

Aufgabenstellung

Titel der Masterarbeit:

Entwicklung und Implementierung einer Software zur aufgabenorientierten Programmierung von Cyber-Physischen-Systemen in der roboterbasierten Montage

Inv.- Nr.: 25867

Verfasser: Christoph Schmitt Betreuer: Joachim Michniewicz

Ausgabe: 01.04.2015

Abgabe: 30.09.2015

Ausgangssituation:

Im Rahmen des Zukunftsprojekts Industrie 4.0 soll die Informatisierung der Produktionstechnik weiter vorangetrieben werden. Dieses Bestreben wird angetrieben durch sich verändernde Rahmenbedingungen im wirtschaftlichen Umfeld. Kürzer werdende Produktlebenszyklen, steigende Variantenvielfalt, Produktindividualisierung sowie sinkende Losgrößen erfordern die Nutzbarmachung der inhärenten Flexibilität bestehender Produktionsanlagen. Unter Einsatz digitaler Repräsentanzen Cyber-Physischer Systeme können Industrieroboter, welche oftmals lediglich repetitive Bewegungen ausführen, flexibilisiert werden. Um dieses Potential speziell im Bereich der Montage demonstrieren zu können, wurde bereits eine Roboterzelle geschaffen, welche Baugruppen aus Lego-Steinen fügen soll. Darüber hinaus wurde die CAD-Software *BrickCAD* auf Basis der *jMonkeyEngine* entwickelt, mit deren Hilfe Baugruppen aus Lego-Steinen durch den Anwender virtuell konstruiert werden können. Im Anschluss daran wird die Lego-Baugruppe durch die Software auf Fügbarkeit hin überprüft und ein Montageplan erstellt. Dieser enthält neben Informationen über die zu fügenden Lego-Steine sowie die Montagereihenfolge auch Informationen über die zu verwendenden Betriebsmittel der Roboterzelle.

Zielsetzung:

Ziel dieser Arbeit ist die Entwicklung und Implementierung einer Software, welche die aufgabenorientierte Programmierung der vorhandenen Roboterzelle auf Grundlage des Montageplans der CAD-Software ermöglicht. Einleitend soll hierzu eine entsprechende Software-Architektur entwickelt werden. Darüber hinaus soll eine Client-Server-Verbindung zwischen Leitrechner und Robotersteuerung geschaffen werden. Die an die Robotersteuerung übermittelten Daten sollen analysiert und der Roboter entsprechend den enthaltenden Informationen aufgabenorientiert programmiert werden. Um die Erweiterbarkeit der Software zu gewährleisten, soll die entwickelte Software mittels UML dokumentiert werden.

Vorgehensweise und Arbeitsmethodik:

- Demonstration der Potentiale von CPS in der Montage
- Entwicklung einer Software-Architektur zur Umsetzung der aufgabenorientierten Programmierung der Roboterzelle
- Programmierung einer Client-Server-Verbindung zum Übertragen der Ausgabedaten von BrickCAD an die Robotersteuerung
- Umsetzung der aufgabenorientierten Programmierung zur Montage der Lego-Steine auf Grundlage der Daten von BrickCAD
- Inbetriebnahme der Roboterzelle
- Dokumentation der erstellten Software mittels UML
- Dokumentation der Arbeit

Vereinbarung:

Mit der Betreuung von Herrn cand.-M.Sc. Christoph Schmitt durch Herrn Dipl.-Ing. Michniewicz fließt geistiges Eigentum des *iwb* in diese Arbeit ein. Eine Veröffentlichung der Arbeit oder eine Weitergabe an Dritte bedarf der Genehmigung durch den Lehrstuhlinhaber. Der Archivierung der Arbeit in der *iwb*-eigenen und nur für *iwb*-Mitarbeiter zugänglichen Bibliothek als Bestand und in der digitalen Studienarbeitsdatenbank des *iwb* als PDF-Dokument stimme ich zu.

