *Mapper* [235].

Durch *Mapper* können Actions, RealtimeValues und Actuators in *Fragments* übersetzt werden. Für die Verknüpfung von Kommandos können *Mapper* zudem States und EventEffects in ein *Fragment* transformieren. Schließlich werden auch Commands über *Mapper* in ein *Fragment* bzw. *Realtime Primitives Net* übersetzt. Schematisch ist dies in Abbildung 4.9 an einem einfachen Beispiel dargestellt. Das RuntimeCommand, bestehend aus einer Action und einem Actuator, wird in ein *Realtime Primitives Net* transformiert. Dabei wird jeder Bestandteil einzeln über einen eigenen *Mapper* übersetzt. Ausführlich wird dieser Vorgang in [235] beschrieben.

Die beiden weiteren *org.roboticsapi.runtime.rpi.** Pakete stellen bereits fertige *Mapper* und Fragmente zur Verfügung:

- In *org.roboticsapi.runtime.rpi.core* sind *Mapper* und Fragmente definiert, um die Konzept aus dem Paket *org.roboticsapi.core* auf Echtzeitprimitive des Pakets *org.roboticsapi.rpi.core* zu übersetzen.

- In *org.roboticsapi.runtime.rpi.world* sind ebenfalls *Mapper* und Fragmente definiert, um die Konzept aus *org.roboticsapi.world* auf Echtzeitprimitive aus *org.roboticsapi.rpi.world* zu übersetzen.

Durch das Konzept der *Mapper* können neue Aktionen, Sensoren und Aktuatoren der *Robotics API* flexibel und erweiterbar für eine Echtzeitrobotersteuerung umgesetzt werden.

Die bisher vorgestellten Komponenten des *Runtime Adapters* waren noch unabhängig von einer konkreten Implementierung und haben ausschließlich Konzepte von RPI in das *Robotics Application Framework* eingeführt. Dagegen ist im Paket *org.roboticsapi.runtime.softrobot* die Kommunikation mit der Referenzimplementierung des Robot Control Cores implementiert. Somit stellt dieses Paket den eigentlichen Adapter zu dem Robot Control Core dar. Zudem sind dort einige Schnittstellen definiert und Funktionen implementiert, die ein Laden und Konfigurieren von *Realtime Devices* ermöglichen. Dies wird ausführlich in Kapitel 6 behandelt.

4.4 VERWANDTE ARBEITEN

Die vorgestellte Softwarearchitektur unterstützt die echtzeitfähige Ausführung von Steuerungsoperation. Allerdings wird über eine objektorientierte Programmierschnittstelle von der eigentlichen Echtzeitprogrammierung abstrahiert. Ähnlich versuchen auch Roboterprogrammiersprachen (vgl. Abschn. 3.2) die echtzeitfähige Interpolation von Bewegungen vor dem Bediener zu verstecken. Dazu stehen auf den Robotersteuerungen spezielle Compiler und Interpreter zur Verfügung. Diese kompilieren im Hintergrund ein Roboterprogramm und interpretieren es echtzeitfähig auf der Robotersteuerungen (vgl. Abschn. 3.1). Jedoch bieten die Sprachen nur einen begrenzten Funktionsumfang, der sich in der Regel auf die Steuerung eines Roboters beschränkt. Die Einschränkungen aktueller Roboterprogrammiersprachen wurden bereits in Abschnitt 3.4 beschrieben.

Eine der ersten und einflussreichsten Architekturen für Robotersysteme wurde in den 1980er Jahren von Brooks [47] als *subsumption architecture* vorgeschlagen. Die Architektur besteht aus verschiedenen Ebenen, die aufeinander aufbauen und über endliche Zustandsmaschinen miteinander kommunizieren. Jede Zustandsmaschine repräsentiert ein *Behaviour* bzw. Verhalten des Roboters. Da mehrere *Behaviours* gleichzeitig aktiv sein können, kennt die Architektur einen Mechanismus, um einfaches Verhalten durch intelligentere bzw. höhere Verhaltensweisen außer Kraft zu setzen. Durch miteinander interagierende Ebenen von unterschiedlichen Verhaltensweisen können schrittweise komplexe Robotersysteme entwickelt werden.

Eine der ersten dreischichtigen Roboterarchitekturen, die nach der Anzahl der Schichten als *3T architectures* [40] bezeichnet werden, wurde von Firby [89] vorgestellt. Sie verbindet zum ersten Mal planerische und reaktive Aspekte in einer Roboterarchitektur. Jedoch wurde mit den Reactive Action Packages (RAP) nur die mittlere Schicht umgesetzt. Zeitgleich und unabhängig von Firby haben Bonasso et al. [39] ebenfalls eine dreischichtige Roboterarchitektur entwickelt. Auf der untersten Ebene werden reaktive Roboterfähigkeiten (engl.: *skills*) mit synchronen Schaltungen beschrieben. Die mittlere Schicht ermöglicht eine bedingte Ablauflogik, indem Roboterfähigkeit aktiviert und deaktiviert werden, um eine Roboteraufgabe zu erfüllen. Eine Variante der mittleren Schicht wurde von Simmons und Apfelbaum [246] mit hierarchischen *Task Trees* vorgestellt. Die oberste Schicht ist für die (autonome) Planung von Aufgaben zuständig.

Weitere Vertreter dreischichtiger Roboterarchitekturen sind LAAS [5] und ATLANTIS [98]. Eine Erweiterung für mehrere Roboter wurde von Simmons et al. in [247] vorgestellt. Die in *SoftRobot* entstandene Architektur weist eine Ähnlichkeiten mit den beiden unteren Schichten der 3T-Architekturen [40] auf. Eine ACTIVITY kann eine reaktive Roboterfähigkeit repräsentieren, die auf dem RCC unter harten Echtzeitanforderungen ausgeführt wird. Mithilfe der *Robotics API* in der darüber gelagerten Schicht kann der Arbeitsablauf des Robotersystems spezifiziert werden. Die selbstständige Planung von Aufgaben wurde in *SoftRobot* nicht betrachtet, kann aber mit der Architektur umgesetzt werden.

Neben den oben vorgestellten dreischichtigen Roboterarchitekturen wurden auch Architekturen mit nur zwei Schichten entwickelt. Ein bekannter Vertreter dieser Architekturen ist die Coupled Layered Architecture for Robot Autonomy (CLARAty) [275], die von der NASA entwickelt wurde. Die Architektur, die vorwiegend für Weltraumrover gedacht ist, besteht aus einem *Functional Layer* und einem *Decision Layer*. Die funktionale Schicht besteht aus einer objektorientierten Hierarchie, die eine immer abstraktere Steuerung des Roboters ermöglicht (z. B. von einer einfachen Motorsteuerung über eine Fahrzeugsteuerung hin zur sensorbasierten Navigation des Rovers). Der *Decision Layer* ist sowohl für die Planung als auch die Ausführung von Aufgaben zuständig. Eine mögliche Instanz des *Decision Layers* in CLARAty ist CLEaR [83], das eine kontinuierliche Ausführung von Aufgaben ermöglicht. Obwohl CLARAty sehr interessante Aspekte aufweist, die für die industrielle Robotik interessant sind, liegt der Fokus doch auf autonomen Landefahrzeugen.

Eine weitere zweischichtige Architektur ist die Cooperative Intelligent Real-time Control Architecture (CIRCA) [186]. Dabei liegt der Fokus darauf, ein zuverlässiges Roboterverhalten zu garantieren. Die Architektur besteht aus einem Real-Time Subsystem (RTS) und einem Artificial Intelligence Subsystem (AIS), die beide unabhängig voneinander sind. Die Aufgabe des AIS ist es, ausführbare Pläne für Roboteraufgaben zu erstellen, die bereits mögliche Fehler oder Sicherheitsrisiken zu berücksichtigen. Diese Pläne können anschließend vom RTS mit garantierten Reaktionszeiten ausgeführt werden. Falls ein Plan aufgrund einer geänderten Umgebung nicht mehr ausgeführt werden kann, kann das AIS den Plan dynamisch adaptieren.

Bei ORCCAD [43] handelt es sich ebenso um eine zweischichtigen Roboterarchitektur, die Echtzeit-kritische Roboteraufgaben ermöglicht. Während sich auf der unteren Schicht kontinuierliche Regelkreise beschreiben lassen, werden auf der darüber liegenden Schicht Ereignis-gesteuerte, diskrete Aktionen verwendet. Dazu unterscheidet ORCCAD zwischen *Robot Tasks*, die einfaches Roboterverhalten als Regler abbilden, und *Robot Procedures*, mit denen sich komplexes Roboterverhalten implementieren lässt. Eine *Robot Procedure* wird in der Programmiersprache MAESTRO [61] beschrieben und besteht dabei aus einer Menge von *Robot Tasks*. Für die Ausführung werden sowohl die *Robot Tasks* als auch die *Robot Procedures* in die synchrone Datenflusssprache ESTEREL [30] übersetzt. Durch formale Verifikation können Aussagen, die Sicherheit und Lebendigkeit von Robotersystemen betreffend, gezeigt werden [82]. Die Verwendung einer synchronen Datenflusssprache weist Parallelen auf. Allerdings ist das *Robotics Application Framework* durch die dynamische Verknüpfung und Übersetzung von Kommandos flexibler und erlaubt zudem eine Programmierung in einer universellen Programmiersprache im Vergleich zu ORCCAD.

In den letzten Jahren hat sich ein starker Trend zur komponentenbasierten Softwareentwicklung für experimentelle Robotersysteme herausgebildet (vgl. [48]). Dabei werden Robotersysteme aus einer Menge wiederverwendbarer Komponenten zusammengesetzt (vgl. Abschn. 2.1). Die Kommunikation erfolgt über Schnittstellen oder den Austausch von Nachrichten. Im Gegensatz zu den oben beschriebenen Architekturen entsteht dadurch keine Unterteilung des Robotersystems in unterschiedliche Schichten mit klar definierten Zuständigkeiten. Vielmehr entscheidet eine geeignete Wahl der Komponenten bezüglich Funktionalität und Granularität über die Güte der Architektur [49]. Um Zuständigkeiten besser zu trennen, schlagen Radestock und Eisenbach [224] für verteilte Systeme eine Aufteilung (*Separation of Concerns*) der Komponenten anhand von vier Aspekten – *Coordination*, *Computation*, *Configuration* und *Communication* – vor. Diese Aufteilung wird auch im Umfeld der Robotik stark propagiert (vgl. [220, 264]).

Im Zuge dieser Entwicklung hat sich eine Vielzahl komponentenbasierter Frameworks herausgebildet. Dabei liegt der Fokus dieser Frameworks sehr stark auf der mobilen bzw. autonomen Robotik [48]. Bekannte Beispiele für komponentenbasierte Frameworks sind Player [60, 102], ORCA [46, 171], MARIE [62], MIRO [261]. Durch den Fokus auf die mobile Robotik sind die Anforderungen für harte Echtzeit gering und dementsprechend bei diesen Frameworks nicht komponentenübergreifend gegeben [27]. Aktuell wird das Robot Operating System (ROS) [223] für eine Vielzahl an Anwendungen und Experimenten verwendet, da es als Open-Source-Software und durch eine wachsende Zahl an frei verfügbaren Komponenten und Algorithmen sehr beliebt ist. Zudem wird ROS im Moment durch das Konsortium *ROS-Industrial* [230] stärker für industrielle Anwendungen erschlossen. Jedoch müssen Echtzeitanforderungen innerhalb einer Komponente erfüllt werden, was eine komponentenübergreifende Synchronisation verhindert [251].

Ein weiteres Framework ist OpenRTM-aist [8, 9], welches die von der OMG standardisierte Spezifikation [198] für Robotics Technology Middleware (RTM) implementiert. Durch die Wahl eines geeigneten Ausführungskontextes können Komponenten unter Echtzeitbedingungen verwendet werden. Für die Programmierung des humanoiden Roboters *Justin* [209] hat das Deutsche Zentrum für Luft- und Raumfahrt (DLR) mit aRD ein eigenes echtzeitfähiges Framework entwickelt [27, 117]. Der als aRDx [111] bezeichnete Nachfolger ist darauf optimiert, große Datenmengen (z. B. für Bildverarbeitung) performant zwischen Komponenten zu übertragen. Ebenfalls für humanoide Roboter ist YARP [179] konzipiert, das ebenfalls Echtzeitanforderungen erfüllen kann. Eine Programmierschnittstelle für die Programmierung von Manipulatoren und deren Endeffektoren ist nicht Fokus dieser Frameworks. Allerdings könnten diese Frameworks für die Implementierung eines Robot Control Cores genützt werden.

Mit MiRPA [86] wurde an der TU Braunschweig eine echtzeitfähige Middleware entwickelt, die unter dem Echtzeitbetriebssystem QNX implementiert wurde. Als zentrales Element der Kommunikationsarchitektur stellt ein *Object Server* Dienste zum Nachrichtenaustausch (z. B. über TCP/IP) oder für einen kontrollierten Zugriff auf geteilte Speicherbereiche bereit. Das Programmiermodell basiert ebenfalls auf kommunizierenden, verteilten Komponenten, die als eigene Prozesse realisiert sind. Zur Implementierung der Kommunikation zwischen Prozessen kann auf eine spezielle API zurückgegriffen werden. Aufbauend auf MiRPA wurden verschiedene Programmierkonzepte für Industrieroboter entwickelt (vgl. [87, 88, 258]), die jedoch als Programmierschnittstelle nicht so universell sind wie die *Robotics API*.

Das Open RObot COntrol Software (OROCOS) Projekt [50] stellt ein bekanntes und verbreitetes komponentenbasiertes Framework für die Robotik bereit. Es besteht aus mehreren Open-Source-Bibliotheken, die verschiedene Aspekte der Robotik abdecken (vgl. [203]). Dazu gehört das Real-Time Toolkit (RTT), das die Entwicklung echtzeitfähiger Komponenten ermöglicht. Dazu bietet es eine Abstraktionsschicht für das darunterliegende Betriebssystem. Basierend auf OROCOS wurden ebenfalls verschiedene Programmierkonzepte entwickelt (vgl. [250]). OROCOS wird für die Referenzimplementierung des RCC verwendet (vgl. Abschn. 4.3.1).

Um komponentenbasierte Software für Roboter zu entwickeln, werden für die Anwendung üblicherweise spezifische Komponenten implementiert und mit vorhandenen Komponenten kombiniert. So entsteht eine Komposition, um eine spezifische Aufgabe mit dem Roboter umzusetzen. Damit der komplexe Arbeitsablauf einer industriellen Applikation realisiert werden kann, sind mehrere Komponeten zur Konfiguration und Koordination notwendig [220]. Dabei werden in Koordinationskomponenten oft endliche Zustandsautomaten (vgl. rFSM [146] oder ROS smach [231]) oder domänenspezifische Sprachen (vgl. Orocos RTT-Lua [147]) verwendet. Goldhoorn und Joyeux [104] verwenden einen speziellen *Component Manager*, um die Komposition der Komponenten zur Laufzeit zu adaptieren.

Während die komponentenbasierte Softwareentwicklung für bestimmte Klassen von Robotersystemen (z. B. von autonomen Robotern) geeignet sind, ist die *Robotics API* als objektorientierte Programmierschnittstelle konzipiert, um komplexe und echtzeitkritische Abläufen einer industriellen Applikation zu beschreiben [18]. Daher liegt der Fokus der in dieser Arbeit präsentierten Architektur auf einer flexiblen und erweiterbaren Programmierschnittstelle für die Anforderungen der industriellen Robotik. Größere Automatisierungssysteme oder Roboterzellen können jedoch auch aus einzelnen Komponenten zusammengesetzt werden, um so eine höhere Wiederverwendbarkeit und Änderbarkeit zu erreichen. Dazu wurde im Rahmen dieser Dissertation ein Konzept serviceorientierter Roboterzellen erarbeitet. Dies wird im dritten Teil dieser Arbeit anhand einer Fallstudie vorgestellt.

4.5 ZUSAMMENFASSUNG

Zusammenfassend ist festzuhalten, dass mit der gewählten Architektur die Anforderungen (vgl. Abschn. 4.2) erreicht wurden. Angerer [10] beschreibt sehr ausführlich, wie die funktionalen Anforderungen mit der *Robotics API* umgesetzt wurden. So fordert bspw. die funktionale Anforderung FR–8, dass eine bewegungsabhängige Synchronisierung von Kommandos möglich sein muss. Der zu beobachtende Zustand der Bewegung kann durch einen State diskretisiert werden. Das Eintreten dieses States ist ein Ereignis und kann über einen EventEffect das Starten oder Stoppen eines oder mehrerer Kommandos auslösen (vgl. Abschn. 4.3.2).

Die identifizierten Echtzeitmuster (vgl. Abschn. 4.2) finden sich ebenso in der Architektur wieder. Konkret bedeutet dies, dass sich flexibel Steuerungsoperationen definieren lassen, die jedes der vier Echtzeitmuster abdecken:

1. Das *MOTION-Pattern* besagt, dass geplante Bewegungen präzise und reproduzierbar ausgeführt werden. Dies wird durch die Ausführung in einer echtzeitfähigen Umgebung, dem Robot Control Core, sichergestellt. Durch den flexiblen Aufbau von *Realtime Primitives Nets* können Bewegungen frei definiert werden. Dieses Muster findet sich in den funktionalen Anforderungen FR–1 bis FR–5 wieder.

2. Das *SENSOR-Pattern* besagt, dass der Roboter durch einen oder mehrere Sensoren geführt wird. Durch die Integration der Sensoren in die Robotersteuerung ist ein echtzeitfähiger Zugriff auf die Sensorwerte möglich. Die Sensorwerte können nachbearbeitet und mit anderen Datenwerten verrechnet werden, um daraus Bewegungsvorgaben für einen oder mehrere Roboter zu generieren. Dieses Muster findet sich in den funktionalen Anforderungen FR–10 bis FR–14 wieder.

3. Das *SYNC-Pattern* besagt, dass räumliche Synchronisationsbeziehungen beschrieben werden, die während der Ausführung von Bewegungen aufrechterhalten werden. Auch hier ermöglicht der flexible Aufbau von *Realtime Primitives Nets* die Definition beliebiger Synchronisationsbeziehungen, die während der Ausführung geprüft und aufrechterhalten werden. Dieses Muster findet sich in den funktionalen Anforderungen FR–6 und FR–9 wieder.

4. Das *EVENT-Pattern* besagt, dass Ereignisse spezifiziert werden, deren Eintreten während der Ausführung Reaktionen auslösen können. Diese Ereignisse können in der *Robotics API* beschrieben und in ein *Fragment* übersetzt werden. Ähnlich generisch können auch Reaktionen auf Ereignisse beschrieben werden. Dieses Muster findet sich in den funktionalen Anforderungen FR–6 bis FR–8 und FR–12 wieder.

Dadurch, dass die Echtzeitmuster vollständig in der Architektur abgebildet werden, können die bestehenden Technologiepakete adäquat mit der neuen Architektur umgesetzt werden.

Das *Robotics Application Framework* ermöglicht die einfache Realisierung traditioneller Roboteranwendungen (vgl. FR–1 bis FR-5). Statt einer proprietären Roboterprogrammiersprachen kann dazu mit Java eine universelle, objektorientierte Sprache verwendet werden, die weit verbreitet ist und für ein breites Spektrum an Frameworks und Bibliotheken existiert. Auch kooperative Anwendungen (vgl. FR–6 bis FR-9) und eine stärkere Integration von Sensoren (vgl. FR–10 bis FR-14) werden durch das *Robotics Application Framework* möglich [18]. Zudem ist durch Java bzw. C/C++ der Fokus viel stärker auf der Verwendung von Standardsoftware, die auf PC-basierter Computerhardware läuft. Damit wird die Architektur den wichtigsten Zielen des *SoftRobot*-Projekts gerecht (vgl. Abschn. 4.1).

Die Erweiterbarkeit der Programmierschnittstelle bzw. der ganzen Softwarearchitektur war eine weitere, zentrale Forderung bei der Definition der Ziele von *SoftRobot* und wurde auf allen Ebenen berücksichtigt. Die Modularisierung der Architektur (vgl. Kap. 6) über alle Ebenen hinweg ist einer der wichtigsten Beiträge dieser Arbeit und ermöglicht eine flexible Plattform zur Entwicklung von Roboter- bzw. Automatisierungsanwendungen (vgl. Kap. 7). Neben der Erweiterbarkeit war Wiederverwendung ein zentrales Ziel der Architektur. Dies wird deutlich bei der Umsetzung und Modularisierung des *Robotics Application Frameworks* in Kapitel 6 und ist dort anhand einiger Beispiele illustriert. Daneben beschäftigt sich diese Arbeit auch damit, inwiefern Roboterprogramme generisch gestaltet werden können, um dort ebenfalls eine hohe Wiederverwendung zu erreichen. Grundlage dafür ist die objektorientierte Modellierung von Roboterzellen, die in Kapitel 5 vorgestellt wird. Sie ermöglicht es, dass wiederverwendbare Strategien und Fähigkeiten implementiert werden können (vgl. Kap. 10), durch die eine serviceorientierte Roboterzelle (vgl. Kap. 9) nach dem Baukastenprinzip zusammengesetzt werden kann.

5 ELEMENTE EINER ROBOTERZELLE

In Kapitel 4 wurde die entwickelte Softwarearchitektur vorgestellt. Zentraler Bestandteil ist die objektorientierte *Robotics API*, die als Programmierschnittstelle die Möglichkeit bietet, alle wichtigen Bestandteile einer Roboterzelle zu modellieren. Das umfasst die in der Roboterzelle vorhandenen Roboterarme und Werkzeuge, die Bauteile der Fertigungsprozesse, sowie die relevanten geometrischen Informationen wie z. B. Koordinatensysteme. Eine objektorientierte Beschreibung der Zelle ist die Voraussetzung für die serviceorientierte Implementierung einer Roboterzelle (vgl. Kap. 7). Die Objekte bilden die Eingabeparameter der Services und sind die Voraussetzung dafür, dass Services auf Basis des aktuellen Zellenzustands planen und entscheiden können (vgl. Kap. 9 und 10).

Die wichtigsten Konzepte zur Modellierung einer Roboterzelle werden in diesen Kapitel mithilfe der Unified Modeling Language (UML) [197] beschrieben. Dazu werden diese Konzepte als Klassen definiert und ihre Beziehungen zu anderen Konzepten in Form von Generalisierungen und Assoziationen erklärt. Um die Konzepte in Software abzubilden, existieren in der *Robotics API* Schnittstellen und (abstrakte) Klassen, die entweder direkt verwendet werden können oder von denen abgeleitet werden kann, um eigene Klassen zu realisieren. Daher folgen am Ende jedes Abschnitts Details, wie die jeweiligen Konzepte in der *Robotics API* implementiert wurden. Hier wird insbesondere zwischen Schnittstellen und (abstrakten) Implementierungen unterschieden. Durch entsprechende Stereotypen sind implementierungsnahe Modelle zu erkennen.

Einige der hier beschriebenen Konzepte wurden zum ersten Mal in [19] vorgestellt und in [120] detailliert. Das objektorientierte Modell der *Robotics API* wurde auch intensiv in der Dissertation von Andreas Angerer [10] behandelt[1]. Sein Fokus liegt jedoch sehr stark auf der funktionalen Modellierung von Geräten und Aktuatoren. Daher wird dort das Ausführungsmodell von Kommandos und Aktivitäten ausführlich beschrieben. Insbesondere die Kombination elementarer Aktionen zu komplexen Kommandos steht im Mittelpunkt. Bis auf eine Beschreibung geometrischer Konzepte wird bei Angerer [10] eine physische Modellierung von Bauteilen, mechatronischen Geräten oder ganzen Roboterzellen nicht gegeben.

5.1 ALLGEMEINE KONZEPTE

Abbildung 5.1 zeigt ein Klassendiagramm mit den wichtigsten Klassen zur Modellierung einer Roboterzelle. Ein zentrales Konzept stellt dabei die Klasse ENTITY dar. Es wird von allen in der Roboterzelle vorkommenden Konzepten weiter spezialisiert und ist wie folgt definiert:

Definition 5.1. Eine ENTITY (Entität) beschreibt ein materielles oder immaterielles Element, das in der zu betrachtenden Umgebung (im Speziellen in einer Roboterzelle) existiert. Eine Entität ist eindeutig identifizierbar.

1 Aufgrund des gemeinsamen Ursprungs der beiden Dissertationen im Forschungsprojekt *SoftRobot* gibt es vereinzelt Überschneidungen. Insbesondere bei Definitionen, die gleich oder sehr ähnlich sind, wird auf Angerer [10] verwiesen.

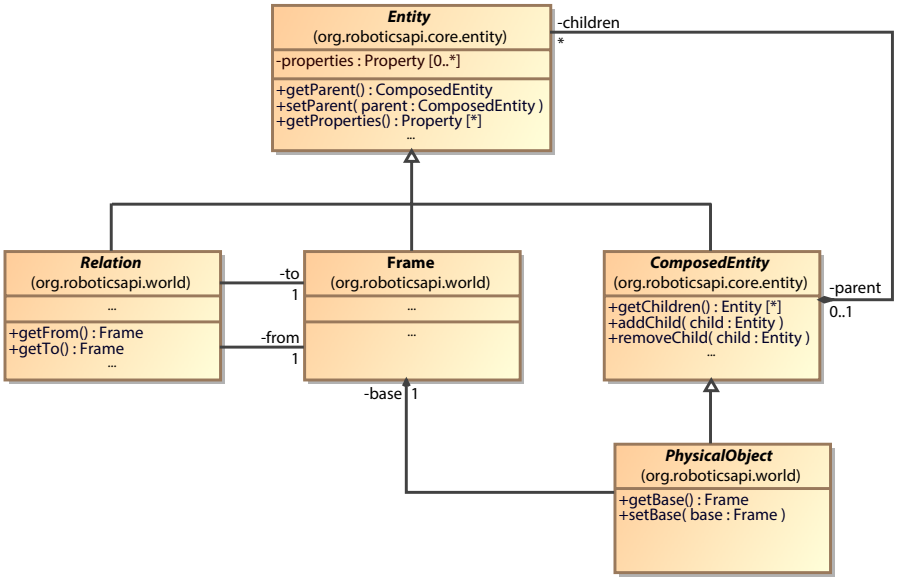


Abbildung 5.1: Eine ENTITY bildet die grundlegende Klasse zur Modellierung der einzelnen Elemente einer Roboterzelle. Daher leiten sich von ihr weitere Spezialisierungen ab.

Folglich kann ein Objekt der Klasse ENTITY entweder einen konkreten Gegenstand der Roboterzelle (z. B. einen Roboter oder ein Werkstück) oder ein abstraktes Konzept (z. B. einen Punkt oder ein Koordinatensystem) repräsentieren. Damit ist eine Entität die grundlegende Klasse zur Modellierung von Roboterzellen in der *Robotics API*. Außerdem besitzt eine Entität eine Menge an generischen Eigenschaften, die variabel und applikationsspezifisch gesetzt werden können.

Entitäten können wiederum Bestandteil einer übergeordneten Entität sein. Dazu hat die Klasse ENTITY eine Assoziation, die auf eine übergeordnete Entität, den optionalen *parent*, verweist. Diese übergeordnete Entität ist dabei immer eine Instanz der Klasse COMPOSEDENTITY, die wie folgt definiert ist:

Definition 5.2. Eine COMPOSEDENTITY (zusammengesetzte Entität) ist eine Spezialisierung einer ENTITY, die sich aus einer Menge von anderen (untergeordneten) Entitäten zusammensetzt.

Eine COMPOSEDENTITY stellt damit eine Komposition [197] von Entitäten dar, die als *children* bezeichnet sind. Eine Entität kann immer nur Teil von maximal einer zusammengesetzten Entität sein. Allerdings muss nicht jede Entität Teil einer anderen sein, sondern kann unabhängig existieren. Ist eine Entität jedoch Bestandteil einer zusammengesetzten Entität, so besteht eine Existenzabhängigkeit, d. h. die zusammengesetzte Entität bestimmt den Lebenszyklus ihrer Bestandteile. Falls eine COMPOSEDENTITY gelöscht wird, so wird auch jede einzelne ENTITY, die zu diesem Zeitpunkt Bestandteil der COMPOSEDENTITY ist, gelöscht.

Beispiel: Entitäten

Ein Beispiel für eine zusammengesetzte Entität ist ein Roboterarm, da dieser u. a. eine Komposition seiner Gelenke ist. Aufgrund der Existenzabhängigkeit können die Gelenke nicht ohne den Roboterarm bestehen. Jedes einzelne Robotergelenk für sich ist ebenfalls eine zusammengesetzte Entität und beinhaltet z. B. speziell ausgezeichnete Koordinatensysteme. Da die Kompositionsbeziehung zwischen Entitäten transitiv ist, sind diese Koordinatensysteme gleichzeitig Teil des Roboterarms.

Sowohl ENTITY als auch COMPOSEDENTITY sind abstrakte und generelle Konzepte für die Modellierung von Roboterzellen. Dementsprechend gibt es Spezialisierungen der beiden Klassen, die konkrete Gegebenheiten einer Roboterzelle repräsentieren (vgl. Abb. 5.1). Ein wichtiges Konzept stellt dabei die Klasse FRAME dar, welche wie folgt definiert ist:

Definition 5.3. Ein FRAME ist ein kartesisches Koordinatensystem und repräsentiert somit einen benannten Punkt im dreidimensionalen Raum mit Orientierung (vgl. [10]).

Ein FRAME ist eine konkrete Ausprägung der Klasse ENTITY und stellt das Basiskonzept dar, um geometrische Gegebenheiten einer Roboterzelle zu modellieren. Ein FRAME hat in der *Robotics API* kein ausgezeichnetes Bezugssystem (engl.: *Reference Frame*). Vielmehr ist ein FRAME mit einer Menge von anderen FRAMES über RELATIONS verknüpft. Die Klasse RELATION ist wie folgt definiert:

Definition 5.4. Eine RELATION stellt eine Beziehung zwischen zwei unterschiedlichen FRAMES her und definiert die geometrische Verschiebung zwischen diesen beiden FRAMES (vgl. [10]).

Damit spannen FRAMES und RELATIONS einen Graphen auf, wobei FRAMES die Knoten (engl.: *vertices*) und RELATIONS die Kanten (engl.: *edges*) darstellen (vgl. Abb. 5.2). Aus Gründen der Konsistenz sollten je zwei FRAMES nur durch jeweils eine RELATION direkt miteinander in Beziehung stehen. Obwohl jede RELATION Assoziationen zu einem FRAME *from* und zu einem FRAME *to* hat (vgl. Abb. 5.1), kann sie aus beiden Richtungen durchlaufen werden. Damit handelt es sich um einen einfachen, ungerichteten Graphen. Die Unterscheidung zwischen *from* und *to* ist für die korrekte Berechnung der geometrischen Verschiebung von Bedeutung (vgl. Abschn. 5.3).

Es ist anzumerken, dass FRAMES und RELATIONS nicht nur einen Graphen, sondern eine Menge an Graphen aufspannen können. Dadurch ist es möglich unterschiedliche räumliche Gegebenheiten zu modellieren (z. B. zwei unterschiedliche Roboterzellen). Durch das Einfügen einer RELATION zwischen zwei FRAMES unterschiedlicher Graphen, werden diese beiden Graphen miteinander vereinigt. Folglich kann die geometrische Verschiebung zwischen zwei beliebigen FRAMES nur bestimmt werden, wenn die beiden FRAMES miteinander verbunden sind:

Definition 5.5. Zwei FRAMES F_0 und F_n sind miteinander verbunden, genau dann wenn mindestens ein Pfad zwischen ihnen existiert. Ein Pfad ist definiert als Folge von paarweise verschiedenen FRAMES F_0 bis F_n, in der aufeinander folgende FRAMES F_i und F_{i+1} im Graphen durch eine RELATION verbunden sind.

In ungerichteten Graphen existiert ein solcher Pfad, falls beide FRAMES Element des gleichen Graphen sind. Durch geeignete Suchalgorithmen (vgl. [66, 114]) kann der Pfad bestimmt werden. Zudem kann ein Graph Zyklen enthalten, d. h. es existiert mindestens ein FRAME, von dem ein nicht leerer Pfad zu sich selbst konstruiert werden kann. Dadurch können FRAMES existieren, die über mehr als einen Pfad miteinander verbunden sind. In der *Robotics API* sind solche Zyklen erlaubt. Es können jedoch damit inkonsistente geometrische Zusammenhänge konstruiert werden, d. h. zwischen zwei FRAMES können – abhängig vom gewählten Pfad – unterschiedliche geometrische Verschiebungen berechnet werden.

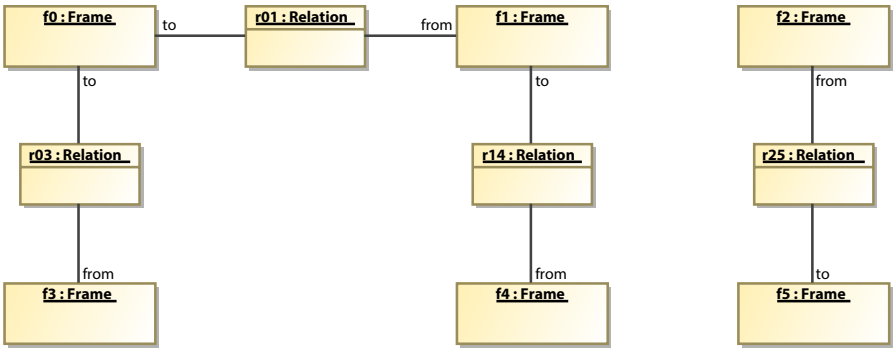


Abbildung 5.2: Die Instanzen der Klassen FRAME und RELATION spannen jeweils einen ungerichteten Graphen auf, wobei die FRAMES die Knoten und die RELATIONS die Kanten sind.

Abbildung 5.2 zeigt in einem Objektdiagramm beispielhaft, wie unterschiedliche Instanzen der Klasse FRAME durch RELATIONS miteinander verbunden sind. Das Beispiel zeigt zwei getrennte Graphen. Folglich kann keine geometrische Verschiebung zwischen *f1* und *f2* bestimmt werden. Durch das Einfügen einer weiteren RELATION zwischen diesen beiden FRAMES können die Graphen miteinander verbunden werden. Im Gegensatz zu *f2* kann ein Pfad zwischen *f1* und *f3* gefunden werden und die geometrische Verschiebung zwischen den beiden FRAMES berechnet werden. Falls man zwischen *f3* und *f4* eine neue RELATION eingefügen würde, würde ein Zyklus im linken Graphen entstehen. Hierbei ist auf Konsistenz zu achten.

Die Klasse FRAME ermöglicht als Koordinatensystem eine Beschreibung räumlicher Gegebenheiten und geometrischer Merkmale. Darüber hinaus ermöglicht die *Robotics API* jedoch auch die Modellierung von festen Körpern mit einer Masse und einer räumlichen Ausprägung. Dazu existiert die Klasse PHYSICALOBJECT, die wie folgt definiert ist:

Definition 5.6. Ein PHYSICALOBJECT (physisches Objekt) ist eine zusammengesetzte Entität, die sowohl eine Masse hat als auch Raum einnimmt und somit eine konkrete physische Ausprägung besitzt.

Dementsprechend kann die Klasse verwendet werden, um materielle Gegenstände einer Roboterzelle zu modellieren.

Aufgrund der sehr heterogenen Menge von Objekten, die in einer Roboterzelle vorkommen können, macht die Klasse PHYSICALOBJECT keine einschränkenden Annahmen über die Gestaltung konkreter Unterklassen. Sie setzt lediglich voraus, dass jedes PHYSICALOBJECT ein Basiskoordinatensystem *base* vom Typ FRAME hat. Das Basiskoordinatensystem kann beliebig gewählt werden, d. h. es kann innerhalb des Körpers, auf dem Körper oder gar außerhalb des Körpers definiert werden. Die Assoziation *parent* des FRAMES, der das Basiskoordinatensystems repräsentiert, sollte in der Regel auf das zugehörige PHYSICALOBJECT verweisen. Da jedes PHYSICALOBJECT über ein Basiskoordinatensystem verfügt, kann es über mindestens eine RELATION im kartesisches Raum platziert werden. Außerdem kann so die Position und Ausrichtung (d. h. die Pose) des Gegenstandes zu einem beliebigen verbundenen Koordinatensystem beschrieben werden. Gleichzeitig kann damit auch die Pose zu einem anderem PHYSICALOBJECT der Roboterzelle bzw. zu dessen Basiskoordinatensystem beschrieben werden.

Da jedes PHYSICALOBJECT auch eine COMPOSEDENTITY ist, kann es sich aus einer Menge von untergeordneten Entitäten zusammensetzen. Damit

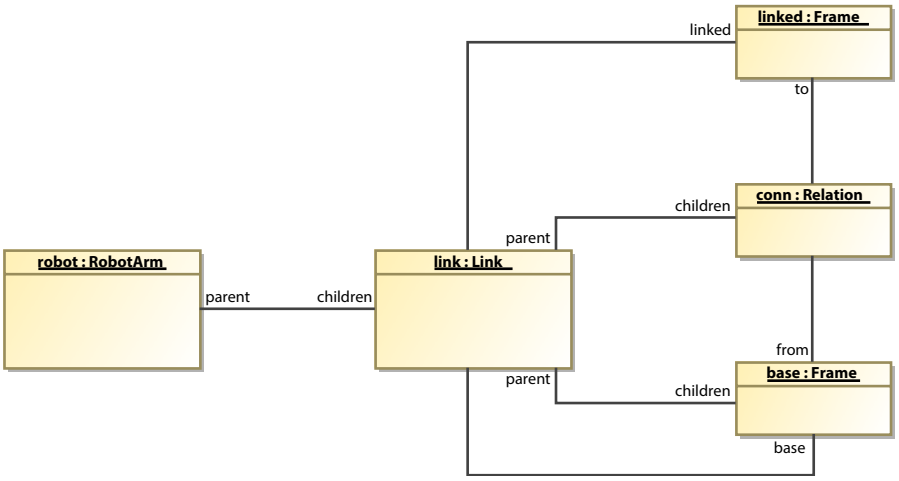


Abbildung 5.3: Die beiden Koordinatensysteme beschreiben zusammen mit der RELATION die geometrische Merkmale des LINKS, der Bestandteil eines Manipulators ist.

kann ein PHYSICALOBJECT mit einer Menge von FRAMES und RELATIONS verknüpft werden. Durch diese FRAMES können bspw. wichtige Merkmale auf einem PHYSICALOBJECT definiert werden. Dementsprechend können die zu einem PHYSICALOBJECT zugeordneten RELATIONS die Verbindung zwischen den FRAMES definieren. Durch diese (eigentlich rein auf Entitäten spezifizierte) Assoziation wird eine Verknüpfung zwischen den geometrischen Komponenten FRAME und RELATION auf der einen Seite und dem gegenständlichen PHYSICALOBJECT auf der anderen Seite hergestellt. Hierbei ist eine Navigation in beide Richtungen möglich.

Beispiel: Physische Objekte

Anhand eines Beispiels zeigt Abbildung 5.3 den Zusammenhang zwischen PHYSICALOBJECTS, FRAMES und RELATIONS. Ein LINK repräsentiert ein Strukturbauteil eines Manipulators und ist deswegen als Spezialisierung eines PHYSICALOBJECTS modelliert. Neben dem Basiskoordinatensystem *base* verfügt der LINK über ein weiteres Koordinatensystem *linked*, an dem (geometrisch) die nächste Achse beginnt. Die geometrische Verschiebung zwischen beiden FRAMES ist durch die RELATION *conn* definiert, die wie das Koordinatensystem *base* dem LINK zugeordnet ist. Sowohl der FRAME als auch die RELATION sind durch die Komposition untergeordnete Entitäten und damit Bestandteile des LINKS. Durch diese Zuordnung verweist die Assoziation *parent* des FRAMES bzw. der RELATION auf den LINK und ermöglicht eine Navigation in Richtung des PHYSICALOBJECTS.

Außerdem besteht durch die Komposition eine Existenzabhängigkeit zwischen dem FRAME und dem PHYSICALOBJECT. Analog gilt die Existenzabhängigkeit auch für die RELATION. Beide Abhängigkeiten sind notwendig, da sowohl der FRAME als auch die RELATION nur in Kombination mit LINK definiert sind und nicht eigenständig existieren sollten. Da das Koordinatensystem *linked* im Beispiel einer Roboterachse zugeordnet ist, ist es nicht Bestandteil des LINKS. Der LINK wiederum ist Bestandteil eines ROBOTARMS und definiert diesen durch die Assoziation *parent* als übergeordnete Entität.

Details zur Implementierung

Die oben vorgestellten Konzepte ENTITY, COMPOSEDENTITY und PHYSICALOBJECT sind in der *Robotics API* als Schnittstellen definiert, um eine größtmögliche Flexibilität in der Implementierung zu gewährleisten. Daneben existieren jedoch ebenfalls abstrakte Klassen, von denen direkt abgeleitet werden kann, um eigene Spezialisierungen der oben beschrieben

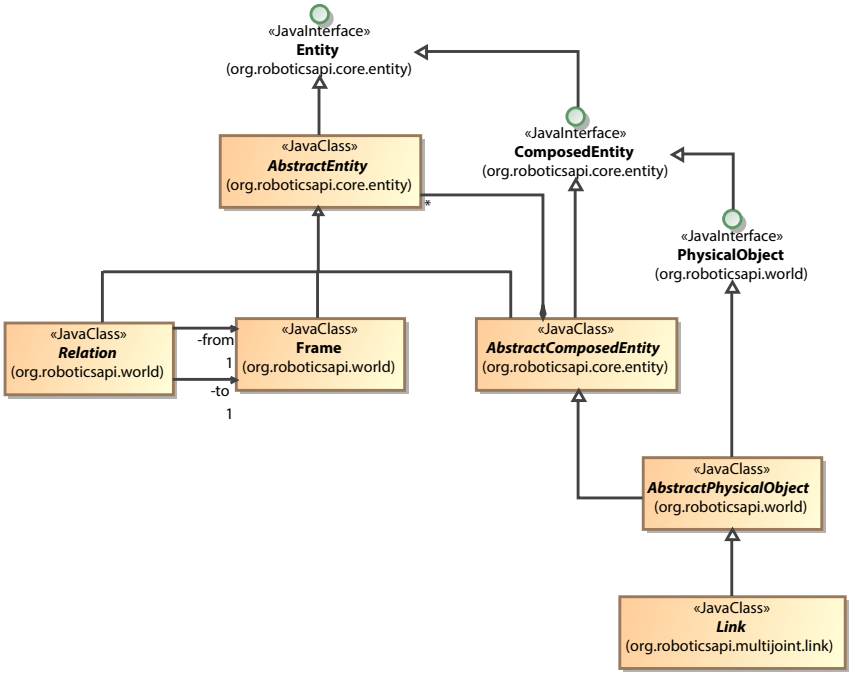


Abbildung 5.4: Das Klassendiagramm zeigt den Zusammenhang zwischen beiden Schnittstellen Entity und ComposedEntity sowie ihren abstrakten Implementierungen

Konzepte zu implementieren. Der Zusammenhang zwischen den Schnittstellen und ihren abstrakten Implementierungen ist in Abbildung 5.4 dargestellt. Ein wichtiges Merkmal der abstrakten Klassen ist die Implementierung der Kompositionsbeziehung zwischen Entity und ComposedEntity, sodass beide Enden der Komposition konsistent abgebildet werden. Die Klassen AbstractEntity und AbstractComposedEntity sind im Paket *org.roboticsapi.core.entity* definiert, die Klasse AbstractPhysicalObject dagegen im Paket *org.roboticsapi.world*. Abbildung 5.4 zeigt zudem, dass die bereits vorgestellte Klasse Link ebenfalls eine Spezialisierung von AbstractPhysicalObject ist. Die Klassen Frame und Relation sind beides Spezialisierungen einer AbstractEntity.

5.2 GERÄTE UND AKTUATOREN

Ein wichtiger Aspekt jeder Roboterzelle sind die verfügbaren Aktuatoren und Sensoren, da sie die möglichen Anwendungen definieren. Über Sensoren kann eine Applikation den Zustand der Umgebung, d. h. der Roboterzelle, wahrnehmen. Aktuatoren können daraufhin mit der Umgebung interagieren und sie so verändern. Daher stellt die Modellierung mechatronischer Geräte ein wichtiges Element der *Robotics API* dar. Die entsprechenden Konzepte dafür sind in Abbildung 5.5 dargestellt. An zentraler Stelle steht dabei das Device. Es stellt die Basisklasse für alle mechatronischen Geräte dar und ist wie folgt definiert:

Definition 5.7. Ein Device steht für ein mechatronisches Gerät, das mit seiner Umgebung interagieren kann (vgl. [10]).

Die Interaktion mit seiner Umgebung kann dabei entweder passiv (z. B. bei Sensoren) oder aktiv (z. B. bei Manipulatoren) sein. Ein Device ist zudem

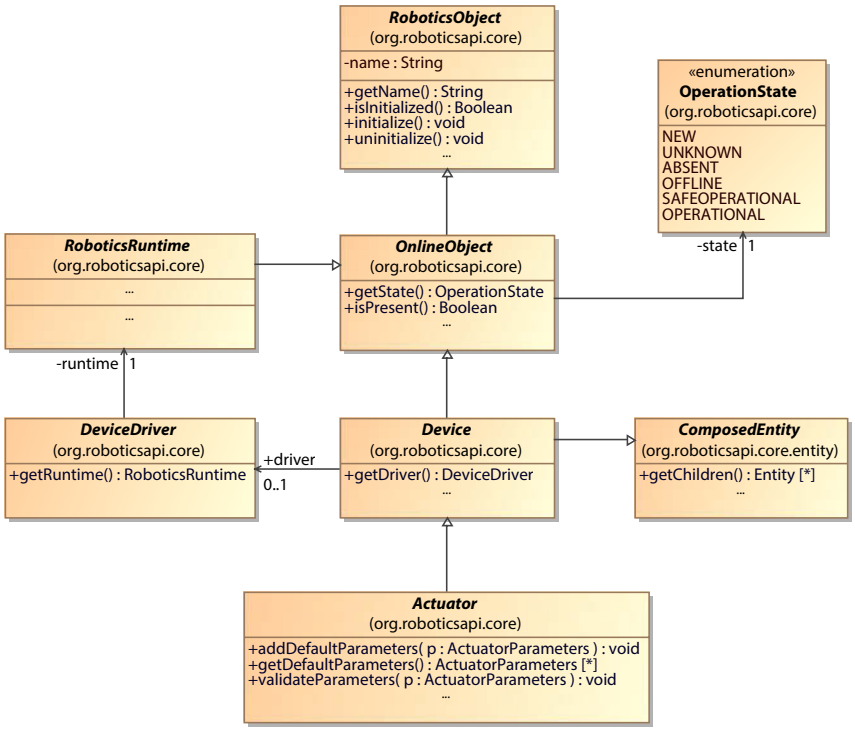


Abbildung 5.5: Ein DEVICE ist die Basisklasse für alle mechatronischen Geräte in der *Robotics API*. Eine Spezialisierung stellt der ACTUATOR dar, der aktiv steuerbare Geräte repräsentiert.

eine Spezialisierung der Klasse ENTITY, da es ein eindeutig identifizierbares Element der betrachtenden Umgebung (d. h. der Roboterzelle) ist. Durch seine, im Vergleich zu einem FRAME oder RELATION, gegebene Komplexität ist jedes DEVICE ein COMPOSEDENTITY und kann sich aus anderen Entitäten zusammensetzen. Ein DEVICE ist jedoch nicht per se ein PHYSICALOBJECT. So ist z. B. ein Robotergelenk zwar ein DEVICE jedoch kein PHYSICALOBJECT, da ein Gelenk konzeptioneller Art ist und damit nicht identisch mit einem Motor oder einem Getriebe.

Ein DEVICE repräsentiert ein mechatronisches Gerät mit seinen (physischen) Eigenschaften, aktuellen Datenwerten und möglichen Operationen. Es dient in der Modellierung bzw. in der Applikation als softwaretechnisches Gegenstück zu dem *realen* Gerät. Für die Kommunikation mit dem *realen* Gerät ist jedoch ein spezieller Gerätetreiber notwendig, über den jedes DEVICE verfügt. Ein Gerätetreiber wird durch die Klasse DEVICEDRIVER repräsentiert und ist wie folgt definiert:

Definition 5.8. Ein DEVICEDRIVER (Gerätetreiber) steht stellvertretend für ein an einer Robotersteuerung angeschlossenes Gerät. Der Gerätetreiber stellt den echtzeitfähigen Zugriff auf das Gerät her (vgl. [10]).

Dadurch ist ein DEVICE unabhängig von der verwendeten Robotersteuerung und des Zugriffs auf das *reale* Gerät. Die Kommunikation übernimmt dafür der Gerätetreiber, der dementsprechend von der konkreten Ausprägung einer Robotersteuerung (d. h. einer ROBOTICSRUNTIME) abhängig ist. Die abstrakte Klasse ROBOTICSRUNTIME ist wie folgt definiert:

Definition 5.9. Eine ROBOTICSRUNTIME steht stellvertretend für eine echtzeitfähige Robotersteuerung.

Dabei entspricht die echtzeitfähige Robotersteuerung einem Robot Control Core (RCC), wie er in Abschnitt 4.3.1 vorgestellt wurde. Folglich kann eine RoboticsRuntime als Proxy [97] für die Robotersteuerung angesehen werden und ist zugleich ein Adapter [97], der die Kommunikation mit dem RCC implementiert (vgl. Abschn. 4.3.2).

Kommunikation mit realen Geräten

Analog ist ein DeviceDriver ein Proxy für ein mechatronisches Gerät, das mit einer Robotersteuerung verbunden ist. Folglich gibt es auf der echtzeitfähigen Robotersteuerung mit dem *Realtime Device* ein Pendant, mit dem der DeviceDriver kommuniziert (vgl. Abschn. 4.3.1). Dieses Pendant implementiert die Schnittstellen mit dem eigentlichen Gerät und definiert so die verfügbaren Operationen auf der Robotersteuerung. Daher muss anhand des Gerätetreibers entschieden werden, wie ein Kommando der *Robotics API* in ein Kommando der Echtzeitrobotersteuerung übersetzt wird. Im Fall von RPI wird unter Berücksichtigung des Gerätetreibers der entsprechende Mapper (vgl. Abschn. 4.3.2) ausgewählt und das Kommando der *Robotics API* in ein *Realtime Primitives Net* übersetzt. Details zu dieser Übersetzung finden sich in [235]. Es ist somit möglich, ein Device mit unterschiedlichen Gerätetreibern zu instanziieren und zu steuern. Diese Trennung macht die Konzepte der *Robotics API* unabhängig von der konkreten Implementierung einer echtzeitfähigen Robotersteuerung.

Sowohl eine RoboticsRuntime als auch ein DeviceDriver haben einen Betriebszustand, der beschreibt, ob und wie mit der Robotersteuerung bzw. dem Gerät kommuniziert werden kann. Ein Device hat ebenfalls einen solchen Betriebszustand. Allerdings spiegelt ein Device lediglich den Betriebszustand seines Gerätetreibers wider. Um alle drei Klassen mit Betriebszustand konsistent zu betrachten, ist jede eine Spezialisierung der Klasse OnlineObject, die wie folgt definiert ist:

Definition 5.10. Ein OnlineObject ist ein RoboticsObject, welches einen Betriebszustand hat (vgl. [10]).

Die Klasse RoboticsObject, auf die sich die Klasse OnlineObject abstützt, ist wie folgt definiert:

Definition 5.11. Ein RoboticsObject ist ein benanntes Objekt, das vor der Benutzung initialisiert werden muss. Nach der Initialisierung sind seine Konfigurationseigenschaften nicht mehr veränderbar; erst durch explizites De-Initialisieren können diese wieder verändert werden (vgl. [10]).

Damit teilen sich eine RoboticsRuntime, ein DeviceDriver und ein Device folgende zwei Eigenschaften:

1. Sie müssen explizit initialisiert werden und können erst nach einer erfolgreichen Initialisierung verwendet werden. Dementsprechend müssen sie zuvor konfiguriert werden.

2. Sie haben einen expliziten Betriebszustand, der die mögliche Kommunikation mit der Robotersteuerung bzw. dem an der Robotersteuerung angeschlossenen Gerät definiert.

Diese beide Eigenschaften hängen zusammen, da erst nach der Initialisierung eine RoboticsRuntime bzw. ein DeviceDriver eine Verbindung mit der entsprechenden Robotersteuerung bzw. mit dem an der Robotersteuerung angeschlossenen Gerät aufbaut.

Ein Device ist allgemein als mechatronisches Gerät definiert, das mit seiner Umgebung interagieren kann. Das bedeutet zuerst einmal nur, dass es

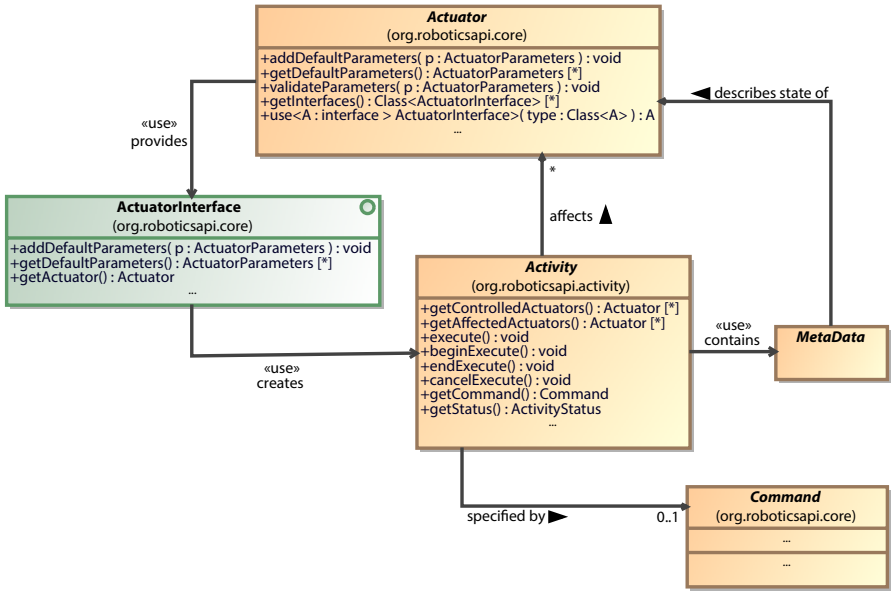


Abbildung 5.6: Ein ACTUATOR stellt seine verfügbaren Operationen durch eine Menge von ACTUATORINTERFACES bereit. Diese erstellen ACTIVITIES, die jeweils eine echtzeitkritische Operation repräsentieren.

über Sensoren Veränderungen seiner Umgebung wahrnehmen kann. Im Gegensatz dazu kann ein Aktuator aktiv gesteuert werden und so seine Umgebung auch verändern. Er wird durch die Klasse ACTUATOR repräsentiert und ist wie folgt definiert: *Aktuatoren*

Definition 5.12. Ein ACTUATOR (Aktuator) ist ein DEVICE, das aktiv in einem oder mehreren Freiheitsgraden (engl.: *degrees of freedom*) gesteuert werden kann. Ein Aktuator kann seine Umgebung dadurch verändern (vgl. [10]).

Als DEVICE benötigt ein ACTUATOR ebenso einen Gerätetreiber, um mit dem *realen* Aktuator, der an einer echtzeitfähigen Robotersteuerung angeschlossen ist, zu kommunizieren und diesen zu steuern. Allerdings verfügt ein ACTUATOR nicht direkt über Methoden zur Steuerung seiner Freiheitsgrade. Stattdessen werden mögliche Steuerungsoperationen in sogenannten ACTUATORINTERFACES zusammengefasst, die ein ACTUATOR wie *Services* zur Steuerung des realen Aktuators anbietet. Ein ACTUATORINTERFACE ist als Schnittstelle ausgelegt und wie folgt definiert:

Definition 5.13. Ein ACTUATORINTERFACE stellt eine Sammlung von Methoden dar, um semantisch verwandte, echtzeitfähige Operationen für eine bestimmte Art von ACTUATOR bereitzustellen (vgl. [10]).

In Abbildung 5.6 ist der Zusammenhang zwischen einem ACTUATOR und einem ACTUATORINTERFACE dargestellt: Der ACTUATOR verfügt über eine Menge von ACTUATORINTERFACES, für die er bei Bedarf eine Implementierung bereitstellt. Durch diese Trennung zwischen der Beschreibung eines ACTUATORINTERFACES als Schnittstelle und der konkreten Instanz können unterschiedlichen Aktuatoren das gleiche ACTUATORINTERFACE unterschiedlich implementieren. Durch die Komposition von ACTUATORINTERFACES ist außerdem die Menge an Steuerungsoperationen, die ein ACTUATOR ausführen kann, beliebig erweiterbar.

In der Regel definiert ein ACTUATORINTERFACE eine Menge an semantisch verwandten Steuerungsoperationen für einen bestimmten Typ von Aktuator

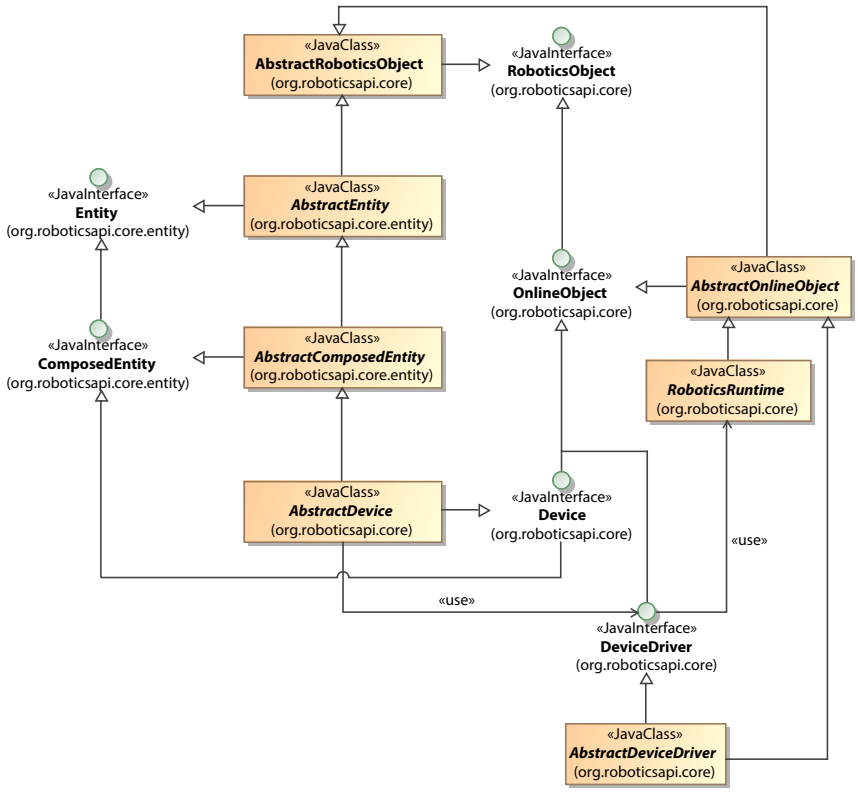


Abbildung 5.7: Das Klassendiagramm zeigt den Zusammenhang zwischen den Schnittstellen DEVICE und ENTITY sowie ihren abstrakten Implementierungen.

(z. B. einen Roboter). Diese Steuerungsoperationen sind als ACTIVITY (vgl. Abschn. 4.3.2) realisiert und werden von den Methoden eines ACTUATOR-INTERFACES erzeugt, parametrisiert und anschließend zurückgegeben. Um eine Steuerungsoperationen auf einem Aktuator auszuführen, muss eine ACTIVITY explizit gestartet werden. Die Klasse ACTIVITY ist Bestandteil des Pakets *org.roboticsapi.activity* und wie folgt definiert:

Definition 5.14. Eine ACTIVITY (Aktivität) ist eine echtzeitkritische Operation, an der mehrere Aktuatoren beteiligt sein können. Zudem verfügt eine ACTIVITY über Informationen, die den Zustand der beteiligten Aktuatoren während oder zumindest nach der Ausführung beschreiben (vgl. [10]).

Intern kapselt eine Aktivität eine Instanz der Klasse COMMAND. Diese kann, wie in Abschnitt 4.3.2 vorgestellt, in einen Graphen von Echtzeitprimitiven übersetzt und anschließend auf einem RCC ausgeführt werden. Zusätzlich verfügt eine ACTIVITY über Metadaten, um eine kontext-abhängige Ausführung zu ermöglichen [10]. Die Struktur einer ACTIVITY ist in Abbildung 5.6 zu erkennen. Der Unterschied zwischen einer ACTIVITY und einem COMMAND ist ausführlich in [10, 18] beschrieben. Im weiteren Verlauf dieser Arbeit werden ausschließlich Aktivitäten verwendet.

Details zur Implementierung

Die oben vorgestellten Konzepte sind in der *Robotics API* hauptsächlich als Schnittstellen realisiert. Dies gilt insbesondere für das Konzept eines DEVICE als mechatronisches Gerät. Daneben existieren in der *Robotics API* abstrakte Klassen, um die elementare Funktionalität der jeweiligen Schnittstelle zu implementieren. Der Zusammenhang zwischen den Schnittstellen und ihren abstrakten Implementierungen ist in Abbildung 5.7 dargestellt.

Dort ist zu erkennen, dass ein DEVICE sowohl ein ONLINEOBJECT als auch ein COMPOSEDENTITY spezialisiert. Dieser Mehrfachvererbung wird durch zwei parallele Vererbungshierarchien Rechnung getragen.

Demzufolge erbt die Klasse ABSTRACTDEVICE direkt von der Klasse ABSTRACTCOMPOSEDENTITY. Daher ist jedes Objekt der Klasse ABSTRACTDEVICE sowohl ein ROBOTICSOBJECT, ein ENTITY, ein COMPOSEDENTITY als auch ein DEVICE. Die noch fehlende Funktionalität eines ONLINEOBJECTS ist im ABSTRACTDEVICE implementiert und stützt sich direkt auf den assoziierten DEVICEDRIVER ab, d. h. der Betriebszustand des Gerätetreibers wird gespiegelt. Das Konzept eines ACTUATORS ist ebenfalls in der *Robotics API* als Schnittstelle realisiert. Sie spezialisiert dabei die Schnittstelle DEVICE. Analog ist die Klasse ABSTRACTACTUATOR eine abstrakte Implementierung der Schnittstelle ACTUATOR und leitet sich von der Klasse ABSTRACTDEVICE ab. Aus Gründen der Übersicht wurde in Abbildung 5.7 darauf verzichtet.

In der zweiten Vererbungshierarchie leitet sich die abstrakte Klasse ABSTRACTONLINEOBJECT ebenfalls von ABSTRACTROBOTICSOBJECT ab und implementiert die Schnittstelle ONLINEOBJECT. Von der Klasse ABSTRACTONLINEOBJECT leiten sich dann die beiden abstrakten Klassen ROBOTICSRUNTIME und ABSTRACTDEVICEDRIVER ab. Konkrete Implementierungen beider Klassen müssen die entsprechende Kommunikation mit der Robotersteuerung bzw. mit dem an der Robotersteuerung angeschlossenen Gerät realisieren.

5.3 GEOMETRISCHE KONZEPTE

Da Roboter in ihren Bewegungsfolgen frei programmierbar sind, können sie flexibel für Fertigungsaufgaben eingesetzt werden. Um diese Bewegungsfolgen zu spezifizieren, ist es notwendig, dass man geometrische Beziehungen zwischen unterschiedlichen Objekten beschreibt. Daher existieren die bereits vorgestellten Konzepte FRAME (vgl. Def. 5.3) und RELATION (vgl. Def. 5.4). Eine Übersicht über die wichtigsten geometrischen Konzepte der *Robotics API* findet sich in Abbildung 5.8. Diese werden – in Verbindung mit den mathematischen Grundlagen – in diesem Abschnitt vorgestellt.

Im Gegensatz zu den üblichen Formalismen (vgl. [63, 277]) haben FRAMES in der *Robotics API* kein festes Referenzkoordinatensystem, sondern bilden einen ungerichteten Graphen. Betrachtet man jedoch zwei über eine RELATION verbundene FRAMES separat, kann der FRAME *from* als Referenzkoordinatensystem des FRAMES *to* interpretiert werden. Folglich kann man die üblichen mathematischen Konventionen verwenden. Formal besteht ein FRAME i demnach aus einem Ursprung O_i und drei normierten, zueinander orthogonalen Basisvektoren $\hat{x}_i$, $\hat{y}_i$ und $\hat{z}_i$ [277]. Sie bilden die Orthonormalbasis eines dreidimensionalen euklidischen Vektorraums $\mathbb{R}^3$.

Exkurs: Mathematische Beschreibung von Translationen

Die Position des Koordinatensystems i bzw. genauer seines Ursprungs O_i kann relativ zu einem anderen Koordinatensystem j durch einen (3×1)-Vektor beschrieben werden:

$$ {}^{j}\mathbf{p}_i = \begin{pmatrix} {}^{j}p_i^x \\ {}^{j}p_i^x \\ {}^{j}p_i^x \end{pmatrix} \tag{5.1} $$

Dieser Vektor wird *Positionsvektor* genannt und beschreibt die Verschiebung bzw. *Translation* von O_i oder eines mit O_i assoziierten Körpers in Bezug zu dem (Referenz-)Koordinatensystem j. Die Komponenten ${}^{j}p_i^x$, ${}^{j}p_i^x$ und ${}^{j}p_i^x$ des Vektors repräsentieren dabei die kartesischen Koordinaten von O_i

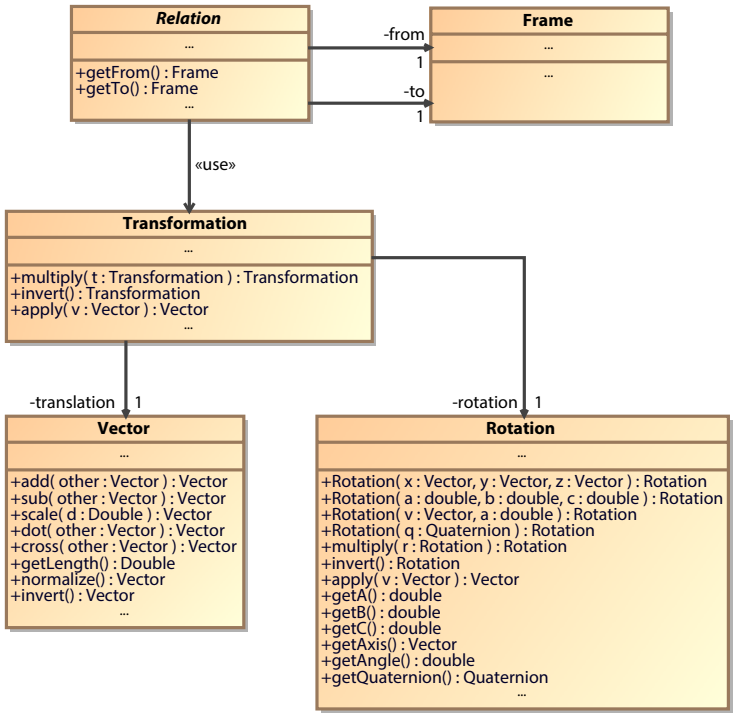


Abbildung 5.8: Ein FRAME repräsentiert ein Koordinatensystem und ist über RELATIONS mit anderen FRAMES verbunden. Durch eine TRANSFORMATION wird die Position und Orientierung bestimmt.

in Koordinatensystem j. In der *Robotics API* wird ein Positionsvektor durch die Klasse VECTOR abgebildet und ist wie folgt definiert:

Definition 5.15. Ein VECTOR (Vektor) ist ein 3-Tupel reeller Zahlen und repräsentiert einen (3×1)-Positionsvektor (vgl. [10]).

Die Klasse besitzt mehrere Methoden, um die gebräuchlichsten Rechenoperation zu implementieren, die auf Vektoren definiert sind (z. B. Addition, Skalarprodukt, Kreuzprodukt, Normierung). In der *Robotics API* wird die Klasse durchgängig verwendet, um die Translation eines Koordinatensystems zu beschreiben.

Oft reicht es aber nicht aus, nur die Position eines Objekts durch die Translation seines Koordinatensystems zu einem Bezugssystem zu beschreiben. Stattdessen muss zusätzlich eine Orientierung angegeben werden, um ein Objekt korrekt im Raum auszurichten. Die Orientierung kann durch eine Rotation des Objekts bzw. seines Koordinatensystems zu einem Bezugssystem beschrieben werden.

Exkurs: Mathematische Beschreibung von Rotationen

Mathematisch wird die Orientierung eines Koordinatensystems i zu einem anderen Koordinatensystem j durch die Basisvektoren beider Koordinatensysteme beschrieben [63, 277]. Dabei werden die Basisvektoren $\hat{x}_i$, $\hat{y}_i$ und $\hat{z}_i$ von Koordinatensystem i bezüglich den Basisvektoren $\hat{x}_j$, $\hat{y}_j$ und $\hat{z}_j$ von Koordinatensystem j ausgedrückt. Dies lässt sich kompakt als (3×3)-Matrix beschreiben:

$$ {}^{j}\mathbf{R}_i = \begin{pmatrix} \hat{x}_i \cdot \hat{x}_j & \hat{y}_i \cdot \hat{x}_j & \hat{z}_i \cdot \hat{x}_j \\ \hat{x}_i \cdot \hat{y}_j & \hat{y}_i \cdot \hat{y}_j & \hat{z}_i \cdot \hat{y}_j \\ \hat{x}_i \cdot \hat{z}_j & \hat{y}_i \cdot \hat{z}_j & \hat{z}_i \cdot \hat{z}_j \end{pmatrix} . \tag{5.2} $$

Die Komponenten von ${}^{j}\mathbf{R}_{i}$ sind das Skalarprodukt der Basisvektoren beider Koordinatensysteme. Die Matrix wird als Dreh- oder *Rotationsmatrix* bezeichnet.

Falls man die Matrix aus Gleichung (5.2) transponiert, so erhält man eine Matrix, die die Rotation von Koordinatensystem j um die Basisvektoren von Koordinatensystem i beschreibt. Da außerdem für alle Rotationsmatrizen die transponierte Matrix und die inverse Matrix gleich sind, gilt

$$ {}^{i}\mathbf{R}_{j} = ({}^{j}\mathbf{R}_{i})^{-1} = ({}^{j}\mathbf{R}_{i})^{T} . \tag{5.3} $$

Des weiteren können Drehungen durch die Multiplikation von Rotationsmatrizen kombiniert werden. Die Drehung eines Koordinatensystems i um ein Koordinatensystem k kann ausgedrückt werden als

$$ {}^{k}\mathbf{R}_{i} = {}^{k}\mathbf{R}_{j}\,{}^{j}\mathbf{R}_{i} . \tag{5.4} $$

In der *Robotics API* wird die Drehung zweier Koordinatensysteme durch die Klasse ROTATION abgebildet, die wie folgt definiert ist:

Definition 5.16. Die Klasse ROTATION repräsentiert eine Drehung im dreidimensionalen Raum und wird für Drehung im $\mathbb{R}^3$ durch eine (3×3)-Rotationsmatrix beschrieben (vgl. [10]).

Die Klasse ist intern als (3×3)-Rotationsmatrix implementiert und besitzt Methoden, um die oben beschriebenen Matrixoperationen auszuführen.

Exkurs: Eulersche Winkel

Drehungen können zudem mit anderen Formalismen beschrieben werden. In der *Robotics API* werden u. a. *Eulersche Winkel* in der (Z, Y′, X″)-Konvention verwendet. Gemäß dieser Konvention wird die Drehung eines Koordinatensystems i gegenüber einem Koordinatensystem j durch drei Winkel α, β und γ, die Eulerschen Winkel, gedreht. Dabei sind zu Beginn beide Koordinatensysteme deckungsgleich. Dann wird begonnen, das Koordinatensystem i um einen Winkel α um die Achse $\hat{\mathbf{z}}_{i}$ zu drehen. Die zweite Rotation des Koordinatensystems um den Winkel β wird um die bereits gedrehte Achse $\hat{\mathbf{y}}_{i}$, bezeichnet als $\hat{\mathbf{y}}'_{i}$, durchgeführt. Schließlich wird noch eine Rotation um den Winkel γ durchgeführt und zwar um die mittlerweile zweifach gedrehte Achse $\hat{\mathbf{x}}_{i}$, bezeichnet als $\hat{\mathbf{x}}''_{i}$. Die resultierende Rotationsmatrix lautet:

$$ {}^{j}\mathbf{R}_{i}(\alpha,\beta,\gamma) = \begin{pmatrix} c_\alpha c_\beta & c_\alpha s_\beta s_\gamma - s_\alpha c_\gamma & c_\alpha s_\beta c_\gamma + s_\alpha s_\gamma \\ s_\alpha c_\beta & s_\alpha s_\beta s_\gamma + c_\alpha c_\gamma & s_\alpha s_\beta c_\gamma - c_\alpha s_\gamma \\ -s_\beta & c_\beta s_\gamma & c_\beta c_\gamma \end{pmatrix} \tag{5.5} $$

mit $c_\alpha := \cos\alpha$ und $s_\alpha := \sin\alpha$. Aus der Rotationsmatrix lassen sich zudem die drei Eulerschen Winkel wieder ableiten (vgl. Craig [63]).

Insgesamt gibt es 12 mögliche Konventionen für Rotationen um Eulersche Winkel [63, S. 374 ff.]. Neben diesen 12 Konventionen, die jeweils um bereits gedrehte Koordinatensystem rotieren (z. B. Z, Y′, X″), gibt es 12 weitere Konventionen, die um stationäre Koordinatensystem rotieren. Verbreitet ist die (X, Y, Z)-Konvention. Dabei sind zu Beginn beide Koordinatensysteme wieder deckungsgleich. Anschließend wird begonnen, das Koordinatensystem i erst um γ um die Achse $\hat{\mathbf{x}}_{j}$, dann um β um die Achse $\hat{\mathbf{y}}_{j}$ und letztendlich um α um die Achse $\hat{\mathbf{z}}_{j}$ zu drehen. Diese Winkel werden als *Roll* (γ, dt.: Roll-Winkel), *Pitch* (β, dt.: Nick-Winkel) und *Yaw* (α, dt.: Gier-Winkel) bezeichnet und werden oft in der Luftfahrt verwendet. Sie sind identisch mit den Winkeln der (Z, Y′,X″)-Konvention [63]. Gleiches gilt für die Rotationsmatrizen.

Die *Robotics API* unterstützt die Konstruktion einer Instanz der Klasse ROTATION durch die Angabe dreier Winkel. Dabei werden jedoch die Bezeichnungen A für α, B für β und C für γ verwendet. Die interne Matrix wird analog zu Gleichung (5.5) erstellt. Neben der Beschreibung einer Drehung durch drei Winkel gibt es noch weitere Formalismen. Dazu gehören bspw. *Quaternionen* und die *Axis-Angle*-Darstellung, die beide von der *Robotics API* unterstützt werden. Für eine detaillierte Beschreibung sei auf die Literatur [63, 277] verwiesen.

Exkurs: Homogene Transformationen

Nachdem bisher die Translation und Rotation getrennt betrachtet wurden, werden sie durch homogene Transformationen miteinander verknüpft. Das bedeutet, dass jeder Vektor ${}^{i}\mathbf{r}$, der relativ zu einem Koordinatensystem i definiert ist, in einen Vektor ${}^{j}\mathbf{r}$ transformiert werden kann, der relativ zu einem Koordinatensystem j definiert ist. Sind die Translation ${}^{j}\mathbf{p}_{i}$ aus Gleichung (5.1) und die Rotation ${}^{j}\mathbf{R}_{i}$ aus Gleichung (5.2) bekannt, kann die Transformation beschrieben werden als

$$ {}^{j}\mathbf{r} = {}^{j}\mathbf{R}_{i}\,{}^{i}\mathbf{r} + {}^{j}\mathbf{p}_{i}\,. $$

Analog zu [277] kann diese Gleichung umgeschrieben werden zu

$$ \begin{pmatrix} {}^{j}\mathbf{r} \\ 1 \end{pmatrix} = {}^{j}\mathbf{T}_{i} \begin{pmatrix} {}^{i}\mathbf{r} \\ 1 \end{pmatrix}, \tag{5.6} $$

wobei die (4×4)-Matrix ${}^{j}\mathbf{T}_{i}$ auch *homogene Transformationsmatrix* genannt wird und definiert ist als

$$ {}^{j}\mathbf{T}_{i} = \begin{pmatrix} {}^{j}\mathbf{R}_{i} & {}^{j}\mathbf{p}_{i} \\ 0^{T} & 1 \end{pmatrix}. \tag{5.7} $$

Die (4×1)-Vektoren $({}^{i}\mathbf{r}\ 1)^{T}$ und $({}^{j}\mathbf{r}\ 1)^{T}$ werden *homogene Koordinaten* genannt. Homogene Transformationsmatrizen haben die Eigenschaft, dass sie durch Matrixmultiplikation verknüpft werden können; d. h. es gilt

$$ {}^{k}\mathbf{T}_{i} = {}^{k}\mathbf{T}_{j}\,{}^{j}\mathbf{T}_{i}\,. \tag{5.8} $$

Im Gegensatz zu Rotationsmatrizen kann die invertierte Matrix jedoch nicht durch Transponieren berechnet werden. Analog zu Gleichung (5.7), ist das Inverse einer homogenen Transformationsmatrix definiert als

$$ {}^{i}\mathbf{T}_{j} = ({}^{j}\mathbf{T}_{i})^{-1} = \begin{pmatrix} ({}^{j}\mathbf{R}_{i})^{T} & -({}^{j}\mathbf{R}_{i})^{T}\,{}^{j}\mathbf{p}_{i} \\ 0^{T} & 1 \end{pmatrix}. \tag{5.9} $$

Die inverse Matrix $({}^{j}\mathbf{T}_{i})^{-1}$ transformiert Vektoren von Koordinatensystem j in das Koordinatensystem i.

In der *Robotics API* werden homogene Transformationen in Form der Klasse TRANSFORMATION verwendet (vgl. Abb. 5.8), um die Verschiebung und Drehung zwischen zwei FRAMES zu spezifizieren.

Definition 5.17. Eine TRANSFORMATION beschreibt eine geometrische Verschiebung vollständig und besteht aus einem translatorischen Anteil, repräsentiert durch einen VECTOR, und einem rotatorischen Anteil, repräsentiert durch eine ROTATION (vgl. [10]).

Wie Abbildung 5.8 zeigt, besteht eine Instanz intern aus einem VECTOR und einer ROTATION. Nach außen verhält sich eine TRANSFORMATION jedoch wie

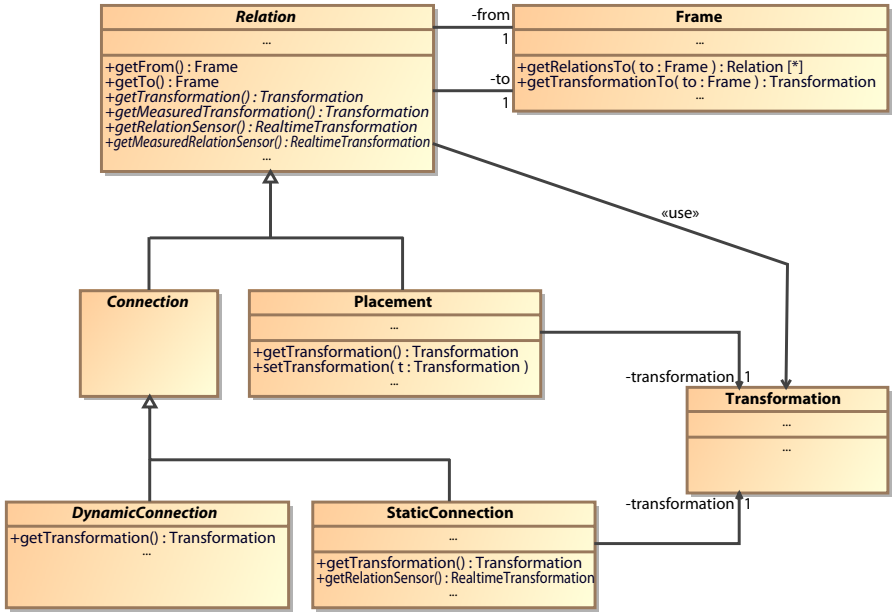


Abbildung 5.9: Der konkrete Typ einer RELATION definiert, in welcher Beziehung zwei FRAMES stehen und wie die Transformation zwischen diesen beiden FRAMES bestimmt wird.

eine homogene Transformationsmatrix. Daher stehen Methoden zur Multiplikation und zum Invertieren zur Verfügung.

Details zur Implementierung

Die oben vorgestellten mathematischen Konzepte VECTOR, ROTATION und TRANSFORMATION wurden als Klassen des Pakets *org.roboticsapi.world* in der *Robotics API* implementiert. Sie können dementsprechend direkt verwendet werden; daher ist eine weitere Spezialisierung nicht notwendig.

5.4 ABBILDUNG GEOMETRISCHER BEZIEHUNGEN

In der *Robotics API* wird eine TRANSFORMATION verwendet, um die Position und Ausrichtung zweier FRAMES in einer RELATION geometrisch zu beschreiben (vgl. Abb. 5.9). Formal beschreibt sie die Translation und Rotation des FRAMES *to* in Bezug zu FRAME *from*. Durch das Invertieren der homogenen Transformationsmatrix kann die Transformation in die umgekehrte Richtung berechnet werden. Daher bestimmt die Unterscheidung zwischen den beiden FRAMES einer RELATION nur die Richtung der mathematischen Transformation. Ansonsten ist eine RELATION ungerichtet und erlaubt eine Navigation in beide Richtungen (vgl. Assoziationen in Abb. 5.9).

Die abstrakte Klasse RELATION hat keine Assoziation zur Klasse TRANSFORMATION, d. h. es ist keine Instanz einer TRANSFORMATION direkt als Attribut vorhanden. Vielmehr definiert die Klasse RELATION zwei abstrakte Methoden, um die aktuelle Transformation zu bestimmen. Dabei gibt die Methode *getTransformation()* die aktuell *kommandierte* Transformation zwischen den beiden FRAMES zurück. Kommandiert bedeutet in diesem Zusammenhang, dass es sich um eine ideale bzw. gewünschte Transformation handelt. Die Methode *getMeasuredTransformation()* dagegen gibt die aktuell *gemessene* Transformation zwischen den beiden FRAMES zurück.

Bei statischen Transformationen, bei denen sich die Translation und Rotation während der Laufzeit nicht ändern, sind die beiden Werte für gewöhnlich identisch. Bei einer dynamischen Transformation dagegen, die z. B.

durch eine Roboterachse bestimmt wird, kann sich jedoch ein Unterschied zwischen der real gemessenen Transformationen, die durch den Encoder der Roboterachse bestimmt wird, und der gewünschten Transformationen, die durch die Applikation vorgegeben wird, ergeben.

Neben den beiden oben beschriebenen Methoden, die den aktuellen Wert der kommandierten bzw. gemessenen Transformation abfragen, gibt es weitere Methoden, die einen entsprechenden RealtimeValue zurückgeben. Somit repräsentiert eine RealtimeTransformation eine Datenquelle für die aktuelle Transformation einer Relation in der Echtzeitrobotersteuerung. Der Wert dieses Sensors ist bei statischen Transformationen durchgehend konstant, während er sich bei dynamischen Transformation über die Zeit hinweg verändern kann.

Um den unterschiedlichen Beziehungen zwischen Frames gerecht zu werden, gibt es verschiedene Ausprägungen der Klasse Relation. Die wichtigsten davon sind in Abbildung 5.9 dargestellt. Durch die Spezialisierung einer Relation werden folgende zwei Aspekte implementiert:

1. Die Spezialisierung definiert, wie sowohl die kommandierte als auch die gemessene Transformation zwischen den über die Relation verbundenen Frames bestimmt wird. Dazu werden die oben vorgestellten abstrakten Methoden implementiert.

2. Die konkrete Ausprägung einer Relation bestimmt die Semantik der zwischen den beiden Frames bestehenden Verbindung, d. h. sie beschreibt bestimmte Eigenschaften, die in dieser Verbindung gelten.

Eine Spezialisierung der Klasse Relation ist die Klasse Connection, die wie folgt definiert ist:

Definition 5.18. Eine Connection ist eine Relation, die eine dauerhafte Verbindung zwischen zwei Frames definiert. Diese Verbindung besteht während der gesamten Laufzeit der Roboterapplikation (vgl. [10]).

Die Klasse Connection adressiert somit nur den zweiten Aspekt, d. h. sie definiert ausschließlich eine semantische Eigenschaft. Daher ist die Klasse abstrakt und muss entsprechend spezialisiert werden, um auch den ersten Aspekt zu adressieren. Die Information, dass eine Relation zwischen Frames dauerhaft besteht, kann jedoch essentiell sein und bspw. in Berechnungen ausgenutzt werden. Ein typisches Beispiel für eine Connection ist eine mechanische Verbindung.

Um den ersten der beiden obigen Aspekte zu adressieren gibt es mit den Klassen StaticConnection und DynamicConnection (vgl. Abb. 5.9) zwei Spezialisierungen der Klasse Connection, die sich bei der Bestimmung der Transformation unterscheiden. Die Klasse StaticConnection ist wie folgt definiert:

Definition 5.19. Eine StaticConnection ist eine Connection, deren Transformation konstant ist, d. h. der Betrag der geometrischen Verschiebung zwischen den beiden Frames ändert sich während der Laufzeit der Roboterapplikation nicht (vgl. [10]).

Beispiel: Statische Verbindungen

Dementsprechend verfügt eine Klasse StaticConnection über eine direkte Assoziation zur Klasse Transformation (vgl. Abb. 5.9) und hat folglich ein Attribut *transformation*. Diese Art einer Connection wird verwendet, um eine feste mechanische Verbindung wie z. B. die Verankerung eines Roboters oder die Befestigung eines Werkzeuges am Roboterflansch zu modellieren.

Auch um FRAMES in Beziehung zu setzen, die Bestandteil eines PHYSICALOBJECTS sind, sollte eine STATICCONNECTION gewählt werden. Von der Klasse kann zudem abgeleitet werden, um eigene Geräte- oder applikationsspezifische Spezialisierungen zu implementieren.

Während sich die Transformation einer STATICCONNECTION nicht über die Zeit verändert, ist genau das Gegenteil bei einer DYNAMICCONNECTION der Fall. Daher ist sie wie folgt definiert:

Definition 5.20. Eine DYNAMICCONNECTION ist eine CONNECTION, deren Transformation dynamisch ist, d. h. der Betrag der geometrische Verschiebung zwischen den beiden FRAMES kann sich während der Laufzeit der Roboterapplikation ändern (vgl. [10]).

Dementsprechend werden DYNAMICCONNECTIONS verwendet, um dynamische Beziehungen zwischen FRAMES zu beschrieben. Typischerweise werden solche dynamischen Beziehungen durch Aktuatoren definiert, d. h. durch eine Linearachse, die ihren Schlitten verfährt, oder eine Roboterachse, die sich dreht. Dabei verdreht die Achse zwei aufeinanderfolgende Strukturbauteile des Roboters und verändert so deren Position und Ausrichtung zueinander.

Beispiel: Dynamische Verbindungen

Die Klasse DYNAMICCONNECTION ist abstrakt, d. h. konkrete Unterklassen müssen implementieren, wie sich die TRANSFORMATION über die Zeit verhält. Alle RELATIONS, die eine dauerhaft Verbindung mit veränderlicher Transformation spezifizieren, sollten von der Klasse DYNAMICCONNECTION abgeleitet werden. Für Aktuatoren, die über Achsen bewegt werden (z. B. ein Roboterarm), existieren bereits spezielle Implementierungen, die sowohl rotatorische als auch translatorische Achsen abbilden. Das bedeutet, dass die Bewegung der Achse als relative Bewegung zweier FRAMES zueinander abgebildet wird.

Eine Unterscheidung zwischen kommandierter und gemessener Transformation kommt vor allem bei einer DYNAMICCONNECTION zu tragen. Da sie in der Regel von einem Aktuator abhängig ist, können Unterschiede zwischen dem von z. B. einem Encoder gemessenen Ist-Wert und dem kommandierten Soll-Wert auftreten. Dies spiegelt sich im Fall der DYNAMICCONNECTION in unterschiedlichen Werten der beiden Transformationen wider. Jede konkrete Unterklasse muss das Verhalten gerätespezifisch implementieren.

Neben den oben vorgestellten CONNECTIONS, die eine dauerhafte Beziehung zwischen zwei FRAMES darstellen, existiert mit der Klasse PLACEMENT eine *kurzlebige* Spezialisierung einer RELATION (vgl. Abb. 5.9). Sie ist wie folgt definiert:

Definition 5.21. Ein PLACEMENT ist eine RELATION, die eine lose Verbindung zwischen zwei FRAMES definiert. Diese Verbindungen können während der Laufzeit einer Roboterapplikation erzeugt, verändert und aufgelöst werden (vgl. [10]).

Ein PLACEMENT wird verwendet, um einen FRAME kurzzeitig, d. h. für eine spezielle Aufgabe, mit einem anderen FRAME zu verbinden. Es kann, wie der Name suggeriert, benutzt werden, um Objekte (in Form von PHYSICALOBJECTS) über einen ihrer FRAMES in der Welt bzw. in der Roboterzelle zu *platzieren*. Ein solche Platzierung ist jedoch nicht dauerhaft, wie folgendes Beispiel zeigt. Dabei liegt zu Beginn einer Aufgabe ein Werkstück in einem entsprechenden Werkstückträger. Diese Beziehung wird in der *Robotics API* durch ein PLACEMENT abgebildet. Sobald das Werkstück von einem an einem Roboterflansch montierten Greifer gegriffen wird, muss die Beziehung

Beispiel: Platzierung von Objekten

zum Werkstückträger aufgelöst und eine PLACEMENT zum Greifer (bzw. zu einem FRAME des Greifer) erstellt werden. Das Werkstück bewegt sich in Folge dessen mit dem Greifer und dem Roboter zu einer Zielposition. Dort wird die Beziehung zum Greifer aufgelöst und eine neues PLACEMENT wird an der Zielposition erstellt.

Das Erstellen und Auflösen von PLACEMENTS während einer Aufgabe muss entweder explizit programmiert werden oder kann das Resultat einer Roboter- oder Werkzeugaktivität sein (z. B. einer komplexen Greifoperation). Analog zu einer STATICCONNECTION verfügt ein PLACEMENT über eine direkte Assoziation zur Klasse TRANSFORMATION (vgl. Abb. 5.9). Während diese Transformation bei einer STATICCONNECTION fest ist, kann sie bei einem PLACEMENT über die Zeit verändert werden. Von der Klasse PLACEMENT kann zudem abgeleitet werden, um eigene Geräte- oder applikationsspezifische Spezialisierungen zu implementieren.

Neben den oben vorgestellten RELATIONS gibt es zusätzlich die Klasse TEMPORARYRELATION, die zum Erzeugen von Hilfspunkten verwendet wird. Sie ist wie folgt definiert:

Definition 5.22. Eine TEMPORARYRELATION ist eine kurzlebige RELATION, die ein Hilfskoordinatensystem bzgl. eines FRAMES definiert.

Analog zu einem PLACEMENT hat eine TEMPORARYRELATION ebenfalls eine direkte Assoziation einer TRANSFORMATION. Sie verfügt aber über keine spezielle Semantik und wird daher nur für Hilfspunkte verwendet.

Durch die verschiedenen Ausprägungen der Klasse RELATION können zwei FRAMES vielfältig miteinander in Bezug gesetzt werden. Dabei ist dieser Bezug nicht nur geometrisch, sondern hat eine Semantik. Diese Semantik lässt sich von den FRAMES auf Entitäten bzw. PHYSICALOBJECTS übertragen. Dazu können sowohl FRAMES als auch RELATIONS über ihre *parent* Assoziation mit einer Entität verknüpf werden, zu der sie logisch gehören. Zum Beispiel verfügt ein LINK, d. h. das Strukturbauteil eines Manipulators, über einen *base* FRAME und eine RELATION (vgl. Abb. 5.3). Diese beiden Entitäten wiederum sind durch ihre *parent* Assoziation jeweils Bestandteil des LINKS.

Während FRAMES einer Entität über eine CONNECTION verbunden sein sollten, können FRAMES unterschiedlicher Entitäten beliebig verbunden sein. Die durch eine RELATION definierte semantische Beziehung zwischen zwei FRAMES kann auf die übergeordneten PHYSICALOBJECTS übertragen werden. Dadurch lässt sich ausdrücken, ob zwei Objekte fest miteinander verbunden sind oder ob das eine Objekt nur auf das andere gestellt wurde. Es ist wichtig anzumerken, dass bei dem Übertragen der Semantik einer RELATION auf PHYSICALOBJECTS die Richtung der RELATION, d. h. die Wahl des *from* und *to* FRAMES, eine Rolle spielen kann. Dies sollte als Teil der Semantik einer RELATION wohldefiniert werden.

Der Graph, der von FRAMES (Knoten) und RELATIONS (Kanten) aufgespannt wird, kann ausgehend von einem FRAME durchlaufen werden. Durch die beidseitig navigierbaren RELATIONS kann jeder FRAME, der Bestandteil des Graphen ist, erreicht werden. Um zu bestimmen, ob zwei FRAMES verbunden sind (vgl. Def. 5.5), spezifiziert die Klasse FRAME eine Methode:

- Die Methode *getRelationsTo(Frame, Predicate)* sucht den kürzesten Pfad zwischen dem FRAME, auf dem die Methode aufgerufen wurde, und einem übergebenen FRAME. Falls kein Pfad gefunden wurde, wird ein leerer Pfad zurückgegeben. Durch das übergebene Prädikat können die RELATIONS, die bei der Suche berücksichtigt werden sollen, eingeschränkt werden.

Ein leerer Pfad bedeutet, dass die beiden FRAMES nicht miteinander verbunden sind. Das Prädikat ermöglicht es, bestimmte Typen von RELATIONS zu filter. So können bspw. nur FRAMES betrachtet werden, die gegenseitig über eine STATICCONNECTION verbunden sind. Aufbauend darauf, kann mithilfe der Methode *getTransformationTo(...)* die Transformation zwischen zwei beliebigen, verbundenen FRAMES berechnet werden.

Durch die Graphstruktur gibt es in der *Robotics API* keinen expliziten Ursprung und demnach auch keine Menge von Basisvektoren [10]. Vielmehr ist die Wahl eines FRAMES als Ursprung (engl.: *origin*) eine applikationsspezifische Entscheidung. In der *Robotics API* ist dieses Konzept als WORLDORIGIN integriert:

Definition 5.23. Die Klasse WORLDORIGIN ist ein Singleton und repräsentiert einen global eindeutigen FRAME (vgl. [10]).

Im weiteren Verlauf hat jede Roboterzelle einen ausgezeichneten FRAME, der als Ursprung interpretiert werden kann und mit dem alle anderen relevanten FRAMES der Roboterzelle verbunden sind.

Details zur Implementierung

Die oben beschriebenen RELATIONS sind in der *Robotics API* im Paket *org.roboticsapi.world* implementiert. Sowohl die Klassen STATICCONNECTION als auch PLACEMENT können direkt in einer Applikation verwendet werden. Die Klasse TEMPORARYRELATION kann nicht extern instanziiert werden. Stattdessen können Hilfspunkte über Methoden der Klasse FRAME erstellt werden. Diese Hilfspunkte sind dann über eine TEMPORARYRELATION mit dem FRAME verbunden, auf dem sie erstellt wurden. Die Klasse DYNAMICCONNECTION ist abstrakt und kann daher nicht instanziiert werden. Vielmehr definiert ein Aktuator eine konkrete DYNAMICCONNECTION und beschreibt damit, wie sich die geometrische Verbindung zweier FRAMES über die Zeit verhält.

Basierend auf den Klassen ENTITY, FRAME und RELATION definiert das Paket *org.roboticsapi.world.util* Hilfsklassen, um spezielle Datenstrukturen und Algorithmen zu implementieren. Die Klasse ENTITYSEARCH bietet statische Methoden an, um ausgehend von einem FRAME die Menge aller FRAMES, RELATIONS und ENTITIES generisch zu durchsuchen. Dabei kann man zwischen einer Breiten- und einer Tiefensuche auswählen. Dadurch ist es einfach Abfragen auf dem objektorientierten Modell einer Roboterzelle zu stellen. So können bspw. alle am Roboterflansch montierten Werkzeuge oder gegriffenen Gegenstände erfasst werden. Analog ist es möglich herauszufinden, ob sich ein Gegenstand in einer Kiste befindet oder von einem Greifer gehalten wird.

Die Klasse FRAMEGRAPH baut ausgehend von einem FRAME einen zusammenhängenden Graphen auf, der alle verbundenen FRAMES enthält. Zudem informiert der FRAMEGRAPH entsprechend dem Observer-Pattern [97] über Veränderungen des Graphen, d. h. über neu hinzugefügte bzw. gelöschte FRAMES. Basierend auf einem FRAMEGRAPH baut die Klasse FRAMETREE einen Spannbaum auf. Der Spannbaum enthält alle Knoten (d. h. alle FRAMES) des zusammenhängenden Graphen und repräsentiert diesen als Baum. Diese Funktionalität wird u. a. benötigt, um FRAMES in einer hierarchischen Struktur darzustellen. Hierbei existiert ebenfalls die Möglichkeit, sich über Veränderungen des Spannbaums benachrichtigen zu lassen. Durch den FRAMEGRAPH bzw. den FRAMETREE kann ein gezielter Ausschnitt des Modells einfach beobachtet werden. Dies wird bspw. für eine Visualisierung oder Kollisionserkennung verwendet (vgl. Abschn. 6.5).

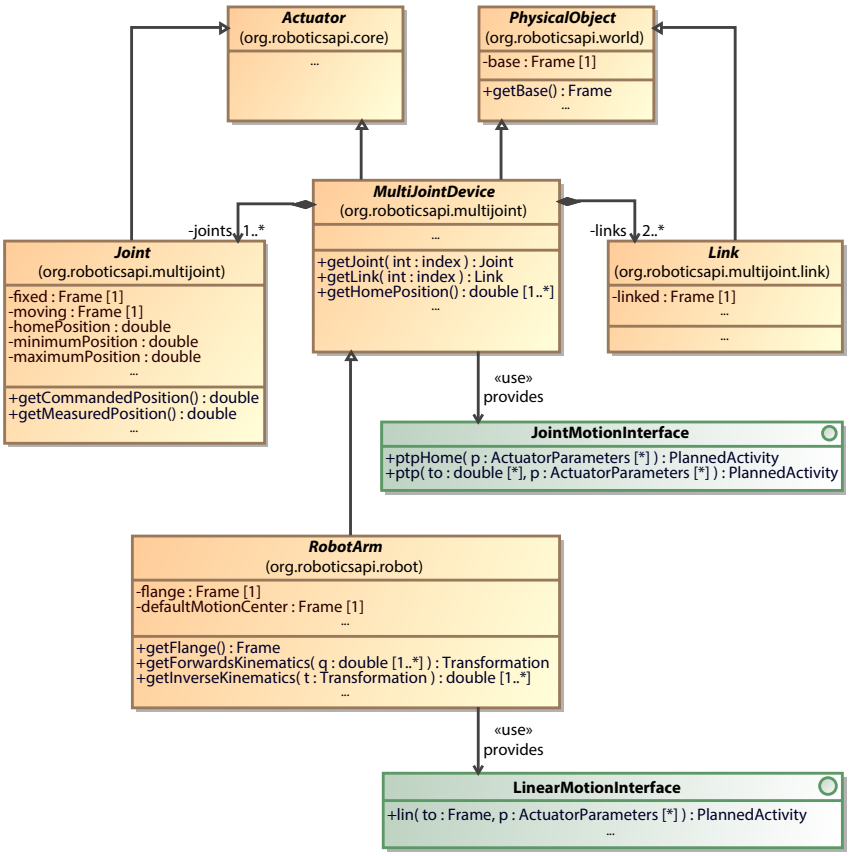


Abbildung 5.10: Ein MultiJointDevice ist die Basisklasse für alle Geräte, die aus steuerbaren Gelenken bestehen. Eine Spezialisierung stellt der RobotArm dar, der eine serielle Kinematik hat.

5.5 MANIPULATOREN UND WERKZEUGE

Eine wichtige Kategorie mechatronischer Aktuatoren stellen Industrieroboter dar. Sie sind – wie der Name schon besagt – elementarer Bestandteil einer Roboterzelle. Durch ihre freie Programmierbarkeit und flexible Bestückung mit Werkzeugen sind sie als universelle Handhabungsgeräte für eine Vielzahl industrieller Prozesse geeignet. Ein Industrieroboter besteht für gewöhnlich aus einen Manipulator (d. h. einem beweglichen Roboterarm), einer programmierbaren Steuerung und einem oder mehreren Werkzeugen, die am Manipulator montiert sind.

Ein Manipulator kann nach Craig [63] wie folgt klassifiziert werden:

> "A manipulator may be thought of as a set of bodies connected in a chain by joints. These bodies are called links. Joints form a connection between a neighboring pair of links." [63, S. 62]

Folglich besteht ein Manipulator aus einer Menge von angetriebenen Achsen bzw. Gelenken (engl.: *joints*), die über starre Körper bzw. Glieder (engl.: *links*) miteinander verbunden sind. Dabei unterscheidet man zwischen rotatorischen Achsen (Drehgelenken) und translatorischen Achsen (Lineargelenken). Die Achsen sind steuerbar und erlauben somit eine relative Bewegung benachbarter Verbindungsbauteile.

Allgemein ist in der *Robotics API* ein mechatronisches Gerät mit mehreren angetriebenen Achsen wie folgt definiert:

Definition 5.24. Ein MultiJointDevice ist ein Actuator, der ein oder mehrere steuerbare Gelenke besitzt.

Die Klasse MultiJointDevice ist die Basisklasse für alle Geräte, die aus angetriebenen und damit steuerbaren Gelenken bestehen (vgl. Abb. 5.10). Das MultiJointDevice erbt als steuerbares Gerät von der Klasse Actuator und, da es eine konkrete physische Ausprägung hat, gleichzeitig auch von PhysicalObject. Die Klasse macht keine Einschränkung über den mechanischen Aufbau, d. h. die Verbindung der Gelenke und Glieder. Jedoch beeinflusst der Aufbau eines MultiJointDevices seine Eigenschaften in Bezug auf die möglichen Freiheitsgrade, den erreichbaren Arbeitsraum, die Traglast, die Geschwindigkeit und die Genauigkeit [63, S. 230 ff.]. Daher sollte die Struktur eines MultiJointDevices in der Regel von der jeweiligen Aufgabe abhängig sein und geeignet gewählt werden.

Jedes MultiJointDevice hat eine Grundstellung, die als *home position* bezeichnet wird und sich aus den Grundstellungen der einzelnen Achsen ableitet. In der Regel bietet jedes MultiJointDevice ein ActuatorInterface an, das es erlaubt, achsspezifische Bewegungen zu definieren. Das in Abbildung 5.10 dargestellte JointPtpInterface besitzt zwei Methoden, um das Geräte an eine beliebige Gelenkstellung bzw. an seine Grundstellung zu verfahren. Dabei wird jeweils eine Activity zurückgegeben, die direkt ausgeführt oder mit anderen Aktivitäten verknüpft werden kann.

Joints & Links

Ein einzelnes Gelenk der Klasse MultiJointDevice wird durch einen Joint repräsentiert, der als Bestandteil des Pakets *org.roboticsapi.multijoint* folgendermaßen definiert ist:

Definition 5.25. Ein Joint ist ein Actuator und repräsentiert entweder ein rotatorisches oder translatorisches Gelenk.

Die Position eines Gelenks wird durch eine Gleitkommazahl dargestellt und zeigt bei rotatorischen Gelenken den aktuellen Drehwinkel in Radianten an, d. h. eine Winkelstellung gegenüber einer definierten Nullstellung. Bei translatorischen Gelenken wird damit die aktuelle Vorschublänge in Metern, d. h. eine Verschiebung gegenüber einer Nullstellung, angegeben. Die Position ist durch eine fest obere und untere Grenze eingeschränkt, die als *minimum position* und *maximum position* bezeichnet werden (vgl. Abb. 5.10). Außerdem hat jedes Gelenk eine eigene, ebenfalls als *home position* bezeichnete, Grundstellung. Die aktuelle Position kann durch zwei Methoden erfragt werden, die jeweils die kommandierte bzw. die gemessene Position zurückgeben.

Darüber hinaus verfügt jeder Joint über zwei Frames, die über eine DynamicConnection (vgl. Abschn. 5.4) verbunden sind und durch eine Bewegung gegeneinander verdreht bzw. verschoben werden (vgl. Abb. 5.10). Dabei stellt der *fixed* Frame das feste Bezugssystem dar. Dementsprechend wird der *moving* Frame bei einem rotatorischen Joint um die z-Achse des Bezugssystems gedreht. Analog wird er bei einem translatorischen Joint entlang der z-Achse verschoben. Über diese beiden Frames sind zwei Glieder mit dem Joint verbunden. Folglich verändert durch eine Bewegung das mit dem *moving* Frame verbundene Glied seine relative Position gegenüber dem mit dem *fixed* Frame verbundenen Glied.

Ein Glied oder Strukturbauteil eines MultiJointDevices wird durch die Klasse Link aus dem Paket *org.roboticsapi.multijoint* repräsentiert und ist in Anlehnung an Craig [63, S. 64] folgendermaßen definiert:

Definition 5.26. Ein Link (Glied) ist als starrer Körper ein PhysicalObject und verbindet zwei Joints miteinander. Abweichend davon ist ein Link,

der am Beginn oder Ende einer Abfolge von LINKS und JOINTS steht, nur mit einem JOINT verbunden.

Analog zu einem JOINT verfügt auch ein LINK über zwei FRAMES, die jedoch fest über eine STATICCONNECTION verbunden sind. In der Regel ist der *linked* FRAME eines LINKS identisch mit dem *fixed* FRAME eines JOINTS. Analog ist der *base* FRAME eines LINKS mit dem *moving* FRAME eines JOINTS identisch. Durch diese Verknüpfung von JOINTS und LINKS ergibt sich eine kinematische Kette und dementsprechend der Aufbau eines MULTIJOINTDEVICES (vgl. [63, S. 62–67]).

In der Literatur (vgl. [234]) unterscheidet man zwischen offenen und geschlossen kinematischen Ketten. Eine offene kinematische Kette besteht aus einer Baumstruktur von LINKS und JOINTS, wobei jedes Glied mit maximal zwei Gelenken verbunden ist. Im einfachsten Fall ist dies eine serielle Abfolge von Gliedern und Gelenken. Wenn jedes Glied einer kinematischen Kette mit mindestens zwei Gelenken verbunden ist, so bezeichnet man diese kinematische Kette als geschlossen (z. B. eine Stewart-Plattform [63, S. 243]).

Roboterarme

Ein generischer Roboterarm, wie z. B. der klassische industrielle Manipulator, besteht aus einer seriellen kinematischen Kette, die an der Basis des Manipulator beginnt und am Flansch bzw. Endeffektor endet. Dementsprechend ist ein Roboterarm im Paket *org.roboticsapi.robot* wie folgt definiert:

Definition 5.27. Ein ROBOTARM ist ein MULTIJOINTDEVICE, das einen seriellen Aufbau von Gelenken und Gliedern hat. Am Ende der seriellen Kette befindet sich der Roboterflansch.

Der Roboterflansch (engl.: *flange*) wird durch einen LINK und ein spezielles Koordinatensystem, dem *flange* FRAME, repräsentiert (vgl. Abb. 5.10). Da die JOINTS und LINKS eines Manipulators über FRAMES und CONNECTIONS verbunden sind, wird neben der Struktur auch immer die aktuelle Konfiguration des Manipulators abgebildet. Das bedeutet, dass die aktuelle Position eines LINKS bzw. seines Basiskoordinatensystems gegenüber einem Bezugssystem (z. B. der Roboterbasis) bestimmt werden kann. Der Aufbau eines generischen Roboterarms mit sechs Gelenken und sieben Gliedern ist in Abbildung 5.11 als Objektdiagramm dargestellt.

Hierbei zeigt sich auch der hierarchische Aufbau eines Roboterarms, der aus mehreren LINKS und JOINTS besteht. Diese wiederum bestehen aus FRAMES und CONNECTIONS, die über die Assoziation *parent* mit ihrer übergeordneten Entität, d. h. einem LINK oder JOINT, verbunden sind. Dadurch ist es möglich, ausgehend von einem FRAME oder einer CONNECTION einen Rückschluss auf den Roboterarm zu ziehen. Die serielle kinematische Kette, die von den FRAMES und CONNECTIONS aufgebaut wird, kann bei der Basis des Roboterarms beginnend bis zum Roboterflansch durchlaufen werden.

Die Pose, d. h. die Position und Orientierung, des Roboterflansches bzw. des Endeffektors in Bezug zum Basiskoordinatensystem des Arms lässt sich aus der aktuellen Stellung der Gelenke bestimmen, da sie die einzigen variablen Größen darstellen. Hierbei spricht man von der *direkten Kinematik* [63, S. 62–100] oder Vorwärtstransformation. Im Umkehrschluss befasst sich die *inverse Kinematik* [63, S. 101–134] oder Rückwärtstransformation mit der Frage, wie sich die Gelenkwinkel anhand der Pose des Endeffektors bestimmen lassen. Während sich bei der direkten Kinematik eine eindeutige Lösung bestimmen lässt, muss die Position der einzelnen Glieder bei einer gegebenen Pose des Endeffektors nicht eindeutig sein. Es wird im Allgemeinen mehrere Konfigurationen geben, die zur gewünschten Lage des Endeffektors führen. Folglich muss aus den vorhandenen Lösungen eine sinnvolle

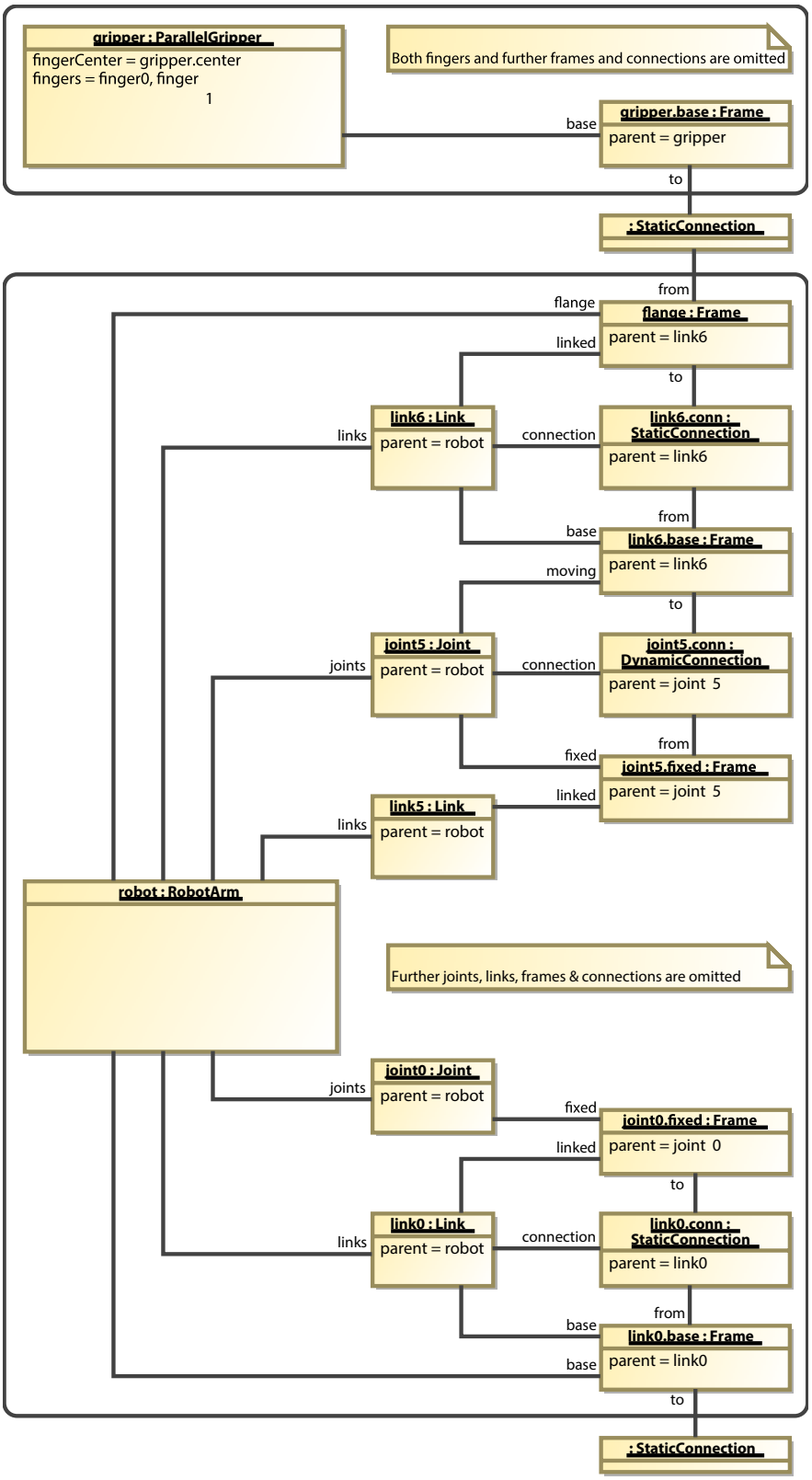


Abbildung 5.11: Ein RobotArm besteht aus mehreren Links und Joints, die über Frames und Connections miteinander geometrisch verbunden sind. Der ParallelGripper ist am Flansch des Roboters montiert.

Konfiguration ausgewählt werden, die zulässig ist und durch die Gelenke eingenommen werden kann.

Für die Berechnung der direkten bzw. inversen Kinematik definiert die Klasse RobotArm mehrere Methoden (vgl. Abb. 5.10). Zum einen kann zu einer gegebenen Gelenkstellung eine Transformation berechnet werden, die die Translation und Rotation des Flansch-Koordinatensystems in Bezug zum Basiskoordinatensystem beschreibt. Zum anderen kann zu einer gegebenen Transformation des Flansch-Koordinatensystems in Bezug zum Basiskoordinatensystem eine Gelenkstellung ermittelt werden, die eine zulässige Lösung der inversen Kinematik beschreibt und möglichst nahe an der aktuellen Gelenkstellung ist. Daneben gibt es weitere Methoden, die eine genauere Auswahl möglicher Lösungen erlauben.

Da ein RobotArm zugleich ein MultiJointDevice ist, können über das JointPtpInterface achsspezifische Bewegungen definiert werden. Daneben sind bei einem Roboterarm auch Bahnbewegungen möglich. Diese beschreiben durch eine mathematische Funktion die Bewegung eines Frames im kartesischen Raum. Dazu muss dieser als Motion Center Point (MCP) bezeichnete Frame jedoch statisch mit dem Flansch-Koordinatensystem verbunden sein. Für lineare Bahnbewegungen existiert das in Abbildung 5.10 dargestellte LinearMotionInterface. Die abgebildete Methode gibt eine Activity zurück, die eine lineare Bewegung von der aktuellen Position zur angegeben Zielposition repräsentiert. In der *Robotics API* könnten achsspezifische Bewegungen und Bahnbewegungen beliebig verwendet werden.

Werkzeuge

Am Roboterflansch ist in der Regel ein Endeffektor montiert. Dabei handelt es sich um eine Vorrichtung bzw. ein Werkzeug, mit der ein Roboter die vorgegebene Aufgabe erfüllen kann. Ein Endeffektor kann über eine StaticConnection mit dem Roboterflansch verbunden werden (vgl. Abb. 5.11) und bewegt sich so mit dem Roboterarm. In der *Robotics API* wird ein Endeffektor durch die Klasse Tool repräsentiert, die in Abbildung 5.12 dargestellt und wie folgt definiert ist:

Definition 5.28. Ein Tool ist ein Actuator, der an einen Roboterflansch montiert werden kann.

Die Klasse Tool erbt als steuerbares Gerät von der Klasse Actuator und, da es eine konkrete physische Ausprägung hat, zugleich von PhysicalObject (vgl. Abb. 5.12). Jedes Werkzeug hat einen *effector* Frame, der ein ausgezeichnetes Merkmal des Werkzeugs beschreibt (z. B. die Spitze eines Schweißbrenners). Dieser Frame sollte als MCP für Bahnbewegungen des Werkzeugs geeignet sein.

Greifer

Eine Spezialisierung eines Werkzeugs ist Klasse Gripper, die ebenfalls in Abbildung 5.12 dargestellt und folgendermaßen definiert ist:

Definition 5.29. Ein Gripper ist ein Tool zum mechanischen Greifen von Gegenständen.

Dabei kann es sich um einfache 2- oder 3-Finger-Greifer aber auch um komplexe Greifhände [178] handeln. Jedoch verfügt jeder dieser Greifer über Finger, die mithilfe von Kraft- oder Formschluss den gewünschten Gegenstand greifen. Die Anzahl der Finger ist von der konkreten Ausgestaltung der Greifers abhängig. Analog zu einem Link sind Finger als PhysicalObjects modelliert und haben eine physische Ausprägung. Über eine DynamicConnection ist es möglich, die Bewegung der Finger abzubilden. Darüber hinaus hat jeder Greifer ein Attribut, das das maximale bzw. bevorzugte Werkstückgewicht spezifiziert.

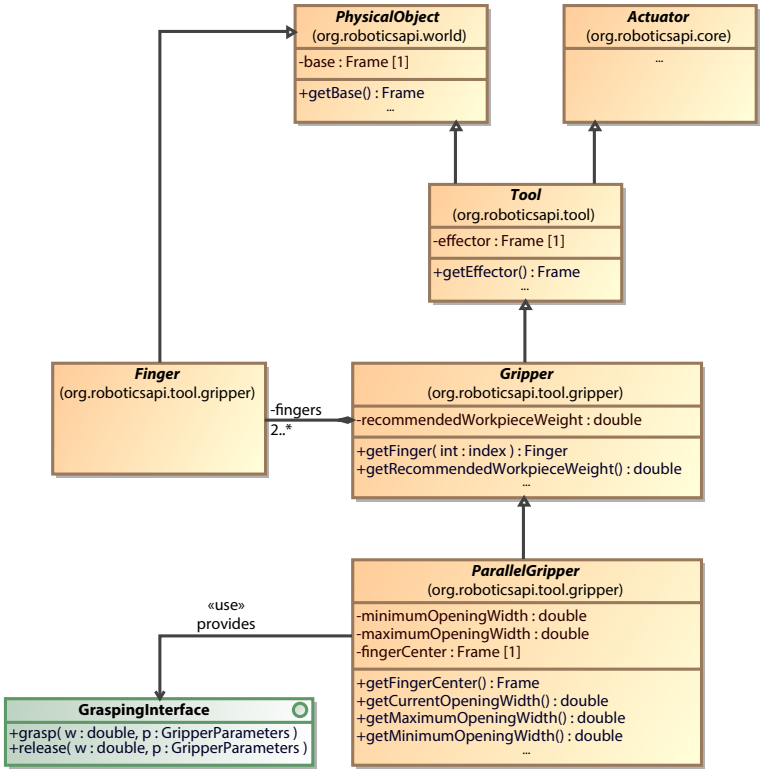


Abbildung 5.12: Ein Tool ist die Basisklasse für alle Geräte, die als Endeffektor an einen Roboterflansch montiert werden können. Eine Spezialisierung stellt bspw. ein Greifer dar.

Eine konkrete Ausprägung eines Greifers ist der parallele 2-Finger-Greifer, der durch die Klasse ParallelGripper repräsentiert wird (vgl. Abb. 5.12). Bei diesem Greifertyp sind die beiden Finger gegenüber angebracht und bewegen sich beim Öffnen oder Schleißen des Greifers gleichförmig nach außen bzw. innen. Er ist folgendermaßen definiert:

Definition 5.30. Ein ParallelGripper ist ein Gripper, der über zwei parallel angeordnete Greiffinger verfügt.

Jeder ParallelGripper verfügt über ein als *fingerCenter* bezeichnetes Koordinatensystem, das den Mittelpunkt zwischen den beiden Greiffingern darstellt. An diese Greiffinger können nochmals zusätzliche Greiferbacken angebracht werden, die in der Regel abhängig von der Aufgabe konstruiert und gefertigt werden. In der *Robotics API* wird dies ebenfalls über eine StaticConnection realisiert. Diese zusätzlichen Greiferbacken vergrößern oder verkleinern üblicherweise die Öffnungsweite der Greifer-eigenen Finger. Bei der Bestimmung der aktuellen Öffnungsweite (vgl. Abb. 5.12) wird dies berücksichtigt. Zudem wird über zwei Attribute eine minimale und maximale Öffnungsweite des Greifers spezifiziert.

Die Funktionalität eines ParallelGrippers wird durch ActuatorInterfaces angeboten. Das in Abbildung 5.12 dargestellte GraspingInterface besitzt zwei Methoden, um einen Gegenstand zu greifen bzw. wieder freizugeben. Dabei wird eine Öffnungsweite spezifiziert, bei der der Gegenstand idealerweise fest gegriffen ist. Der Greifer stellt daraufhin seine beiden Greiffinger auf diese Öffnungsweite ein. Falls über interne Sensoren kein Gegenstand erkannt wurde, wird die Aktivität mit einem Fehler beendet. Nachdem ein Gegenstand gegriffen wurde, wird über Sensoren aktiv überprüft,

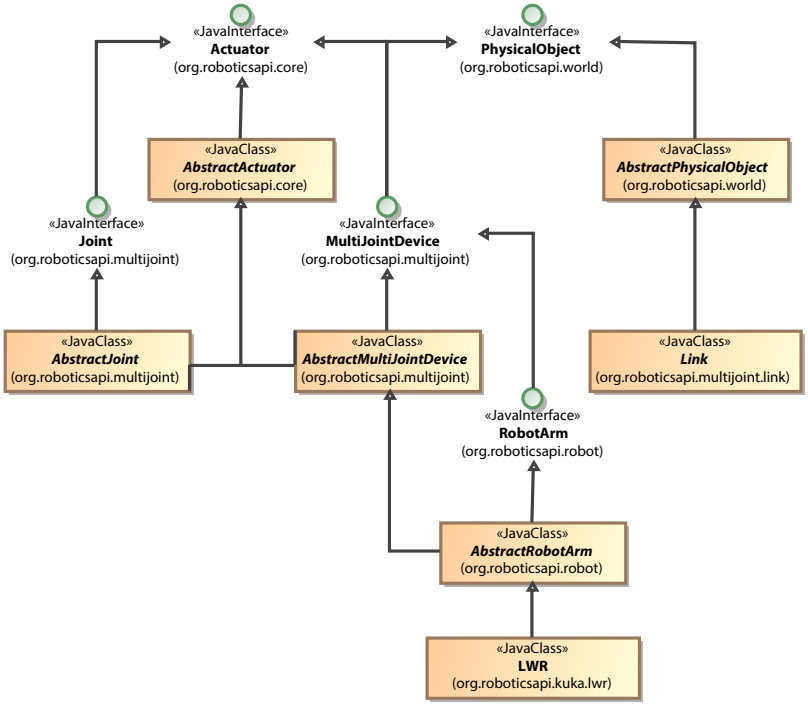


Abbildung 5.13: Das Klassendiagramm zeigt den Zusammenhang zwischen den Schnittstellen MultiJointDevice und RobotArm sowie ihren abstrakten Implementierungen.

ob der Gegenstand noch im Greifer sitzt. Hierbei wird das objektorientierte Weltmodell der *Robotics API* noch nicht ausgenützt, d. h. es muss die notwendige Öffnungsweite bekannt sein. Außerdem müssen Aktualisierungen des Weltmodells manuell durchgeführt werden (z. B. eine Verknüpfung zwischen gegriffenem Gegenstand und Greifer hinzufügen). Erst durch die in Kapitel 10 vorgestellten Dienste werden umfassendere Fähigkeiten für Werkzeuge realisiert.

Details zur Implementierung

Die oben vorgestellten Konzepte sind in der *Robotics API* hauptsächlich als Schnittstellen realisiert. Dies gilt insbesondere für das MultiJointDevice und den RobotArm. Daneben existieren in der *Robotics API* abstrakte Klassen, um die elementare Funktionalität der jeweiligen Schnittstelle zu implementieren. Der Zusammenhang zwischen den Schnittstellen und ihrer abstrakten Implementierung ist in Abbildung 5.13 dargestellt. Dort ist zu erkennen, dass ein MultiJointDevice sowohl eine Spezialisierung eines Actuators als auch eines PhysicalObjects ist. Um eine höhrere Wiederverwendung zu erreichen, erbt die Klasse AbstractMultiJointDevice direkt von der Klasse AbstractActuator. Die noch fehlende Funktionalität eines PhysicalObjects ist direkt im AbstractMultiJointDevice implementiert. Das Konzept eines RobotArms ist ebenfalls in der *Robotics API* als Schnittstelle realisiert. Sie spezialisiert dabei die Schnittstelle MultiJointDevice und wird abstrakt von der Klasse AbstractRobotArm implementiert.

5.6 VERWANDTE ARBEITEN

Lozano-Pérez [167] stellt eine geeignete Modellierung der Umwelt bzw. Umgebung eines Roboters als elementare Anforderung für dessen Programmierung heraus. Dillmann und Huck [67] sehen ein repräsentatives Umwelt-

modell als „Voraussetzung für die Informationsverarbeitung in der Robotik“ [67, S. 43]. Das bedeutet, dass das Umweltmodell sowohl die Grundlage für eine Analyse der Roboterzelle als auch für eine mögliche Planung von Aufgaben ist. Daher sollten alle realen Objekte der Roboterzelle (z. B. Roboter, Endeffektoren, Peripheriegeräte und Sensoren) mit ihren geometrischen Merkmalen abgebildet werden. Die Struktur der Roboterzelle ergibt sich „aus der räumlichen Organisation und Topologie der Zelle“ [67, S. 49]. Die in diesem Kapitel vorgestellten Mechanismen der *Robotics API* eignen sich, um alle von Dillmann und Huck [67] beschriebenen Anforderungen hinreichend zu erfüllen.

Bei der von Hayward und Paul [115] vorgeschlagenen Robot Control C Library (RCCL) wird das Umweltmodell durch eine Menge von geometrischen Gleichungen beschrieben. Dieses Gleichungssystem definiert die geometrischen Abhängigkeiten zwischen unterschiedlichen Objekten bzw. deren Koordinatensystemen. PasRo [36], eine Bibliothek zur Steuerung von Robotern in der Programmiersprache Pascal, verwendet Koordinatensysteme, um Aufgaben und Bewegungen in einer Roboterzelle zu spezifizieren.

Einer der ersten objektorientierten Ansätze zur Programmierung von Robotern stellt RIPE dar, das von Miller und Lennox [182] entwickelt wurde. Sie unterscheiden zwischen dem Konzept einer *Station*, eines *Device* und eines *WorkPiece*. Dabei besteht eine *Station* aus *Devices* und *WorkPieces* und definiert deren geometrische Position. *Devices* führen Aktionen auf einem *WorkPiece* aus und bearbeiten es somit. Durch Vererbung können Spezialisierungen dieser Konzepte für die Realisierung einer eigenen Roboterzelle erstellt werden. Ähnlich beschreiben Jurkevich et al. [143] eine objektorientierte Modellierung einer Roboterzelle zum Punktschweißen. Weitere objektorientierten Bibliotheken zur Roboterprogrammierung sind MRROC++ [287, 288] und ZERO++ [214]. Bei den vorgestellten Ansätzen werden, im Gegensatz zur *Robotics API*, keine semantischen Beziehungen zwischen Koordinatensystemen bzw. Objekten modelliert. Dadurch kann dieses Wissen während der Ausführung nicht verwendet werden, um z. B. eine optimale Greifstrategie auszuwählen oder zu berechnen

Musliner et al. [185] verfolgen bei der Umweltmodellierung für die Cooperative Intelligent Real-time Control Architecture (CIRCA) einen anderen Ansatz. Dementsprechend besteht das Umweltmodell aus einer Menge von endlichen Zustandsautomaten. Jedem Zustand sind eine Menge von Prädikaten zugeordnet, die Eigenschaften über die Umwelt beschreiben. Falls ein Zustand aktiv ist, so gelten dessen Prädikate. Die Transitionen zwischen Zuständen werden durch Aktionen des Robotersystems aktiviert. Dadurch ist das Umweltmodell die Grundlage für die automatische Planung von Roboteraufgaben. Analog zu CIRCA, ändern Aktivitäten der *Robotics API* das Umweltmodell. Zum Beispiel wird durch eine Roboterbewegung der am Flansch montierte Endeffektor neu positioniert. Jedoch ändern Aktivitäten bisher die Topologie der Roboterzelle nicht automatisch (z. B. nach dem Greifen eines Werkstückes).

McKee et al. [177] verwenden bei ihrem MARS-Modell objektorientierte Konzepte, um die Beziehung zwischen Modulen zu beschreiben. Ein Modul repräsentiert eine Ressource des Robotersystems und kann entweder physikalisch (z. B. ein Manipulator) oder abstrakt (z. B. ein Algorithmus) sein. Durch Annotationen werden die Eigenschaften eines Moduls beschrieben und das Modul damit spezifiziert. Dazu gehört bspw. die Position eines physikalischen Moduls. Zudem können zwischen Modulen Relationen beschrieben werden, um bspw. auszudrücken, dass ein Werkzeug an einem

Manipulator montiert ist. Dadurch erbt das Werkzeug gleichzeitig die Fähigkeiten des Manipulators und kann sich bewegen. Die Verwendung von Relationen weist Parallelen zur *Robotics API* auf. Jedoch können die Relationen nur statisch zur Laufzeit definiert werden.

Die von Loffler et al. [166] beschriebene Robotic Platform ist eine objektorientierte Softwareplattform für Roboterapplikationen. Die Robotic Platform kennt analog zur *Robotics API* eine Klasse PhysicalObject, um reale Objekte der Roboterzelle zu repräsentieren. Insbesondere beschreibt ein PhysicalObject seine (dreidimensionale) Visualisierung, seine Position und Orientierung im Raum sowie Verbindungen zu anderen Objekten. Dadurch lässt sich ausdrücken, dass ein Greifer am Flansch des Roboters montiert ist. Allerdings sind diese Verbindungen im Gegensatz zur *Robotics API* statisch und werden zum Start der Applikation mithilfe einer Datei konfiguriert. Auch verzichten Loffler et al. weitere Objekte der Roboterzelle (z. B. Werkstücke) zu modellieren.

Die in Abschnitt 4.4 bereits behandelte Coupled Layered Architecture for Robot Autonomy (CLARAty) verwendet Objektorientierung, um mit der Vielfalt an Geräten und Algorithmen in der Robotik zurecht zu kommen und so wiederverwendbare Software zu entwickeln (vgl. [190]). Daher verwendet CLARAty ein objektorientiertes Modell des zu steuernden Roboters und seiner einzelnen Bestandteile. Insbesondere betrachten sie geometrische Transformationen nicht isoliert sondern immer im Kontext der betroffenen Koordinatensysteme und Modellelemente. Dementsprechend haben sie einen mit der *Robotics API* vergleichbaren Ansatz, die Struktur der Roboters und seine Koordinatensysteme gemeinsam zu modellieren [190]. Da jedoch der Fokus von CLARAty auf autonomen Landefahrzeugen für Weltraummissionen liegt, kann kein Vergleich für die ganzheitliche Modellierung einer Roboterzelle gezogen werden.

Der Graph-basierte Ansatz der *Robotics API* mit Frames als Knoten und Relations als Kanten weist Ähnlichkeiten zu den topologischen Karten auf, die Burgard und Hebert [53] beschreiben. Einer der ersten Ansätze stammt von Kuipers und Byun [155], die durch eine topologische Karte die Struktur der Umgebung erfassen. Dadurch werden neben rein geometrischen Daten weitere Informationen (z. B. Verbindungen zwischen Räumen) erfasst und können von Roboter verwendet werden. Der Ansatz wird später von Kuipers [154] zu einer *Spatial Semantic Hierarchy* erweitert. Analog zur *Robotics API* wird durch mehrere verbundene Ebenen unterschiedlicher Semantik die Umwelt des Roboters detaillierter beschrieben. Der Fokus beider Ansätze liegt jedoch auf einer zweidimensionalen Karte für die Navigation mobiler Roboter.

Analog zur *Robotics API* gibt es einige Ansätze, die die Umgebung des Roboters als Szenengraph beschreiben. Ein Szenengraph ist eine objektorientierte Datenstruktur aus dem Bereich der Computergrafik, mit der sowohl die räumliche als auch die logische Anordnung der darzustellenden, dreidimensionalen Szene beschrieben wird. Smits [249, S. 121 ff.] schlägt in seiner Dissertation ein Format für einen *Robot Scene Graph* vor. Das Unified Robot Description Format (URDF) [232] ist eine Format, um Roboter und damit Teile des Szenengraphen in ROS zu beschreiben. Beide Ansätze sind weniger komplex verglichen zur *Robotics API* und enthalten keine semantische Informationen zwischen Objekten. Blumenthal et al. [37] stellen einen verteilten Szenengraphen für Roboteranwendungen vor, berücksichtigen jedoch auch keine semantischen Informationen zwischen Objekten.

6 ERWEITERBARES FRAMEWORK FÜR DIE AUTOMATISIERUNG

Das Ziel dieser Arbeit ist es, Roboterzellen flexibel und modular unter Zuhilfenahme serviceorientierter Architekturen zu strukturieren und zu programmieren. Um dies zu erreichen, können alle Komponenten einer Roboterzelle, z. B. Manipulatoren und deren Werkzeuge, einheitlich mit dem *Robotics Application Framework* programmiert und echtzeitfähig gesteuert werden. Daher muss es flexibel und modular gestaltet sein, um alle relevanten Aspekte einer Roboterzelle vollständig abzudecken. Folglich ist das *Robotics Application Framework* in einzelne Bestandteile gegliedert, die entsprechend den Anforderungen einer konkreten Roboterzelle zusammengesetzt werden können.

Abhängig von den Geräten und Aufgaben der Roboterzelle ergibt sich immer wieder eine neue Komposition einzelner Bestandteile. Bei Veränderungen kann dementsprechend das Ensemble der einzelnen Softwaremodule adaptiert werden, um den neuen Bedürfnissen zu genügen. Durch eine passende Auswahl von Softwaremodulen kann eine Roboterzelle durchgängig mit dem *Robotics Application Framework* programmiert werden. Demzufolge stellt es eine ganzheitliche Plattform dar, um Automatisierungssoftware zu entwickeln.

Um die Modularisierung vorzunehmen, wurde die in Kapitel 2 beschriebene *OSGi Service Platform* verwendet. Somit bestehen die *Robotics API* und der *Runtime Adapter* aus einer Menge von Bundles, die miteinander in Beziehung stehen und über Services interagieren. Für die modulare Entwicklung einer Automatisierungssoftware können die erforderlichen Bundles ausgewählt und verwendet werden. Allerdings endet der modulare Aufbau der Software nicht innerhalb des *OSGi Frameworks*, sondern erstreckt sich auch auf die echtzeitfähige Robotersteuerung. Die Integration des *Robotics Application Frameworks* in die *OSGi Service Platform* stellt eine Grundlage für die weitere serviceorientierte Implementierung einer Roboterzelle dar.

Dieses Kapitel stellt vor, wie die zuvor beschriebene Architektur, die eine Anwendungs- und Echtzeitprogrammierung von Robotern harmonisiert (vgl. Kap. 4), ein ganzheitliches Framework für die Programmierung von Roboterzellen bildet. Das Framework besteht aus einem schmalen Kern, der in eine vertikale und in eine horizontale Dimension erweitert werden kann. Die Erweiterungsmöglichkeiten des *Robotics Application Frameworks* werden detailliert in Abschnitt 6.1 diskutiert. Die Abhängigkeiten zwischen den einzelnen Bundles und Ebenen der Architektur werden in Abschnitt 6.2 erläutert. Die durchgängige Konfiguration einer Roboterzelle und die Installation der entsprechenden Bundles wird in Abschnitt 6.3 vorgestellt.

Dieses Kapitel beschreibt darüber hinaus exemplarisch einige Erweiterungen des *Robotics Application Frameworks* und zeigt dadurch die unterschiedlichen Möglichkeiten auf, das Framework sowohl in die vertikale Dimension (vgl. Abschn. 6.4) als auch die horizontale Dimension (vgl. Abschn. 6.5) zu erweitern. Die vorgestellten Beispiele werden im weiteren Verlauf der Arbeit verwendet, um eine serviceorientierte Roboterzelle zu implementieren. Abschließend werden in Abschnitt 6.6 verwandte Arbeiten diskutiert. Einige der Überlegungen und Ergebnisse wurden erstmals in [119] veröffentlicht.

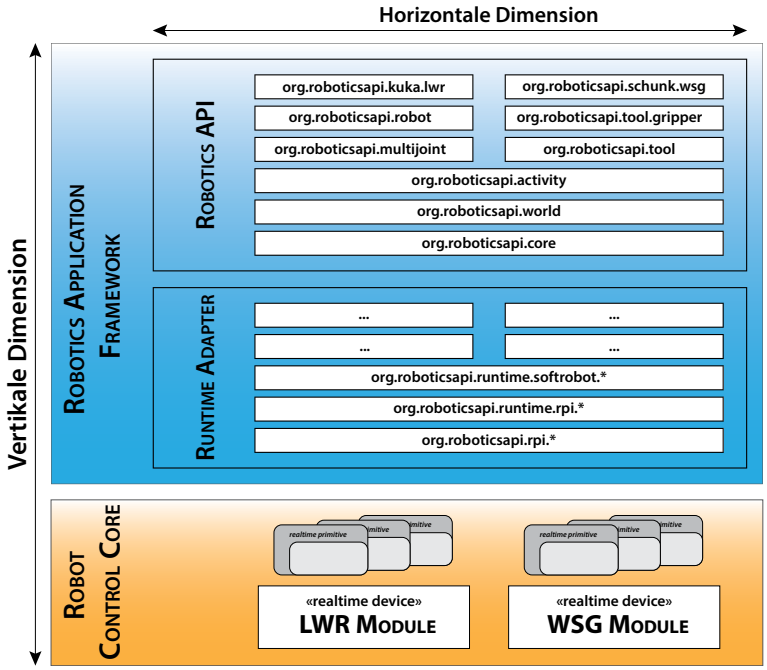


Abbildung 6.1: Die Erweiterbarkeit des *Robotics Application Frameworks* erstreckt sich auf zwei Dimensionen – einer horizontalen und einer vertikalen Dimension – und betrifft auch den Robot Control Core.

6.1 ERWEITERUNGSMÖGLICHKEITEN

In Kapitel 4 wurde die in *SoftRobot* entwickelte Softwarearchitektur für Industrieroboter vorgestellt. Während der Robot Control Core (RCC) für die echtzeitfähige Steuerung von Aktuatoren zuständig ist, stellt das *Robotics Application Framework* eine Programmierschnittstelle zur Entwicklung von Roboter- bzw. Automatisierungsapplikationen bereit. Das *Robotics Application Framework* besteht intern aus der *Robotics API* als der eigentlichen Programmierschnittstelle und einem *Runtime Adapter* zur Kommunikation mit der echtzeitfähigen Steuerung. Diese beiden Teile der Architektur sind in viele einzelne, aber inhaltlich zusammengehörige, Pakete gegliedert (vgl. Abschn. 4.3). Dieselbe Gliederung in Pakete ist auch aus Kapitel 5 bekannt, da dort die einzelnen Element zur objektorientierten Modellierung einer Roboterzelle unterschiedlichen Paketen zugeordnet waren.

Jedoch reicht eine Aufteilung in Pakete nicht aus, um flexible und modulare Software in Java zu erhalten [285, S. 17]. Zudem kann nicht zwischen öffentlicher Programmierschnittstelle und interner Implementierung unterschieden werden [278, S. 6]. Daher wird OSGi (vgl. Kap. 2) verwendet, um das *Robotics Application Framework* in einzelne Komponenten bzw. Bundles zu gliedern. Jedes Bundle enthält inhaltlich zusammengehörige Pakete, Schnittstellen und Klassen, um eine hohe Kohäsion [162] innerhalb eines Bundles zu erreichen. Gleichzeitig wird die Kopplung zwischen Bundles minimiert. Die öffentlich verfügbaren Klassen und Schnittstellen werden jeweils als Pakete exportiert, sodass andere Bundles darauf Zugriff haben. Dies ermöglicht die Trennung zwischen öffentlich verfügbaren Klassen und Schnittstellen sowie der internen Implementierung eines Bundles.

Der modulare Aufbau der *Robotics API* und des *Runtime Adapters* sind die Voraussetzung für die einfache Erweiterbarkeit der Softwarearchitek-

tur. Dadurch entsteht ein erweiterbares und flexibles Framework für die Programmierung von Industrierobotern bzw. im Allgemeinen für Automatisierungskomponenten. Damit kann ein ganzheitliches Framework für die Softwareentwicklung und Automatisierung von Roboterzellen bereitgestellt werden. Das *Robotics Application Framework* bietet eine einheitliche Programmierung für ein breites Spektrum unterschiedlicher Automatisierungskomponenten und Robotern, wobei es die besonderen Anforderungen bzgl. Echtzeit für alle Geräte in gleicher Weise berücksichtigt. Die Erweiterbarkeit des *Robotics Application Frameworks* kann in zwei unterschiedliche Dimensionen aufgeteilt werden (vgl. Abb. 6.1): Die horizontale Dimension erweitert die Programmierschnittstelle, d. h. die *Robotics API*, um neue Gerätetypen und Fähigkeiten. Dadurch wird die Programmierschnittstelle immer universeller und deckt eine größere Bandbreite an funktionalen Anforderungen und Applikationen ab. Die vertikale Dimension beschreibt die Integration eines neuen Geräts oder neuer Steuerungsoperationen von der *Robotics API* bis hin zur echtzeitfähigen Robotersteuerung.

Dimensionen der Erweiterbarkeit

Oft sind durch eine neue Erweiterung beide Dimensionen betroffen. Wenn ein neuer Roboter vertikal integriert wird, so ist in der Regel der *Runtime Adapter* und der RCC betroffen. Allerdings erweitert der neue Roboter auch die Programmierschnittstelle. Gleiches gilt für eine Steuerungsoperation, die eine Erweiterung des *Runtime Adapters* und des RCC benötigt, aber auch adäquat durch die Programmierschnittstelle angeboten werden muss. Eine Erweiterung kann jedoch nur die horizontale Dimension betreffen, wenn sie sich z. B. auf bestehende Bundles der *Robotics API* abstützt (z. B. Planungsalgorithmen). Erweiterungen können ebenfalls, wenn auch selten, nur die vertikale Dimension betreffen, z. B. wenn ein Gerätetreiber ergänzt bzw. ausgewechselt wird, ohne das die *Robotics API* davon betroffen wäre.

Bei Erweiterbarkeit ist es ausschlaggebend, ein möglichst hohes Maß an Wiederverwendbarkeit zu erreichen. Gemäß Gamma et al. [97] gibt es in objektorientierten Systemen mit der *Vererbung* und der *Komposition* zwei Techniken, um Funktionalität wiederzuverwenden. *Vererbung* wird als *White-Box*-Wiederverwendung bezeichnet, da die interne Implementierung der Superklasse zumindest teilweise bekannt sein muss. Dagegen handelt es sich bei *Komposition* um *Black-Box*-Wiederverwendung, da die interne Implementierung des Objekts nicht bekannt ist. Zudem kann sie dynamisch zur Laufzeit definiert und ggf. geändert werden. Durch die beiden Techniken wird im *Robotics Application Framework* sichergestellt, dass Erweiterungen nicht komplett neu entwickelt werden müssen (vgl. [18, 119]). Nur die originären Aspekte müssen neu implementiert werden. Dies bedeutet beispielsweise, dass für jeden neuen Roboter dessen Bewegungen nicht neu implementiert, sondern wiederverwendet werden.

Erweiterungsmöglichkeiten

Neue (physische) Objekte oder Geräte einer Roboterzelle können erstellt werden, indem eine neue Klasse von der entsprechenden Superklasse abgeleitet wird. Ein konkreter Roboter, z. B. ein KUKA Leichtbauroboter (LBR)[1] durch die Klasse LWR, ist eine Spezialisierung der Klasse AbstractRobotArm und ergänzt alle Eigenschaften, die dem abstrakten RobotArm fehlen. Dazu gehören insbesondere die einzelnen Längen der mechanischen Struktur, maximale Gelenkbereiche und Geschwindigkeiten. Die konkrete Klasse legt die Anzahl der Gelenke fest und bestimmt die Länge der Roboterstruktur. Da der LBR über weitere Sensoren (z. B. Drehmomentsensoren) verfügt, erweitert er seine Basisklasse, um die entsprechenden Attribute und Methoden. Durch eine Property, d. h. eine generische Eigenschaft, können zusätz-

1 Im Englischen als *Light-Weight Robot* oder kurz LWR bezeichnet.

ERWEITERUNGSPUNKT	KONZEPTE	WIEDERVERWENDUNG
Objekte mit physischer Ausprägung	PhysicalObject	Vererbung
Geräte: Sensoren & Aktuatoren	Device, Actuator	Vererbung
Nativer Gerätetreiber	DeviceDriver, *Mapper*	Vererbung, Komposition
Zusammengesetzter Gerätetreiber	DeviceDriver	Komposition
Operationen bereitstellen	ActuatorInterface	Vererbung, Komposition
Operationen kombinieren	Activity, Command, *Mapper*	Komposition
Operationen implementieren	Action, *Mapper*	Komposition
Semantische Beziehungen zw. Frames	Relation	Vererbung
Generische Eigenschaften ergänzen	Entity, Property	Komposition

Tabelle 6.1: Das *Robotics Application Framework* weist eine Vielzahl an unterschiedlichen Erweiterungspunkten auf, um es sowohl in die horizontale als auch in die vertikale Dimension zu erweitern. Dabei wurde auf ein hohes Maß an Wiederverwendung geachtet.

liche Informationen zu einem Entity (z. B. einem Device, einem PhysicalObject, einem Frame oder einer Relation) hinzugefügt werden. Diese Informationen können spezifisch für eine Applikation oder eine Erweiterung sein und daher nicht fester Bestandteil des Objekts (vgl. Kap. 6.5).

Die Kommunikation mit dem realen Gerät über den RCC wird vollständig an einen DeviceDriver delegiert. Dieser ist dafür zuständig, aktuelle Daten (z. B. die Gelenkstellung eines Roboters) über den RCC abzurufen. Dadurch findet eine vollständige Entkoppelung der *Robotics API* zu einer konkreten Implementierung des Robot Control Cores statt [119]. Die Implementierung eines Treibers ist jedoch spezifisch für den verwendeten Robot Control Core und der Schnittstelle eines *Realtime Devices*. Durch diese Schnittstellen wird die echtzeitfähige Ansteuerung des Aktuators definiert [237] und folglich die vorhandenen *Realtime Primitives*. Um einen neuen Manipulator in den RCC und damit in das *Robotics Application Framework* einzubinden, muss ein neues *Realtime Device* mit der entsprechenden Schnittstelle implementiert werden. Die Echtzeitprimitive und Übersetzungsregeln (d. h. *Mapper*) können dadurch wiederverwendet werden.

Neben dem Einsatz von *nativen* Treibern, die als Proxy für einen Aktuator der Robotersteuerung stehen, kann sich ein Treiber auf andere, innere Aktuatoren abstützen. Ein solcher Treiber stellt eine Komposition von Geräten dar. Folglich muss sich der Zustand des Treibers bzw. des eigentlichen Aktuators vollständig durch die komponierten Aktuatoren beschreiben lassen. Ebenso müssen sich sämtliche Steuerungsoperationen des Aktuators aus Steuerungsoperationen der inneren Aktuatoren zusammensetzen (z. B. durch die Komposition von Steuerungskommandos). Oft werden zusammengesetzte Treiber für Roboterwerkzeuge verwendet, die in der Regel über digitale und analoge Ein- und Ausgänge angesteuert werden (vgl. Abschn. 6.4.1). Ein Beispiel dafür stellt der MEG-50 Parallelgreifer der Fa. Schunk dar, der in Abschnitt 6.4.2 detailliert vorgestellt wird.

Damit die möglichen Steuerungsoperationen dynamisch erweiterbar bzw. austauschbar sind, wurden sie als eigene Objekte modelliert, die zur Laufzeit flexibel kombiniert, übersetzt und zur Echtzeitsteuerung übertragen werden (vgl. Abschn. 4). Dem Applikationsentwickler stehen die möglichen Operationen eines Aktuators als Activity über ein ActuatorInterface zur Verfügung. Dazu bündelt ein ActuatorInterface jeweils semantisch zusammengehörende Operationen. Über eine Activity kann ein Applikationsentwickler Steuerungsoperationen eines Aktuators starten, überwachen

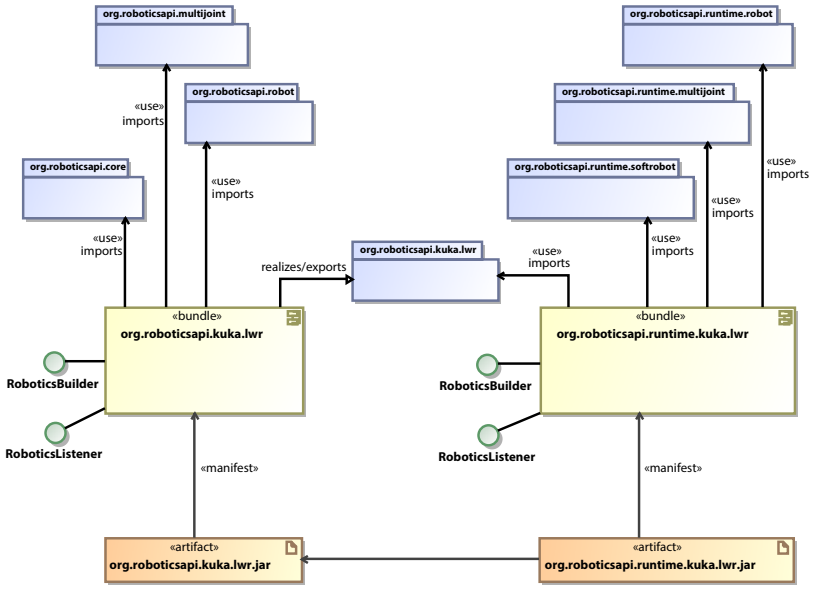


Abbildung 6.2: Das Komponentendiagramm zeigt exemplarisch, wie zwei Pakete als Bundle realisiert wurden.

und abbrechen. Intern besteht eine ACTIVITY aus einem COMMAND und ergänzt dieses um zusätzliche Informationen für die Ausführung. Um komplexe Steuerungsoperationen, insbesondere von mehreren Aktuatoren, nicht grundlegend neu implementieren zu müssen, können Steuerungsoperationen, d. h. Instanzen der Klasse COMMAND, durch Komposition verknüpft werden. Dazu existiert, wie in Abschnitt 4.3.2 vorgestellt, ein generischer Mechanismus, um einzelne Kommandos miteinander zu verknüpfen. Dieser Mechanismus kann auf die spezifischen Bedürfnisse einer Anwendung angepasst werden.

Bereitstellen von Erweiterungen

Alle Erweiterungsmöglichkeiten des *Robotics Application Frameworks* sind mit den betreffenden Konzepten in Tabelle 6.1 aufgelistet. Dabei werden sowohl *Vererbung* als auch *Komposition* verwendet, um einen möglichst hohen Grad an Wiederverwendung bei Erweiterungen zu erreichen. Für die Applikationsentwicklung wird eine Erweiterung, d. h. eine logisch zusammengehörige Menge von Paketen, als eigenes Bundle bereitgestellt und als JAR verpackt. Es exportiert seine öffentlichen Klassen bzw. Schnittstellen und definiert seine Abhängigkeiten zu anderen Paketen bzw. zu anderen Bundles.

Das Komponentendiagramm in Abbildung 6.2 zeigt dies exemplarisch für zwei Pakete. Auf der linken Seite ist das Bundle *org.roboticsapi.robot.lwr* dargestellt, welches Bestandteil der *Robotics API* ist. Es exportiert das gleichnamige Paket und realisiert die entsprechenden Klassen und Schnittstellen. Dazu gehört u. a. die Klasse LWR, die eine objektorientierte Repräsentation eines KUKA Leichtbauroboters darstellt. Allerdings ist das Bundle auf andere Pakete bzw. Bundles der *Robotics API* angewiesen, um seine Klassen zu implementieren.

Das Bundle *org.roboticsapi.runtime.robot.lwr* ist in Abbildung 6.2 auf der rechten Seite dargestellt und Bestandteil des *Runtime Adapters* für die Referenzimplementierung des RCC. Es realisiert Gerätetreiber für den KUKA Light-Weight Robot (LWR). Zudem sind dort Repräsentationen für Fragmente und Echtzeitprimitive sowie *Mapper* definiert, um roboterspezifische Fähigkeiten (z. B. kraftsensitive Bewegungen) abzubilden. Während es über keine öffentliche Programmierschnittstelle verfügt, importiert es andere Pa-

kete, die bspw. durch das oben beschriebene Bundle *org.roboticsapi.robot.lwr* exportiert werden. Die Abhängigkeit zwischen den Paketen löst das *OSGi Framework* zur Laufzeit auf, indem es beide Bundles verknüpft.

Um die Abhängigkeiten zwischen Bundles so gering wie möglich zu halten, sollte die öffentliche Programmierschnittstelle auf ein notwendiges Minimum beschränkt werden und nur (abstrakte) Klassen und Schnittstellen enthalten, die in anderen Bundles oder Applikationen verwendet werden. Um konkrete Implementierungen im *Robotics Application Framework* bereitzustellen, können Erweiterungen zwei spezielle Schnittstellen – RoboticsBuilder und RoboticsObjectListener – implementieren und als Service im *OSGi Framework* gemäß dem *Whiteboard Pattern* [151] veröffentlichen. Dadurch findet eine Entkopplung statt, um Erweiterungen und deren Implementierung auszutauschen bzw. zu aktualisieren.

RoboticsBuilder

Die erste Erweiterungsschnittstelle, als RoboticsBuilder bezeichnet, ist wie folgt definiert:

Definition 6.1. Durch die Schnittstelle RoboticsBuilder können Erweiterungen konkrete Implementierungen eines RoboticsObjects bei Bedarf erzeugen und bereitstellen.

Durch die Verwendung des *Builder Patterns* [97] wird die Konstruktion der Roboterzelle unabhängig von der Erzeugung konkreter Objekte. Dadurch lassen sich neue RoboticsObjects durch einen entsprechenden RoboticsBuilder einfügen und die Implementierung bestehender RoboticsObjects einfach austauschen. Außerdem kann der Konstruktionsprozess der Roboterzelle an einer dedizierten Stelle gesteuert werden, sodass sich eine spätere Änderung des Konstruktionsprozesses ohne Anpassung der RoboticsBuilder realisieren lässt. Beispielsweise kann eine Roboterzelle als Konfiguration einer Applikation durch eine Datei vollständig beschrieben werden. Zur Laufzeit erzeugt der Konstruktionsprozess sämtliche RoboticsObjects, die für die Roboterzelle und somit für die Applikation notwendig sind.

Gemäß Abschnitt 5.2 ist ein RoboticsObject als benanntes Objekt definiert, das konfiguriert und, bevor es benutzt werden kann, initialisiert werden muss. Da es sich um eine generische Schnittstelle handelt, wird sie bspw. von Device, Actuator, DeviceDriver und PhysicalObject erweitert. Aufgrund dieses breiten Spektrums lassen sich über die RoboticsBuilder-Schnittstelle Erweiterungen abdecken, welche die *Robotics API* um neue Konzepte, Spezialisierungen bestehender Konzepte oder Implementierungen vorhandener Schnittstellen erweitern.

Das Bundle *org.roboticsapi.robot.lwr* aus Abbildung 6.2 stellt eine Implementierung der Schnittstelle RoboticsBuilder als Service bereit, um Instanzen der Klasse Lwr zu erzeugen. Analog stellt das zweite Bundle ebenfalls einen RoboticsBuilder-Service bereit, um Instanzen einer Implementierung der Schnittstelle LwrDriver zu erzeugen. Durch dieses Vorgehen wird sichergestellt, dass Applikationsentwickler weder Instanzen von internen Klassen direkt erzeugen noch gegen diese programmieren. Da alle RoboticsBuilder als Service zur Verfügung stehen, können Infrastrukturkomponenten des *Robotics Application Frameworks* leicht auf diese zugreifen.

RoboticsObject-Listener

Neben dem RoboticsBuilder gibt es eine zweite Erweiterungsschnittstelle, die als RoboticsObjectListener bezeichnet wird:

Definition 6.2. Über die Schnittstelle RoboticsObjectListener werden Erweiterungen informiert, wenn ein RoboticsObject erfolgreich erstellt und konfiguriert wurde.

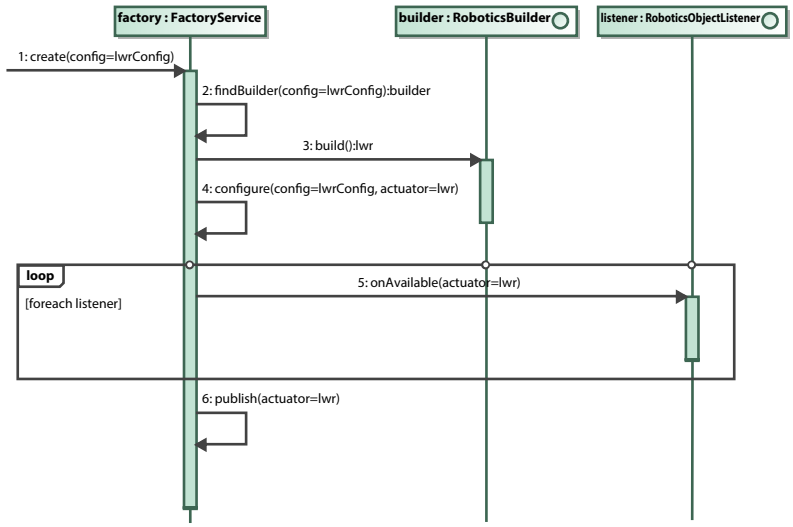


Abbildung 6.3: Das Sequenzdiagramm zeigt exemplarisch, wie ein RobotArm über einen RoboticsBuilder erzeugt und anschließend konfiguriert wird. Falls dies erfolgreich war, wird jeder RoboticsObjectListener informiert, das ein neuer RobotArm verfügbar ist.

Folglich kann jede Erweiterung RoboticsObjectListener als Service registrieren, um gemäß dem *Observer Pattern* [97] informiert zu werden, sobald ein neues RoboticsObject zur Verfügung steht. Bevor ein RoboticsObject in einer Applikation verwendet wird, kann es durch einen RoboticsObjectListener manipuliert bzw. erweitert werden. Das bedeutet, dass er ein RoboticsObject bzw. eine entsprechende Unterklasse durch Komposition in seiner Funktionalität erweitern kann.

Beide Bundles in Abbildung 6.2 implementieren diese Schnittstelle und stellen sie als Service bereit. Dadurch werden beide Bundles informiert, sobald ein neuer Actuator, eine neue RoboticsRuntime oder ein neuer Treiber erzeugt und erfolgreich konfiguriert wurde. Zum Beispiel fügt das Bundle *org.roboticsapi.robot.lwr* jedem LWR eine *Factory* für ein CompliantMotionInterface hinzu. Bundles des *Runtime Adapters* können zudem neue *Mapper* einer RoboticsRuntime hinzufügen. Infolgedessen ergänzt das Bundle *org.roboticsapi.runtime.robot.lwr* die RoboticsRuntime für die Referenzimplementierung des RCC um *Mapper* für kraftsensitive Bewegungen. Über diesen Weg kann zudem eine Property zu einem Device oder einem PhysicalObject hinzugefügt werden.

Abbildung 6.3 zeigt das Zusammenspiel zwischen den beiden Erweiterungsschnittstellen RoboticsBuilder und RoboticsObjectListener. Eine zentrale Infrastrukturkomponente ist für die Konstruktion des Softwaremodells der Roboterzelle zuständig und delegiert die Erzeugung eines einzelnen Element an einen FactoryService. In dem dargestellten Beispiel handelt es sich um eine Instanz eines Lwr, wobei alle notwendigen Parameter und Informationen Bestandteil der übergebenen Konfiguration sind. Um den Lwr zu erzeugen, wird zuerst ein passender RoboticsBuilder gesucht. Sobald dieser gefunden wurde, wird die Instanz vom FactoryService angefordert und vom RoboticsBuilder erzeugt. Anschließend wird das Objekt durch den FactoryService konfiguriert und initialisiert. Falls dies erfolgreich ist, werden alle RoboticsObjectListener informiert, dass ein neues RoboticsObject verfügbar ist. Erst danach wird das Objekt Teil des Softwaremodells der Roboterzelle und ist damit in der Applikation verfügbar.

6.2 ABHÄNGIGKEITEN ZUR LAUFZEIT

Eine Applikation besteht bei OSGi aus einer Menge an Bundles, zwischen denen gegenseitige Abhängigkeiten bestehen und die über Services miteinander interagieren. Um eine Applikation zu installieren, müssen alle dazugehörigen Bundles in dem *OSGi Framework* installiert werden. Im Fall des *Robotics Application Frameworks* bedeutet dies, alle notwendigen Bundles der *Robotics API*, des *Runtime Adapters* und des eigentlichen Roboterprogramms zu installieren. Folglich entsteht ein Geflecht aus einer Vielzahl von Bundles.

Abbildung 6.4 zeigt die Abhängigkeiten zwischen verschiedenen Bundles zur Laufzeit einer Applikation. Das Roboterprogramm – im Beispiel vereinfacht als einzelnes Bundle dargestellt – verwendet einen LWR als Manipulator und die Referenzimplementierung des RCC. Während die Klassen und Schnittstellen der *Robotics API* direkt in der Applikation verwendet werden, sind die Klassen des *Runtime Adapters* für den Applikationsentwickler nicht direkt sichtbar, sondern werden nur bei der Konfiguration einer RoboticsRuntime oder eines Gerätetreibers für ein Device benötigt.

Abhängigkeiten der Robotics API

Die in Abbildung 6.4 dargestellten Bundles der *Robotics API* haben untereinander Abhängigkeiten, die der Klassenhierarchie entsprechen. So benötigt *org.roboticsapi.robot* bspw. das Bundle *org.roboticsapi.multijoint*, welches wiederum auf *org.roboticsapi.core* aufbaut. Dies entspricht der Vererbungshierarchie der Klasse RobotArm. Transitive Abhängigkeiten (z. B. zwischen *org.roboticsapi.robot* und *org.roboticsapi.core*) sind in Abbildung 6.4 aus Gründen der Übersichtlichkeit nicht eingezeichnet. Da das Roboterprogramm, wie oben beschrieben, nur Klassen und Schnittstellen der *Robotics API* verwendet, entstehen nur Abhängigkeiten vom Bundle, welches das Roboterprogramm repräsentiert, zu den Bundles der *Robotics API*.

Demzufolge ist ein Roboterprogramm vollständig unabhängig vom verwendeten *Runtime Adapter* und somit auch von der verwendeten Robotersteuerung. Dies ermöglicht es, den *Runtime Adapter* oder die Robotersteuerung bzw. Teile davon auszutauschen, ohne dass eine Änderung an dem Roboterprogramm vorgenommen werden muss. Insbesondere können die Gerätetreiber und die auf der Robotersteuerung verwendeten *Realtime Devices* für das Roboterprogramm transparent ausgetauscht werden. Dadurch kann ein Programm gegen simulierte Treiber entwickelt und getestet werden und später ohne Änderung in der realen Roboterzelle verwendet werden. Dieser Wechsel zwischen einem simulierten und realen Gerätetreiber kann immer wieder und in beide Richtungen stattfinden.

Abhängigkeiten des Runtime Adapters

Bundles des *Runtime Adapters* haben, wie in Abbildung 6.4 dargestellt, Abhängigkeiten zu Bundles der *Robotics API*, da sie deren Klassen und Schnittstellen benutzen. Zudem bestehen zwischen den Bundles des *Runtime Adapters* Abhängigkeiten, welche weitgehend der Vererbungshierarchie der Gerätetreiber entsprechen. So baut bspw. das Bundle *org.roboticsapi.runtime.robot* auf dem Bundle *org.roboticsapi.runtime.multijoint* auf. Damit hat jedes Bundle des *Runtime Adapters* Abhängigkeiten zu den korrespondierenden Bundles der *Robotics API* und zu Bundles innerhalb des *Runtime Adapters*.

Zusätzlich hat jedes Bundle des *Runtime Adapters* noch weitere Abhängigkeiten, die über das *OSGi Framework* und die Java Virtual Machine (JVM) hinausgehen. Um korrekt funktionieren zu können, ist eine spezifische Implementierung und Version des RCC notwendig, der außerdem über die entsprechenden Erweiterungsbibliotheken verfügen muss. Diese Bibliotheken sind in Abbildung 6.4 als *Shared Object Libraries* eingezeichnet und enthalten

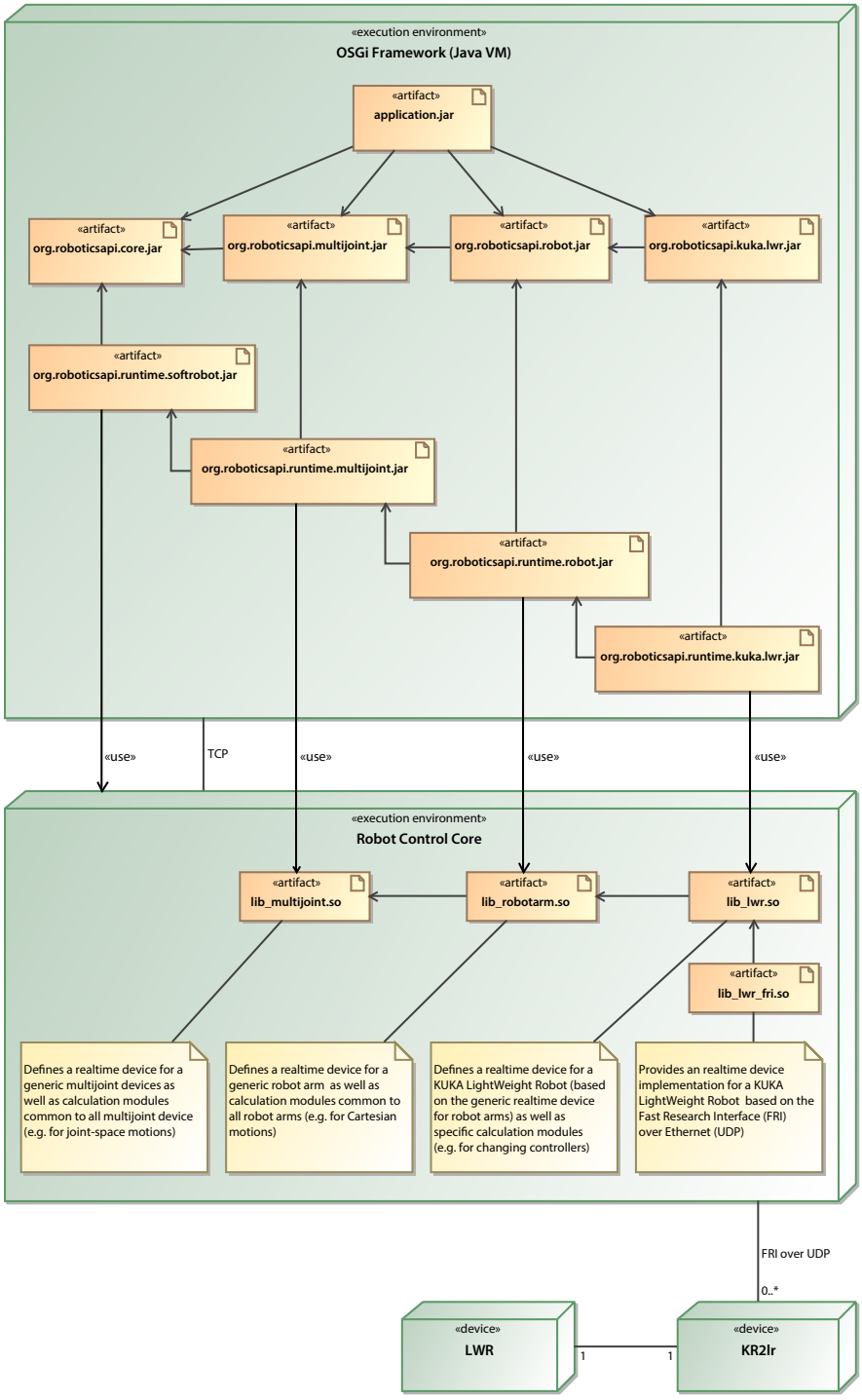


Abbildung 6.4: Das UML-Verteilungsdiagramm zeigt die Abhängigkeiten zwischen den Bundles der *Robotics API* und dem *Runtime Adapter*. Die Applikation hat dabei nur Abhängigkeiten zu der *Robotics API*. Neben den Abhängigkeiten innerhalb des *OSGi Frameworks* gibt es weitere Abhängigkeiten zwischen den Bundles des *Runtime Adapters* und den Programmbibliotheken der Echtzeitrobotersteuerung.

die Implementierung von *Realtime Primitives* sowie von *Realtime Devices* zur Ansteuerung von Sensoren und Aktuatoren.

Als Bestandteil des Bundles *org.roboticsapi.runtime.softrobot* implementiert die Klasse SoftRobotRuntime die Kommunikation mit der RCC-Referenzimplementierung und ist darauf angewiesen, dass die Gegenseite die gleiche Version des Kommunikationsprotokolls verwendet. Andere Bundles erwarten, dass eine bestimmte Menge von *Realtime Primitives* auf dem RCC vorhanden sind. Das Bundle *org.roboticsapi.runtime.multijoint* benötigt bspw. eine spezifische Menge an *Realtime Primitives*, um eine Achsbewegung zu interpolieren. Diese Abhängigkeiten können nicht automatisch durch das *OSGi Framework* verwaltet werden, da sie sich außerhalb der JVM befinden.

Abhängigkeiten mit dem RCC

Um eine modulare Softwareentwicklung für die Automatisierung von Roboterzellen zu gewährleisten, muss eine Möglichkeit geschaffen werden, um die Abhängigkeiten zwischen Bundles des *Runtime Adapters* (Java) und den Bibliotheken der Echtzeitrobotersteuerung (C++) zur Laufzeit aufzulösen. Dazu werden beide Systeme, d. h. das *Robotics Application Framework* innerhalb des *OSGi Frameworks* und der Robot Control Core, als separate Laufzeitumgebungen angesehen, die miteinander synchronisiert werden müssen. Um dies zu erreichen, kann eine Implementierung einer RoboticsRuntime die eigene Version mit der Version der verbundenen Echtzeitrobotersteuerung vergleichen. Dadurch kann sichergestellt werden, dass die RoboticsRuntime und die Echtzeitrobotersteuerung übereinstimmen.

Anschließend kann geprüft werden, ob die Bundles des *Runtime Adapters* mit den Bibliotheken der Echtzeitrobotersteuerung kompatibel sind. Dies ist möglich, da es sich hierbei um eine direkte Beziehung handelt. Das heißt, dass für jede Erweiterungsbibliotheken mit Echtzeitmodulen ein Bundle des *Runtime Adapters* existiert (vgl. Abb. 6.4). Für die Prüfung, ob das Bundle des *Runtime Adapters* und die Erweiterungsbibliotheken übereinstimmen, kann eine Versionsnummer oder Prüfsumme im Bundle gespeichert und verwendet werden. Dabei ist das Betriebssystem des RCC zu berücksichtigen.

Diese direkte Beziehung kann ausgenutzt werden, um die Erweiterungsbibliotheken der Echtzeitrobotersteuerung zusammen mit den entsprechenden Bundles zur Laufzeit zu installieren. Dadurch wird die Installation einheitlich innerhalb des *OSGi Frameworks* gebündelt. Dies führt dazu, dass nur noch die RoboticsRuntime und der RCC als Laufzeitumgebung synchronisiert werden müssen. Da die Installation der Erweiterungsbibliotheken automatisch über die Bundles des *Runtime Adapter* erfolgt, erhöht sich die Wartbarkeit des gesamten Systems. Bei der Installation einer Applikation wird die Trennung zwischen der Programmierschnittstelle (in Java) und der Echtzeitansteuerung (in C++) aufgehoben.

Um die Installation von Erweiterungsbibliotheken während der Laufzeit zu ermöglichen, können Fragmente des *OSGi Frameworks* verwendet werden. Damit können vorhandene Bundles um zusätzliche Ressourcen, Klassen und Dateien erweitert werden. Der Inhalt eines Fragmente wird genauso behandelt wie der ursprüngliche Inhalt des Bundles. Normalerweise werden Fragmente eingesetzt, um sprachabhängige Ressourcen oder plattformspezifische Bibliotheken auszuliefern. Im Kontext des *Runtime Adapters* können sie kompilierte Erweiterungsbibliotheken für verschiedene Betriebssysteme, auf denen eine Instanz des RCC laufen kann (z. B. Linux Xenomai, Windows) beinhalten. Abhängig vom Betriebssystem handelt es sich um eine *Shared Object Library* (Linux Xenomai) oder eine *Dynamic Link Library* (Windows), die Module für den RCC beinhalten. Zur Laufzeit kann das

Betriebssystem und die Version des RCC überprüft und die passende Erweiterungsbibliothek in Fragmenten lokalisiert werden.

Für das Bundle *org.roboticsapi.runtime.robot* kann es verschiedene Fragmente geben, die jeweils ein kompiliertes Artefakt der Erweiterungsbibliothek für den RCC enthalten. Das Artefakt `lib_robotarm.so` ist in Abbildung 6.4 als *Shared Object Library* für Linux Xenomai dargestellt. Dieses Artefakt enthält die entsprechenden *Realtime Primitives* für die *Mapper* des Bundles und kann bei Bedarf auf den RCC übertragen und dort geladen werden. Zusätzlich kann der Mechanismus auch angewandt werden, um vorhandene Artefakte zu validieren (z. B. über eine Prüfsumme) und so sicherzustellen, dass diese zum aktuellen *Runtime Adapter* kompatibel sind.

6.3 KONFIGURATION & INSTALLATION

Auf Basis des modularen *Robotics Application Frameworks* kann man die Automatisierungssoftware für eine Roboterzelle spezifisch zusammenstellen. Dazu werden zuerst alle erforderlichen Aktuatoren und Sensoren der Roboterzelle identifiziert. Falls es für diese Geräte entsprechende softwaretechnische Gegenstücke in der *Robotics API* gibt, kann man die Geräte direkt verwenden. Andernfalls müssen die noch fehlenden Geräte mithilfe der in Abschnitt 6.1 beschriebenen Erweiterungspunkte implementiert werden. Dies betrifft insbesondere Elemente, die spezifisch für eine Roboterzelle sind (z. B. Handhabungsobjekte, Funktionsträger oder Zuführeinrichtungen).

Die verwendeten Objekte der *Robotics API* bzw. die für die Roboterzelle spezifischen Objekte definieren zu weiten Teilen die Komposition des *Robotics Application Frameworks*. Sie bestimmen die Auswahl der Bundles, mit denen die Roboterzellen softwaretechnisch modelliert wird. Ergänzt werden diese Bundles um weitere, die ACTIVITIES bereitstellen, die Implementierung der Roboterzelle unterstützen (z. B. für die automatische Planung von Roboterbewegungen) oder den Zustand der Roboterzelle visualisieren. Für die Verbindung mit der Echtzeitrobotersteuerung (RCC) müssen außerdem Bundles des entsprechenden *Runtime Adapters* ausgewählt werden. Dabei werden für jedes an der Robotersteuerung angeschlossenes Gerät die Bundles benötigt, um sowohl die Gerätetreiber als auch die *Realtime Primitives* und *Realtime Devices* für den RCC bereitzustellen. Daraus ergibt sich schließlich die Auswahl notwendiger Bundles, um die Roboterzelle abzubilden.

Konfiguration einer Roboterzelle

Die Roboterzelle kann unabhängig von der eigentlichen Applikation (z. B. über eine Konfigurationsdatei) beschrieben werden und wird über *Dependency Injection* [92] der Applikation zur Verfügung gestellt. Damit ist es dem *Single-Responsibility*-Prinzip [173, S. 149 ff.] folgend möglich, die Verantwortlichkeit für den Aufbau der Roboterzelle und deren Elemente in eine zentrale Komponente zu überführen. Dadurch müssen Objekte der *Robotics API* nicht über Kenntnisse verfügen, die zur Erfüllung der eigentlichen Aufgabe (z. B. der Steuerung eines Manipulators) nicht nötig sind. Da alle Elemente einer Roboterzelle als ROBOTICSOBJECT (vgl. Abschn. 5.2) eine gemeinsame Schnittstelle teilen und über einen (eindeutigen) Namen verfügen, können sie über einen zentralen Service von der Applikation erfragt werden.

Zu den konfigurierbaren Elementen einer Roboterzelle gehören die verwendeten Geräte, z. B. Manipulatoren und Endeffektoren. Darüber hinaus benötigt jedes Gerät einen DEVICEDRIVER. Zudem können auch passive, statische Aufbauten der Roboterzelle konfiguriert werden. Handhabungsobjekte dagegen sind nicht statisch, da sie in der Regel während des Programmablaufs angeliefert, bearbeitet und abtransportiert werden. Daher sind sie

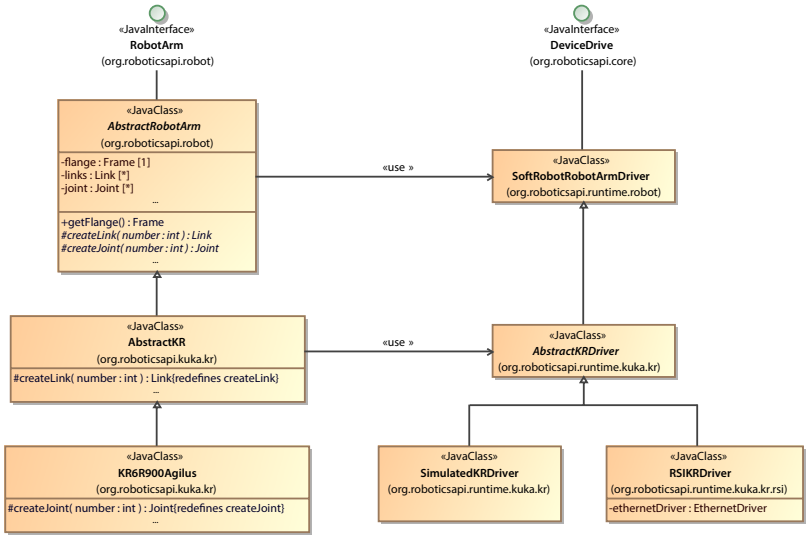


Abbildung 6.5: Für jeden konkreten DeviceDriver kann es zwei Ausprägungen geben: Während der eine Treiber ein simuliertes Gerät lädt, steuert der andere Treiber das reale Gerät an. Ein Wechsel zwischen den beiden Treibern ist für die *Robotics API* transparent.

kein konfigurierbarer Teil der Roboterzelle, sondern müssen dynamisch zur Laufzeit erzeugt werden. Durch die Konfiguration von Geräten ist die *Robotics API* ein Bestandteil der Zellenbeschreibung. Der *Runtime Adapter* wird durch die Konfiguration von RoboticsRuntimes und DeviceDrivers ebenfalls Teil der Zellenbeschreibung.

Die Elemente der Roboterzelle sind bisher ohne Zusammenhang zueinander. Daher werden sie bzw. ihre Frames über StaticConnections miteinander logisch und geometrisch in Verbindung gesetzt (vgl. Abschn. 5.4). Damit bilden die einzelnen Elemente der Roboterzelle sowohl einen Graphen aus Frames als auch einen Graphen aus physikalischen Körpern. Durch die Verknüpfung des Basiskoordinatensystems eines Greifers mit dem Flanschkoordinatensystem werden nicht nur beide Frames geometrisch in Relation gesetzt, sondern auch der Greifer logisch an den Roboter montiert. Das Resultat ist ein objektorientiertes Modell der realen Roboterzelle, das alle Bestandteile beschreibt und miteinander in Beziehung setzt.

Konfiguration des Robot Control Cores

Jede Roboterzelle muss mit mindestens einem RCC konfiguriert werden, kann aber auch mehrere enthalten. Ein RCC kann als echtzeitfähige Laufzeitumgebung für Aktuatoren und Sensoren angesehen werden. Die notwendigen Bibliotheken und Treiber werden zur Laufzeit bei Bedarf über das *Robotics Application Framework* geladen. Das bedeutet, dass durch jedes in der Applikation verwendete Bundle des *Runtime Adapters* die entsprechende Bibliothek auf den RCC übertragen und geladen werden kann. Dadurch wird jede Echtzeitrobotersteuerung individuell auf die Bedürfnisse der Applikation angepasst. Die Konfiguration dafür wird zentral über das *Robotics Application Framework* gesteuert und ist implizit über die Gerätetreiber Teil der oben beschriebenen Konfiguration der Roboterzelle.

Über die Klasse RoboticsRuntime, die als Proxy für einen RCC dient, können die verfügbaren Erweiterungsbibliotheken abgefragt werden. Zudem ist eine RoboticsRuntime – über die nativen Gerätetreiber – in der Lage, eine Erweiterungsbibliothek auf dem RCC nachzuladen. Demnach wird eine Erweiterungsbibliothek nur geladen, falls sie durch einen DeviceDriver

bzw. dessen *Realtime Device* verwendet wird. Durch mehrere Treiber gleichen Typs werden auch die Erweiterungsbibliotheken mehrmals geladen. Dies muss durch die RoboticsRuntime oder den RCC abgefangen werden. Neben den Erweiterungsbibliotheken wird über einen DeviceDriver auch das *Realtime Device* geladen. Alle Daten zum Laden des *Realtime Devices* müssen dem DeviceDriver bekannt sein: entweder als Konfigurationseigenschaft, als inhärente Information oder über das Device, für welches der Treiber zuständig ist.

In der Referenzimplementierung des RCC ist die Implementierung des DeviceDrivers bzw. des *Realtime Devices* für KUKA Roboter kompatibel zu allen 6-Achs-Knickarmrobotern (vgl. AbstractKR in Abb. 6.5). Dennoch benötigt der Treiber Daten über die Struktur, Gelenkgrenzen und Bewegungsparameter des Roboters. Diese Daten sind Teil eines konkreten RobotArms (z. B. des Kr6R900Agilus) und werden beim Instanziieren des *Realtime Devices* verwendet. Dazu kann ein DeviceDriver über zwei unterschiedliche Ausprägungen verfügen. Der SimulatedKRDriver lädt eine eingeschränkte Menge an Erweiterungsbibliotheken und instanziiert nur ein simuliertes *Realtime Device*. Der RSIKRDriver dagegen lädt alle Erweiterungsbibliotheken und instanziiert ein *Realtime Device*, das mit dem realen Roboter kommuniziert kann. Um dieses *Realtime Device* zu instanziieren sind teilweise andere und aufwendigere Konfigurationseigenschaft notwendig. Abbildung 6.5 zeigt, dass der *reale* Treiber über Ethernet und RSI mit dem Roboter kommuniziert. Der Socket für diese Verbindung ist ebenfalls ein *Realtime Device* und wird über den EthernetDriver erzeugt.

Zusammenfassend lässt sich festhalten, dass eine Roboterzelle vollständig innerhalb des *Robotics Application Frameworks* konfiguriert werden kann. Diese betrifft nicht nur die Geräte der *Robotics API*, sondern auch die geometrischen Relationen zwischen den Geräten und darüber hinaus die Gerätetreiber. Diese können die entsprechenden Erweiterungsbibliotheken und *Realtime Devices* auf dem Robot Control Core laden. Somit befindet sich das ganze Wissen über die Zellenkonfiguration gebündelt an einem Ort und wird automatisch auf die Ebenen verteilt. Das *Robotics Application Framework* zur Steuerung einer gesamten Roboterzelle setzt sich modular aus den benötigten Bundles der *Robotics API* und des *Runtime Adapters* zusammen und kann einfach erweitert oder aktualisiert werden.

6.4 BEISPIELE VERTIKALER ERWEITERUNGEN

Dieser Abschnitt beschreibt exemplarisch vertikale Erweiterungen des *Robotics Application Frameworks*, die neben der *Robotics API* auch den *Runtime Adapter* und den RCC betreffen. Die vorgestellten Beispiele werden im weiteren Verlauf der Arbeit verwendet, um eine serviceorientierte Roboterzelle zu modellieren bzw. zu implementieren. In Abschnitt 6.4.1 wird die Modellierung und Implementierung eines Feldbuskopplers mit digitalen und analogen Ein- und Ausgängen vorgestellt. Dadurch kann mit Roboterwerkzeugen oder Speicherprogrammierbaren Steuerungen kommuniziert werden. Die Implementierung von zwei Parallelgreifern wird in Abschnitt 6.4.2 beschrieben. Obwohl es sich um unterschiedliche Greifer handelt, können sie identisch programmiert werden. Diese Stärke des vorgestellten Ansatzes ermöglicht es generische und wiederverwendbare Programmfragmente zu erstellen. Analog wird in Abschnitt 6.4.3 die Implementierung von zwei Manipulatoren beschrieben, die ebenfalls identisch programmiert werden.

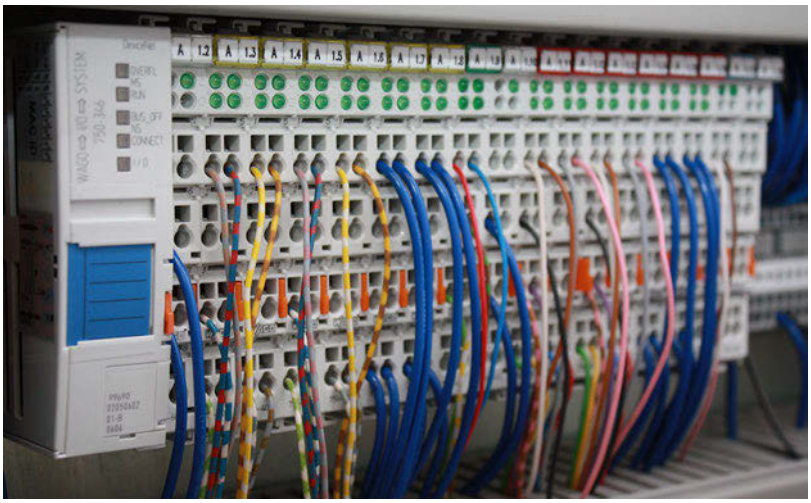


Abbildung 6.6: Ein Feldbuskoppler ist modular aufgebaut aus Baugruppen für das Einlesen bzw. Schreiben von binären und analogen Signalen.

6.4.1 *Integration von Feldbuskomponenten*

Eine Roboterzelle bildet in der Regel einen technischen Prozess ab. Dort treten physikalische Größen (z. B. Temperatur, elektrische Spannung) auf, die man über elektrische Signale erkennen muss. Man unterscheidet in der Automatisierungstechnik dabei zwischen binären und analogen Signalen [165]. Binäre Signale sind zweiwertige Signale, die nur das Anliegen oder das Fehlen eines Signals als logische 1 bzw. 0 ausdrücken können (z. B. ob eine Lichtschranke offen oder geschlossen ist). Ein weit verbreiteter Industriestandard ist es, ein binäres Signal als elektrische Spannung mit 0 V als logische 0 und 24 V als logische 1 zu codieren. Analoge Signale sind kontinuierlich und können innerhalb bestimmter Grenzen jeden beliebigen Wert annehmen. Dabei werden oft Spannungs- und Stromsignale abgebildet [165]. Gebräuchlich sind Spannungssignale im Bereich 0 V – 10 V und Stromsignale im Bereich 0 mA – 20 mA bzw. 4 mA – 20 mA.

Feldbuskoppler und Signalbaugruppen

In der Automatisierungstechnik müssen binäre und analoge Signale nicht nur eingelesen, sondern zur Beeinflussung des technischen Prozesses durch eine Steuerung auch ausgegeben werden können. Damit eine computerbasierte, digitale Steuerung solche Signale schreiben und lesen kann, gibt es entsprechende Baugruppen. Durch eine Digitalausgabebaugruppe können 0 V bzw. 24 V als binäres 1-Bit-Signal geschaltet werden. Eine Digitaleingabebaugruppe kann die anliegende Spannung messen und durch eine Potenzialtrennung über einen Optokoppler in ein binäres 1-Bit-Signal umwandeln. Demzufolge wandeln Analogeingabebaugruppen analoge Spannungs- oder Stromsignale in ein digitales Signal mit 16 Bit Genauigkeit um. Eine Analogausgabebaugruppe setzt ein digitales Signal mit bis zu 16 Bit Genauigkeit in ein analoges Spannungs- oder Stromsignal um.

Um diese Baugruppen mit einer Robotersteuerung bzw. dem RCC zu verbinden, wird ein Feldbuskoppler benötigt (vgl. Abb. 6.6). Dieser verfügt über ein internes Bussystem, an dem die Baugruppen modular eingesteckt werden können. Damit kann ein Feldbuskoppler flexibel und den Bedürfnissen der Roboterzelle folgend ausgestattet werden. Der Koppler wiederum ist über ein industrielles Feldbussystem mit der Robotersteuerung verbunden und kann so mit dieser kommunizieren. Durch das Bussystem sind

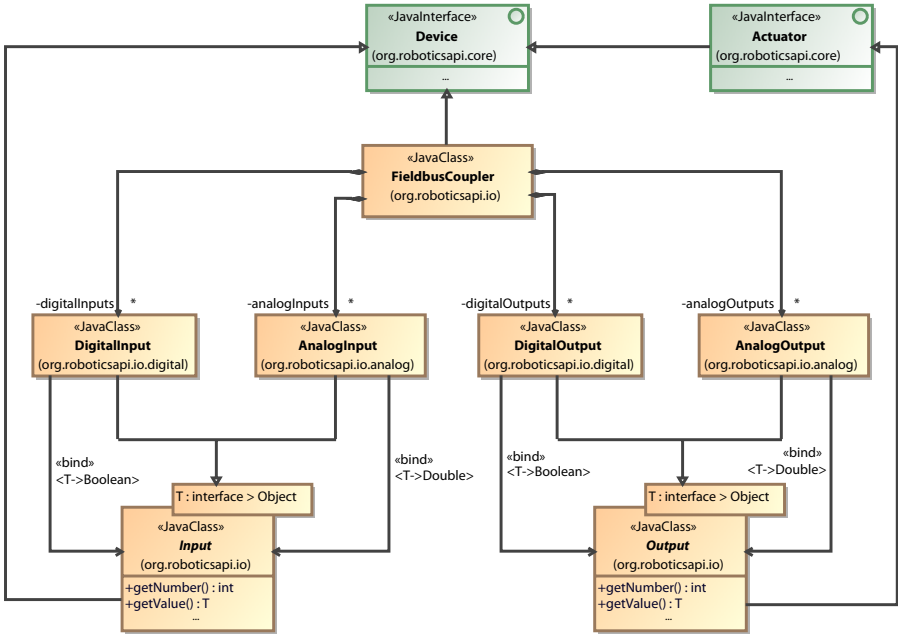


Abbildung 6.7: Ein Feldbuskoppler besteht aus einer konfigurierbaren Menge von binären und analogen Ein- und Ausgängen. Dadurch kann mit Roboterwerkzeugen oder Speicherprogrammierbaren Steuerungen kommuniziert werden.

alle Automatisierungskomponenten, darunter bspw. ein Koppler, seriell mit der Steuerung verdrahtet. Da über dieselbe Leitung alle Kommunikationsteilnehmer ihre Daten senden, definiert das Protokoll des Feldbussystems den Nachrichtenaustausch, der abhängig vom Protokoll echtzeitfähig ist. Beispiele für industrielle Feldbusse sind CAN [140] oder PROFIBUS [137]. Mittlerweile setzen sich immer öfters echtzeitfähige Ethernet-basierte Feldbusse wie z. B. EtherCAT [137] oder PROFINET [137] durch.

Für die Kommunikation mit Automatisierungskomponenten über binäre und analoge Signale wurden Feldbuskoppler in die *Robotics API* aufgenommen (vgl. Abb. 6.7). Ein Feldbuskoppler ist wie folgt definiert:

Definition 6.3. Ein FieldbusCoupler ist ein Device, das einen an die Robotersteuerung verbundenen Feldbuskoppler repräsentiert. Dieser verfügt in der Regel über eine Menge digitaler und analoger Ein- und Ausgänge, d. h. Inputs/Outputs (I/Os).

RCC als flexible SPS

Ein Feldbuskoppler ist über ein Feldbussystem mit der Echtzeitrobotersteuerung verbunden. Dadurch kann die *Robotics API* synchron zur Robotersteuerung binäre und analoge Signale schreiben. Zugleich können binäre und analoge Signale über den Feldbuskoppler eingelesen werden. Dadurch kann ein laufendes Kommando auf dem RCC ähnlich wie eine speicherprogrammierbare Steuerung (SPS) Automatisierungssysteme überwachen und steuern. Dazu werden binäre und analoge Signale über Eingangsbaugruppen in einem deterministischen Takt eingelesen und verarbeitet. Zum Abschluss eines Taktes werden Steuersignale über Ausgangsbaugruppen geschrieben. Damit kann über den RCC und das *Robotics Application Framework* die Funktionalität einer Roboterzelle vollständig abgedeckt werden.

Der FieldbusCoupler in der *Robotics API* besteht wie der *reale* Feldbuskoppler aus einer flexibel konfigurierbaren Menge von Eingängen und Ausgängen, die entweder binäre oder analoge Signale lesen bzw. schreiben

können. Der interne Aufbau ist in Abbildung 6.7 dargestellt. Der FieldbusCoupler verfügt als Device über einen Gerätetreiber. Der verwendete Feldbus (z. B. EtherCAT) kann über den Gerätetreiber konfiguriert werden. Dabei ist der Feldbus selbst ein Gerätetreiber – jedoch ohne dazugehöriges Gerät. Damit wird erreicht, dass ein Feldbus ebenfalls über das *Robotics Application Framework* konfiguriert und das dazugehörige *Realtime Device* geladen wird. Dieses *Realtime Device* kann von allen an diesem Feldbus angeschlossenen Geräten zur Kommunikation verwendet werden. Damit können bspw. mehrere Feldbuskoppler an einem Feldbus angeschlossen werden.

Binäre und analoge Eingänge

Der Gerätetreiber des FieldbusCouplers dient auch als *Factory* [97] für die Gerätetreiber der einzelnen Signale, die jeweils als separates Device bzw. bei Ausgängen als Actuator modelliert sind. Für Eingänge gibt es mit der Klasse Input eine gemeinsame Oberklasse, die entsprechend typisiert werden kann (vgl. [10]). Ein Input stellt sein Signal als RealtimeValue zur Verfügung, der beliebig in einem Command bzw. einer Activity verwendet werden kann (vgl. Abschn. 4.3.2). Zudem kann der aktuelle Wert abgefragt werden. Um ein binäres Signal einzulesen, existiert die Klasse DigitalInput, die wie folgt definiert ist:

Definition 6.4. Ein DigitalInput ist ein Device bzw. Input, der ein binäres Signal über einen digitalen Eingang einlesen kann und als booleschen Wert repräsentiert.

Für analoge Signale existiert demzufolge die Klasse AnalogInput, die wie folgt definiert ist:

Definition 6.5. Ein AnalogInput ist ein Device bzw. Input, der ein analoges Signal über einen analogen Eingang einlesen kann und als Gleitkommazahl repräsentiert.

Binäre und analoge Ausgänge

Für Ausgänge gibt es mit der Klasse Output eine gemeinsame Oberklasse, die entsprechend typisiert werden kann (vgl. [10]). Ein Output stellt – ähnlich wie ein Input – das aktuelle Signal als RealtimeValue zur Verfügung. Darüber hinaus kann der Wert des Signals über ein Command bzw. eine Activity geschrieben werden. Über die Komposition von Commands und Activity in der *Robotics API* können so komplexe Schaltaktionen abgebildet werden. Um ein binäres Signal zu schreiben, existiert die Klasse DigitalOutput, die wie folgt definiert ist:

Definition 6.6. Ein DigitalOutput ist ein Actuator bzw. ein Output, der einen digitalen Ausgang repräsentiert und über einen booleschen Wert ein binäres Signal schalten kann.

Für analoge Signale existiert demzufolge die Klasse AnalogOutput, die wie folgt definiert ist:

Definition 6.7. Ein AnalogOutput ist ein Actuator bzw. ein Output, der einen analogen Ausgang repräsentiert und über eine Gleitkommazahl ein analoges Signal schreiben kann.

In einer Roboterzelle können Feldbuskoppler und deren Ein- und Ausgänge verwendet werden, um mit industriellen Automatisierungskomponenten, d. h. mit Feldgeräten wie Sensoren und Aktuatoren, zu kommunizieren. Insbesondere die Programmierung von am Roboterflansch montierter Werkzeuge erfolgt oft über Ein- und Ausgänge und kann somit ohne großen Programmieraufwand im *Robotics Application Framework* abgebildet werden.

Dabei werden über analoge Eingänge Werkzeugparameter (z. B. eine Geschwindigkeit) eingestellt. Über binäre Eingänge kann wiederum eine Werkzeugaktion gestartet oder abgebrochen werden. Unter Zuhilfenahme der *Robotics API* kann man ein solches Werkzeug objektorientiert modellieren. Von der Programmierung mittels binärer und analoger Signale kann vollständig abstrahiert werden (vgl. Abschn. 6.4.2).

Durch die Integration von Feldbuskomponenten kann die Funktionalität, die eine SPS in einer Roboterzelle hat, durch das *Robotics Application Framework* vollständig nachgebildet werden. Infolgedessen ist eine SPS – mit Ausnahme sicherheitstechnischer Aufgaben – nicht notwendig. Im Vergleich zu einer SPS erfolgt die Programmierung im *Robotics Application Framework* jedoch auf einem höheren Abstraktionsniveau. Durch eine Kommunikation mit binären und analogen Signalen erfolgt in der Regel ein Verlust an Informationen bei der Kommunikation zwischen einer übergeordneten Zellensteuerung und einer Robotersteuerung. Hier werden üblicherweise nur Programme der Robotersteuerung angewählt, welche die Geometrie des Bauteils implizit beinhalten. Durch eine objektorientierte Modellierung kann das notwendige Wissen ohne Verlust von oben nach unten propagiert werden (vgl. Kap. 7).

6.4.2 *Integration von Parallelgreifern*

Zur flexiblen Handhabung von Gegenständen oder Bauteilen werden Roboter mit Greifern ausgestattet. In Abschnitt 5.5 wurde daher die Vererbungshierarchie für Roboterwerkzeuge und Greifer eingeführt. Eine Spezialisierung eines allgemeinen Greifers ist der Parallelgreifer, der über zwei parallel angebrachte Finger verfügt. Der Parallelgreifer ist als Schnittstelle ParallelGripper und als abstrakte Implementierung dieser Schnittstelle Teil der *Robotics API* (vgl. Abb. 6.8). Die Klasse AbstractParallelGripper vereinfacht als abstrakte Oberklasse die Implementierung eigener Parallelgreifer.

Merkmale von Parallelgreifern

Der allgemeine ParallelGripper definiert insbesondere die Eigenschaften und geometrischen Merkmale, die allen Parallelgreifern gemein sind. In Abbildung 6.8 werden – neben dem obligatorischen Basiskoordinatensystem – weitere Koordinatensysteme ausgewiesen, die signifikante Punkte des Greifers darstellen. Der *fingerCenter* beschreibt als Frame den Mittelpunkt zwischen den beiden Fingern. Dagegen beschreibt der *contactFrame* den zum Basiskoordinatensystem nächstgelegenen Punkt, der mit einem gegriffenen Gegenstand in Berührung kommt. Teilweise können Frames auch aus den Eigenschaften der Greiffinger berechnet werden. Dazu ist der Greiffinger – als GrippingFinger – ebenfalls Teil der *Robotics API* und beschreibt dessen Merkmale. Dazu gehört bspw. die Länge oder ein Koordinatensystem, das die Spitze des Fingers markiert. Damit wird das Greifspektrum der Finger vollständig beschrieben.

Die GrippingFinger sind als Schnittstelle realisiert und können entsprechend implementiert werden. Dadurch ist es möglich, beliebige Greiffinger einer Roboterzelle mithilfe der *Robotics API* abzubilden. Diese können entweder programmatisch oder über eine Konfiguration mit dem Parallelgreifer verknüpft werden (vgl. Abschn. 6.3). Zusätzlich können zu jedem Greiffinger spezifische Properties für die Visualisierung oder die Kollisionserkennung erzeugt und diesem hinzugefügt werden. Damit beschreibt der generische ParallelGripper in Kombination mit den frei konfigurierbaren Fingern die Eigenschaften und geometrischen Merkmale des Werkzeugs hinreichend.

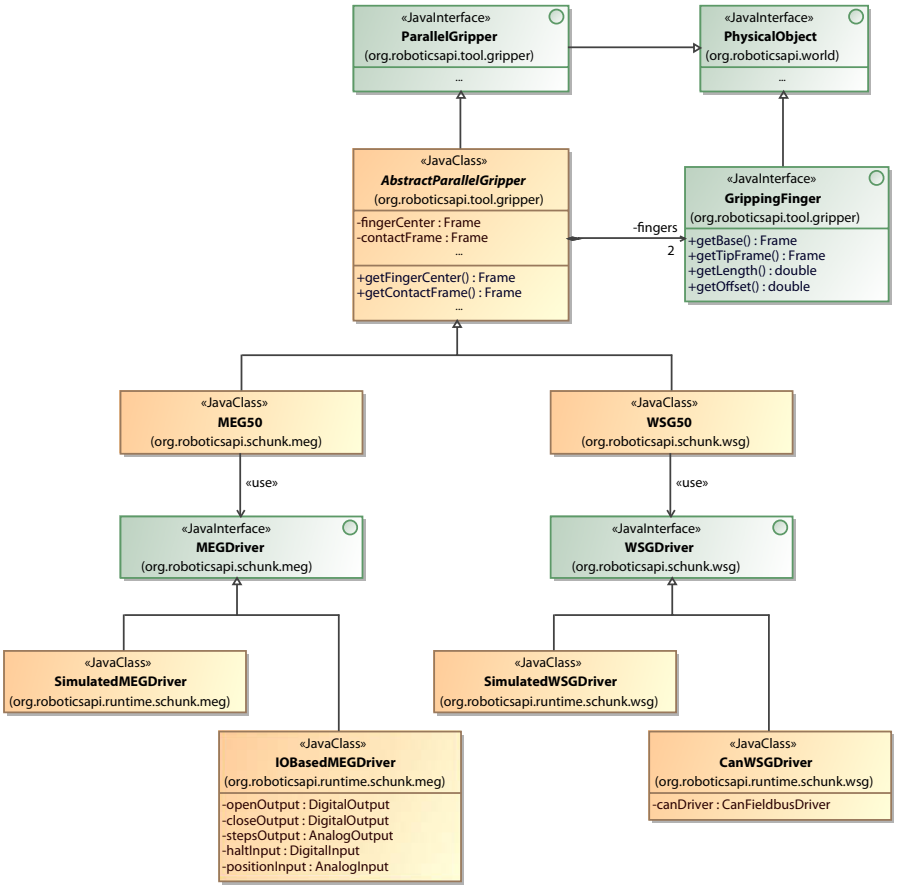


Abbildung 6.8: Die beiden Parallelgreifer leiten sich von der Oberklasse ParallelGripper ab und können gegeneinander ausgetauscht werden. Die Treiber sind spezifisch für den jeweiligen Greifertyp.

MEG-50 und WSG-50

Mit dem Schunk MEG-50 und dem Schunk WSG-50 sind in Abbildung 6.8 zwei konkrete Parallelgreifer dargestellt, die als Spezialisierung eines AbstractParallelGripper realisiert wurden. Sie verfügen beide über dieselben geometrischen Merkmale, definieren sie aber nach ihren Gegebenheiten und Abmessungen. Beide Greifer sind Teil einer eigenen Erweiterung des *Robotics Application Frameworks* und können unabhängig voneinander verwendet werden. Allerdings teilen sie sich die gemeinsamen Oberklassen und damit haben sie ähnliche Abhängigkeiten. Die Klasse MEG50 ist eine Spezialisierung eines ParallelGrippers und repräsentiert einen Schunk MEG-50. Analog dazu repräsentiert die Klasse WSG50 einen Schunk WSG-50. Es handelt sich dabei ebenfalls um einen speziellen ParallelGripper, der zusätzlich über Sensoren verfügt, um eine definierte Greifkraft aufzuwenden. Die beiden Parallelgreifer sind in Abbildung 6.9 dargestellt.

Unterschiedliche Treiber

Trotz der gemeinsamen Oberklasse und der dadurch gemeinsamen Eigenschaften bzw. Merkmale benötigen beide Greifer einen eigenen Treiber, da sie komplett unterschiedlich angesteuert werden. Während der WSG50 über CAN und einem eigenen Nachrichtenformat gesteuert wird, verwendet der MEG50 bzw. dessen externe Steuerung binäre und analoge Signale, die über einen Feldbuskoppler und entsprechende Baugruppen geschrieben und gelesen werden können (vgl. Abschn. 6.4.1). Die unterschiedlichen Gerätetreiber für beide Greifer sind in Abbildung 6.8 dargestellt.

(a) MEG-50 (b) WSG-50

Abbildung 6.9: In der *Robotics API* werden u. a. mit dem Schunk MEG-50 und dem Schunk WSG-50 zwei Parallelgreifer unterstützt. Beide abgebildeten Greifer können mit demselben manuellen Wechseladapter an den Roboterflansch montiert werden.

Der als MEGDriver bezeichnete Gerätetreiber für den Schunk MEG-50 unterteilt sich in zwei konkrete Gerätetreiber. Der IOBasedMEGDriver verwendet digitale und analoge Ein- und Ausgänge eines Feldbuskopplers, um den realen MEG-50 Parallelgreifer zu steuern (vgl. [243]). Dabei muss die Konfiguration der Ein- und Ausgänge der Verdrahtung in der Roboterzelle entsprechen. Der Greifer kann durch das Schreiben der Ausgänge gesteuert werden (vgl. auch Angerer [10]). Durch Eingänge können Rückmeldungen eingelesen werden. Durch das Setzen des Ausgangs *openOutput* öffnet sich der Greifer. Analog dazu schließt sich der Greifer durch das Setzen des Ausgangs *closeOutput*. Durch analoge Signale kann der Greifer zuvor parametrisiert werden (z. B. die Schrittweite).

Da sich dieser Treiber vollständig auf die Ein- und Ausgänge eines Feldbuskopplers abstützt, wird kein eigenes *Realtime Device* auf dem RCC benötigt. Dagegen verwendet der simulierte Gerätetreiber für den Schunk MEG-50, der SimulatedMEGDriver, ein *Realtime Device*. Es verhält sich auf dem RCC wie ein Feldbuskoppler und simuliert die externe Steuerung des Greifers. Dadurch werden die Rückmeldungen des Greifers, insbesondere die Position und Geschwindigkeit der Greiferbacken, realistisch abgebildet. Zudem werden die Parameter, die über analoge Signale gesetzt werden, berücksichtigt. Beide Treiber unterstützen neben dem Schunk MEG-50 auch die kleinere Ausführung, den MEG-40, und den größeren MEG-64, da deren Steuerung weitgehend identisch funktioniert.

Der als WSGDriver bezeichnete Gerätetreiber für den Schunk WSG-50 unterteilt sich ebenfalls in zwei konkrete Gerätetreiber. Um den realen Greifer anzusteuern, verwendet der CanWSGDriver proprietäre Nachrichten über einen CAN-Feldbus (vgl. [242, 279]). Der Feldbus wird ebenfalls über einen Gerätetreiber konfiguriert. Sowohl für den Gerätetreiber des WSG50 als auch den Feldbus existiert bzw. wird ein *Realtime Device* auf dem RCC geladen. Um den Greifer zu öffnen bzw. zu schließen und um Parameter zu setzen, werden über CAN spezifizierte Nachrichten gesendet. Analog zu einem simulierten Schunk MEG-50 imitiert der SimulatedWSGDriver die Steuerung des WSG-50. Dazu existiert ebenfalls ein *Realtime Device* für den RCC. Dieses benötigt jedoch keinen Feldbustreiber.

Identische Programmierung

Die beiden Greifer funktionieren auf eine vollständig unterschiedliche Art und Weise. Nicht nur die Kommunikation unterscheidet sich dabei, sondern auch die elementaren Befehle der Parallelgreifer. Während der MEG-50 über binäre Signale geöffnet und geschlossen werden kann, besitzt der WSG-50 native Steuerungsoperationen, die einen Verlust des Bauteils während des Greifens erkennen. Der MEG-50 kann zudem nur schrittweise geöffnet werden. Die Einstellung einer absoluten Öffnungsweite unterstützt die Steuerung nicht direkt und kann nur über eine komplexe Verschaltung der Signale erreicht werden.

Für beide Greifer wurden folgende Steuerungsoperationen implementiert:

- Öffnen und Schließen des Greifers bis zu einem Kontakt mit dem Gegenstand oder bis die Endschalter der Finger erreicht werden.
- Schrittweises Öffnen und Schließen des Greifers.
- Absolute Positionierung der Greiffinger, um eine definierte Öffnungsweite einzustellen. Der Versatz, der durch die Greiferbacken bedingt wird, wird dabei berücksichtigt.
- Positionsgesteuerte Greifoperation, die aktiv das kraftschlüssige Halten eines Gegenstandes überwacht.

Diese Steuerungsoperationen wurden mithilfe mehrerer ActuatorInterfaces umgesetzt und besitzen ein identisches Verhalten. Wie in Kapitel 6.1 beschrieben wurde, kann es von einem ActuatorInterface unterschiedliche, gerätespezifische Implementierungen geben. Dadurch ist es möglich das beide Parallelgreifer, die intern eine vollständig unterschiedliche Funktionsweise haben, über die *Robotics API* vollkommen identisch programmiert werden können.

Da beide Greifer über den abstrakten ParallelGripper die gleiche Menge an Merkmalen besitzen und identisch programmiert werden können, sind sie zu einem gewissen Maß austauschbar. Das bedeutet, dass die Aufgabe auf Basis des ParallelGrippers und der allgemeinen ActuatorInterfaces umgesetzt wird. Zur Laufzeit kann durch Polymorphie ein konkreter Greifer (z. B. ein MEG50 oder ein WSG50) konfiguriert und dafür eingesetzt werden. Experimentell konnte gezeigt werden, dass man Pick-and-Place-Aufgaben unabhängig von einem konkreten Greifer programmieren und mit verschiedenen Ausprägungen ausführen kann.

6.4.3 *Integration von Roboterarmen*

Zentrales Element jeder Roboterzelle ist ein Manipulator, der als flexibles und frei programmierbares Handhabungsgerät dient. In Abschnitt 5.5 wurde die Vererbungshierarchie für Manipulatoren eingeführt. Basis dafür ist ein MultiJointDevice, das prinzipiell einen Aktuator mit mehreren frei programmierbaren Gelenken definiert. Darauf aufbauend stellt ein RobotArm in der *Robotics API* eine serielle Kinematik dar, d. h. die Gelenke und Glieder sind hintereinander angebracht. Der Manipulator bildet eine offene kinematische Kette, die am Roboterflansch endet. Dort ist in der Regel der Endeffektor montiert. Die Klasse AbstractRobotArm vereinfacht als abstrakte Oberklasse die Implementierung eigener Manipulatoren.

Merkmale von Manipulatoren

Der allgemeine RobotArm definiert insbesondere die Eigenschaften und geometrischen Merkmale, die allen seriellen Robotern bzw. Manipulatoren

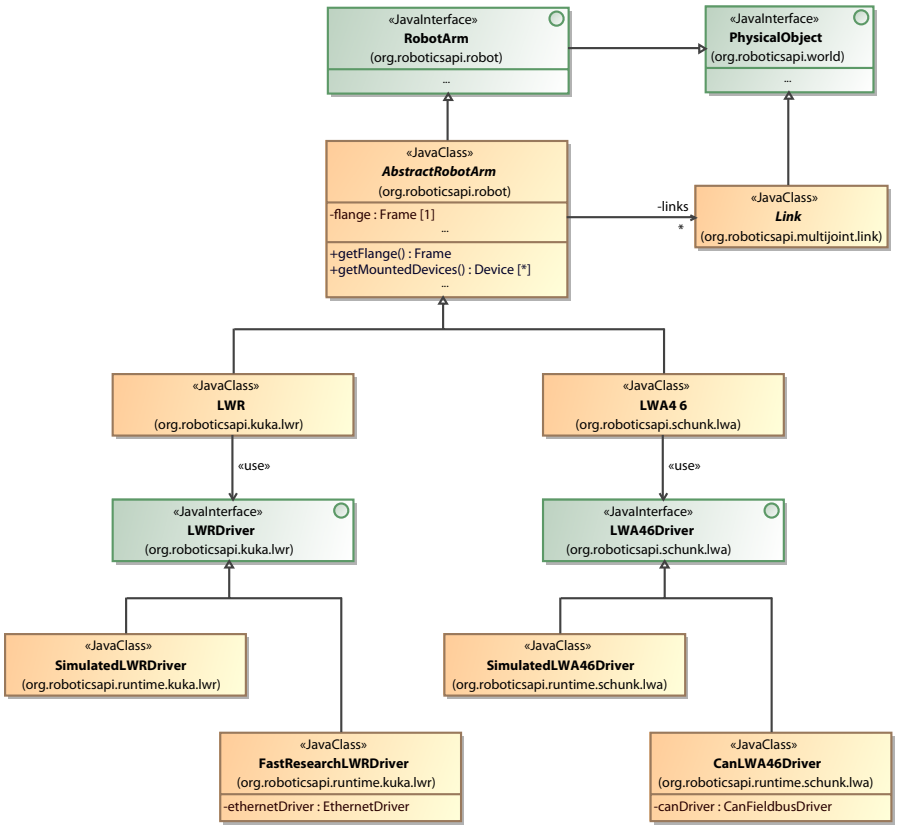


Abbildung 6.10: Die beiden Manipulatoren leiten sich von der Oberklasse RobotArm ab und können gegeneinander ausgetauscht werden. Die Treiber sind spezifisch für den jeweiligen Robotertyp.

gemein sind. In Abbildung 6.10 wird – neben dem obligatorischen Basiskoordinatensystem – das Flanschkoordinatensystem ausgewiesen, das einen signifikanten Punkt des Roboters darstellt. Da am realen Roboterflansch der Endeffektor montiert ist, wird über das Flanschkoordinatensystem auf die verbundenen Werkzeuge zugegriffen. Dazu gibt es in der *Robotics API* eigene Funktionen. Zudem ist jeder Motion Center Point (MCP), d. h. der zu bewegende Punkt einer Bewegung, mit dem Flanschkoordinatensystem verbunden.

Die Verbindung zwischen dem Basis- und dem Flanschkoordinatensystem wird durch die mechanische Struktur und die aktuelle Stellung des Roboters beeinflusst. Dazu spezifiziert jede konkrete Implementierung eines Manipulators die Anzahl und Beschaffenheit seiner Gelenke und seiner Glieder, die als Link in Abb. 6.10 dargestellt sind. Das betrifft insbesondere die Länge der Glieder und die Ausrichtung der Gelenke. Damit beschreibt der generische RobotArm die Eigenschaften und geometrischen Merkmale eines Manipulators hinreichend.

LWR 4 und LWA 4.6

Mit dem KUKA LBR 4 und dem Schunk Powerball Lightweight Arm LWA 4.6 sind in Abbildung 6.10 zwei Leichtbauroboter modelliert, die als Spezialisierung eines AbstractRobotArm realisiert wurden. Beide Roboter definieren die entsprechende Anzahl an Gelenken und Gliedern sowie deren Beschaffenheit, d. h. insbesondere deren Lage und Länge. Beide Roboter sind Teil einer eigenen Erweiterung des *Robotics Application Frameworks* und können unabhängig voneinander verwendet werden. Allerdings teilen sie sich die gemeinsamen Oberklassen und damit haben sie ähnliche Abhän-

(a) LWR 4 (b) LWA 4.6

Abbildung 6.11: In der *Robotics API* werden u. a. mit dem KUKA LWR 4 und dem Schunk LWA 4.6 zwei Manipulatoren unterstützt.

gigkeiten. Die Klasse LWR repräsentiert einen KUKA Leichtbauroboter (LBR) der 4. Generation (vgl. [33]). In Ergänzung zu einem normalen RobotArm hat er Sensoren in jedem Gelenk, um die extern auf das Gelenk wirkenden Drehmomente zu messen. Der Schunk Powerball Lightweight Arm (LWA) 4.6 wird durch die Klasse LWA46 repräsentiert. Die beiden Roboter sind in Abbildung 6.11 dargestellt.

Unterschiedliche Treiber

Trotz der gemeinsamen Oberklasse und der dadurch gemeinsamen Eigenschaften bzw. Merkmale benötigen beide Manipulatoren einen eigenen Treiber, da sie auf dem RCC unterschiedlich angesteuert werden. Während der LWR über Ethernet mit einer proprietären Schnittstelle gesteuert wird, verwendet der LWA eine standardisierte Kommunikationsschnittstelle für elektrische Antriebe. Beide Schnittstellen definieren jedoch, dass dem Roboter zyklisch neue Stützpunkte für die Gelenke gesendet werden müssen. Die Zykluszeit variiert dabei – je nach Konfiguration – zwischen 1 ms und 12 ms. Die unterschiedlichen Gerätetreiber der beiden Roboter sind in Abbildung 6.10 dargestellt.

Der als LWRDriver bezeichnete Gerätetreiber für den KUKA Leichtbauroboter unterteilt sich in zwei Ausprägungen. Der FastResearchLWRDriver verwendet das eigens für die Forschung entwickelte, proprietäre Fast Research Interface (FRI) [240]. Dabei werden über UDP dem Roboter bzw. seiner nativen Steuerung zyklisch neue Stützpunkte mitgeteilt. Außerdem wurde die Schnittstelle so erweitert, dass es möglich ist den verwendeten Regler und dessen Parameter zu verändern. Da der LBR neben der klassischen Positionsregelung auch einen gelenkbasierten und einen kartesischen Impedanzregelmodus [6] unterstützt, sind kraftgesteuerte und nachgiebige Bewegungen möglich. Für die Kommunikation mit dem LBR wird ein eigenes *Realtime Device* verwendet, welches wiederum auf einem *Realtime Device* für den echtzeitfähigen Netzwerksocket aufbaut (vgl. Abb. 6.10). Zur Simulation existiert mit dem SimulatedLWRDriver ein eigener Gerätetreiber.

Der als LWA46Driver bezeichnete Gerätetreiber für den Schunk Powerball LWA unterteilt sich ebenfalls in zwei verschiedene Ausprägungen. Um den realen Roboter anzusteuern, verwendet der CanLWA46Driver ein stan-

dardisiertes Nachrichtenformat (vgl. [135]) über CANopen [56]. CANopen ist ein auf CAN basierendes Kommunikationsprotokoll in der Automatisierungstechnik. Der CAN-Feldbus wird ebenfalls über einen Gerätetreiber konfiguriert. Sowohl für den Gerätetreiber des LWA46 als auch den Feldbus existiert bzw. wird ein *Realtime Device* auf dem RCC geladen. Analog zum LBR simuliert der SimulatedLWA46Driver einen Lightweight Arm. Dazu existiert ebenfalls ein *Realtime Device* für den RCC. Dieses benötigt jedoch keinen CAN-Feldbus.

Identische Programmierung

Mit beiden Manipulatoren wird auf unterschiedlichste Art und Weise kommuniziert. Der Inhalt der übertragenen Nachrichten ist wie die generelle Funktionsweise jedoch sehr ähnlich. Beide Roboter werden über eine zyklische Positionssteuerung auf Gelenkebene gesteuert. Größter Unterschied sind die zusätzlichen Impedanzregelmodi des KUKA Leichtbauroboters, die das Äquivalent von Schunk nicht unterstützt. Um dem irgendwie nahezukommen, müsste der LWA mit einer Kraftmessdose am Roboterflansch ausgestattet sein. Dadurch wären kraftgesteuerte und nachgiebige Bewegungen ebenso möglich.

Durch die zyklische Positionssteuerung auf Gelenkebene können folgende Steuerungsoperationen für beide Leichtbauroboter richtig parametrisiert verwendet werden:

- Punkt-zu-Punkt-Bewegungen, deren Ziel als Achsstellung sowie als kartesische Position angegeben werden kann.
- Lineare Bewegung des MCP zu einem Zielpunkt

Diese Steuerungsoperationen wurden mithilfe mehrerer ActuatorInterfaces umgesetzt. Dadurch ist es möglich, dass beide Manipulatoren über die *Robotics API* vollkommen identisch programmiert werden können.

Da beide Manipulatoren über den abstrakten RobotArm die gleichen Merkmale besitzen und identisch programmiert werden können, sind sie zu einem gewissen Maß austauschbar. Das bedeutet, dass eine Aufgabe auf Basis eines RobotArms und unter Zuhilfenahme von ActuatorInterfaces umgesetzt werden kann. Zur Laufzeit wird ein konkreter Roboter (z. B. ein LWR oder ein LWA46) konfiguriert und verwendet. Experimentell konnte gezeigt werden, dass Pick-and-Place-Aufgaben unabhängig von einem konkreten Manipulator programmiert und mit verschiedenen Ausprägungen eines RobotArms ausgeführt werden konnten (vgl. auch Abschn. 11.4). Dies ist die Grundlage dafür, dass man wiederverwendbare und parametrierbare Programmfragmente erstellen kann. Diese können als flexibler Baustein in einer serviceorientierten Roboterzelle verwendet werden.

Allerdings müssen alle Zielpunkte als kartesisches Koordinatensystem (d. h. als Frame) Roboter-unabhängig definiert sein. Zudem müssen diese Punkte jeweils von den eingesetzten Roboter erreichbar sein, d. h. die Zielpunkte müssen sich im gemeinsamen Arbeitsraum der Roboter befinden. Idealerweise sollten Roboter mit ähnlichem Arbeitsraum gewählt werden. Zu beachten ist dabei, dass eine reine Angabe von Zielpunkten keine Roboterkonfiguration definiert. Somit befindet sich zwar der Roboterflansch bzw. der MCP am definierten Zielpunkt, jedoch kann die Konfiguration des Roboters beliebig gewählt sein. Dies kann verhindert werden, indem ein Frame um eine HintJointProperty erweitert wird. Die HintJointProperty definiert Roboter-spezifisch eine optimale Gelenkstellung zum Erreichen des Zielpunktes. Eine Alternative dazu ist die Verwendung eines kollisionsfreien Bahnplaners (vgl. Abschn. 6.5.3), der verschiedene Zielkonfiguration berücksichtigt und bewertet.

6.5 BEISPIELE HORIZONTALER ERWEITERUNGEN

Dieser Abschnitt beschreibt exemplarisch horizontale Erweiterungen des *Robotics Application Frameworks*. Die vorgestellten Beispiele zeigen die vielfältigen Möglichkeiten des Ansatzes und werden im weiteren Verlauf der Arbeit verwendet, um eine serviceorientierte Roboterzelle zu implementieren. In Abschnitt 6.5.1 wird zuerst eine Erweiterung beschrieben, die es erlaubt den Zustand einer Roboterapplikation anhand von 3D-Modellen zu visualisieren. Die Möglichkeit unterstützt den Applikationsentwickler, Unterschiede zwischen der realen Zelle und der Implementierung zu erkennen. Außerdem ist in Kombination mit simulierten Treibern eine Offlineprogrammierung serviceorientierter Roboterzellen möglich.

Das objektorientierte Modell kann nicht nur zur Visualisierung der Roboterzelle verwendet werden, sondern darüber hinaus auch zur kollisionsfreien Planung komplexer Roboterbewegungen. Dazu wird jedes PhysicalObject der Roboterzelle als potentieller Kollisionskörper angesehen (vgl. Abschn. 6.5.2). Anhand der in der Applikation bekannten Position und Orientierung können alle Objekte der Roboterzelle gegenseitig auf Kollisionen überprüft werden. Dafür kann man erprobte Physik-Engines verwenden, die eine effiziente Kollisionserkennung für starre Körper implementieren. Die Erkennung von Kollisionen anhand des Modells kann in einem Bahnplaner (vgl. Abschn. 6.5.3) verwendet werden, um kollisionsfreie Trajektorien für Manipulatoren der *Robotics API* zu planen. Dadurch wird die Flexibilität von Roboterzellen weiter erhöht.

6.5.1 *Visualisierung*

Die *Robotics API* verfügt über ein umfangreiches objektorientiertes Modell, um die Elemente einer Roboterzelle und deren geometrische Abhängigkeiten zu beschreiben (vgl. Kap. 5). Dieses Modell stellt einen Ausschnitt des aktuellen Zustands einer Roboterapplikation dar und setzt sich aus der Menge der aktuell vorhandenen Elemente der Roboterzelle und deren geometrischer Position in der Zelle zusammen. Es repräsentiert den Zustand der realen Roboterzelle in der Software und dient demzufolge als Grundlage für die Planung von Roboterbewegungen und Werkzeugaktionen. Umso präziser dieses Modell ist und je mehr es mit der Realität übereinstimmt, umso einfacher ist die Planung zukünftiger Bewegungen und Aktionen.

Um in einer Applikation den aktuell angenommenen Zustand der Roboterzelle darzustellen, kann eine dreidimensionale Visualisierung verwendet werden. Das objektorientierte Modell der *Robotics API* bietet dafür eine gute Grundlage, da es bereits sehr ähnlich zu einem Szenengraphen ist. Ein solcher Szenengraph [37] ist in der Computergrafik die zentrale Datenstruktur, die die Objekte und deren Position in der virtuellen Welt beschreibt. Daher wird diese konzeptionelle Nähe ausgenutzt, um die virtuelle Roboterzelle darzustellen.

Darstellung einer Roboterzelle

Es ist naheliegend, dass alle physikalische Objekte, d. h. Instanzen der Klasse PhysicalObject, in einer virtuellen Roboterzelle dargestellt werden. Diese Objekte besitzen per Definition eine physikalische Ausprägung und damit eine Gestalt, die angezeigt werden kann. Jedoch sollten auch Koordinatensysteme und deren Verbindungen, d. h. Frames und Relations, angezeigt werden. Koordinatensysteme besitzen zwar keine physikalische Gestalt, sind aber das zentrale geometrische Konzept in der Robotik und für die Programmierung von Robotern unverzichtbar. Eine Visualisierung der

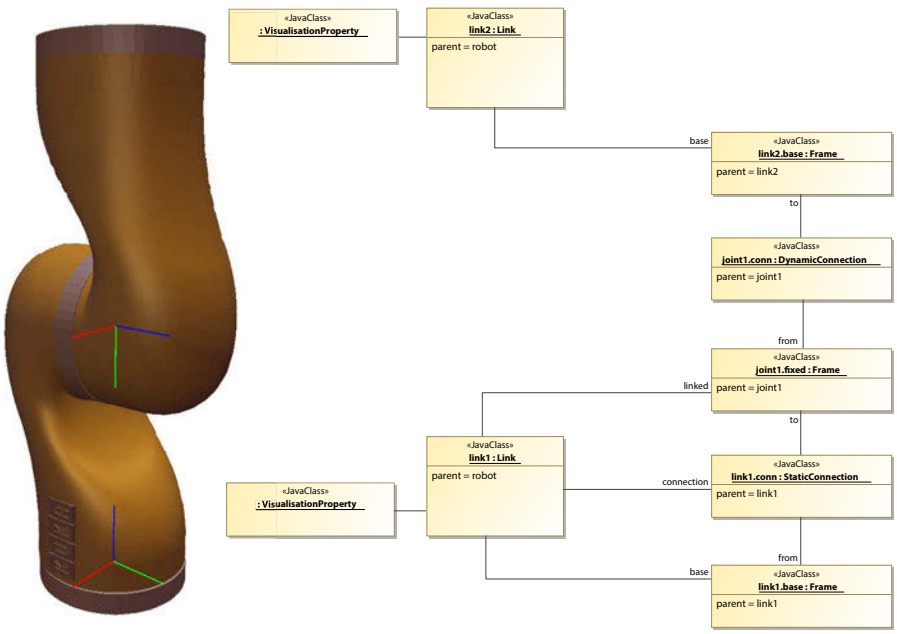


Abbildung 6.12: Durch ein VisualisationProperty können 3D-Modelle an ein PhysicalObject gebunden werden. Sobald das Objekt in den beobachteten Graphen aus Frames und Relations eingefügt wird, erscheint das Modell im Szenengraphen der Visualisierung.

Koordinatensysteme in Kontext zu den vorhandenen Geräten und Objekten einer Roboterzelle ist ein große Unterstützung bei der Programmierung von Roboterapplikationen.

Für die Visualisierung eines PhysicalObject wird ein Gitternetzmodell oder *Mesh* benötigt. Um ein Gitternetzmodell bei Bedarf, d. h. wenn eine Visualisierung gewünscht wird, zur Verfügung zu stellen, wird das in Abschnitt 6.1 vorgestellte Konzept von generischen Eigenschaften bzw. Properties genutzt. Dementsprechend wurde für die Visualisierung eine eigene Eigenschaft, die VisualisationProperty, eingeführt. Sie beschreibt das Gitternetzmodell u. a. als COLLADA-Datei, einem offenen XML-basiertem Austauschformat für 3D-Daten [25]. Eine VisualisationProperty kann jedem PhysicalObject über einen RoboticsObjectListener hinzugefügt werden (vgl. Abschn. 6.1). Dadurch kann man jedes beliebige PhysicalObject bei Bedarf um ein Gitternetzmodell erweitern.

Abbildung 6.12 zeigt auf der linken Seite einen Teil eines Roboters in der Visualisierung. Auf der rechten Seite wird als Objektdiagramm der entsprechende Ausschnitt der RobotArm-Instanz dargestellt. Sowohl beide Links als auch ein Teil der Frames (d. h. beide Basiskoordinatensysteme) werden in der Visualisierung angezeigt. Für die Links wird das Gitternetzmodell über ein VisualisationProperty bereitgestellt. Frames benötigen keine eigene VisualisationProperty, sondern werden standardmäßig als farbiges Koordinatensystem mit drei Vektoren dargestellt.

Aufbau eines Szenengraphen

Um einen Szenengraph für die Visualisierung zu erzeugen, wird ausgehend von einem Frame der Graph aus Frames und Relations analysiert und ein Spannbaum erzeugt (vgl. Abschn. 5.4). Dadurch wird über den ungerichteten Graph aus Frames und Relations eine baumförmige Datenstruktur gelegt, die ausgehend von einem Wurzelknoten alle verbundenen Frames und damit auch alle PhysicalObjects enthält. Diese Datenstruktur kann einfach visualisiert werden, denn trotz seines Namens ist ein Szenen-

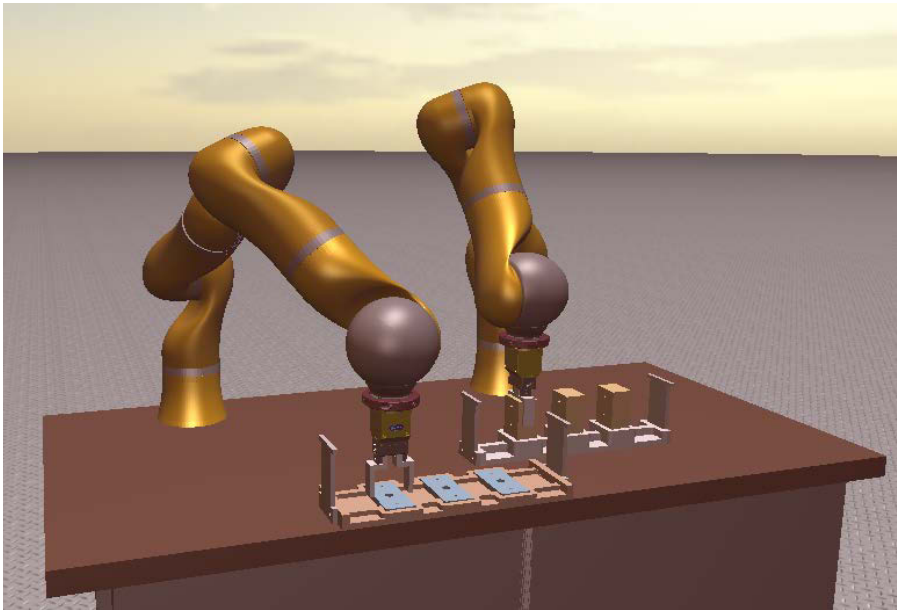


Abbildung 6.13: Die Visualisierung stellt alle Objekte der Roboterzelle dar und damit den internen Zustand einer Roboterapplikation. In dem Beispiel werden neben den Robotern und Greifern auch die Werkstücke und die Werkbank abgebildet.

graph ebenfalls ein gerichteter Baum, dessen Wurzelelement die Gesamtszene enthält. Dementsprechend kann aus dem Spannbaum und den darüber identifizierten PhysicalObjects ein Szenengraph aufgebaut werden (vgl. Abb. 6.12). Der Spannbaum bzw. der Szenengraph der *Robotics API* wird gemäß dem *Observer*-Pattern [97] beobachtet, um Änderungen festzustellen und die Visualisierung anzupassen. Das betrifft einerseits die Topologie der Szene, d. h. es werden neue Objekte dargestellt und nicht mehr vorhandene Objekt gelöscht. Andererseits wird die Postion und Orientierung aller Frames und PhysicalObjects bei einer Änderung in der Visualisierung aktualisiert.

Folglich entsprechen die in der Visualisierung dargestellten Informationen dem Zustand, den die Applikation in der Roboterzelle erwartet. Abbildung 6.13 zeigt eine virtuelle Roboterzelle mit zwei Manipulatoren, Endeffektoren und den Handhabungsobjekten auf einer Werkbank. Solange die echte Roboterzelle noch nicht existiert, kann das virtuelle Modell verwendet werden, um die Roboterzelle offline zu programmieren. Dazu kann man die in Abschnitt 6.3 vorgestellten simulierten Treiber benutzen. Später, wenn die echte Roboterzelle aufgebaut wurde, müssen nur die simulierten Treiber ersetzt werden. Nun kann man den visualisierten Zustand während der weiteren Online-Programmierung mit der Wirklichkeit abgleichen, um Fehler zu vermeiden. Nach der Inbetriebnahme kann die Visualisierung weiter verwendet werden, um den aktuellen Zustand der Roboterzelle fortlaufend mit dem angenommenen Zustand abzugleichen und zu kontrollieren. Die Visualisierung ist in *Eclipse* [77], einer integrierten Entwicklungsumgebung für Java, eingebettet, für die ebenfalls ein Plug-in [59] zur Programmierung mit der *Robotics API* existiert (vgl. [10]). Eclipse basiert ebenfalls auf OSGi und bietet mit der Rich Client Platform (RCP) [175] eine modulare Plattform zur Entwicklung von Anwendungen mit graphischer Benutzerschnittstelle. Die Plattform stellt ein Grundgerüst bereit, das um eigene Anwen-

Integration der Visualisierung

dungsfunktionalitäten erweitert werden kann. Da sich sowohl das *Robotics Application Framework* als auch die Eclipse Rich Client Platform auf OSGi abstützen, kann das *Robotics Application Framework* als Teil einer graphischen Anwendung ausgeführt werden. So kann der Zustand einer Roboterzelle direkt in der Visualisierung angezeigt und aktualisiert werden. Für andere Anwendungsfälle laufen die Visualisierung und die Roboterapplikation in jeweils einer eigenen JVM und kommunizieren über Remote Method Invocation (RMI) miteinander.

Somit bietet die Visualisierung eine einfache aber umfassende Methode, um den Zustand einer Roboterzelle weitreichend darzustellen (vgl. Abschn. 6.13). Dabei nutzt die Visualisierung die objektorientierte Modellierung der *Robotics API*. Die Erweiterung um eine VisualisationProperty ist unabhängig von der Modellierung eines konkreten PhysicalObjects und erfolgt ausschließlich bei Bedarf. Durch die Visualisierung ist es möglich, das *Robotics Application Framework* zur Offline-Programmierung von Roboterzellen zu verwenden (vgl. [103, 187]). Darüber hinaus erleichtert die Visualisierung aber auch während der Online-Programmierung dem Entwickler den Abgleich zwischen erwartetem Zustand im Programm und der Realität in der Zelle.

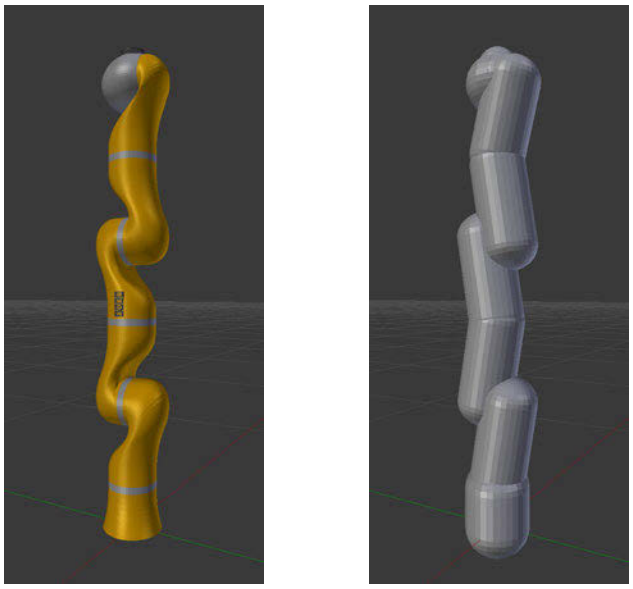
6.5.2 *Kollisionserkennung*

Für die Erkennung von Kollisionen wird ein PhysicalObject analog zur Visualisierung durch eine generische Property mit einer Kollisionshülle erweitert. Dementsprechend wurde für die Kollisionserkennung CollisionHullProperty eingeführt, um die Kollisionshülle eines PhysicalObjects zu beschreiben. Zur effizienten Berechnung besteht eine Kollisionshülle jedoch nicht aus einem komplexen Drahtgittermodell, sondern setzt sich aus primitiven Formen (d. h. Quadern, Kapseln und Kugeln) zusammen. Durch eine geeignete Kombination dieser Formen kann ein PhysicalObject approximiert werden. Damit Kollisionen zuverlässig erkannt werden, wird die Gestalt eines PhysicalObjects in der Regel überapproximiert, d. h. die Kollisionshülle wird im Vergleich zur Realität größer gewählt.

Approximation durch Kollisionshüllen

Abbildung 6.14 zeigt einen KUKA Leichtbauroboter mit seiner Darstellung als Drahtgittermodell sowie seiner Kollisionshülle. Die Kollisionshüllen sind im Gegensatz zum Drahtgittermodell aus Kapseln und Kugeln definiert. Durch die Überapproximierung der Hüllen kann es vorkommen, dass benachbarte physikalische Objekte, die dynamisch miteinander verbunden sind, in der Kollisionswelt kollidieren. Dies wird in der Realität durch mechanische Einschränkungen (z. B. maximale Gelenkstellungen) unterbunden. Ebenso kann es auch zu gewollten Kollisionen kommen, wie dem Schließen eines Greifers, bei dem die Greiferfinger das Werkstück formschlüssig fassen. In beiden Fällen sind die Kollisionen nicht zu verhindern.

Um diese inhärent stattfindenden Kollisionen zu ignorieren, können zwei PhysicalObjects als *Freunde* definiert werden und auftretende Kollisionen zwischen diesen Freunden werden ignoriert. Ein Beispiel für Freunde sind zwei benachbarte Links eines Roboters. Um dies festzulegen, wird eine weitere Property, die CollisionFriendsProperty, definiert. Dadurch werden Objektpaare definiert, zwischen denen keine Kollisionen berechnet und gemeldet werden soll. Wie die CollisionHullProperty kann auch die CollisionFriendsProperty über einen RoboticsObjectListener zu Objekten (z. B. zu einem Joint, der zwei Links verbindet) hinzugefügt werden.



(a) Drahtgittermodell (b) Kollisionshülle

Abbildung 6.14: Im Gegensatz zu einem Drahtgittermodell approximiert eine Kollisionshülle die Gestalt des Manipulators mit einfachen geometrischen Körpern und ermöglicht so eine effiziente Kollisionserkennung.

Eine weitere Ausnahme sind physikalische Objekte, die statisch miteinander verbunden sind, d. h. ausschließlich über eine oder mehrere StaticConnections. Dadurch kann sich die Lage der Objekte zueinander nicht verändern. Das bedeutet, dass zwei statisch verbundene Objekte entweder immer kollidieren, z. B. ein Werkstück, das auf einem Tisch liegt, oder niemals, z. B. zwei Roboter, die entfernt voneinander auf einem Tisch montiert sind. Diese als *Nachbarn* bezeichneten Objekte werden automatisch erkannt und ebenfalls als Ausnahmen gekennzeichnet. Endet die statische Verbindung zwischen beiden PhysicalObjects, endet auch die Nachbarschaft und Kollisionen werden wieder gemeldet.

Anbindung einer Physik-Engine

Zur Kollisionserkennung wurde eine generische Schnittstelle definiert, über die verschiedene Algorithmen oder Bibliotheken angeschlossen werden können. Zur Evaluierung wurde die *Bullet Physics Library* [51] verwendet. Es handelt sich dabei um eine Physik-Engine zur Simulation starrer Körper, die in Computerspielen aber auch für visuelle Effekte in Filmen verwendet wird. Bullet verfügt als Physik-Engine sowohl über eine diskrete als auch eine stetige Kollisionserkennung und ist als Open-Source-Software frei nutzbar. Für die Kollisionserkennung mit Bullet wird eine *Kollisionswelt* erstellt, die alle relevanten Objekte mit der jeweiligen Gestalt und der aktuellen Position enthält.

Die Kollisionserkennung in Bullet verläuft zweistufig. In der *Broad Phase* wird mithilfe von *Bounding Volumes* paarweise überprüft, welche Objekte sich überlagern können. Ein *Bounding Volume* besitzt dabei eine – selbst im Vergleich zu einer Kollisionshülle – sehr einfache Struktur (z. B. ein Quader) und approximiert die geometrische Gestalt des Körpers noch stärker. Zudem können in der *Broad Phase* Filter aktiviert werden, damit Objektpaare nicht überprüft werden (z. B. für die oben definierten Ausnahmen). Falls sich zwei *Bounding Volumes* schneiden, werden in der *Narrow Phase* die Kollisionhüllen auf Schnittpunkte überprüft, wobei dies mit hohem Rechen-

aufwand verbunden sein kann. Das Erkennen einer Kollision führt zu einer Benachrichtigung.

Aufbau einer Kollisionswelt

Ausgehend vom Ursprungskoordinatensystem der Roboterzelle wird ein FrameGraph aufgebaut (vgl. Abschn. 5.4). Alle sich in diesem Graphen befindlichen PhysicalObjects werden – sofern sie über eine Kollisionshülle verfügen – erfasst. Ein PhysicalObject bildet zusammen mit seiner Kollisionshülle ein eigenständiges Kollisionsobjekt und wird der oben beschriebenen *Kollisionswelt* hinzugefügt. Jede Änderung des PhysicalObjects wird überwacht und der *Kollisionswelt* mitgeteilt (z. B. eine Positionsänderung). Dadurch bleibt das Modell der Roboterzelle und der *Kollisionswelt* synchron. Somit können Kollisionsabfragen einfach formuliert und für die weitere Planung verwendet werden.

6.5.3 *Kollisionsfreie Bahnplanung*

Die Konfiguration einer Roboterzelle ist ausreichend, um Kollisionen in der *Robotics API* zu erkennen. Dazu müssen lediglich alle Geräte und Gegenstände der Zelle mit einer Kollisionshülle ausgestattet werden. Durch Mechanismen der *Robotics API* wird die erzeugte *Kollisionswelt* – auch bei Bewegungen der Roboter oder Änderungen der Topologie – aktuell gehalten. Diese einfache Möglichkeit der Kollisionserkennung kann verwendet werden, um Kollisionen a priori zu vermeiden. Dies ist insbesondere bei der Planung und Ausführung von Roboterbewegungen interessant. Die Trajektorien eines Manipulators müssen nicht mehr während der Entwicklung oder der Inbetriebnahme einer Roboterzelle fest definiert, sondern können dynamisch zur Laufzeit erzeugt werden, was die Flexibilität einer serviceorientierten Roboterzelle enorm erhöht.

Die Planung kollisionsfreier Bewegungen wird bei vielen Ansätzen im Konfigurationsraum $\mathbf{C}$ [145] durchgeführt, der bei einem Manipulator der Menge aller Gelenkstellungen entspricht. Die Dimension von $\mathbf{C}$ entspricht der Anzahl der Gelenke. Aufgrund der Komplexität wird keine Transformation der Hindernisse aus dem kartesischen Arbeitsraum in den Konfigurationsraum vorgenommen. Stattdessen wird die Form des kollisionsfrei erreichbaren Konfigurationsraums $\mathbf{C}_{\text{free}}$ durch *Sampling*, d. h. durch zufällige Stichproben, angenähert [145]. Um effiziente Mehrfachanfragen zu ermöglichen, bauen *Multi-Query-Planner* in einem ersten Schritt einen approximierten frei erreichbaren Konfigurationsraum $\mathbf{C}_{\text{free}}$ auf. Anschließend kann bei einer Anfrage diese Approximation verwendet werden, um einen Pfad zu finden. Für veränderliche Umgebungen eignen sich *Single-Query-Planner* wie die von LaValle [163] vorgeschlagenen Rapidly-exploring Random Trees (RRTs). Dabei wird der kollisionsfreie Konfigurationsraum $\mathbf{C}_{\text{free}}$ stichprobenartig durch einen Baum approximiert. Die Besonderheit bei RRTs ist, dass die Ausbreitungsrichtung sehr stark in Richtung des noch unbekannten Konfigurationsraums tendiert [163].

Bidirektionaler RRT

Auf Basis der *Robotics API* wurde ein bidirektionales RRT-Verfahren [153] implementiert. Hier werden zwei Bäume gleichzeitig aufgebaut, wobei der erste Baum bei der Startkonfiguration $\mathbf{q}_{\text{init}}$ und der zweite Baum bei der Zielkonfiguration $\mathbf{q}_{\text{goal}}$ beginnt. Iterativ wird eine zufällige Konfiguration $\mathbf{q}_{\text{rand}}$ gewählt und versucht, diese mit dem ersten Baum zu verbinden. Falls dies gelingt, wird versucht, $\mathbf{q}_{\text{rand}}$ mit dem zweiten Baum zu verbinden. Falls auch dies gelingt, sind die Startkonfiguration $\mathbf{q}_{\text{init}}$ und die Zielkonfiguration $\mathbf{q}_{\text{goal}}$ miteinander verbunden und es kann ein kollisionsfreier Pfad bestimmt werden. Andernfalls werden die beiden Bäume vertauscht, und eine neue

Iteration mit einer neu gewählten Konfiguration q_{rand} wird gestartet. Der gefundene Pfad besteht aus einer Liste von Konfigurationen q, die nacheinander angefahren werden müssen.

Um zu überprüfen, ob eine Konfiguration q kollisionsfrei ist, wird die im vorherigen Abschnitt vorgestellte Kollisionserkennung verwendet. Dazu wurde der FrameGraph, der für den Aufbau einer Kollisionswelt zuständig ist, erweitert, um eine Roboterkonfiguration q zu simulieren. Dazu repräsentiert die Kollisionswelt die aktuelle Roboterzelle bis auf eine festgelegte Menge von Relations, die z. B. die DynamicConnections zwischen den Gliedern eines Roboters darstellen (vgl. Abschn. 5.4). Für diese Relations können beliebige Transformationen definiert werden und damit auch die Roboterkonfiguration q. Die Kollisionswelt nimmt daraufhin die gewählte geometrische Struktur an und überprüft alle Elemente auf Kollisionen. Durch dieses allgemeine Verfahren können beliebige Zustände einer Roboterzelle simuliert und auf Kollisionen überprüft werden.

Optimierung des gefundenen Pfads

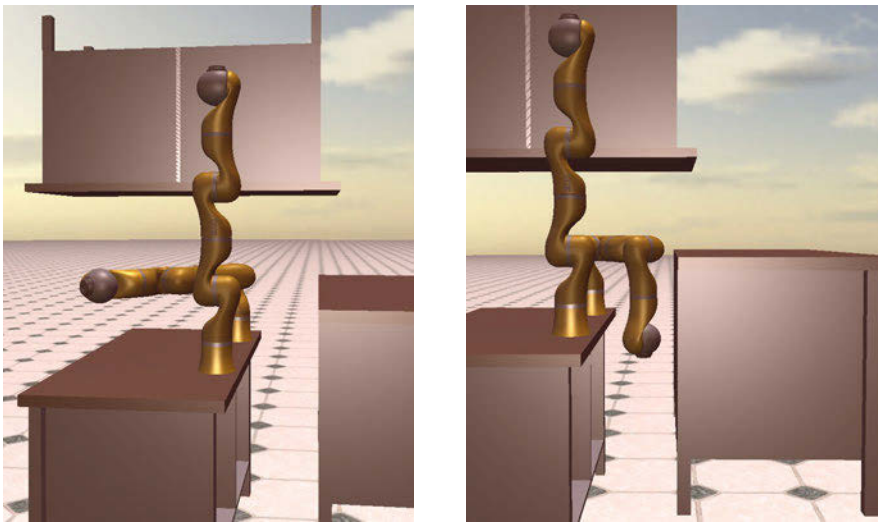
Der durch die bidirektionalen RRTs ermittelte Pfad entspricht aufgrund des probabilistischen Verfahrens nicht zwangsläufig dem kürzesten Weg zwischen der Startkonfiguration q_{init} und der Zielkonfiguration q_{goal}. Daher wurden Verfahren entwickelt, um den gefundenen Pfad zu optimieren. Dazu werden zuerst alle Konfigurationen q entfernt, die auf einer geraden Linie zweier anderer Konfigurationen des Pfades liegen. Dadurch kann die Länge des Pfades signifikant verkürzt werden, was zu einer gleichförmigen und schnelleren Bewegung des Roboters führt. Anschließend werden kollisionsfreie *Abkürzungen* im Pfad gesucht. Dabei wird der Pfad Konfiguration für Konfiguration durchgegangen und überprüft, ob zwischen der aktuellen Konfiguration q_i und einer Konfiguration q_j mit $i < j$ eine direkte und kollisionsfreie Verbindung besteht. Falls dies der Fall ist, können alle Konfigurationen zwischen q_i und q_j aus dem Pfad entfernt werden.

Zum Schluss wird der Pfad für ein *Überschleifen* aller Konfigurationen zwischen q_{init} und q_{goal} angepasst. Das heißt, dass diese Konfigurationen nicht exakt angefahren werden, sondern, ohne abzubremsen, bereits vorher auf eine Bahn in Richtung der nächsten Konfiguration gewechselt wird. Dadurch entstehen deutlich schnellere und den Roboter schonendere Bewegungen. Allerdings kann es vorkommen, dass der überschliffene Pfad nicht mehr kollisionsfrei ist. Dies wird geprüft und der Pfad kann entsprechend angepasst werden. Durch diese drei Optimierungen entstehen effizientere und trotzdem immer noch kollisionsfreie Bahnen.

Durch eine Evaluation konnte gezeigt werden, dass ein optimierter Pfad in akzeptabler Zeit gefunden wird (vgl. [276]). In der Evaluationsumgebung, die in Abbildung 6.15 dargestellt ist, konnte ein optimierter Pfad in durchschnittlich 1642 ms bei insgesamt 30 Durchläufen gefunden werden[2]. Die Optimierungsverfahren konnten die Pfadlänge von ca. 700 Konfigurationen auf unter 10 Konfigurationen reduzieren. Die durch die Optimierung in Anspruch genommene Zeit wird durch die signifikante Reduktion der Pfadlänge deutlich kompensiert. Während ein nicht optimierter Pfad mit ca. 700 Konfigurationen etwa 50 s für die Ausführung benötigt, kann der optimierte Pfad in unter 4 s ausgeführt werden. Grund für diese stark unterschiedliche Ausführungszeit sind die vielen Beschleunigungs- und Bremsphasen des nicht optimierten Pfades.

Zusammenfassend lässt sich festhalten, dass auf Basis des objektorientierten Modells der *Robotics API* eine effiziente Bahnplanung realisiert werden konnte. Diese nimmt den aktuellen Zustand der Roboterzelle und plant als

2 Dabei wurde ein Intel Core i7-3537U mit 2.00 GHz und 8 GB RAM verwendet.



(a) Startposition (b) Endposition

Abbildung 6.15: Die beiden Ausschnitte der Visualisierung zeigen die Start- und Endposition des Manipulators bei der durchgeführten Evaluation. Letztendlich kann ein optimierter Pfad mit einer Länge von ca. 10 Konfigurationen gefunden werden. Jede Konfiguration wird dabei durch einen PTP angefahren (vgl. [276]).

Single-Query-Planner eine kollisionsfreie Trajektorie für einen Manipulator. Das Verfahren kann nicht nur für einen Manipulator angewandt werden, sondern ist auf Roboterteams übertragbar. Durch die Möglichkeit der kollisionsfreien Bahnplanung sind flexible Bewegungen und damit flexible Fertigungsprozesse in einer Roboterzelle möglich.

6.6 VERWANDTE ARBEITEN

Um die Komplexität von großen Robotikanwendungen in den Griff zu bekommen, haben Hägele et al. [108] die Aufteilung eines Systems in kleinere Komponenten und die „composition of subsystems“ [108, S. 984] als die wichtigsten Anforderungen identifiziert. Das bedeutet, dass Robotersysteme aus kleineren, wiederverwendbaren Einheiten mit standardisierten Schnittstellen zusammengesetzt werden können. Dies betrifft neben Aktuatoren und Sensoren insbesondere auch die Software. Diese Aufteilung wird durch die meisten Robotik-Frameworks adressiert, da sie als komponentenbasiertes System entworfen wurden. Bekannte Beispiele sind Player [60], ROS [223], YARP [90] oder Orca [171]. Dies hat zu einem Trend für komponentenbasiertes Software Engineering in der Robotik geführt [48].

Robotik-Frameworks können als Middleware angesehen werden, die die Kommunikation zwischen einzelnen Komponenten übernimmt. Player verwendet Schnittstellen, um von einer Komponentenimplementierung zu abstrahieren. Dagegen nutzt ROS zum Beispiel Serviceanfragen und Topicbasierte Nachrichten über ein *Publish-Subscribe*-Verfahren [97] für die Kommunikation zwischen verschiedenen Prozessen, die *Nodes* genannt werden. Da die *Nodes* nur Nachrichten austauschen, sind sie lose gekoppelt und verlassen sich nur auf eine gemeinsame Spezifikation der Nachrichten.

Weil diese Systeme eine Zusammenstellung lose gekoppelter Komponenten darstellen, sind sie per se erweiterbar. Jedoch unterstützen die meis-

ten komponentenbasierten Robotik-Frameworks keine harten Echtzeitanforderungen bei der Kommunikation zwischen einzelnen Komponenten. Daher ist es schwierig, die erforderliche Genauigkeit und Performanz der Industrierobotik zu erreichen. Eine Ausnahme stellt Orocos [50] dar, welches auch in der Referenzimplementierung des RCC verwendet wurde (vgl. Abschn. 4.3.1). Durch die *Robotics API* und dem entsprechenden *Runtime Adapter* wurde eine erweiterbare und einfach bedienbare Programmierschnittstelle oberhalb von Orocos hinzugefügt. Dabei wurden die Abhängigkeiten der drei verschiedenen Bestandteile ganzheitlich betrachtet.

Nach Brugali und Scandurra [49] wird Wiederverwendung in der komponentenbasierten Softwareentwicklung erreicht, indem die Spezifikation (d. h. die bereitgestellten und benötigten Schnittstellen) und die Implementierung einer Komponente getrennt werden. Dadurch können sich Implementierungen in funktionalen vor allem aber in nicht-funktionalen Eigenschaften unterscheiden. Da Komponenten als *Black Boxes* verwendet werden, die nur durch ihre Schnittstellen kommunizieren, können Implementierungen zudem gegeneinander ausgetauscht und mehrfach verwendet werden (vgl. Abschn. 2.1). Die meisten Robotik-Frameworks erreichen Wiederverwendung auf diese Art. Dabei ist es im Gegensatz zu dem in dieser Arbeit verwendeten Ansatz nicht möglich, dass Funktionalität durch Vererbung – über mehrere Komponenten hinweg – wiederverwendet werden kann.

Das Komponentenframework CLARAty [191], zur Programmierung autonomer Roboter, verwendet ebenso Vererbung um Erweiterbarkeit und Wiederverwendung zu erreichen. Dabei unterstützt CLARAty, das vom Jet Propulsion Laboratory entwickelt wurde, eine echtzeitfähige Ausführung von Roboteroperationen auf einer objektorientierten Ebene, verbirgt die Echtzeitprogrammierung jedoch nicht vor dem Applikationsentwickler. Eine genauere Aussage zur Wartbarkeit des Frameworks und der damit implementierten Software werden von Nesnas et al. [191] nicht gemacht. Weitere Ansätze, die ebenfalls objektorientierte Konzepte für Wiederverwendung benutzen, sind MARS [177] und RIPE [182]. Allerdings betrachten auch dort die Autoren nicht explizit die Wartbarkeit der Softwaresysteme. Mithilfe von OSGi wird im *Robotics Application Framework* eine *White-Box*-Wiederverwendung stark vereinfacht. Durch die Deklaration einer öffentlichen Programmierschnittstelle können (abstrakte) Implementierung einfach und gezielt zur Verfügung gestellt werden. Die Abhängigkeiten werden durch das *OSGi Framework* automatisch aufgelöst.

Luo et al. [168] haben eine auf OSGi basierende Softwareevolution für Robotersysteme vorgestellt. Dabei benutzen sie „Behavior Networks“ [169], wobei jedes *Behavior* (Verhalten) als Bundle repräsentiert wird. Somit können neue *Behaviors* und neue Netzwerke als neues Bundle zur Laufzeit im *OSGi Framework* installiert werden. Jedoch behandeln sie weder die Wiederverwendbarkeit noch die echtzeitfähige Ansteuerung von Robotersystemen.

Georgas und Taylor [101] haben verschiedene Softwarearchitekturen in der Robotik untersucht und herausgearbeitet, ob und wie diese Softwarearchitekturen Änderungen zur Laufzeit unterstützen. Sie kommen zu dem Schluss, dass sowohl ein Mangel an ausgezeichneten Schichten in der Architektur als auch eine fehlende Kapselung von Verhalten die größten Hemmnisse sind, damit sich Robotersysteme zur Laufzeit verändern bzw. weiterentwickeln können. Jedoch werden genau diese Punkte in dieser Arbeit und im modularen *Robotics Application Framework* berücksichtigt.

Teil III

SERVICEORIENTIERTE ROBOTERZELLEN

Der dritte Teil der Arbeit beschreibt die serviceorientierte Modellierung von Roboterzellen. In dieser Arbeit wird dazu die Vision einer intelligenten Fabrik aufgezeigt, die aus einer Menge serviceorientierter Roboterzellen besteht. Diese stellen ihre Fertigungsaufgaben als Service zur Verfügung. Intern ist eine Roboterzelle ebenfalls aus einer Menge von Services aufgebaut. Die Dienste sind auf unterschiedlichen Abstraktionsebenen jeweils am Fertigungsprozess ausgerichtet und haben dabei immer und das Bauteil im Fokus.

Infolgedessen kann die Automatisierungssoftware einer Roboterzelle wie nach einem Baukastenprinzip aus einzelnen wiederverwendbaren Services zusammengestellt werden. Um diesen Vorgang zu vereinfachen, wird eine Modellierung von Roboterzellen mit SysML vorgestellt. Das Wissen über den Herstellungsprozess ist jedoch im Bauteil gebündelt und wird auf allen Ebenen durch die Dienste verwendet, um eine optimale Fertigung zu ermöglichen. Durch die hier vorgestellten Konzepte und Ideen leistet diese Arbeit einen Beitrag, um anpassbare Automatisierungssoftware für flexible Roboterzellen zu entwickeln.

7 KONZEPT SERVICEORIENTIERTER ROBOTERZELLEN

Industrie 4.0 ist ein Zukunftsprojekt der deutschen Bundesregierung [52], um die Vision einer vernetzten und intelligenten Produktion der Zukunft voranzutreiben. Ein zentrales Element ist die intelligente Fabrik (*Smart Factory*) [144], in der „Menschen, Maschine und Ressourcen so selbstverständlich wie in einem sozialen Netzwerk“ [144, S. 23] miteinander kommunizieren. Durch intelligente Fertigungsmaschinen und Betriebsmittel entstehen cyber-physische Produktionssysteme (CPPS) [144], die gemeinsam eine intelligente Fabrik mit einer neuen Produktionslogik bilden. Anstatt starrer Abläufe verfügen intelligente Produkte über „das Wissen ihres Herstellungsprozesses“ [144, S. 23] und unterstützen den Fertigungsprozess, indem sie bspw. wissen, mit welchen Parametern sie bearbeitet werden müssen. Die Produkte sind nicht nur identifizierbar und lokalisierbar, sondern kennen auch ihren aktuellen Zustand und wissen, wie ihr Zielzustand, d. h. das fertige Produkt, erreicht wird.

In dieser Arbeit wird eine Vision entwickelt, wie sich serviceorientierte Roboterzellen in den Kontext einer intelligenten Fabrik der Zukunft eingliedern. Dabei besteht eine intelligente Fabrik aus einer Menge serviceorientierter Roboterzellen, die jeweils unterschiedliche Fertigungsdienstleistungen anbieten. Diese Dienstleistungen werden durch Schnittstellen und Kontrakte beschrieben und können durch intelligente Produkte in Anspruch genommen werden. Somit wird der aktuelle Zustand des Produkts sukzessive transformiert, um letztendlich den gewünschten Zielzustand zu erreichen. Das Wissen über den Herstellungsprozess jedoch ist Teil des Produkts und wird vom Service verwendet, um eine optimale Produktion zu gewährleisten. Diese aufgezeigte, generelle Idee serviceorientierter Roboterzellen als Teil einer intelligenten Fabrik wird in Abschnitt 7.1 vorgestellt.

Anschließend wird in Abschnitt 7.2 erläutert, wie eine Roboterzelle intern aus einer Menge von Services flexibel strukturiert werden kann. Dabei wird die Granularität der Services von der externen Schnittstelle der Roboterzelle immer weiter verfeinert, bis schlussendlich Roboter und Endeffektoren gesteuert werden. Diese serviceorientierte Modellierung und Strukturierung leitet thematisch auf die folgenden Kapitel dieser Arbeit über. Einige der in diesem Kapitel beschriebenen Konzepte und Ideen wurden erstmals in [120] und [121] vorgestellt.

7.1 SERVICES ALS EXTERNE SCHNITTSTELLE VON ROBOTERZELLEN

Durch das *Internet der Dinge und Dienste* entsteht eine Vielzahl an autonomen, eingebetteten Systemen, die drahtlos untereinander und mit dem Internet vernetzt sind. Durch die Vernetzung bilden sich cyber-physische Systeme [100], die sich durch eine Verknüpfung softwaretechnischer Komponenten mit mechanischen und elektronischen Teilen auszeichnen. Durch ein Cyber-Physical System (CPS) werden aus Alltagsgegenständen intelligente Objekte, die programmierbar sind, mittels Sensoren physikalische Daten erfassen und kommunizieren. Diese Objekte können eigenständig Informationen austauschen, Aktionen auslösen und sich wechselseitig steuern.

In der intelligenten Fabrik kommunizieren Produkte, Bauteile und Betriebsstoffe untereinander und mit Fertigungsmaschinen, um ein Cyber-Physical Production System (CPPS) zu bilden. Sie stellen ein Äquivalent zu den oben erwähnten cyber-physischen Systemen in der Produktion dar. Das bedeutet auch, dass es keinen festen Ablauf in der Fertigung gibt, der in der Automatisierungspyramide von oben nach unten propagiert wird. Stattdessen handeln die zu fertigenden Produkte, die notwendigen Betriebsstoffe und die vorhandenen Produktionsressourcen eigenständig den Fertigungsprozess aus, sie tauschen Informationen aus und steuern sich gegenseitig.

Eine Roboterzelle kann von außen betrachtet als cyber-physisches Produktionssystem angesehen werden. Sie besteht aus miteinander vernetzten, eingebetteten Systemen, die über Sensoren Daten erfassen und mithilfe von Aktuatoren auf physikalische Vorgänge einwirken. Die Roboterzelle verfügt über eine Menge von Softwarekomponenten, um die einzelnen Systeme miteinander zu koordinieren und so Produktionsprozesse abzubilden. Die Roboterzelle kommuniziert als cyber-physisches Produktionssystem mit den Bauteilen und Produkten, um deren optimale Handhabung und Bearbeitung zu bestimmen. Das Wissen über die Handhabungs- und Fertigungsprozesse ist Teil des Bauteils, das es mit der Roboterzelle teilen kann.

Serviceorientierte Architekturen sind verteilte Systeme, die Mechanismen zur transparenten Kommunikation der einzelnen Dienste anbieten. Dadurch sind Dienste in der Regel gekapselt und plattformunabhängig. Die Anbieter und Nutzer von Diensten können in unterschiedlichen Technologien und auf verschiedenen Plattformen realisiert sein. In diesem Kontext ist eine Roboterzelle ein Service-Provider, d. h. es werden eine Menge an Dienstleistungen, die die Handhabung und Bearbeitung von Bauteilen und Produkten betreffen, bereitgestellt. Die Roboterzelle exponiert dementsprechend alle Fertigungsprozesse, die sie durchführen kann, als Service.

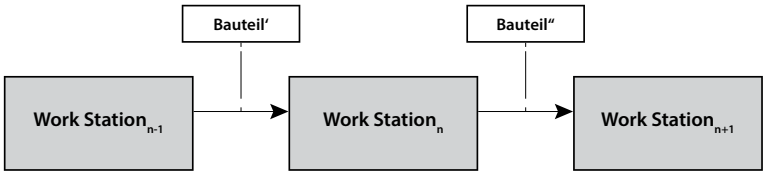
Robotics Manufacturing as a Service

Diese Dienstleistungen können daraufhin in Anspruch genommen werden. Dies setzt jedoch voraus, dass die Fertigungsprozesse der Roboterzelle flexibel auf die Bedürfnisse der Bauteile anpassbar sind, die den Dienst in Anspruch nehmen. Daher ist es notwendig, dass die Roboterzelle als (cyber-physisches) Produktionssystem und das Bauteil als intelligentes Objekt miteinander kommunizieren. Jede, von einer Roboterzelle angebotene, Dienstleistung verändert ein Bauteil, indem es mindestens einen Bearbeitungsschritt anwendet. Es transformiert den softwaretechnischen und den realen Zustand des Bauteils. Durch diese Dualität spiegelt die Softwarerepräsentation das reale Bauteil wider. Durch eine geeignete Reihenfolge von Fertigungsdienstleistungen können so aus den Ausgangsmaterialien fertige Bauteile oder Produkte entstehen.

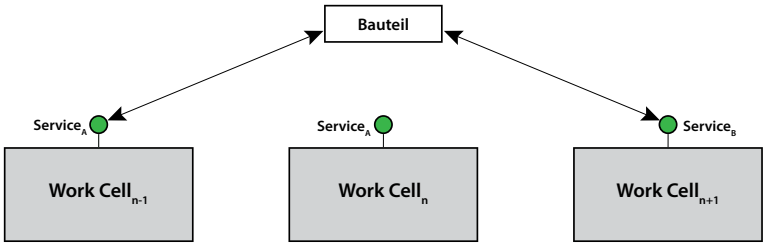
Gemäß dem von Banerjee et al. [23] beschriebenen Paradigma *Everything as a Service* kann man kann in diesem Zusammenhang von *Robotics Manufacturing as a Service* sprechen:

Definition 7.1. Bei *Robotics Manufacturing as a Service* werden die möglichen Fertigungsaufgaben einer Roboterzelle als Dienstleistung angeboten, die von einem (noch zu fertigenden) Produkt in Anspruch genommen werden kann.

Dazu führen die Produkte und Bauteile eine Beschreibung von sich selbst mit. Diese beinhaltet Informationen über das Aussehen des Werkstücks, geometrische Merkmale und Fertigungsparameter. Daraus kann eine serviceorientierte Roboterzelle die von ihr angebotenen Dienste konfigurieren, die intern jeweils einen bestimmtem Handhabungs- und Fertigungsprozess parametrisieren und ausführen. Wie ein Service die fachliche Funktionalität



(a) Fließfertigung



(b) Werkstattfertigung

Abbildung 7.1: Aktuell sind Roboterzellen oft Bestandteil einer zeitlich streng getakteten Fließfertigung. Eine *Smart Factory* dagegen verfügt über Roboterzellen, die über Dienste unterschiedliche Fertigungs- und Handhabungsaufgaben anbieten. Der Aufbau einer *Smart Factory* entspricht weitgehend dem Werkstattprinzip.

eines Geschäftsprozesses in einem Unternehmen kapselt, so kapselt eine serviceorientierte Roboterzelle die fachliche Funktionalität eines Fertigungsprozesses in einer intelligenten Fabrik. Die Dienste stellen genauso wie die Roboterzelle eine abgeschlossene Einheit dar.

Vision einer serviceorientierten Fabrik

Die Struktur einer intelligenten Fabrik ist dabei nicht mehr nach dem Fließbandprinzip aufgebaut (vgl. Abb. 7.1 (a)). Die Roboterzellen sind nicht wie in der Automobilfertigung in einer festen Reihenfolge und mit einer festen Taktzeit angeordnet. Stattdessen stellt jede Roboterzelle für sich eine abgeschlossene Einheit dar. Gemäß dem Werkstattprinzip [241] werden die Bauteile zu der Werkstatt bzw. der Roboterzelle gebracht und dort bearbeitet (vgl. Abb. 7.1 (b)). Im Unterschied zu der klassischen Werkstattfertigung bieten die Roboterzellen ihre Dienstleistungen in der ganzen Fabrik als Service an. Daher kann der Dienst flexibel in Anspruch genommen werden. Idealerweise wird das Werkstück oder die Werkstücke autonom (z. B. über eine mobile Transportplattform) an die Zelle gebracht.

Der Servicekontrakt stellt die softwaretechnische Schnittstelle dar. Diese kann in einer *Registry* (vgl. Abschn. 2.2) veröffentlicht und von überall in der Fabrik angefragt werden. Damit kann ein Werkstück einen Plan an Diensten, die verwendet werden müssen, erstellen. Zudem kann ein Bauteil – bevor es sich an oder in der Nähe der Roboterzelle befindet – die softwaretechnische Schnittstelle anfragen und so die Verfügbarkeit der Roboterzelle und ihrer Dienstleistungen erfragen. Allerdings ist der Servicekontrakt nicht die einzige Schnittstelle einer serviceorientierten Roboterzelle.

Neben dem Servicekontrakt als softwaretechnische Schnittstelle gibt es noch zusätzlich die mechanischen Schnittstellen einer Roboterzelle. Diese definieren, wie die Bauteile und Betriebsstoffe in die Zelle und damit zu den Robotern und Handhabungsgeräten gelangen. Obwohl Roboter flexible Handhabungsgeräte sind, ist das Spektrum an möglichen Zuliefervarianten dennoch begrenzt. Daher muss die mechanische Schnittstelle adäquat in

die softwaretechnische Schnittstelle integriert werden. So können bspw. die Argumente der Serviceaufrufe die mechanischen Schnittstellen zum Teil beschreiben. Sie definieren, wie die Roboterzelle mit ihrer Umgebung, d. h. ihren Zuliefersystemen, interagiert. Daher bietet es sich an, die möglichen Zuliefervarianten (z. B. Förderbänder, Werkstückträger, mobile Transportplattformen) als Teil des Servicekontrakts zu beschreiben.

Die konsequente Umsetzung einer serviceorientierten Architektur kann die Vision der intelligenten Fabrik ermöglichen. Die Fertigungsressourcen der Fabrik (d. h. Roboterzellen, Fertigungsmaschine) sind über Internettechnologien miteinander vernetzt. Dabei können die Dienste bspw. als Web Service [42] bereitgestellt werden. Erste eigene Experimente mit dem Devices Profile for Web Services (DPWS) [76] waren hierbei sehr vielversprechend. Die Werkstücke wiederum sind die Akteure in der Fabrik, da sie die Fertigungsprozesse anstoßen und dabei transformiert werden. Das Werkstück kann dabei selber agieren oder von einem Softwareagenten vertreten werden, der die Randbedingungen und Kosten der Fertigung berücksichtigt.

7.2 SERVICEORIENTIERTE REALISIERUNG VON ROBOTERZELLEN

Bei der Modellierung serviceorientierter Architekturen nehmen die zugrundeliegenden Geschäftsprozesse des Unternehmens eine zentrale Rolle ein, da Services bzw. Dienste fachliche Funktionalität kapseln und damit einen Teil des Geschäftsprozesses repräsentieren. Ein wesentliches Ziel ist eine hohe Wiederverwendbarkeit von Diensten, um Geschäftsprozesse flexibel und kostengünstig anpassen bzw. neu aufsetzen zu können. Einzelne Dienste wiederum können kombiniert werden und so, entsprechend orchestriert, komplexere fachliche Funktionalität repräsentieren. Darüber hinaus ist es für die Nutzung eines Dienstes ausreichend, nur die Schnittstelle und den Kontrakt zu kennen.

Bei dem internen Aufbau einer Roboterzelle aus Services wird in dieser Dissertation ein ähnliches Konzept verfolgt. Die Dienste sollten in sich abgeschlossen sein und über wohldefinierte Schnittstellen verfügen. Dabei sollten sie immer eine am *Geschäftsprozess* der Roboterzelle ausgerichtete Funktion haben. Der Geschäftsprozess einer Roboterzelle ist der dort durchgeführte Fertigungsschritt oder Automatisierungsprozess. Dies muss die Grundlage jeder serviceorientierten Roboterzelle sein. Der Automatisierungsprozess kann dann schrittweise heruntergebrochen und verfeinert werden, um die notwendigen Services zu identifizieren. Letztendlich ergibt sich eine Hierarchie von Services, die in einer geeigneten Orchestrierung die Fertigungsprozesse der Roboterzelle abbilden.

Hierarchie von Services

Die Hierarchie der Services hat unterschiedliche Ebenen mit einer unterschiedlichen Granularität, was die fachliche Funktionalität des Dienstes anbelangt. Dabei haben Dienste auf der selben Ebene die gleiche Granularität, kapseln jedoch unterschiedliche Funktionalität. Die fachliche Funktionalität, die jeder Service kapselt, wird in Form einer Schnittstelle oder eines Kontraktes anderen Diensten angeboten. Dementsprechend ergibt sich analog zu serviceorientierten Architekturen in Unternehmen (vgl. Krafzig et al. [150]) eine Klassifikation von Diensten einer Roboterzelle, welche die Dienste in unterschiedliche Kategorien einteilt.

In einer klassischen Roboterzelle ist überlicherweise eine SPS für die Steuerung der Abläufe zuständig, wohingegen die Robotersteuerung für die Bewegungen des Manipulators und die Ansteuerung des Werkzeugs verantwortlich ist. Durch die Strukturierung mit unterschiedlichen Kategorien von

SICHTWEISE	BESCHREIBUNG
Produkt (*product-centric*)	Beschreibung der Aufgabe durch die Angabe des Zielzustandes eines Werkstückes und der einzelnen Bestandteile.
Prozess (*process-centric*)	Beschreibung der Aufgabe als ein Ablauf elementarer Handhabungs- und Fertigungsaufgaben.
Werkzeug (*tool-centric*)	Beschreibung der Aufgabe aus Sicht des Werkzeugs, das elementare Operationen bereitstellt und entsprechend parametrisiert werden muss.
Roboter (*arm-centric*)	Beschreibung der Aufgabe als Sequenz kartesischer Roboterbewegungen mit entsprechenden Schaltaktionen.
Gelenk (*joint-centric*)	Beschreibung der Aufgabe als Sequenz von Gelenkstellungen.

Tabelle 7.1: Abstraktionsebenen für die Programmierung von industriellen Robotersystemen gemäß [108]. Während auf hoher Ebene die Aufgaben auf Basis des zu fertigenden Produkts definiert werden, werden sie auf unterster Ebene als elementare Roboterbewegungen definiert.

Diensten ergibt sich analog zu einer klassischen Roboterzelle eine klare Verteilung der Zuständigkeiten. Allerdings entsteht durch die sehr rudimentäre Kommunikation zwischen einer SPS und einer Robotersteuerung (vgl. Kap. 3) ein Informationsbruch, der zu einer Verteilung des Fertigungswissens führt. So ist bspw. die Geometrie eines Bauteils nur auf unterster Ebene implizit durch das Roboterprogramm vorhanden. Dies macht eine Adaption schwierig, da an vielen verschiedenen Stellen das dort vorhandene Wissen erweitert bzw. angepasst werden muss. In einer serviceorientierten Roboterzelle findet dagegen kein Informationsbruch statt und das gesamte Wissen kann in Form einer Bauteilbeschreibung von oben nach unten propagiert werden, wobei jeder Dienst nur die für ihn notwendigen Daten verwendet.

Jede Kategorie von Diensten spiegelt auch eine mögliche Ebene einer SOA wider. Umso mehr Ebenen von einer SOA in einer Roboterzelle abgedeckt werden, umso höher ist der Reifegrad der Architektur und die Hebelwirkung bei der flexiblen Umsetzung der Fertigungsprozesse. Ziel ist es, eine Sammlung von Services verschiedener Ebenen zu entwickeln, die flexibel in unterschiedlichen Roboterzellen eingesetzt und zu Fertigungsprozessen orchestriert werden können. Letztendlich hängt die maximal erzielbare Hebelwirkung vom Grad an Wiederverwendbarkeit der Dienste ab. Zudem ist es möglich, Services durch andere Implementierungen zu ersetzen, ohne dass die darüber liegenden Ebenen geändert werden müssen.

Abstraktionsebenen der Roboterprogrammierung

Hägele et al. [108] definieren unterschiedliche Abstraktionsebenen für die Programmierung von industriellen Robotersystemen, die in Tabelle 7.1 beschrieben sind. Die niedrigste Abstraktionsebene ist die Programmierung des Roboters auf Gelenkebene (vgl. Abschn. 3.2). Dabei wird jede Aufgabe in Form einer Achsbewegung als Sequenz von Gelenkstellungen beschrieben. Die gemäß [108] höchste Abstraktionsebene ist das Produkt, d. h. es wird der initiale Zustand und Zielzustand eines Bauteils spezifiziert und der Roboter bzw. die Roboterzelle setzt die Aufgabe entsprechend um. Vom Abstraktionsniveau darunter ist die prozesszentrierte Programmierung, bei der die Aufgabe als Abfolge elementarer Handhabungs- und Fertigungsaufgaben beschrieben wird.

Eine weitere Programmierebene stellt das Werkzeug dar, da es als Endeffektor maßgeblich die möglichen Bearbeitungsaufgaben eines Roboters bestimmt. Daher kann die Aufgabe aus Sicht des Werkzeugs beschrieben werden, das elementare Operationen bereitstellt, die passend parametrisiert werden müssen. Die gebräuchlichste Programmiermethode ist die roboterzentrierte, wobei die Aufgabe als Sequenz von in der Regel kartesischen

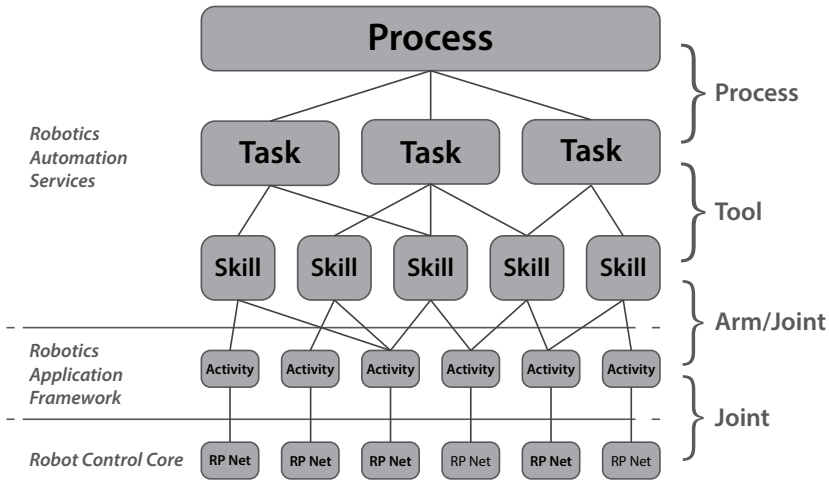


Abbildung 7.2: Funktionale Dekomposition einer Roboterzelle über die verschiedenen Abstraktionsebenen der Roboterprogrammierung hinweg. Über die *Automation Services* wird der Fertigungsprozess schrittweise aufgeteilt. Die Ausführung erfolgt durch Aktivitäten der *Robotics API*.

Roboterbewegungen beschrieben wird. Auf den Roboterbewegungen sind Schaltaktionen definiert, um das Werkzeuge an den entsprechenden Positionen zu steuern (vgl. Abschn. 3.2).

Laut Hägele et al. [108] muss sich der Applikationsentwickler eines Robotersystems in der Regel für eine dieser Abstraktionsebenen entscheiden, um seinen Roboter bzw. das Werkzeug zu programmieren. In einer serviceorientierten Roboterzelle jedoch hat jede dieser Abstraktionsebenen ihre Berechtigung und stellt für sich eine Hierarchieebene der serviceorientierten Architektur dar. Das bedeutet, dass sich in einer voll ausgebauten SOA einer Roboterzelle jede dieser Abstraktionsebenen wiederfindet. Das Problem – d. h. der Fertigungsschritt bzw. der Automatisierungsprozess – wird über die Dienste schrittweise zerlegt und durch einfachere Aufgaben einer *niedrigeren* Programmierebene beschrieben. Diese Zerlegung ist inhärenter Teil einer solchen serviceorientierten Architektur.

Die unterschiedlichen Abstraktionsebenen in der Programmierung von industriellen Roboterzellen erfordern, dass es eine unterschiedliche Granularität der Tätigkeiten innerhalb einer Roboterzelle gibt. Jede Hierarchieebene realisiert eine solche Granularität, indem eine Tätigkeit in viele einzelne Tätigkeiten feinerer Granularität zerlegt wird, die auf einer niedrigeren Abstraktionsebene angesiedelt sind. Auf feinster Granularität werden dann die in Kapitel 5 vorgestellten Activities der *Robotics API* zur Spezifikation der elementaren Roboter- und Werkzeugoperationen verwendet. Die Ausführung erfolgt auf dem Robot Control Core (RCC) als *Realtime Primitives Net* unter Echtzeitbedingungen (vgl. Abschn. 4.3).

Funktionale Dekomposition

Die funktionale Dekomposition einer Roboterzelle in Tätigkeiten unterschiedlicher Granularität ist in Abbildung 7.2 dargestellt. Dabei setzt sich jede Tätigkeit aus einer Menge von Tätigkeiten mit einer feineren Granularität und niedriger Abstraktion zusammen. Bei der Wahl der Granularität wurden mit *Skill* und *Task* zwei Begriffe verwendet, die in der Robotik gebräuchlich sind und z. B. bei Huckaby [129], Malec et al. [172], Michniewicz und Reinhart [181] oder Pfrommer et al. [216] in einem ähnlichen Kontext eingesetzt werden.

Oberhalb der *Robotics API* ist eine Fähigkeit oder ein *Skill* die feinste Granularität, um ein Tätigkeit zu beschreiben. Sie ist folgendermaßen definiert:

Definition 7.2. Eine Fähigkeit (engl.: *skill*) stellt eine Tätigkeit dar, die ein Roboter und ein Werkzeug gemeinsam ausführen können. Der Fokus liegt dabei auf dem Werkzeug, während der Roboter ausschließlich für die Bewegung bzw. den Transport von Werkzeug oder Bauteil zuständig ist.

Ein Beispiel für eine Fähigkeit ist das Greifen eines Gegenstandes mit einem bestimmten Typ von Greifer. Auch das Aufnehmen, Einsetzen und Festziehen von Schrauben mit einem elektrischen Schraubsystem sind Beispiele für Fähigkeiten. Der Fokus liegt, wie die Beispiele zeigen, auf dem Werkzeug. Der Roboter oder Manipulator spielt bei der Spezifikation der Fähigkeit keine Rolle. Er ist ausschließlich bei der internen Realisierung der Fähigkeit relevant und bewegt das Werkzeug an die notwendige Position. Dadurch wird eine Trennung von Ablauflogik und Bewegungsprogrammierung erreicht. Während in der klassischen Roboterprogrammierung die Werkzeugfähigkeiten und die Bauteileigenschaften ein Teil des Roboterprogramms werden, steht hier das Werkzeug und das Bauteil im Mittelpunkt.

In Abbildung 7.2 ist dargestellt, dass sich eine Fähigkeit bzw. ein *Skill* auf der Abstraktionsebene des Werkzeugs befindet. Intern werden zur Umsetzung einer Fähigkeit in der Regel mehrere Activities der *Robotics API* verwendet, um die Bewegungen des Roboters und die elementaren Operationen des Werkzeugs zu spezifizieren und auszuführen (vgl. Abschn. 5.5). Elementare Operationen eines Greifers sind bspw. das Öffnen und Schließen der Finger. Für ein elektrisches Schraubsystem ist es z. B. das mit einem maximalen Drehmoment begrenzte Anlaufen im Uhrzeigersinn.

Um komplexere Fähigkeiten wie das Greifen eines Gegenstandes zu realisieren, ist in der Regel innerhalb eines *Skills* der Einsatz eines Roboters notwendig. Dadurch stellt eine Fähigkeit die Transition zwischen einer werkzeug- und einer roboterzentrierten Programmierung dar (vgl. Abb. 7.2). Die Ausführung jeder Fähigkeit beruht in der Regel darauf, dass bestimmte Vorbedingungen gelten. Zum Beispiel darf ein Greifer vor dem Greifen eines Gegenstandes nicht bereits einen anderen Gegenstand halten. Gleichzeitig verändert die Ausführung einer Fähigkeit den Zustand und gegebenenfalls die Topologie der Roboterzelle, d. h. es gelten bestimmte Nachbedingungen, auf denen wiederum andere Fähigkeiten aufbauen.

Man kann die Vorbedingungen, die Fähigkeit und die Nachbedingungen als Softwarekontrakt [180] interpretieren: Wenn die Vorbedingung zutrifft, gilt nach der Ausführung der Fähigkeit die Nachbedingung. Vor- und Nachbedingungen werden auch als Zusicherungen bezeichnet. Durch das in Kapitel 5 vorgestellte objektorientierte Modell einer Roboterzelle kann der aktuelle Zustand softwaretechnisch modelliert werden. Somit kann die Vorbedingung entsprechend geprüft und anschließend die Fähigkeit ausgeführt werden. Dabei wird der softwaretechnische Zustand der Roboter aktualisiert und so den realen Gegebenheiten angepasst. Dies sollte bei einer erfolgreichen Ausführung des *Skills* der Nachbedingung entsprechen.

Aus einzelnen Fähigkeiten lassen sich Aufgaben oder *Tasks* zusammenstellen, die wie folgt definiert sind:

Definition 7.3. Eine Aufgabe (engl.: *task*) ist eine auf ein Bauteil bezogene Handhabungs- und Fertigungsfunktion. Die Aufgabe ist ausschließlich über die Betriebs- und Hilfsstoffe des Fertigungsprozesses definiert, wird aber intern über eine Sammlung von Fähigkeiten realisiert.

Beispiele für *Tasks* sind das Zuteilen von Bauteilen oder deren Montage. Hierbei wird die Aufgabe unabhängig von der Realisierung, d. h. von den beteiligten Werkzeugen oder Manipulatoren, beschrieben. Da sie Handhabungs- und Fertigungsfunktionen beschreiben, befinden sich *Tasks* dementsprechend auf der Abstraktionsebene des Fertigungsprozesses (vgl. Tab. 7.1). Wie Abbildung 7.2 zeigt, werden *Tasks* intern allerdings durch eine Menge von Fähigkeiten realisiert. Somit stellt ein *Task* die Transition zwischen einer werkzeugzentrierten und einer prozesszentrierten Programmierung dar.

Analog zu *Skills* beruht auch die Ausführung von *Tasks* darauf, dass bestimmte Vorbedingungen gelten. So muss bspw. das Bauteil, welches den weiteren Fertigungsschritten zugeteilt werden soll, in einer bestimmten Art und Weise positioniert sein (z. B. in einer Kiste). Nach der Anwendung eines *Tasks* hat sich dementsprechend der Zustand und gegebenenfalls die Topologie der Roboterzelle verändert. Hierbei spricht man analog zu den Fähigkeiten ebenfalls von der Nachbedingung, die in der Roboterzelle gilt. Im Gegensatz zu den *Skills* können die Zusicherungen ausschließlich über die Bauteile, d. h. die Betriebs- und Hilfsstoffe, formuliert werden. Bei Fähigkeiten können dagegen auch die Werkzeuge miteinbezogen werden.

Oberhalb der *Tasks* ist der Automatisierungsprozess angesiedelt, der wie folgt definiert ist:

Definition 7.4. Ein (Automatisierungs-)Prozess (engl.: *process*) beschreibt einen von der Roboterzelle angebotenen Fertigungsschritt.

Der Automatisierungsprozess stellt das Gesamtziel der Roboterzelle dar, d. h. es ist eine Beschreibung dessen, was in dieser Zelle durchgeführt werden kann. Dabei ist zu beachten, dass es nicht nur einen Prozess sondern – abhängig von der Flexibilität der Zelle – durchaus mehrere Prozess geben kann. Jeder dieser Prozesse wird – dem in Abschnitt 7.1 vorgestellten Prinzip serviceorientierter Fabriken – nach außen als Service zur Verfügung gestellt. Dort kann er über eine *Registry* aufgefunden und genutzt werden.

Ein Automatisierungsprozess transformiert ein Bauteil oder dessen einzelne Bestandteile und stellt so einen gewünschten Zielzustand oder zumindest einen gewünschten Zwischenzustand dar. Der Prozess ist eine abgeschlossene Einheit und wird vollständig über das Bauteil spezifiziert. Er entspricht damit der Abstraktionsebene des Produktes (vgl. Tab. 7.1). Da sich ein Automatisierungsprozess aus einzelnen *Tasks* zusammensetzt, stellt er die Transition zwischen einer Prozess-zentrierten und einer Produkt-zentrierten Programmierung dar.

Die vorgestellten Tätigkeiten unterschiedlicher Granularität, d. h. der *Process*, die *Tasks* und die *Skills*, werden im Rahmen dieser Arbeit jeweils als Services unterschiedlicher Abstraktion realisiert. Sie bilden die Hierarchie der serviceorientierten Architektur innerhalb der Roboterzelle. Der modulare Aufbau einer solchen Roboterzelle ist in Abbildung 7.3 dargestellt. Dabei ist die Trennung zwischen dem serviceorientierten Aufbau der Automatisierungssoftware, d. h. den *Robotics Automation Services*, und dem objektorientierten *Robotics Application Framework* eingezeichnet. Da beide Teile innerhalb der *OSGi Service Platform* umgesetzt wurden, setzen sie sich aus einer Menge von Bundles zusammen.

Das *Robotics Application Framework* kann individuell zusammengestellt werden, um die Bedürfnisse der Roboterzelle exakt abzubilden, d. h. es werden nur die notwendigen Bundles ausgesucht, um die erforderlichen Roboter und Werkzeuge verwenden zu können (vgl. Kap. 6). Zusätzlich müssen weitere Elemente, die spezifisch für die Roboterzelle sind, mit den Mitteln

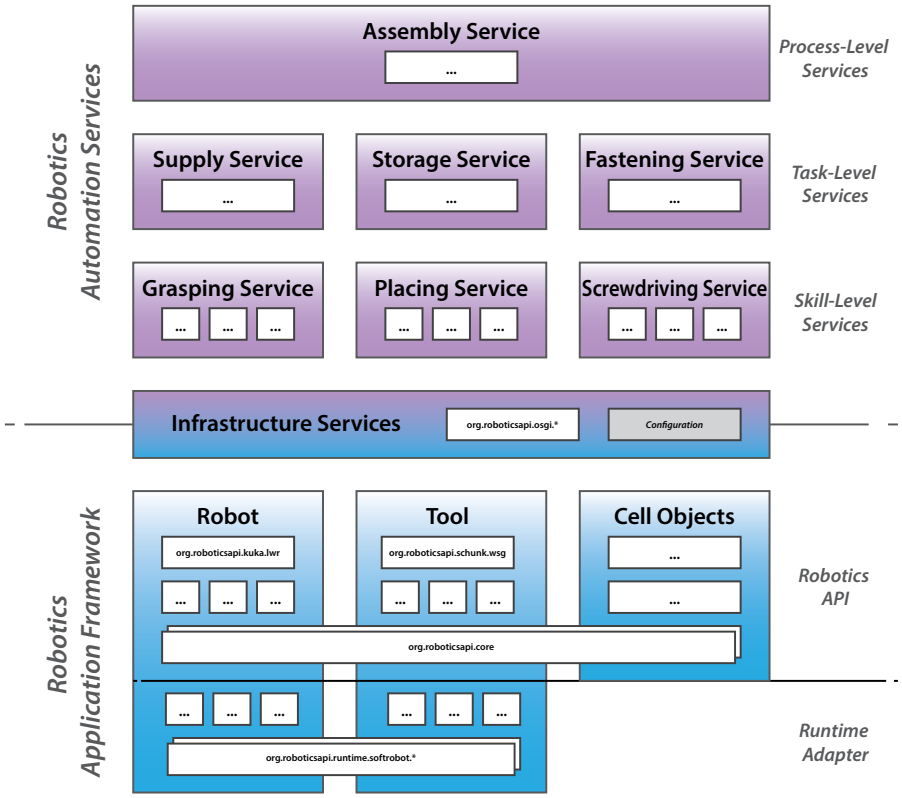


Abbildung 7.3: Eine Roboterzelle ist modular aufgebaut und besteht aus verschiedenen Bundles, die über Services miteinander agieren. Zusammen bilden sie eine hierarchisch gegliederte serviceorientierten Architektur innerhalb der Roboterzelle.

des *Robotics Application Frameworks* modelliert und implementiert werden. Diese werden dann ebenfalls als Bundles zur Verfügung gestellt. Der Aufbau der Roboterzelle kann als Konfiguration vorliegen und wird dementsprechend über Infrastrukturdienste instanziiert (vgl. Abb. 7.3) und den *Robotics Automation Services* zur Verfügung gestellt. Wie in Kapitel 6 beschrieben, können auch die notwendigen *Realtime Primitives* und *Realtime Devices* über das *Robotics Application Framework* geladen werden.

Fertigungsprozesse als Service

Die *Robotics Automation Services* sind in drei unterschiedliche Abstraktionsebenen eingeteilt und repräsentieren dabei die drei vorgestellten Arten der Granularität für Tätigkeiten innerhalb einer Roboterzelle. Dabei werden, wie in serviceorientierten Architekturen üblich, Dienste mit niedriger Abstraktion verwendet und orchestriert, um eine komplexere Funktionalität mit höherer Abstraktion zu erreichen. Demnach hat ein Dienst, der einen Automatisierungsprozess implementiert, die höchste Abstraktionsebene:

Definition 7.5. Ein prozesszentrierter Dienst (engl.: *process-level service*) orchestriert Handhabungs- und Fertigungsfunktionen anderer Dienste, um einen Fertigungsprozess oder einen Prozessschritt abzubilden. Selbst stellt er ebenfalls Handhabungs- und Fertigungsfunktionen höherer Abstraktion als Methoden bereit.

Damit wird ein Automatisierungsprozess immer durch einen prozesszentrierten Dienst abgebildet. Jedoch können auch nur einzelne Prozessschritte so implementiert werden. Dabei wird immer eine höhere Abstraktion erzielt.

Handhabungs- und Fertigungsfunktionen können auch durch aufgabenzentrierte Dienste bereitgestellt werden, die wie folgt definiert sind:

Definition 7.6. Durch einen aufgabenzentrierten Dienst (engl.: *task-level service*) werden Handhabungs- und Fertigungsfunktionen als Methoden bereitgestellt. Intern bildet er diese Funktionen auf die Fähigkeiten konkreter Werkzeuge der Roboterzelle ab.

Die Eingabeparameter der Dienste bestehen dabei nur aus den Betriebs- und Hilfsstoffen des Fertigungsprozesses, die über die objektorientierte *Robotics API* in Software abgebildet werden können. Ausführlich wird die Identifikation und Modellierung von prozess- und aufgabenzentrierten Diensten in Kapitel 9 vorgestellt.

Werkzeugfähigkeiten als Service

Während die Dienste bisher sehr abstrakt auf den Bauteilen und Werkstücken der Roboterzelle definiert sind, werden über *Skill-level Services* erstmals die Werkzeuge betrachtet. Das bedeutet konkret, dass der Service für eine bestimmte Klasse von Werkzeugen Fähigkeiten bereitstellt. Der Service entscheidet dabei intern, wie diese Fähigkeit implementiert und angewendet wird. Definiert ist ein solcher Dienst wie folgt:

Definition 7.7. Durch einen fähigkeitenzentrierten Dienst (engl.: *skill-level service*) werden abstrakte Fähigkeiten eines Endeffektors für Objekte der Roboterzelle bereitgestellt. Die aktuelle Situation und der Kontext entscheiden, wie die Fähigkeit umgesetzt wird.

Diese Fähigkeiten umfassen neben dem Werkzeug und dem Objekt in der Regel auch einen Manipulator, der für die Bewegung seines Endeffektors bzw. des Objekts zuständig ist. Dabei wird ersichtlich, dass das Objekt bzw. das Bauteil in allen drei Arten von Diensten eine zentrale Rolle spielt. Das im Bauteil gebündelte Wissen für den eigenen Fertigungsprozess ist durchgängig vorhanden und wird von oben nach unten propagiert. Da erst innerhalb eines *Skill-level Services* die Bewegungsprogrammierung der Roboter stattfindet, kann die Ablauflogik – über mehrere Abstraktionsebenen hinweg – getrennt davon betrachtet werden. Der Aufbau von *Skill-level Services* und die Fragestellung, wie anhand der aktuellen Situation und des Kontextes entschieden wird, werden detailliert in Kapitel 10 beschrieben.

Durch die Verwendung der *OSGi Service Platform* und der Aufteilung des *Robotics Application Frameworks* und der *Robotics Automation Services* auf Bundles entsteht eine modulare Automatisierungssoftware, die auf die Bedürfnisse der Roboterzelle zugeschnitten werden kann. Insbesondere Erweiterungen wie die Visualisierung oder die Kollisionserkennung (vgl. Abschn. 6.5) sind bei Bedarf verfügbar. Abgesehen von sicherheitsgerichteten Aufgaben, ist damit auch keine SPS für die Implementierung der Roboterzelle mehr notwendig. Für das *Robotics Application Framework* konnte bereits in Kapitel 6 gezeigt werden, dass ein modularer Aufbau mit einem hohen Grad an Wiederverwendung erreicht werden kann. Die Identifikation und die Modellierung der *Robotics Automation Services* wird in den nächsten beiden Kapiteln erläutert. Anhand der im nächsten Kapitel vorgestellten Fallstudie – Factory 2020 – wird die Flexibilität des serviceorientierten Ansatzes in Kapitel 11 evaluiert.

8 FALLSTUDIE: FACTORY 2020

Um die Ergebnisse des Forschungsprojekts *SoftRobot* zu demonstrieren, wurde eine robotergestützte Montageanwendung, die Factory 2020 [18], entwickelt. Dabei sollten die Besonderheiten der Softwarearchitektur hervorgehoben werden. Aus diesem Grund wurde die Anwendung so gestaltet, das mehrere Roboter zusammenarbeiten müssen, um eine komplexe Handhabungsaufgabe zu erfüllen. Dabei wird ein Bauteil aus zwei Werkstücken zusammengefügt und anschließend verschraubt (vgl. Abb. 8.1). Die Montagestation besteht aus zwei Leichtbaurobotern, die mit entsprechenden Endeffektoren ausgestattet sind. Die Anlieferung der Werkstücke und der Abtransport des fertigen Bauteils erfolgt autonom über eine mobile Plattform.

Kooperative Montage

Die einzelnen Aufgaben der Roboter wurden so gewählt, dass gezeigt werden konnte, dass die vor dem Projekt aufgestellten Ziele an die Softwarearchitektur (vgl. Abschn. 4.1) erfolgreich erreicht wurden. Die Herausforderungen der Factory 2020 waren im Einzelnen:

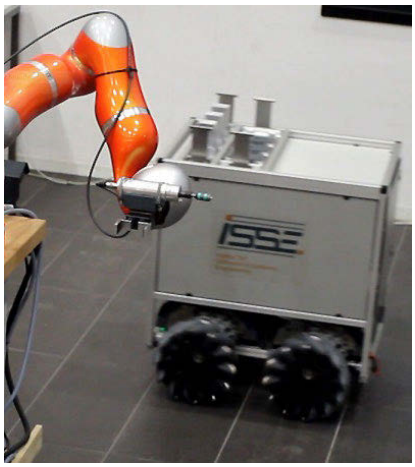
- Koordination unterschiedlicher Roboter
- Echtzeitsynchronisierte Bewegungen
- Echtzeitauswertung von Sensordaten
- Kraftgesteuerte Montagebewegungen
- Integration einer mobilen Plattform

Da zudem klassische Pick-&-Place-Aufgaben vorkommen, konnte gezeigt werden, dass sich traditionelle Anwendungen ebenso realisieren lassen.

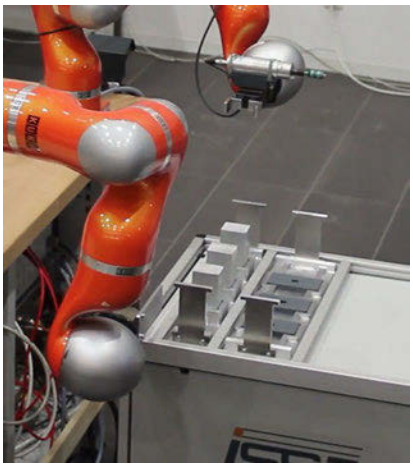
Die Factory 2020 wurde vollständig mit dem *Robotics Application Framework* und der Referenzimplementierung des RCC entwickelt. Die Automatisierungssoftware wurde als OSGi-Anwendung serviceorientiert entwickelt. Die einzelnen Aufgaben der Roboter wurden in Form von Diensten gekapselt. Das Ziel war es, die serviceorientierte Architektur der Software an dem Automatisierungsprozess auszurichten und die Prozessschritte mithilfe von Diensten zu orchestrieren. Dieses Vorgehen und die entstandene Anwendung wird in den nächsten Kapiteln detailliert erläutert.

(a) Unterteil (b) Deckel (c) Fertiges Bauteil

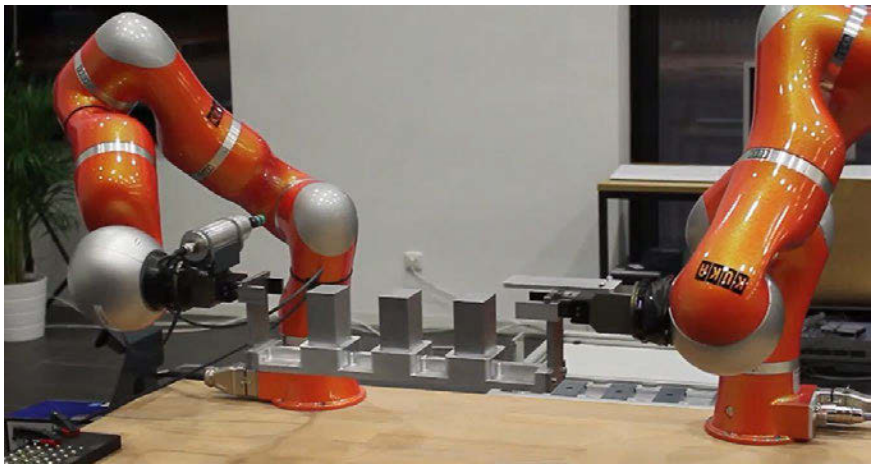
Abbildung 8.1: Aus dem Unterteil und dem Deckel wird das fertige Bauteil zusammengesetzt (jeweils im entsprechenden Werkstückträger).



(a) Anfahrt der Plattform



(b) Position der Plattform bestimmen

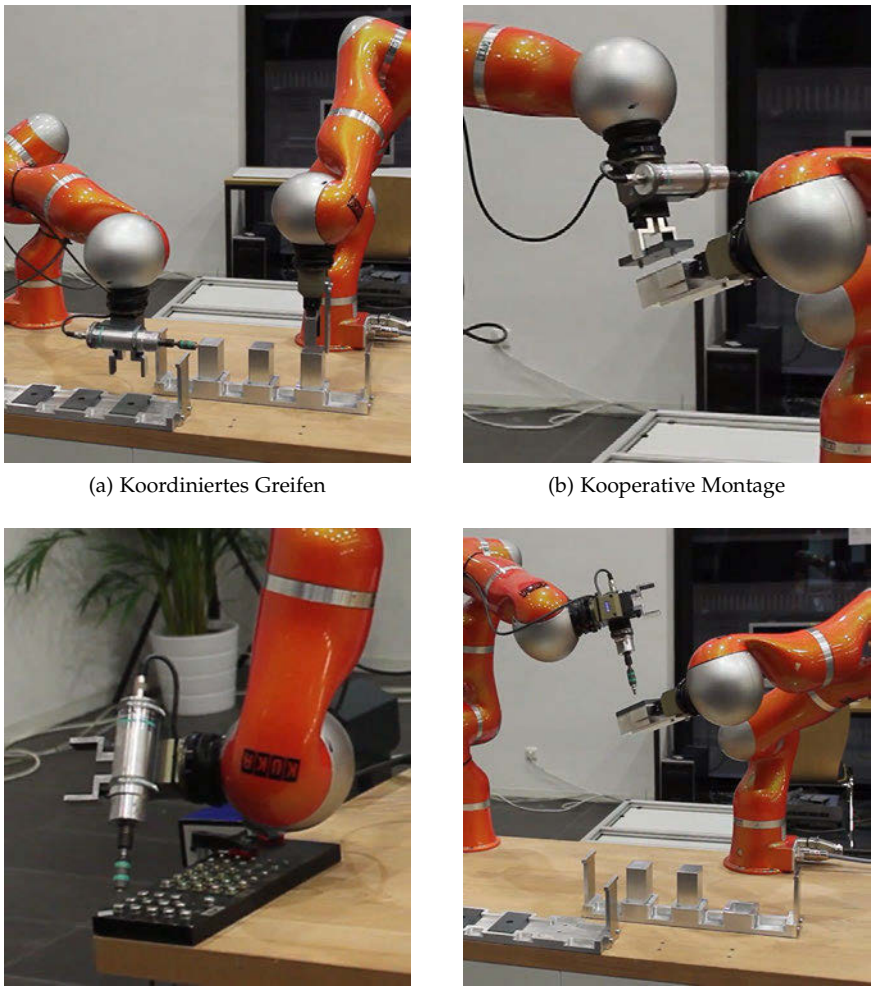


(c) Synchronisierter Transport der Werkstückträger

Abbildung 8.2: Die Werkstücke werden durch eine mobile Plattform an die Montagestation geliefert. Beide Roboter heben die Werkstückträger synchronisiert auf die Montagestation.

Die Abbildungen 8.2 und 8.3 geben einen Überblick über die einzelnen Aufgaben der Factory 2020. Die beiden Bestandteile des zu fertigenden Bauteils, d. h. das Unterteil (vgl. Abb. 8.1 (a)) und der Deckel (vgl. Abb. 8.1 (b)), werden in Werkstückträgern an eine Montagestation gebracht. Die Anlieferung der Werkstückträger erfolgt autonom durch eine mobile Plattform (vgl. Abb. 8.2 (a)). Nachdem die mobile Plattform die Montagestation erreicht hat, erfolgt durch die beiden Leichtbauroboter ein Transport der Werkstückträger auf die Arbeitsfläche der Montagestation.

Da die Position der mobilen Plattform und damit der Werkstückträger mit einer großen Unsicherheit behaftet ist, wird zuerst die Position der mobilen Plattform bzw. des ersten Werkstückträgers ermittelt (vgl. Abb. 8.2 (b)). Dazu positioniert einer der beiden Leichtbauroboter seinen Endeffektor in der ungefähren Nähe des Werkstückträgers. Anschließend fährt er kraftgesteuert in dessen Richtung, bis ein Kontakt festgestellt wird. Dies wiederholt der Roboter aus einer anderen Richtung, um die Position des Werkstückträgers zu berechnen. Um die Ausrichtung des Werkstückträgers zu bestimmen, positioniert der andere Leichtbauroboter seinen Endeffektor ebenso in der



(a) Koordiniertes Greifen

(b) Kooperative Montage

(c) Aufnehmen einer Schraube

(d) Einsetzen einer Schraube

Abbildung 8.3: Die Montage der Werkstücke erfolgt gemeinsam. Während beim Greifen die Roboter nur koordiniert sind, erfolgt die Montage und das Verschrauben kooperativ.

ungefähren Nähe des Werkstückträgers und fährt danach kraftgesteuert in dessen Richtung, bis ein Kontakt festgestellt wird.

Sobald die Position und Ausrichtung des ersten Werkstückträgers bestimmt wurde, greift jeder Leichtbauroboter eine Seite des länglichen Werkstückträgers. Anschließend transportieren sie den gegriffenen Werkstückträger kooperativ mit einer synchronisierten Bewegung auf die Arbeitsfläche und stellen ihn an einer definierten Position ab (vgl. Abb. 8.2 (c)). Die Bewegung muss in der Echtzeitrobotersteuerung exakt synchronisiert sein, um eine Beschädigung der Roboter, Greifer oder Werkstückträger auszuschließen. Dieser Vorgang wird für den zweiten Werkstückträger wiederholt. Allerdings wird auf die Bestimmung der exakten Position verzichtet, da diese sich aus der Position des ersten Werkstückträgers bestimmen lässt.

Nachdem die beiden Werkstückträger auf der Arbeitsfläche abgestellt wurden, beginnt der eigentliche Montageprozess, dessen Schritte in Abbildung 8.3 dargestellt sind. Dazu nimmt jeder Leichtbauroboter eines der beiden Einzelteile mit seinem Greifer auf (vgl. Abb. 8.3 (a)). Die Einzelteile sind geeignet in den Werkstückträgern angeordnet, damit sie für die weite-

re Montage von den Robotern gegriffen werden können. Das Greifen erfolgt koordiniert, sodass es zu keiner Kollision der beiden Roboterarme kommt. Anschließend transportieren die Roboter das jeweils gegriffene Einzelteil zu einer definierten Position, von der aus die Montage gestartet wird.

Dazu hält der rechte Roboter das Unterteil sehr steif, während der linke Roboter mit Nachgiebigkeit und einer definierten Kraft den Deckel aufsetzt (vgl. Abb. 8.3 (b)). Durch die Nachgiebigkeit und die Kraftanwendung können leichte Abweichungen in der Passgenauigkeit der beiden Teile kompensiert werden. Sobald der fügende Roboter festgestellt hat, dass der Deckel sicher auf dem Unterteil platziert wurde, lässt er den Deckel mit seinem Greifer los und fädelt sicher aus dem Bauteil aus. Danach beginnt er mit dem Verschrauben des Bauteils, um den Deckel an das Unterteil zu fixieren.

Um die beiden Teile verschrauben zu können, ist am linken Roboter neben einem Greifer auch ein elektrisches Schraubsystem mit Stromsteuerung montiert. Das Steuergerät des Schraubsystems ist mit der Robotersteuerung gekoppelt und kann bei Erreichen eines eingestellten Ziel-Drehmomentes den Motor abschalten. Mit diesem Schraubsystem als primären Endeffektor, fährt der Roboter als nächsten Schritt ein Magazin mit Schrauben an (vgl. Abb. 8.3 (c)). Über eine kraftgesteuerte Bewegung, setzt der Roboter das Schraubsystem in die Schraube ein und löst diese aus dem Magazin [10]. Durch einen magnetischen Bithalter wird die Schraube sicher gehalten und kann zu dem Bauteil transportiert werden. Dort wird die Schraube ebenfalls kraftgesteuert in das Gewinde eingesetzt und über das Schraubsystem mit einem definierten Drehmoment festgezogen (vgl. Abb. 8.3 (d)). Auch hier werden durch die Nachgiebigkeit und die Kraftanwendung leichte Abweichungen in der Passgenauigkeit kompensiert. Dieser Vorgang wird für alle Schrauben wiederholt. Danach wird das fertige Bauteil vom linken Roboter gegriffen und in den Werkstückträger, der ursprünglich die Deckel enthielt, platziert. Nachdem alle Bauteile gefertigt sind, werden die beiden Werkstückträger und damit die Bauteile zurück auf die mobile Plattform gestellt. Von dort aus werden sie für eine weitere Verarbeitung abtransportiert.

Variationsmöglichkeiten

Für die Realisierung der Factory 2020 wurden ursprünglich zwei KUKA Leichtbauroboter (LBR) verwendet. Diese waren jeweils mit einem Parallelgreifer des Typs MEG-50 der Fa. Schunk ausgestattet. Zusätzlich war einer der beiden Roboter mit dem Schraubsystem PLUTO der Fa. Kolver ausgestattet. Das Steuergerät kann über digitale I/Os mit dem RCC kommunizieren und berechnet das Drehmoment anhand der Spannung, der Frequenz und des Stroms. Der oben beschriebene Fertigungsprozess sowie die Ausstattung der Montagestation wurden anschließend variiert, um die Flexibilität des serviceorientierten Ansatzes zu evaluieren.

Für die Factory 2020 wurden folgende Variationspunkte definiert und zu jedem dieser Punkte wurden mögliche Variationen identifiziert:

- Variation des Fertigungsprozesses
- Variation der Topologie der Roboterzelle
- Variation des Werkstückes
- Variation der eingesetzten Roboter und Greifer

In den nächsten Kapitel der Arbeit wird eine serviceorientierte Modellierung der Factory 2020 vorgestellt. Anschließend wird in Kapitel 11 aufgezeigt, wie sich die möglichen Variationen umsetzen lassen. Die Factory 2020 wurde zudem vollständig in der Visualisierung abgebildet (vgl. Abschn. 6.5.1), um Variationen auch simuliert testen zu können.

9 SERVICEORIENTIERTE MODELLIERUNG

Im vorherigen Kapitel wurde gezeigt, dass der Aufbau einer Roboterzelle in einzelne Komponenten unterteilt werden sollte, um die Roboterzelle modular, wiederverwendbar und flexibel zu gestalten. Daher wird in diesem Kapitel ein neuer Ansatz vorgestellt, wie eine solche Unterteilung vorgenommen werden kann. Diese Unterteilung ist die Grundlage für die serviceorientierte Modellierung der Roboterzelle, da jede Komponente die von ihr zu erbringenden Aufgaben als Dienstleistung über einen Service bereitstellt.

Zur Modellierung der Roboterzelle wird in diesem Kapitel die Systems Modeling Language (SysML) [196] verwendet. SysML ist eine auf der Unified Modeling Language (UML) [197] basierende, standardisierte Modellierungssprache, die ihre Anwendung im Bereich des *Systems Engineering* für die Modellierung komplexer Systeme hat. Die Menge der in SysML vorhandenen Diagramme besteht aus einer Untermenge von Diagrammen aus der UML und wird ergänzt durch spezifische Diagramme. SysML erweitert die UML vor allem um für das Systems Engineering wichtige Sichten. Dadurch kann ein System als hierarchische Komposition einzelner Teile oder Subsysteme modelliert werden. Dies ist in UML durch den Fokus auf die objektorientierte Softwareentwicklung nicht möglich [149]. Ein Basisstrukturelement wird in SysML als *Block* bezeichnet und kann für eine mechanische Konstruktion, Hardware, Software, Verfahren oder Anlagen stehen [149]. Ein Block entspricht einer Klasse in UML, besitzt aber weder Methoden noch Attribute. Durch ein Blockdefinitionsdiagramm (BDD) [95] kann ein Teil-Ganzes-Sicht auf einen Block und damit die Bestandteile des Systems, d. h. die Roboterzelle, dargestellt werden.

Das Interne Blockdiagramm (IBD) dient der Darstellung des inneren Aufbaus eines Blocks. Dadurch wird die interne Struktur eines Blocks sichtbar und es ergibt sich die Möglichkeit, die hierarchische Struktur des System zu modellieren. Die Verschaltung von Blöcken in einem IBD erfolgt durch Konnektoren und Ports [149]. Dabei können wie in der UML nicht nur Nachrichten sondern auch kontinuierliche Flüsse (z. B. Wasser und Energie) modelliert werden. Ebenfalls kann mit der SysML beschrieben werden, welche Elemente zwischen zwei Ports bzw. über die Konnektoren zweier Systemkomponenten fließen. Durch Elementflüsse wird der generelle Austausch von Elementen zwischen Blöcken beschrieben. Der tatsächliche Zeitpunkt für den Austausch spielt hierbei keine Rolle. Um letzteres zu modellieren, steht das aus UML bekannte Aktivitätsdiagramm in SysML in erweiterter Form zur Verfügung. Durch die tokenbasierte Semantik können Kontroll- und Datenfluss kombiniert dargestellt werden. Aktivitätsdiagramme werden verwendet, um die zeitliche Reihenfolge von Aktionen zu beschreiben und dabei gleichzeitig den Austausch von Elementen darzustellen.

Die vorliegende Arbeit verwendet die SysML, um eine umfassende Modellierung von Roboterzellen in logische und funktional zusammengehörende Einheiten einzuführen. Damit wird erstmals der Weg zu einer systematischen Modellierung von Roboterzellen bereitet. In Abschnitt 9.1 wird dazu erläutert, wie Roboterzellen in einzelne logische Einheiten und Services zu strukturieren sind. Auf Grundlage dieser Services lassen sich Produktionsabläufe als Sequenz von Serviceaufrufen realisieren (vgl. Abschn. 9.2).

Die Implementierung der definierten Services und Produktionsabläufe unter Zuhilfenahme der in der Zelle verfügbaren Aktuatoren und Sensoren wird in Abschnitt 9.3 beschrieben. Anschließend werden in Abschnitt 9.4 Vorgehensweisen vorgeschlagen, um serviceorientierte Roboterzellen zu implementieren. Zum Abschluss werden verwandte Arbeiten in Abschnitt 9.5 beschrieben. Einige der in diesem Kapitel beschriebenen Konzepte und Ideen wurden erstmals in [121] und [254] vorgestellt.

9.1 LOGISCHE MODELLIERUNG VON ROBOTERZELLEN

Eine direkte und einfache Unterteilung einer Roboterzelle würde anhand der vorhandenen Aktuatoren bzw. der Manipulatoren der Zellen und ihrer Endeffektoren erfolgen. Sie sind die zentralen Elemente einer Roboterzelle. Eine serviceorientierte Modellierung würde in diesem Fall jedem Manipulator aufgrund seiner durch die Endeffektoren vorhandenen Fähigkeiten Services zuweisen. Mithilfe dieser Services können die Aufgaben der Roboterzelle abgebildet und realisiert werden. Der Nachteil dieses Ansatzes ist die sehr starke Kopplung zu den vorhandenen Geräten. Daher bedingt eine strukturelle Änderung an den vorhandenen Geräten der Roboterzelle eine vollständige Überprüfung der gesamten Servicestruktur, Prozesse und Dienstimplementierungen.

Roboter als polyfunktionale Maschinen

Roboter müssen vielmehr als *polyfunktionale Maschinen* angesehen werden, die frei programmierbar sind. Als solche können sie – abhängig von ihren Endeffektoren – eine Vielzahl von Aufgaben in der Zelle wahrnehmen. Abhängig vom Fertigungsprozess bzw. dessen aktuellem Stand kann der Roboter als Manipulator und sein Endeffektor als Werkzeug in unterschiedliche Rollen „schlüpfen“. Daher sollte keine direkte Abbildung zwischen den verfügbaren Robotern und den notwendigen Services erfolgen. Zudem können in einer kooperativen Roboterzelle die Aufgaben nicht auf nur einen Roboter und dessen Endeffektoren verteilt werden.

Daher wird in dieser Arbeit ein neuer Ansatz verfolgt. Die Roboterzelle wird aufgrund der stattfindenden Prozesse in logische Einheiten unterteilt. Diese Einheiten haben jeweils eine klar definierte Aufgabe im System der Roboterzelle und stellen einen Service bereit, um diese Aufgaben abzurufen. Abgebildet werden die logischen Einheiten auf die in der Zelle vorhandenen Geräte. Im Vergleich zu dem obigen Ansatz besteht eine geringere Kopplung, da die darüber liegende Ebene keine Kenntnis von der physischen Struktur der Zelle hat und braucht. Demzufolge muss bei einer Änderung der physikalischen Struktur (z. B. weil ein Aktuator ersetzt wird) die Implementierung der logischen Einheiten angepasst und sichergestellt werden, dass die Services weiterhin funktionsfähig sind. Die Struktur der Services und die bereitgestellten Aufgaben in der Zelle bleiben jedoch identisch. Zudem können für neue Aufgaben neue logische Einheiten gebildet werden. Der Ansatz trägt dazu bei, eine Roboterzelle flexibel und erweiterbar zu gestalten (vgl. Kap. 11).

Modellierung der Handhabungsobjekte

Der Startpunkt einer logischen Modellierung und Unterteilung einer Roboterzelle liegt bei den dort notwendigen Bauteilen, Baugruppen, Werkstückträgern und Zuführeinrichtungen. Alle in einer Roboterzelle durchgeführten Aufgaben beschränken sich in der Regel auf das Handhaben, Bearbeiten und Prüfen dieser Objekte. Dabei ist Handhaben definiert als „das Schaffen, definierte Verändern oder vorübergehende Aufrechterhalten einer vorgegebenen räumlichen Anordnung von geometrisch bestimmten Körpern in einem Bezugssystem. Es können [dabei] weitere Bedingungen

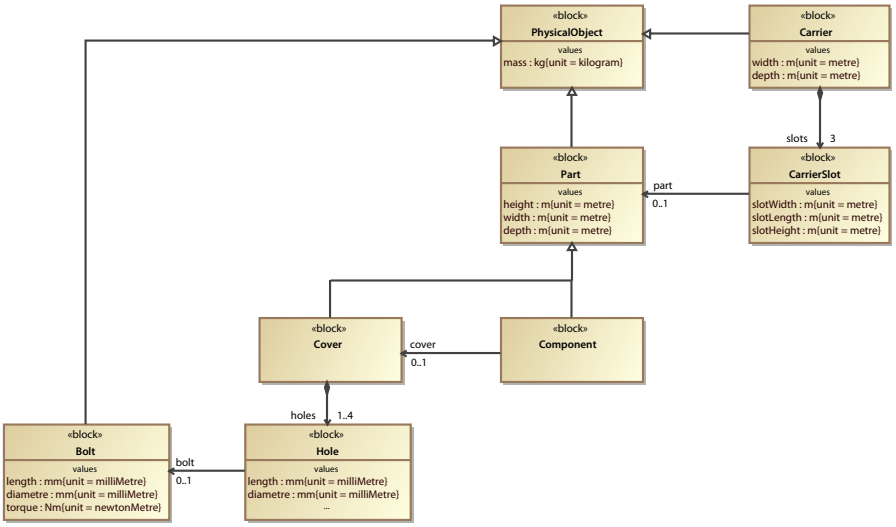


Abbildung 9.1: Die Handhabungsobjekte einer Roboterzelle werden mit SysML modelliert. Das abgebildete Blockdefinitionsdiagramm zeigt die wichtigsten Handhabungsobjekte der Factory 2020.

– wie z. B. Zeit, Menge oder Bewegungsbahn – vorgegeben sein.“ [269, S. 2]. Fertigungsverfahren sind gemäß DIN 8580 [68] Formgeben, Formändern, Behandeln und Fügen. Unter Fügen [69] versteht man das „dauerhafte Verbinden mehrerer Körper oder von Körpern mit formlosen Stoff“ [269, S. 8].

Demnach bestehen die Aufgaben einer Roboterzelle aus einer Folge von mehreren Fertigungs-, Handhabungs- und ggf. Prüfschritten. Die in der Zelle vorhandenen Objekte werden im weiteren Verlauf als Handhabungsobjekte bezeichnet. Sie sind elementarer Teil jeder Handhabungs- und Fertigungsaufgabe der Zelle. Das bedeutet, dass jeder Prozess und jede Aufgabe einer Roboterzelle auf Basis der Handhabungsobjekte definiert ist und mindestens den Zustand eines Handhabungsobjektes verändert. Um die zu realisierenden Aufgaben zu verstehen und analysieren zu können, müssen daher diese Objekte zuerst modelliert werden.

Abbildung 9.1 zeigt die Handhabungsobjekte der Factory 2020 in einem Blockdefinitionsdiagramm. Die beiden zentralen Elemente sind das Basisbauteil, repräsentiert durch den Block Component, und der zu montierende Deckel, repräsentiert durch den Block Cover. Der Deckel verfügt über eine Menge an Bohrungen (vgl. Hole) mit dem er über Schrauben (vgl. Bolt) am Bauteil fixiert wird. Sowohl das Bauteil als auch der Deckel sind Spezialisierungen der Blöcke Part bzw. PhysicalObject und erben damit wie in der UML deren Eigenschaften wie z. B. Masse öder Größe. Die Anlieferung der beiden Werkstücke erfolgt über einen Werkstückträger, der in Abbildung 9.1 durch den Block Carrier repräsentiert wird. Jeder Werkstückträger verfügt über drei Abteile (vgl. CarrierSlot), von denen jedes ein Bauteil aufnehmen kann. Abhängig von der Ausprägung des Werkstückträgers kann entweder ein Unterteil (ohne Deckel) oder ein Deckel bzw. ein fertiges Bauteil aufgenommen werden. Der Block Carrier erbt als Spezialisierungen eines PhysicalObjects ebenfalls dessen Eigenschaften.

Funktionsfolgen und Symbole

Nachdem die Handhabungsobjekte modelliert wurden, können die einzelnen Aufgaben bzw. Funktionen der Roboterzelle modelliert und analysiert werden. Für die Planung ist eine eindeutige Formulierung und Darstellung dessen, in welcher Abfolge was passieren soll von großer Bedeutung [116]. Der VDI empfiehlt in der Richtlinie 2860 [269] bei der Lösung

SYMBOL	FUNKTION	BESCHREIBUNG
	Geordnetes Speichern (*arranged storage*)	Aufbewahren geometrisch bestimmter Körper, wobei Orientierung und Position in allen Freiheitsgrads definiert sind.
	Abteilen (*separate*)	Bilden von Teilmengen definierter Größe oder Anzahl aus einer Menge. Vereinzeln ist eine Sonderform mit Zielmenge 1.
	Zuteilen (*allocate*)	Bilden von Teilmengen definierter Größe oder Anzahl und das Bewegen dieser Teilmengen zu definierten Zielorten.
	Positionieren (*position*)	Bewegen eines Körpers aus einer unbestimmten in eine vorgegebene Position.
	Orientieren (*orientate*)	Bewegen eines Körpers aus einer unbestimmten in eine vorgegebene Orientierung.
	Ordnen (*arrange*)	Bewegen eines Körpers aus einer unbestimmten in eine vorgegebene Orientierung und Position bzw. Bewegungsrichtung.
	Weitergeben (*transfer*)	Bewegen von Körpern aus einer vorgegebenen in eine andere Position entlang einer nicht definierten Bahn.
	Fügen (*join*)	Dauerhaftes Verbinden von zwei oder mehr Körpern geometrisch bestimmter Form; Fertigungsverfahren (vgl. DIN 8593 [69])
	Spannen (*clamp*)	Vorübergehendes Sichern eines Körpers in einer bestimmten Orientierung und Position unter Beteiligung von Kraftschluss
	Entspannen (*release*)	Umkehrung des Spannens
	Prüfen (*test*)	Feststellen, ob Körper vorgegebene Bedingungen (z. B. Position oder Orientierung) erfüllen

Tabelle 9.1: Wichtige Handhabungs- und Fertigungsaufgaben in symbolischer Darstellung gemäß VDI-Richtlinie 2860 [269].

von Handhabungsaufgaben als erstes eine „klare und eindeutige Formulierung der Aufgabenstellung durch lösungsneutrale Funktionen“ [269, S. 1]. In der Montage- und Handhabungstechnik gilt das „Denken in Funktionen“ [116, S. 47] als erster Schritt einer analytischen Betrachtung und ist die Voraussetzung zur Lösung einer Handhabungsaufgabe.

Dementsprechend gibt die Richtlinie eine eindeutige Einordnung und Abgrenzung des Handhabens und definiert Symbole für die einzelnen Teilfunktionen des Handhabens. Durch die symbolische Darstellung lässt sich eine „leicht überschaubare Aufgabenbeschreibung als Symbolfolge“ [269, S. 1] erstellen. Die Richtlinie gibt dabei keinen Detailgrad vor, sondern ermöglicht explizit die Modellierung in unterschiedlichen Detailschärfen. Dadurch kann die Aufgabenstellung entweder in einem symbolischen Funktionsplan beschrieben werden oder in einer sehr genauen Ausarbeitung der einzelnen Funktionen und ihrer Zuordnung zu einzelnen Geräten.

Das Handhaben untergliedert sich gemäß Richtlinie [269] in folgende fünf Teilfunktionen, die sich immer auf geometrisch bestimmte Körper beziehen:

1. Speichern (Aufbewahren bzw. Halten von Mengen)
2. Mengen verändern (Teilen bzw. Vereinigen von Mengen)
3. Bewegung (Schaffen bzw. Verändern einer räumlichen Anordnung)
4. Sichern (Aufrechterhalten einer räumlichen Anordnung)
5. Kontrollieren (Prüfen bestimmter Eigenschaften oder Zustände)

Ein Ausschnitt der Teilfunktionen und ihrer Symbole ist in Tabelle 9.1 dargestellt. Dabei findet sich mit Fügen auch ein Fertigungsschritt darunter. Ob-

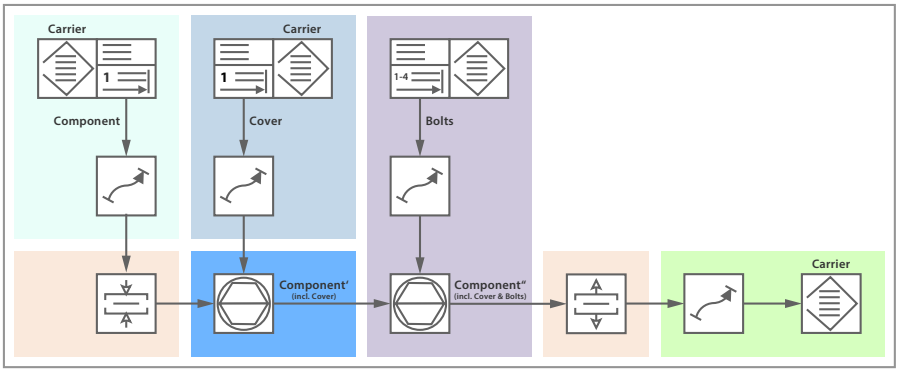


Abbildung 9.2: Der Prozess *Montage und Verschrauben* der Factory 2020 als symbolischer Funktionsplan gemäß VDI-Richtlinie 2860 [269].

wohl sich die Richtlinie nur mit Handhaben befasst, werden verschiedene Symbole für Fertigungsschritte (d. h. Formgeben, Formändern, Behandeln und Fügen) vorgeschlagen, um eine ganzheitliche Darstellung zu erlauben.

Modellierung der Handhabungsfunktionen

Auf Basis der in der VDI-Richtlinie 2860 definierten Teilfunktionen wurden die Handhabung- und Fertigungsschritte der Factory 2020 symbolisch als Funktionsplan beschrieben. Dazu wurden auf Basis der oben beschriebenen Handhabungsobjekte alle notwendigen Funktionen modelliert. Es wurde ein symbolischer Detaillierungsgrad gewählt, da es sich in diesem Schritt um eine Darstellung lösungsneutraler Funktionen handelt. Diese Funktionen sollen Grundlage der notwendigen Dienste der Zelle sein und später auf Roboter- bzw. Werkzeugaktionen abgebildet werden.

Der symbolische Funktionsplan für die Montage und das Verschrauben der beiden Werkstücke der Factory 2020 ist in Abbildung 9.2 dargestellt. Alle dafür verwendeten Symbole sind in Tabelle 9.1 zu finden[1]. Die beiden Werkstückträger (Carrier) sind als *geordnete Speicher* symbolisiert. Um ein Unterteil (Component) bzw. einen Deckel (Cover) für den Prozess verfügbar zu machen, ist es notwendig, diese mit der Funktion *Zuteilen* aus dem Werkstückträger zu vereinzeln und an einen definierten Ort zu bewegen. Danach finden jeweils ein *Weitergeben* an den Fertigungsschritt statt. Dort wird das Unterteil für die weitere Bearbeitung durch die Funktion *Spannen* vorübergehend gesichert. Bei der Montage der beide Werkstücke handelt es sich um *Fügen* durch Zusammensetzen [70]. Als Ergebnis ist das Basisbauteil verändert (Component') und verfügt über einen Deckel.

Bei dem anschließenden Verschrauben handelt es sich ebenfalls um *Fügen* als Fertigungsverfahren und verändert das Bauteil (Component''). Im Gegensatz zum Zusammensetzen werden die beiden Werkstück durch Anpressen [71] dauerhaft verbunden. Für die Verbindung sind jedoch Schrauben notwendig, die ebenfalls durch mehrere Handhabungsfunktionen dem Fertigungsschritt zugeteilt werden. Da die Schrauben in der Factory 2020 geordnet vorliegen, wird das entsprechende Depot ebenfalls als *geordneter Speicher* symbolisiert. Nach dem Schraubvorgang wird das Bauteil durch die Funktion *Entspannen* gelöst und durch *Weitergeben* wieder zum Werkstückträger transportiert, der wie zu Beginn als *geordneter Speicher* symbolisiert ist. Dieses Symbol repräsentiert hierbei auch die Funktion des geordneten Speicherns, d. h. das fertige Bauteil wird in den Werkstückträger gelegt. Nachdem *Zuteilen* bzw. dem *Fügen* ist jeweils das Handhabungsobjekt bzw. dessen Zustand (zusammengesetzt, verschraubt) dargestellt.

1 Die verwendeten Funktionen der VDI-Richtline 2860 werden im Weiteren kursiv geschrieben.

Der Funktionsplan stellt die Grundlage dar, um die Roboterzelle in logische Einheiten mit klar definierten Services zu unterteilen. Dabei wurden folgende zwei Möglichkeiten betrachtet:

- ein Service pro Werkstücktyp
- ein Service pro Funktionsträger

Bei der ersten der beiden Möglichkeiten wären die Zuständigkeiten des Services exakt definiert. Während sich bspw. ein Dienst vollständig um die Handhabung des Covers kümmert, ist ein anderer für das Basisbauteil zuständig. Zudem würde ein solcher Service die möglichen Zustände seines Werkstückes kennen und könnte ihn überwachen. Jedoch würde im Fall des Basisbauteils ein sehr umfangreicher und wenig modularer Service entstehen. Dementsprechend wäre auch die Wiederverwendung von Diensten nicht oder nur in geringem Umfang gegeben.

Modellierung als logische Einheiten

Daher wurde die zweite Möglichkeit der Modellierung als die vielversprechendere erachtet und weiterverfolgt. Das bedeutet, dass die Teilfunktionen nach einer möglichst starken prozeduralen Kohäsion zusammengefasst werden. Diese liegt vor, wenn die Teilfunktionen nach ihrer Ausführungsreihenfolge gruppiert sind. Zudem spielen semantische Zusammenhänge zwischen den Handhabungsfunktionen eine Rolle. Die Zusammenfassung einzelner Teilfunktionen zu einer logischen Einheit und damit zu einem Service ist eine Abwägung zwischen einem hohen Maß an Modularität und einer sinnvollen und überschaubaren Menge an Diensten in der Zelle. Für die logische Gruppierung haben sich folgende Vorgehensweisen empfohlen:

- Zusammenhängende Handhabungsfunktionen eines Werkstückes sollten als Einheit gruppiert werden. Es hat sich gezeigt, dass diese Funktionen eine hohe Kohäsion aufweisen. Insbesondere hängen die Vor- und Nachbedingungen dieser Funktionen oft zusammen. Falls diese Funktionen in einer logischen Einheit zusammengefasst werden, weist die logische Einheit ebenfalls eine hohe Kohäsion auf, was wünschenswert ist. In Abbildung 9.2 folgt bspw. einem *Zuteilen* der beiden Werkstücke immer ein *Weitergeben*. Der Vorgehensweisen folgend werden daher beide Funktionen in einer logischen Einheit zusammengefasst. Aufgrund der hohen Kohäsion der beiden Funktionen hat dies keine negativen Auswirkungen auf die Modularität der Services.
- Während ein Handhabungsschritt in der Regel nur die Position und Lage eines Objekts ändert, transformiert ein Fertigungsschritt ein Objekt nachhaltig. Es gibt einem Objekt die Form, ändert diese oder bearbeitet dessen Oberfläche. Beim Fügen werden sogar zwei Objekte (dauerhaft) zusammengesetzt. Daher sollte ein Fertigungsschritt als eine Einheit gruppiert (vgl. *Fügen* in Abb. 9.2) und nicht mit den Handhabungsfunktionen kombiniert werden. Treten mehrere ähnliche Fertigungsschritte ohne zwischenzeitliches *Bewegen* auf, können diese zusammengefasst werden.
- Eine Ausnahme bilden Funktionen, die zur Zuführung von Hilfsstoffen (z. B. Schrauben) zu einem Fertigungsschritt notwendig sind. Diese sollten mit dem Fertigungsschritt in einer Einheit gruppiert werden, da sie in der Regel nicht getrennt von dem Fertigungsschritt auftreten können. In Abbildung 9.2 sind die Schrauben ein Hilfsstoff der Fertigungsfunktionen *Fügen (durch Schrauben)*. Die Schrauben sind essentieller Bestandteil dieser Funktion, während das zu schraubende

Objekt ausgetauscht werden kann. Auf dieser Ebene werden die Funktionen immer zusammen verwendet. Daher hat dies keine negativen Auswirkungen auf die Modularität der Services.

- Wiederkehrende Funktionen oder Funktionsfolgen, besonders wenn sich die betroffenen Bauteile oder Speicher ähnlich sind, sollten identisch gruppiert werden. Dadurch steigt die Möglichkeit, dass diese Funktionen oder Funktionsfolgen durch dieselbe logischen Einheit implementiert werden können, was zu einer höheren Wiederverwendung führt. In Abbildung 9.2 wird zweimal aus einem Werkstückträger ein Bauteil *zugeteilt* und anschließend *weitergegeben*. Diese Funktionen weisen eine gewisse Ähnlichkeit auf, die in der gleichen oder einer ähnlichen Implementierung resultieren kann.

- Bei gleichen Funktionsfolgen – insbesondere in der Handhabung – kommt es vor, dass nur bei einem Bauteil vor- oder nachgestellte Funktionen auftreten. In Abb. 9.2 ist dies bspw. beim Basisbauteil gegeben. Nach dem *Weitergeben* wird das Werkstück durch die Funktion *Spannen* gesichert. Der Deckel dagegen muss nicht gesichert werden. Um das *Zuteilen* und *Weitergeben* beider Bauteile identisch modellieren und implementieren zu können, sollten solche vor- oder nachgestellten Funktion in eine eigene logische Einheit mit einer eigenen Service-Schnittstelle verschoben werden.

Unter Zuhilfenahme der Vorgehensweisen können logische Gruppierungen für die Handhabung- und Fertigungsschritte einer Roboterzelle identifiziert werden. Dementsprechend gruppiert sich die Factory 2020 gemäß der farblichen Markierungen in Abbildung 9.2 in sechs logische Einheiten für die zusammengehörigen Teilfunktionen der Montage und des Verschraubens.

Diese sechs Einheiten bilden zum Einen die Grundlage für eine logische Strukturierung der Roboterzelle und zum Anderen die Definition der einzelnen Services. Analog zu den Handhabungsobjekten der Roboterzelle kann auch diese Struktur mit SysML modelliert werden. Durch ein Blockdefinitionsdiagramm wird folglich der hierarchische Aufbau der Roboterzelle dargestellt, d. h. die oben identifizierten logischen Einheiten werden zu logischen Subsystemen in hierarchisch gegliederten Roboterzellen. Dabei werden nur die einzelnen Subsystem und deren Multiplizität in dem gesamten System der Roboterzelle modelliert.

Zusätzlich werden für jedes logische Subsystem Ports definiert, die die externen Schnittstellen des Systems darstellen. Im Gegensatz zu UML ermöglicht es SysML mit Port mechanische, elektrische und softwaretechnische Schnittstellen zu modellieren [95, S. 148]. Daher wird durch Ports dargestellt, welche Handhabungsobjekte der Zelle in ein Subsystem eingegeben bzw. ausgegeben werden. Innerhalb des Subsystems werden auf diesen Objekten Handhabungs- oder Fertigungsschritte angewandt. Allerdings wird in dem Blockdefinitionsdiagramm durch Ports nicht modelliert, wie die einzelnen Subsysteme diese Objekte miteinander austauschen.

Logischer Aufbau einer Roboterzelle

Der hierarchische Aufbau der Montagezelle der Factory 2020 ist in Abbildung 9.3 als Blockdefinitionsdiagramm dargestellt. Der zentrale Systembaustein der Montagezelle wird als AssemblySystem bezeichnet und bietet ihre möglichen Dienste (d. h. die Montage von Bauteilen) über eine Schnittstelle nach außen an. Zudem verfügt sie über zwei externe Ports, die den Standort der beiden Werkstückträger als CarrierStation symbolisieren. Dadurch wird die externe, mechanische Schnittstelle der Montagezelle definiert. Das bedeutet, dass der Fertigungsprozess gestartet werden kann, sofern sich

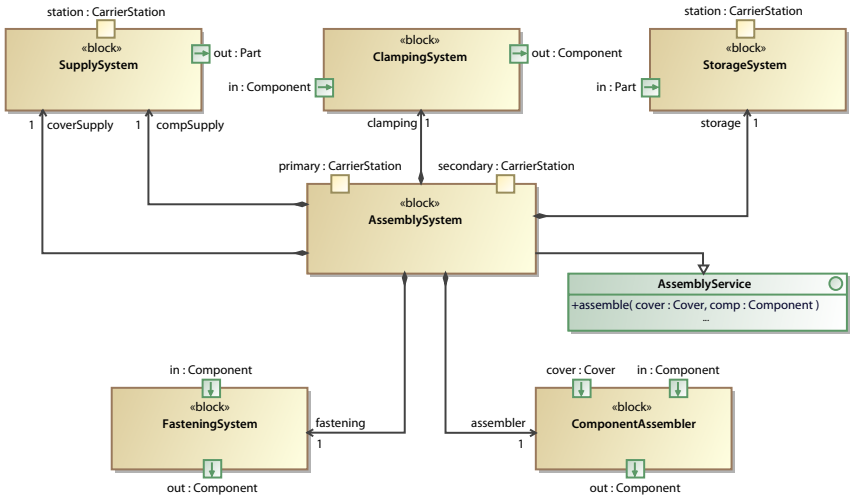


Abbildung 9.3: Logischer Aufbau einer Roboterzelle am Beispiel der Factory 2020: Das Blockdefinitionsdiagramm zeigt auf, aus welchen logischen Systembausteinen sich die Montagezelle zusammensetzt. Die beiden Ports des AssemblySystem stellen die externe mechanische Schnittstelle der Montagezelle dar.

dort zwei Werkstückträger in einer definierten Position befinden. Die beiden Stationen sind als *Full Ports* [95, S. 149] modelliert und damit ein fester Bestandteil des Systems.

Im Gegensatz zur UML kennzeichnet in SysML eine Komposition eine Teil-Ganzes-Beziehung. Das bedeutet, dass das AssemblySystem aus sechs weiteren Systembausteinen besteht, welche die oben beschriebenen logischen Einheiten repräsentieren. Sie sind fester Bestandteil des Gesamtsystems. Dabei ist das SupplySystem allgemein definiert, sodass es sowohl für die Anlieferung der Basisbauteile als auch der Deckel dienen kann. Das SupplySystem verfügt über einen Port, der seine externe, mechanische Schnittstelle definiert und als CarrierStation vom gleichen Typ ist wie die Ports der gesamten Montagezelle. Über diesen Port hat das SupplySystem Zugriff auf einen der beiden Werkstückträger und kann darüber das System mit den entsprechenden Teilen versorgen. Insgesamt benötigt das AssemblySystem zwei Instanzen eines SupplySystems, um die Anlieferung sowohl mit Basisbauteilen als auch mit Deckeln sicherzustellen (vgl. Abb. 9.3).

Das StorageSystem ist analog zum SupplySystem aufgebaut. Über einen Port wird eine Verbindung zu einer CarrierStation hergestellt, um darüber Zugriff auf einen Carrier als geordneten Speicher für fertiggestellte Bauteile zu haben. Dadurch ist das StorageSystem in der Lage die Bauteile in den Werkstückträger zu legen. Daneben sind ein ClampingSystem für das Einspannen des Basisbauteils sowie ein ComponentAssembler und ein FasteningSystem für das Montieren bzw. Verschrauben der Bauteile vorgesehen. Diese Systembausteine sind jeweils in einfacher Ausführung für die Montagezelle notwendig. Damit werden alle farblich markierten logischen Einheiten, die im symbolischen Funktionsplan (vgl. Abb. 9.2) identifiziert wurden, modelliert.

Die Handhabungsobjekte, die einem Systembaustein als Eingabe oder als Ausgabe dienen, werden als *Flow Ports* [95, S. 179] modelliert. Diese Ports sind in Abbildung 9.3 mit einem Pfeil dargestellt, der die Flussrichtung des Objekts anzeigt. Durch einen eingehenden *Flow Port* wird signalisiert, dass der entsprechende Systembaustein ein Handhabungsobjekt des spezifizier-

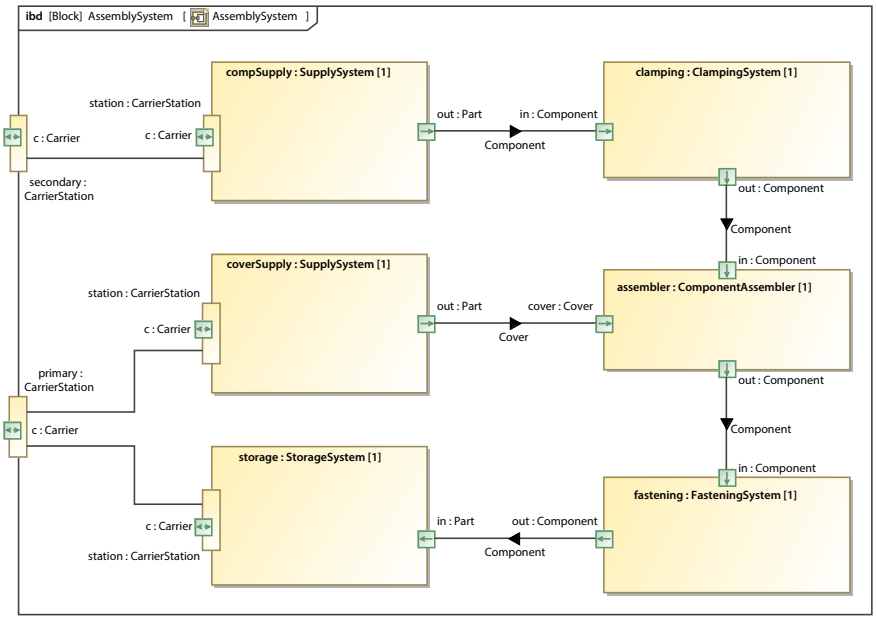


Abbildung 9.4: Logischer Aufbau einer Roboterzelle am Beispiel der Factory 2020: Das Interne Blockdiagramm zeigt, wie die einzelnen Systembausteine zusammenhängen und welche Handhabungsobjekte zwischen diesen ausgetauscht werden.

ten Typs und in einer entsprechenden Rolle benötigt. Der Systembaustein wendet einen oder mehrere Handhabungs- bzw. Fertigungsschritte auf das Objekt an. Folglich wird durch einen ausgehenden *Flow Port* signalisiert, dass der entsprechende Systembaustein ein Handhabungsobjekt des spezifizierten Typs nach Anwendungen der notwendigen Funktionen wieder dem System der Roboterzelle zuführt. Dementsprechend kann ein SupplySystem ein Bauteil aus einem Werkstückträger in das System geben. Analog nimmt ein StorageSystem ein Bauteil aus dem System und legt es in einen Werkstückträger.

Interne Struktur einer Roboterzelle

Die Ports in einem Blockdefinitionsdiagramm (BDD) geben noch keine Auskunft darüber, wie die interne Verschaltung der einzelnen Komponenten eines Systems realisiert ist. Daher existiert in der SysML das interne Blockdiagramm (IBD) für die Modellierung des inneren Aufbaus eines Systems oder einer Komponente des Systems. In Abbildung 9.4 ist dementsprechend die interne Struktur für das oben vorgestellte AssemblySystem dargestellt. Dabei sind alle mit dem AssemblySystem verbundenen Systembausteine aus Abbildung 9.3 in der richtigen Multiplizität instanziiert, d. h. das interne Blockdiagramm enthält sechs Systembausteine.

Im internen Blockdiagramm sind die Ports der inneren Systembausteine und die beiden *Full Ports* des Gesamtsystem, d. h. der Montagezelle, eingezeichnet. Die beiden *Full Ports* des AssemblySystems werden mit dem SupplySystem verbunden, der die Station und den darin befindlichen Werkstückträger bedient. Der interne *Flow Port* zeigt an, dass über eine CarrierStation ein Carrier in das System gelangen kann. Gleiches gilt analog für das StorageSystem. Konkret bedeutet dies, dass Basisbauteile über den Werkstückträger der zweiten Station und dem zugeordneten SupplySystem in die Zelle gelangen. Dagegen werden die notwendigen Deckel über den Werkstückträger der ersten Station und dem dieser Station zugeordneten

SupplySystem dem Montageprozess zugeführt. Fertig montierte und verschraubte Bauteile werden über das StorageSystem ebenfalls in den Werkstückträger der ersten Station gelegt. Damit kann der externe Zufluss von Betriebsstoffen eindeutig modelliert werden.

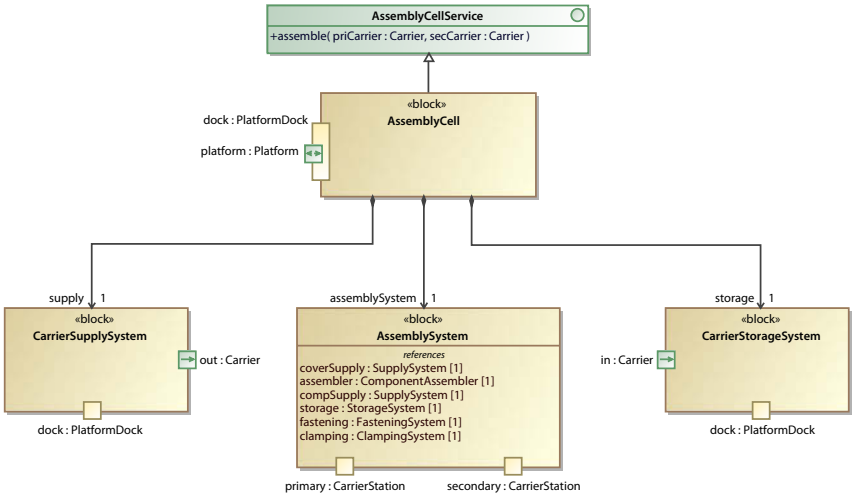
Der Fluss der Handhabungsobjekte durch das System wird ebenfalls modelliert. Dazu werden die *Flow Ports* der Systembausteine miteinander verbunden und der *Item Flow* [95, S. 165] zwischen den Ports spezifiziert, d. h. der Typ des über den Konnektor ausgetauschten Elements wird festgelegt. Der eingebettete *Item Flow* für einen Carrier an den *Full Ports* für die CarrierStations kann ignoriert werden, da sich der Carrier bereits an seiner Station befindet. Dagegen wird in Abbildung 9.4 ein Basisbauteil (vom Typ Component) von einem SupplySystem an das ClampingSystem weitergegeben. Anschließend werden dem ComponentAssembler über das ClampingSystem ein Basisbauteil und über ein SupplySystem ein Deckel (vom Typ Cover) zugeführt. Über das FasteningSystem führt der Weg des Bauteils zum StorageSystem und von dort zurück in den Werkstückträger.

Die Ports der Systembausteine repräsentieren die internen Schnittstellen einer Roboterzelle. Obwohl die Systembausteine nur logische Einheiten repräsentieren, sind die Port als Schnittstellen von großer Bedeutung für den Systementwurf, da sie einen Kontrakt zwischen beteiligten Systembausteinen definieren. Das bedeutet, jeder eingehende Port hat gewisse Vorbedingungen, die von dem übergebenem Element erfüllt sein müssen, und jeder ausgehende Port definiert Nachbedingungen, die das Element erfüllt. Dementsprechend müssen die Vor- und Nachbedingungen der Ports übereinstimmen. Folglich muss bei der Implementierung der (serviceorientierten) Automatisierungssoftware darauf geachtet werden, dass die Schnittstellen der internen Systembausteine miteinander kompatibel sind.

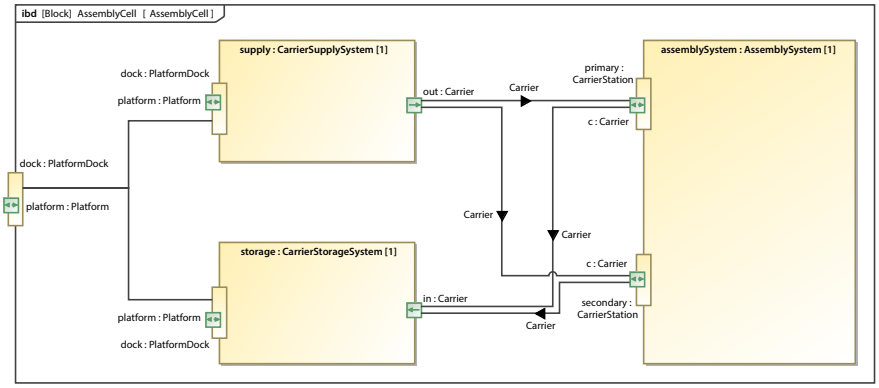
Hierarchische Gliederung von Roboterzellen

In Abbildung 9.3 wurden die beiden Stationen der Werkstückträger als externe, mechanische Schnittstelle der Montagezelle der Factory 2020 definiert. Jedoch wurde in Kapitel 8 beschrieben, dass die Werkstückträger mit einer mobilen Plattform an die Werkbank angeliefert wurden. Daher existiert eine weitere externe Schnittstelle, um die Anlieferung über die mobilen Plattform zu realisieren. Die Werkstückträger stellen somit nur eine mechanische Schnittstelle für einen inneren Systembaustein der Factory 2020 dar. Dieser Systemteil kann unabhängig von der Anlieferung über eine mobilen Plattform betrachtet werden. Maßgeblich sind ausschließlich die beiden Stationen und Werkstückträger als Schnittstelle. Durch eine Erweiterung der ursprünglichen Montagezelle kann eine weitere externe Schnittstelle realisiert werden. Dazu wird das AssemblySystem als Systembaustein eines größeren Systems gekapselt und durch weitere Systembausteine ergänzt. Dieses Vorgehen ermöglicht eine schrittweise hierarchische Gliederung von Roboterzellen. Mit dem in dieser Arbeit entwickelten Modellierungsansatz können neue mechanische Schnittstellen einer Roboterzelle elegant hinzugefügt bzw. existierende Schnittstellen geändert werden, ohne dass das bestehende Kernsystem von der Änderung betroffen wäre.

Die erweiterte und als AssemblyCell bezeichnete Roboterzelle ist in Abbildung 9.5 (a) als Blockdefinitionsdiagramm dargestellt. Der zentrale Systembaustein der AssemblyCell ist das bereits vorgestellte AssemblySystem mit den beiden Ports für die Werkstückträger und den Verweisen auf die inneren Systembausteine. Dazu kommen mit dem CarrierSupplySystem und dem CarrierStorageSystem noch zwei weitere Systembausteine, die für den Transport der Werkstückträger von der mobilen Plattform auf die Werkbank und zurück zuständig sind. Die AssemblyCell weist nach außen



(a) Blockdefinitionsdiagramm



(b) Internes Blockdiagramm

Abbildung 9.5: Eine Roboterzelle kann schichtweise modelliert werden, um weitere externe Schnittstellen zu realisieren. Die Factory 2020 kann so um eine Schnittstelle für die mobile Plattform erweitert werden.

einen als PlatformDock ausgewiesenen Port auf, der die Parkposition der mobilen Plattform vor der Werkbank repräsentiert. Über diesen Port kann eine mobile Plattform – repräsentiert als Platform – mit der Roboterzelle interagieren.

Die innere Struktur der AssemblyCell ist in Abbildung 9.5 (b) als internes Blockdiagramm dargestellt. Darin ist das PlatformDock als externe Schnittstelle der AssemblyCell dargestellt. Über das PlatformDock bedient das CarrierSupplySystem die beiden Ports des Montagesystems mit Werkstückträgern. Die ist durch einen *Item Flow* zum inneren Port einer CarrierStation gekennzeichnet. Analog wird der Transport der Werkstückträger über das CarrierStorageSystem auf die Plattform als *Item Flow* dargestellt. Hierbei zeigt sich ebenfalls die im Vergleich zu Abbildung 9.4 unterschiedliche externe Nutzung der CarrierStation als Port.

Sowohl die AssemblyCell als auch das AssemblySystem verfügen über eine softwaretechnische Schnittstelle, um ihre Funktionalität als Service anzubieten. Sie unterscheiden sich in der Granularität, was jedoch durch die unterschiedlichen mechanischen Schnittstellen bedingt ist. Demzufolge spiegeln die softwaretechnischen Schnittstellen durch ihre Argumente in einem gewissen Maße die mechanischen Schnittstellen wieder. Mit der SysML ist es

jedoch möglich, beide Arten von Schnittstellen, d. h. die mechanischen und softwaretechnischen, explizit zu modellieren. Wie in diesem Abschnitt gezeigt wurde, eignet sich die SysML daher, um die logische Gliederung einer Roboterzelle mit ihren Schnittstellen zu modellieren.

9.2 ARBEITSABLÄUFE ALS ORCHESTRIERUNG VON SERVICES

Zur Modellierung der Handhabungs- und Fertigungsfunktionen werden Services verwendet. Das bedeutet, dass jede modellierte logische Einheit die ihr zugewiesenen Funktionen in einer softwaretechnischen Schnittstelle bündelt und als Service zu Verfügung stellt. Dabei bietet ein logischer Systembaustein immer über einen Service sein Funktionalität an. Durch die serviceorientierte Architektur (SOA) wird eine Entkopplung zwischen der Schnittstelle und der eigentlichen Implementierung erreicht. Letztere muss dem übergeordneten Service bzw. Prozess nicht bekannt sein, solange der Service seine Aufgabe erledigt und den Kontrakt, d. h. die festgelegten Vor- und Nachbedingungen, erfüllt. Jeder Dienst kapselt so eine bestimmte Funktionalität in der Roboterzelle.

Automatisierungsprozesse durch Orchestrierung

Folglich kann sich ein Fertigungsprozess durch eine geeignete Orchestrierung von Services bzw. von deren Methoden ergeben [121]. Unter Orchestrierung (engl.: *orchestration*) wird das flexible Kombinieren mehrerer Services zu einer Komposition beschrieben [150]. Diese Komposition beschreibt in der Regel einen ausführbaren Geschäftsprozess oder im Fall einer Roboterzelle einen Fertigungsprozess. Der Prozess wird von einem ausgewiesenen Teilnehmer überwacht und kontrolliert. Innerhalb der Orchestrierung hat jeder Service einen eingeschränkten Sichtbereich und kann ausschließlich innerhalb seines eigenen Sichtbereichs entscheiden. Die Implementierung und damit die weiteren Aktivitäten hinter einem Service bleiben für den Prozess verborgen.

Zum Beispiel verwendet die AssemblyCell bei der Realisierung ihrer Services Dienste, die von den inneren Systembausteinen und damit auch von dem AssemblySystem angeboten werden. Die AssemblyCell orchestriert diese Dienste zu einem Fertigungsprozess und kontrolliert den Ablauf durch den Zeitpunkt, wann ein Service aufgerufen wird. Das AssemblySystem wiederum stützt sich auf die Dienste seiner inneren Systembausteine ab und orchestriert diese zu einem untergeordneten Fertigungsprozess in der Roboterzelle. Somit ergibt sich automatisch eine hierarchische Gestaltung der softwaretechnischen Komponenten einer Roboterzelle, die mit der hierarchischen Gliederung der Systembausteine übereinstimmt.

Die Services für die Komponenten im AssemblySystem und ihre Zuordnung zu den Systembausteinen sind in Abbildung 9.6 dargestellt. Die einzelnen Methoden der Services entsprechen – auch was ihre Benennung betrifft – weitgehend den in Abbildung 9.2 identifizierten Handhabungs- und Fertigungsfunktionen. Zudem entsprechen die Parameter der Methoden bis auf zwei Ausnahmen den eingehenden Pfeilen der Handhabungs- und Fertigungsfunktionen. So benötigt im SupplyService die Zuteilungsfunktion (*allocate*) ein Bauteil als Eingabeparameter. Damit kann das zuteilende Bauteil spezifiziert werden. Alternativ kann die Funktion ohne Eingabeparameter implementiert werden, wenn es ausreicht, ein zufälliges Bauteil zuzuteilen. Darüber hinaus kapselt der FasteningService den kompletten Prozess des Verschraubens in einer Methode (*fasten*).

Das Abstraktionsniveau der Funktionen ist dabei auf der in Abschnitt 7.2 vorgestellten Prozessebene, da die Aufgabe ausschließlich als Ablauf ele-

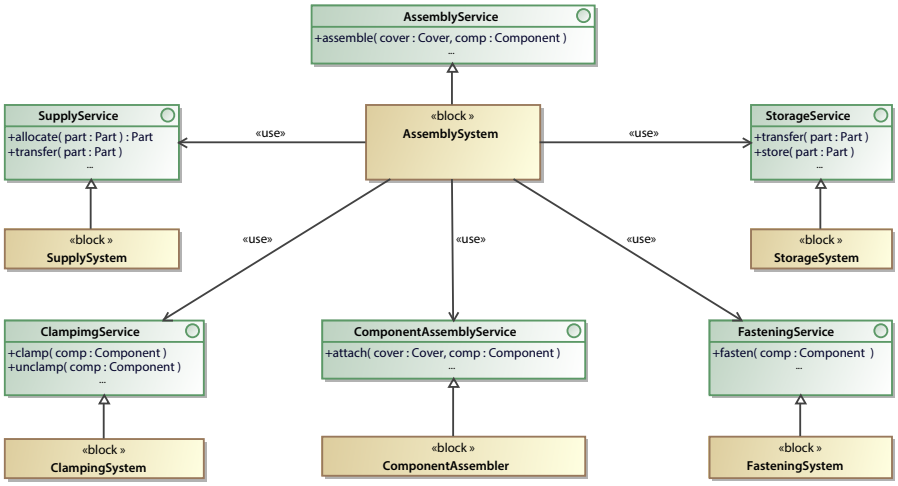


Abbildung 9.6: Um einen Dienst für die Montage der Bauteile anbieten zu können, ist das AssemblySystem auf die Dienste seiner inneren Systembausteine angewiesen. Diese stellen ihre Handhabungs- und Fertigungsfunktionen als Service zur Verfügung. Dabei bietet ein logischer Systembaustein immer über einen Service sein Funktionalität an.

mentarer Handhabungs- und Fertigungsaufgaben beschrieben werden kann. Dabei basiert die Spezifikation der Funktionen lediglich auf den Handhabungsobjekten. Da auf dieser Ebene vollständig von den verwendeten Manipulatoren und Werkzeugen abstrahiert wird, haben die vorgestellten Funktionen die Granularität von *Tasks* (vgl. Abb. 7.2). Erst die Implementierung durch aufgabenzentrierte Dienste oder *Task-Level Services* definiert die notwendigen Werkzeuge und Manipulatoren (z. B. die Implementierung für einen SupplyService). Da sich eine Implementierung für den AssemblyService vollständig auf aufgabenzentrierte Dienste abstützt, handelt es sich hierbei um einen prozesszentrierten Dienst oder *Process-Level Service*.

Modellierung mit Aktivitätsdiagrammen

Für die Modellierung der Prozesse werden die ebenfalls in SysML enthaltenen Aktivitätsdiagramme verwendet. Durch Aktivitätsdiagramme können zeitliche Abläufe beschrieben sowie Ein- und Ausgabedaten spezifiziert werden, die während des Ablaufs benötigt werden oder durch den Ablauf entstehen. Damit kann nicht nur der Kontrollfluss der Roboterzelle isoliert betrachtet werden, sondern gleichzeitig auch der Objektfluss. Aktivitätsdiagramme besitzen eine Tokensemantik [95], die sehr ähnlich zu Petrinetzen [228] ist. Damit existieren präzise Regeln für den Kontroll- und Objektfluss, der geteilt oder wieder synchronisiert werden kann. Ein Token repräsentiert das Fortschreiten des Kontroll- bzw. Datenflusses und entspricht einem Ausführungsstrang, der erzeugt und vernichtet werden kann. Damit können Abläufe in einem Aktivitätsdiagramm nicht nur sequentiell, sondern auch nebenläufig sein.

Ein Aktivitätsdiagramm repräsentiert eine Aktivität mit Eingabe- und Ausgabeparametern, d. h. eine Tätigkeit im System. Während der Ausführung werden dabei die Eingaben in Ausgaben transformiert [95]. Durch ein Aktivitätsdiagramm werden die Aktionen einer Aktivität zusammen mit der zeitlichen Abfolge und dem Fluss der Ein- und Ausgabeparameter definiert. Eine Aktion ist dabei u. a. der Aufruf einer weiteren Aktivität, die ihrerseits über ein Aktivitätsdiagramm beschrieben wird. Dadurch lassen sich Prozesse und zeitliche Abläufe verschiedener Abstraktionsniveaus sehr gut beschreiben, d. h. auch hierarchisch gegliederte Prozesse, wie sie in Ro-

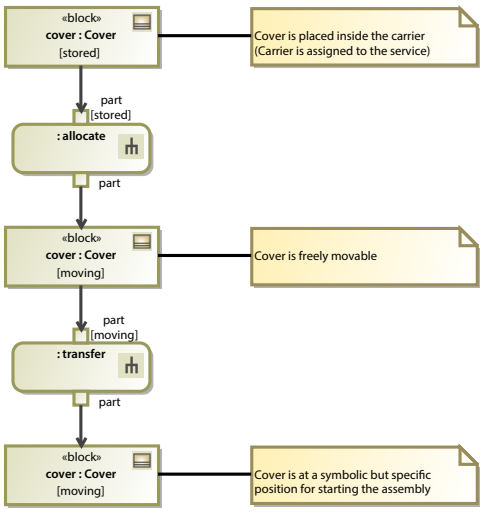


Abbildung 9.7: Der zu montierende Deckel wird über zwei Handhabungsfunktionen aus dem Werkstückträger genommen und der Montagefunktion zugeführt. Während der Ausführung gelten für das Bauteil unterschiedliche Zustände, die an den Eingabepins dargestellt und durch eine Notiz beschrieben sind.

boterzellen vorkommen. Neben Aktionen besitzt eine Aktivität eine Menge von Objektknoten, die Eingabe- und Ausgabedaten sowie zwischenzeitlich entstandene Objekte repräsentieren.

Abbildung 9.7 zeigt den sequentiellen Ablauf für das Zuteilen und Weitergeben eines Bauteils. Der Ablauf besteht aus zwei Aktivitäten, die als Rechteck mit abgerundeten Ecken dargestellt sind und den Funktionen des SupplyService entsprechen. Durch die *Pins* der Aktivitäten wird jeweils ein Eingabe- und Ausgabeparameter repräsentiert. Dabei kann spezifiziert werden, in welchem Zustand sich ein Objekt bei einem Eingabeparameter befinden muss, um akzeptiert zu werden. Analog kann spezifiziert werden, in welchem Zustand ein Objekt als Ausgabeparameter die Aktivität verlässt. Der erwartete bzw. resultierende Zustand des Objektes wird in einer eckigen Klammer dargestellt.

In Abbildung 9.7 repräsentieren die Rechtecke als Objektknoten das Bauteil, das durch die Aktivitäten transformiert wird. Diese Transformation kann zum einen räumlicher und zum anderen topologischer Art sein. Bei einer räumlichen Transformation wird das Objekt gemessen zu einem festen Bezugssystem in eine andere Lage gebracht. Bei einer topologischen Transformation ändert sich die Struktur der Roboterzelle, d. h. das Bauteil befindet sich nach der Aktivität nicht mehr im Werkstückträger. Bei einer topologischen Transformation ändert sich immer auch der Zustand des Handhabungsobjekts.

Der Prozessablauf als Objektfluss

Der Ablauf wird allein durch den Objektfluss des Bauteils bestimmt. Der zu montierende Deckel, der als *cover* bezeichnet ist, fließt zuerst in die Aktivität *allocate* und wird dort dem Prozess zugeteilt. Dabei ändert er seinen Zustand. Während er davor statisch im Werkstückträger platziert war, ist er mittlerweile nicht mehr dort und kann für den Fertigungsprozess frei im Raum bewegt werden. Der Zustand des Bauteils repräsentiert die Vor- und Nachbedingungen der Aktivitäten. Die möglichen Zustände eines Bauteils können als Zustandsdiagramm modelliert werden. Somit kann der erwartete Zustand an den Ein- und Ausgabepins einer Aktivität gekennzeichnet

werden. Ergänzend oder weniger formal können diese Informationen auch durch Notizen beschrieben werden. Sobald die Aktivität *allocate* abgeschlossen ist, steht der nun zugeteilte Deckel dem weiteren Ablauf wieder zur Verfügung und kann über die Aktivität *transfer* weitergegeben werden. Dabei ändert er seine räumliche Lage.

Die beiden in Abbildung 9.7 dargestellten Aktivitäten sind, wie auch der SupplyService, generisch für ein abstraktes Bauteil, repräsentiert als Part, definiert. Durch die Verwendung der Objektknoten wird der Typ spezialisiert, z. B. zu einem Deckel (Cover) wie in Abbildung 9.7. Jedoch lassen sich beide Aktivitäten auch für das Basisbauteil (Component) verwenden. Die an den Ein- und Ausgabepins sowie den Objektknoten markierten Zustände müssen daher für den abstrakten Typ, d. h. das Part, definiert werden. Durch die Generalisierung von Aktivitäten und Services kann auf diesem Abstraktionsniveau Wiederverwendung erreicht werden.

Abbildung 9.8 zeigt den vollständigen Ablauf der Montage und des Verschraubens der Bauteile als Aktivitätsdiagramm. Diese Aktivität entspricht der im AssemblyService spezifizierten Methode *assemble* (vgl. Abb. 9.6). Dabei werden Deckel und Basisbauteil durch die bereits vorgestellten Aktivitäten nebenläufig angeliefert (vgl. Abb. 9.7). Um diese Parallelität zu erreichen, wird der Kontrollfluss zu Beginn geteilt. Danach wird der Ablauf ausschließlich durch einen Objektfluss beschrieben. Nach der Anlieferung der beiden Bauteile bzw. nach dem Einspannen des Basisbauteils wird der Ablauf durch die Aktivität des Fügens wieder synchronisiert. Ein separater Kontrollfluss ist dazu nicht notwendig. Der Fluss der Handhabungsobjekte durch das System ist für diese Beschreibung ausreichend.

Vor- und Nachbedingungen

Nach dem Fügen wird der Fertigungsprozess sequentiell fortgesetzt bis das fertige Bauteil in den Werkstückträger zurückgelegt wurde. An den Eingabe- und Ausgabepins sind die erwarteten und resultierenden Zustände der Bauteile markiert. Dabei sind in Abbildung 9.8 nur signifikante Zustandsänderungen dargestellt. Die Zustände an den Eingabe- und Ausgabepins zeigen an, wie die Handhabungsobjekte im Verlauf der Aktivität transformiert werden. Dabei markieren drei disjunkte Zustände (*stored*, *moving* und *clamped*), wie und ob die Bauteile gehandhabt werden können. Drei weitere ebenfalls disjunkte Zustände (*unprocessed*, *cover attached* und *cover bolted*) geben den aktuellen Bearbeitungsstand des Basisbauteils (Component) an. Durch ein Aktivitätsdiagramm lassen sich folglich nicht nur die zeitlichen Abläufe von Fertigungsprozessen modellieren, sondern auch die unterschiedlichen Zustände, die ein Handhabungsobjekt während der Fertigung einnimmt bzw. einnehmen muss. Daraus lassen sich später die Vor- und Nachbedingungen der einzelnen Servicemethoden extrahieren.

Im symbolischen Funktionsplan (vgl. Abb. 9.2) wurde das Verschrauben der beiden Bauteile als eine Folge mehrerer Handhabungs- und Fertigungsfunktionen dargestellt. Dagegen wurde dieser Vorgang sowohl in Abbildung 9.8 als auch im FasteningService (vgl. Abb. 9.6) als eine einzige Aktivität bzw. Methode aufgeführt. Intern wird diese Aktivität auf mehrere Handhabungs- und Fertigungsfunktionen abgebildet, die wiederholt durchlaufen werden. Um den repetitiven Ablauf zu verbergen, wurde eine Kapselung der Funktionen vorgenommen.

Der interne Ablauf des Schraubens ist in Abbildung 9.9 dargestellt. Durch einen Mengenverarbeitungsbereich (vgl. den Stereotyp *iterative*) wird der Vorgang für jede Schraube bzw. jede Bohrung des Bauteils durchgeführt. Für eine einzelne Schraube besteht der Vorgang wie im symbolischen Funktionsplan (vgl. Abb. 9.2) aus drei elementaren Funktionen. Demzufolge wird

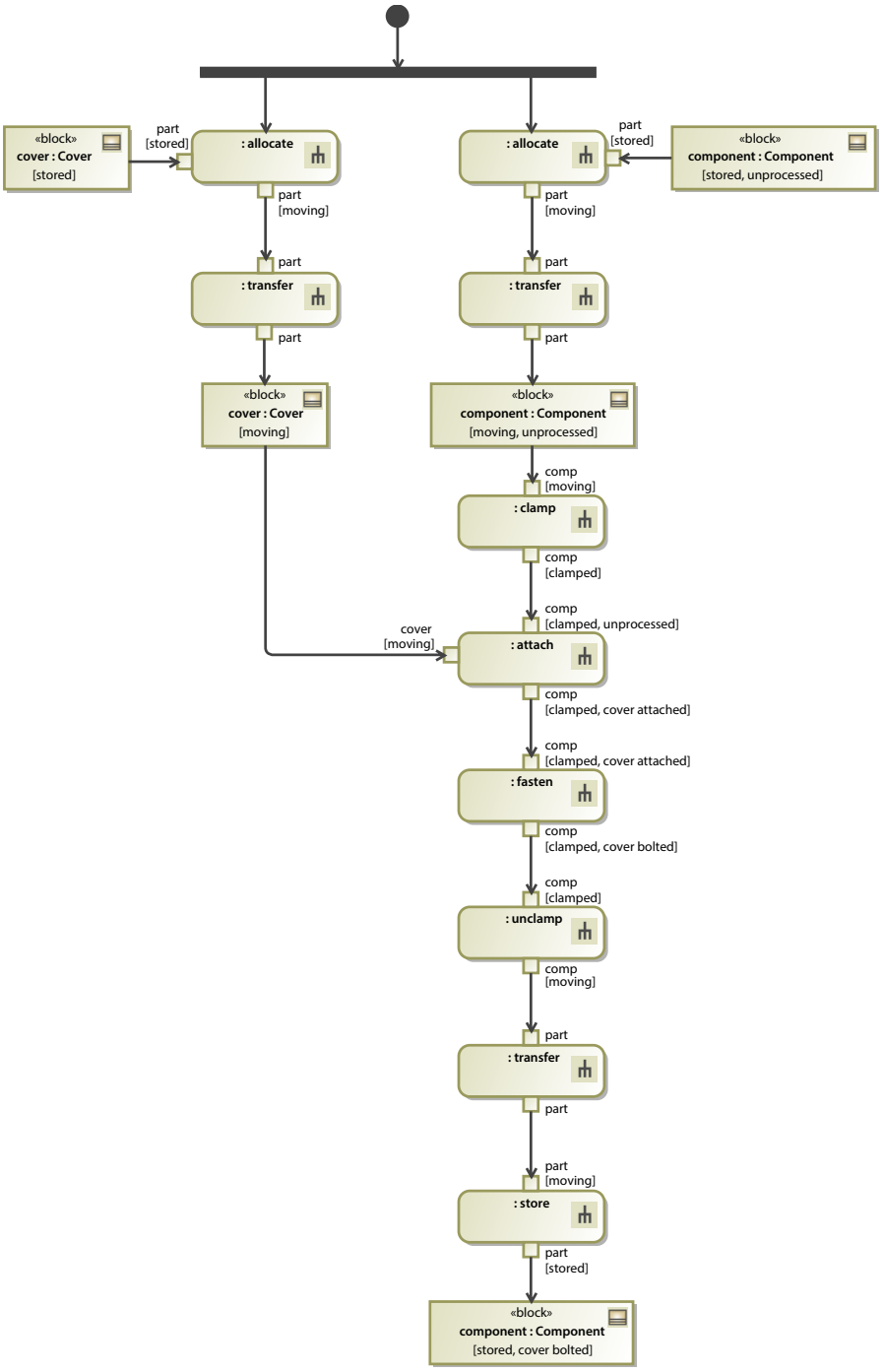


Abbildung 9.8: Das Aktivitätsdiagramm zeigt den vollständigen Ablauf der Montage und des Verschrauben eines Bauteils. Die Anlieferung der beiden Teile erfolgt dabei nebenläufig. Vor der Montage wird der Prozess synchronisiert und läuft anschließend sequentiell ab. An den Ein- und Ausgabepins werden signifikante Zustandsänderungen der Handhabungsobjekte dargestellt.

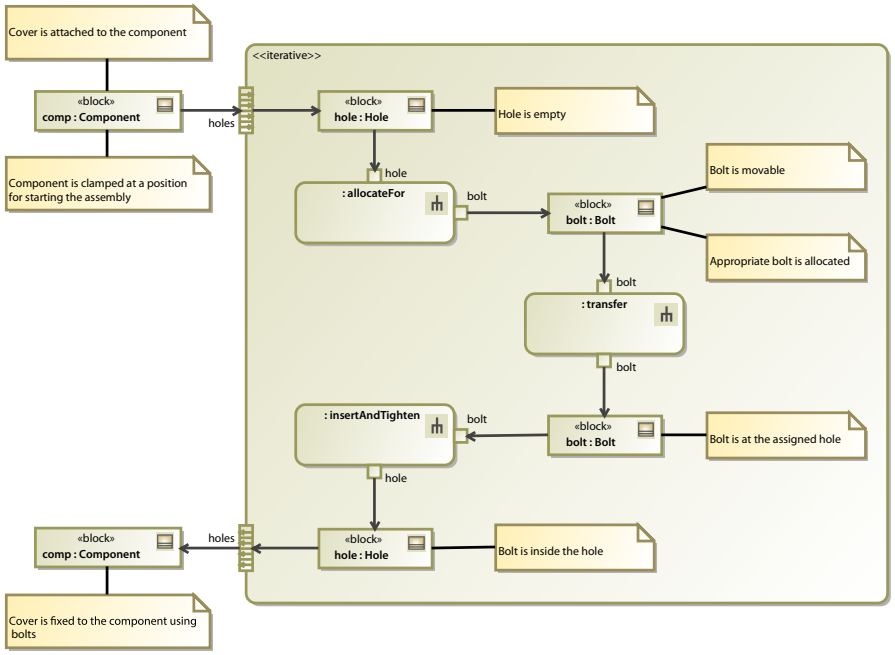


Abbildung 9.9: Das Aktivitätsdiagramm zeigt wiederholt den Vorgang des Verschraubens an. Dabei wird jede Schraube durch den Mengenverarbeitungsbereich separat betrachtet.

zuerst eine Schraube zugeteilt. Die Schraube wird anhand der Bohrung ausgewählt. Anschließend wird die Schraube weitergegeben und im letzten Schritt in die Bohrung gesteckt und angezogen.

Alle in diesem Abschnitt vorgestellten Funktionen sind nur auf Basis der Handhabungsobjekte definiert und daher lösungsneutral. Sie ermöglichen eine Programmierung auf Prozessebene, wobei ihre Granularität den in Abschnitt 7.2 vorgestellten *Tasks* entspricht. In der vorliegenden Arbeit wird damit gezeigt, dass die einzelnen lösungsneutralen Funktionen bzw. *Tasks* durch eine geeignete Orchestrierung zu komplexen Fertigungsprozessen komponiert werden können. Einige dieser Funktionen, z. B. das Verschrauben, können sich auf andere lösungsneutrale Funktionen abstützen. Jedoch müssen elementare Funktionen durch die Implementierung des entsprechenden Services auf die verfügbaren Werkzeuge und Manipulatoren der Zelle abgebildet werden.

9.3 ABBILDUNG DER LOGISCHEN STRUKTUR UND IHRER SERVICES

Die in Abschnitt 9.1 vorgestellte logische Modellierung und Unterteilung von Roboterzellen abstrahiert vollständig von den vorhandenen Manipulatoren, Werkzeugen und sonstigen Geräten, die Bestandteil der Zelle sind. Durch die logische Modellierung werden rein virtuelle Einheiten geschaffen, die jedoch durch die ihnen zugewiesenen Services eine klare definierte Zuständigkeit in der Roboterzelle erhalten. Ihre Funktionen erlauben eine auf die Handhabungsobjekte zentrierte Beschreibung der Fertigungsprozesse (vgl. Abschn. 9.2). Da die Dienste bisher lösungsneutral sind, stellen sie das Grundgerüst einer serviceorientierten Roboterzelle dar.

In einem nächsten Schritt ist jedoch eine Abbildung der logischen und rein virtuellen Einheiten auf die in der Roboterzelle vorhandenen Elemente

notwendig. Nur die in der Zelle vorhandenen Aktuatoren oder eine Kombinationen dieser Aktuatoren können die notwendigen Handhabungs- und Fertigungsaufgaben durchführen. Dementsprechend sind die Manipulatoren und ihre Endeffektoren dafür zuständig, die spezifizierten Funktionen als Teil einer virtuellen Einheit zu implementieren. Dabei werden sie unter Umständen von weiteren Geräten und Einrichtungen (z. B. Spannvorrichtungen oder Zuführeinrichtungen für Hilfsstoffe) unterstützt.

Dazu wird die physische Struktur der Roboterzelle zuerst mit der SysML modelliert. Das umfasst die in der Zelle verfügbaren Aktuatoren, Sensoren und sonstigen Einrichtungen sowie deren geometrischen Aufbau. Anschließend findet eine Zuordnung der realen Systembausteine (z. B. der Manipulatoren oder Greifer) zu den logischen Systembausteinen (z. B. dem SupplySystem) statt. Da sich die logische Struktur einer Roboterzelle wieder ändern kann, können die Geräte und Einrichtungen der Zelle bei Bedarf den logischen Einheiten zugeordnet werden und auch Teil mehrerer logischer Einheiten sein.

Während die Services auf dieser Granularität von der physischen oder gegenständlichen Struktur der Roboterzelle abstrahieren, muss ihre Implementierung diese kennen, um die Manipulatoren und ihre Endeffektoren gemäß der Vorgaben zu steuern. Daher werden die logischen Einheiten vollständig durch Software abgebildet. Durch die objektorientierte *Robotics API* kann die physische oder gegenständliche Struktur der Roboterzelle, wie in Kapitel 5 beschrieben wurde, in Software abgebildet werden. Zudem werden die Handhabungsobjekte ebenfalls mit der *Robotics API* als Softwareobjekte nachgebildet. Damit sind alle Elemente einer serviceorientierten Roboterzelle als Software bzw. Softwareobjekt verfügbar. Die Implementierungen der logischen Einheiten bilden immer wieder neue Kompositionen der realen Geräte, um die durch ihre Dienstschnittstelle vorgegebenen Funktionen zu realisieren. Durch die Abbildung der realen Zelle als Softwareobjekte können die Vorbedingungen geprüft werden. Das setzt jedoch voraus, dass jeder Service das objektorientierte Modell der Roboterzelle konsistent hält.

Die Abbildung der logischen Einheiten und ihrer Services wird anhand der in Kapitel 8 beschriebenen Konfiguration der Roboterzelle erläutert, die in Abbildung 9.10 als Blockdefinitionsdiagramm dargestellt ist. Zentraler Systembaustein ist dort die CooperativeRobotCell, die alle anderen Elementen umfasst. Dazu gehören eine Werkbank (Workbench) und die beiden Stationen (CarrierStation) für die Aufnahme der Werkstückträger. Durch dieses statische Element besteht eine Verbindung zwischen dem logischen und dem physischen Modellen. Zudem ist ein Depot (BoltDepot) als geordneter Speicher für Schrauben (Bolt) ein Bestandteil der Zelle.

Darüber hinaus gehören zwei KUKA Leichtbauroboter zu der kooperativen Roboterzelle. Diese sind in jeweils einer unterschiedlichen Rolle – als linker und als rechter Manipulator – der Zelle zugeordnet. Gleiches gilt auch für die elektrischen Parallelgreifer, die ebenfalls einmal als linker und einmal als rechter Greifer ausgewiesen sind. Das elektrische Schraubsystem kommt nur einmal in der Zelle vor und benötigt daher keine ausgewiesene Rolle. Obwohl sie in der Zelle vorhanden sind, wurden weitere Bestandteile der Zelle (z. B. Werkzeugwechseladapter) in Abbildung 9.10 aus Gründen der Übersichtlichkeit nicht modelliert. Diese haben keine aktive Funktion, beeinflussen jedoch die Geometrie der Roboterzelle und sollten zumindest in der Implementierung berücksichtigt werden.

Eine Besonderheit sind die in Abbildung 9.10 dargestellten *Ports* der einzelnen Systembausteine. Im Gegensatz zu den *Ports* der logischen System-

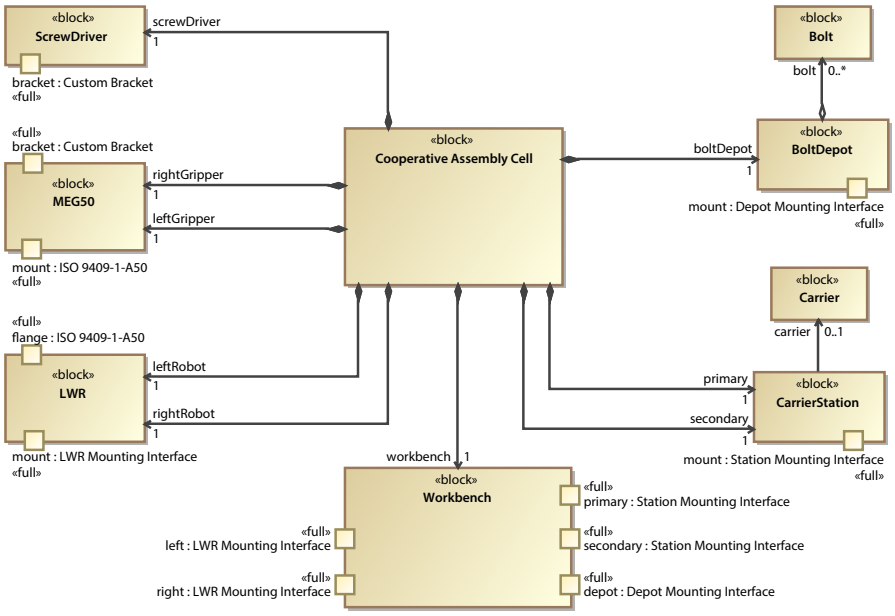


Abbildung 9.10: Die physische Struktur der Multiroboterzelle wird ebenfalls mit SysML als Blockdefinitionsdiagramm modelliert.

bausteine (vgl. Abb. 9.5) modellieren sie nicht den Fluss der Handhabungsobjekte durch das System, sondern repräsentieren die mechanischen Schnittstellen der Zellenelemente. Dementsprechend werden die einzelnen Systembausteine über diese Ports miteinander verbunden, um die Topologie der Zelle zu kreieren. Durch die Verwendung der SysML können – neben softwaretechnischen und elektrischen – auch mechanische Schnittstellen modelliert werden.

Die topologische Struktur der Roboterzelle ist in Abbildung 9.11 als internes Blockdiagramm dargestellt. Unter Verwendung der Ports werden die einzelnen Entitäten der Zelle miteinander verknüpft. Jeder Konnektor repräsentiert eine statische, mechanische Verbindung zwischen den beiden betroffenen Entitäten. So hat die Werkbank insgesamt fünf verschiedene mechanische Schnittstellen (z. B. in Form von Bohrmustern), an denen die Roboter, die Stationen der Werkstückträger und das Schraubenmagazin angebracht sind. Die mechanischen Schnittstellen sind dabei sehr spezifisch für die Roboterzelle und die verwendeten Elemente. Jeweils am Flansch eines Roboters ist der entsprechende Greifer befestigt. Zusätzlich ist an einem der beiden Greifer das elektrische Schraubsystem über eine spezielle mechanische Konstruktion befestigt.

Die topologische und geometrische Struktur kann vollständig durch die *Robotics API* in Software abgebildet werden. Dazu kann auf vorhandene Aktuatoren wie die beiden KUKA Leichtbauroboter (vgl. Abschn. 6.4.3) oder die Parallelgreifer von Schunk (vgl. Abschn. 6.4.2) zurückgegriffen werden. Die anderen Entitäten oder physischen Objekte, speziell solche die sehr spezifisch für die Zelle sind, müssen neu modelliert und implementiert werden. Danach kann die virtuelle Roboterzelle konfiguriert und instanziiert werden. Gegebenenfalls müssen exakte Positionen an der realen Zelle eingemessen und in die Konfiguration übertragen werden.

Anschließend kann die Abbildung von logischen Systembausteinen (vgl. Abb. 9.3) auf die realen Entitäten der Roboterzelle (vgl. Abb. 9.10) erfolgen. Das bedeutet, dass für jeden identifizierten logischen Systembaustein eine

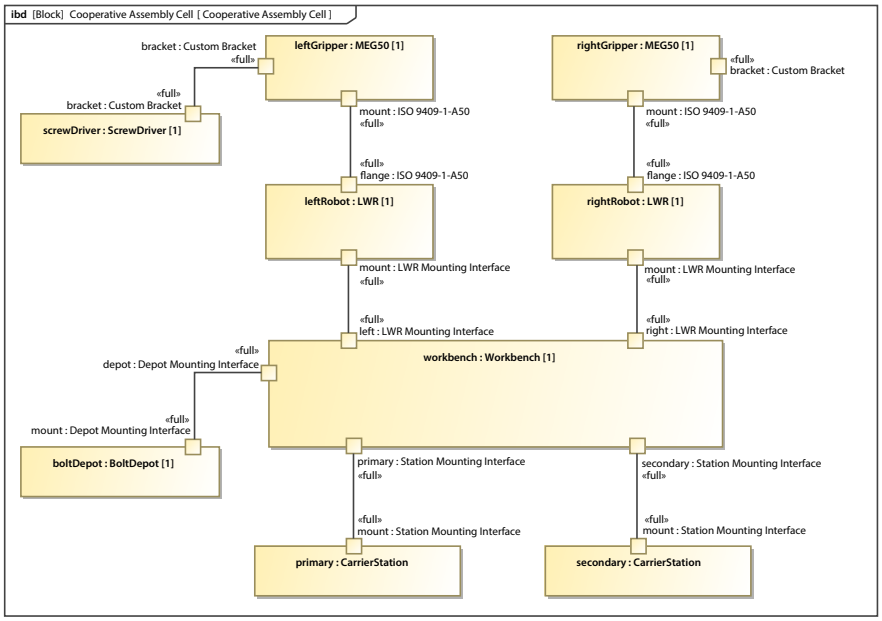


Abbildung 9.11: Das interne Blockdiagramm stellt den topologischen Aufbau der kooperativen Factory 2020 dar.

Komponente entwickelt werden muss, die einerseits den Service implementiert und anderseits die verwendeten Aktuatoren und Sensoren über die *Robotics API* gemäß den spezifizierten Anforderungen ansteuert. Die Zuordnung der Aktuatoren, Sensoren und weiteren physischen Entitäten zu den logischen Systembausteinen ist in Tabelle 9.2 dargestellt.

Die Services können mit den zugeordneten physischen Entitäten wie folgt implementiert werden:

- Der SupplyService für den Deckel bewegt den linken Manipulator zum linken Werkstückträger an der primären CarrierStation, um dort das entsprechende Bauteil mit seinem Parallelgreifer aufzunehmen. Anschließend transportiert der Roboter das Bauteil in die Mitte der Werkbank zu einer Bauteil-abhängigen Position.

- Der SupplyService für das Basisbauteil bewegt den rechten Manipulator zum rechten Werkstückträger an der sekundären CarrierStation, um dort das entsprechende Bauteil mit seinem Parallelgreifer aufzunehmen. Anschließend transportiert der Roboter das Basisbauteil in die Mitte der Werkbank zu einer Bauteil-abhängigen Position.

- Da das Basisbauteil bereits vom rechten Manipulator bzw. dessen Greifer kraftschlüssig gehalten wird, führt der ClampingService beim Einspannen keine weitere Aktion durch. Eventuell müssen die Regelungsparameter des Manipulators angepasst werden. Allerdings muss beim Entspannen das fertige Bauteil vom rechten Manipulator zum linken Manipulator übergeben werden.

- Der AssemblerService steuert den linken Roboter kraftgesteuert, um den Deckel sicher auf dem Bauteil zu platzieren. Der Greifer muss anschließend den Deckel loslassen. Der rechte Greifer hält dabei weiterhin das Basisbauteil. Gegebenenfalls passt der rechte Roboter die Parameter für den Steifigkeitsregler an.

SYSTEMBAUSTEIN (SERVICE)	GERÄTE	PHYSISCHE OBJEKTE
coverSupplySystem : SUPPLYSYSTEM (SUPPLYSERVICE)	*leftRobot* : LWR *leftGripper* : MEG50	*primary* : CARRIERSTATION
componentSupplySystem : SUPPLYSYSTEM (SUPPLYSERVICE)	*rightRobot* : LWR *rightGripper* : MEG50	*secondary* : CARRIERSTATION
clampingSystem : CLAMPINGSYSTEM (CLAMPINGSERVICE)	*leftRobot* : LWR *leftGripper* : MEG50 *rightRobot* : LWR *rightGripper* : MEG50	
assembler : COMPONENTASSEMBLER (ASSEMBLERSERVICE)	*leftRobot* : LWR *leftGripper* : MEG50	
fasteningSystem : FASTENINGSYSTEM (FASTENINGSERVICE)	*leftRobot* : LWR *screwdriver* : SCREWDRIVER	*depot* : BOLTDEPOT
storageSystem : STORAGESYSTEM (STORAGESERVICE)	*leftRobot* : LWR *leftGripper* : MEG50	*primary* : CARRIERSTATION

Tabelle 9.2: Zuordnung der Services und logischen Systembausteine auf die vorhandenen Entitäten der Roboterzelle für die kooperative Montage mit zwei Leichtbaurobotern.

- Beim FASTENINGSERVICE wird das zweite Werkzeug des linken Roboters aktiv. Folglich wird das elektrische Schraubsystem zuerst am Schraubendepot positioniert, um eine Schraube aufzunehmen. Diese wird danach zum Bauteil bzw. dem Gewinde transportiert und dort eingeschraubt. Der Vorgang wird wiederholt, bis alle Gewinde verschraubt sind. Der rechte Greifer hält dabei weiterhin das nun montierte Bauteil. Gegebenenfalls passt der rechte Roboter die Parameter für den Steifigkeitsregler an.
- Nach dem Entspannen wird das Bauteil bereits vom linken Greifer gehalten. Daher muss der STORAGESERVICE den rechten Manipulator zum linken Werkstückträger an der primären CARRIERSTATION bewegen und dort das Bauteil in einen leeren Platz legen.

Durch die Abbildung auf konkrete Manipulatoren und Endeffektoren werden die Vor- und Nachbedingungen verfeinert.

Bei den oben aufgezählten Services handelt es sich um aufgabenzentrierte Dienste, da ihre Schnittstelle nur auf Basis der Handhabungsobjekte definiert ist. Ihre Implementierung stützt sich dagegen auf die notwendigen Werkzeuge und Manipulatoren ab. Eine geeignete Strukturierung dieser Dienste wird später in Abschnitt 9.4 vorgestellt. Der darüber liegende Service implementiert den Fertigungsprozess als Abfolge von Handhabungs- und Fertigungsfunktionen anderer Dienst. Daher handelt es sich hierbei um einen prozesszentrierten Dienst.

Gemäß dem oben beschriebenen Vorgehen können die logischen Einheiten und Services zum Transport der Werkstückträger zwischen der mobilen Plattform und der Werkbank auf beide Manipulatoren und beide Parallelgreifer abgebildet werden. Dadurch kann die kooperative Factory 2020 vollständig mithilfe von prozess- und aufgabenzentrierten Diensten implementiert werden. Eine Abbildung auf eine reduzierte Factory 2020 mit nur einem Manipulator wird in Kapitel 11 vorgestellt. Dadurch wird in dieser Dissertation gezeigt, dass eine logische Modellierung der Aufgaben unabhängig von den verwendeten Manipulatoren und Werkzeugen möglich ist.

9.4 UMSETZUNG VON HANDHABUNGSFUNKTIONEN ALS SERVICE

Auf Basis der logischen Einheiten können lösungsneutrale Handhabungs- und Fertigungsfunktionen als Service (vgl. Abschn. 9.1) definiert und anschließend zu komplexeren Fertigungsprozessen orchestriert werden (vgl. Abschn. 9.2). Trotz unterschiedlicher Bauteile und Werkstückträger ist der generelle Ablauf und die abstrakten Aktionen von Manipulator und Greifer bei einem SupplyService oder einem StorageService ähnlich. Daher gilt es, diese Gemeinsamkeiten in wiederverwendbare und leicht anpassbare Implementierungen zu vereinen. Demzufolge wird in Abschnitt 9.4.1 gezeigt, wie durch eine Abstraktion der Eingabeparameter bzw. der Eingabeobjekte aufgabenzentrierte Dienste wiederverwendbar implementiert werden können. Die Implementierung eines solch wiederverwendbaren Dienstes kann daraufhin durch geeignete Konfiguration oder Spezialisierung auf unterschiedliche Anwendungsfälle hin angepasst werden.

Die Abbildung der Handhabungs- und Fertigungsfunktionen auf die Fähigkeiten der Werkzeuge und Manipulatoren wird in Abschnitt 9.4.2 vorgestellt. Dazu werden wiederum Services mit einer feineren Granularität verwendet. In Abschnitt 9.4.3 wird erläutert, wie das interne Verhalten eines Dienstes als Zustandsmaschine modelliert und implementiert werden kann. Durch diese Techniken kann die Implementierung logischer Einheiten und ihrer Services wiederverwendbar werden. Damit bereitet diese Arbeit den Weg, dass die Automatisierungssoftware von Roboterzellen in Zukunft keine Unikate mehr sind.

9.4.1 *Wiederverwendung durch Abstraktion*

Prozess- oder aufgabenzentrierten Dienste bieten durch ihre Methoden lösungsneutrale Handhabungs- und Fertigungsfunktionen an. Die Handhabungsfunktionen sind allgemein in der VDI-Richtlinie 2860 [269] definiert und nehmen keinen Bezug zu den konkreten Handhabungsobjekten. Einzig die Art des Speicherns, d. h. geordnetes, teilgeordnetes oder ungeordnetes Speichern, gibt einen Aufschluss über die Positionierung und den Ordnungszustand der Handhabungsobjekte. Die Anwendung von Fertigungsfunktionen dagegen ist abhängig von den Bauteilen, da sie die Aufgabe näher beschreiben bzw. bestimmen müssen. Fügen durch Verschrauben ist auf Schrauben, Bohrlöcher und deren Parameter angewiesen. Jedoch ist es auch hier möglich, diese Aufgaben sehr abstrakt und allgemein zu beschreiben.

Die Implementierung eines prozesszentrierten und eines aufgabenzentrierten Dienstes unterscheiden sich allerdings deutlich. Ein prozesszentrierter Dienst verwendet intern entweder andere prozesszentrierte Dienste oder stützt sich auf aufgabenzentrierte Dienste ab. Damit ist er unabhängig von den verwendeten Aktuatoren der Roboterzelle. Beispiele sind die AssemblyCell bzw. dessen AssemblyCellService (vgl. Abschn. 9.1). Dagegen bildet ein aufgabenzentrierter Dienst seine Handhabungs- oder Fertigungsfunktionen auf konkret in der Zelle verfügbare Hardware ab. Hierbei weisen die gleichen Funktionen bei gleicher oder ähnlicher Konfiguration der Aktuatoren Ähnlichkeiten auf, die man in einer abstrakten Serviceimplementierung vereinigen kann, um ein höheres Maß an Wiederverwendung zu erreichen.

In Abbildung 9.12 ist mit dem AbstractSupplySystem ein Beispiel für eine abstrakte Serviceimplementierung gegeben. Diese basiert auf einem geordneten Speicher (ArrangedStorage), der Annahmen trifft über die darin enthaltenen (physischen) Objekte. Ein konkreter geordneter Speicher ist

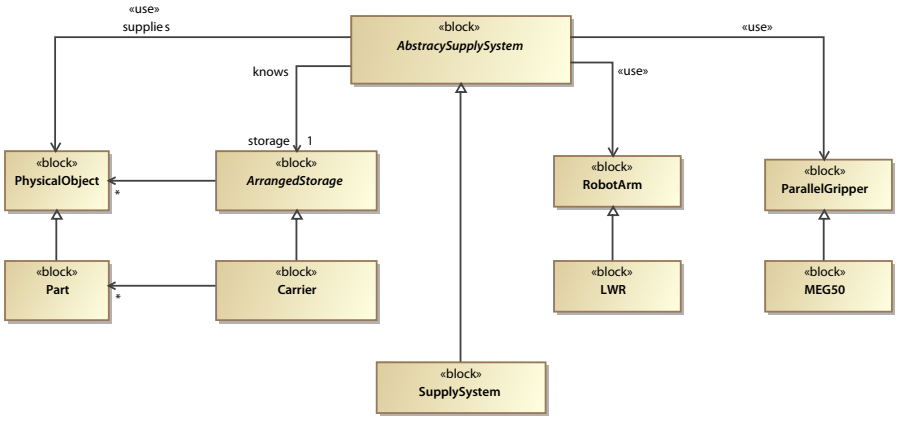


Abbildung 9.12: Das SupplySystem kann abstrakt als Anliefersystem auf einem geordneten Speicher mit einem Roboterarm und einem Greifer implementiert werden.

beispielsweise der Werkstückträger der Factory 2020, der die entsprechenden Bauteile beinhaltet. Softwaretechnisch kann ein ArrangedStorage als Schnittstelle realisiert werden, die Methoden für den Zugriff auf die Objekte und freien Speicherplätze enthält. Ein konkreter Speicher (z. B. der Werkstückträger) implementiert diese Schnittstelle und gibt darüber Auskunft, welche Objekte er aktuell speichert und wie er neue Objekte speichern kann. Die abstrakte Implementierung des Services kann diese Informationen nutzen, um die notwendigen Handhabungsfunktionen zu implementieren.

Abgebildet werden die Handhabungsfunktionen des AbstractSupplySystems auf einen Manipulator und einen Greifer. Folglich wird das *Zuteilen* eines Bauteils realisiert, indem das Objekt zuerst gegriffen und anschließend aus dem geordneten Speicher entnommen wird. Das *Weitergeben* wird als Roboterbewegung zu einem definierten Zielpunkt realisiert. Die genaue Umsetzung wird in Abschnitt 9.4.2 beschrieben. In der Factory 2020 kann das AbstractSupplySystem verwendet und erweitert werden, um die Bauteile aus den Werkstückträgern zu nehmen. Dazu wird der geordneten Speicher mit einem Carrier konkretisiert. Ebenso werden der verwendete Manipulator und Greifer, d. h. ein LWR und ein MEG50, konfiguriert.

Ein zweites Beispiel für eine abstrakte Serviceimplementierung ist in Abbildung 9.13 dargestellt. Das AbstractFasteningSystem implementiert den in Abbildung 9.9 dargestellten Vorgang des Verschraubens für ein abstraktes Bauteil das als Boltable bezeichnet und über die vorhandenen Bohrlöcher und Gewinde definiert ist. Softwaretechnisch kann ein Boltable als Schnittstelle realisiert werden, die Methoden für den Zugriff auf die Bohrlöcher (Hole) enthält. Ein konkretes Boltable ist bspw. das Bauteil der Factory 2020. Der AbstractFasteningSystem stellt die Implementierung für einen prozesszentrierten Dienst bereit, da er sich auf einen weiteren Service, den BoltingService, abstützt und dementsprechend keine Aktuatoren der Roboterzelle direkt steuert.

Der BoltingService dagegen wird durch das BoltingSystem mithilfe eines Manipulators und eines an dessen Flansch montiertem Schraubsystems realisiert (vgl. Abb. 9.13). Das BoltingSystem ist dadurch als aufgabenzentrierter Dienst zu kategorisieren und stützt sich zur Beschreibung der Handhabungsfunktionen nur auf das Bohrloch (Hole) und die Schraube (Bolt) ab. Daneben kennt das BoltingSystem ein Schraubenmagazin (BoltDepot), aus dem die Schrauben dem Prozess zugeführt werden. Durch die Verwen-

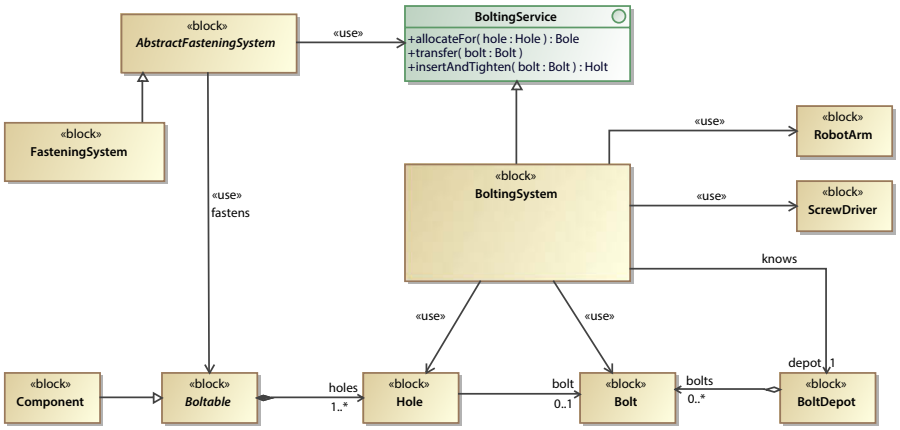


Abbildung 9.13: Das FASTENINGSYSTEM kann abstrakt als Schraubsystem werden und sich auf einen BOLTINGSERVICE abstützen. Dieser wiederum kann von einem BOLTINGSYSTEM mit einem Roboterarm und einem elektrischen Schraubsystem implementiert werden

dung eines Roboters werden die Handhabungs- und Fertigungsfunktionen auf Bewegungen und Aktionen des Schraubsystems abgebildet. Die Umsetzung wird ebenfalls in Abschnitt 9.4.2 beschrieben.

Zusammenfassend lässt sich feststellen, dass sich Handhabungs- und Fertigungsfunktionen durch eine geeignete Abstraktion der Handhabungsobjekte generisch implementieren lassen. Durch Anpassungen und Konfiguration lässt sich eine abstrakte Serviceimplementierung einfach wiederverwenden. Damit können diese Funktionen nicht nur in der aktuellen Roboterzelle verwendet werden, sondern als wiederverwendbare Implementierung eines Dienstes in einer Vielzahl von Roboterzellen eingesetzt werden. Die Wiederverwendung ist ein entscheidender Schritt, um die Programmierkosten von Roboterzellen zu senken.

9.4.2 *Aufteilung & Delegation*

Für eine Auswahl an Aktuatoren kann ein aufgabenzentrierter Dienst sowohl Handhabungs- als auch Fertigungsfunktionen implementieren. Dabei liegt das Augenmerk auf dem Endeffektor bzw. dem Werkzeug. Der Roboter ist Mittel zum Zweck, d. h. er bewegt oder positioniert den Endeffektor oder die Handhabungsobjekte. Obwohl jede Handhabungsfunktion unterschiedlich realisiert werden kann, ist die Implementierung für ein bestimmtes Werkzeug oder eine bestimmte Werkzeugart sehr ähnlich. Daher kann der generelle Ablauf eines aufgabenzentrierten Dienstes generisch implementiert werden. Er stützt sich dabei auf abstrakte Fähigkeiten bzw. *Skills* (vgl. Abschn. 7.2) der Werkzeuge ab, die ebenfalls durch Services bereitgestellt werden. Details dazu werden in Kapitel 10 erläutert.

Die Abbildung der Handhabungsfunktion *Zuteilen* auf die abstrakten Fähigkeiten eines Greifers ist in Abbildung 9.14 dargestellt. Dabei sind neben dem Greifer als Teil der Serviceimplementierung auch das zuzuteilende Objekt und der geordneter Speicher involviert. In Abbildung 9.14 ist das Objekt als Bauteil (PART) der Factory 2020 und der geordneter Speicher als Werkstückträger (CARRIER) dargestellt. Die Implementierung des Dienstes kann jedoch mit den generischen Objekten erfolgen.

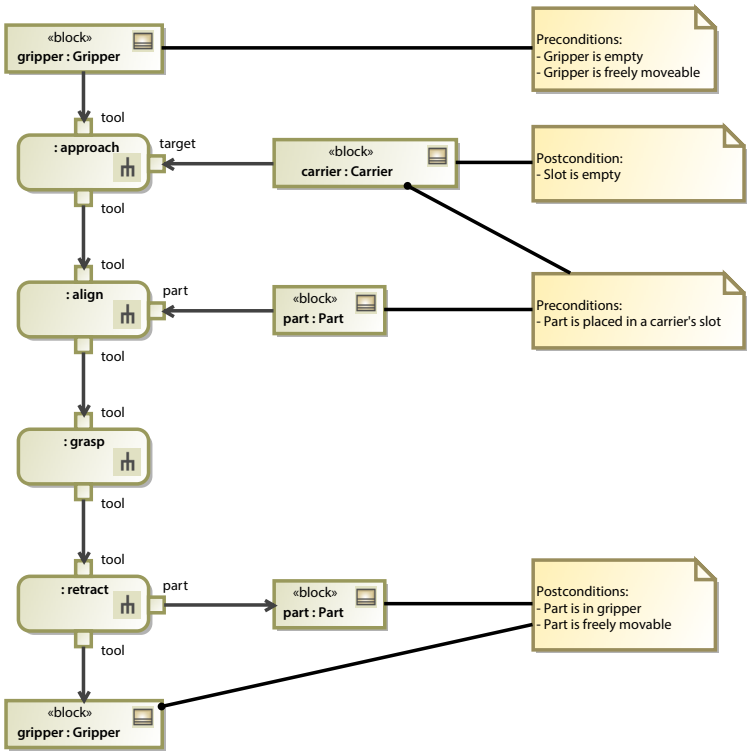


Abbildung 9.14: Abbildung der Handhabungsfunktion *Zuteilen* auf die abstrakten Fähigkeiten eines Greifers, der an einen Manipulator montiert ist.

Der Ablauf der Handhabungsfunktion *Zuteilen* als Folge abstrakter Fähigkeiten eines Greifer gliedert sich wie folgt:

1. *approach*: Der Greifer (GRIPPER) wird zuerst in der Nähe bzw. oberhalb des Werkstückträgers (CARRIER) positioniert.
2. *align*: Anschließend wird der Greifer am entsprechenden Werkstück (PART) ausgerichtet und so in eine Position gebracht, von der aus der Greifvorgang gestartet werden kann.
3. *grasp*: Schließlich wird der Greifer an die endgültige Position gebracht und greift das Werkstück. Das Werkstück ist aufgrund der vorherigen Aktion bereits bekannt.
4. *retract*: Das gegriffene Werkstück wird aus dem Werkstückträger an eine Position geführt, von der aus eine Transferbewegung starten kann. Das Werkstück ist aufgrund der vorherigen Aktion bereits bekannt.

Die Vor- und Nachbedingungen der Abfolge sind in Abbildung 9.14 durch Notizen dargestellt. Durch die Verwendung eines Werkzeugtyps zur konkreten Umsetzung der Handhabungsfunktion können sowohl Vor- als auch Nachbedingungen im Vergleich zu Abbildung 9.7 konkreter definiert werden. Die allgemeinen, bereits definierten Vor- und Nachbedingungen bleiben jedoch erhalten.

Die abstrakten Fähigkeiten eines Greifers stehen nur in Verbindung mit einem Manipulator zur Verfügung, da dieser den Greifer bewegen kann und nur so diese Fähigkeiten umgesetzt werden können. Zur Spezifikation dieser Aufgaben ist der Roboter aber nicht von Belang. Er muss einzig zur Verfügung stehen und der Greifer muss an seinem Flansch montiert sein.

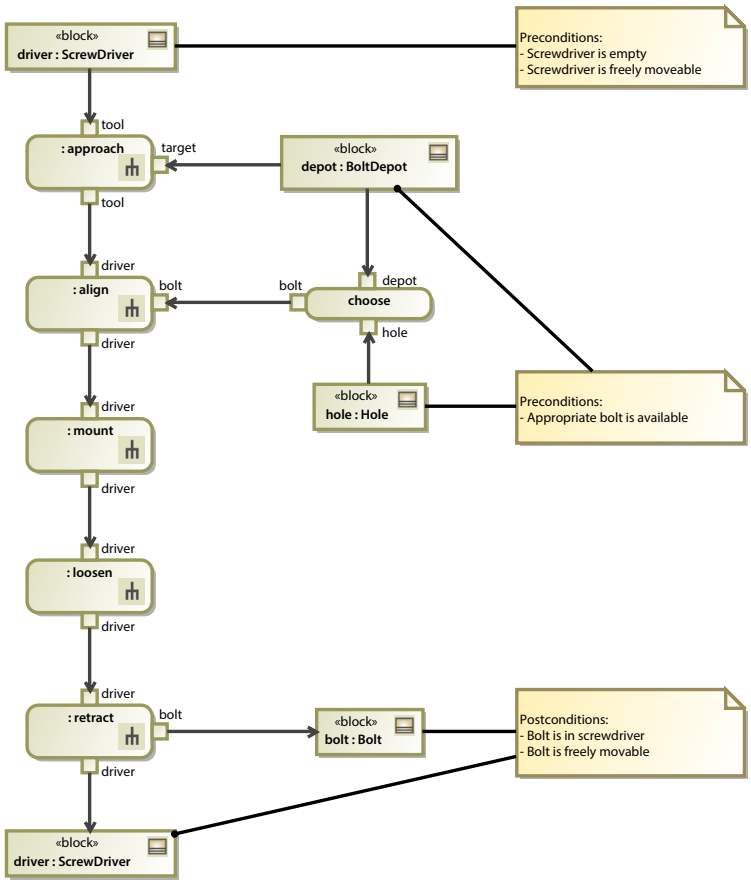


Abbildung 9.15: Abbildung der Handhabungsfunktion *Zuteilen* auf die abstrakten Fähigkeiten eines elektrischen Schraubsystems, der an einen Manipulator montiert ist.

Die konkrete Umsetzung der oben beschriebenen Fähigkeiten übernehmen die in Kapitel 7 definierten *Skill-Level Services*. Deren Implementierung wird in Kapitel 10 vorgestellt.

Das *Zuteilen* einer Schraube für den Fertigungsprozess entspricht in etwa der gleichen Handhabungsfunktion wie das *Zuteilen* eines Bauteils. Allerdings wird diese Funktion mit einem anderen Werkzeug und dadurch mit einem anderen aufgabenzentrierten Dienst implementiert. Daher unterscheidet sich auch die Abbildung beider Handhabungsfunktionen. Die Abbildung der Handhabungsfunktion *Zuteilen* auf die Fähigkeiten eines elektrischen Schraubsystems ist in Abbildung 9.15 dargestellt. Obwohl drei Fähigkeiten identisch sind, werden insgesamt mehr und unterschiedliche Fähigkeiten verwendet.

Der Ablauf der Handhabungsfunktion *Zuteilen* als Folge abstrakter Fähigkeiten eines elektrischen Schraubsystems gliedert sich wie folgt:

1. *approach*: Das Schraubsystem (ScrewDriver) wird zuerst in der Nähe bzw. oberhalb des Schraubenmagazins (BoltDepot) positioniert.

2. *choose*: Auf Basis des Bohrlochs bzw. des Gewindes (Hole) wird eine passende Schraube (Bolt) des Magazins ausgewählt. Hierbei findet keine Werkzeugaktion oder Roboterbewegung statt. Vielmehr handelt es sich um eine spezifische Funktion des Schraubvorgangs.

3. *align*: Das Schraubsystem wird nun an der ausgewählten Schraube ausgerichtet und in eine Position gebracht, von der aus der nächste Vorgang gestartet werden kann.
4. *mount*: Anschließend wird das Bit des Schraubsystems in die Schraube geführt.
5. *loosen*: Die Schraube wird aus dem Schraubenmagazin gelöst und befindet sich danach sicher am Bit des Schraubsystems.
6. *retract*: Die Schraube wird an eine Position oberhalb des Schraubenmagazins gebracht, von der aus eine Transferbewegung starten kann.

Die Vor- und Nachbedingungen der Abfolge sind ebenfalls in Form von Notizen dargestellt (vgl. Abb. 9.15). Wie bereits in dem vorherigen Beispiel werden sowohl Vor- als auch Nachbedingungen durch die Verwendung eines Werkzeugtyps, d. h. des Schraubsystems, konkreter definiert.

Die beiden Beispiele zeigen, dass die lösungsneutralen Handhabungs- und Fertigungsfunktionen durch einen aufgabenzentrierten Dienst unterschiedlich implementiert werden können. Das bedeutet gleichzeitig, dass erst durch einen aufgabenzentrierten Dienst die Ausgestaltung der Roboterzelle bei der Realisierung des Fertigungsprozesses relevant wird. Da jedoch das Abstraktionsniveau der Werkzeugfähigkeiten noch sehr hoch ist, kann der aufgabenzentrierte Dienst für eine Vielzahl abstrakter Bauteile wiederverwendet werden. Erst bei der konkreten Umsetzung der Werkzeugfähigkeiten durch weitere Services zeigt sich, ob und wie die Fähigkeiten angewendet werden.

9.4.3 *Realisierung als Zustandsmaschinen*

Intern kann ein prozess- oder aufgabenzentrierter Dienst grafisch als Zustandsmaschine [113] modelliert werden. Der Aufruf von Handhabungs- und Fertigungsfunktionen, die der Service anbietet, stellt ein Ereignis dar, auf das der Dienst – in Abhängigkeit von seinem internen Zustand – reagieren muss. So kann insbesondere bei mehrstufigen Funktionen, die eine Ausführungsreihenfolge bedingen, sichergestellt werden, dass diese nur in der richtigen Reihenfolge aufgerufen werden können. Zustandsübergangsdiagramme oder *State Machines* sind ebenfalls Bestandteil der SysML und eignen sich daher zur Modellierung des internen Verhalten prozess- oder aufgabenzentrierter Dienste.

Es bietet sich an, für jede Handhabungs- oder Fertigungsfunktion mindestens zwei Zustände vorzusehen. Dabei beschreibt ein Zustand die Durchführung der Funktion und der zweite Zustand das erfolgreiche Beenden der Funktion. Von diesem Zustand aus kann die folgende Funktion gestartet werden. Zudem ist es möglich weitere Zustände vorzusehen. Insbesondere sind nach dem Durchführen einer Handhabungs- oder Fertigungsfunktion mehrere resultierende Zustände möglich, die zum Einen einen erfolgreichen Abschluss der Funktion und zum Anderen Fehlerfälle repräsentieren. Dementsprechend sind auch unterschiedliche Reaktionen möglich.

Ein einfaches Beispiel für die Modellierung des internen Verhaltens einer Serviceimplementierung ist in Abbildung 9.16 dargestellt. Das Zustandsübergangsdiagramm zeigt das interne Verhalten eines SupplySystems bzw. dessen Service. Die Ausführung der beiden Handhabungsfunktionen *Zuteilen* und *Weitergeben* sind als separate Zustände *Allocating* bzw. *Transfering* modelliert. Nachdem ein Bauteil über den Service zugeteilt wurde, befindet

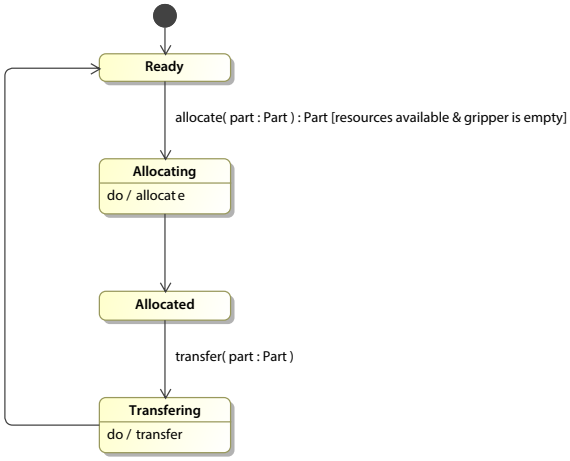


Abbildung 9.16: Das SUPPLYSYSTEM kann intern als Zustandsmaschine modelliert und später implementiert werden.

sich die Implementierung im Zustand *Allocated*, d. h. es kann zwingend nur ein *Weitergeben* folgen. Vor einem Ablauf befindet sich die Zustandsmaschine initial in dem Zustand *Ready*. Als *Trigger* für eine Transition dienen die Dienstaufrufe oder das Beenden der jeweiligen Funktion. Auf Fehlerfälle wurde in der Darstellung verzichtet.

Diese oben beschriebenen Zustände und Transitionen sind unabhängig von der konkreten Abbildung auf ein Werkzeug. Jedoch kann die Zustandsmaschine über Bedingungen (engl.: *Guards*) und Unterzustandsmaschinen spezialisiert werden. Bedingungen bieten sich an, um die Vorbedingungen von Handhabungsfunktionen zu spezifizieren. So kann sichergestellt werden, dass nur wenn die Bedingung erfüllt ist, die Transition geschaltet und die Handhabungsfunktionen ausgeführt wird. Auch können leicht die Zustände identifiziert werden, an denen die entsprechenden Ressourcen, d. h. die notwendigen Aktuatoren der Roboterzelle, reserviert bzw. freigegeben werden.

Eine Unterzustandsmaschine kann verwendet werden, um während der Ausführung einer Handhabungsfunktionen die verwendeten Service richtig zu orchestrieren. Im Falle eines aufgabenzentrierten Dienstes kann so die Abfolge der einzelnen Fähigkeiten des oder der verwendeten Werkzeuge modelliert werden. Die Unterzustandsmaschine stellt die korrekte Ausführungsreihenfolge sicher. Für Abbildung 9.16 bedeutet dies, dass der Zustand *Allocating* durch eine Unterzustandsmaschine repräsentiert wird. Die in Abbildung 9.14 dargestellten Fähigkeiten werden als Zustände der Unterzustandsmaschine modelliert und aufgerufen.

Abbildung 9.17 zeigt ein weiteres Beispiel für die Modellierung des internen Verhaltens einer Serviceimplementierung. Das BOLTINGSYSTEM ist dabei als Zustandsmaschine modelliert. Eine Besonderheit dabei ist, dass die komplexe Handhabungsfunktion des Verschrauben durch drei separate Zustände modelliert wurde, die jeweils eine Fähigkeit des elektrischen Schraubsystems repräsentieren und aufrufen. Die weiteren Zustände weisen Ähnlichkeiten zu dem oben beschriebenen SUPPLYSYSTEM und das Handhabungsfunktionen *Zuteilen* und *Weitergeben* auf.

Die Modellierung einer Serviceimplementierung als Zustandsmaschine impliziert, dass der Service zustandsbehaftet ist. Jedoch werden Services oft als zustandslos charakterisiert [80]. Dies ist jedoch kein zwingendes Merkmal eines Services. Insbesondere in einer Roboterzelle ist es angebracht, den

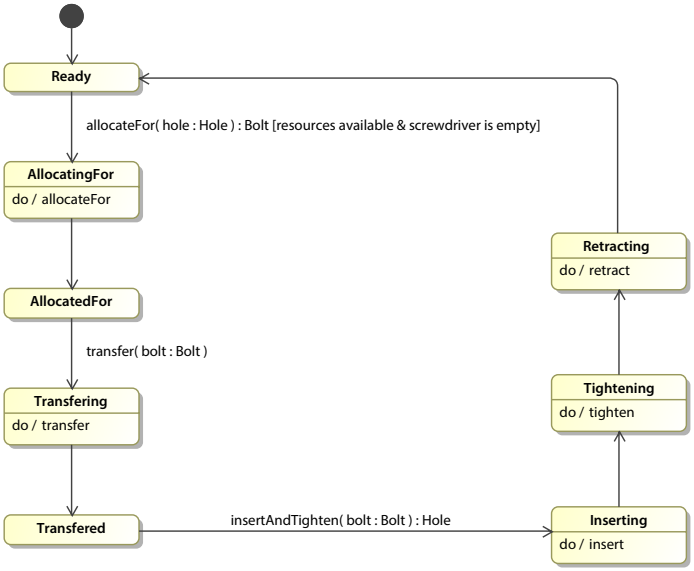


Abbildung 9.17: Das BoltingSystem kann intern als Zustandsmaschine modelliert und später implementiert werden.

aktuellen Zustand des Fertigungsprozesses bzw. der Systeme, die ihn umsetzen, abzubilden und zu beobachten. Jeder prozess- oder aufgabenzentrierte Dienst wird für einen gezielten Einsatz implementiert, konfiguriert und dann instanziiert. Das bedeutet auch, dass eine Serviceimplementierung mehrmals mit einer unterschiedlichen Konfiguration instanziiert wird. Der Service hat eine klar definierte Aufgabe in der Roboterzelle, die er erfüllen muss.

Eine Serviceimplementierung kann nicht nur als Zustandsmaschine modelliert werden, sondern durch grafische Programmierung auch direkt implementiert. In Abschnitt 11.5 wird bspw. vorgestellt, wie das Verhalten eines Dienstes grafisch als Zustandsmaschine spezifiziert wird und diese grafische Spezifikation direkt interpretiert werden kann. Dadurch kann während der Ausführung eine direkte Visualisierung des Services angezeigt und überwacht werden.

9.5 VERWANDTE ARBEITEN

Einen Ansatz, der ein Cyber-Physical Production System (CPPS) verwendet, wird bei Michniewicz und Reinhart [181] vorgestellt. Dabei wird eine cyberphysische Roboterzelle beschrieben, die über eine zentrale Steuereinheit verfügt und Cyber-Physical Devices (CPDs) steuert. Die zentrale Steuereinheit verfügt über ein virtuelles Abbild aller CPDs. Dadurch können die möglichen Konfigurationen der Roboterzelle ermittelt werden, um einen Fertigungsprozess abzubilden. Jede CPD verfügt zudem über eine Menge an Fähigkeiten, sogenannten *Functional Primitives*. Die Fähigkeiten zweier CPDs können zu erweiterten Fähigkeiten, sogenannten *Advanced Primitives*, kombiniert werden. Immer wiederkehrende Abfolgen von Fähigkeiten ergeben *Tasks*, die von einem Cyber-Physical Product (CPP) zur Fertigung seiner selbst in der cyberphysischen Roboterzelle angefordert werden können. Das von Michniewicz und Reinhart [181] beschriebene Konzept entspricht in Teilen den Ergebnissen dieser Arbeit, wobei keine serviceorientierte Modellierung und Implementierung gewählt wurde. Da Michniewicz und Reinhart [181]

ihr Konzept noch nicht (prototypisch) umgesetzt haben, sind keine weiteren Ergebnisse oder eine Evaluierung vorhanden.

In der Literatur sind verschiedene Forschungsansätze zu finden, in denen Automatisierungssoftware von Roboterzellen vollständig objektorientiert strukturiert und implementiert wird. Becker und Pereira [28] schlagen mit SIMOO-RT ein echtzeitfähiges, objektorientiertes Framework für industrielle Automatisierungssysteme vor. Es basiert auf dem Konzept verteilter, *aktiver* Objekte, wobei aktiv bedeutet, dass jedes Objekt einen eigenen Ausführungsstrang besitzt. Dadurch können innerhalb eines aktiven Objekts periodische Aufgaben mit harten Zeitschranken ausgeführt werden. Eine Besonderheit ist, dass explizit Fehlerroutinen definiert werden können, falls Zeitschranken nicht eingehalten werden. Da die Objekte verteilt sind, findet die Kommunikation über Remote Method Invocation (RMI) oder Real-time CORBA [238] statt. Die Granularität der Objekte in SIMOO-RT entspricht den mechatronischen Geräten und Aktuatoren der *Robotics API*. Das heißt, dass keine höheren Abstraktionsebenen modelliert werden, die eine wiederkehrende Zusammenstellung dieser Objekte zu komplexeren Subsystemen ermöglichen. Auch werden die Besonderheiten von Roboterzellen nicht betrachtet.

Schäfer und Lopez [233] schlagen ein objektorientiertes Modell für die Integration von Roboterzelle in flexible Fertigungssysteme vor. Dabei betrachten sie verschiedene Abstraktionsebenen der Automatisierungspyramide. Auf unterster Ebene befinden sich die Betriebsmittel bzw. die für die Fertigung verfügbaren Ressourcen. Darüber befindet sich die Zellebene, die die einzelnen Betriebsmittel integriert. Die einzelnen Zellen werden auf der Fabrikebene integriert. Analog zur *Robotics API*, werden in diesem Ansatz vor allem die unteren beiden Ebenen objektorientiert modelliert. Die Kommunikation zwischen den Ebenen findet streng hierarchisch statt. Neben diesen drei vorgestellten Ebenen werden keine weiteren Abstraktionsebenen definiert, die eine Modellierung erleichtern und die Wiederverwendung von Programmen erhöhen würden.

Der Einsatz serviceorientierter Architekturen (SOA) in eingebetteten Systemen und damit in der Automation wurde ab 2003 in mehreren europäischen Projekten untersucht. Das SIRENA Projekt [38] hat gezeigt, dass serviceorientierte Architekturen nicht nur für webbasierte Anwendungen geeignet sind, sondern auch im Bereich der eingebetteten Systemen eingesetzt werden können. Jammes und Smit [141] zeigen anhand einer Fallstudie den Einsatz in der industriellen Automation. Sie betrachten die einzelnen Dienste, die ein Gerät anbieten kann, und orchestrieren diese zu komplexeren Systemen bzw. Diensten im Rahmen einer *device-level* SOA. Daher wurde auch eine erste Implementierung für das Devices Profile for Web Services (DPWS) [76] entwickelt. Im SOCRADES [57] Projekt wurden die Ergebnisse von SIRENA weiterentwickelt und untersucht, wie sich eine *device-level* SOA in eine Unternehmensinfrastruktur integrieren lässt. Diese grundlegenden Ansätze wurden jedoch nie auf (flexible) Roboterzellen übertragen. Daher wird auch die interne, serviceorientierte Modellierung der unterschiedlichen Abstraktionsebenen nicht betrachtet.

Barbosa und Leitão [24] schlagen einen Ansatz vor, in dem sie Serviceorientierung mit Multi-Agentensystemen [164] und holonischen Fertigungssystemen (vgl. [263]) verbinden. Ein holonisches Fertigungssystem zeichnet sich dadurch aus, dass es aus einer Menge von autonomen aber kooperativen Einheiten – sogenannten Holonen (z. B. eine Fertigungsmaschine oder ein Roboter) – besteht und diese durch eine hierarchische Anordnung das

Gesamtsystem bilden. Durch den Einsatz eines Agentensystems wird jedes Holon mit einem Softwareagenten ausgestattet. Der Agent verfolgt dabei seine eigenen Ziele, arbeitet aber bei Bedarf auch mit anderen Agenten zusammen. Dadurch entsteht eine flexibles und anpassbares Fertigungssystem. Den Einsatz für eine serviceorientierter Architektur sehen Barbosa und Leitão [24] bei der Kommunikation zwischen den Agenten, um die Interoperabilität und vertikale IT-Integration zu erhöhen. Somit wird jeder Agent mit Diensten versehen, die er anderen Agenten anbietet. Die Kommunikation zwischen Agenten findet über Services statt und abstrahiert von der internen Realisierung des Agenten. Barbosa und Leitão [24] zeigen damit einen interessanten Ansatz für die Strukturierung einer intelligenten Fabrik auf (vgl. Kap. 7). Sie betrachten jedoch nicht die interne Strukturierung einer Roboterzelle, die in ihrem Ansatz ein einziges Holon wäre.

Naumann et al. [188] stellen eine Architektur für Roboterzellen vor, die *Plug-and-Produce* unterstützen soll. Der Grundgedanke ist dabei ähnlich zu *Plug-and-Play*, d. h. die Roboterzelle wird aus einzelnen Bestandteilen *zusammengesteckt* und soll dann automatisch anfangen können, zu produzieren. Sie unterscheiden dabei Ebenen von *Plug-and-Produce*. Auf der untersten Ebene müssen die einzelnen Geräte mit einer zentralen Steuerung verbunden werden und automatisch kommunizieren können. Hier wurden verschiedene Kommunikationsprotokolle (z. B. UPnP, EDDLoder XIRP) evaluiert [189]. Auf einer darüber liegenden Ebene müssen die Geräte mit der Steuerung Konfigurationsdaten und Einstellungen automatisch austauschen. Der Anwender soll dabei nicht eingreifen müssen.

Die oberste Ebene wird als *Application Plug-and-Produce* bezeichnet und stellt dem Benutzer Services bereit, die eine prozessorientierte Funktionalität als „process command“ [188] bereitstellen. Ein solches Kommando ist bspw. *DrillHole(x=50, y=90, d=30)* und weist eine höhere Abstraktion auf als ein „device command“ [188] (z. B. *MoveTo(x=50, y=90, z=70)* oder *SetPort(port=20)*. Durch ein *Interconnector Module* werden die Fähigkeiten der einzelnen Geräte zu Prozessen verknüpft und dem Anwender angeboten. Die Prozesse sind dabei in einer Bibliothek gespeichert und mithilfe von State Charts modelliert. Bei Naumann et al. [188] liegt demnach der Fokus auf dem Bereitstellen anwenderfreundlicher Befehle. Diese haben jedoch Parameter, die auf Informationen über ein Bauteil verzichten. Diese Informationen muss der Benutzer manuell spezifizieren.

Nilsson und Bengel [193] betrachten auch eine serviceorientierte Realisierung von Roboterzellen. Dabei liegt ihr Fokus auf der Integration der einzelnen Geräte (vgl. *device-level* SOA) und nicht einer weitergehenden internen Modellierung oder Strukturierung. Sie stellen die Vorteile bzgl. Flexibilität und Modularität serviceorientierter Roboterzellen für kleine und mittlere Unternehmen heraus. Sie geben allerdings zu Bedenken, dass aktuell keine echtzeitfähige Middleware existiert, um serviceorientierte Roboterzellen entsprechend umzusetzen.

Veiga [265] hat sich in seiner Dissertation mit serviceorientierten Architekturen für Roboterzellen beschäftigt. Seine Fallstudie ist dabei eine Pick-and-Place-Applikation [268]: Auf einem von einer SPS gesteuerten Fließband werden über ein Kamerasystem die Bauteile erkannt und deren Position berechnet. Daraufhin nimmt ein Industrieroboter von ABB die Teile auf und legt sie in eine Kiste. Zudem wurden unterschiedliche Mensch-Maschine-Schnittstellen entwickelt. Sein Schwerpunkt liegt jedoch in der Evaluierung unterschiedlicher SOA-Technologien zum Einsatz in Roboterzellen für kleine und mittlere Unternehmen. Dabei beschäftigt er sich hauptsäch-

lich mit der Anwendung der serviceorientierten Technologien UPnP [4] und DSSP [192] für die Kommunikation innerhalb der Roboterzelle [267]. Dabei wird für jedes Gerät (Kamera, Fließband, Roboter) ein Service implementiert, der die proprietäre Kommunikation zu dem Gerät herstellt. Über einen zentralen *ControlPoint* werden die Services orchestriert (vgl. [267]). Dementsprechend verfolgt Veiga [265] eine *device-level* SOA, wie sie auch schon bei Jammes und Smit [141] vorgeschlagen wurde. Er verzichtet im Gegensatz zu dem in dieser Arbeit vorgeschlagenen Ansatz auf eine weitergehende serviceorientierte Modellierung der Roboterzelle. Auch werden die einzelnen Element oder Bauteile der Roboterzelle nicht weiter modelliert. Die angebotenen Methoden der Geräte bzw. ihrer Services sind jedoch auf die Bedürfnisse der konkreten Roboterzelle angepasst und daher auf der Ebene aufgabenzentrierter Dienste (vgl. Abschn. 7.2). Diese Granularität der Services wird als geeignet angesehen, um die Roboterzelle aus Services zusammenzustellen und den Fertigungsprozess zu orchestrieren [266]. Veiga verwendet dazu ebenfalls Zustandsmaschinen [113] und entwickelt einen eigenen Formalismus, der ähnlich zu SCXML [26] ist. Seine Intention ist, einen einfachen Formalismus zur Orchestrierung von Services für kleine und mittlere Unternehmen zur Verfügung zu stellen.

Zusammenfassend verfolgt Veiga [265] bei den *device-level* Service für den Roboter einen interessanten Ansatz. Er sieht die Roboterprogrammiersprachen als unabdingbar für die Industrierobotik und räumt ihnen eine besonderen Stellenwert ein. Durch Annotationen ergänzt er ein Roboterprogramm, um die einzelnen Services, deren Zustandsvariablen und Methoden zu identifizieren [266]. Auf Basis dieser Annotationen (d. h. Quellcodekommentare in einem speziellen Format) werden die Services erzeugt und können so in die serviceorientierte Struktur der Roboterzelle eingebunden werden. Damit können relativ einfach und schnell auf Basis der Robotersteuerung neue Services generiert werden. Innerhalb eines solchen Services sind die Roboterbewegungen unflexibel und können in der Regel nicht wiederverwendet werden. Durch den Ansatz will Veiga erreichen, dass man weiterhin Roboterprogrammiersprachen zur Programmierung des einzelnen Roboters einsetzen kann, aber Services als Kommunikationsplattform verwendet. Damit ist der Ansatz fundamental unterschiedlich zu den in dieser Arbeit und im Forschungsprojekt *SoftRobot* verfolgten Zielen.

Analog zu Veiga [265] wurde vom Autor dieser Dissertation eine serviceorientierte Architektur für die Fertigung von carbonfaserverstärktem Kunststoff (CFK) evaluiert (vgl. [14, 15]). Dazu wurden in dem Forschungsprojekt *CFK-Tex* [227] zwei Endeffektoren für das automatisierte Handhaben bzw. Konfektionieren von Faserverbundhalbzeugen entwickelt. Die beiden Endeffektoren wurden in jeweils eine eigene Roboterzelle integriert. Die einzelnen Systeme der Zelle, d. h. der Endeffektor, ein Industrieroboter, ein Lagersystem und ein Cuttertisch, wurden über einen Web Service [42] angesteuert. Durch die Windows Communication Foundation (WCF), eine serviceorientierte Kommunikationsplattform für verteilte Anwendungen, wurden die einzelnen Systeme über ihre Web Services integriert. Der Arbeitsablauf der Zelle wurde folglich als eine Orchestrierung der Services realisiert. Allerdings ist in diesem Ansatz die Robotersteuerung unflexibel. Es können zwar die Positionen für das Handhaben der Faserverbundhalbzeuge flexibel spezifiziert werden, aber die meisten Bewegungen und Abläufe sind – analog zu Veiga [265] – als Teil des Services fest programmiert. Einen ähnlichen Ansatz verfolgen auch Valera et al. [262]. Sie verwenden UPnP, um die Elemente

einer Roboterzelle zu integrieren. Für die Programmierung haben sie eine eigene graphische Sprache entwickelt.

Auch die Systems Modeling Language (SysML) wird vereinzelt zur Modellierung von Robotersystemen verwendet. So stellen Huckaby und Christensen [130] die SysML als eine geeignete Modellierungssprache für die Robotik vor. Durch ihre graphische und deskriptive Art eignet sie sich laut den Autoren gut, um komplexe Systeme akkurat zu beschreiben. Gleichzeitig kann sie verwendet werden, um Informationen über Systeme zwischen verschiedenen Stakeholdern zu kommunizieren. Zudem ist die SysML eine standardisierte Sprache mit einer formalen Semantik, was zu einer großen Verbreitung in der Robotik führen kann. Weitere Vorteile sind einfache Verifikation und Validierung von Robotersystemen durch die formale Semantik und die Möglichkeit einer automatischen Quellcodegenerierung. Huckaby und Christensen stellen beispielhaft dar, wie die SysML verwendet werden kann, um robotergestützte Montageaufgaben zu modellieren vgl. [132]. Dabei werden die notwendigen Fähigkeiten eines Robotersystems in einem Blockdefinitionsdiagramm modelliert. Die Montageaufgabe wird als Sequenzdiagramm modelliert, wobei der Roboter, das Werkstückmagazin und eine Halteeinrichtung Lebenslinien darstellen. Durch Nachrichten kann der Roboter Fähigkeiten beim Werkstückmagazin oder der Halteeinrichtung anwenden. Der Ansatz wurde anhand des *Cranfield Assembly Benchmarks* [128] und der Montage von hölzernen Modellflugzeugen evaluiert. Insgesamt lässt sich festhalten, dass die Modellierung der Montageaufgaben sehr einfach gehalten ist. Insbesondere der Aufbau der Roboterzelle wird nur sehr abstrakt durch die Lebenslinien der Sequenzdiagramme abgebildet. Zudem wird nicht erklärt, wie die einzelnen Fähigkeiten parametrisiert und ausgeführt werden. Auch die zu Beginn vorgestellten Vorteil der SysML werden bei dem Beispiel nicht sichtbar.

Ohara et al. [199] verwenden ebenfalls SysML, um die Softwaremodule eines mobilen Roboters für *Pick-and-Place*-Aufgaben zu modellieren. Da in diesem Beispiel hauptsächlich Softwarekomponenten modelliert wurden, wurden die Besonderheiten von SysML kaum verwendet. Nur für einen Motorregler wurden der kontinuierliche Datenfluss zwischen Komponenten explizit modelliert. Rahman et al. [225] stellen einen ähnlichen Ansatz für einen mobilen Roboter vor. Ein weiteres Beispiel für die Verwendung von SysML in der Robotik liefern Chhaniyara et al. [58]. Dabei wird SysML für ein robotisches Weltraumsystem zur Wartung von Satelliten verwendet. Eine weitergehende Modellierung und Strukturierung von Roboterzellen wurde – soweit bekannt – in der Forschung bisher noch nicht betrachtet.

10 BAUSTEINE SERVICEORIENTIERTER ROBOTERZELLEN

Bisher wurde die serviceorientierte Modellierung von Roboterzellen auf einem sehr hohen Abstraktionsniveau vorgestellt. Auf höchster Ebene werden die Automatisierungsprozesse der Roboterzelle beschrieben. Dabei werden nur die Handhabungsobjekte und Bauteile betrachtet. Jeder Prozessschritt stellt eine Handhabungs- oder Fertigungsfunktion dar, die das Bauteil räumlich oder physisch verändert. Bereitgestellt werden die notwendigen Handhabungs- oder Fertigungsfunktion von aufgabenzentrierten Diensten bzw. *Task-level Services*. Innerhalb eines solchen Dienstes findet eine erste Verknüpfung zwischen der Handhabungs- bzw. Fertigungsfunktion und dem ausführenden Werkzeug statt. Dieses Werkzeug ist dabei in der Regel als Endeffektor am Flansch des Roboters montiert und kann von diesem so bewegt werden.

Diese Tatsache wird jedoch innerhalb eines aufgabenzentrierten Dienstes nicht berücksichtigt. Stattdessen werden die Abläufe und damit die Realisierung der Handhabungs- bzw. Fertigungsfunktionen durch abstrakte Fähigkeiten des Werkzeugs beschrieben. Die konkrete Umsetzung dieser Fähigkeit wird an Services einer niedrigeren Abstraktionsebene ausgelagert. Das bedeutet, dass erst innerhalb eines fähigkeitenzentrierten Dienstes bzw. eines *Skill-level Services* die konkrete Umsetzung einer Fähigkeit implementiert ist. Dort findet erst die Verknüpfung zwischen Roboter und Werkzeug und folglich deren Programmierung statt. *Skill-level Services* können damit als grundlegende Bausteine serviceorientierter Roboterzellen bezeichnet werden. Infolgedessen leistet die Arbeit einen Beitrag zur Trennung von Ablauf- und Bewegungsprogrammierung.

In Abschnitt 10.1 wird die Idee und die Funktionsweise fähigkeitenzentrierter Dienste vorgestellt. Dabei wird insbesondere auch das Zusammenspiel mehrerer Dienste anhand einer einfachen Pick-and-Place-Aufgabe beschrieben. Anschließend wird die Idee von Strategien in Abschnitt 10.2 erläutert. Bei Strategien handelt es sich um Programmfragmente, die eine Lösungsmöglichkeit für eine Fähigkeit darstellen. Während in Abschnitt 10.3 Beispiele für fähigkeitenzentrierte Dienste vorgestellt werden, werden unterschiedliche Strategien für die Fähigkeit des Greifens in Abschnitt 10.4 erläutert. Das Kapitel endet in Abschnitt 10.5 mit einer Übersicht verwandter Arbeiten, von denen die vorgestellten Ideen und Konzepte abgegrenzt werden. Einige dieser Ideen und Konzepte wurden erstmals in [121] vorgestellt und in [120] detailliert dargestellt.

10.1 SKILLS: FÄHIGKEITEN ALS SERVICE

Als Fähigkeit oder *Skill* wurde in Kapitel 7 eine Tätigkeit definiert, die ein Roboter und ein Werkzeug gemeinsam ausführen können. Dabei liegt der Fokus auf dem Werkzeug, während der Roboter ausschließlich für die Bewegung, d. h. den Transport von Werkzeug oder Bauteil, zuständig ist. Obwohl das Werkzeug die Tätigkeit nicht selbstständig ausführen kann, ist es der Mittelpunkt einer Tätigkeit. Der Ablauf einer Roboterzelle lässt sich damit deklarativ und schrittweise beschreiben, ohne dass Roboterbewegungen im Detail betrachtet werden müssen. Vom Roboter und seinen Bewegungen

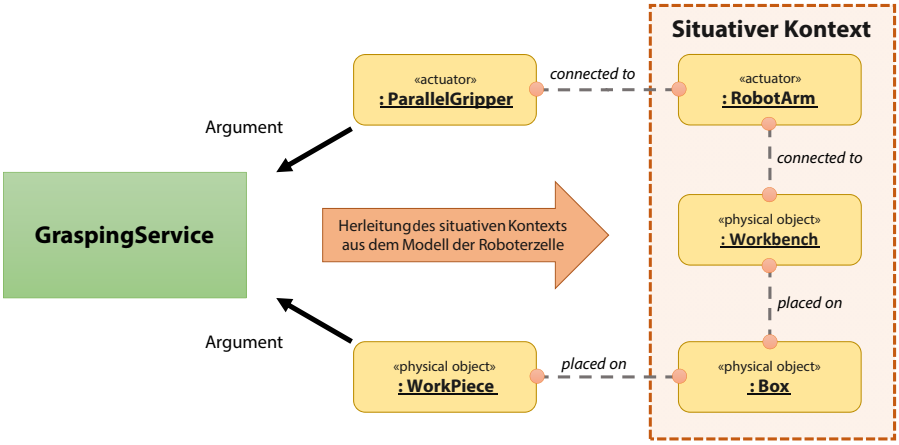


Abbildung 10.1: Durch die Eingabeparameter (z. B. den Greifer und das zu greifende Werkstück) leitet der Service den situativen Kontext her. Er kann dabei erkennen, ob und an welchem Roboter der Greifer montiert ist bzw. wo sich das Werkstück befindet (z. B. in einer Kiste). Daraufhin kann er entscheiden, durch welche Lösungsstrategie die Fähigkeit am Besten umgesetzt wird.

kann auf dieser Ebene noch abstrahiert werden. Dadurch wird zwar eine Trennung zwischen der Ablaufsteuerung und der eigentlichen Roboterprogrammierung erzwungen. Es erfolgt im Vergleich zu einer klassischen Robotersteuerung aber kein Bruch (vgl. Kap. 3), da die Informationen und damit das Wissen über die Fertigung durchgehend von oben nach unten propagiert werden.

Fähigkeiten werden in einer serviceorientierten Roboterzelle als Dienst bereitgestellt und können so in Anspruch genommen werden. Da eine Fähigkeit in der Regel eine sehr abstrakte Beschreibung einer Tätigkeit darstellt, kümmert sich der Dienst darum, wie diese Fähigkeit umgesetzt wird. Die Fähigkeit definiert weitgehend das Resultat, d. h. was passieren muss, jedoch nicht den Vorgang, d. h. wie die Tätigkeit passieren soll. Dementsprechend stellt – wie in Kapitel 7 definiert – ein *Skill-level Service* eine abstrakte Fähigkeit eines Endeffektors für Objekte der Roboterzelle bereit und entscheidet auf Grundlage der aktuellen Situation und des Kontextes, wie die Fähigkeit konkret umgesetzt wird. Diese Rahmenbedingungen werden als *situativer Kontext* bezeichnet und können vom Service über das objektorientierte Modell der Roboterzelle hergeleitet werden.

Situativer Kontext

Dies ist in Abbildung 10.1 anhand eines Beispiels dargestellt. Der *GraspingService* bietet für Greifer die Fähigkeit an, Gegenstände aufzunehmen. Um diese Fähigkeit zu beschreiben, werden der Greifer als Subjekt oder ausführendes Gerät und der Gegenstand als Objekt oder Ziel der Fähigkeit benötigt. Infolgedessen bilden sie die Argumente des fähigkeitenzentrierten Dienstes. Da sie als Teil der Roboterzelle mit anderen (physischen) Objekten der *Robotics API* in Verbindung stehen, kann der Service von ihnen ausgehend den situativen Kontext inspizieren und herleiten. So kann der Service den Roboter ausfindig machen, an dem der Greifer als Endeffektor montiert ist. Gleichzeitig kann er feststellen, wo sich der zu greifende Gegenstand befindet. Dies ist nicht nur geometrisch zu verstehen, sondern auch logisch. Wie in Abbildung 10.1 dargestellt ist, kann der Service erkennen, dass sich der Gegenstand geordnet in einer Kiste befindet, die auf der Werkbank steht, auf der auch der Roboter montiert ist.

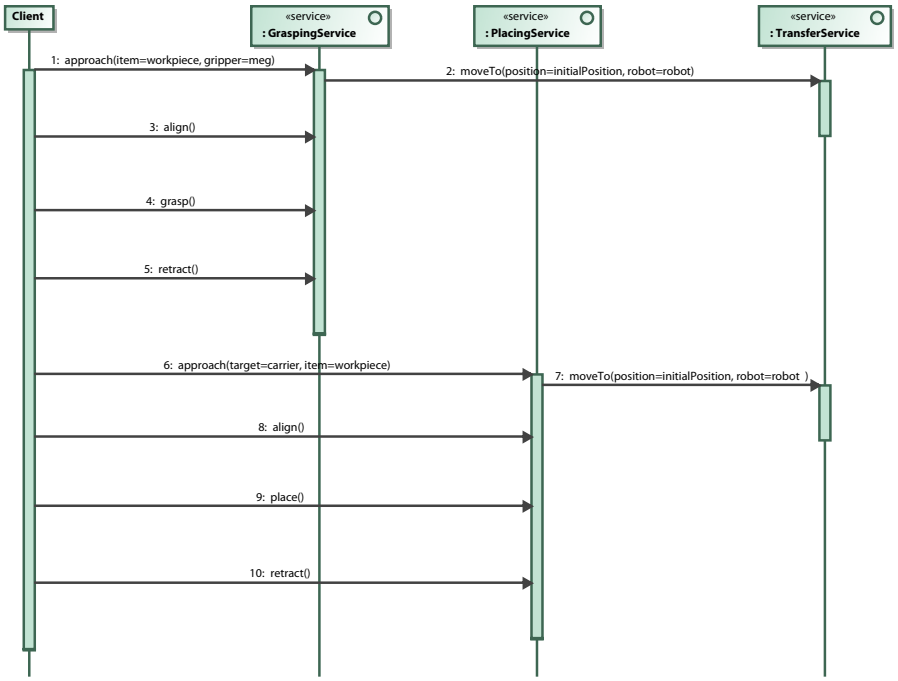


Abbildung 10.2: Um eine einfache Pick-and-Place-Aufgabe zu realisieren, werden zwei fähigkeitenzentrierte Dienste benötigt. Der *GraspingService* ist dafür zuständig, den Gegenstand mit dem Greifer zu nehmen. Anschließend wird der Gegenstand über den *PlacingService* wieder abgelegt. Der *TransferService* ist für die beiden Transferbewegungen innerhalb der Roboterzelle zuständig.

Abstrakte Fähigkeiten

Das Greifen eines Gegenstand ist ein gutes Beispiel für eine abstrakte Fähigkeit, die durch einen Greifer – in Kombination mit einem Manipulator – ausgeführt werden kann. Das Resultat dieser Tätigkeit ist klar definiert: Der Greifer hat den betreffenden Gegenstand fest gegriffen. Wie und wo er ihn jedoch gegriffen hat, wird dabei nicht spezifiziert. Auch ist unklar, ob jede mögliche Greifpose geeignet ist, um den Fertigungsprozess erfolgreich zu Ende zu bringen. Wenn der Gegenstand ungünstig gegriffen ist, kann er bspw. nicht mehr in einen Werkstückträger eingelegt werden. Daher ist der situative Kontext wichtig bei der konkreten Ausführung einer Tätigkeit. Neben dem Greifen von Gegenständen ist auch das Ablegen der Gegenstände eine Fähigkeit des Greifers, die als eigenständiger Dienst (*PlacingService*) realisiert wurde. Dabei wird jedoch der Greifer zu Beginn nicht betrachtet, sondern nur der abzulegende Gegenstand und dessen Ablegeposition, die bspw. im Fall einer Kiste oder eines Werkstückträgers ebenfalls als Objekt spezifiziert werden kann.

Durch einen *GraspingService* und einen *PlacingService* lässt sich eine einfache Pick-and-Place-Aufgabe realisieren, die in Abbildung 10.2 als Sequenzdiagramm dargestellt ist. Dabei wird zuerst ein Bauteil gegriffen und anschließend in einen Werkstückträger gelegt. Dabei fällt auf, dass sowohl das Greifen als auch Ablegen eines Gegenstandes nicht als atomare Tätigkeit modelliert sind. Stattdessen bestehen beide Fähigkeiten aus verschiedenen Schritten, die nacheinander ausgeführt werden müssen. Diese feingranulare Modellierung eröffnet mehr Möglichkeiten bei der Synchronisation von Tätigkeiten, insbesondere wenn die Roboterzelle über mehrere Manipulatoren verfügt.

Verteilt auf die beiden *Skill-level Services* untergliedert sich der Ablauf für die Pick-and-Place-Aufgabe in insgesamt acht verschiedene Schritte. Dabei werden die ersten vier Schritte durch den *GraspingService* ausgeführt. Sie sind schematisch in Abbildung 10.3 dargestellt:

1. Durch die Methode *approach* wird der Greifer in der Nähe bzw. oberhalb des zu greifenden Gegenstands positioniert.
2. Anschließend wird der Greifer über *align* an dem zu greifenden Gegenstand ausgerichtet. Von dort startet die eigentlich Greifbewegung.
3. Durch die Methode *grasp* wird der Gegenstand mit dem Greifer form- oder kraftschlüssig gegriffen. Dabei findet zum ersten Mal eine Aktion des Greifers statt. Zusätzlich kann eine Roboterbewegung geschehen.
4. Nachdem sich der Gegenstand im Greifer befindet wird dieser durch die Methode *retract* von seiner aktuellen Position langsam entfernt. Dabei muss berücksichtigt werden, dass sich der Gegenstand evtl. in einer Vorrichtung (z. B. einer Kiste) befindet und von dort mit entsprechender Vorsicht entnommen werden muss.

Durch den *GraspingService* wurde der Gegenstand erfolgreich mit dem Greifer gegriffen und gegebenenfalls aus seiner Vorrichtung entnommen (vgl. Abschn. 9.4.2). Der Service ist ebenfalls dafür zuständig, dass das objektorientierte Modell der Roboterzelle aktualisiert wurde. Demnach befindet sich auch das Softwareobjekt, das den Gegenstand repräsentiert, im GRIPPER.

Anschließend übernimmt der *PlacingService*, um den Gegenstand an der Zielposition, d. h. in einer Kiste, abzulegen:

5. Analog zum *GraspingService* wird durch die Methode *approach* der Gegenstand und damit implizit der Greifer in der Nähe bzw. oberhalb der Zielposition (d. h. der Kiste) positioniert.
6. Durch die Methode *align* wird der Gegenstand anschließend an der Kiste ausgerichtet und damit sehr nahe an seine Zielposition gebracht.
7. Durch *place* wird der Gegenstand schließlich an der Zielposition freigegeben und richtig positioniert. Analog zum *GraspingService* findet hier die Aktion des Greifers statt.
8. Zum Schluss wird der Greifer durch die Methode *retract* vom Gegenstand und aus der Kiste entfernt. Dies muss mit entsprechender Vorsicht geschehen, um den Gegenstand nicht zu deplatzieren.

Der *PlacingService* ist ebenfalls dafür zuständig, dass das objektorientierte Modell der Roboterzelle aktualisiert wird. Demnach befindet sich der Gegenstand im Modell der Roboterzelle in der Kiste.

Die beiden Dienste weisen Ähnlichkeiten auf, was die Methoden und Abläufe betrifft. Dabei ist zu beachten, dass beide Annäherungsbewegungen, die jeweils durch die Methode *approach* gestartet werden, nicht in die direkte Zuständigkeit der Dienste fallen. Stattdessen übernimmt diese Bewegungen ein weiterer Dienst, der *TransferService* (vgl. Abb. 10.3). Dadurch wird eine klare Verteilung der Zuständigkeiten erreicht. Der *GraspingService* ist verantwortlich für das Greifen von Gegenständen mithilfe eines Greifwerkzeugs. Er kennt sowohl die Position, von der aus der Greifvorgang starten soll, als auch die Position, an der sich der Greifer und der Gegenstand nach einer erfolgreichen Durchführung befinden. Der *GraspingService* kann vor der

Verteilung von Zuständigkeiten

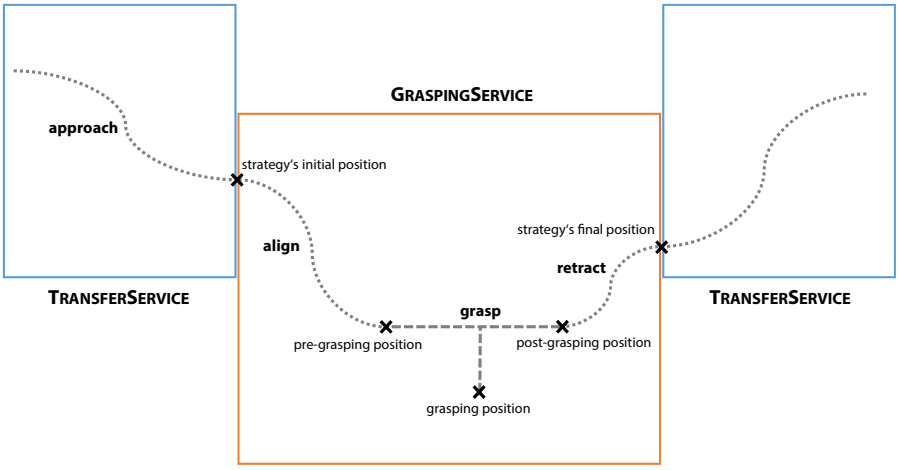


Abbildung 10.3: Um einen Gegenstand zu greifen, werden verschiedene Bewegungen des Greifers benötigt, die schematisch dargestellt sind. Dabei zeigt sich die unterschiedlichen Verantwortlichkeiten der Dienste. Während der *GraspingService* für das eigentliche Greifen des Gegenstands zuständig ist, wird die komplexe Anfahrtsbewegung an einen Service ausgelagert, der die Topologie der Zelle oder zumindest die möglichen Pfade innerhalb der Zelle kennt.

Durchführung der Fähigkeit prüfen, ob alle notwendigen Vorbedingung erfüllt sind, z. B. ob der Greifer noch keinen Gegenstand gegriffen hat. Darüber hinaus stellt er sicher, dass das objektorientierte Modell der Roboterzelle nach der erfolgreichen Durchführung der Fähigkeit aktualisiert ist. Er kennt alle Daten, um den Greifvorgang zu parametrisieren, und weiß, wie das Werkzeug und der Manipulator (mithilfe der *Robotics API*) angesteuert werden.

Dagegen weiß der *TransferService*, wie er die Manipulatoren der Roboterzelle in dieser bewegen kann. Er kann dazu die Topologie der Zelle kennen und mithilfe der in Abschnitt 6.5.3 vorgestellten kollisionsfreien Bahnplanung Trajektorien berechnen. Die Trajektorien führen den Manipulator, dessen Werkzeug und ggf. einen gegriffenen Gegenstand sicher und ohne Kollision durch die Zelle. Dies wird bspw. benötigt, um den Greifer oder den Gegenstand vor dem Greifen bzw. Ablegen zu positionieren (vgl. Abb. 10.3). Alternativ kennt der *TransferService* eine Menge von festen Trajektorien, die ihm dasselbe, wenn auch weniger flexibel, erlauben. Somit ist der *TransferService* für große Transferbewegungen der Manipulatoren innerhalb der Roboterzelle zuständig.

Die Struktur eines *GraspingServices* und eines *PlacingServices* ist in Abbildung 10.4 dargestellt. Beide *Skill-level Services* sind als Schnittstellen definiert, um eine Entkoppelung von ihrer Implementierung zu erreichen. In Abbildung 10.4 ist ist auch das Zusammenspiel der beiden Dienste mit dem bereits erwähnten *TransferService* zu erkennen. Während sowohl der *GraspingService* als auch der *PlacingService* als fähigkeitenzentrierte Dienste auf einem Werkzeug definiert sind, ist der *TransferService* zwar ebenfalls ein fähigkeitenzentrierter Dienst, der jedoch auf Manipulatoren definiert ist. Er hat daher ein anderes Abstraktionsniveau. Da die Dienste intern einen Ablauf verwalten, können sie gleichzeitig nur eine Fähigkeit ausführen. Allerdings können mehrere Dienste angefordert werden, sodass z. B. gleichzeitig zwei Gegenstände mit zwei unterschiedlichen Greifern aufgenommen werden können.

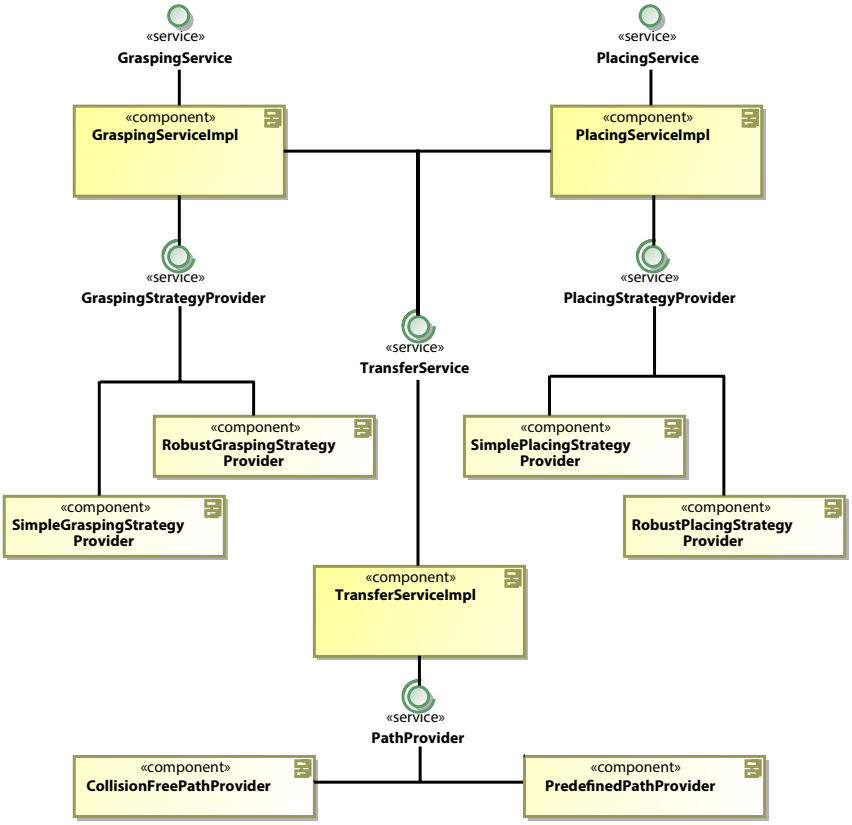


Abbildung 10.4: Der *GraspingService* und der *PlacingService* sind als Schnittstellen definiert. Ihre Implementierungen benötigen einen *TransferService*, um Bewegungen innerhalb der Roboterzelle ausführen zu können. Um konkrete Lösungsstrategien anbieten zu können, sind sie auf *StrategyProvider* angewiesen. Die Implementierung des *TransferService* verwendet dementsprechend *PathProvider*.

Austauschbare Fähigkeiten durch Strategien

Die Implementierungen aller in Abbildung 10.4 dargestellten Dienste benötigen *StrategyProvider* bzw. *PathProvider*, um zu funktionieren. Während ein fähigkeitenzentrierter Dienst nur abstrakt spezifiziert, dass er eine Fähigkeit umsetzen kann, stellt eine Strategie eine konkrete Lösung einer Fähigkeit dar. Es ist ein wiederverwendbares und flexibel parametrierbares Programmfragment, das eine Fähigkeit auf eine bestimmte Art und Weise realisiert. Welche Strategie gewählt werden soll, kann mithilfe des oben vorgestellten situativen Kontextes bestimmt werden. Dazu weiß jedes Bauteil bzw. jeder zu bearbeitende Gegenstand, wann er wie gehandhabt bzw. bearbeitet werden möchte. Das *Wie* bezieht sich dabei auf eine Strategie und das *Wann* auf den situativen Kontext. Diese Informationen eines Objekts werden von einem fähigkeitenzentrierten Dienst verwendet, um sie mit dem aktuellen Zustand der Roboterzelle zu vergleichen. Anschließend wird die richtige Strategie ausgewählt, parametrisiert und ausgeführt. Dieses Vorgehen wird detailliert im nächsten Abschnitt erläutert.

Zusammenfassend lässt sich festhalten, dass durch einen fähigkeitenzentrierten Dienst von der Roboterprogrammierung abstrahiert wird. Es werden ausschließlich abstrakte Tätigkeiten beschrieben, die unterschiedlich ausgeführt werden können. Damit wurde in dieser Arbeit eine Möglichkeit geschaffen, dass eine allgemeine Beschreibung der Arbeitsabläufe innerhalb einer Roboterzelle entstehen kann. Erst durch die Objekte, die gegriffen,

bewegt oder bearbeitet werden, wird entschieden, wie die konkrete Umsetzung aussieht. Dazu weiß jedes Objekt, wann es wie gegriffen, bewegt oder bearbeitet werden möchte. Jeder fähigkeitenzentrierte Dienst ist für ein bestimmtes Werkzeug bzw. eine bestimmte Klasse von Werkzeugen konzipiert (z. B. Greifer). Für jede Werkzeugklasse (z. B. Schweißbrenner oder Klebepistole) kann es wiederum mehrere solcher Dienste geben, um alle Fähigkeiten hinreichend abzudecken.

10.2 STRATEGIES: FLEXIBLE UND WIEDERVERWENDBARE LÖSUNGEN

Wie in Abbildung 10.4 gezeigt wurde, stützt sich die Implementierung eines fähigkeitenzentrierten Dienstes auf Strategien ab. Das bedeutet, dass der Dienst keine eigene Implementierung einer Fähigkeit besitzt. Stattdessen wertet er den situativen Kontext anhand des objektorientierten Modells der Roboterzelle aus und entscheidet, wie die Fähigkeit umgesetzt werden kann. Die Umsetzung einer Fähigkeit wird in einer Strategie gekapselt..

Eine *Strategy* stellt in der Softwaretechnik ein Pattern dar, das von Gamma et al. [97] wie folgt definiert wird:

> "Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it." [97]

Folglich stellt jede Fähigkeit eine neue *Familie* bzw. Menge von Algorithmen dar. Im Rahmen der serviceorientierten Modellierung von Roboterzellen wird eine Strategie wie folgt definiert:

Definition 10.1. Eine Strategie (engl.: *strategy*) stellt eine konkrete Implementierung einer Werkzeugfähigkeit dar, die von einem *Skill-level Service* verwendet werden kann. Um wiederverwendbar zu sein, ist eine Strategie parametrierbar.

Damit definiert die Werkzeugfähigkeit die von Gamma et al. [97] beschriebene Familie von Algorithmen. Jede konkrete Strategie kapselt eine eigene Lösung dieser Fähigkeit. Dazu kann bspw. eine Schnittstelle oder abstrakte Klasse verwendet werden, um die Methoden einer Fähigkeit zu definieren. Dadurch sind die einzelnen Lösungen austauschbar und können bei Bedarf bzw. Eignung angewandt werden.

So gibt es bspw. mehrere Möglichkeiten ein Bauteil mit einem Parallelgreifer über den GraspingService aufzunehmen (vgl. Abschn. 10.4). Eine einfache Strategie besteht darin, das Bauteil von einer gegebenen Richtung anzufahren und es anschließend sowohl kraftschlüssig als auch von drei Seiten formschlüssig zu greifen. Die Greifweite und -kraft ist dabei ebenso einstellbar, wie die Anfahrtspunkte und die Greifposition. Eine Alternative dazu stellt ein Greifen mit den Fingerspitzen dar. Hier wird das Bauteil ausschließlich kraftschlüssig zwischen die beiden Fingerspitzen geklemmt. Diese Strategie ist ebenfalls parametrierbar. Zusätzlich kann es deutlich komplexere Strategien geben, die bspw. die Kraftsensorik eines Manipulators oder einer Kraftmessdose nutzen.

Das Objekt weiß, wie es gegriffen wird

Jedoch ist es nicht ausreichend mit Strategien austauschbare und wiederverwendbare Werkzeugfähigkeit zu haben. Zusätzlich muss definiert werden, welche Strategie wann und mit welcher Parametrisierung angewandt wird. Dabei übernimmt der fähigkeitenzentrierte Dienst die Auswahl und Auswertung der Strategien mithilfe des situativen Kontextes. Allerdings

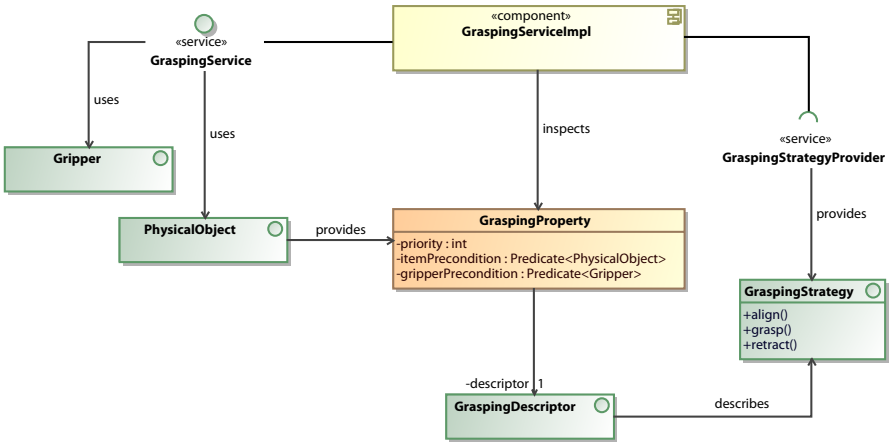


Abbildung 10.5: Eine *Strategy* wird dem fähigkeitenzentrierten Dienst durch einen *StrategyProvider* bereitgestellt. Die Auswahl der richtigen Strategie und dessen Parametrierung wird über einen *Descriptor* durch das Bauteil bestimmt.

kann er nicht entscheiden, welche Strategien geeignet sind. Dies wird stattdessen durch das betroffene Objekt vorgegeben. Es weiß, wie es gegriffen, bewegt oder bearbeitet werden möchte. Daher ist in dem handzuhabenden bzw. zu bearbeitenden Objekt gespeichert, unter welchen Rahmenbedingungen welche Strategie mit welchen Parametern ausgewählt, instanziiert und ausgeführt wird.

Das Zusammenspiel zwischen einem fähigkeitenzentrierten Dienst, einer Strategien und dem (physischen) Objekt ist in Abbildung 10.5 anhand des GraspingService dargestellt. Der *Skill-level Service* hat einen Greifer und das zu greifende Objekt als Argumente seiner Methoden. Die Implementierung stellt über die GraspingService-Schnittstelle einen fähigkeitenzentrierten Dienst zur Verfügung. Um die entsprechende Funktionalität anbieten zu können, benötigt die Implementierung ihrerseits einen oder mehrere GraspingStrategyProvider. Diese werden ebenfalls als Service registriert und können jeweils eine GraspingStrategy bereitstellen bzw. instanziieren.

Situative Auswahl einer Strategie

Die Information, welche Strategie wann verwendet werden soll, ist über eine Property (vgl. Abschn. 6.1) Bestandteil des zu greifenden Objekts. Für den GraspingService existiert dazu die GraspingProperty, von der ein PhysicalObject mehrere besitzen kann. Intern untergliedert sich eine solche Eigenschaft in zwei Teile: den *Descriptor* und die *Precondition*. Der *Descriptor* definiert die zu verwendende Strategie und enthält alle notwendigen Informationen, um die Strategie zu parametrisieren (z. B. den Greifpunkt bezogen auf das Basiskoordinatensystem des Gegenstands). Die *Precondition* besteht aus einer Konjunktion mehrerer Prädikate, die jeweils auf einem Argument des fähigkeitenzentrierten Dienstes (z. B. dem Greifer und den zu greifenden Gegenstand) definiert sind und auf dem Modell der Roboterzelle ausgewertet werden. Dadurch kann man definieren, unter welchen Bedingungen eine Strategie angewandt werden soll.

Für die Auswahl und Instanziierung einer Strategie geht ein fähigkeitenzentrierter Dienst wie folgt vor:

1. Zuerst werden alle zutreffenden Properties des PhysicalObjects erfasst (z. B. jede GraspingProperty im Fall des GraspingService).

2. Daraufhin werden die PROPERTIES gefiltert, indem die *Precondition* ausgewertet wird. Falls die *Precondition* nicht gilt, wird die PROPERTY verworfen. Im Fall des GRASPINGSERVICE wird eine Konjunktion zweier PREDICATES ausgewertet, die über den Greifer bzw. dem zu greifenden PHYSICALOBJECT definiert sind (vgl. Abb. 10.5).

3. Alle verbleibenden PROPERTIES werden nach ihrer Priorität sortiert. Die Priorität ist Teil der PROPERTY und ermöglicht es, mehrere unterschiedlich gewichtete Strategien zu spezifizieren. Falls eine Strategie momentan noch nicht oder nicht mehr verfügbar ist, kann eine Ausweichstrategie gewählt werden.

4. Anschließend wird der passende *StrategyProvider* für den *Descriptor* mit der höchsten Priorität gesucht und die Strategie wird zu instanzieren versucht. Falls dies nicht möglich ist, wird der nächste *Descriptor* ausgewählt und der Vorgang wiederholt.

Der letzte Vorgang kann bspw. scheitern, wenn die Strategie aktuell nicht verfügbar ist oder der *StrategyProvider* aufgrund des aktuelle Zustands der Roboterzelle die Strategie nicht für anwendbar hält (z. B. weil Aktuatoren nicht über ausreichende Steueroperationen verfügen).

Nachdem eine passende Strategie gefunden und erfolgreich instanziiert wurde, kann sie vom *Skill-level Service* ausgeführt werden. In der Regel verfügt dazu jeder Dienst pro Strategie über eine interne Zustandsmaschine, die schrittweise abgearbeitet werden muss (vgl. Abschn. 10.3). Dadurch wird sichergestellt, dass die Strategie in der richtigen Reihenfolge und vollständig ausgeführt wird. Die Strategie definiert zusätzlich ihren Start- und Endpunkt, d. h. sie definiert, an welcher Position sie den Endeffektor (z. B. den Greifer) vor Anwendung der ersten Aktion erwartet. Diese Information wird vom *Skill-level Service* verwendet, um den Endeffektor über einen *TransferService* (vgl. Abb. 10.5) an die gewünschte Position zu bringen. Der Endpunkt einer Strategie ist wichtig, damit nachfolgende *Skill-level Service* Transferbewegungen planen und ausführen lassen können.

Flexible Auswahlkriterien

Die oben erwähnten *Preconditions* oder Vorbedingungen ermöglichen es, die richtigen Strategien für den entsprechenden situativen Kontext zu spezifizieren. Sie bilden ein Auswahlkriterium, dass flexibel zu definieren ist und dabei das objektorientierte Modell der Roboterzelle mit seinen geometrischen, topologischen und semantischen Informationen ausnutzt. Über den Endeffektor lassen sich bspw. folgende Aussagen formulieren:

- Der Endeffektor ist von einem bestimmten Typ (d. h. er ist die Instanz einer bestimmten Klasse) und besitzt einen festgelegt Namen.
- Der Endeffektor verfügt über eine bestimmte Menge von Steuerungsoperationen, d. h. er verfügt über ein bestimmtes ACTUATORINTERFACE.
- Der Endeffektor ist an einem Roboter montiert, der bestimmte Eigenschaften aufweist (z. B. ist er von einem vorgegebenem Typ, hat einen festgelegten Namen oder besitzt eine erforderliche Menge von Steuerungsoperationen).

Analog dazu lassen sich ebenfalls Aussagen über einen Gegenstand, d. h. ein PHYSICALOBJECT, formulieren, wie zum Beispiel:

- Das Objekt besitzt einen bestimmten Typ (d. h. es ist die Instanz einer bestimmten Klasse).

- Das Objekt liegt in einem Werkstückträger, in einer Kiste oder auf einem Tisch.
- Das Objekt ist aktuell von einem Greifer mit bestimmten Eigenschaften gegriffen.

Somit bieten die *Preconditions* und die verwendeten Prädikate ein mächtiges Instrument, um Strategien zielgerichtet aber dennoch flexibel zu definieren. Sie sind immer spezifisch für einen *Skill-level Service* und werden daher über dessen Argumente ausgewertet.

Zusammenfassend lässt sich festhalten, dass Strategien wiederverwendbare Programmfragmente sind, die durch ihre Parametrisierung in unterschiedlichen Situationen flexibel eingesetzt werden können. Alle Informationen, um eine Strategie auszuwählen, zu konfigurieren und zu verwenden, sind über einen *Descriptor* und *Preconditions* Bestandteil des Handhabungsobjekts bzw. des Bauteils. Dadurch können die verwendeten Strategie eines Bauteils direkt über das Bauteil verändert werden. Damit ebnet diese Arbeit den Weg für eine bauteilzentrierte Adaption der Roboterzelle – selbst auf unterster Ebene bei der Roboterprogrammierung (vgl. Kap. 11).

10.3 BEISPIELE FÜR FÄHIGKEITENZENTRIERTE DIENSTE

Die Idee fähigkeitenzentrierter Dienste wurde in Abschnitt 10.1 vorgestellt. In diesem Abschnitt werden insgesamt vier *Skill-level Services* vorgestellt, die in der Factroy 2020 benötigt werden. In Abschnitt 10.3.1 wird zuerst der GraspingService zum Greifer von Gegenständen detailliert vorgestellt. Anschließend wird der PlacingService zum Ablegen vorher gegriffener Gegenstände beschrieben. Die beiden Dienst wurden bereits früher als Beispiele eingeführt und im folgenden Abschnitt zur Vollständigkeit detailliert diskutiert. Zum Abschluss werden in Abschnitt 10.3.2 zwei Dienste für elektrische Schraubsysteme erläutert.

10.3.1 *Greifen und Ablegen von Gegenständen*

Der GraspingService ist ein fähigkeitenzentrierter Dienst, um einen Gegenstand (d. h. ein PhysicalObject) über einen mit zwei oder mehreren Fingern ausgestatteten Greifer (d. h. einen Gripper) aufzunehmen und diesen kraft- oder formschlüssig zwischen den Fingern zu halten. Diese Fähigkeit benötigt mindestens einen Manipulator, um den Greifer zu dem Gegenstand zu bewegen und eine passende Greifpose einzunehmen. Dies ist ein Ablauf, der sich situativ unterscheiden kann. Daher gibt es unterschiedliche Strategien und Lösungsmöglichkeiten, um einen Gegenstand zu greifen. Diese sind als GraspingStrategy gekapselt und können vom Service bei Bedarf verwendet werden (vgl. Abschn. 10.4).

Erst greifen...

Die Struktur des GraspingService und seiner Implementierung ist in Abbildung 10.6 dargestellt. Strategien werden durch einen GraspingStrategyProvider als Service zur Verfügung gestellt. Zur Laufzeit informiert die *OSGi Service Platform* die Implementierung des GraspingService über neue *Provider*, woraufhin deren Strategien zur Verfügung stehen. Die Auswahl einer GraspingStrategy erfolgt über eine GraspingProperty, die durch einen *Descriptor* alle Parameter der Strategie enthält. Über eine *Precondition* und Predicates werden zudem die notwendigen Vorbedingungen für die Anwendung der Strategie deklariert.

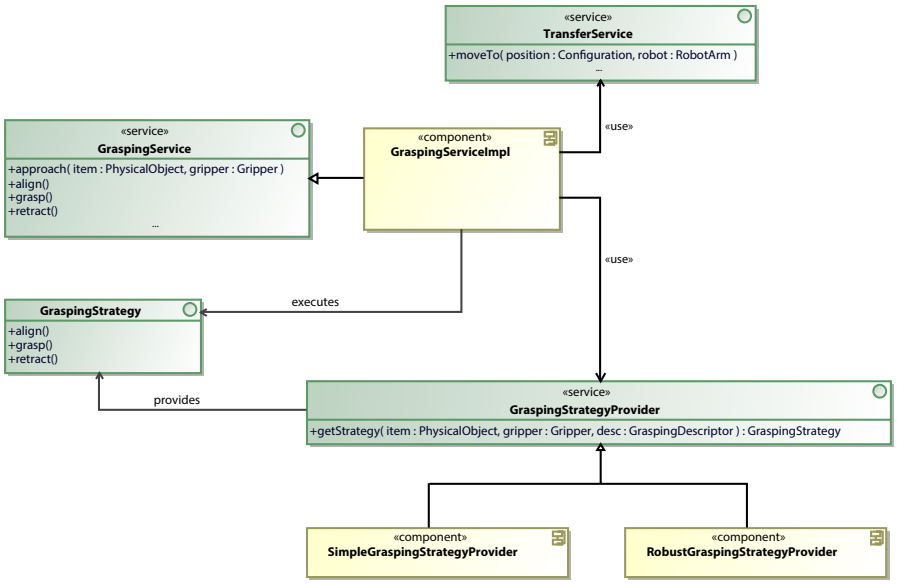


Abbildung 10.6: *Skill-level Service* für das Greifen von Gegenständen: Die Fähigkeit ist als Service öffentlich verfügbar. Die Implementierung verwendet eine GraspingStrategy für die Umsetzung der Fähigkeit. Diese kann über als Dienst registrierte *Provider* bezogen werden.

Die Schnittstelle des GraspingService verfügt über vier Methoden, um die einzelnen Schritte der Fähigkeit zu beschreiben und eine interne Zustandsmaschine anzusteuern. Dadurch wird ein korrekter Ablauf der Fähigkeit sichergestellt. Für jede ausgeführte Fähigkeit existiert eine Zustandsmaschine, die die Ausführung überwacht und steuert. Die Zustandsmaschine ist in Abbildung 10.7 dargestellt. Innerhalb von Zuständen wird die Strategie in einer *do*-Methode ausgeführt. Dadurch kann der Zustand sowohl im Fehlerfall frühzeitig als auch nach erfolgreicher Abarbeitung verlassen werden. Dementsprechend gibt es für beide Fälle nachfolgende Zustände.

Durch den Aufruf der Methode *approach* wird die Zustandsmaschine gestartet und geht in Approaching über:

- Im Zustand Approaching wird der Gripper in die Nähe des zu greifenden PhysicalObjects bewegt. Dabei definiert eine zuvor gewählte GraspingStrategy den Zielpunkt der Bewegung. Geplant und ausgeführt wird die Bewegung vom *TransferService*. Nach Abschluss der Bewegung bzw. nach dem Beginn des letzten Bewegungsabschnittes geht die Zustandsmaschine per *completion event* in Considered über.

Im Zustand Considered befindet sich der Gripper an dem durch die Strategie vorgegebenen Startpunkt. Daher kann durch den Aufruf der Methode *align* der Greifvorgang gestartet werden und die Zustandsmaschine geht nach Aligning über:

- Im Zustand Aligning wird der Gripper am zu greifenden PhysicalObject ausgerichtet. Dieser Vorgang wird durch die gewählte Strategie ausgeführt. Daher bestimmt die Strategie bzw. deren Parametrierung, wie der Greifer ausgerichtet wird. Nach Abschluss des Vorgangs geht die Zustandsmaschine per *completion event* in Ready über.

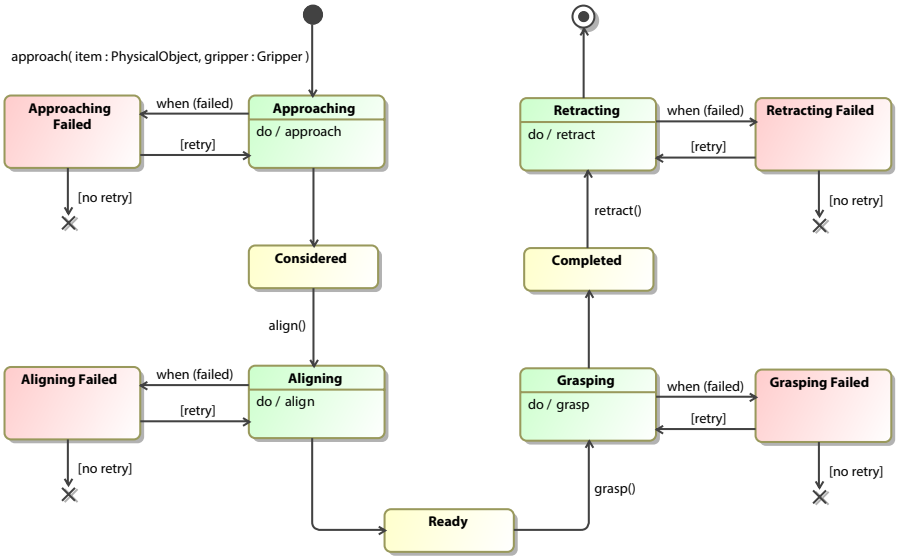


Abbildung 10.7: Das interne Verhalten des *Skill-level Services* für das Greifen von Gegenständen kann als Zustandsmaschine implementiert werden. Die farbliche Abstufung unterscheidet Fehler- und Wartezustände von aktiven Zuständen, in denen Greifer und Manipulator innerhalb der Strategie gesteuert werden.

Von dort aus kann der Gegenstand mit dem Greifer letztendlich gegriffen werden. Daher geht die Zustandsmaschine nach dem Aufruf der Methode *grasp* in den Zustand GRASPING über:

- Im Zustand GRASPING nimmt der GRIPPER das PHYSICALOBJECT kraft- oder formschlüssig auf, wobei der Vorgang durch die gewählte Strategie ausgeführt wird. Zudem wird das vorhandene PLACEMENT zwischen dem PHYSICALOBJECT und seiner Umgebung (z. B. einer Kiste) aufgelöst und eine CONNECTION mit dem Greifer hergestellt. Nach Abschluss des Vorgangs geht die Zustandsmaschine per *completion event* in COMPLETED über.

Vom Zustand COMPLETED aus kann durch den Aufruf von *retract* der letzte Schritt des Greifens gestartet werden und die Zustandsmaschine geht nach RETRACTING über:

- Im Zustand RETRACTING bewegt der Greifer bzw. der Manipulator den Gegenstand aus seiner Umgebung (z. B. einer Kiste) in den freien Raum der Roboterzelle, von wo aus bspw. eine Transferbewegung folgen kann.

Diese Zustandsmaschine besitzt darüber hinaus Fehlerzustände, über die eine wiederholte Ausführung eines Schritts gestartet werden kann. Dies hängt jedoch davon ab, ob die zugrunde liegende Strategie dies unterstützt. In Abschnitt 10.4.2 wird bspw. eine robuste und fehlertolerante Greifstrategie vorgestellt, die mithilfe von Sensoren erkennt, ob sie das Bauteil erfolgreich gegriffen hat. Zudem kann das Verhalten des Dienstes, d. h. die Zustandsmaschine, direkt mithilfe einer grafischen Notation implementiert werden (vgl. Abschn. 11.5).

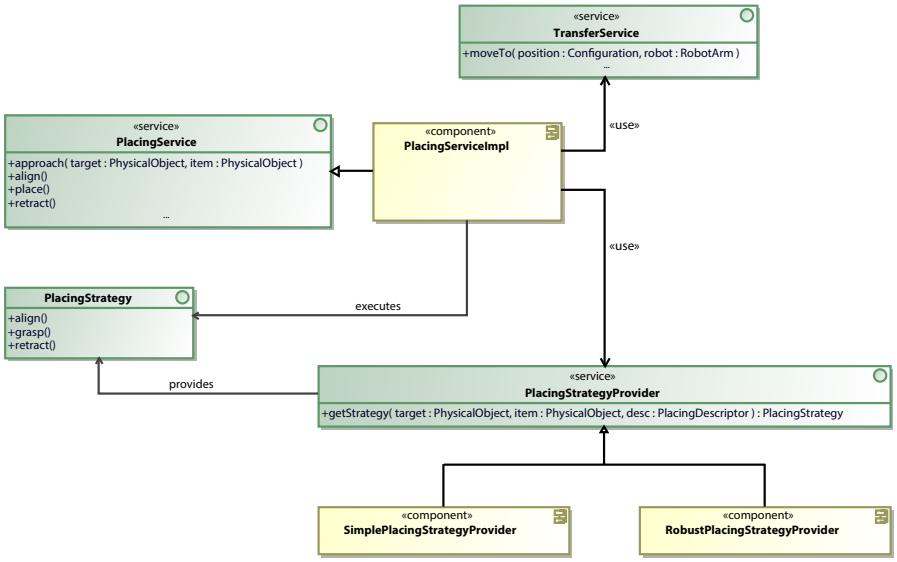


Abbildung 10.8: *Skill-level Service* für das Ablegen von Gegenständen: Die Fähigkeit ist als Service öffentlich verfügbar. Die Implementierung verwendet eine PlacingStrategy für die Umsetzung der Fähigkeit. Diese kann über als Dienst registrierte *Provider* bezogen werden.

... und dann ablegen

Nachdem ein Gegenstand mithilfe des GraspingServices von einem Greifer aufgenommen wurde, ist er im objektorientierten Modell der Roboterzelle über eine Relation mit dem Greifer verbunden. Das heißt, dass anschließend ein anderer Dienst den Gegenstand und damit den Greifer und dessen Manipulator weiterverwenden kann, um den Gegenstand gezielt abzulegen oder in einer Vorrichtung einzusetzen. Ein Beispiel für einen solchen Dienst stellt der PlacingService dar. Er ist ein fähigkeitenzentrierter Dienst, um einen mit einem Gripper gehaltenen Gegenstand an einer Zielposition abzulegen. Die Zielposition kann entweder als PhysicalObject (z. B. ein Tisch) definiert oder durch Angabe eines Koordinatensystem weiter spezifiziert werden. Da das Ablegen eines Gegenstands ebenfalls ein Ablauf ist, der sich situativ unterscheiden kann, gibt es unterschiedliche Strategien und Lösungsmöglichkeiten, um einen vorher gegriffenen Gegenstand abzulegen. Diese sind als PlacingStrategy gekapselt und können vom Service bei Bedarf verwendet werden.

Die Struktur des PlacingService ist in Abbildung 10.8 dargestellt. Strategien werden ebenfalls als Service durch einen PlacingStrategyProvider zur Verfügung gestellt. Die Auswahl einer PlacingStrategy erfolgt wiederum über eine PlacingProperty, die durch einen *Descriptor* die notwendigen Parameter der Strategie enthält. Über eine *Precondition* und Predicates werden zudem die Vorbedingungen für die Anwendung der Strategie deklariert. In Abbildung 10.8 sind vier Methoden des PlacingService dargestellt, welche die einzelnen Schritte der Fähigkeit beschreiben. Analog zum GraspingService steuern diese Methoden eine interne Zustandsmaschine an, um einen korrekten Ablauf der Fähigkeit sicherzustellen. Die Zustandsmaschine des PlacingServices ist analog zur Zustandsmaschine des GraspingServices aufgebaut (vgl. Abb. 10.7). Allerdings verwendet sie eine PlacingStrategy, um die einzelnen Schritte zu realisieren.

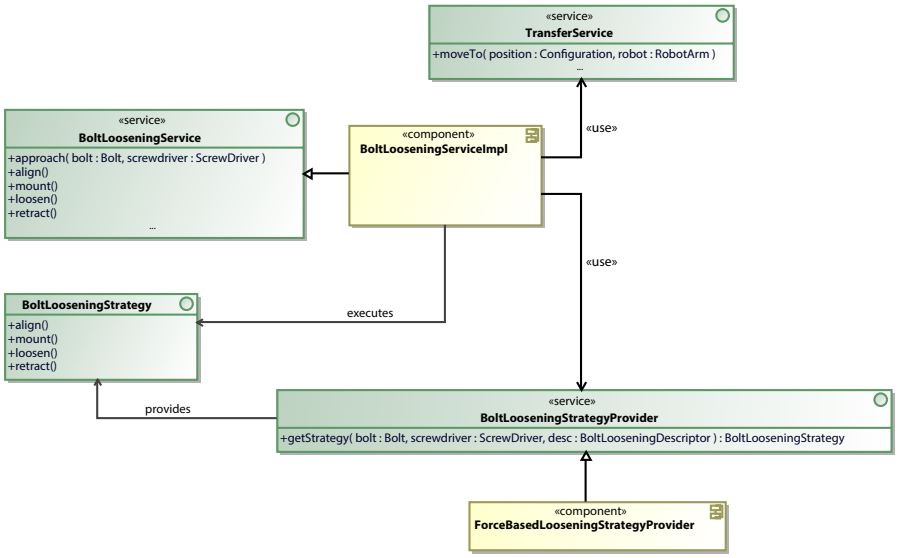


Abbildung 10.9: *Skill-level Service* für das Lösen einer Schraube: Die Fähigkeit ist als Service öffentlich verfügbar. Die Implementierung verwendet eine BoltLooseningStrategy für die Umsetzung der Fähigkeit. Diese kann über als Dienst registrierte *Provider* bezogen werden.

10.3.2 *Lösen und Anziehen von Schrauben*

Der BoltLooseningService ist ein fähigkeitenzentrierter Dienst, um mit einem am Roboterflansch montierten ScrewDriver eine Schraube zu lösen. Der ScrewDriver ist ein Device der *Robotics API* und repräsentiert ein elektrisches Schraubsystem (vgl. Kap. 8). Zudem sind Schrauben durch die Klasse Bolt und Gewinde durch die Klasse Hole mithilfe der *Robotics API* modelliert. Die Fähigkeit des Lösens von Schrauben benötigt mindestens einen Manipulator, um den ScrewDriver zur Schraube zu bewegen. Insbesondere das Aufnehmen einer Schraube ist ein Vorgang, der sich situativ unterscheiden kann. Daher gibt es unterschiedliche Lösungsmöglichkeiten, um einen Schraube zu lösen. Diese sind als eigene Strategie gekapselt und können vom Dienst bei Bedarf verwendet werden.

Lösen einer Schraube

Die Struktur des BoltLooseningServices ist in Abbildung 10.9 dargestellt. Die Schnittstelle des Dienstes verfügt über fünf Methoden, um die Schritte der Fähigkeit zu beschreiben und eine interne Zustandsmaschine anzusteuern. Dabei fällt auf, dass sich die Dienste ähnlich sind. Sie verfügen über einen verwandten Ablauf, der sich letztendlich über die werkzeugspezifischen Aktionen unterscheidet. Diese Ähnlichkeit ermöglicht es unter anderem, die Abläufe einer Roboterzelle zu gliedern und somit eine Trennung zwischen dem Arbeitsabläufen einer Roboterzelle und der eigentlichen Roboter- und Werkzeugprogrammierung zu erreichen.

Die in Abbildung 10.10 dargestellte Zustandsmaschine wird durch den Aufruf der Methode *approach* gestartet und geht in Approaching über:

- Im Zustand Approaching wird der ScrewDriver in die Nähe der Schraube bewegt. Dabei definiert die zuvor ausgewählte Strategie den Zielpunkt dieser Bewegung. Geplant und ausgeführt wird die Bewegung vom *TransferService*. Nach Abschluss der Bewegung bzw. nach dem Beginn des letzten Bewegungsabschnittes geht die Zustandsmaschine per *completion event* in Considered über.

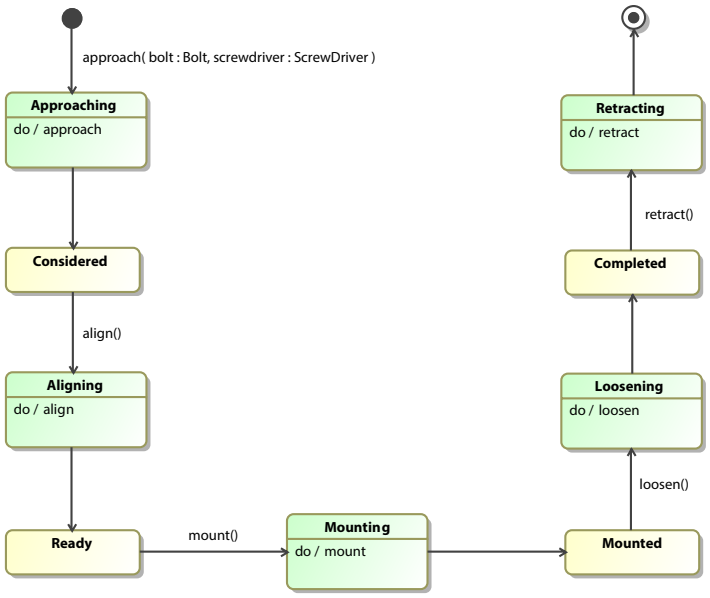


Abbildung 10.10: Das interne Verhalten des *Skill-level Services* für das Lösen einer Schraube kann als Zustandsmaschine implementiert werden. Die farbliche Abstufung unterscheidet Wartezustände von aktiven Zuständen, in denen Schraubsystem und Manipulator innerhalb der Strategie gesteuert werden. Fehlerfälle werden aus Gründen der Übersichtlichkeit hier nicht dargestellt.

Im Zustand CONSIDERED befindet sich der SCREWDRIVER an dem von der Strategie vorgegebenen Startpunkt. Daher kann durch den Aufruf der Methode *align* der nächste Vorgang gestartet werden und die Zustandsmaschine geht in ALIGNING über:

- Im Zustand ALIGNING wird der SCREWDRIVER an der Schraube ausgerichtet. Dieser Vorgang wird durch die gewählte Strategie ausgeführt. Daher bestimmt die Strategie bzw. deren Parametrierung, wie der SCREWDRIVER ausgerichtet wird. Nach Abschluss des Vorgangs geht die Zustandsmaschine per *completion event* in READY über.

Von dort aus kann der Bit des SCREWDRIVERS schließlich in die Schraube eingesetzt werden. Daher geht die Zustandsmaschine nach dem Aufruf der Methode *mount* in MOUNTING über:

- Im Zustand MOUNTING wird der Bit des SCREWDRIVERS in die Schraube geführt, wobei der Vorgang durch die gewählte Strategie ausgeführt wird. Es wird eine CONNECTION zwischen SCREWDRIVER und BOLT hergestellt. Nach Abschluss des Vorgangs geht die Zustandsmaschine per *completion event* in MOUNTED über.

Vom Zustand MOUNTED aus kann durch den Aufruf der Methode *loosen* das Lösen der Schraube gestartet werden. Die Zustandsmaschine geht in den Zustand LOOSENING über:

- Im Zustand LOOSENING wird die Schraube aus ihrem bisherigen Gewinde gelöst. Nach Abschluss des Vorgangs geht die Zustandsmaschine per *completion event* in COMPLETED über.

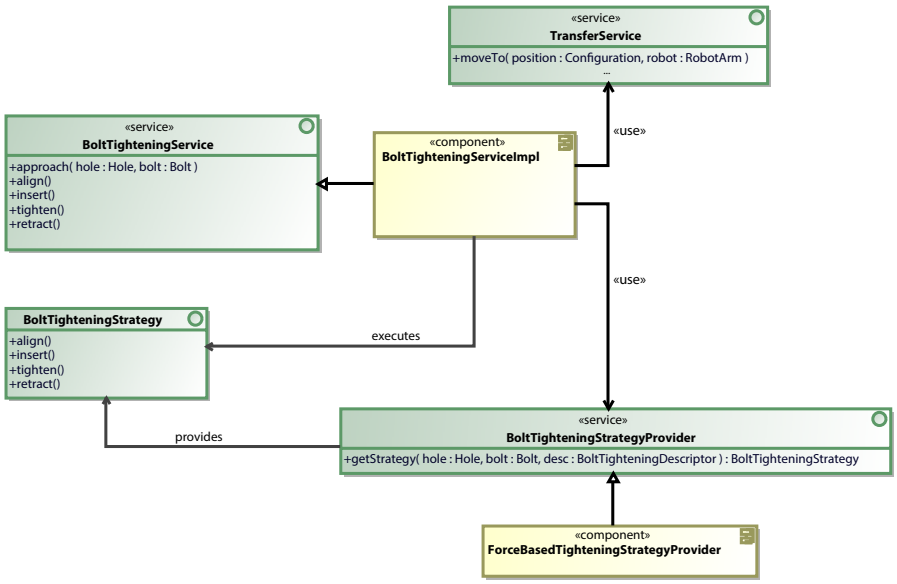


Abbildung 10.11: *Skill-level Service* für das Festziehen einer Schraube: Die Fähigkeit ist als Service öffentlich verfügbar. Die Implementierung verwendet eine BoltTighteningStrategy für die Umsetzung der Fähigkeit. Diese kann über als Dienst registrierte *Provider* bezogen werden.

Vom Zustand Completed aus kann mit Aufruf der Methode *retract* der letzte Schritt gestartet werden. Dementsprechend geht die Zustandsmaschine in Retracting über:

- Im Zustand Retracting wird die Schraube und das Schraubsystem durch den Manipulator vom Gewinde in den freien Raum der Roboterzelle bewegt, von wo aus bspw. eine Transferbewegung folgen kann. Der Vorgang wird ebenfalls durch die gewählte Strategie ausgeführt.

Festziehen einer Schraube

Zusätzlich existiert mit dem BoltTighteningService ein fähigkeitenzentrierter Dienst, um mit einem am Roboterflansch montierten ScrewDriver eine Schraube in ein Gewinde einzusetzen und anzuziehen. Dabei muss die Schraube (Bolt) bereits vom Schraubsystem (ScrewDriver) aufgenommen sein. Um dann die Schraube und damit das Schraubsystem zum Gewinde (Hole) zu bewegen, ist ein Manipulator notwendig. Insbesondere das Einsetzen einer Schrauben ist ein Vorgang, der sich situativ unterscheiden kann. Daher gibt es unterschiedliche Lösungsmöglichkeiten, um einen Schraube festzuziehen, die wiederum als Strategie gekapselt sind (vgl. Abb. 10.11).

Der Ablauf und die Zustandsmaschine sind sehr ähnlich zum BoltLooseningService und dessen in Abbildung 10.10 dargestellter Zustandsmaschine. Das bedeutet, dass am Anfang die Schraube und damit implizit das Schraubsystem in die Nähe des Gewindes bewegt wird. Anschließend wird die Schraube am Gewinde ausgerichtet. Die Zustandsmaschine befindet sich dann ebenfalls in einem Zustand Ready. Nach dem Aufruf der Methode *insert* geht diese jedoch in dem Zustand Inserting über:

- Im Zustand Inserting wird die Schraube in das Gewinde geführt, wobei der Vorgang durch eine zuvor gewählte Strategie ausgeführt wird. Nach Abschluss des Vorgangs geht die Zustandsmaschine per *completion event* in Inserted über.

Von diesem Zustand aus kann durch den Aufruf der Methode *tighten* das Festziehen der Schraube gestartet werden. Die Zustandsmaschine geht dazu in TIGHTENING über:

- Im Zustand TIGHTENING wird die Schraube in dem Gewinde mit einem definierten Drehmoment festgezogen. Danach geht die Zustandsmaschine per *completion event* in COMPLETED über.

Ab dem Zustand COMPLETED sind die Transitionen wieder identisch, d. h. durch den Aufruf der Methode *retract* wird der letzte Schritt gestartet. Dabei wird SCREWDRIVER durch den Manipulator von der Schraube gelöst und in den freien Raum der Roboterzelle bewegt, von wo aus bspw. eine Transferbewegung folgen kann.

10.4 BEISPIELE FÜR STRATEGIEN

Im vorherigen Abschnitt wurden unterschiedliche Services für Fähigkeiten, d. h. *Skill-level services*, vorgestellt. Diese verwenden den situativen Kontext und die in den Objekten enthaltenen Informationen, um die bestmögliche Lösung zu finden und anzuwenden. Diese Lösungen sind als *Strategy* in Form eines parametrierbaren Programmfragments implementiert. Eine Strategie für die Umsetzung einer Fähigkeit ist die feinste Granularität einer serviceorientierten Roboterzelle. Dementsprechend findet innerhalb einer Strategie die Werkzeug- und Roboterprogrammierung statt, von der bisher abstrahiert werden konnte. In diesem Abschnitt werden unterschiedliche Strategien für das Greifen von Bauteilen mithilfe eines Parallelgreifers vorgestellt. Dabei werden die geometrischen Merkmale des Greifer bzw. des Bauteils verwendet, um eine möglichst hohe Wiederverwendbarkeit zu gewährleisten. Spezifische Informationen, die nicht aus den Merkmalen abgeleitet werden können, sind Teil der Parametrisierung der Strategie.

Zu Beginn wird in Abschnitt 10.4.1 ein Ablauf einer einfachen Greifstrategie vorgestellt, der generisch parametrisiert und einfach angepasst werden kann. Mithilfe dieses Ablaufs lässt sich ein formschlüssiges Greifen eines Bauteils realisieren. Zudem kann durch eine Anpassung der Ablauf zum Greifen eines Bauteils mit den Fingerspitzen des Greifers verwendet werden. In Abschnitt 10.4.2 wird eine robuste Strategie vorgestellt, die fehlerhafte Greifversuche mittels Sensorik erkennt und den Greifvorgang optimiert wiederholt. Die beiden Strategien zeigen, dass durch den in dieser Arbeit beschriebenen Ansatz eine greifer- und roboterunabhängige Programmierung möglich ist.

10.4.1 *Einfache Greifstrategien*

In Listing 10.1 ist eine einfache Greifstrategie als Programmfragment abgedruckt. Die Strategie verwendet einen PARALLELGRIPPER, der an einem ROBOTARM montiert ist und folglich über diesen bewegt werden kann. Angesteuert werden beide Geräte über jeweils ein ACTUATORINTERFACE, das passend konfiguriert ist (vgl. Abschn. 5.5). Für den ROBOTARM wird ein als *motions* bezeichnetes MOTIONINTERFACE verwendet, das sowohl PTP-Bewegungen als auch lineare Bewegungen erlaubt. Der Greifer wird über ein als *actions* bezeichnetes GRASPINGINTERFACE geöffnet bzw. geschlossen.

Generischer Ablauf

Der Ablauf ist einfach gehalten und startet an einer initialen Postion, an die der Greifer bereits über einen TRANSFERSERVICE bewegt wurde. In der Methode *align* wird der Greifer in Zeile 5 als erstes zu einer Vorposition

```
// Align with work-piece
public void align() throws RoboticsException {
  // move MCP to pre-grasping position
  Frame preGraspingPosition = graspingPosition.plus(preGraspingOffset);
  motions.ptp(preGraspingPosition).beginExecute();
  // preposition gripping fingers
  actions.preposition(initialWidth).execute();
  // move MCP to grasping position
  motions.lin(graspingPosition).execute();
}

// Grasp the work-piece
public void grasp() throws RoboticsException {
  // close gripper
  actions.grasp(graspingWidth).execute();
  // add a new connection between gripper and item
  addConnection(item, gripper);
  // find and remove the placement to the environment
  removePlacement(item);
}

// Retract with work-piece
public void retract() throws RoboticsException {
  // move MCP to post-grasping position
  Frame postGraspingPosition = graspingPosition.plus(postGraspingOffset);
  motions.lin(postGraspingPosition).beginExecute();
  // move MCP to the strategy's final position
  motions.ptp(finalPosition).beginExecute();
}
```

Listing 10.1: Generische Strategie für das Greifen eines Gegenstands. Die Strategie kann allgemein parametriert werden und dadurch die geometrischen Merkmale des Bauteils und des Greifers berücksichtigen.

(*preGraspingPosition*) bewegt. Dabei werden die Greiferbacken ebenfalls vorpositioniert (vgl. Zeile 7). Anschließend wird der Gegenstand in Zeile 9 mit einer linearen Bewegung angefahren. In der Methode *grasp* wird der Greifer geschlossen und das Modell der Roboterzelle aktualisiert. Zum Schluss wird der Greifer mit dem Gegenstand in einer definierten Richtung linear zur Nachposition (*postGraspingPosition*) in den *freien* Raum bewegt (vgl. Zeile 26). Von dort wird in Zeile 28 eine PTP-Bewegung zur finalen Position gestartet und die Strategie wird beendet.

Die Strategie ist generisch gestaltet, um alle Einflussfaktoren anzupassen. Dazu gehören die eigentliche Greifposition (*graspingPostion*), die Vor- und Nachposition des Greifens sowie die Anfangs- und Endposition der Strategie (*initialPosition* bzw. *finalPosition*). Auch die Greifparameter (z. B. die Öffnungsweite der Backen oder die Greifkraft) sind flexibel. Die Parameter eine Strategie können zudem unterschiedlich hergeleitet werden. Jeder *StrategyProvider* kann mehr als einen *Descriptor* akzeptieren, um die jeweilige Strategie zu instanziieren und zu parametrieren. So kann bspw. die Greifposition der oben vorgestellten Strategie auf unterschiedliche Arten hergeleitet werden:

- Direkte Angabe eines (eingelernten) Koordinatensystems (d. h. eine Instanz der Klasse FRAME) als Bestandteil des *Descriptors*.

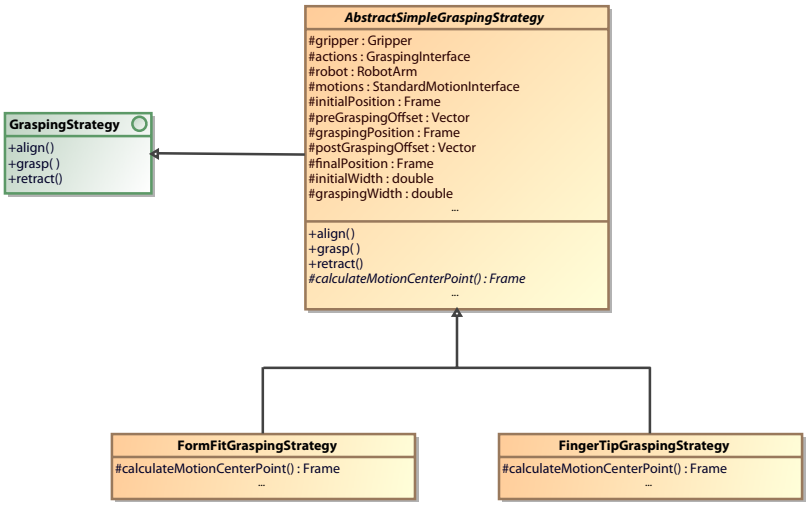


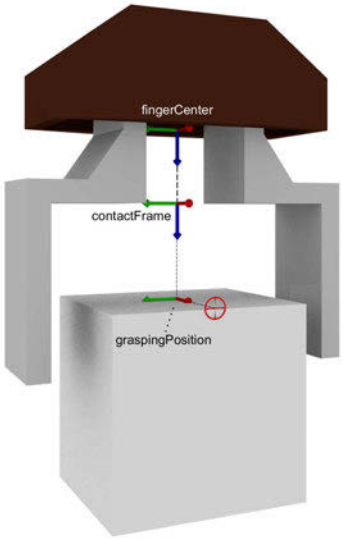
Abbildung 10.12: Die Greifstrategie kann unterschiedlich parametriert werden. Der generelle Ablauf ist davon jedoch nicht betroffen. Die Parameter sind als *Descriptor* über eine GRASPINGPROPERTY ein Teil des zu greifenden Gegenstands.

- Angabe einer Transformation (d. h. eine Instanz der Klasse TRANSFORMATION), die relativ zum Basiskoordinatensystems oder eines anderen Koordinatensystems des Gegenstands die erwünschte Greifposition beschreibt.

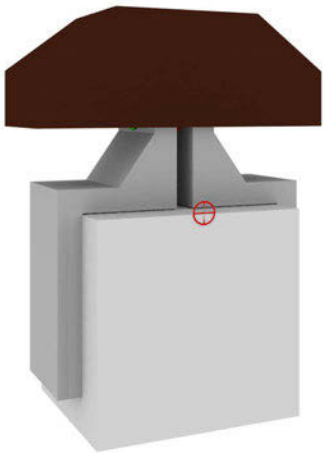
Letztendlich befindet sich die Strategie damit auf der unteren Abstraktionsebene der Arm-zentrierten Programmierung (vgl. Kap. 7). Dieses Niveau wird jedoch erst bei der Ausführung erreicht. Bis zur Instanziierung und Parametrierung der Strategie wird eine höhere Abstraktionsebene erhalten.

Dies zeigt sich auch bei der Spezialisierung der oben vorgestellten (abstrakten) Strategie. In Abbildung 10.12 wird gezeigt, wie die abstrakte Strategie in zwei konkrete Strategien verfeinert wird. Während die FORMFITGRASPINGSTRATEGY ein Bauteil formschlüssig greift, wird bei der FINGERTIPGRASPINGSTRATEGY der Gegenstand nur mit den Spitzen des Greifers aufgenommen. Der Unterschied zwischen den beiden Strategien ist schematisch in Abbildung 10.13 dargestellt. Dabei fällt auf, dass sich die Greifposition einmal auf der Oberseite und einmal auf der Unterseite des Werkstückes befindet. Die Position wird spezifisch für eine Strategie über den *Descriptor* und damit über das Bauteil definiert. Außerdem unterscheiden sich die beiden Strategien von der Wahl des Bezugpunktes am Greifer. Beim formschlüssigen Greifen wird ein Koordinatensystem auf der Kontaktfläche des Greifers gewählt. Beim Greifen mit den Fingerspitzen dagegen, ist der relevante Bezugspunkt ein Koordinatensysteme in der Mitte zwischen den Enden der Greiferbacken.

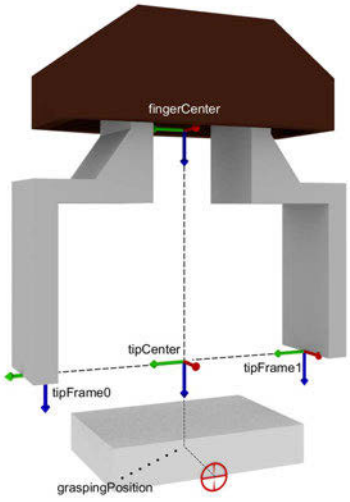
Beides sind geometrische Merkmale des verwendeten Greifers und müssen daher über den Greifer definiert bzw. berechnet werden. Die Strategie ist dadurch unabhängig von der Gestaltung des Greifers und seiner geometrischen Merkmale, berücksichtigt diese aber durch die objektorientierte Modellierung des Greifers. Bisher wurde der Bezugspunkt des Greifens in der abstrakten Strategie in Listing 10.1 nicht explizit aufgeführt. Der Bezugspunkt entspricht dem Motion Center Point (MCP) des Roboters bzw. seiner Bewegungen. Der MCP definiert den Punkt am Flansch, der am Ende einer Bewegung mit der Zielposition in Übereinstimmung ist. Das bedeutet, dass



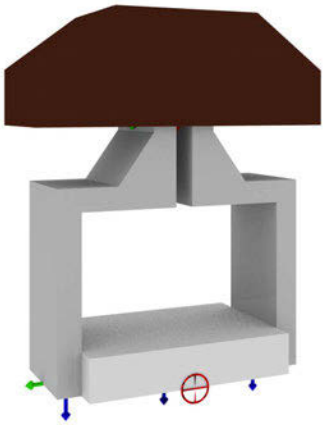
(a) Vor dem Greifen



(b) Nach dem Greifen



(c) Vor dem Greifen



(d) Nach dem Greifen

Abbildung 10.13: Gegenüberstellung unterschiedlicher Greifstrategien: In (a) und (b) wird der Gegenstand formschlüssig auf der Oberseite gegriffen. Dagegen wird der Gegenstand in (c) und (d) nur mit den Fingerspitzen gegriffen. Die beiden Greifstrategien sind schematisch mit den betreffenden Koordinatensystemen vor und nach dem Greifen dargestellt. Das Kreuz symbolisiert die Höhe der Greifposition, die über einen *Descriptor* sowohl strategie- als auch bauteilspezifisch konfiguriert wird.

```
1  // Calculate the MCP for the FingerTipGraspingStrategy
   protected Frame calculateMotionCenterPoint() throws RoboticsException {
     // get the center between the fingers
     Frame center = gripper.getFingerCenter();

6    // get the first finger's tip frame
     GrippingFinger finger0 = gripper.getBaseJaw(0).getMountedFinger();
     Frame tipFrame0 = finger0.getTipFrame();

     // get the second finger's tip frame
11   GrippingFinger finger1 = gripper.getBaseJaw(1).getMountedFinger();
     Frame tipFrame1 = finger1.getTipFrame();

     // check the displacement
     Vector t0 = center.getTransformationTo(tipFrame0).getTranslation();
16   Vector t1 = center.getTransformationTo(tipFrame1).getTranslation();

     if(t0.getZ() != t1.getZ())
       throw new RoboticsException("Finger length must be equal.")

21   // calculate and return the MCP
     Frame tipCenter = center.plus(0, 0, t0.getZ());
     return tipCenter;
   }
```

Listing 10.2: Berechnung des Bezugspunkts für Greifen mit den Fingerspitzen. Der Bezugspunkt kann über die geometrischen Merkmale des verwendeten Parallelgreifers berechnet werden (vgl. Abb. 10.13 (c)) und ist gleichzeitig der MCP der Roboterbewegungen.

beide in Abbildung 10.12 dargestellten Strategien, den MCP unterschiedlich berechnen. Die Berechnung erfolgt dabei jedoch immer relativ zum Greifer bzw. zu dessen geometrischen Merkmalen.

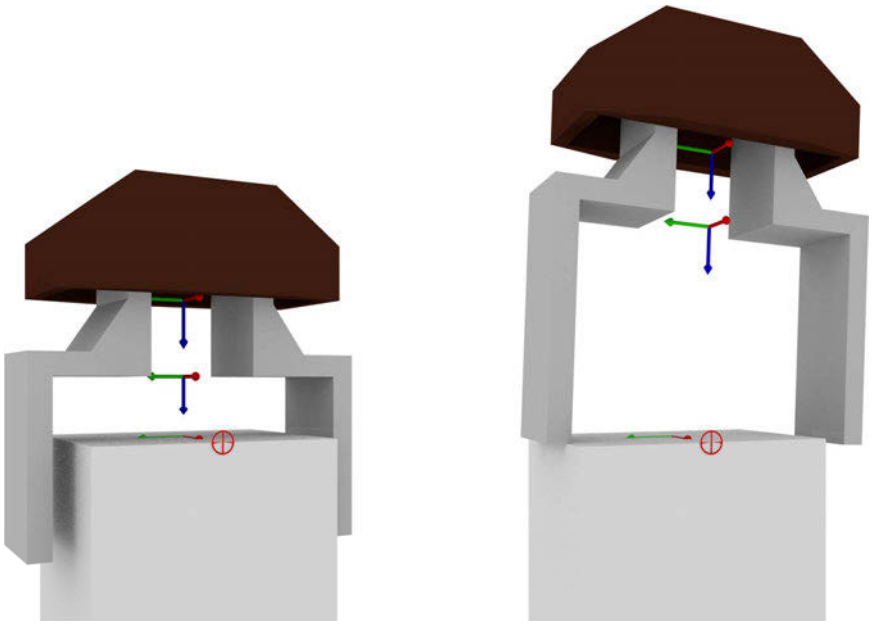
Fest umgreifen oder...

Da bei der FormFitGraspingStrategy formschlüssig gegriffen werden soll, müssen die Greiferbacken den Gegenstand möglichst gut umschließen. Dies kann im einfachsten Fall erreicht werden, indem man die beiden Greiferbacken und die zum Greifer zeigende Fläche über drei Koordinatensysteme modelliert (vgl. Abb. 10.13 (a) bzw. (b)). Beide Greiferbacken verfügen jeweils bereits über ein Koordinatensystem, das die Kontaktstelle mit dem Gegenstand approximiert. Diese beiden Koordinatensysteme werden bei der Einstellung der Öffnungsweite des Greifers bereits berücksichtigt (vgl. Lst. 10.1). Daher ist es ausreichend, den *contactFrame* des Greifers richtig zu positionieren. Folglich wird dieser Frame einfach als MCP verwendet.

... oder nur mit den Fingerspitzen

Für die FingerTipGraspingStrategy dagegen müssen die Spitzen der Greiferbacken richtig positioniert werden (vgl. Abb. 10.13 (c) bzw. (d)). Dazu wird der MCP in die Mitte zwischen beiden Greiferenden gelegt. In Listing 10.2 ist aufgeführt, wie sich der MCP aus dem Greifer bzw. den Greiferbacken berechnet. Jeder GrippingFinger verfügt über einen Frame, der einen Punkt an der Spitze repräsentiert (vgl. Zeilen 8 und 12). Daraus wird in Zeile 22 ein Punkt berechnet, der zwischen den beiden Spitzen liegt. Zur Sicherheit wird in den Zeilen 15 bis 19 überprüft, ob beide Greiferbacken gleich lang sind.

Die beiden Strategien haben gezeigt, dass es möglich ist, generische Programmfragmente für eine Werkzeugfähigkeit zu entwickeln. Die Parameter der Strategien sind entweder über eine GraspingProperty Teil des zu grei-



(a) Erfolgreiche Anfahrt (b) Kontakt einer Greiferbacke

Abbildung 10.14: Falls der zu greifende Gegenstand nicht an der vorgegebenen Position ist, kann der Greifversuch, insbesondere das Ausrichten des Greifers an dem Gegenstand, scheitern. Während in (a) ein erfolgreicher Versuch schematisch dargestellt ist, berührt in (b) ein Greiferbacke den Gegenstand und lässt den Versuch fehlschlagen.

fenden Gegenstands oder können aus diesem abgeleitet werden. Auch die geometrischen Merkmale des Greifers sind bei der Parametrierung relevant und ermöglichen es, dass eine Strategie ohne programmatische Änderung für unterschiedliche Greifer genutzt werden kann (vgl. Abschn. 11.4).

10.4.2 *Fehlertolerante Greifstrategien*

Die beiden vorgestellten einfachen Greifstrategien sind darauf ausgerichtet, dass sie einen Gegenstand an exakt der vorgegebenen Position greifen können. Der Unterschied zwischen den beiden Strategien liegt ausschließlich in der Wahl des Bezugspunktes bzw. des MCPs. Der generelle Ablauf ist identisch und basiert auf der Angabe einer Greifposition. Allerdings kann es bspw. aufgrund von Toleranzen oder Ungenauigkeiten (insbesondere die Absolutgenauigkeit von Robotern) dazu kommen, dass sich der Gegenstand nicht exakt an der vorgegebenen Position befindet. Diese Abweichungen können durch Sensorik festgestellt und evtl. korrigiert werden. Dementsprechend kann man durch den Einsatz von Sensorik eine robuste Strategie entwickeln, die ein fehlertolerantes Greifen ermöglicht.

Erkennen von Fehlern

Bisher wurden quaderförmige Gegenstände mithilfe eines Parallelgreifers und seiner beiden Greiferbacken aufgenommen. Dazu wird der Greifer bzw. dessen Greiferbacken am Gegenstand ausgerichtet (vgl. Abb. 10.14 (a)). Falls sich der zu greifende Gegenstand nicht an der vorgegebenen Position befindet, kann der Greifer den Gegenstand beim Anfahren bzw. Ausrichten mit seinen Greiferbacken berühren und dort hängen bleiben (vgl. Abb. 10.14 (b)). Dies würde im schlimmsten Fall zum Aufbau einer undefinierten Kraft füh-

ren und der Manipulator würde die Bewegung mit einem Fehler abbrechen. Ein definiertes Greifen des Gegenstands ist nicht möglich.

Ein weiteres Problem tritt auf, falls der Gegenstand zu weit von der vorgegebenen Position abweicht . Die bisher vorgestellten Strategien würden den Greifer trotzdem an die vorgegebene Position bewegen und versuchen, den Gegenstand dort zu greifen. Wenn die Greiferbacken den Gegenstand umschließen, kann der Vorgang erfolgreich erscheinen, da der Greifer einen Kontakt seiner Greiferbacken innerhalb der vorgegebenen Öffnungsweite registiert. Allerdings wurde die Greifposition nicht exakt erreicht und die aufgebaute geometrische Beziehung zwischen Greifer und Gegenstand ist fehlerhaft. Mithilfe kraftüberwachter Bewegungen kann der Greifer über einen physischen Kontakt mit dem Werkstück die optimale Greifposition erreichen. In (vgl. Abb. 10.14 (a)) fährt der Greifer bspw. bis zum Kontakt mit dem Bauteil nach unten und gleicht damit die Unsicherheit in einer Dimension aus. Demzufolge kann eine fehlertolerante Greifstrategie die optimale Greifposition trotz Ungenauigkeiten und Toleranzen erreichen.

Insgesamt lässt sich die Problemstellung auf das inverse Bolzen-in-Öffnung-Problem [270] (engl.: *Peg-in-Hole-Problem*) zurückführen. Dabei ist der Greifer mit seinen Greiferbacken die Öffnung, die in den Bolzen, d. h. den zu greifenden Gegenstand, einzuführen ist. In der entwickelten Greifstrategie werden kraftüberwachte Bewegungen verwendet, um die Greiferbacken erfolgreich in den Gegenstand einzuführen. Die Bewegungen werden über ein CompliantMotionInterface bereitgestellt. Für dieses MotionInterface gibt es bspw. eine Implementierung, welche die internen Drehmomentsensoren des KUKA Leichtbauroboters benutzt (vgl. Abschn. 6.4.3). Damit ist es möglich lineare Bewegungen zu definieren, die beim Überschreiten einer maximalen Kraft bzw. eines maximalen Drehmoments beendet werden, wobei sich der Roboter in der nachgiebigen Impedanzregelung befindet. Der Frame, in dem die Kräfte und Drehmomente gemessen werden, kann beliebig in Bezug zum Roboterflansch gewählt werden und wird als *Center of Compliance* bezeichnet.

Optimale Greifposition finden

In Abbildung 10.15 sind schematisch die Bewegungsrichtungen der Greifstrategie dargestellt. Der Greifer wird vom nachgiebigen Manipulator in Richtung der Z-Achse des Kontaktkoordinatensystems linear zu dem Gegenstand bewegt. Diese Bewegung endet, sobald eine definierte Kontaktkraft erreicht wird. Anschließend wird die Kraft abgebaut und der Greifer linear in Richtung der X-Achse bewegt. Nachdem diese Bewegung ebenfalls mit dem Erreichen einer Kontaktkraft endet, hat der Greifer die Greifposition erfolgreich eingenommen. Folglich wird die optimale Greifposition von zwei Kontaktflächen definiert.

Der oben geschilderte Ablauf betrachtet nur den erfolgreichen Fall, indem der Gegenstand zum Schluss optimal erreicht wird. Allerdings wurden in dieser Strategie auch mögliche Fehlerfälle definiert, die über die Sensorik erkannt werden. Daraufhin kann die Situation analysiert und die Strategie adaptiert werden. Dadurch entsteht eine robuste Strategie, die ein fehlertolerantes Greifen ermöglicht. Dementsprechend wird der Greifer zuerst in Richtung der Z-Achse des Kontaktkoordinatensystems angefahren. Die Bewegung ist kraftüberwacht, d. h. sie wird gestoppt, falls eine vorher definierte Kontaktkraft erreicht wurde. Sobald die Bewegung beendet ist, werden die Position und die gemessenen Kontaktkräfte ausgewertet:

- Falls keine Kraft aufgebaut wurde, bedeutet dies, dass kein Werkstück festgestellt wurde. Die Strategie wird nach einer Bewegung zum Ausgangspunkt beendet.

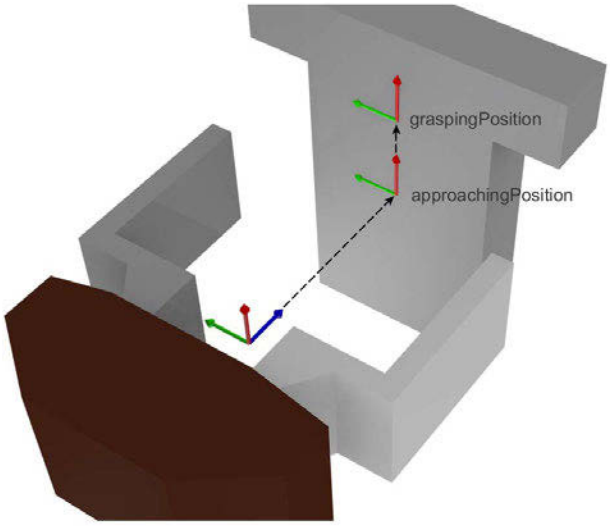


Abbildung 10.15: Die Ansicht zeigt schematisch die Bewegungsrichtungen der Greifstrategie. Zuerst wird der Gegenstand in Richtung der blauen Z-Achse bzw. der Stoßrichtung des Greifers angefahren. Daraufhin wird die finale Greifposition durch eine Bewegung in Richtung der roten X-Achse erreicht. Der Bezugpunkt ist dabei das Kontaktkoordinatensystem des Greifers. Der (nicht dargestellte) *Center of Compliance* kann dagegen an den Enden der Greiferbacken liegen.

- Andernfalls ist ein Werkstuck vorhanden. Daraufhin wird die aktuelle Position im Vergleich zum Zielpunkt der Bewegung ausgewertet:
  - Die beiden Positionen sind innerhalb einer definierten Toleranz identisch, d. h. das Werkstück wurde erfolgreich in Richtung der Z-Achse angefahren (vgl. Abb. 10.14 (a)). Die Anfahrt wird in die zweite Richtung fortgesetzt.
  - Falls die beiden Positionen stark abweichen, berührt eine der beiden Greiferbacken das Werkstück (vgl. Abb. 10.14 (b)). Auf Basis des gemessenen Drehmoments kann die entsprechende Seite identifiziert werden. Es folgt eine Anpassung der Greifposition und eine Bewegung an die Vorposition. Die Anfahrt kann erneut gestartet werden.

Im Erfolgsfall wurden die Greiferbacken am Werkstück vorbeigeführt und das Werkstück ist folglich von beiden Seiten umfasst. Es kann nun eine lineare Bewegung in Richtung der X-Achse erfolgen, um die finale Greifposition zu erreichen.

Das Programmfragment für das Ausrichten des Greifers in Richtung der Z-Achse am Werkstück ist in Listing 10.3 vereinfacht dargestellt. Als MCP wird der *contactFrame* des Greifers verwendet. Zuerst wird der Greifer in Zeile 5 an die Vorposition (*preGraspingPosition*) bewegt. Diese Position ist gegenüber der Greifposition (*graspingPosition*) sowohl um einen *Offset* in Richtung der X-Achse als auch der Z-Achse verschoben. Die Annäherungsposition (*appPosition*) ist dagegen nur um einen *Offset* in Richtung der X-Achse versetzt. Sie wird in Zeile 10 mit einer linearen Bewegung angefahren, die bei einer Kontaktkraft (vgl. *contactForceZ*) stoppt. Falls keine Kontaktkraft aufgebaut wurde, beendet sich die Strategie in Zeile 16 mit einer Ausnahme. Andernfalls wird in Zeile 19 überprüft, inwieweit die aktuelle Position

```
1  // Align with work-piece in Z-axis
   protected boolean alignInZ() throws RoboticsException {
     // move MCP to pre-grasping position
     Frame preGraspingPosition = graspingPosition.plus(offsetX,0,offsetZ);
     motions.ptp(preGraspingPosition).beginExecute();
6    // preposition gripping fingers
     actions.preposition(initialWidth).execute();
     // move MCP linear to approaching position
     Frame appPos = graspingPosition.plus(offsetX,0,0);
     boolean success = linToContact(appPos, toleranceZ, contactForceZ);
11   // no item was detected
     if(! success) {
       // move back to pre-grasping position
       motions.lin(preGraspingPosition).beginExecute();
       // throw exception
16     throw new RoboticsException("No item detected");
     }
     // check distance between contact frame & approaching postion
     success = checkPosition(gripper.getContactFrame(), appPos);
     // finger contact was detected
21   if(! success) {
       // adjust grasping position
       Vector adjustment = evaluateAndAdjustContact();
       graspingPosition = graspingPosition.plus(adjustment);
       // move back to pre-grasping position
26     motions.lin(preGraspingPosition).beginExecute();
       // indicate that a retry is possible
       return false;
     }
     // alignment in Z was successful
31   return true;
   }
```

Listing 10.3: Das Werkstück wird in Richtung der Z-Achse angefahren, wobei die Kontaktkräfte überwacht werden. Falls es zu einem ungewünschten Kontakt zwischen dem Werkstück und den Greiferbacken kommt, wird dies festgestellt und kann im folgenden Versuch korrigiert werden.

mit der Zielposition übereinstimmt. Bei einer Abweichung wird der Kontakt analysiert und die Greifposition wird angepasst (vgl. Zeile 23). Die Strategie ist dabei flexibel konfigurierbar, um in unterschiedlichen Situationen verwendbar zu sein.

10.5 VERWANDTE ARBEITEN

Huckaby und Christensen [131] stellen eine Taxonomie vor, um Montageaufgaben mithilfe eines Roboters zu beschreiben. Sie definieren dafür eine Menge von *skill primitives*, d. h. elementaren Fähigkeiten, die eine atomare Roboter- oder Werkzeugaktion darstellen. Ein Beispiel für eine solche elementare Fähigkeit ist *Transport*, um eine Bewegung zu spezifizieren. Dabei wird vollständig von der konkreten Umsetzung oder dem zugrunde liegenden Algorithmus abstrahiert. Über *constraints*, d. h. Nebenbedingungen, kann jedoch ein *skill primitive* parametrisiert werden. Das Ziel der Taxonomie ist es, die Programmierung eines Roboters für Montageaufgaben sowohl zu vereinheitlichen als auch zu vereinfachen. Darüber hinaus soll

das Wissen, um bestimmte Aufgaben umzusetzen, geteilt und wiederverwendet werden können. Um einen Montageablauf zu beschreiben werden Sequenzdiagramme verwendet. Jedes *skill primitive* stellt dabei eine Nachricht dar, die vom ausführenden Objekt, d. h. dem Roboter, zu der betroffenen Station (z. B. das Teilelager) gesendet wird. Die *constraints* werden hierbei nicht explizit erwähnt. Daneben stellen Huckaby et al. [133] vor, wie die *skill primitive* zur automatischen Planung einer Montageaufgabe verwendet werden können. Huckaby [129] schlägt mit der allgemeinen Taxonomie für Montageaufgaben einen interessanten Ansatz vor. Einige der *skill primitives* haben eine ähnliche Granularität und Bedeutung mit den Methoden eines *skill-level services* (z. B. *Transport*, *Align*, *Grasp*). Die Ausführung eines *skill primitives* ist nur abhängig von den *constraints*, die bspw. Koordinatensysteme beschreiben. Eine Modellierung der Bauteile findet nicht statt, d. h. es ist nicht möglich *skill primitives* auf Basis der Bauteile zu spezifizieren. Daher ist in der Spezifikation einer Montageaufgabe implizit die Geometrie der Bauteile kodiert, was die geforderten Wissenstransfer erschwert. Der in dieser Dissertation beschriebene Ansatz kann dies durch die unterschiedlichen Abstraktionsebenen und die Bauteilorientierung mit den Strategien erreichen. Die bei Huckaby et al. [133] vorgeschlagene automatische Planung kann unter Formalisierung der Vor- und Nachbedingungen auf die fähigkeitenzentrierten Dienste übertragen werden.

Malec et al. [172] schlagen eine wissensbasierte Rekonfiguration von roboterbasierten Automatisierungssystemen vor. Das Wissen des Systems ist in Form einer Ontologie zentral gespeichert und enthält Informationen über die verfügbaren *Skills*, über Sensoren und Aktuatoren sowie über die Bauteile. Die Aufgabe wird von einem Anwender über eine graphische Benutzerschnittstelle definiert und parametrisiert. Anschließend wird die Aufgabe auf einem zentralen *Skill Server* analysiert und validiert. Gegebenenfalls wird die Aufgabe dort re-parametrisiert bzw. re-konfiguriert. Der Ansatz zeigt sehr gut, dass die Aufgaben mit Unterstützung einer Ontologie von einem Anwender zusammengestellt werden können. Jedoch berücksichtigt dieser Ansatz keine Strukturierung der Automatisierungssoftware und auch keine Adaption zur Laufzeit.

Eine Weiterentwicklung des obigen Ansatzes wird von Stenmark und Malec [252] vorgestellt. Zentrales Element dort ist das Knowledge Integration Framework (KIF), das eine dynamisches Wissensdatenbank (in Form von Ontologien) darstellt. Das Wissen ist dort hierarchisch gegliedert, d. h. komplexe Zusammenhänge lassen sich aus einfacheren Informationen ableiten. Das KIF bietet zudem eine Menge von Diensten an, um das Wissen verfügbar zu machen. Ein in [252] vorgestellter Dienst leitet Roboteraufgaben aus natürlichsprachlichen Sätzen ab (z. B. *Robot, please put the camera socket on the fixture*). Dabei wird der Satz analysiert und die notwendige Fähigkeit (*put*) und deren Argumente (*robot*, *camera socket*, *fixture*) ermittelt. In diesem Ansatz werden die geometrischen Merkmale der Bauteile (als Teil einer Ontologie) modelliert und für die Spezifikation der Aufgaben verwendet. Zur Ausführung werden die *Skills* in eine native Roboterprogrammiersprache (hier: RAPID von ABB) übersetzt (vgl. [253]). Für sensorgeführte Bewegungen existiert noch ein zweites externes System. Koordiniert werden die beiden Systeme, d. h. die Robotersteuerung und das externe System, über ein graphisches Tool zur Modellierung und Ausführung von StateCharts. Nach der Generierung der *Skills* geht sämtliches Wissen über die Bauteile verloren. Dies machte eine Adaption zur Laufzeit schwierig.

Pfrommer et al. [216] schlagen einen Ansatz vor, der das bekannte Datenmodell *Product, Process & Resource* (PPR) um Fähigkeiten erweitert. Ein *Process* ist für sie bspw. der Transport von Bauteilen, Schneiden oder Schweißen. *Resources* sind Automatisierungsgeräte wie z. B. ein Roboter oder eine Werkzeugmaschine. Als *Product* bezeichnen sie die Bauteile bzw. Baugruppen des Produktionssystems. In ihrem Ansatz bezeichnen sie einen *Skill* bzw. eine Fähigkeit als eine Kombination aus *Process* und *Resource*. Ein *Task* dagegen ist eine Kombination aus *Skill* und *Product*. Damit wird eine konkrete Aufgabe beschrieben, d. h. die Bearbeitung eines Produkts auf einer spezifischen Maschine. Dieses Konzept wird vor allem dafür gedacht, um automatisch Pläne zu erstellen. Die Umsetzung der *Skills* und *Tasks* ist auf den Maschinen weitgehend vorgegeben. Zudem werden Roboter als Teil einer größeren Anlage betrachtet. Die Flexibilität einer Roboterzelle und deren interner Aufbau ist daher nicht der Fokus dieses Ansatzes.

11 EVALUIERUNG AM BEISPIEL DER FACTORY 2020

Das Konzept einer serviceorientierten Automatisierung von Roboterzellen wurde anhand der in Kapitel 7 vorgestellten Fallstudie – der Factory 2020 – evaluiert. Durch die Benutzung mehrerer Werkzeuge, die Kooperation zweier Roboter und den Einsatz von sensorgeführten Bewegungen vereinigt die Fallstudie eine Reihe von Herausforderungen. Die Fallstudie wurde voolständig als serviceorientierte Automatisierungssoftware mithilfe von OSGi realisiert. Des weiteren wurde die Fallstudie und ihr Automatisierungssoftware modifiziert, um die Flexibilität aber auch die Wiederverwendbarkeit der Services und Strategien zu evaluieren. Dabei konnte gezeigt werden, dass die service-orientierte Modellierung, die im Rahmen dieser Dissertation entwickelt wurde, unterschiedliche Variationsmöglichkeiten einer Roboterzelle elegant und ohne hohen Aufwand unterstützt.

Insgesamt wurden die in Kapitel 7 aufgelisteten Variationspunkte der Factory 2020 betrachtet. Zuerst wird daher in Abschnitt 11.1 der Fertigungsprozess variiert. Dadurch soll die Roboterzelle Bauteile ohne Verschrauben montieren bzw. bereits vormontierte Werkstücke verschrauben. Eine Betrachtung der Factory 2020 mit einer geänderten Topologie der Roboterzelle (d. h. nur ein Leichtbauroboter und dafür eine stationäre Spannvorrichtung) wird in Abschnitt 11.2 gegeben. Eine mögliche Variation der Werkstücke wird in Abschnitt 11.3 beschrieben. Anschließend wird in Abschnitt 11.4 ein Austausch eines bzw. beider Roboter oder Greifer betrachtet. Dabei wird auf die notwendigen Auswirkungen bzw. Anpassungen eingegangen. Zum Abschluss wird in Abschnitt 11.5 auf grafische Beschreibungssprachen und deren Anwendung in einer serviceorientierten Roboterzelle eingegangen.

11.1 VARIATION DES FERTIGUNGSPROZESSES

Bei der Variation des Fertigungsprozesses ändert sich nichts an der physikalischen Struktur, d. h. der Konfiguration der Roboter, Werkzeuge und Hilfseinrichtungen. Stattdessen werden auf der obersten Ebene die Fertigungsschritte variiert. Daher kann zu jeder Variation des Fertigungsprozesses ein neuer symbolischer Funktionsplan gemäß VDI-Richtlinie 2860 (vgl. Abschn. 9.1) angegeben werden. Daraufhin wird eine neue logische Struktur der Roboterzelle entwickelt unter Berücksichtigung und Verwendung bereits vorhandener logischer Einheiten.

Anschließend kann der Prozess als Abfolge von Service-Aufrufen realisiert werden. Eventuell müssen neue Serviceimplementierungen konfiguriert und instanziiert werden. Bevor der Prozess vollständig lauffähig ist, müssen die Strategien und Bewegungen angepasst bzw. über das Handhabungsobjekt neu konfiguriert werden. Der Aufwand für die Variation des Fertigungsprozesses wird anhand zweier Beispiele illustriert. Dazu wird in Abschnitt 11.1.1 eine Montage der Bauteile ohne Verschrauben betrachtet. Danach wird Abschnitt 11.1.2 das Verschrauben bereits vormontierter Werkstücke untersucht.

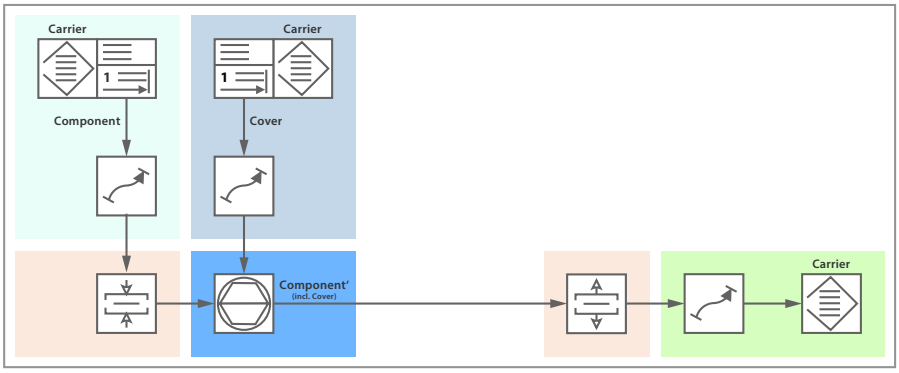


Abbildung 11.1: Der Prozess *Montage ohne Verschrauben* der Factory 2020 als symbolischer Funktionsplan gemäß VDI-Richtlinie 2860 [269].

11.1.1 *Montage ohne Verschrauben*

Die Factory 2020 soll in einem ersten Schritt so ergänzt werden, dass zusätzlich nur die Montage ohne ein anschließendes Verschrauben stattfinden kann. Der symbolische Funktionsplan dafür ist in Abbildung 11.1 dargestellt. Er entspricht von der grundsätzlichen Struktur dem ursprünglichen Funktionsplan (vgl. Abb. 9.2). Lediglich die Fertigungsfunktion des *Fügens (durch Schrauben)* und die Handhabungsfunktionen für die Hilfsstoffe (d. h. die Schrauben) entfallen. Alle weiteren Funktionen bleiben im Vergleich zu Abschnitt 9.1 erhalten. Das bedeutet, dass das Basisbauteil (COMPONENT) nach dem *Zuteilen* und *Weitergeben* durch *Spannen* gesichert wird. Der Deckel wird ebenfalls *zugeteilt* und *weitergegeben*. Er kann auf das zuvor gesicherte Basisbauteil gefügt werden. Anschließend wird das montierte Bauteil (COMPONENT') durch *Entspannen* gelöst und durch die Funktion *Weitergeben* zum Werkstückträger transportiert, der als *geordneter Speicher* symbolisiert ist. Das Symbol repräsentiert auch die Funktion des geordneten Speicherns, d. h. das fertige Bauteil wird in den Werkstückträger gelegt.

Flexibilität durch die Rekonfiguration logischer Einheiten

Aus dem oben beschriebenen symbolischen Funktionsplan können – analog zu dem in Abschnitt 9.1 beschrieben Vorgehen – logische Einheiten definiert werden. Dabei sollten sich diese Einheiten, soweit es möglich ist, auf bereits bestehende logische Einheiten abstützen. Dadurch kann eine höhere Wiederverwendung bereits vorhandener Einheiten bzw. Systembausteine erreicht werden. Das bedeutet jedoch auch, dass sich die logische Systemstruktur der Roboterzelle ändern kann. Sie wird folglich je nach Bedarf aus bestehenden Einheiten bzw. Subsystemen neu gebildet, um den aktuellen Fertigungsprozess abzubilden. Durch die Rekonfiguration logischer Einheiten entsteht eine hohe Flexibilität der Automatisierungssoftware, was zu einer hohen Flexibilität der möglichen Fertigungs- und Bearbeitungsprozess der Roboterzelle führt.

Der hierarchische Aufbau der Factory 2020 für den oben beschriebenen Funktionsplan ist in Abbildung 11.2 als Blockdefinitionsdiagramm dargestellt. Der zentrale Systembaustein wird weiterhin als ASSEMBLYSYSTEM bezeichnet. Über eine Schnittstelle kann der neue Automatisierungsprozess (vgl. *assembleOnly*) als Dienst angeboten werden. Die äußere Struktur des ASSEMBLYSYSTEMS bleibt im Vergleich zur ursprünglichen Modellierung (vgl. Abb. 9.3) erhalten, d. h. der Systembaustein verfügt weiterhin über zwei externe Ports, die den Standort der beiden Werkstückträger symbolisieren.

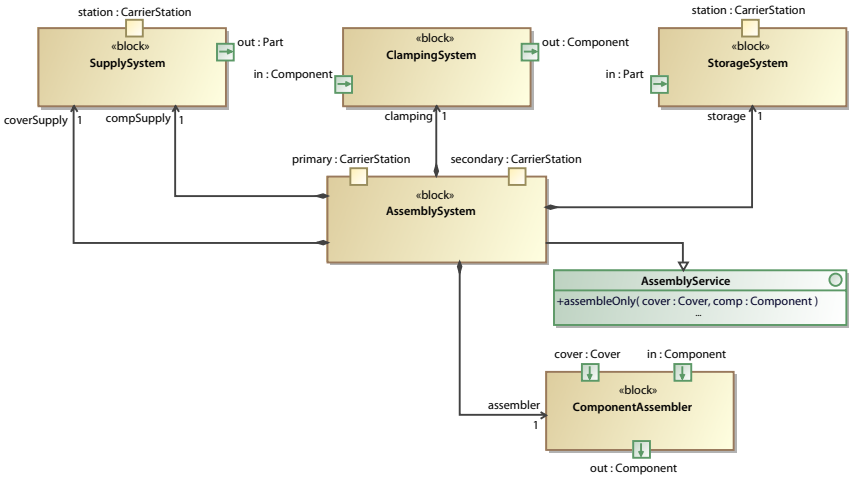


Abbildung 11.2: Logischer Aufbau einer Roboterzelle am Beispiel der Factory 2020: Durch diese Struktur der logischen Systembausteine wird eine Montage der beiden Bauteile ohne anschließendes Verschrauben realisiert. Daher wird im Gegensatz zu Abbildung 9.3 kein logischer Systembaustein für das Verschrauben benötigt.

Im Vergleich zum ursprünglichen Aufbau besteht das neue AssemblySystem nur aus fünf Systembausteinen, welche jeweils eine logische Einheit repräsentieren. Das StorageSystem wurde ebenso wie das SupplySystem ohne Änderung übernommen. Daneben sind die aus der ursprünglichen Aufteilung bekannten Systembausteine für das Einspannen des Basisbauteils und das Montieren der Bauteile vorgesehen (d. h. das ClampingSystem und der ComponentAssembler). Damit werden alle farblich markierten logischen Einheiten, die im symbolischen Funktionsplan (vgl. Abb. 11.1) identifiziert wurden, modelliert. Im Vergleich zum ursprünglichen Aufbau fällt demnach das FasteningSystem weg.

Im internen Blockdiagramm in Abbildung 11.3 sind die Ports der inneren Systembausteine und die beiden *Full Ports* des Gesamtsystem eingezeichnet. Analog zur ursprünglichen Konfiguration werden die *Full Ports* der Montagezelle mit jeweils einem SupplySystem verbunden, der eine Station und damit einen Werkstückträger bedient. Gleiches gilt für das StorageSystem. Wenn man das interne Blockdiagramm mit dem ursprünglichen Diagramm aus Abbildung 9.4 vergleicht, fällt auf, dass sich der *Item Flow* für ein Component nach dem ComponentAssembler unterscheidet. Ursprünglich wird das Bauteil an das FasteningSystem weitergegeben, während in der neuen Konfiguration der Weg direkt zum StorageSystem führt. Das bedeutet, dass der Übergang zwischen diesen beiden Systembausteinen überprüft werden muss.

Die Services und ihre Zuordnung zu den Systembausteinen entspricht im neu konfigurierten AssemblySystem der ursprünglichen Modellierung, wie sie in Abbildung 9.6 dargestellt wurde. Daraus kann der neue Ablauf abgeleitet und ebenfalls als Aktivitätsdiagramm modelliert werden (vgl. Abb. 11.4). Dabei werden der Deckel und das Basisbauteil nebenläufig zugeteilt. Nach der Anlieferung der beiden Bauteile bzw. nach dem Einspannen des Basisbauteils wird der Ablauf durch die Aktivität des Fügens wieder synchronisiert. Nach dem Fügen wird der Fertigungsprozess sequentiell mit dem *Entspannen* und dem *Weitergeben* fortgesetzt bis das vormontierte Bauteil schließlich in den Werkstückträger zurückgelegt wurde.

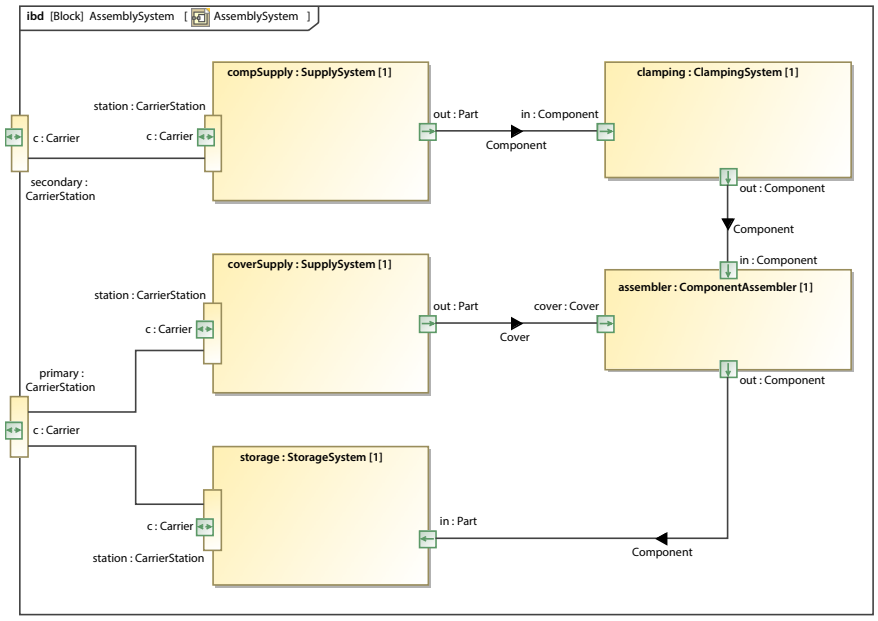


Abbildung 11.3: Logischer Aufbau einer Roboterzelle am Beispiel der Factory 2020. Das Interne Blockdiagramm zeigt, wie die einzelnen Systembausteine für eine Montage der beiden Bauteile ohne anschließendes Verschrauben zusammenhängen und welche Handhabungsobjekte zwischen diesen ausgetauscht werden.

Die Zustände an den Eingabe- und Ausgabepins, die nur Bewegungsabläufe beschreiben, sind identisch zu dem ursprünglichen Ablauf in Abbildung 9.8. Allerdings unterscheiden sich im Vergleich dazu die Bearbeitungszustände des Bauteils (COMPONENT) in Abbildung 11.4. Während es bei der Anlieferung, d. h. bei Beginn, auch den Zustand *unprocessed* hat, verlässt es die Roboterzelle im Zustand *cover attached* und nicht wie ursprünglich im Zustand *cover bolted*. Durch den Bearbeitungsstand des Bauteils lassen sich somit die Unterschiede zwischen den Abläufen gut erkennen.

Aus der geänderten Struktur der Systembausteine und dem neuen Ablauf ergeben sich Erweiterung bzw. Anpassungen, die an der Zelle durchgeführt werden mussten:

- Eine Erweiterung des ASSEMBLYSYSTEM, um die neue Konfiguration und die erweitere Serviceschnittstelle realisieren zu können.
- Überprüfung und ggf. Erweiterung der Pfade, um nach dem Fügen (*attach*) das Lösen des Bauteils (*unclamp*) durchführen zu können.
- Überprüfung und ggf. Anpassung der *GraspingStrategy* bzw. *PlacingStrategy* für den Transport des ausschließlich vormontierten Bauteils in den Werkstückträger. Aufgrund der fehlenden Fixierung durch die Schrauben (vgl. Bearbeitungszustand *cover attached*) ist eine Überprüfung angebracht.

Durch die Variation des Fertigungsprozesses zeigt sich, dass die logischen Systembausteine und ihre softwaretechnische Abbildung durch Services eine einfache Möglichkeit darstellen, eine Roboterzelle schnell und flexibel umzukonfigurieren. Die Modellierung zeigt zudem die Stellen auf, an denen Erweiterungen, Überprüfungen und Anpassungen ggf. notwendig sind.

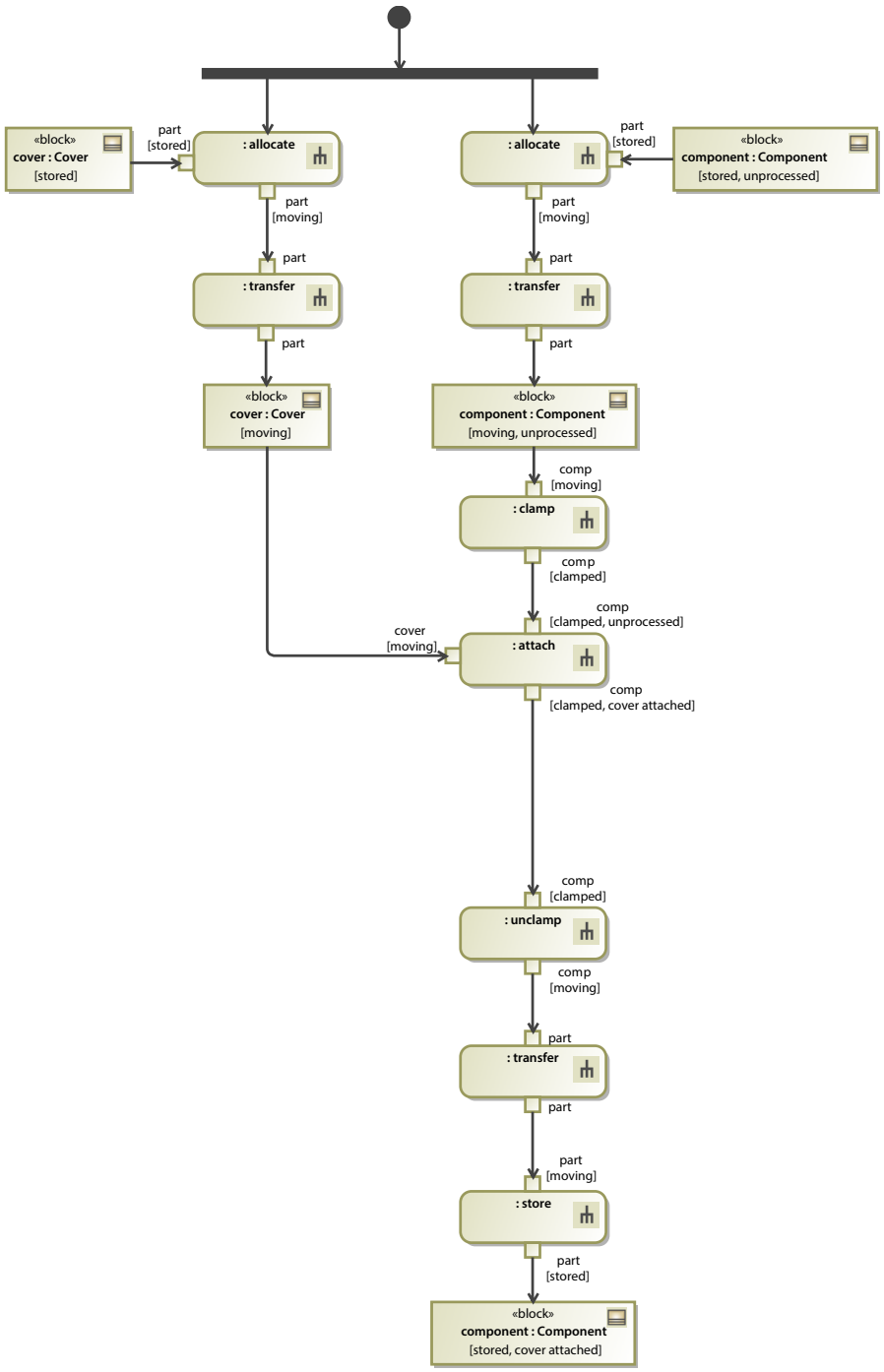


Abbildung 11.4: Das Aktivitätsdiagramm zeigt den Ablauf der Montage eines Bauteils (ohne Verschrauben). Die Anlieferung der beiden Teile erfolgt dabei nebenläufig. Vor der Montage wird der Prozess synchronisiert und läuft anschließend sequentiell ab. In der Lücke zwischen dem *Fügen* und dem *Entspannen* befindet sich im ursprünglichen Ablauf das Verschrauben der beiden Bauteile. An den Ein- und Ausgabepins werden zudem signifikante Zustandsänderungen der Handhabungsobjekte dargestellt.

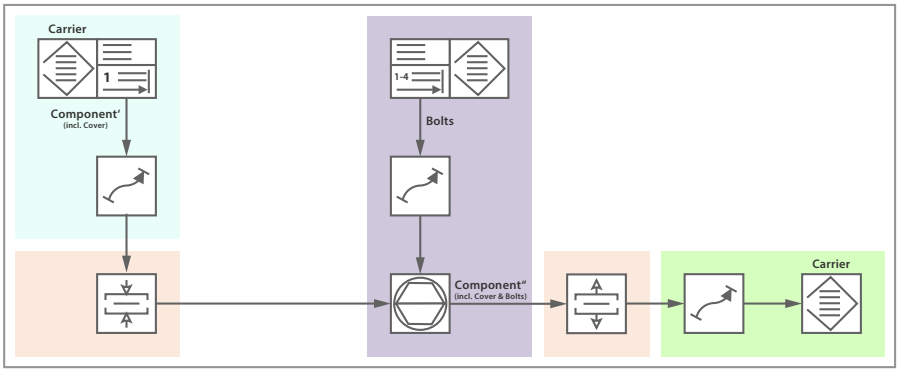


Abbildung 11.5: Der Prozess *Verschrauben vormontierter Werkstücke* der Factory 2020 als symbolischer Funktionsplan gemäß VDI-Richtlinie 2860 [269].

11.1.2 *Verschrauben vormontierter Werkstücke*

Anschließend wird die Factory 2020 in einem zweiten Schritt verändert, sodass zusätzlich ein Verschrauben bereits vormontierter Werkstücke stattfinden kann. Der symbolische Funktionsplan dafür ist in Abbildung 11.5 dargestellt und entspricht von der Struktur dem ursprünglichen Funktionsplan (vgl. Abb. 9.2). Allerdings entfallen die Handhabungsfunktionen für das *Zuteilen* und *Weitergeben* des Deckels, da dieser nicht mehr benötigt wird. Das bedeutet auch, dass nur ein Werkstückträger benötigt wird. Zudem kann der Fertigungsschritt des *Fügens* beider Bauteile ebenso entfallen. Alle weiteren Funktionen bleiben im Vergleich zu Abschnitt 9.1 erhalten.

Allerdings ändert sich die noch vorhandenen Handhabungsfunktionen für das *Zuteilen* und *Weitergeben* des Bauteils. Statt eines Basisbauteils (d. h. nur ein Unterteil) muss nun ein vormontiertes Bauteil (Component'), das bereits einen Deckel besitzt, *zugeteilt* und *weitergegeben* werden. Dieses wird vor dem Schraubvorgang ebenfalls durch *Spannen* gesichert. Anschließend findet mit dem *Fügen (durch Schrauben)* die eigentliche Fertigungsfunktion statt, um beide Bauteile dauerhaft zu verbinden. Die nötigen Schrauben werden wie im ursprünglichen Funktionsplan (vgl. Abb. 9.2) zuvor durch mehrere Handhabungsfunktionen dem Fertigungsschritt zugeteilt. Nach dem Schraubvorgang wird das Bauteil durch die Funktion *Entspannen* gelöst und durch *Weitergeben* wieder zum Werkstückträger transportiert und durch *geordnetes Speichern* in den Werkstückträger gelegt.

Aus dem oben beschriebenen symbolischen Funktionsplan können logische Einheiten definiert werden. Das bedeutet, dass der neue Automatisierungsprozess eine weitere Rekonfiguration der logischen Einheiten erfordert. Im Vergleich zur ursprünglichen Modellierung setzt sich das neue AssemblySystem aus vier Systembausteinen zusammen. Das bereits bekannte SupplySystem wird wieder verwendet. Allerdings wird dieses Mal nur eine Instanz für die Anlieferung der vormontierten Bauteile aus einem Werkstückträger (d. h. einem Carrier) benötigt. Das StorageSystem wurde dagegen ohne Änderung übernommen. Daneben sind die aus der ursprünglichen Aufteilung bekannten Systembausteine für das Einspannen des Bauteils und das Verschrauben vorgesehen (d. h. das ClampingSystem und das FasteningSystem). Dagegen wird der ComponentAssembler nicht mehr benötigt. Folglich sind alle farblich markierten logischen Einheiten modelliert, die im symbolischen Funktionsplan (vgl. Abb. 11.5) identifiziert wurden.

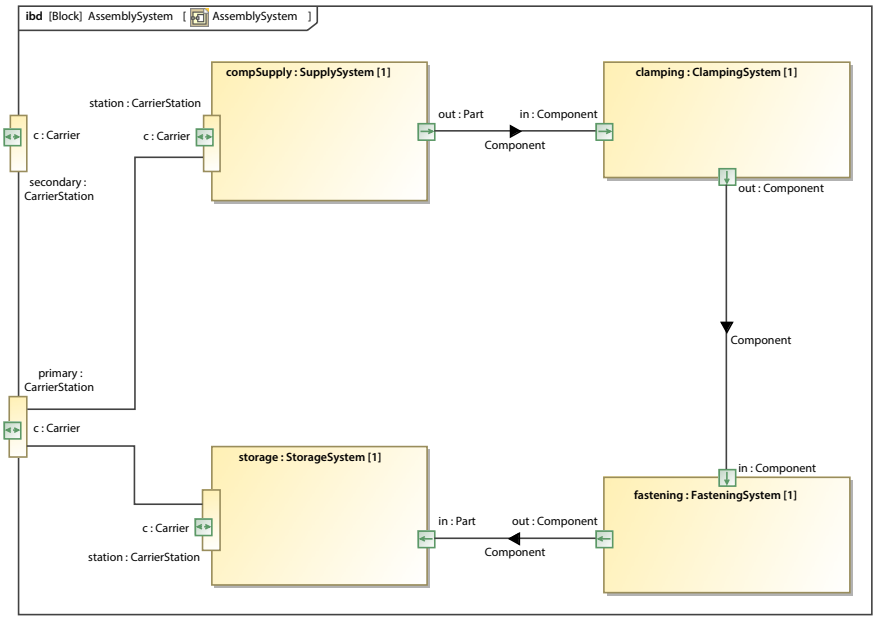


Abbildung 11.6: Durch diese Struktur der logischen Systembausteine wird ein Verschrauben bereits vormontierter Bauteile realisiert. Daher wird im Gegensatz zu Abbildung 9.4 kein logischer Systembaustein für das Fügen (durch Zusammensetzen) bzw. für die Zulieferung der Cover benötigt.

Im internen Blockdiagramm in Abbildung 11.6 sind die Ports der inneren Systembausteine und die beiden *Full Ports* des Gesamtsystem eingezeichnet. Im Gegensatz zur ursprünglichen Konfiguration wird nur der primäre *Full Port* der Montagezelle mit einem SupplySystem verbunden, um den Fertigungsprozess mit vormontierten Bauteilen zu versorgen. Dabei ist zu beachten, dass nicht das ursprüngliche SupplySystem verwendet wurde, das Bauteile (Component) aus einem Werkstückträger der sekundären Station anliefert. Stattdessen wird eine neue Instanz des SupplySystems benötigt, das die primäre Station bedient.

Der Fluss der Handhabungsobjekte durch das System ist ebenfalls dargestellt. Das vormontierte Bauteil (vom Typ Component) wird von dem neuen SupplySystem an das ClampingSystem weitergegeben. Von dort gelangt es zum FasteningSystem und wird dort verschraubt. Anschließend führt der Weg des nun verschraubten Bauteils zum StorageSystem und von dort zurück in den Werkstückträger. Wenn man das interne Blockdiagramm in Abbildung 11.6 mit dem ursprünglichen Diagramm in Abbildung 9.4 vergleicht, fällt auf, dass sich der *Item Flow* für ein Component nach dem ClampingSystem unterscheidet. Ursprünglich wird das Bauteil an den ComponentAssembler weitergegeben, während in der neuen Konfiguration der Weg direkt zum FasteningSystem führt. Das bedeutet, dass der Übergang zwischen diesen beiden Systembausteinen überprüft werden muss. Zudem fällt jeglicher *Item Flow* für Cover weg.

Feststellen von Änderungen

Eine gravierende Änderung stellt die neue Verknüpfung der externen Schnittstellen des AssemblySystems mit den internen Systembausteinen dar. Während die sekundäre Station in dieser Konfiguration nicht mehr benötigt wird, ist die primäre Station mit der neuen Instanz eines SupplySystems verbunden. Daher ist es notwendig ein neues SupplySystem zu instanziie-

ren und der serviceorientierten Roboterzelle hinzuzufügen. Dieses ist dafür zuständig, ein Bauteil bzw. ein COMPONENT aus einem an der primären CARRIERSTATION befindlichen Werkstückträger anzuliefern. Der Fertigungsprozess läuft im Gegensatz zu den beiden anderen Prozesses komplett sequentiell ab. Nach der Anlieferung bzw. dem Einspannen des vormontierten Bauteils wird das Bauteil verschraubt. Danach wird der Fertigungsprozess mit dem *Entspannen* und dem *Weitergeben* fortgesetzt bis das nun verschraubte Bauteil schließlich in den Werkstückträger zurückgelegt wurde.

Aus der geänderten Struktur der Systembausteine und dem neuen Ablauf ergeben sich Erweiterung bzw. Anpassungen, die an der Zelle durchgeführt werden mussten:

- Instanziierung eines neuen SUPPLYSYSTEMS und Registrierung eines neuen Service. Das System ist für die Anlieferung eines vormontierten Bauteils aus einem Werkstückträger an der primären CARRIERSTATION zuständig.
- Definition bzw. Parametrierung einer neuen *GraspingStrategy* bzw. einer neuen *PlacingStrategy* für die Aufnahme des vormontierten Bauteils aus dem Werkstückträger und dem Einspannen des Bauteils. Dazu muss der zweite Roboter als Spannvorrichtung dienen.
- Eine Erweiterung des ASSEMBLYSYSTEM, um die neue Konfiguration und die erweitere Serviceschnittstelle realisieren zu können.
- Überprüfung und ggf. Erweiterung der Pfade, um nach dem Einspannen (*clamp*) das Verschrauben (*fasten*) durchführen zu können.

Durch eine weitere Variation des Fertigungsprozesses zeigt sich, dass die logischen Systembausteine und ihre softwaretechnische Abbildung durch Services stabil sind und zudem wiederverwendet werden können. Die Modellierung zeigt außerdem die Stellen auf, an denen ein neuer Systembaustein notwendig ist.

11.2 VARIATION DER TOPOLOGIE

Um die Unabhängigkeit der logischen Einheiten von der Topologie der Roboterzelle zu demonstrieren, wurde eine geänderte Factory 2020 definiert. Bei dieser Variante besteht die Roboterzelle aus nur einem Leichtbauroboter, der auf einer Werkbank montiert ist. Der Roboter ist mit einem elektrischen Parallelgreifer und dem elektrischen Schraubsystem ausgestattet. Zusätzlich verfügt die Zelle über eine stationäre Spannvorrichtung, die als fest auf der Werkbank montierter Parallelgreifer ausgeführt ist. Die beiden Werkstückträger und das Depot für Schrauben befinden sich wie bei der kooperativen Montage ebenfalls auf der Werkbank.

Montage mit einem Roboter

Die topologische Struktur dieser Roboterzelle ist in Abbildung 11.7 als internes Blockdiagramm dargestellt. Die Werkbank hat weiterhin fünf mechanische Schnittstellen, an denen der Roboter, die Stationen der Werkstückträger, das Schraubenmagazin und die Spannvorrichtung angebracht sind. Am Flansch des Roboters ist der Greifer und an diesem das elektrische Schraubsystem befestigt. Die neue topologische und geometrische Struktur kann mit der *Robotics API* ebenfalls vollständig in Software abgebildet werden.

Wie bei der kooperativen Montage erfolgt eine Abbildung von logischen Systembausteinen (vgl. Abb. 9.3) auf die realen Entitäten der Roboterzelle

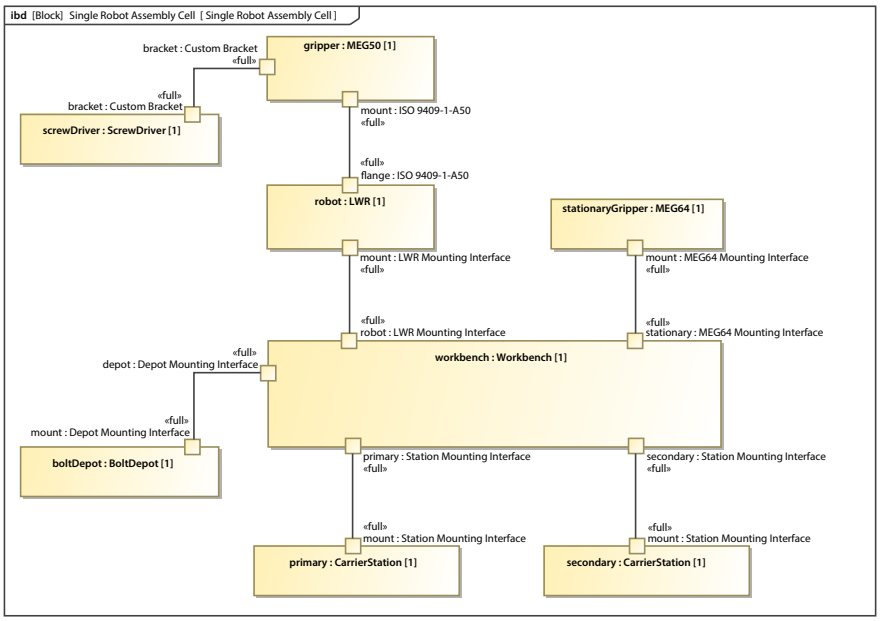


Abbildung 11.7: Das interne Blockdiagramm stellt den topologischen Aufbau der vereinfachten Factory 2020 dar. Diese besteht im Vergleich zur kooperativen Montage nur aus einem Leichtbauroboter. Um das Bauteil zu fixieren, wird eine stationäre Spannvorrichtung verwendet.

(vgl. Abb. 11.7). Dazu muss für jeden logischen Systembaustein eine Komponente entwickelt werden, die einerseits den Service implementiert und anderseits die verwendeten Aktuatoren und Sensoren über die *Robotics API* gemäß den spezifizierten Anforderungen ansteuert. Die Zuordnung der Aktuatoren, Sensoren und weiteren physischen Entitäten zu den logischen Systembausteinen ist in Tabelle 11.1 dargestellt.

Die Services können mit den zugeordneten physischen Entitäten wie folgt implementiert werden:

- Zuerst bewegt der SupplyService für das Basisbauteil den Manipulator zum Werkstückträger an der sekundären CarrierStation und nimmt dort das Bauteil mit seinem Parallelgreifer auf. Anschließend transportiert der Roboter das Basisbauteil zur stationären Spannvorrichtung.
- Das Basisbauteil wird durch den ClampingService mit dem Manipulator in die stationäre Spannvorrichtung gelegt und dort kraftschlüssig gehalten. Beim Entspannen wird das fertige Bauteil vom Manipulator aus der stationären Spannvorrichtung genommen.
- Vor der Montag bewegt der SupplyService für den Deckel den Manipulator zum Werkstückträger an der primären CarrierStation und nimmt dort das Bauteil mit dem Greifer auf. Anschließend transportiert der Roboter das Bauteil zur stationären Spannvorrichtung.
- Der AssemblerService steuert den Manipulator kraftgesteuert, um den Deckel sicher auf dem eingespannten Bauteil zu platzieren. Der Greifer muss nach erfolgreichem Fügen den Deckel loslassen.
- Beim FasteningService wird das zweite Werkzeug des Manipulators aktiv. Folglich wird das elektrische Schraubsystem zuerst am Schrau-

SYSTEMBAUSTEIN (SERVICE)	GERÄTE	PHYSISCHE OBJEKTE
coverSupplySystem : SUPPLYSYSTEM (SUPPLYSERVICE)	*robot* : LWR *gripper* : MEG50	*primary* : CARRIERSTATION
componentSupplySystem : SUPPLYSYSTEM (SUPPLYSERVICE)	*robot* : LWR *gripper* : MEG50	*secondary* : CARRIERSTATION
clampingSystem : CLAMPINGSYSTEM (CLAMPINGSERVICE)	*robot* : LWR *gripper* : MEG50 *stationaryGripper* : MEG64	
assembler : COMPONENTASSEMBLER (ASSEMBLERSERVICE)	*robot* : LWR *gripper* : MEG50	
fasteningSystem : FASTENINGSYSTEM (FASTENINGSERVICE)	*robot* : LWR *screwdriver* : SCREWDRIVER	*depot* : BOLTDEPOT
storageSystem : STORAGESYSTEM (STORAGESERVICE)	*robot* : LWR *gripper* : MEG50	*primary* : CARRIERSTATION

Tabelle 11.1: Zuordnung der Services und logischen Systembausteine auf die vorhandenen Entitäten der Roboterzelle für die Montage mit einem Roboter.

bendepot positioniert, um eine Schraube aufzunehmen. Diese wird anschließend zum Bauteil bzw. dem Gewinde transportiert und dort eingeschraubt. Der Vorgang wird wiederholt, bis alle Gewinde verschraubt sind.

- Nach dem Entspannen wird das Bauteil bereits vom Greifer gehalten. Daher bewegt der STORAGESERVICE den Manipulator zum Werkstückträger an der primären CARRIERSTATION und legt dort das Bauteil in einen leeren Platz.

Durch die Abbildung auf eine andere Zellenkonfiguration werden die Vor- und Nachbedingungen im Vergleich zur kooperativen Montage unterschiedlich verfeinert.

Es konnte gezeigt werden, dass sich die lösungsneutralen Handhabungs- und Fertigungsfunktionen, die durch die Services spezifiziert werden, auf unterschiedliche Zellenkonfigurationen abbilden lassen. Dadurch kann eine Beschreibung des Fertigungsprozesses erfolgen, die unabhängig ist von den verwendeten Manipulatoren und Werkzeugen. Erst durch die aufgabenzentrierten Dienste findet eine Konkretisierung der Aktuatoren statt. Die Implementierung dieser Dienste kann, wie in Abschnitt 9.4 gezeigt wird, unterstützt und vereinfacht werden. Sogar die Roboterbewegungen und Werkzeugaktionen, die teilweise identisch in der kooperativen Montage verwendet werden (z. B. beim SUPPLYSERVICE oder STORAGESERVICE), können wiederverwendbar konzipiert und implementiert werden (vgl. Kap. 10).

11.3 VARIATION DER WERKSTÜCKE

Der gesamte Fertigungsprozess wird auf die Handhabungsobjekte, d. h. die in der Roboterzelle vorhandenen Werkstücke, abgestimmt. Der Prozess besteht aus einer Abfolge von Handhabungs- und Fertigungsfunktionen, die jeweils auf eine Bauteil-bezogene Aufgabe abgebildet werden. Diese Aufgabe ist, wie in Kapitel 7 beschrieben wurde, ausschließlich über die Werkstücke des Fertigungsprozesses definiert. Intern wird eine Aufgabe über eine Sammlung von Fähigkeiten realisiert, die wiederum die Interaktion ei-

nes Roboterwerkzeugs mit den Werkstücken der Zelle beschreiben. Folglich sind die Werkstücke ein entscheidender Bestandteil einer serviceorientierten Roboterzelle.

Die eigentliche Handhabung bzw. Bearbeitung eines Werkstückes findet in Strategien statt. Die Parameter einer Strategie sollten über eine PROPERTY, z. B. über eine GRASPINGPROPERTY, Bestandteil jedes Werkstückes sein (vgl. Kap. 10). Dadurch kann sichergestellt werden, dass die Roboter und deren Endeffektoren jedes Werkstück individuell handhaben und bearbeiten. Dies kann jedoch nicht generell durch eine serviceorientierte Implementierung einer Roboterzelle geschehen, sondern muss mit Bedacht in die Modellierung und Umsetzung der Services einfließen.

Daher ist die Modellierung der Werkstücke ein entscheidender Aspekt. Die Modellierung muss so gewählt sein, dass spätere Anpassungen der Werkstücke umgesetzt werden können. Das Attribut oder die geometrische Eigenschaft, die einer Änderung unterliegt, muss Teil der Werkstückmodellierung sein. Ansonsten ist es schwierig vorherzusagen, was eine Variation der Werkstücke an Änderungen der Prozesse und Services bedingt. Auch die Prozesse und darunter liegenden Services müssen flexibel modelliert sein, um potentielle Änderungen zu berücksichtigen. Falls der in Abschnitt 9.2 beschriebene Schraubvorgang für genau eine Schraube ausgelegt wird, bedingt eine Änderung der Anzahl einen größeren Adaptionsaufwand. Werden diese Funktionen jedoch, wie in Abschnitt 9.4.1 erläutert, durch abstrakte Werkstücke (vgl. BOLTABLE) abgebildet, ist eine Adaption leichter möglich.

Im Fall der Factory 2020 können folgende Variationen der Werkstücke betrachtet werden:

- Anzahl und Position der Gewinde (BOLT) auf dem COVER.
- Art der Schrauben, d. h. Größe, Kopfform (z. B. Zylinder-, Rund- oder Senkkopf) oder Antrieb (z. B. Torx oder Inbus).
- Größe der beiden Werkstücke.

Abhängig von der Modellierung und den verwendeten Strategien können diese Variationen unterschiedliche Anpassungen bedingen.

Die in Abschnitt 9.2 vorgestellte Modellierung des Schraubvorgangs iteriert über die Anzahl der Schrauben. Daher ist eine Variation der Anzahl unproblematisch. Ebenso bereitet eine Variation der Gewindeposition keine Schwierigkeiten. Die Position ist fester Teil der Modellierung der Werkstücke und wird im Fertigungsprozess bzw. in den Strategien aus dem Objekt erfragt.

Lokale Anpassung der Strategien

Die Art der Schrauben bzw. Gewinde wurde sehr ausführlich modelliert und ist daher als Teil der Werkstücke implementiert. Jedoch wurde in der Factory 2020 die Strategie, um eine Schraube aufzunehmen bzw. in ein Gewinde einzusetzen, für eine bestimmte Schraubenart implementiert und experimentell validiert. Durch den Einsatz kraftgesteuerter Bewegungen (vgl. [10]) ist eine Übertragung dieser Strategie auf andere Schraubenarten nicht gewiss. Daher kann es nötig sein, für jede Schraubenart eine eigene Strategie zu implementieren und zu validieren. Allerdings ist weder eine Änderung der Fertigungsprozesse noch der aufgabenorientierten Dienste notwendig.

Bei der Größe der Werkstücke können die Breite, Länge und Höhe variiert werden. Dabei müssen die mechanischen Einschränkungen, die durch die Werkstückträger, die Greiferbacken und die maximale Öffnungsweite der

Greifer vorgegeben wird, beachtet werden. Ansonsten ist die Variation der Werkstückgröße eine Frage der Parametrisierung der entsprechenden Greif- und Ablegestrategien. Falls z. B. beim Aufnehmen eines Werkstückes eine absolute Greifposition spezifiziert wurde, wird eine Änderung des Softwareobjekts keine Auswirkung auf den Greifvorgang haben. Daher ist es wichtig, die Strategien soweit wie möglich anhand der geometrischen Merkmale der Bauteile zu parametrisieren.

Die Beispiele zeigen, dass das in dieser Arbeit vorgestellte Konzept zur serviceorientierten Modellierung und Umsetzung einer Roboterzelle mit einer Variation der Werkstücke elegant umgehen kann. Durch die Unterscheidung zwischen den Prozessen und Arbeitsabläufen der Zelle auf der einen Seite und der Roboterprogrammierung auf der anderen Seite reduzieren sich Variationen der Werkstücke in der Regel auf eine Anpassung der Strategien. Das bedeutet, dass die Änderungen lokal und auf das Bauteil bezogen sind. Insgesamt ist aber die Wahl und Parametrisierung der Strategien entscheidend. Änderungen, die a priori nicht in den Arbeitsabläufen berücksichtigt wurden, können auch durch die vorgestellte Modellierung nur mit höherem Aufwand umgesetzt werden.

11.4 VARIATION DER MANIPULATOREN UND GREIFER

Wie auch bei der oben vorgestellten Variation der Werkstücke ist ein entscheidender Vorteil serviceorientierter Roboterzellen, dass die Fertigungsprozesse und Arbeitsabläufe auf den höheren Abstraktionsebenen ohne die konkreten Roboter und deren Werkzeuge modelliert werden (vgl. Kap. 7). Zum ersten Mal werden die Roboterwerkzeuge bei der Implementierung aufgabenorientierter Dienste verwendet. Sie sind die Eingabeargumente der darunter liegenden *Skill-level Services*. Hierbei sollten in der Regel abstrakte Oberklassen verwendet werden, d. h. statt einem Schunk MEG50 sollte eine ParallelGripper verwendet werden.

Die konkreten Aktuatoren, d. h. Roboter und Werkzeuge der Roboterzelle, werden erst in den Strategien relevant, da dort die eigentliche Roboter- und Werkzeugprogrammierung stattfindet. Allerdings können Strategien, wie in Abschnitt 10.4 gezeigt wurde, auch sehr generisch umgesetzt werden und sind dadurch zu einem gewissen Maß unabhängig von den Geräten. Probleme mit den konkret verwendeten Aktuatoren zeigen sich in einer generischen Strategie bspw. durch ein fehlendes ActuatorInterface oder einen nicht den Erfordernissen entsprechenden Arbeitsraum. Dies kann in der Regel a priori durch den *Provider* einer Strategie überprüft werden, sodass die Strategie nicht erst angewendet wird. Somit kann sich eine Strategie auf abstrakte Aktuatoren und deren geometrische Merkmale abstützen.

Im Fall der Factory 2020 wurden folgende Variationen der Aktuatoren evaluiert:

- Statt einem KUKA Leichtbauroboter, repräsentiert durch die Klasse LBR, wurde ein Schunk Lightweight Arm, repräsentiert durch die Klasse LWA46, verwendet (vgl. Abschn. 6.4.3).
- Statt einem Schunk MEG-50 Parallelgreifer, repräsentiert durch die Klasse MEG50, wurde ein Schunk WSG-50 Parallelgreifer, repräsentiert durch die Klasse WSG50, verwendet (vgl. Abschn. 6.4.2).

Beide Variationen wurden in unterschiedlichen Kombinationen evaluiert. Eine mögliche Kombination ist in Abbildung 11.8 dargestellt.

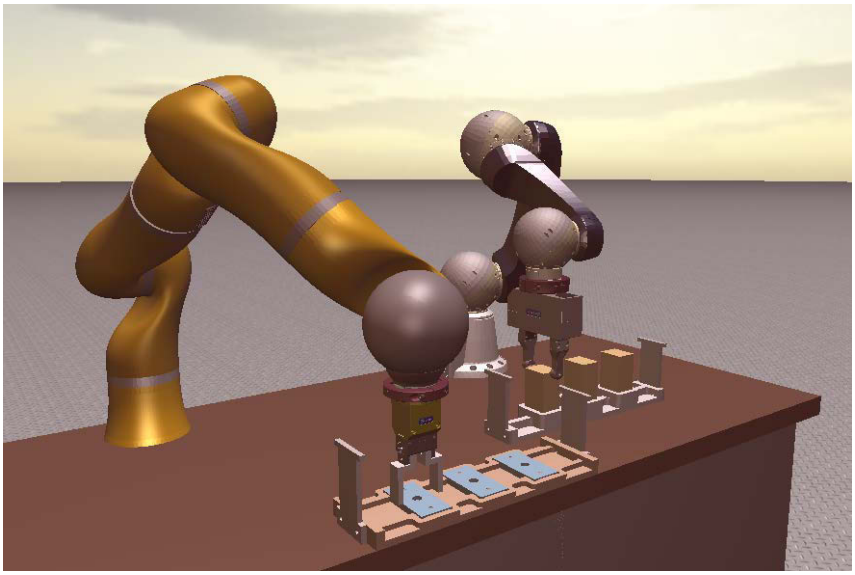


Abbildung 11.8: Die Visualisierung zeigt eine mögliche Variation der Aktuatoren in der Factory 2020.

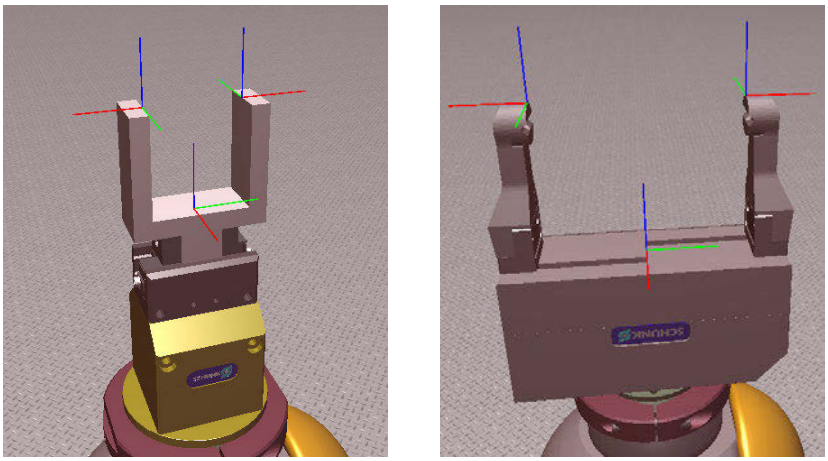
Bei der Variation der Roboter ist zu berücksichtigen, dass der LWA46 nicht über eine integrierte Kraft- und Drehmomentmessung verfügt. Daher sind kraftgeregelte Bewegungen mit diesem Roboter nicht möglich. Durch die fehlende siebte Achse ist auch der kooperative Transport der Werkstückträger nicht realisierbar. Um die generelle Austauschbarkeit der Modellierung zu evaluieren, eignet er sich aufgrund der ähnlichen Größe und Traglast jedoch sehr gut. Mit dem LWA46 wurde infolgedessen die Aufnahme, Übergabe und Ablage der Werkstücke getestet.

Austausch der Roboter

Eine wichtige Voraussetzung für die Austauschbarkeit der Manipulatoren ist es, dass in den Strategien keine gelenkspezifischen Bewegungen verwendet werden dürfen. Diese ergeben für jeden Roboter eine andere kartesische Position und sind daher ungeeignet. Stattdessen müssen alle Bewegungen durch kartesische Koordinatensysteme spezifiziert werden. Dies ergibt sich jedoch automatisch, falls die Strategien auf Basis der geometrischen Merkmale eines Bauteils ausgerichtet sind. Ist dies der Fall, gibt es noch zwei Aspekte zu beachten. Zum einen müssen die Arbeitsräume in etwa identisch sein, zum anderen spezifiziert ein Frame die Pose eines Roboters nicht eindeutig. Zwar wird der Roboterflansch korrekt positioniert, die Konfiguration des Roboterarms und seiner Gelenke wird jedoch nicht vorgegeben.

Dafür sieht die *Robotics API* sogenannte *Hint Joints* vor, d. h. die Konfiguration des Manipulators sollte für einen kartesischen Zielpunkt möglichst nahe an dieser vorgegebenen Gelenkstellung sein (vgl. Abschn. 6.4.3). Eine solche Gelenkstellung kann z. B. per Konfiguration über eine HintJointProperty zu einem Frame hinzugefügt werden. Dadurch müssen *Hint Joints* nicht programmatisch bei der Spezifikation der Bewegung übergeben werden, sondern sind implizit über die HintJointProperty spezifisch für jeden Roboter Teil des Zielpunktes. Es müssen nicht für jede Bewegung *Hint Joints* definiert werden. In der Regel ist es ausreichend, dass diese bei einer initialen Bewegung oder größeren Transferbewegungen spezifiziert werden.

Mit dem Einsatz von *Hint Joints* war es möglich, dass die Roboter in der Factory 2020 ausgetauscht werden konnten. Es mussten keine programmatischen Änderungen an den Prozessen, Diensten oder Strategien vorgenom-



(a) MEG-50 (b) WSG-50

Abbildung 11.9: Beide Greifer verfügen über dieselbe Menge an geometrischen Merkmalen, die jedoch unterschiedlich ausgeprägt sind. Beide Abbildungen stellen jeweils das Kontaktkoordinatensystem und die beiden Frames an den Enden der Greiffinger dar.

men werden. Dabei wurden, wie oben beschrieben, nur Strategien verwendet, die keine kraftgeregelte Bewegungen verwenden. Das bedeutet, dass für die klassischen positionsgesteuerten Pick-and-Place-Aufgaben beide Roboter ohne programmatische Änderung verwendet werden konnten. Um die beiden Roboter auszutaschen waren demnach nur Änderungen in der Konfiguration der Roboterzelle notwendig (vgl. Abschn. 6.3).

Austausch der Greifer

Neben den Manipulatoren wurden auch zwei unterschiedliche Parallelgreifer verwendet. Statt dem ursprünglich eingesetzten MEG50 von wurde später der ebenfalls von Schunk stammende WSG50 benutzt. Dabei ist zu berücksichtigen, dass die beiden Greifer kompatible Öffnungsweiten und – trotz ihrer unterschiedlichen Ansteuerung (vgl. Abschn. 6.4.2) – eine ähnliche Programmierung aufweisen. Zudem sind die verwendeten Greifbacken sehr spezifisch für die Anwendung, d. h. beide Greifer müssen mit angepassten Greifbacken ausgerüstet werden. Diese können jedoch mit der *Robotics API* modelliert und per Konfiguration spezifiziert werden.

Durch eine explizite Modellierung aller relevanten, geometrischen Merkmale wurde die Austauschbarkeit beider Greifer erreicht. Die für die Strategien entscheidenden Merkmale der Greifer sind in Abbildung 11.9 dargestellt. Die beiden Greifstrategien, die in Abschnitt 10.4 vorgestellt wurden, berücksichtigen die geometrischen Merkmale von Parallelgreifer. Somit ist es möglich, beide Greifer ohne programmatische Änderung auszutauschen.

11.5 Unterstützung durch graphische Notationen

Die in Abschnitt 6.5.1 beschriebene Visualisierung bietet sich an, den aktuellen Zustand einer Roboterzelle darzustellen. Dabei können alle in der Roboterzelle bekannten Geräte und Gegenstände an ihrer vermuteten geometrischen Position dargestellt werden. Dies ist ein wesentlicher Teil des Zustands der Automatisierungssoftware. Ein weiterer Teil sind die internen Zustände der Services und der Fertigungsprozesse. Dies ist in einem Java-Programm durch spezielle grafische Benutzeroberflächen oder mithilfe von Debugging möglich.

Es können jedoch auch grafische Notationen verwendet werden, um diese Services und Prozesses zu implementieren. Dadurch wird – in Anlehnung an die in Kapitel 9 vorgestellte serviceorientierte Modellierung – eine graphische Programmierung möglich. Insbesondere die Fertigungsprozesse und die internen Zustandsmaschinen von Services können so abgebildet werden. Neben der reinen Implementierung ist zur Laufzeit durch entsprechenden Tool-Support eine Visualisierung des aktuellen Fertigungsprozesses oder der aktiven Dienste anhand der grafischen Notation möglich.

SCXML

Um Zustandsmaschinen graphisch zu programmieren, wurde State Chart XML (SCXML) [26] verwendet. Es handelt sich dabei um eine ereignisgetriebene Beschreibungssprache für Zustandsautomaten in XML, welche vom World Wide Web Consortium (W3C) [283] entwickelt wird. Ein SCXML-Dokument kann inhaltlich gesehen in zwei Bereiche zerlegt werden: Der erste Teil beinhaltet die einzelnen Zustände und deren Transitionen untereinander, der zweite Teil den ausführbaren Inhalt (*Executable Content*), welcher aus Aktionen besteht, die als Teil einer Transition und beim Eintreten bzw. Verlassen eines Zustands ausgeführt werden. Zur Interpretation der mit SCXML spezifizierten Zustandsmaschinen wurde Commons SCXML [20] verwendet. Commons SCXML ist eine Java-Implementierung der SCXML-Spezifikation der Apache Foundation [22] und enthält eine Engine, die eine in SCXML definierte Zustandsmaschine ausführen kann.

Dadurch kann man das Verhalten der oben beschriebenen fähigkeitenzentrierter Dienste direkt mit SCXML modellieren. Anschließend können diese Zustandsmaschinen mit der Commons SCXML Engine intepretiert werden. Die Zustandsmaschine spezifiziert das Verhalten des Dienstes und definiert, welche Operationen wann und in welcher Reihenfolge aufgerufen werden können. Auch bei bei der Komposition von Diensten zu komplexerer Funktionalität wie z. B. bei der *Factory 2020* wurden Zustandsmaschinen eingesetzt. Bei der grafischen Modellierung der *Factory 2020* hat sich gezeigt, dass Zustandsmaschinen prinzipiell eine sehr gute Möglichkeit der Spezifikation sind, insbesondere da sie von weit verbreiteten Modellierungssprachen (z.B. UML, SysML) unterstützt werden. Durch die verwendete Engine können die Modelle (ergänzt durch weitere technische Informationen) direkt als „Implementierung“ verwendet werden.

Zur grafischen Modellierung von Zustandsmaschinen mit SCXML wurde ein Plug-in [59] für Eclipse [77] entwickelt, welches auf dem Graphical Modeling Framework (GMF) [79] basiert und einen auf die serviceorientierte Automatisierung angepassten Editor für SCXML darstellt. Der Editor wurde so ausgelegt, dass er zur Laufzeit das Monitoring aktuell ausgeführtet Zustandsmaschinen unterstützt. Dabei werden aktive Zustände und Transitionen farblich hervorgehoben. Der Editor und die farbliche Hervorhebung von aktiven (hierarchischen) Zuständen sind in Abbildung 11.10 dargestellt.

BPMN

Neben State Charts wurden ebenfalls Prozessmodellierungssprachen zur Anlagensteuerung untersucht. Prozessmodellierungssprachen, insbesondere die Business Process Modeling Notation 2.0 (BPMN) [139, 195], werden vor allem im Umfeld serviceorientierter Geschäftsanwendungen verwendet. Diese Sprachen sind zu den aus der UML bekannten Aktivitätsdiagramme verwandt und erlauben es, Prozesse bzw. Arbeitsabläufe einfach grafisch zu beschreiben. Einzelne Aktivitäten bzw. Prozessschritte stellen dabei Knoten dar, die über gerichtete Kanten miteinander verbunden werden. Dadurch wird ein sequentieller Ablauf modelliert. Darüber hinaus können auch nebenläufige Abläufe oder asynchrone Nachrichten modelliert werden.

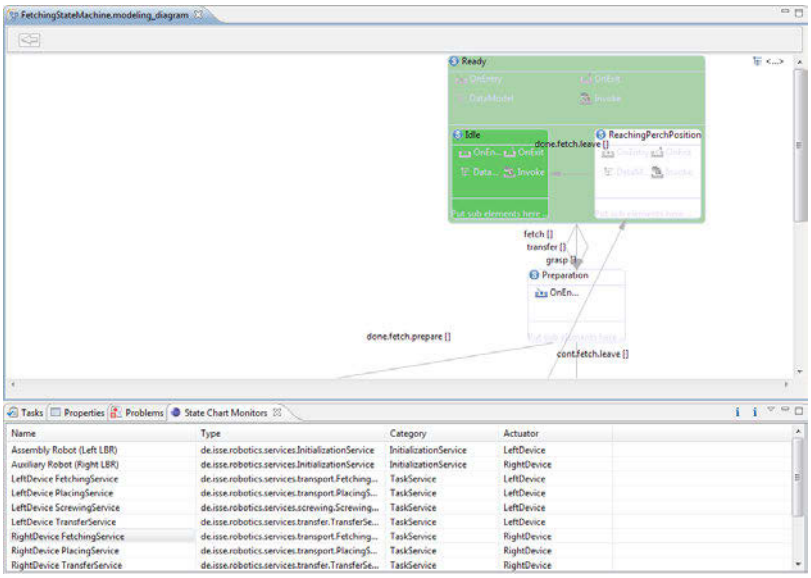


Abbildung 11.10: Das Plug-in für Eclipse stellt sowohl einen Editor als auch eine Visualisierung für hierarchische Zustandsmaschinen (SCXML) zur Verfügung. Die Zustandsmaschinen können zur Programmierung serviceorientierter Roboterzellen verwendet werden.

Durch die im Umfeld serviceorientierter Architekturen vorhandene Infrastruktur an Modellierungssprachen, Ausführungsumgebungen und Werkzeugen bieten sich diese Sprachen an, um für die Fertigungsprozesse einer serviceorientierten Roboterzelle verwendet zu werden. Dementsprechend wurde eine Evaluation bestehender Sprachen und Ausführungsumgebungen durchgeführt. Als geeignete und freie Workflow-Management-Systeme haben sich jBPM [226] und die Activiti BPM Platform [2] herauskristallisiert. Beide Frameworks können einfach in die *OSGi Service Platform* integriert werden und bieten einen Editor als Plugin-in für Eclipse. Sie bieten neben BPMN zudem eigene, aber einfachere Prozessmodellierungssprachen an, die mit dem *Robotics Application Framework* verwendet werden können.

Darauf aufbauend wurden verschiedene Automatisierungsszenarien (z. B. Pick-and-Place) entwickelt. Diese wurden dann mit BPMN und dem jBPM bzw. der Activiti BPM Platform als Workflow-Management-Systeme umgesetzt. Dabei hat sich gezeigt, dass die Integration mit der serviceorientierten Architektur zur Abbildung komplexer Roboterzellen nahtlos funktioniert. Die Prozessmodellierung eignet sich insbesondere für die Programmierung von Zellen mit mehreren Robotern, da es hierfür spezielle Operatoren (z. B. zum Synchronisieren paralleler Abläufe) gibt.

Durch die unterschiedlichen Notationen konnte gezeigt werden, dass die Entwicklung serviceorientierter Roboterzellen durch graphische Beschreibungssprachen unterstützt werden kann. Da mithilfe der serviceorientierten Strukturierung von der eigentlichen Roboter- und Werkzeugprogrammierung stark abstrahiert wird, eignen sich graphische Notationen gut, um die Abläufe zu spezifizieren bzw. zu visualisieren. Dabei können die Beschreibungssprachen auf unterschiedlichen Abstraktionsebenen der Programmierung verwendet werden (vgl. Kap. 7). Zudem konnte gezeigt werden, dass unter Zuhilfenahme graphischer Sprachen die Abläufe einer Roboterzelle zur Laufzeit überwacht werden können.

12 FAZIT UND AUSBLICK

Ziel dieser Arbeit war die Entwicklung und Evaluierung eines Konzepts zur serviceorientierten Automatisierung von Roboterzellen. Dieses Konzept konnte an einer Fallstudie – der Factory 2020 – erfolgreich validiert werden. Dabei wurde eine kooperative Montageaufgabe unter Verwendung zweier Leichtbauroboter realisiert. Die serviceorientierte Architektur ermöglicht eine funktionale Dekomposition der Roboterzelle und ist in unterschiedliche Ebenen gegliedert, wobei jede Ebene aus *Robotics Automation Services* einer bestimmten Granularität besteht. Eine serviceorientierte Roboterzelle kann als cyber-physisches Produktionssystem das Herzstück einer vernetzen und intelligenten Fabrik bilden. Sie erreicht ihre Flexibilität dadurch, dass sie sich auf intelligente Produkte stützt, die „das Wissen ihres Herstellungsprozess“ [144, S. 23] in sich tragen, z. B. mit welchen Parametern sie bearbeitet werden müssen.

Im Kontext von Industrie 4.0 kann eine intelligente Fabrik aus einer Menge serviceorientierter Roboterzellen bestehen, die verschiedene Fertigungsdienstleistungen anbieten. Diese werden mittels Schnittstellen und Kontrakte beschrieben und können durch intelligente Produkte in Anspruch genommen werden. Demzufolge wird der Zustand eines Bauteils sukzessive transformiert, um letztendlich den gewünschten Zielzustand des fertigen Produkts zu erreichen. Die vorliegende Dissertation leistet mit der serviceorientierten Konzeption einer Roboterzelle und ihrer konsequenten Fokussierung auf das Bauteil einen elementaren Beitrag, um die Vision von Industrie 4.0 umzusetzen. Die erzielten Ergebnisse dieser Arbeit werden daher in Abschnitt 12.1 kurz zusammengefasst und resümiert. Die Dissertation schließt mit einem Ausblick auf weiterführende Forschungsmöglichkeiten in Abschnitt 12.2.

12.1 RESÜMEE DER ERZIELTEN ERGEBNISSE

Als Grundlage für die erzielten Ergebnisse ist eine neue Softwarearchitektur für die Programmierung von Industrierobotern und Automatisierungskomponenten im Forschungsprojekt *SoftRobot* entstanden (vgl. Kap. 4). Sie ermöglicht die Entwicklung von (kooperativen) Roboteranwendungen in einer modernen, objektorientierten Programmiersprache und löst die zentrale Frage, wie sich der notwendige Echtzeitbetrieb einer Robotersteuerung mit den Paradigmen moderner Programmiersprachen harmonisieren lässt. Der Lösungsansatz ist die Trennung zwischen der echtzeitkritischen Steuerung von Robotern bzw. Werkzeugen und der Anwendungslogik, d. h. dem eigentlichen Programmablauf. Während Bewegungen und Werkzeugaktionen in echtzeitkritische Transaktionen zusammengefasst und auf einem entsprechenden Betriebssystem als Einheit ausgeführt werden, kann die Anwendungslogik unter einem Standardbetriebssystem ohne spezielle Anforderungen ausgeführt werden. Dadurch können Roboteranwendungen in einer universellen Programmiersprache wie Java entwickelt werden, die durch Merkmale, wie z.B. eine automatische Speicherverwaltung, nur eingeschränkt echtzeitfähig sind. Durch das Konzept echtzeitkritischer Transaktionen kann

von der Steuerungsebene, die eine echtzeitfähige Programmierung voraussetzt, abstrahiert werden.

Stattdessen wird die Funktionalität einer industriellen Roboterzelle über eine wohldefinierte Programmierschnittstelle, die *Robotics API*, zur Verfügung gestellt, die von einem Anwendungsentwickler benutzt werden kann. Durch die objektorientierte Programmierschnittstelle können alle wichtigen Bestandteile einer Roboterzelle modelliert werden (vgl. Kap. 5), welche die vorhandenen Roboterarme und Werkzeuge, die Bauteile der Fertigungsprozesse, sowie die relevanten geometrischen Informationen wie z. B. Koordinatensysteme umfasst. Damit stellen die Aktuatoren einer Roboterzelle nicht nur ihre Funktionalität über die Objekte bereit. Sie bilden die Topologie und die geometrische Struktur der realen Roboterzelle nach. Dieses Wissen wird bspw. durch die Ausführung einer Roboterbewegung automatisch aktualisiert. Damit wurde die von Angerer [10] vorgestellte *Robotics API* um einen wichtigen Aspekt erweitert, der die Voraussetzung dafür ist, dass Services auf Basis des aktuellen Zellenzustands planen und entscheiden können. Eine objektorientierte Beschreibung der Roboterzelle ist dementsprechend die Grundlage für die serviceorientierte Implementierung einer Roboterzelle.

Die Softwarearchitektur und ihre objektorientierte Programmierschnittstelle wurden in ein erweiterbares Framework für die Industrierobotik und Automatisierung überführt (vgl. Kap. 6). Das *Robotics Application Framework* ist mithilfe von OSGi modular und erweiterbar aufgebaut. Somit lässt sich die Automatisierungssoftware einer Roboterzelle individuell aus einer Vielzahl von Komponenten zusammenstellen. Falls darüber hinaus Elemente oder Geräte benötigt werden, können diese ohne größeren Aufwand in das *Robotics Application Framework* integriert werden. Die Erweiterbarkeit betrifft nicht nur die objektorientierte Programmierschnittstelle auf oberster Ebene, sondern zieht sich bis in die Echtzeitrobotersteuerung vertikal durch die Softwarearchitektur. Durch die Integration von Robotern und industriellen Werkzeugen sowie Feldbuskopplern und -komponenten stellt das *Robotics Application Framework* ein umfassendes Framework für die ganzheitliche Steuerung von industriellen Roboterzellen dar, was bspw. eine zusätzliche SPS für die Programmierung überflüssig macht. Zudem gibt es horizontale Erweiterungen, welche ausschließlich die objektorientierte Programmierschnittstelle erweitern und den Anwendungsentwickler durch eine Visualisierung oder eine automatische Planung unterstützen.

Das umfassende Framework ist die Voraussetzung für eine serviceorientierte Strukturierung von Roboterzellen (vgl. Kap. 7). Es ermöglicht die durchgängige Programmierung der Anwendung innerhalb eines geschlossenen Systems. Durch den Wechsel der unterschiedlichen Programmiersysteme (d. h. Robotersteuerungen, Werkzeugsteuerungen, speicherprogrammierbare Steuerungen) in der aktuellen Automatisierung von Roboterzellen entstehen Medien- und Kommunikationsbrüche, die dazu führen, dass das Wissen über die Abläufe der Zelle verteilt gespeichert und realisiert wird. Durch die hier umgesetzte Bündelung und Vereinheitlichung der Programmierung entsteht eine durchgängige Programmierung. Zentrales Element sind dabei die Handhabungsobjekte und Bauteile. Sie werden in der Roboterzelle durch die Fertigungsschritte und -prozesse sowohl mechanisch als auch softwaretechnisch transformiert. Sie sind das zentrale Entscheidungskriterium einer flexiblen Produktion und tragen das Wissen, wie sie gegriffen, bewegt und bearbeitet werden sollen, in sich. Letzteres wird von der höchsten Abstraktionsebene bis hinunter zur eigentlichen Roboterbewegung mitgeführt und verwendet.

Eine serviceorientierte Roboterzelle setzt sich aus einer Menge von Diensten unterschiedlicher Granularität zusammen. Ihnen ist allen gemein, dass sich ihre Funktionalität gemäß ihrer Granularität immer am Fertigungsprozess ausrichtet. Die Dienste haben einen bedeutenden Anteil an der Umsetzung des Automatisierungsprozesses einer Roboterzelle. Daher ist die Wahl der Dienste und der Aufbau einer Zelle aus den einzelnen Diensten entscheidend für ihre Modularität und Flexibilität. In dieser Arbeit wurde in Kapitel 9 ein Ansatz vorgestellt, wie eine solche Unterteilung vorgenommen werden kann. Zur Modellierung der Roboterzelle wird in diesem Kapitel die Systems Modeling Language (SysML) verwendet. SysML ist eine auf der UML basierende, standardisierte Modellierungssprache im Bereich des Systems Engineering.

Die Roboterzelle wird in logische Einheiten aufgeteilt. Auf Basis der logischen Einheiten können lösungsneutrale Handhabungs- und Fertigungsfunktionen als Service definiert und anschließend zu komplexeren Fertigungsprozessen orchestriert werden. Somit werden über einen Service die Aufgaben und Zuständigkeiten einer logischen Einheit definiert und bereitgestellt. Auf Grundlage dieser Services lassen sich Produktionsabläufe als Sequenz von Serviceaufrufen realisieren. Um eine logische Einheit zu implementieren, werden die Services unter Zuhilfenahme der in der Zelle verfügbaren Aktuatoren und Sensoren umgesetzt. Trotzdem sind die Schnittstellen der Services weiterhin unabhängig von den verwendeten Aktuatoren, was die Flexibilität und Variabilität einer serviceorientierten Roboterzelle erhöht. Eine Änderung der Topologie oder der konkreten Aktuatoren bedeutet, dass nur die Implementierung und nicht die Definition der Services verändert wird.

Auf höchster Ebene werden die Automatisierungsprozesse der serviceorientierten Roboterzelle beschrieben. Jeder Prozessschritt stellt eine Handhabungs- oder Fertigungsfunktion dar, die das Bauteil räumlich oder physisch verändert. Bereitgestellt werden die notwendigen Handhabungs- oder Fertigungsfunktionen von *Task-level Services*. Innerhalb eines solchen Dienstes findet eine erste Verknüpfung zwischen der Handhabungs- bzw. Fertigungsfunktion und dem ausführenden Werkzeug statt, wobei die Funktionen ausschließlich durch abstrakte Fähigkeiten eines Werkzeugs beschrieben werden. Die Umsetzung dieser Fähigkeit wird an Services einer niedrigeren Abstraktionsebene, d. h. an *Skill-level Services*, ausgelagert. Dort findet erst die Verknüpfung zwischen dem Werkzeug und dem Roboter, an dem es montiert ist und über den es bewegt werden kann, statt (vgl. Kap. 10). Durch eine Orchestrierung von Services werden die Arbeitsabläufe in der Roboterzelle auf unterschiedlichen Ebenen beschrieben. Damit ist keine SPS für die übergeordnete Steuerung notwendig.

Da jede Werkzeugfähigkeit auf unterschiedliche Art und Weise realisiert werden kann, verwendet ein *Skill-level Service* Strategien für die konkrete Umsetzung der Fähigkeit. Eine Strategie stellt eine austauschbare und wiederverwendbare Lösungsmöglichkeit für eine Werkzeugfähigkeit dar. Je nach Situation und Kontext kann die passende Strategie ausgewählt, parametrisiert und ausgeführt werden. Dabei übernimmt der Service die Auswahl und Auswertung der Strategien mithilfe des aktuellen Kontextes, den er über das objektorientierte Modell der Roboterzelle ableiten kann. Die optimale Strategie und ihre Parametrisierung wird jedoch über das betroffene Objekt vorgegeben. Es weiß, wie es gegriffen, bewegt oder bearbeitet werden möchte. Daher sind im handzuhabenden bzw. zu bearbeitenden Objekt alle relevanten Informationen über die Strategie, deren Rahmenbedin-

gungen und Parameter gespeichert. Insgesamt zeigt der im Rahmen dieser Dissertation entwickelte, innovative und umfassende Ansatz, dass eine durchgängige Zentrierung auf das Bauteil – einerseits bei der Definition der Fertigungsfunktionen und Aufgaben und andererseits bei der Bewegungsprogrammierung des Roboters – die Entwicklung flexibler und anpassbarer Roboterzellen ermöglicht.

12.2 AUSBLICK

Aktuell werden die Ergebnisse dieser Dissertation in weiteren Forschungsprojekten eingesetzt. Im Rahmen des Verbundprojekts AZIMUT [103], einem vom BMWi im Luftfahrtforschungsprogramm geförderten Projekts, wurde eine erweiterbare Offline-Programmierplattform [187] entwickelt, die an die speziellen Bedürfnisse der Fertigung carbonfaserverstärkter Kunststoffe (CFK) angepasst ist. Ein CFK-Bauteil besteht aus einer Menge unterschiedlicher Einzellagen bzw. Textilien, die in eine komplex geformte, dreidimensionale Form drapiert werden. Bei dem Drapieren wird das planare Textil umgeformt, sodass es schließlich glatt und genau positioniert in der dreidimensionalen Form liegt. Während dieser Vorgang noch weitgehend manuell erfolgt, werden in der Forschung immer häufiger Endeffektoren für Roboter entwickelt, die diese Aufgaben übernehmen. Daher liegt ein Augenmerk auf einer erweiterbaren Gestaltung dieser Plattform, um offen für neue Roboter und insbesondere neue Endeffektoren für die Handhabung von CFK-Textilien zu sein. Die Offline-Programmierung der Roboterzelle erfolgt prozessorientiert, d. h. als Eingabeparameter wird die Produktionsvorschrift, das sogenannte *Plybook* verwendet. Unter Zuhilfenahme dieser Daten wird pro Zuschnitt eines CFK-Textils die Bewegungsprogrammierung für den Roboter und den Greifer durchgeführt.

In der Offline-Programmierplattform wurde das *Robotics Application Framework* als Grundlage für die Planung und Ausführung der Roboterbewegungen und -aktionen verwendet. Durch OSGi und die Aufteilung in eine Vielzahl einzelner Bundles ist das *Robotics Application Framework* von sich aus erweiterbar gestaltet. Dadurch war es einfach, weitere Roboterarme oder spezielle Endeffektoren für die CFK-Handhabung zu integrieren. Die Kompatibilität zu OSGi ermöglicht zudem eine direkte Integration des *Robotics Application Frameworks* in Eclipse RCP [175], das die Basis der Offline-Programmierplattform darstellt.

Auch wird die in dieser Arbeit vorgestellte objektorientierte Modellierung einer Roboterzelle sehr intensiv in der Offline-Programmierplattform verwendet. Die Spezifikation und die Programmierung der Aufgaben basiert auf den modellierten Elementen der virtuellen Roboterzelle und ihrer geometrischen Merkmale. Dabei werden statt den vorgestellten Services Generatoren verwendet, die auf Basis der in der virtuellen Roboterzelle verfügbaren Objekte und ihrer Merkmale die Bewegungen und Werkzeugaktionen planen [187]. Die fehlende Parametrierung der Fertigungsaufgaben wird durch Rückfragen an den Benutzer kompensiert. Um erweiterbar zu sein, sind die Generatoren als Service in die *OSGi Service Platform* integriert. Darüber hinaus werden die in Kapitel 6 vorgestellte Visualisierung und Kollisionserkennung verwendet, um die Roboterbewegungen und die Aktionen der Endeffektoren simuliert darzustellen bzw. zu planen.

Ein weiteres, aktuelles Forschungsprojekt, in dem Ergebnisse dieser Arbeit verwendet werden, ist das vom BMBF geförderte Verbundprojekt *SafeAssistance* [134]. Ziel ist es, ein neues Sicherheitssystem auf Basis kapa-

zitiver Sensoren zu erforschen, das ohne Schutzhülle und taktile Sensoren auskommt, aber trotzdem sicher und zuverlässig Personen erkennt. Somit werden frühzeitig gefährliche Kollisionen erkannt und Unterbrechungen im Arbeitsablauf vermieden. Um dies zu erreichen, ist es notwendig die Auswertung kapazitiver Sensoren sicher und intelligent zu gestalten. Eine ganzheitliche Systemgestaltung und neue Algorithmen zur Interpretation der Sensordaten, insbesondere ein Vergleich mit einem vorher aufgezeichneten Umgebungsmodell, sollen das Sensorsystem in die Lage versetzen, Personen trotz verschiedensten Störeinflüssen im Arbeitsraum des Roboters zu erkennen. Somit kann jeder herkömmliche Industrieroboter zu einem SafeAssistance-Roboter für die sichere Mensch-Roboter-Kollaboration nachgerüstet werden.

Innerhalb des Projekts wird vor allem die objektorientierte Modellierung einer Roboterzelle und ihrer (physischen) Gestalt verwendet. Dadurch kann der kapazitive Sensor und seine geometrische Position am Roboter modelliert und spezifiziert werden. Der Sensor bewegt sich automatisch mit dem Roboter. Diese Informationen werden verwendet, um die Entfernung zu einem potentiellen Hindernisse besser abzuschätzen. Das objektorientierte Modell der Roboterzelle dient darüber hinaus als Grundlage, um ein Umgebungsmodell mit den Basiswerten der kapazitiven Sensoren effizient aufzuzeichnen. Es können dadurch kollisionsfreie Bewegungen für die Aufzeichnung des Umgebungsmodells generiert werden. Zudem kann man kollisionsfreie Bereiche für Ausweichbewegungen a priori bestimmen und zur Laufzeit verwenden.

Die in dieser Dissertation erreichten Ergebnisse zeigen vielfältige Möglichkeiten auf, die eine objektorientierte Abbildung der Roboterzelle und die darauf aufbauende serviceorientierten Automatisierung bieten. Die vorliegende Arbeit stellt innovative Lösungen auf konkrete Fragestellungen der Automatisierung bereit. Darüber hinaus lassen sie jedoch genug Spielraum für Weiterentwicklungen und neue Ansätze, insbesondere lassen sich noch vielfältigere geometrische Merkmale modellieren. Aktuell werden dazu nur Koordinatensysteme, d. h. dreidimensionale Punkte mit einer Orientierung, betrachtet. In Zukunft kann man zusätzlich noch Linien oder Flächen betrachten, also können geometrische Merkmale bspw. als Punkt auf einer Fläche spezifiziert werden. Eine Orientierung kann sich automatisch aus dem Normalenvektor der Fläche ergeben. Die Modellierung von Gegenständen und ihrer Merkmale könnte zudem aus einem dreidimensionalen Drahtgittermodell automatisch abgeleitet werden.

Bei der Modellierung serviceorientierter Roboterzellen werden die Fertigungsabläufe bisher fest definiert. Das bedeutet, dass die Reihenfolge der Handhabungs- und Fertigungsfunktionen vorgegeben ist. Durch die zentrale Stellung der Bauteile innerhalb der Prozesse können Variationen berücksichtigt werden und die Roboterbewegungen und Werkzeugaktionen sind spezifisch für ein Bauteil. Dennoch setzt ein vollständig neuer Prozessablauf eine manuelle Modellierung voraus. Durch die vorhandenen Vor- und Nachbedingung einzelner Handhabungs- und Fertigungsfunktionen aber auch einzelner Strategien kann eine automatische Planung der Fertigungsprozesse integriert werden. Bei Huckaby et al. [133] wurde die generelle Machbarkeit gezeigt und kann in einem nächsten Schritt auf die Ergebnisse dieser Arbeit übertragen werden. Insbesondere wenn die Roboterzelle redundant aufgebaut ist und aus mehreren, kooperierenden Robotern besteht, bedeutet die automatische Planung eine deutliche Vereinfachung der Programmierung.

Zukünftig kann jedes Bauteil einen abstrakten Bauplan mit sich tragen. Jeder Bauplan weist unterschiedliche Sichten mit jeweils zunehmender Detailgenauigkeit auf, wobei jede Sicht einer in Kapitel 7 vorgestellten Abstraktionsebene serviceorientierter Roboterzellen entspricht. Über den Bauplan und seine Sichten kann man die Orchestrierung der Dienste automatisch bestimmen. Das bedeutet, dass auf Prozessebene zuerst eine Abfolge lösungsneutraler Handhabungs- und Fertigungsfunktionen geplant wird. Anschließend können aufgabenzentrierte Dienste bestimmt werden, die diese Funktionen am Besten implementieren. Auf unterster Ebene kann man wiederum geeignete Strategien automatisch bestimmen und parametrisieren.

In einer intelligenten Fabrik stellt eine serviceorientierte Roboterzelle ihre Funktionalität, d. h. ihre angebotenen Fertigungsprozess, als Service zur Verfügung. Man kann in diesem Zusammenhang von *Robotics Manufacturing as a Service* sprechen. Diese können autonom von intelligenten Produkten in Anspruch genommen werden. Das Produkt kennt seinen eigenen Bauplan und versucht, sich in der Fabrik eigenständig bauen zu lassen. Erste Ansätze dazu wurden bereits bei Hoffmann et al. [126] vorgestellt. Dabei wurde eine adaptives und selbst-organisierendes System konzipiert, das den Prinzipien des *Organic Computing* [184] folgt. Dabei können die Akteure des Systems innerhalb eines vorgegebenen Korridors selbstständig agieren. Falls dieser Korridor verlassen wird, erfolgt eines Rekonfiguration des Systems. Das Konzept serviceorientierte Roboterzellen und die Idee einer adaptiven Fertigung ergänzen sich gegenseitig. Durch die bauteilzentrierte Fertigung können serviceorientierte Roboterzellen flexibel auf die Bedürfnisse einer vielfältigen Produktion reagieren. Die Selbstorganisation ermöglicht Fertigungsabläufe in einer intelligenten Fabrik flexibel zu planen und fehlertolerant auszuführen. Durch diese Kombination kann eine flexible und gleichzeitig robuste Produktion mithilfe serviceorientierter Roboterzellen in einer intelligenten Fabrik erreicht werden.

Teil IV

ANHANG

A TECHNOLOGIEPAKETE

Folgende Technologiepakete der Firma KUKA Roboter GmbH wurden anhand ihrer Handbücher, interner Dokumentationen und persönlicher Gespräche mit Experten untersucht (in alphabetischer Reihenfolge):

- *BendTech V 2.1*. KUKA Roboter GmbH, 9. Jan. 2006
- *FTCtrl Benutzerhandbuch*. Amatec Robotics GmbH, 10. Juli 2003
- *KUKA.CAMRob 2.1 PC*. KUKA Roboter GmbH, 18. Jan. 2008
- *KUKA.ConveyorTech 3.2*. KUKA Roboter GmbH, 21. Feb. 2007
- *KUKA.CR.ArcTech Digital 2.0*. KUKA Roboter GmbH, 15. Okt. 2007
- *KUKA.CR Motion Cooperation 2.1*. KUKA Roboter GmbH, 21. Aug. 2007
- *KUKA.Ethernet KRL XML 1.1*. KUKA Roboter GmbH, 8. März 2007
- *KUKA.Ethernet RSI XML 1.1*. KUKA Roboter GmbH, 23. Nov. 2007
- *KUKA.GlueTech 3.2*. KUKA Roboter GmbH, 26. Nov. 2007
- *KUKA.Gripper&SpotTech 2.3*. KUKA Roboter GmbH, 24. Mai 2007
- *KUKA.Laser RPFO*. KUKA Roboter GmbH, 17. Dez. 2007
- *KUKA.LaserTech 2.0*. KUKA Roboter GmbH, 17. Dez. 2007
- *KUKA.Occubot VI V2.0*. KUKA Roboter GmbH, 23. Jan. 2006
- *KUKA.PalletTech V3.4*. KUKA Roboter GmbH, 2. März 2006
- *KUKA.PLC Multiprog*. KUKA Roboter GmbH, 23. Nov. 2007
- *KUKA.RobotSensorInterface 2.1*. KUKA Roboter GmbH, 3. Mai 2007
- *KUKA.SeamTech SRT 1.1*. KUKA Roboter GmbH, 26. Nov. 2007
- *KUKA.ServoGun TC 2.0*. KUKA Roboter GmbH, 11. Juni 2007
- *KUKA TouchSense 1.1*. KUKA Roboter GmbH, 1. März 2006
- *PlastTech Version 2.2*. KUKA Roboter GmbH, 26. Juli 2002

Folgende Technologiepakete der Firma MRK Systeme GmbH wurden anhand interner Dokumentationen und persönlicher Gespräche mit Experten untersucht (in alphabetischer Reihenfolge):

- *SafeGuiding*. MRK Systeme GmbH. URL: http://www.mrk-systeme.de/produkte_guiding.html (besucht am 22. 11. 2014)
- *SafeInteraction*. MRK Systeme GmbH. URL: http://www.mrk-systeme.de/produkte_interaction.html (besucht am 22. 11. 2014)

B

VERWENDETE ROBOTER UND ENDEFFEKTOREN

Folgende Manipulatoren bzw. Roboter wurden real und simuliert für diese Arbeit verwendet (in alphabetischer Reihenfolge):

- KUKA KR 6 R900 sixx AGILUS
  - simuliert für die Evaluation einzelner Strategien
- KUKA Leichtbauroboter 4+ (LBR)
  - real für die Umsetzung der Factory 2020
  - simuliert für die Umsetzung der Factory 2020
- Schunk Lightweightarm 4.6 (LWA)
  - real für die Evaluation einzelner Strategien
  - simuliert für die Umsetzung der Factory 2020
- Staubli TX 90 L
  - simuliert für die Evaluation einzelner Strategien

Folgende Endeffektoren bzw. Roboterwerkzeuge wurden real und simuliert für diese Arbeit verwendet (in alphabetischer Reihenfolge):

- Kolver Pluto elektrisches Schraubsystem (Steuerung: Pluto 10 CA/N)
  - real für die Umsetzung der Factory 2020
- Schunk MEG 50 elektrischer Parallelgreifer
  - real für die Umsetzung der Factory 2020
  - simuliert für die Umsetzung der Factory 2020
  - real für die Evaluation einzelner Strategien
  - simuliert für die Evaluation einzelner Strategien
- Schunk WSG 50 elektrischer Parallelgreifer
  - simuliert für die Umsetzung der Factory 2020
  - real für die Evaluation einzelner Strategien
  - simuliert für die Evaluation einzelner Strategien

Folgende mobile Plattform wurde real und simuliert für diese Arbeit verwendet:

- Segway RMP 50
  - real für die Umsetzung der Factory 2020

C

STUDIEN- UND ABSCHLUSSARBEITEN

Folgende von mir betreute Abschlussarbeiten haben zur vorliegenden Dissertation beigetragen (in chronologischer Reihenfolge):

- Markus Stehle. „Analyse und Design einer Softwarearchitektur für die automatisierte Handhabung von trockenen CFK-Textilien“. Diplomarbeit. Universität Augsburg, 2009
- Matthias Frey. „Entwurf und Implementierung einer service-orientierten Anwendung zur automatisierten Handhabung von Kohlefasertextilien“. Bachelorarbeit. Universität Augsburg, 16. Dez. 2009
- Henrik Mühe. „Interpretation von KRL auf Basis der Softrobot API“. Masterarbeit. Universität Augsburg, 21. Aug. 2010
- Markus Bickelmaier. „Entwurf und Umsetzung einer Anwendung zur roboterbasierten Fertigung von Kohlenstofffaserbauteilen“. Diplomarbeit. Universität Augsburg, 28. Okt. 2010
- Miroslav Macho. „Service-orientierte Steuerung einer Roboterzelle zur automatisierten Fertigung von carbonfaserverstärkten Verbundbauteilen“. Bachelorarbeit. Universität Augsburg, 20. Apr. 2011
- Sebastian Pieniack. „Entwicklung eines service-orientierten Modellierungsansatzes für Industrieroboterzellen“. Masterarbeit. Universität Augsburg, 25. Juli 2011
- Matthias Frey. „Steuerung und Überwachung von robotergesteuerten Automatisierungssystemen mit State Charts“. Masterarbeit. Universität Augsburg, 24. Mai 2012
- Christian Böck. „Entwicklung eines Marktplatzes für Robotikanwendungen“. Betreuung zusammen mit A. Angerer. Diplomarbeit. Universität Augsburg, 30. Jan. 2013
- Andreas Osipov. „Konzeption und Entwicklung einer Visualisierungsplattform für die Offline-Programmierung von Robotern“. Betreuung zusammen mit A. Habermaier. Bachelorarbeit. Universität Augsburg, 22. Feb. 2013
- Miroslav Macho. „Entwurf einer Softwareplattform für die Offline-Programmierung von Robotern zur Herstellung von CFK-Bauteilen“. Betreuung zusammen mit H. Seebach. Masterarbeit. Universität Augsburg, 9. Aug. 2013
- Wladislaw Stawrow. „Modellierung und Implementierung sensorgestützter, fehlertoleranter Roboteraktionen in der Montage“. Bachelorarbeit. Universität Augsburg, 30. Sep. 2013
- Sascha Jocham. „Automatisches Erkennen von Kollisionen in der Roboterofflineprogrammierung“. Bachelorarbeit. Universität Augsburg, 30. Sep. 2013

- Samir WAFA. „Efficient Planning Of Collision-Free Robot Motions“. Betreuung zusammen mit A. Schierl. Masterarbeit. Universität Augsburg, 30. Nov. 2014

Außerdem wurden von mir folgende Studienarbeiten betreut, die ebenfalls zur vorliegenden Dissertation beigetragen haben (in chronologischer Reihenfolge):

- Wladislaw STAWROW. „Service-orientierte Programmierung von Roboterzellen mit Activiti“. Praxismodul. 2011
- Jürgen HAUG. „Visualisierung aktueller Fertigungsdaten einer Roboterzelle in Eclipse“. Praxismodul. 2012
- Matthias STÜBEN. „Service-orientierte Programmierung von Roboterzellen mit jBPM“. Praxismodul. 2012

D

EIGENE PUBLIKATIONEN

Einige Ideen, Überlegungen und Ansätze dieser Arbeit wurden bereits früher in den folgenden Publikationen vorgestellt (in chronologischer Reihenfolge):

1. Alwin Hoffmann, Florian Nafz, Frank Ortmeier, Andreas Schierl und Wolfgang Reif. „Prototyping Plant Control Software with Microsoft Robotics Studio“. In: *Software Development and Integration in Robotics. SDIR-III. Workshop at the 2008 IEEE Intl. Conference on Robotics and Automation*. Proceedings. (Pasadena, USA, 20. Mai 2008). IEEE. 2008

2. Frank Ortmeier, Alwin Hoffmann, Wolfgang Reif, Ulrich Huggenberger und Thomas Stumpfegger. „Simulations-basierte Programmierung von Industrierobotern“. In: *Internationales Forum Mechatronik. IFM 2008*. Tagungsband. (Stuttgart, 22.–25. Sep. 2008). 2008

3. Andreas Angerer, Alwin Hoffmann, Frank Ortmeier, Michael Vistein und Wolfgang Reif. „Object-Centric Programming: A New Modeling Paradigm for Robotic Applications“. In: *2009 IEEE Intl. Conference on Automation and Logistics. ICAL 2009*. Proceedings. (Shenyang, China, 5.–7. Aug. 2009). IEEE. 2009, S. 18–23

4. Alwin Hoffmann, Andreas Angerer, Frank Ortmeier, Michael Vistein und Wolfgang Reif. „Hiding Real-Time: A new Approach for the Software Development of Industrial Robots“. In: *2009 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems. IROS 2009*. Proceedings. (St. Louis, MO, USA, 11.–15. Okt. 2009). IEEE. 2009, S. 2108–2113

5. Alwin Hoffmann, Florian Nafz, Hella Seebach, Andreas Schierl und Wolfgang Reif. „Developing Self-Organizing Robotic Cells using Organic Computing Principles“. In: *Bio-Inspired Self-Organizing and Self-Reconfigurable Robotic Systems. Workshop at the 2010 IEEE Intl. Conference on Robotics and Automation*. Proceedings. (Anchorage, AL, USA, 3. Mai 2010). IEEE. 2010

6. Alwin Hoffmann, Andreas Angerer, Andreas Schierl, Michael Vistein und Wolfgang Reif. „Software Engineering in der Industrierobotik: der SoftRobot-Ansatz“. In: *Entwurf komplexer Automatisierungssysteme. EKA 2010. Beschreibungmittel, Methoden, Werkzeuge und Anwendungen*. Tagungsband. (Magdeburg, 25.–27. Mai 2010). 2010

7. Alwin Hoffmann, Andreas Angerer, Andreas Schierl, Michael Vistein und Wolfgang Reif. „Towards Object-Oriented Software Development for Industrial Robots“. In: *7th Intl. Conference on Informatics in Control, Automation and Robotics. ICINCO 2010*. Proceedings. (Funchal, Madeira, Portugal, 15.–18. Juni 2010). Hrsg. von Joaquim Filipe, Juan Andrade-Cetto und Jean-Louis Ferrier. Bd. 2. INSTICC. 2010, S. 437–440

8. Michael Vistein, Andreas Angerer, Alwin Hoffmann, Andreas Schierl und Wolfgang Reif. „Interfacing Industrial Robots using

Realtime Primitives“. In: *2010 IEEE Intl. Conference on Automation and Logistics. ICAL 2010*. Proceedings. (Hong Kong, China, 16.–20. Aug. 2010). IEEE. 2010, S. 468–473

9. Andreas Angerer, Claudia Ehinger, Alwin Hoffmann, Wolfgang Reif, Gunther Reinhart und Gerhard Strasser. „Automated Cutting and Handling of Carbon Fiber Fabrics in Aerospace Industries“. In: *2010 IEEE Conference on Automation Science and Engineering. CASE 2010*. Proceedings. (Toronto, ON, Canada, 21.–24. Aug. 2010). IEEE, 2010, S. 861–866

10. Henrik Mühe, Andreas Angerer, Alwin Hoffmann und Wolfgang Reif. „On reverse-engineering the KUKA Robot Language“. In: *Domain-Specific Languages and models for ROBotic systems. DSLRob '10. Workshop at the 2010 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems*. Proceedings. (Taipei, Taiwan, 22. Okt. 2010). 2010. URL: http://arxiv.org/abs/1009.5004

11. Andreas Angerer, Alwin Hoffmann, Andreas Schierl, Michael Vistein und Wolfgang Reif. „The Robotics API: An Object-Oriented Framework for Modeling Industrial Robotics Applications“. In: *2010 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems. IROS 2010*. Proceedings. (Taipei, Taiwan, 18.–22. Okt. 2010). IEE, 2010, S. 4036–4041

12. Andreas Angerer, Markus Bischof, Alexander Chekler, Alwin Hoffmann, Wolfgang Reif, Andreas Schierl, Christian Tarragona und Michael Vistein. „Objektorientierte Programmierung von Industrierobotern“. In: *Internationales Forum Mechatronik. IFM 2010*. Tagungsband. (Winterthur, Schweiz, 3.–4. Nov. 2010). ZHAW, IMS Institut für Mechatronische Systeme. 2010

13. Alwin Hoffmann, Florian Nafz, Hella Seebach, Andreas Schierl und Wolfgang Reif. „Developing Self-Organizing Robotic Cells using Organic Computing Principles“. In: *Bio-Inspired Self-Organizing Robotic Systems*. Hrsg. von Yan Meng und Yaochu Jin. Bd. 355. Studies in Computational Intelligence. Berlin: Springer, 2011, S. 253–274

14. Andreas Angerer, Claudia Ehinger, Alwin Hoffmann, Wolfgang Reif und Gunther Reinhart. „Design of an Automation System for Preforming Processes in Aerospace Industries“. In: *2011 IEEE Conference on Automation Science and Engineering. CASE 2011*. Proceedings. (Trieste, Italy, 24.–27. Aug. 2011). Nominated for IEEE Best Paper Award. IEEE. 2011, S. 557–562

15. Alwin Hoffmann, Andreas Angerer, Andreas Schierl, Michael Vistein und Wolfgang Reif. „Service-orientierte Modellierung einer Robotermontagezelle“. In: *Internationales Forum Mechatronik. IFM 2011*. Tagungsband. (Cham, 21.–22. Sep. 2011). Cluster Mechatronik & Automation e.V. 2011, S. 179–191

16. Dominik Haneberg, Alwin Hoffmann, Hella Seebach, Michael Vistein und Wolfgang Reif. „Towards Model-based Evolution for Robot-based Automation“. In: *Modellierung in der Automatisierungstechnik. (MAT 2012), Workshop im Rahmen der Modellierung 2012*. Tagungsband. (Bamberg, 15. März 2012). 2012

17. Andreas Schierl, Andreas Angerer, Alwin Hoffmann, Michael Vistein und Wolfgang Reif. „Using Java for Real-Time Critical Industrial Robot Programming“. In: *Software Development and Integration in Robotics. SDIR-VII. Workshop at the 2012 IEEE Intl. Conference on Robotics and Automation*. Proceedings. (St. Paul, MN, USA, 14. Mai 2012). IEEE. 2012

18. Andreas Angerer, Andreas Bareth, Alwin Hoffmann, Andreas Schierl, Michael Vistein und Wolfgang Reif. „Two-arm Robot Teleoperation Using a Multi-touch Tangible User Interface“. In: *9th Intl. Conference on Informatics in Control, Automation and Robotics. ICINCO 2012*. Proceedings. (Rome, Italy, 28.–31. Juli 2012). Hrsg. von Jean-Louis Ferrier, Alain Bernard, Oleg Yu. Gusikhin und Kurosh Madani. INSTICC. 2012

19. Andreas Schierl, Andreas Angerer, Alwin Hoffmann, Michael Vistein und Wolfgang Reif. „From Robot Commands To Real-Time Robot Control – Transforming High-Level Robot Commands into Real-Time Dataflow Graphs“. In: *9th Intl. Conference on Informatics in Control, Automation and Robotics. ICINCO 2012*. Proceedings. (Rome, Italy, 28.–31. Juli 2012). Hrsg. von Jean-Louis Ferrier, Alain Bernard, Oleg Yu. Gusikhin und Kurosh Madani. INSTICC. 2012, S. 1–6

20. Michael Vistein, Andreas Angerer, Alwin Hoffmann, Andreas Schierl und Wolfgang Reif. „Instantaneous switching between real-time commands for continuous execution of complex robotic tasks“. In: *2012 IEEE Intl. Conference on Mechatronics and Automation. ICMA 2012*. Proceedings. (Chengdu, China, 5.–8. Aug. 2012). IEEE. 2012, S. 1329–1334

21. Andreas Angerer, Alwin Hoffmann, Andreas Schierl, Remi Smirra, Michael Vistein und Wolfgang Reif. „A Graphical Language for Real-Time Critical Robot Commands“. In: *Domain-Specific Languages and models for ROBotic systems. DSLRob '12. Workshop at the 2012 Simulation, Modeling, and Programming for Autonomous Robots*. Proceedings. (Tsukuba, Japan, 5. Nov. 2012). 2012. URL: http://arxiv.org/abs/1302.5082

22. Andreas Angerer, Christian Böck, Alwin Hoffmann, Andreas Schierl, Michael Vistein und Wolfgang Reif. „Eclipse als Werkzeug zur objektorientierten Roboterprogrammierung“. In: *Internationales Forum Mechatronik. IFM 2012*. Tagungsband. (Mayerhofen, Österreich, 21.–22. Nov. 2012). Cluster Mechatronik Tirol. 2012

23. Andreas Schierl, Alwin Hoffmann, Andreas Angerer, Michael Vistein und Wolfgang Reif. „Towards Realtime Robot Reactions: Patterns for Modular Device Driver Interfaces“. In: *Software Development and Integration in Robotics. SDIR-VIII. Workshop at the 2013 IEEE Intl. Conference on Robotics and Automation*. Proceedings. (Karlsruhe, Germany, 6. Mai 2013). IEEE. 2013

24. Andreas Angerer, Alwin Hoffmann, Andreas Schierl, Michael Vistein und Wolfgang Reif. „Robotics API: Object-Oriented Software Development for Industrial Robots“. In: *Journal of Software Engineering for Robotics* 4.1 (2013), S. 1–22

25. Alwin Hoffmann, Andreas Angerer, Andreas Schierl, Michael Vistein und Wolfgang Reif. „Managing Extensibility and Maintainability of Industrial Robotics Software“. In: *16th Intl. Conference on Advanced Robotics. ICAR 2013*. Proceedings. (Montevideo, Uruguay, 25.–29. Nov. 2013). IEEE. 2013, S. 1–7

26. Michael Vistein, Andreas Angerer, Alwin Hoffmann, Andreas Schierl und Wolfgang Reif. „Flexible and Continuous Execution of Real-Time Critical Robotic Tasks“. In: *Intl. Journal of Mechatronics and Automation* 4.1 (2014)

27. Alwin Hoffmann, Andreas Angerer, Andreas Schierl, Michael Vistein und Wolfgang Reif. „Service-oriented Robotics Manufacturing by reasoning about the Scene Graph of a Robotics Cell“. In: *45th International Symposium on Robotics (ISR 2014) and the 8th German Conference on Robotics (Robotik 2014)*. Proceedings. (Munich, Germany, 2.–3. Juni 2014). Berlin: VDE Verlag, 2014, S. 756–763

28. Sven Stumm, Alwin Hoffmann, Hella Seebach, Bernd Kuhlenkötter und Wolfgang Reif. „Towards combining layout and process models for mixed assembly facilities“. In: *AUTOMATION 2014*. Tagungsband. (Baden-Baden, 1.–2. Juli 2014). Bd. 2231. VDI-Berichte. VDI Verlag, Aug. 2014

29. Ludwig Nägele, Miroslav Macho, Andreas Angerer, Alwin Hoffmann, Michael Vistein, Manfred Schönheits und Wolfgang Reif. „A backward-oriented approach for offline programming of complex manufacturing tasks“. In: *6th Intl. Conference on Automation, Robotics and Applications. ICARA 2015*. Proceedings. (Queenstown, New Zealand, 17.–19. Feb. 2015). IEEE. 2015

[1] ABB. *Bedienungsanleitung RobotStudio 5.61*. 23. Apr. 2014.

[2] Activiti. *Activiti BPM Platform*. URL: http://www.activiti.org/ (besucht am 28. 12. 2014).

[3] M. Adamski. „SFC, Petri nets and application specific logic controllers“. In: *1998 IEEE Intl. Conference on Systems, Man, and Cybernetics*. Proceedings. (San Diego, CA, USA, 11.–14. Okt. 1998). Bd. 1. IEEE, Okt. 1998, S. 728–733.

[4] Sang Chul Ahn, Jin Hak Kim, Kiwoong Lim, Heedong Ko, Yong-Moo Kwon und Hyoung-Gon Kim. „UPnP Approach for Robot Middleware“. In: *2005 IEEE Intl. Conference on Robotics and Automation. ICRA 2005*. Proceedings. (Barcelona, Spain, 18.–22. Apr. 2005). IEEE, 2005, S. 1959–1963.

[5] Rachid Alami, Raja Chatila, Sara Fleury, Malik Ghallab und Félix Ingrand. „An architecture for autonomy“. In: *The International Journal of Robotics Research* 17.4 (1998), S. 315–337.

[6] Alin Albu-Schäffer, Christian Ott und Gerd Hirzinger. „A Unified Passivity-based Control Framework for Position, Torque and Impedance Control of Flexible Joint Robots“. In: *Int. Journal Robotics Research* 26.1 (2007), S. 23–39.

[7] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid F. King und Shlomo Angel. *Eine Muster-Sprache. Städte, Gebäude, Konstruktion*. Übers. von Hermann Czech. Wien: Löcker Verlag, 1995.

[8] Noriaki Ando, Takashi Suehiro, Kosei Kitagaki und Tetsuo Kotoku. „RT(Robot Technology)-Component and its Standardization - Towards Component Based Networked Robot Systems Development“. In: *SICE-ICASE Intl. Joint Conference 2006. SICE-ICCAS 2006*. Proceedings. (Busan, Korea, 18.–21. Okt. 2006). IEEE, 2006, S. 2633–2638.

[9] Noriaki Ando, Takashi Suehiro und Tetsuo Kotoku. „A Software Platform for Component Based RT-System Development: OpenRTM-Aist“. In: *Simulation, Modeling, and Programming for Autonomous Robots. First International Conference, SIMPAR 2008*. Proceedings. (Venice, Italy, 3.–6. Nov. 2008). Hrsg. von Stefano Carpin, Itsuki Noda, Enrico Pagello, Monica Reggiani und Oskar von Stryk. Bd. 5325. Lecture Notes of Computer Science. Springer, 2008, S. 87–98.

[10] Andreas Angerer. „Object-oriented Software for Industrial Robots“. Diss. Universität Augsburg, 28. Mai 2014. URL: http://opus.bibliothek.uni-augsburg.de/opus4/files/3064/Dissertation_Angerer.pdf (besucht am 30. 06. 2015).

[11] Andreas Angerer, Andreas Bareth, Alwin Hoffmann, Andreas Schierl, Michael Vistein und Wolfgang Reif. „Two-arm Robot Teleoperation Using a Multi-touch Tangible User Interface“. In: *9th Intl. Conference on Informatics in Control, Automation and Robotics. ICINCO 2012*. Proceedings. (Rome, Italy, 28.–31. Juli 2012). Hrsg. von Jean-Louis Ferrier, Alain Bernard, Oleg Yu. Gusikhin und Kurosh Madani. INSTICC. 2012.

[12] Andreas Angerer, Markus Bischof, Alexander Chekler, Alwin Hoffmann, Wolfgang Reif, Andreas Schierl, Christian Tarragona und Michael Vistein. „Objektorientierte Programmierung von Industrierobotern“. In: *Internationales Forum Mechatronik. IFM 2010*. Tagungsband. (Winterthur, Schweiz, 3.–4. Nov. 2010). ZHAW, IMS Institut für Mechatronische Systeme. 2010.

[13] Andreas Angerer, Christian Böck, Alwin Hoffmann, Andreas Schierl, Michael Vistein und Wolfgang Reif. „Eclipse als Werkzeug zur objektorientierten Roboterprogrammierung“. In: *Internationales Forum Mechatronik. IFM 2012*. Tagungsband. (Mayerhofen, Österreich, 21.–22. Nov. 2012). Cluster Mechatronik Tirol. 2012.

[14] Andreas Angerer, Claudia Ehinger, Alwin Hoffmann, Wolfgang Reif und Gunther Reinhart. „Design of an Automation System for Preforming Processes in Aerospace Industries“. In: *2011 IEEE Conference on Automation Science and Engineering. CASE 2011*. Proceedings. (Trieste, Italy, 24.–27. Aug. 2011). Nominated for IEEE Best Paper Award. IEEE. 2011, S. 557–562.

[15] Andreas Angerer, Claudia Ehinger, Alwin Hoffmann, Wolfgang Reif, Gunther Reinhart und Gerhard Strasser. „Automated Cutting and Handling of Carbon Fiber Fabrics in Aerospace Industries“. In: *2010 IEEE Conference on Automation Science and Engineering. CASE 2010*. Proceedings. (Toronto, ON, Canada, 21.–24. Aug. 2010). IEEE, 2010, S. 861–866.

[16] Andreas Angerer, Alwin Hoffmann, Frank Ortmeier, Michael Vistein und Wolfgang Reif. „Object-Centric Programming: A New Modeling Paradigm for Robotic Applications“. In: *2009 IEEE Intl. Conference on Automation and Logistics. ICAL 2009*. Proceedings. (Shenyang, China, 5.–7. Aug. 2009). IEEE. 2009, S. 18–23.

[17] Andreas Angerer, Alwin Hoffmann, Andreas Schierl, Remi Smirra, Michael Vistein und Wolfgang Reif. „A Graphical Language for Real-Time Critical Robot Commands“. In: *Domain-Specific Languages and models for ROBotic systems. DSLRob '12. Workshop at the 2012 Simulation, Modeling, and Programming for Autonomous Robots*. Proceedings. (Tsukuba, Japan, 5. Nov. 2012). 2012. URL: http://arxiv.org/abs/1302.5082.

[18] Andreas Angerer, Alwin Hoffmann, Andreas Schierl, Michael Vistein und Wolfgang Reif. „Robotics API: Object-Oriented Software Development for Industrial Robots“. In: *Journal of Software Engineering for Robotics* 4.1 (2013), S. 1–22.

[19] Andreas Angerer, Alwin Hoffmann, Andreas Schierl, Michael Vistein und Wolfgang Reif. „The Robotics API: An Object-Oriented Framework for Modeling Industrial Robotics Applications“. In: *2010 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems. IROS 2010*. Proceedings. (Taipei, Taiwan, 18.–22. Okt. 2010). IEE, 2010, S. 4036–4041.

[20] Apache Software Foundation. *SCXML – Commons SCXML*. URL: http://commons.apache.org/proper/commons-scxml/ (besucht am 28. 12. 2014).

[21] Apache Software Foundation. *Welcome to Apache Felix*. URL: http://felix.apache.org/ (besucht am 22. 10. 2014).

[22] Apache Software Foundation. *Welcome to The Apache Software Foundation*. URL: http://www.apache.org/ (besucht am 28. 12. 2014).

[23] P. Banerjee, R. Friedrich, C. Bash, P. Goldsack, B.A. Huberman, J. Manley, C. Patel, P. Ranganathan und A. Veitch. „Everything as a Service: Powering the New Information Economy“. In: *Computer* 44.3 (März 2011), S. 36–43.

[24] Josef Papenfort Barbosa und Paulo Leitão. „Enhancing Service-Oriented Holonic Multi-Agent Systems with Self-organization“. In: *Intl. Conference on Industrial Engineering and Systems Management. IESM 2011*. Proceedings. (Metz, France, 25.–27. Mai 2011). 2011, S. 1373–1381.

[25] Mark Barnes und Ellen Levy Finch, Hrsg. *COLLADA - Digital Asset Schema Release*. Specification. Version 1.5.0. The Khronos Group, Apr. 2008. URL: https://www.khronos.org/files/collada_spec_1_5.pdf (besucht am 04. 12. 2014).

[26] Jim Barnett, Rahul Akolkar, R. J. Auburn, Michael Bodell, Daniel C. Burnett, Jerry Carter, Scott McGlashan, Torbjörn Lager, Mark Helbing, Rafah Hosn, T. V. Raman, Klaus Reifenrath, No'am Rosenthal und Johan Roxendal, Hrsg. *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. W3C Last Call Working Draft. World Wide Web Consortium (W3C), 29. Mai 2014. URL: http://www.w3.org/TR/scxml/ (besucht am 28. 12. 2014).

[27] Berthold Bäuml und Gerd Hirzinger. „When Hard Realtime Matters: Software for Complex Mechatronic Systems“. In: *Robotics and Autonomous Systems* 56.1 (2008), S. 5–13.

[28] Leandro B. Becker und Carlos E. Pereira. „SIMOO-RT – An Object Oriented Framework for the Development of Real-Time Industrial Automation Systems“. In: *IEEE Transactions on Robotics and Automation* 18.4 (Aug. 2002), S. 421–430.

[29] Albert Benveniste und Gérard Berry. „The synchronous approach to reactive and real-time systems“. In: *Proceedings of the IEEE* 79.9 (Sep. 1991), S. 1270–1282.

[30] Gérard Berry und Georges Gonthier. „The ESTEREL synchronous programming language: design, semantics, implementation“. In: *Science of Computer Programming* 19.2 (1992), S. 87–152.

[31] Luigi Biagiotti und Claudio Melchiorri. *Trajectory Planning for Automatic Machine and Robots*. Berlin: Springer, 2008.

[32] Rainer Bischoff und Tim Guhl. „The Strategic Research Agenda for Robotics in Europe“. In: *IEEE Robotics & Automation Magazine* 17.1 (März 2010), S. 15–16.

[33] Rainer Bischoff, Johannes Kurth, Günter Schreiber, Ralf Koeppe, Alin Albu-Schäffer, Alexander Beyer, Oliver Eiberger, Sami Haddadin, Andreas Stemmer, Gerhard Grunwald und Gerhard Hirzinger. „The KUKA-DLR Lightweight Robot arm - A new reference platform for robotics research and manufacturing“. In: *41st International Symposium on Robotics (ISR 2010) and the 6th German Conference on Robotics (Robotik 2010)*. Proceedings. (Munich, Germany, 7.–9. Juni 2010). VDE Verlag, 2010, S. 1–8.

[34] S.R. Biyabani, J.A. Stankovic und K. Ramamritham. „The integration of deadline and criticalness in hard real-time scheduling“. In: *9th Real-Time Systems Symposium. RTSS 88*. Proceedings. (Huntsville, AL, USA, 6.–8. Dez. 1988). IEEE Computer Society. 1998, S. 152–160.

[35] Anders Blomdell, Gunnar Bolmsjo, Torgny Brogardh, Per Cederberg, Mats Isaksson, Rolf Johansson, Mathias Haage, Klas Nilsson, Magnus Olsson, Tomas Olsson, Anders Robertsson und Jianjun Wang. „Extending an Industrial Robot Controller – Implementation and Applications of a Fast Open Sensor Interface“. In: *IEEE Robotics & Automation Magazine* 12.3 (2005), S. 85–94.

[36] Christian Blume und Wilfried Jakob. *Programming Languages for Industrial Robots*. Springer-Verlag, 1986.

[37] Sebastian Blumenthal, Herman Bruyninckx, Walter Nowak und Erwin Prassler. „A scene graph based shared 3D world model for robotic applications“. In: *2013 IEEE Intl. Conference on Robotics and Automation. ICRA 2013*. Proceedings. (Karlsruhe, Germany, 16.–10. Mai 2013). IEEE. 2013, S. 453–460.

[38] H. Bohn, A. Bobek und F. Golatowski. „SIRENA – Service Infrastructure for Real-time Embedded Networked Devices: A service oriented framework for different domains“. In: *Intl. Conference on Networking, Intl. Conference on Systems, Intl. Conference on Mobile Communications and Learning Technologies. ICN 2006, ICONS 2006, MCL 2006*. Proceedings. (Morne, Mauritius, 23.–29. Apr. 2006). IEEE, IARIA. 2006, S. 43.

[39] R. Peter Bonasso. „Integrating reaction plans and layered competences through synchronous control“. In: *12th Intl. Joint Conference on Artificial Intelligence. IJCAI 91*. Proceedings. (Tblisi, USSR, 24.–30. Aug. 1991). Bd. 2. Morgan Kaufmann, 1991, S. 1225–1231.

[40] R. Peter Bonasso, R. James Firby, Erann Gat, David Kortenkamp, David P. Miller und Mark G. Slack. „Experiences with an architecture for intelligent, reactive agents“. In: *Journal of Experimental & Theoretical Artificial Intelligence* 9.2/3 (1997), S. 237–256.

[41] Grady Booch, James Rumbaugh und Ivar Jacobson. *Unified Modeling Language User Guide*. Addison Wesley, 1999.

[42] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris und David Orchard, Hrsg. *Web Services Architecture*. W3C Working Group Note. World Wide Web Consortium (W3C), 11. Feb. 2004. URL: http://www.w3.org/TR/ws-arch/ (besucht am 28. 12. 2014).

[43] Jean-Jacques Borrelly, Eve Coste-Manière, Bernard Espiau, Konstantinos Kapellos, Roger Pissard-Gibollet, Daniel Simon und Nicolas Turro. „The ORCCAD Architecture“. In: *Intl. Journal of Robotics Research* 17.4 (Apr. 1998), S. 338–359.

[44] Jan Bosch, Clemens Szyperski und Wolfgang Week. „WCOP 98 – Summary of the Third International Workshop on Component-Oriented Programming“. In: *Object-Oriented Technology: ECOOP 98 Workshop Reader*. Hrsg. von Serge Demeyer und Jan Bosch. Bd. 1543. Lecture Notes in Computer Science. Berlin: Springer, 1998, S. 130–135.

[45] Jan Bosch, Wolfgang Weck und Clemens Szyperski. „2nd Workshop on Component-Oriented Programming (WCOP 97)“. In: *Object-Oriented Technology: ECOOP 97 Workshop Reader*. Hrsg. von Jan Bosch und Stuart Mitchell. Bd. 1357. Lecture Notes in Computer Science. Berlin: Springer, 1998, S. 323–326.

[46] Alex Brooks, Tobias Kaupp, Alexei Makarenko, Stefan Williams und Anders Orebäck. „Orca: A component model and repository.“ In: *Software Engineering for Experimental Robotics*. Hrsg. von Davide Brugali. Springer Tracts in Advanced Robotics 30. Berlin: Springer, 2007, S. 231–251.

[47] Rodney A. Brooks. „A robust layered control system for a mobile robot“. In: *IEEE Journal of Robotics and Automation* 2.1 (März 1986), S. 14–23.

[48] Davide Brugali, Hrsg. *Software Engineering for Experimental Robotics*. Springer Tracts in Advanced Robotics 30. Berlin: Springer, 2007.

[49] Davide Brugali und Patrizia Scandurra. „Component-Based Robotic Engineering (Part I)“. In: *IEEE Robotics & Automation Magazine* 16.4 (Dez. 2009), S. 84–96.

[50] Herman Bruyninckx. „Open robot control software: the OROCOS project“. In: *2001 IEEE Intl. Conference on Robotics and Automation. ICRA 2001*. Proceedings. (Seoul, Korea, 21.–26. Mai 2001). IEEE, 2001, S. 2523–2528.

[51] Bullet Physics Library. *Real-Time Physics Simulation*. URL: http://www.bulletphysics.org/ (besucht am 22. 12. 2014).

[52] Bundesministerium für Bildung und Forschung. *Industrie 4.0*. 2015. URL: http://www.hightech-strategie.de/de/Industrie-4-0-59.php (besucht am 03. 01. 2015).

[53] Wolfram Burgard und Martial Hebert. „World Modeling“. In: *Springer Handbook of Robotics*. Hrsg. von Bruno Siciliano und Oussama Khatib. Berlin: Springer, 2008. Kap. 36, S. 853–869.

[54] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal. *Pattern-Oriented Software Architecture: A system of Patterns*. Bd. 1. Chichester: John Wiley & Sons, 1996.

[55] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*. Boston, MA, USA: Kluwer Academic Publishers, 1997.

[56] CAN in Automation e. V. *CANopen – Application Layer and Communication Profile*. CiA Draft Standard DS301. Version 4.2. 2007.

[57] A. Cannata, M. Gerosa und M. Taisch. „SOCRADES: A framework for developing intelligent systems in manufacturing“. In: *IEEE Intl. Conference on Industrial Engineering and Engineering Management. IEEM 2008*. Proceedings. (Singapore, 8.–11. Dez. 2008). 2008, S. 1904–1908.

[58] S. Chhaniyara, C. Saaj, B. Maediger, M. Althoff-Kotzias, B. Langpap und I. Ahrns. „SysML based system engineering: A case study for Space Robotic Systems“. In: *62nd International Astronautical Congress. IAC*. Proceedings. (Cape Town, South Africa, 3.–7. Okt. 2011). International Astronautical Federation. 2011, S. 1–8.

[59] Eric Clayberg und Dan Rubel. *Eclipse Plug-ins*. 3. Aufl. the eclipse series. Upper Saddle River, NJ: Addison Wesley, Dez. 2008.

[60] Toby Collett, Bruce MacDonald und Brian Gerkey. „Player 2.0: Toward a Practical Robot Programming Framework“. In: *Australasian Conference on Robotics and Automation 2005. ACRA 05*. Proceedings. (Sydney, Australia, 5.–7. Dez. 2005). 2005, S. 12–19.

[61] E. Coste-Maniere und N. Turro. „The MAESTRO language and its environment: specification, validation and control of robotic missions“. In: *1997 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems. IROS '97*. Proceedings. (Grenoble, France, 7.–11. Sep. 1997). Bd. 2. IEEE. 1997, S. 836–841.

[62] Carle Côté, Dominic Létourneau, Clément Raïevsky, Yannick Brosseau und François Michaud. „Using MARIE for Mobile Robot Component Development and Integration“. In: *Software Engineering for Experimental Robotics*. Hrsg. von Davide Brugali. Springer Tracts in Advanced Robotics 30. Berlin: Springer, 2007, S. 211–230.

[63] John J. Craig. *Introduction to Robotics: Mechanics and Control*. 3. Auflage. Upper Saddle River, NJ: Pearson Prentice Hall, 2005.

[64] Dassault Systèmes. *DELMIA Robotics Offline Programming*. url: http://www.3ds.com/products-services/delmia/products/robotics-programmers/robotics-offline-programming/ (besucht am 30. 10. 2014).

[65] Eric Dégoulange und Pierre Dauchez. „External force control of an industrial Puma 560 robot“. In: *Journal of Robotic Systems* 11.6 (1994), S. 523–540.

[66] Edsger Dijkstra. „A note on two problems in connexion with graphs“. In: *Numerische Mathematik* 1 (1959), S. 269–271.

[67] Rüdiger Dillmann und Martin Huck. *Informationsverarbeitung in der Robotik*. Berlin: Springer, 1991.

[68] DIN Deutsches Institut für Normung e.V. „Fertigungsverfahren – Begriffe, Einteilung“. Deutsche Norm DIN 8580:2003-09. Sep. 2003.

[69] DIN Deutsches Institut für Normung e.V. „Fertigungsverfahren Fügen – Teil 0: Allgemeines; Einordnung, Unterteilung, Begriffe“. Deutsche Norm DIN 8593-0:2003-09. Sep. 2003.

[70] DIN Deutsches Institut für Normung e.V. „Fertigungsverfahren Fügen – Teil 1: Zusammensetzen; Einordnung, Unterteilung, Begriffe“. Deutsche Norm DIN 8593-1:2003-09. Sep. 2003.

[71] DIN Deutsches Institut für Normung e.V. „Fertigungsverfahren Fügen – Teil 3: Anpressen, Einpressen; Einordnung, Unterteilung, Begriffe“. Deutsche Norm DIN 8593-3:2003-09. Sep. 2003.

[72] DIN Deutsches Institut für Normung e.V. „Industrial Robot Language“. Deutsche Norm DIN 66312. 1996.

[73] DIN Deutsches Institut für Normung e.V. „Industrieroboter Wörterbuch (ISO 8373:1994)“. Deutsche Fassung EN ISO 8373:1996. Deutsche Norm DIN EN ISO 8373:1996. Aug. 1996.

[74] DIN Deutsches Institut für Normung e.V. „Informationsverarbeitung“. Deutsche Norm DIN 44300. Dokument zurückgezogen. Nov. 1988.

[75] DIN Deutsches Institut für Normung e.V. „Regelungstechnik und Steuerungstechnik. Allgemeine Grundbegriffe“. Deutsche Norm DIN 19226 Teil 1. Feb. 1994.

[76] Dan Driscoll und Antoine Mensch, Hrsg. *Devices Profile for Web Services. OASIS Standard*. Spezifikation. OASIS, 2009-07-01. url: http://docs.oasis-open.org/ws-dd/dpws/wsdd-dpws-1.1-spec.pdf.

[77] Eclipse Foundation. *Eclipse*. url: http://www.eclipse.org/ (besucht am 04. 12. 2014).

[78] Eclipse Foundation. *Equinox*. url: http://www.eclipse.org/equinox/ (besucht am 13. 11. 2014).

[79] Eclipse Foundation. *Graphical Modeling Framework (GMF) Tooling*. url: http://www.eclipse.org/gmf-tooling (besucht am 04. 12. 2014).

[80] Thomas Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Upper Saddle River, NJ: Prentice Hall PTR, 2005.

[81] Heinrich Arnold Ernst. „MH-1, a computer-operated mechanical hand“. Diss. Massachusetts Institute of Technology, 1962.

[82] Bernard Espiau, Konstantinos Kapellos und M. Jourdan. „Formal Verification in Robotics: Why and How?“ In: *7th Intl. Symposium on Robotics Research*. Proceedings. (Herrsching, Germany, 21.–24. Okt. 1995). Hrsg. von Georges Giralt und Gerhard Hirzinger. Herrsching, Germany: Springer, 1995.

[83] T. Estlin, D. Gaines, C. Chouinard, F. Fisher, R. Castano, M. Judd, R.C. Anderson und I. Nesnas. „Enabling autonomous rover science through dynamic planning and scheduling“. In: *2005 IEEE Aerospace Conference*. Proceedings. (Big Sky, USA, 5.–12. März 2005). 2005, S. 385–396.

[84] Raphael Finkel, Russell Taylor, Robert Bolles, Richard Paul und Jerome Feldman. „An Overview of AL, a Programming System for Automation“. In: *4th Intl. Joint Conference on Artificial Intelligence. IJCAI 75*. Proceedings. (Tblisi, USSR, 8.–13. Sep. 1975). Morgan Kaufmann, 1975, S. 758–765.

[85] Bernd Finkemeyer. „Robotersteuerungsarchitektur auf der Basis von Aktionsprimitiven“. Diss. Technische Universität Braunschweig, 2004. Shaker Verlag.

[86] Bernd Finkemeyer, Torsten Kröger, Daniel Kubus, Markus Olschewski und Friedrich M. Wahl. „MiRPA: Middleware for Robotic and Process Control Applications“. In: *Measures and Procedures for the Evaluation of Robot Architectures and Middleware. Workshop at the 2007 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems*. Proceedings. (San Diego, USA, 29. Okt. 2007). Hrsg. von Erwin Prassler, Gerhard Kraetzschmar und Azamat Shakhimardanov. IEEE. 2007, S. 76–90.

[87] Bernd Finkemeyer, Torsten Kröger und Friedrich M. Wahl. „Executing Assembly Tasks Specified by Manipulation Primitive Nets“. In: *Advanced Robotics* 19.5 (2005), S. 591–611.

[88] Bernd Finkemeyer, Torsten Kröger und Friedrich M. Wahl. „The adaptive selection matrix – A key component for sensor-based control of robotic manipulators“. In: *2010 IEEE Intl. Conference on Robotics and Automation. ICRA 2010*. Proceedings. (Anchorage, AL, USA, 3.–8. Mai 2010). IEEE, 2010, S. 3855–3862.

[89] Robert James Firby. „Adaptive Execution in Complex Dynamic Worlds“. Diss. Yale University, 1989.

[90] Paul Fitzpatrick, Giorgio Metta und Lorenzo Natale. „Towards long-lived robot genes“. In: *Robotics & Autonomous systems* 56.1 (Jan. 2008), S. 29–45.

[91] Simon Forge und Colin Blackman. *A Helping Hand for Europe: The Competitive Outlook for the EU Robotics Industry*. JRC Scientific and Technical Reports. European Commision, Joint Research Centre, Institute for Prospective Technological Studies, 2010.

[92] Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. 23. Jan. 2004. url: http://www.martinfowler.com/articles/injection.html (besucht am 22. 11. 2014).

[93] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002.

[94] Georg Frey und Kleanthis Thramboulidis. „Integration of IEC 61131 into Model Driven Development Processes“. In: *AUTOMATION 2011*. Tagungsband. (Baden-Baden, 28.–29. Juni 2011). Bd. 2143. VDI-Berichte. VDI Verlag, 2011.

[95] Sanford Friedenthal, Alan Moore und Rick Steiner. *A Practical Guide to SysML – The Systems Modeling Language*. 3. Aufl. Waltham, USA: Elsevier, 2014.

[96] Rainer Gallus. *VxWin®. Add Windows® XP functionality to VxWorks® on single or multicore CPUs*. White Paper. KUKA Roboter GmbH, Apr. 2008.

[97] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design patterns: Elements of reusable object-oriented software*. Reading, MA, USA: Addison Wesley Longman, 1995.

[98] Erann Gat. „Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots“. In: *10th AAAI National Conference on Artificial Intelligence. AAAI-92*. Proceedings. (San Jose, USA, 12.–16. Juni 1992). 1992, S. 809–815.

[99] Jing Guo Ge und Xing Guo Yin. „An Object Oriented Robot Programming Approach in Robot Served Plastic Injection Molding Application“. In: *Robotic Welding, Intelligence & Automation*. Hrsg. von Tzyh-Jong Tarn, Shan-Ben Chen und Changjiu Zhou. Bd. 362. Lecture Notes in Control & Information Sciences. Berlin: Springer, 2007, S. 91–97.

[100] Eva Geisberger und Manfred Broy, Hrsg. *agendaCPS. Integrierte Forschungsagenda Cyber-Physical Systems*. acatech STUDIE. Berlin: Springer, März 2012.

[101] John Georgas und Richard Taylor. „An Architectural Style Perspective on Dynamic Robotic Architectures“. In: *Software Development and Integration in Robotics. SDIR-II. Workshop at the 2007 IEEE Intl. Conference on Robotics and Automation*. Proceedings. (Rome, Italy, 14. Apr. 2007). IEEE. Rome, Italy, 2007.

[102] Brian P. Gerkey, Richard T. Vaughan, Kasper Stoy, Andrew Howard, Gaurav S. Sukhatme und Maja J. Mataric. „Most Valuable Player: A Robot Device Server for Distributed Control“. In: *2001 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems. IROS 2001*. Proceedings. (Maui, USA, 29. Okt.–3. Nov. 2001). IEEE. 2001, S. 1226–1231.

[103] Tobias Gerngross und Dorothea Nieberl. „Automated manufacturing of large, three-dimensional CFRP parts from dry textiles“. In: *SAMPE Europe Intl. Technical Conference and Table Top. SETEC 14*. Proceedings. (Tampere, Finland, 10.–11. Sep. 2014). SAMPE. 2014.

[104] Matthias Goldhoorn und Sylvain Joyeux. „Extension of a plan-based component manager for real time adaption“. In: *45th International Symposium on Robotics (ISR 2014) and the 8th German Conference on Robotics (Robotik 2014)*. Proceedings. (Munich, Germany, 2.–3. Juni 2014). Berlin: VDE Verlag, 2014, S. 764–769.

[105] Nils Gotha, Klaus Bernzen, Wilfried Plass, Joachim Unfried, Martin Schrott, Roland Schaumburg, Jan Braun, Alfred Möltner, Ryszard Bochniak, Djafar Hadiouche, Juergen Hipp, Harald Buchgeher, Candido Ferrio, Josep Lario, Yoshikazu Tachibana, Klas Hellmann, Jan Kosa, Burkhard Werner, Wolfgang Fien, Willi Gagsteiger, Hilmar Panzer, Edwin Schwellinger, Lutz Augenstein, Heiko Berner und Eelco van der Wal. *Function blocks for motion control*. Technische Spezifikation. Version 2.0. PLCopen – Technical Committee 2, 17. März 2011.

[106] Mikell P. Groover. *Automation, Production Systems, and Computer-Integrated Manufacturing*. 3. Aufl. Upper Saddle River, NJ: Prentice-Hall, 2008.

[107] Martin Hägele, Nikolaus Blümlein und Oliver Kleine. *Wirtschaftlichkeitsanalysen neuartiger Servicerobotik-Anwendungen und ihre Bedeutung für die Robotik-Entwicklung (EFFIROB)*. Studie. Fraunhofer-Institut für Produktionstechnik und Automatisierung (IPA), 2011.

[108] Martin Hägele, Klas Nilsson und J. Norberto Pires. „Industrial Robotics“. In: *Springer Handbook of Robotics*. Hrsg. von Bruno Siciliano und Oussama Khatib. Berlin: Springer, 2008. Kap. 42, S. 963–986.

[109] Martin Hägele, Thomas Skordas, Stefan Sagert, Rainer Bischoff, Torgny Brogårdh und Manfred Dresselhaus. *Industrial Robot Automation*. White Paper. European Robotics Network, Juli 2005.

[110] N. Halbwachs, P. Caspi, P. Raymond und D. Pilaud. „The synchronous dataflow programming language LUSTRE“. In: *Proceedings of the IEEE* 79.9 (Sep. 1991), S. 1305–1320.

[111] Tobias Hammer und Berthold Bäuml. „The Highly Performant and Realtime Deterministic Communication Layer of the aRDx Software Framework“. In: *16th Intl. Conference on Advanced Robotics. ICAR 2013*. Proceedings. (Montevideo, Uruguay, 25.–29. Nov. 2013). IEEE. 2013.

[112] Dominik Haneberg, Alwin Hoffmann, Hella Seebach, Michael Vistein und Wolfgang Reif. „Towards Model-based Evolution for Robot-based Automation“. In: *Modellierung in der Automatisierungstechnik. (MAT 2012), Workshop im Rahmen der Modellierung 2012*. Tagungsband. (Bamberg, 15. März 2012). 2012.

[113] David Harel. „Statecharts: A Visual Formalism for Complex Systems“. In: *Science of Computer Programming* 8 (1987), S. 231–274.

[114] P. E. Hart, N. J. Nilsson und B. Raphael. „A Formal Basis for the Heuristic Determination of Minimum Cost Paths“. In: *IEEE Transactions on Systems Science and Cybernetics* 2 (1968), S. 100–107.

[115] Vincent Hayward und Richard P. Paul. „Robot Manipulator Control under Unix RCCL: A Robot Control C Library“. In: *International Journal of Robotics Research* 5.4 (1986), S. 94–111.

[116] Stefan Hesse. *Grundlagen der Handhabungstechnik*. 2. Aufl. München: Carl Hanser Verlag, 2010.

[117] Gerd Hirzinger und Berthold Bäuml. „Agile Robot Development (aRD): A Pragmatic Approach to Robotic Software“. In: *2006 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems. IROS 2006*. Proceedings. (Beijing, China, 9.–15. Okt. 2006). IEEE. 2006.

[118] Alwin Hoffmann, Andreas Angerer, Frank Ortmeier, Michael Vistein und Wolfgang Reif. „Hiding Real-Time: A new Approach for the Software Development of Industrial Robots“. In: *2009 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems. IROS 2009*. Proceedings. (St. Louis, MO, USA, 11.–15. Okt. 2009). IEEE. 2009, S. 2108–2113.

[119] Alwin Hoffmann, Andreas Angerer, Andreas Schierl, Michael Vistein und Wolfgang Reif. „Managing Extensibility and Maintainability of Industrial Robotics Software“. In: *16th Intl. Conference on Advanced Robotics. ICAR 2013*. Proceedings. (Montevideo, Uruguay, 25.–29. Nov. 2013). IEEE. 2013, S. 1–7.

[120] Alwin Hoffmann, Andreas Angerer, Andreas Schierl, Michael Vistein und Wolfgang Reif. „Service-oriented Robotics Manufacturing by reasoning about the Scene Graph of a Robotics Cell“. In: *45th International Symposium on Robotics (ISR 2014) and the 8th German Conference on Robotics (Robotik 2014)*. Proceedings. (Munich, Germany, 2.–3. Juni 2014). Berlin: VDE Verlag, 2014, S. 756–763.

[121] Alwin Hoffmann, Andreas Angerer, Andreas Schierl, Michael Vistein und Wolfgang Reif. „Service-orientierte Modellierung einer Robotermontagezelle“. In: *Internationales Forum Mechatronik. IFM 2011*. Tagungsband. (Cham, 21.–22. Sep. 2011). Cluster Mechatronik & Automation e.V. 2011, S. 179–191.

[122] Alwin Hoffmann, Andreas Angerer, Andreas Schierl, Michael Vistein und Wolfgang Reif. „Software Engineering in der Industrierobotik: der SoftRobot-Ansatz“. In: *Entwurf komplexer Automatisierungssysteme. EKA 2010. Beschreibungmittel, Methoden, Werkzeuge und Anwendungen*. Tagungsband. (Magdeburg, 25.–27. Mai 2010). 2010.

[123] Alwin Hoffmann, Andreas Angerer, Andreas Schierl, Michael Vistein und Wolfgang Reif. „Towards Object-Oriented Software Development for Industrial Robots“. In: *7th Intl. Conference on Informatics in Control, Automation and Robotics. ICINCO 2010*. Proceedings. (Funchal, Madeira, Portugal, 15.–18. Juni 2010). Hrsg. von Joaquim Filipe, Juan Andrade-Cetto und Jean-Louis Ferrier. Bd. 2. INSTICC. 2010, S. 437–440.

[124] Alwin Hoffmann, Florian Nafz, Frank Ortmeier, Andreas Schierl und Wolfgang Reif. „Prototyping Plant Control Software with Microsoft Robotics Studio“. In: *Software Development and Integration in Robotics. SDIR-III. Workshop at the 2008 IEEE Intl. Conference on Robotics and Automation*. Proceedings. (Pasadena, USA, 20. Mai 2008). IEEE. 2008.

[125] Alwin Hoffmann, Florian Nafz, Hella Seebach, Andreas Schierl und Wolfgang Reif. „Developing Self-Organizing Robotic Cells using Organic Computing Principles“. In: *Bio-Inspired Self-Organizing and Self-Reconfigurable Robotic Systems. Workshop at the 2010 IEEE Intl. Conference on Robotics and Automation*. Proceedings. (Anchorage, AL, USA, 3. Mai 2010). IEEE. 2010.

[126] Alwin Hoffmann, Florian Nafz, Hella Seebach, Andreas Schierl und Wolfgang Reif. „Developing Self-Organizing Robotic Cells using Organic Computing Principles“. In: *Bio-Inspired Self-Organizing Robotic Systems*. Hrsg. von Yan Meng und Yaochu Jin. Bd. 355. Studies in Computational Intelligence. Berlin: Springer, 2011, S. 253–274.

[127] Holger Hoffmann, Jan Marco Leimeister und Helmut Krcmar. „Prototyping komplexer Geschäftsanwendungen im Automobil“. In: *Multikonferenz Wirtschaftsinformatik 2010. Automotive Services 2010*. Tagungsband. (Göttingen, Germany, 23.–25. Feb. 2010). Hrsg. von Matthias Schumann, Lutz M. Kolbe, Michael H. Breitner und Arne Frerichs. 2010, S. 911–922.

[128] K. Hörmann und U. Negretto. „Programming of the Cranfield assembly benchmark“. In: *Integration of Robots into CIM*. Hrsg. von Rolf Bernhardt, Rüdiger Dillman, Klaus Hörmann und K. Tierney. Springer, 1992, S. 263–283.

[129] Jacob Huckaby. „Knowledge Transfer in Robot Manipulation Tasks“. Diss. Georgia Institute of Technology, Mai 2014.

[130] Jacob Huckaby und Henrik I. Christensen. „A Case for SysML in Robotics“. In: *2014 IEEE Conference on Automation Science and Engineering. CASE 2014*. Proceedings. (Taipei, Taiwan, 18.–22. Aug. 2014). IEEE. 2014, S. 333–338.

[131] Jacob Huckaby und Henrik I. Christensen. „A Taxonomic Framework for Task Modeling and Knowledge Transfer in Manufacturing Robotics“. In: *Cognitive Robotics. Workshop at the 26th AAAI Conference on Artificial Intelligence*. Proccedings. (Toronto, Canada, 22.–23. Juli 2012). Hrsg. von Wolfram Burgard, Kurt Konolige, Maurice Pagnucco und Stavros Vassos. AAAI Press, 2012, S. 94–101.

[132] Jacob Huckaby und Henrik I. Christensen. „Modeling Robot Assembly Tasks in Manufacturing Using SysML“. In: *45th International Symposium on Robotics (ISR 2014) and the 8th German Conference on Robotics (Robotik 2014)*. Proceedings. (Munich, Germany, 2.–3. Juni 2014). VDE Verlag, 2014, S. 1–7.

[133] Jacob Huckaby, Stavros Vassos und Henrik I. Christensen. „Planning with a task modeling framework in manufacturing robotics“. In: *2013 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems. IROS 2013*. Proceedings. (Tokyo, Japan, 3.–8. Nov. 2013). IEEE, 2013, S. 5787–5794.

[134] Institute for Software & Systems Engineering. *SafeAssistance – Intelligent obstacle detection using capacitive sensors for safe human robot interaction*. URL: http://www.isse.uni-augsburg.de/safeassistance/ (besucht am 14. 02. 2015).

[135] International Electrotechnical Commission. „Adjustable speed electrical power drive systems – Part 7-201: Generic interface and use of profiles for power drive systems – Profile type 1 specification“. International Standard IEC 61800-7-201 Edition 1.0. 27. Nov. 2007.

[136] International Electrotechnical Commission. „Function blocks – Part 1: Architecture“. International Standard IEC 61499-1 Edition 2.0. Nov. 2012.

[137] International Electrotechnical Commission. „Industrial communication networks – Fieldbus specifications – Part 1: Overview and guidance for the IEC 61158 and IEC 61784 series“. International Standard IEC 61158-1 Edition 1.0. 23. Mai 2014.

[138] International Electrotechnical Commission. „Programmable controllers – Part 3: Programming languages“. International Standard IEC 61131-3 Edition 3.0. 20. Feb. 2013.

[139] International Organization for Standardization. „Information technology – Object Management Group Business Process Model and Notation“. International Standard ISO/IEC 19510:2013. 15. Juli 2013.

[140] International Organization for Standardization. „Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling“. Norm ISO 11898-1:2003. 2003.

[141] F. Jammes und H. Smit. „Service-oriented paradigms in industrial automation“. In: *IEEE Transactions on Industrial Informatics* 1.1 (2005), S. 62–70.

[142] Java Community Process. *JSR 291: Dynamic Component Support for Java SE*. Java Specification Request. 7. Aug. 2007. URL: https://www.jcp.org/en/jsr/detail?id=291 (besucht am 22. 10. 2014).

[143] J. Jurkevich, J. E. Kukareko und A. Pashkevich. „An object-oriented approach to workcell modelling“. In: *4th Intl. Conference on Intelligent Autonomous Systems. IAS-4*. Proceedings. (Karlsruhe, Germany, 27.–30. März 1995). Hrsg. von U. Rembold, R. Dillmann, L.O. Hertzberger und T. Kanade. IOS Press, 1995, S. 436–440.

[144] Henning Kagermann, Wolfgang Wahlster und Johannes Helbig, Hrsg. *Umsetzungsempfehlungen für das Zukunftsprojekt Industrie 4.0*. Abschlussbericht des Arbeitskreises Industrie 4.0. Apr. 2013. URL: http://www.bmbf.de/pubRD/Umsetzungsempfehlungen_Industrie4_0.pdf (besucht am 22. 12. 2014).

[145] Lydia E. Kavraki und Steven M. LaValle. „Motion Planning“. In: *Springer Handbook of Robotics*. Hrsg. von Bruno Siciliano und Oussama Khatib. Berlin: Springer, 2008. Kap. 5, S. 109–131.

[146] Markus Klotzbucher und Herman Bruyninckx. „Coordinating Robotic Tasks and Systems with rFSM Statecharts“. In: *Journal of Software Engineering for Robotics* 3, no 1 (2012), S. 28–56.

[147] Markus Klotzbücher, Peter Soetens und Herman Bruyninckx. „OROCOS RTT-Lua: an Execution Environment for building Real-time Robotic Domain Specific Languages“. In: *Dynamic languages for RObotic and Sensors systems. DYROS. Workshop at the 2010 Simulation, Modeling, and Programming for Autonomous Robots*. Proceedings. (Darmstadt, Germany, 15. Nov. 2010). 2010, S. 284–289.

[148] Knopflerfish Project. *Knopflerfish OSGi – Open Source OSGi Service Platform*. URL: http://www.knopflerfish.org/ (besucht am 22. 10. 2014).

[149] Andreas Korff. *Modellierung von eingebetteten SySystem mit UML und SysML*. Heidelberg: Spektrum Akademischer Verlag, 2008.

[150] Dirk Krafzig, Karl Blanke und Dirk Slama. *Enterprise SOA: Service-Oriented Architecture Best Practices*. Upper Saddle River, NJ: Prentice Hall PTR, 2005.

[151] Peter Kriens und B.J. Hargrave. *Listeners Considered Harmful: The Whiteboard Pattern*. Whitepaper. 2004. URL: http://www.osgi.org/wiki/uploads/Links/whiteboard.pdf (besucht am 28. 10. 2014).

[152] Torsten Kröger und Friedrich M. Wahl. „Low-Level Control of Robot Manipulators: Sensor-Guided Control and On-Line Trajectory Generation“. In: *Innovative Robot Control Architectures for Demanding (Research) Applications – How to Modify and Enhance Commercial Controllers. Workshop at the 2010 IEEE Intl. Conference on Robotics and Automation*. Proceedings. (Anchorage, AL, USA, 3. Mai 2010). Hrsg. von Daniel Kubus, Klas Nilsson und Rolf Johansson. IEEE. Anchorage, AK, USA, 2010, S. 46–53.

[153] James Kuffner und Steven M. LaValle. „RRT-Connect: An Efficient Approach to Single-Query Path Planning“. In: *2000 IEEE Intl. Conference on Robotics and Automation. ICRA 2000*. Proceedings. (San Francisco, CA, USA, 24.–28. Apr. 2000). IEEE, 2000, S. 995–1001.

[154] Benjamin J. Kuipers. „The spatial semantic hierarchy“. In: *Artificial Intelligence* 119.1 (2000), S. 191–233.

[155] Benjamin J. Kuipers und Yung-Tai Byun. „A robust qualitative method for spatial learning in unknown environments.“ In: *7th AAAI National Conference on Artificial Intelligence. AAAI-88*. Proceedings. (Saint Paul, USA, 21.–26. Aug. 1988). 1988.

[156] KUKA Roboter GmbH. *KUKA System Software 8.2. Bedien- und Programmieranleitung für Systemintegratoren*. Version KSS 8.2 SI V4 de. 19. Juni 2012.

[157] KUKA Roboter GmbH. *KUKA.PLC mxAutomation S7 1.0. Für KUKA System Software 8.2*. Version KST PLC mxAutomation S7 1.0 V1 de. 29. März 2013.

[158] KUKA Roboter GmbH. *KUKA.PLC ProConOS 4-1 4.0. Für KUKA System Software 8.2*. Version KST PLC ProConOS 4.0 V2 de. 22. Juni 2011.

[159] KUKA Roboter GmbH. *KUKA.RoboTeam 2.0. Für KUKA System Software 8.3*. Version KST RoboTeam 2.0 V2 de. 9. Apr. 2013.

[160] KUKA Roboter GmbH. *KUKA.RobotSensorInterface 3.1. Für KUKA System Software 8.2*. Version KST RSI 3.1 V1 de. 2. Dez. 2010.

[161] KUKA Roboter GmbH. *KUKA.Sim Pro*. URL: http://www.kuka-robotics.com/germany/de/products/software/kuka_sim/kuka_sim_detail/PS_KUKA_Sim_Pro.htm (besucht am 20. 10. 2014).

[162] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. 3. Aufl. Upper Saddle River, NJ: Prentice Hall, 2004.

[163] Steven M. LaValle. *Rapidly-exploring random trees: A new tool for path planning*. Techn. Ber. 98-11. Iowa State University, Okt. 1998. URL: http://msl.cs.uiuc.edu/~lavalle/papers/Lav98c.pdf (besucht am 26. 01. 2015).

[164] Paulo Leitão. „Agent-based distributed manufacturing control: A state-of-the-art survey". In: *Engineering Applications of Artificial Intelligence* 22.7 (2009), S. 979–991.

[165] Lothar Litz. *Grundlagen der Automatisierungstechnik. Regelungssysteme – Steuerungssysteme – Hybride Systeme*. 2. Aufl. München: Oldenbourg Verlag, 2013.

[166] Markus S. Loffler, Vilas Chitrakaran und Darren M. Dawson. „Design and Implementation of the Robotic Platform". In: *Journal of Intelligent and Robotic System* 39 (2004), S. 105–129.

[167] Tomás Lozano-Pérez. „Robot Programming". In: *Proceedings of the IEEE* 71.7 (Juli 1983), S. 821–841.

[168] Shengyuan Luo, Ping Jiang und Jin Zhu. „Software Evolution of Robot Control Based on OSGi". In: *2004 IEEE Intl. Conference on Robotics and Biomimetics. ROBIO 2004*. Proceedings. (Shenyang, China, 22.–26. Aug. 2004). IEEE. 2004, S. 221–226.

[169] Pattie Maes. „Situated agents can have goals". In: *Robotics and Autonomous Systems* 6.1 (1990), S. 49–70.

[170] Jeff Magee und Jeff Kramer. „Dynamic Structure in Software Architectures". In: *ACM SIGSOFT Software Engineering Notes* 21.6 (Nov. 1996), S. 3–14.

[171] Alexei Makarenko, Alex Brooks und Tobias Kaupp. „Orca: Components for Robotics". In: *Robotic Standardization. Workshop at the 2006 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems*. Proceedings. (Beijing, China, 10. Okt. 2006). 2006.

[172] Jacek Malec, Anders Nilsson, Klas Nilsson und Slawomir Nowaczyk. „Knowledge-Based Reconfiguration of Automation Systems". In: *3rd Annual IEEE Conference on Automation Science and Engineering. CASE 2007*. Proceedings. (Scottsdale, AZ, USA, 22.–25. Sep. 2007). IEEE, 2007, S. 170–175.

[173] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.

[174] Matthew Mason. „Compliance and Force Control for Computer-Controlled Manipulators". In: *IEEE Transactions on Systems, Man, and Cybernetics* 11.6 (1981), S. 418–432.

[175] Jeff McAffer, Jean-Michel Lemieux und Chris Aniszczyk. *Eclipse Rich Client Platform*. 2. Aufl. the eclipse series. Upper Saddle River, NJ: Addison Wesley, Mai 2010.

[176] Jeff McAffer, Paul VanderLei und Simon Archer. *OSGi and Equinox – Creating Highly Modular Java Systems*. the eclipse series. Upper Saddle River, NJ: Addison Wesley, Feb. 2010.

[177] Gerard T. McKee, J. Andrew Fryer und Paul S. Schenker. „Object-Oriented Concepts for Modular Robotics Systems". In: *39th Intl. Conference and Exhibition on Technology of Object-Oriented Languages and Systems. TOOLS 39*. Proceedings. (Santa Barbara, CA, USA, 29. Juli–3. Aug. 2001). Hrsg. von Qiaoyun Li, Richard Riehle, Gilda Pour und Bertrand Meyer. IEEE Computer Society. 2001, S. 229–238.

[178] Claudio Melchiorri und Makoto Kaneko. „Robot Hands“. In: *Springer Handbook of Robotics*. Hrsg. von Bruno Siciliano und Oussama Khatib. Berlin: Springer, 2008. Kap. 3, S. 67–86.

[179] Giorgio Metta, Paul Fitzpatrick und Lorenzo Natale. „YARP: Yet another robot platform“. In: *Intl. Journal on Advanced Robotics Systems* 3.1 (2006), S. 43–48.

[180] Bertrand Meyer. „Applying 'Design by Contract'“. In: *Computer* 25.10 (Okt. 1992), S. 40–51.

[181] Joachim Michniewicz und Gunther Reinhart. „Cyber-physical Robotics – Automated Analysis, Programming and Configuration of Robot Cells based on Cyber-physical-systems“. In: *Procedia Technology* 15 (2014).

[182] David J. Miller und R. Charleene Lennox. „An Object-Oriented Environment for Robot System Architectures“. In: *IEEE Control System Magazine* 11.2 (1991), S. 14–23.

[183] Henrik Mühe, Andreas Angerer, Alwin Hoffmann und Wolfgang Reif. „On reverse-engineering the KUKA Robot Language“. In: *Domain-Specific Languages and models for ROBotic systems. DSLRob '10. Workshop at the 2010 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems*. Proceedings. (Taipei, Taiwan, 22. Okt. 2010). 2010. URL: http://arxiv.org/abs/1009.5004.

[184] Christian Müller-Schloer, Christoph von der Malsburg und Rolf P. Würtz. „Organic Computing“. In: *Informatik-Spektrum* 27.4 (2004), S. 332–336.

[185] David J. Musliner, Edmund H. Durfee und Kang G. Shin. „World modeling for the dynamic construction of real-time control plans“. In: *Artificial Intelligence* 74.1 (1995), S. 83–127.

[186] David J. Musliner, Michael J. S. Pelican, Robert P. Goldman, Kurt D. Krebsbach und Edmund H. Durfee. „The Evolution of CIRCA, a Theory-Based AI Architecture with Real-Time Performance Guarantees“. In: *Architectures for Intelligent Theory-Based Agents. Papers from the 2008 AAAI Spring Symposium*. Proccedings. Hrsg. von Marcello Balduccini und Chitta Baral. AAAI Press, 2008.

[187] Ludwig Nägele, Miroslav Macho, Andreas Angerer, Alwin Hoffmann, Michael Vistein, Manfred Schönheits und Wolfgang Reif. „A backward-oriented approach for offline programming of complex manufacturing tasks“. In: *6th Intl. Conference on Automation, Robotics and Applications. ICARA 2015*. Proceedings. (Queenstown, New Zealand, 17.–19. Feb. 2015). IEEE. 2015.

[188] Martin Naumann, Kai Wegener und Rolf Dieter Schraft. „Control Architecture for Robot Cells to Enable Plug'n'Produce“. In: *2007 IEEE Intl. Conference on Robotics and Automation. ICRA 2007*. Proceedings. (Roma, Italy, 10.–14. Apr. 2007). Rome, Italy: IEEE, 2007, S. 287–292.

[189] Martin Naumann, Kai Wegener, Rolf Dieter Schraft und Luca Lachello. „Robot Cell Integration by means of Application-P'n'P“. In: *37st International Symposium on Robotics (ISR 2006) and the 4th German Conference on Robotics (Robotik 2006)*. Proceedings. (Munich, Germany, 15.–17. Mai 2006). VDI-Berichte 1956. VDI Verlag, 2006.

[190] Issa A.D. Nesnas. „The CLARAty Project: Coping with Hardware and Software Heterogeneity“. In: *Software Engineering for Experimental Robotics*. Hrsg. von Davide Brugali. Springer Tracts in Advanced Robotics 30. Berlin: Springer, 2007, S. 31–70.

[191] Issa A.D. Nesnas, Reid Simmons, Daniel Gaines, Clayton Kunz, Antonio Diaz-Calderon, Tara Estlin, Richard Madison, John Guineau, Michael McHenry, I-Hsiang Shu und David Apfelbaum. „CLARAty: Challenges and Steps Toward Reusable Robotic Software“. In: *Intl. Journal of Advanced Robotic Systems* 3.1 (2006), S. 23–30.

[192] Henrik Frystyk Nielsen und George Chrysanthakopoulos. *Decentralized Software Services Protocol – DSSP/1.0*. Microsoft Corporation. Juli 2007. URL: http://purl.org/msrs/dssp.pdf (besucht am 12. 01. 2015).

[193] Klas Nilsson und Matthias Bengel. „Plug-and-Produce Technologies Real-time Aspects. Service Oriented Architectures for SME Robots and Plug-and-Produce“. In: *5th Intl. Conference on Informatics in Control, Automation and Robotics. ICINCO 2008*. Proceedings. (Funchal, Madeira, Portugal, 11.–15. Mai 2008). Hrsg. von Joaquim Filipe, Juan Andrade-Cetto und Jean-Louis Ferrier. Bd. 2. INSTICC. 2008, S. 249–254.

[194] OASIS. *Service Component Architecture (SCA)*. URL: http://www.oasis-opencsa.org/sca (besucht am 05. 11. 2014).

[195] Object Management Group. *Business Process Model and Notation (BPMN)*. Spezifikation. Version 2.0. Jan. 2011. URL: http://www.omg.org/spec/BPMN/2.0/ (besucht am 28. 12. 2014).

[196] Object Management Group. *OMG Systems Modeling Language (OMG SysML™)*. Spezifikation. Version 1.3. Juni 2012. URL: http://www.omg.org/spec/SysML/1.3/ (besucht am 04. 12. 2014).

[197] Object Management Group. *OMG Unified Modeling Language™ (OMG UML), Superstructure*. Spezifikation. Version 2.4.1. Aug. 2011. URL: http://www.omg.org/spec/UML/2.4.1/ (besucht am 04. 12. 2014).

[198] Object Management Group. *Robotic Technology Component Specification*. Spezifikation. Version 1.0. Apr. 2008. URL: http://www.omg.org/spec/RTC/1.0/ (besucht am 04. 12. 2014).

[199] Kenichi Ohara, Kyohei Iwanean Tomohito Takubo, Yasushi Mae und Tatsuo Arai. „Component-based robot software design for pick-and-place task described by SysML“. In: *8th Intl. Conference on Ubiquitous Robots and Ambient Intelligence. (URAI 2011)*. Proceedings. (Incheon, Korea, 23.–26. Nov. 2011). IEEE. 2011, S. 124–128.

[200] *PLCopen for efficiency in automation*. URL: http://www.plcopen.org/ (besucht am 28. 10. 2014).

[201] Orocos Project. *Orocos Real-Time Toolkit*. URL: http://www.orocos.org/rtt/ (besucht am 28. 10. 2014).

[202] Orocos Project. *Orocos Kinematics and Dynamics*. URL: http://www.orocos.org/kdl/ (besucht am 28. 10. 2014).

[203] Orocos Project. *Open Robot Control Software*. URL: http://www.orocos.org/ (besucht am 28. 10. 2014).

[204] Frank Ortmeier, Alwin Hoffmann, Wolfgang Reif, Ulrich Huggenberger und Thomas Stumpfegger. „Simulations-basierte Programmierung von Industrierobotern“. In: *Internationales Forum Mechatronik. IFM 2008*. Tagungsband. (Stuttgart, 22.–25. Sep. 2008). 2008.

[205] OSGi Alliance. *OSGi Compendium*. Spezifikation. Version 5.0. Mai 2013. url: http://www.osgi.org/download/r5/osgi.cmpn-5.0.0.pdf (besucht am 22. 10. 2014).

[206] OSGi Alliance. *OSGi Core Release 5 Specification*. Spezifikation. Version 5.0. März 2012. url: http://www.osgi.org/download/r5/osgi.core-5.0.0.pdf (besucht am 22. 10. 2014).

[207] OSGi Alliance. *OSGi Market and Solutions*. url: http://www.osgi.org/Markets/HomePage (besucht am 30. 10. 2014).

[208] OSGi Alliance. *OSGi – The Dynamic Module System for Java*. url: http://www.osgi.org/Main/HomePage (besucht am 30. 10. 2014).

[209] C. Ott, O. Eiberger, W. Friedl, B. Bauml, U. Hillenbrand, C. Borst, A. Albu-Schaffer, B. Brunner, H. Hirschmuller, S. Kielhofer, R. Konietschke, M. Suppa, T. Wimbock, F. Zacharias und G. Hirzinger. „A Humanoid Two-Arm System for Dexterous Manipulation“. In: *6th IEEE-RAS Intl. Conference on Humanoid Robots*. Proceedings. (Genoa, Italy, 4.–6. Dez. 2006). 2006, S. 276–283.

[210] Zengxi Pan, Joseph Polden, Nathan Larkin, Stephen Van Duin und John Norrish. „Recent progress on programming methods for industrial robots“. In: *Robotics and Computer-Integrated Manufacturing* 28.2 (2012), S. 87–94.

[211] Hilmar Panzer, Christian Müller, Klaus Bernzen, Josef Papenfort, Wilfried Plass, Wolfgang Czech, Friedrich Forthuber, Martin Schrott, Ed Baker, Roland Schaumburg, Ryszard Bochniak, Djafar Hadiouche, Jürgen Hipp, Joachim Mayer, Harald Buchgeher, Joachim Strobel, Candido Ferrio, Josep Lario, Yoshikazu Tachibana, Christian Ruf, Klas Hellmann, Markus Müller, Willi Gagsteiger, Hans Peter Otto, Jürgen Fiess, Wolfgang Fien, Istvan Ulvros und Eelco van der Wal. *Function blocks for motion control: Part 4 – Coordinated Motion*. Technisches Papier. Version 1.0. PLCopen – Technical Committee 2, 3. Dez. 2008.

[212] David L. Parnas. „On the Criteria to Be Used in Decomposing Systems into Modules“. In: *Communications of the ACM* 15.12 (Dez. 1972), S. 1053–1058.

[213] Richard Paul. „WAVE: A model based language for manipulator control“. In: *Industrial Robot: An International Journal* 4.1 (1977), S. 10–17.

[214] Christoph Pelich und Friedrich M. Wahl. „ZERO++: An OOP Environment for Multiprocessor Robot Control“. In: *International Journal of Robotics and Automation* 12.2 (1997), S. 49–57.

[215] Jörn Peschke und Arndt Lüder. „Java Technology and Industrial Applications“. In: *The Industrial Information Technology Handbook*. Hrsg. von Richard Zurawski. Industrial Electronics Series. Boca Raton, FL: CRC Press, 2005. Kap. 63.

[216] J. Pfrommer, M. Schleipen und J. Beyerer. „PPRS: Production skills and their relation to product, process, and resource“. In: *18th IEEE Intl. Conference on Emerging Technologies and Factory Automation. ETFA 2013*. Proceedings. (Cagliari, Italy, 10.–13. Sep. 2013). IEEE. 2013, S. 1–4.

[217] J. Noberto Pires. *Industrial Robots Programming. Building Applications for the Factories of the Future*. New York, USA: Springer, 2007.

[218] J. Norberto Pires. „New challenges for industrial robotic cell programming“. In: *Industrial Robot* 36.1 (2009).

[219] J. Norberto Pires, Germano Veiga und Ricardo Araújo. „Programming by demonstration in the coworker scenario for SMEs“. In: *Industrial Robot* 36.1 (2009), S. 73–83.

[220] Erwin Prassler, Herman Bruyninckx, Klas Nilsson und Azamat Shakhimardanov. *The Use of Reuse for Designing and Manufacturing Robots*. White Paper. Robot Standards und Reference Architectures consortium (RoSta), Juni 2009. url: http://www.robot-standards.eu/Documents_RoSta_wiki/whitepaper_reuse.pdf.

[221] Sebastian Preusse, Christian Gerber und Hans-Michael Hanisch. „Design Approaches for IEC 61499 Control Applications“. In: *INFORMATIK 2010. Service Science – Neue Perspektiven für die Informatik*. Proceedings. (Leipzig, Germany, 27. Sep.–1. Okt. 2010). Hrsg. von Klaus-Peter Fähnrich und Bogdan Franczyk. Bd. 176. Lecture Notes in Informatics. Gesellschaft für Informatik. 2010, S. 469–479.

[222] ProSyst Software GmbH. *OSGi Technology Provider*. url: http://www.prosyst.com/startseite/ (besucht am 13. 11. 2014).

[223] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler und Andrew Y. Ng. „ROS: an open-source Robot Operating System“. In: *Open Source Software in Robotics. Workshop at the 2009 IEEE Intl. Conference on Robotics and Automation*. Proceedings. (Kobe, Japan, 17. Mai 2009). IEEE. 2009.

[224] Matthias Radestock und Susan Eisenbach. „Coordination in evolving systems“. In: *Trends in Distributed Systems CORBA and Beyond. International Workshop TreDS '96*. Proceedings. (Aachen, Germany, 1.–2. Okt. 1996). Hrsg. von Otto Spaniol, Claudia Linnhoff-Popien und Bernd Meyer. Bd. 1161. Lecture Notes in Computer Science. Berlin: Springer, 1996, S. 162–176.

[225] Mohd Azizi Abdul Rahman, Katsuhiro Mayama, Takahiro Takasu, Akira Yasuda und Makoto Mizukawa. „Model-Driven Development of Intelligent Mobile Robot Using Systems Modeling Language (SysML)“. In: *Mobile Robots – Control Architectures, Bio-Interfacing, Navigation, Multi Robot Motion Planning and Operator Training*. Hrsg. von Janusz Bedkowski. InTech, Dez. 2011. Kap. 1.

[226] Red Hat, Inc. *jBPM – Open Source Business Process Management - Process Engine*. url: http://www.jbpm.org/ (besucht am 28. 12. 2014).

[227] Gunther Reinhart, Gerhard Strasser und Johannes Scharrer. „Automatisierte Fertigung von Faserverbundbauteilen“. In: *Werkstattstechnik online* 9 (2008), S. 711–716.

[228] Wolfgang Reisig. *A Primer in Petri Net Design*. Springer Compass International. Berlin: Springer, 1992.

[229] Jan S. Rellermeyer, Michael Duller, Ken Gilmer, Damianos Maragkos, Dimitrios Papageorgiou und Gustavo Alonso. „The Software Fabric for the Internet of Things“. In: *The Internet of Things. First International Conference, IOT 2008*. Proceedings. (Zurich, Switzerland, 26.–28. März 2008). Hrsg. von Christian Floerkemeier, Marc Langheinrich, Elgar Fleisch, Friedemann Mattern und Sanjay E. Sarma. Bd. 4952. Lecture Notes in Computer Science. 2008, S. 87–104.

[230] Robot Operation System. *Industrial*. Open Source Robotics Foundation. URL: http://wiki.ros.org/industrial (besucht am 22. 12. 2014).

[231] Robot Operation System. *SMACH*. Open Source Robotics Foundation. URL: http://wiki.ros.org/smach (besucht am 22. 11. 2014).

[232] Robot Operation System. *Unified Robot Description Format*. Open Source Robotics Foundation. URL: http://wiki.ros.org/urdf (besucht am 22. 11. 2014).

[233] Christian Schäfer und Omar Lopez. „An Object-Oriented Robot Model and Its Integration into Flexible Manufacturing Systems“. In: *Multiple Approaches to Intelligent Systems. 12th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, IEA/AIE-99*. Proceedings. (Cairo, Egypt, 31. Mai–3. Juni 1999). Hrsg. von Ibrahim Imam, Yves Kodratoff, Ayman El-Dessouki und Moonis Ali. Bd. 1611. Lecture Notes of Computer Science. Springer, 1999, S. 820–829.

[234] Victor Scheinmann und J. Michael McCarthy. „Mechanisms and Actuation“. In: *Springer Handbook of Robotics*. Hrsg. von Bruno Siciliano und Oussama Khatib. Berlin: Springer, 2008. Kap. 3, S. 67–86.

[235] Andreas Schierl, Andreas Angerer, Alwin Hoffmann, Michael Vistein und Wolfgang Reif. „From Robot Commands To Real-Time Robot Control – Transforming High-Level Robot Commands into Real-Time Dataflow Graphs“. In: *9th Intl. Conference on Informatics in Control, Automation and Robotics. ICINCO 2012*. Proceedings. (Rome, Italy, 28.–31. Juli 2012). Hrsg. von Jean-Louis Ferrier, Alain Bernard, Oleg Yu. Gusikhin und Kurosh Madani. INSTICC. 2012, S. 1–6.

[236] Andreas Schierl, Andreas Angerer, Alwin Hoffmann, Michael Vistein und Wolfgang Reif. „Using Java for Real-Time Critical Industrial Robot Programming“. In: *Software Development and Integration in Robotics. SDIR-VII. Workshop at the 2012 IEEE Intl. Conference on Robotics and Automation*. Proceedings. (St. Paul, MN, USA, 14. Mai 2012). IEEE. 2012.

[237] Andreas Schierl, Alwin Hoffmann, Andreas Angerer, Michael Vistein und Wolfgang Reif. „Towards Realtime Robot Reactions: Patterns for Modular Device Driver Interfaces“. In: *Software Development and Integration in Robotics. SDIR-VIII. Workshop at the 2013 IEEE Intl. Conference on Robotics and Automation*. Proceedings. (Karlsruhe, Germany, 6. Mai 2013). IEEE. 2013.

[238] Douglas C Schmidt und Fred Kuhns. „An overview of the real-time CORBA specification“. In: *Computer* 33.6 (2000), S. 56–63.

[239] Gerhard Schnell und Bernhard Wiedemann, Hrsg. *Bussysteme in der Automatisierungs- und Prozesstechnik. Grundlagen, Systeme und Anwendungen der industriellen Kommunikation*. 8. Aufl. Wiesbaden: Vieweg + Teubner, Okt. 2012.

[240] Günter Schreiber, Andreas Stemmer und Rainer Bischoff. „The Fast Research Interface for the KUKA Lightweight Robot“. In: *Innovative Robot Control Architectures for Demanding (Research) Applications – How to Modify and Enhance Commercial Controllers. Workshop at the 2010 IEEE Intl. Conference on Robotics and Automation*. Proceedings. (Anchorage, AL, USA, 3. Mai 2010). Hrsg. von Daniel Kubus, Klas Nilsson und Rolf Johansson. IEEE. 2010, S. 15–21.

[241] Manfred Schulte-Zurhausen. *Organisation*. 6. Aufl. Vahlens Handbücher der Wirtschafts- und Sozialwissenschaften. München: Verlag Franz Vahlen, 2014.

[242] SCHUNK GmbH & Co. KG. *Electrical 2-Finger Parallel Gripper WSG 50. Assembly and Operating Manual*. 02.03. 30. Juli 2014. URL: http://www.schunk.com/schunk_files/attachments/OM_AU_WSG50__EN.pdf (besucht am 26. 01. 2015).

[243] SCHUNK GmbH & Co. KG. *Elektrischer Parallelgreifer MEG 50 EC. Montage- und Betriebsanleitung*. 01.02. 23. Jan. 2014. URL: http://www.schunk.com/schunk_files/attachments/OM_AU_MEG50-EC__DE.pdf (besucht am 26. 01. 2015).

[244] Bruce E. Shimano, Clifford C. Geschke und Charles H. Spalding. „VAL-II: A new robot control system for automatic manufacturing“. In: *1984 IEEE Intl. Conference on Robotics and Automation. ICRA 1984*. Proceedings. (Atlanta, GA, USA, 13.–15. März 1984). IEEE. 1984, S. 278–292.

[245] Siemens Industry Software GmbH & Co. KG. *Robcad*. URL: http://www.plm.automation.siemens.com/de_de/products/tecnomatix/robotics_automation/robcad/ (besucht am 30. 10. 2014).

[246] Reid Simmons und David Apfelbaum. „A Task Description Language for Robot Control“. In: *1998 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems. IROS '98*. Proceedings. (Victoria, Canada, 13.–17. Okt. 1998). Bd. 3. IEEE. 1998, S. 1931–1937.

[247] Reid Simmons, Trey Smith, M. Bernardine Dias, Dani Goldberg, David Hershberger, Anthony Stentz und Robert Zlot. „A Layered Architecture for Coordination of Mobile Robots“. In: *Multi-Robot Systems: From Swarms to Intelligent Automata. Proceedings from the 2002 NRL Workshop on Multi-Robot Systems*. (März 2002). Hrsg. von Alan C. Schultz und Lynne E. Parker. Kluwer Academic Publishers, 2002.

[248] Herbert A. Simon. *Die Wissenschaft vom Künstlichen*. 2. Aufl. Computerkultur. Wien: Springer-Verlag, 1994.

[249] Ruben Smits. „Robot Skills: Design of a Constraint-Based Methodology and Software Support“. Diss. KU Leuven, 17. Mai 2010. URL: https://lirias.kuleuven.be/handle/123456789/265542 (besucht am 22. 10. 2014).

[250] Ruben Smits, Tinne De Laet, Kasper Claes, Herman Bruyninckx und Joris De Schutter. „iTASC: a Tool for Multi-Sensor Integration in Robot Manipulation“. In: *2008 IEEE Intl. Conference on Multisensor Fusion and Integration for Intelligent Systems. MFI 2008*. Proceedings. (Seoul, Korea, 20.–22. Aug. 2008). IEEE. 2008, S. 426–433.

[251] Sebastian Smolorz und Bernardo Wagner. „Real-Time Capability of Robotic Systems Based on ROS“. In: *ROS-Industrial in European Research Projects. Workshop at the 13th Intl. Conference on Intelligent Autonomous Systems*. Proceedings. (Padova, Italy, 15. Juli 2014). 2014.

[252] Maj Stenmark und Jacek Malec. „Knowledge-Based Industrial Robotics“. In: *Twelfth Scandinavian Conference on Artificial Intelligence. SCAI 2013*. Proceedings. (Aalborg, Denmark, 20.–22. Nov. 2013). Hrsg. von Manfred Jaeger, Thomas Dyhre Nielsen und Paolo Viappiani. Bd. 257. Frontiers in Artificial Intelligence and Applications. IOS Press, 2013, S. 265–274.

[253] Maj Stenmark, Jacek Malec und Andreas Stolt. „From High-Level Task Descriptions to Executable Robot Code“. In: *Intelligent Systems 2014. 7th IEEE International Conference, IS 2014*. Proceedings. (Warsaw, Poland, 24.–26. Sep. 2014). Hrsg. von D. Filev, J. Jablkowski, J. Kacprzyk, M. Krawczak, I. Popchev, L. Rutkowski, V. Sgurev, E. Sotirova, P. Szynkarczyk und S. Zadrozny. Bd. 323. Advances in Intelligent Systems and Computing. Springer, 2015, S. 189–.202.

[254] Sven Stumm, Alwin Hoffmann, Hella Seebach, Bernd Kuhlenkötter und Wolfgang Reif. „Towards combining layout and process models for mixed assembly facilities“. In: *AUTOMATION 2014*. Tagungsband. (Baden-Baden, 1.–2. Juli 2014). Bd. 2231. VDI-Berichte. VDI Verlag, Aug. 2014.

[255] Clemens Szyperski. „Component Software and the Way Ahead“. In: *Foundations of Component-based Systems*. Hrsg. von Gary T. Leavens und Murali Sitaraman. New York, NY: Cambridge University Press, 2000, S. 1–20.

[256] Clemens Szyperski, Dominik Gruntz und Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. 2. Aufl. London: Addison-Wesley, 2002.

[257] Russell H. Taylor, Phillip D. Summers und J. M. Meyer. „AML: a manufacturing language“. In: *Intl. Journal of Robotics Research* 1.3 (1982), S. 19–41.

[258] Ulrike Thomas, Friedrich M. Wahl, Jochen Maass und Jürgen Hesselbach. „Towards a New Concept of Robot Programming in High Speed Assembly Applications“. In: *2005 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems. IROS 2005*. Proceedings. (Edmonton, Canada, 2.–6. Aug. 2005). IEEE. Edmonton, Canada, 2005, S. 3932–3938.

[259] Kleanthis Thramboulidis. „Different Perspectives – IEC 61499 function block model: Facts and fallacies“. In: *IEEE Industrial Electronics Magazine* 3.4 (2009), S. 7–26.

[260] Kleanthis Thramboulidis. „IEC 61499 in Factory Automation“. In: *Advances in Computer, Information, and Systems Sciences, and Engineering. Proceedings of IETA 2005, TeNe 2005, EIAE 2005*. Hrsg. von Khaled Elleithy, Tarek Sobh, Ausif Mahmood, Magued Iskander und Mohammad Karim. Springer, 2006, S. 115–124.

[261] H. Utz, S. Sablatnog, S. Enderle und G. Kraetzschmar. „Miro - Middleware for mobile robot applications“. In: *IEEE Transactions on Robotics and Automation* 18.4 (Aug. 2002), S. 493–497.

[262] A. Valera, J. Gomez-Moreno, A. Sánchez, C. Ricolfe-Viala, R. Zotovic und M. Vallés. „Industrial Robot Programming and UPnP Services Orchestration for the Automation of Factories“. In: *International Journal of Advanced Robotic Systems* 9 (2012).

[263] Hendrik Van Brussel, Jo Wyns, Paul Valckenaers, Luc Bongaerts und Patrick Peeters. „Reference architecture for holonic manufacturing systems: PROSA“. In: *Computers in Industry* 37.3 (1998), S. 255–274.

[264] Dominick Vanthienen, Markus Klotzbuecher und Herman Bruyninckx. „The 5C-based architectural Composition Pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming“. In: *Journal of Software Engineering for Robotics* 5.1 (2014), S. 17–35.

[265] Germano Veiga. „On the orchestration of operations in flexible manufacturing“. Diss. Universidade de Coimbra, 17. Feb. 2010.

[266] Germano Veiga, Pedro Malaca, J. Norberto Pires und Klas Nilsson. „Separation of concerns on the orchestration of operations in flexible manufacturing“. In: *Assembly Automation* 32.1 (2012), S. 38–50.

[267] Germano Veiga, J. Norberto und Klas Nilsson. „Experiments with service-oriented architectures for industrial robotic cells programming“. In: *Robotics and Computer-Integrated Manufacturing* 25.4-5 (2009), S. 746–755.

[268] Germano Veiga, J. Norberto und Klas Nilsson. „On the Use of Service Oriented Software Platforms for Industrial Robotic Cells“. In: *8th IFAC Intl. Workshop on Intelligent Manufacturing Systems. IMS '07*. Proceedings. (Alicante, Spain, 23.–25. Mai 2007). Hrsg. von Francisco A. Candelas, Carlos Pereira und Fernando Torres. International Federation of Automatic Control. Alicante, Spain, 2007.

[269] Verein Deutscher Ingenieure. „Montage- und Hadhabungstechnik. Handhabungsfunktionen, Handhabungseinrichtungen; Begriffe, Definitionen, Symbole“. VDI-Richtlinie 2860. Mai 1990.

[270] Luigi Villani und Joris De Schutter. „Force Control“. In: *Springer Handbook of Robotics*. Hrsg. von Bruno Siciliano und Oussama Khatib. Berlin: Springer, 2008. Kap. 7, S. 161–186.

[271] Michael Vistein. „Embedding Real-Time Critical Robotics Applications in an Object-Oriented Language“. Diss. Universität Augsburg, 21. Mai 2015.

[272] Michael Vistein, Andreas Angerer, Alwin Hoffmann, Andreas Schierl und Wolfgang Reif. „Flexible and Continuous Execution of Real-Time Critical Robotic Tasks“. In: *Intl. Journal of Mechatronics and Automation* 4.1 (2014).

[273] Michael Vistein, Andreas Angerer, Alwin Hoffmann, Andreas Schierl und Wolfgang Reif. „Instantaneous switching between real-time commands for continuous execution of complex robotic tasks“. In: *2012 IEEE Intl. Conference on Mechatronics and Automation. ICMA 2012*. Proceedings. (Chengdu, China, 5.–8. Aug. 2012). IEEE. 2012, S. 1329–1334.

[274] Michael Vistein, Andreas Angerer, Alwin Hoffmann, Andreas Schierl und Wolfgang Reif. „Interfacing Industrial Robots using Realtime Primitives“. In: *2010 IEEE Intl. Conference on Automation and Logistics. ICAL 2010*. Proceedings. (Hong Kong, China, 16.–20. Aug. 2010). IEEE. 2010, S. 468–473.

[275] Richard Volpe, Issa Nesnas, Tara Estlin, Darren Mutz, Richard Petras und Hari Das. „The CLARAty Architecture for Robotic Autonomy“. In: *2001 IEEE Aerospace Conference*. Proceedings. (Big Sky, USA, 10.–17. März 2001). Bd. 1. 2001, S. 121–132.

[276] Samir Wafa. „Efficient Planning Of Collision-Free Robot Motions“. Masterarbeit. 30. Nov. 2014.

[277] Kenneth Waldron und James Schmiedeler. „Kinematics“. In: *Springer Handbook of Robotics*. Hrsg. von Bruno Siciliano und Oussama Khatib. Berlin: Springer, 2008. Kap. 2, S. 9–34.

[278] Craig Walls. *Modular Java. Creating Flexible Applications with OSGi and Spring*. Raleigh, NC: The Pragmatic Bookshelf, 2009.

[279] Weiss Robotics GmbH & Co. KG. *WSG Series of Intelligent Servo-Electric Grippers. Command Set Reference Manual*. Jan. 2013. URL: http://www.schunk.com/schunk_files/attachments/WSG__Command_Set_Reference_Manual__2013-01_EN.pdf (besucht am 26. 01. 2015).

[280] Günter Wellenreuther und Dieter Zastrow. *Automatisieren mit SPS – Theorie und Praxis*. 4. Aufl. Wiesbaden: Vieweg + Teubner, 2008.

[281] Lars Westerlund. *The Extended Arm of Man: A History of Industrial Robot*. Informationsförlaget, 2000.

[282] Wind River. *VxWorks RTOS*. URL: http://www.windriver.com/products/vxworks/ (besucht am 30. 10. 2014).

[283] World Wide Web Consortium (W3C). *Standards*. URL: http://www.w3.org/standards/ (besucht am 28. 12. 2014).

[284] Heinz Wörn und Uwe Brinkschulte. *Echtzeitsysteme: Grundlagen, Funktionsweisen, Anwendungen*. Berlin: Springer, 2005.

[285] Gerd Wütherich, Nils Hartmann, Bernd Kolb und Matthias Lübken. *Die OSGi Service Plattform. Eine Einführung mit Eclipse Equinox*. Heidelberg: dpunkt.verlag, 2008.

[286] *Xenomai*. URL: http://www.xenomai.org/ (besucht am 22. 10. 2014).

[287] Cezary Zieliński. „Object-oriented robot programming“. In: *Robotica* 15.1 (1997), S. 41–48.

[288] Cezary Zieliński, Wojciech Szynkiewicz und Tomasz Winiarski. „Applications of MRROC++ Robot Programming Framework“. In: *5th Intl. Workshop on Robot Motion and Control. RoMoCo '05*. Proceedings. (Dymaczewo, Poland, 23.–25. Juni 2005). Hrsg. von K. Kozlowski. IEEE. 2005, S. 251–257.

[289] Alois Zoitl und Valeriy Vyatkin. „Different Perspectives – IEC 61499 Architecture for Distributed Automation: The Glass Half Full View“. In: *IEEE Industrial Electronics Magazine* 3.4 (2009), S. 7–26.