München, den 01.04.15

Dipl.-Ing. Michniewicz

cand.-M.Sc. Schmitt

Prof. Gunther Reinhart

Inhaltsverzeichnis

Aufgabenstellung	I
Inhaltsverzeichnis.....	III
Abstract	1
Abkürzungsverzeichnis	2
Verzeichnis der Formelzeichen	4
1 Einleitung	6
2 Stand der Technik und Forschung.....	9
2.1 Einordnung bestehender Methoden zur Programmierung von Roboterzellen.....	9
2.2 Aufgabenorientierte Roboterprogrammierung in der Montage	12
2.3 Cyber-Physische Systeme.....	22
2.3.1 Grundlagen	22
2.3.2 Cyber-Physische Robotik.....	23
3 Kritik am Stand der Technik.....	25
4 Wissenschaftlicher Kontext dieser Arbeit	28
5 Rahmenbedingungen	30
5.1 Die vorhandene CPRE	31
5.1.1 Überblick über das Gesamtsystem	31
5.1.2 Relevante Baugruppen der CPRE	33
5.1.3 Beschreibung der Elektrik	35
5.1.4 SPS und implementierte SPS-Programme	37
5.2 Das CAD-Tool BrickCAD	40
5.2.1 Funktionsweise	40
5.2.2 Schnittstellenbeschreibung	41
6 Zielsetzung und erwartete Vorteile dieser Arbeit	45
7 Anforderungen an die zu entwickelnde Software	46
8 Die entwickelte Softwarearchitektur im Überblick.....	47
8.1 Für die Entwicklung relevante Hardwarekomponenten und Bussysteme	47
8.1.1 Rahmenbedingungen durch die FS100	48
8.1.2 Der Leitrechner	49
8.1.3 Der Kraft-Momenten-Sensor	50
8.1.4 Kommunikation via Ethernet	50

8.1.5 CAN-Bus	52
8.2 Architekturmodell des Softwareentwurfs	52
9 Notwendige Anpassungen des Gesamtsystems	55
9.1 Anpassung des SPS-Programms	55
9.2 Adaption der Pneumatik	56
9.3 Koordinatensysteme des Roboters.....	57
9.4 Festlegung der Speicherbereiche.....	59
9.5 Werkzeugwechselvorgänge.....	60
9.6 Einstellen des Massenträgheitsmoments	62
10 Realisierung der Softwarekomponente für den Leitrechner	66
10.1 Anforderungen an den Client.....	66
10.2 Beschreibung der gewählten Entwicklungsumgebung	66
10.3 Das Windows Form als Main-Thread	67
10.3.1 Die graphische Benutzeroberfläche	67
10.3.2 Struktureller Aufbau des Main-Threads	68
10.3.3 Umsetzung der Threadsicherheit.....	70
10.4 Thread zur Datenübertragung per Ethernet.....	71
10.4.1 Struktureller Aufbau des Threads	72
10.4.2 Anfragen und Rückantworten von Client und Server.....	75
10.4.3 Operation- und Process-Handler	77
10.5 Thread zur Umsetzung des CAN-Bus-Netzwerks	80
10.5.1 Struktureller Aufbau des Threads	81
10.5.2 Methoden der CAN-Bus-Library.....	84
11 Realisierung der Softwarekomponente für die FS100.....	86
11.1 Die Entwicklungsumgebung MotoPlusSDK	86
11.1.1 Funktionsweise	86
11.1.2 Entwickeln und Debuggen mit MotoPlusSDK	86
11.2 Architektur des Servers.....	87
11.2.1 Aufgabenzuordnung der einzelnen Tasks	87
11.2.2 Umsetzung der Prozesssynchronisation.....	88
11.3 Der Task zur Befehls-Datenübertragung via Ethernet.....	89
11.3.1 TCP-IP-Kommunikation	90
11.3.2 Operator- und Process-Command.....	91
11.3.3 Beispiel zur Client-Server-Kommunikation	94

11.4 Der Task zur Steuerung des Roboters	95
11.4.1 Struktureller Aufbau des Tasks	96
11.4.2 Initialisieren der Roboterzelle mittels FctDolInitialize.....	97
11.4.3 Einmessen von User-Koordinatensystemen.....	100
11.4.4 Verarbeiten eines Datensatzes.....	104
11.5 Der Task zur Übertragung der Sensordaten via Ethernet	120
12 Untersuchung zur Implementierung des iwb-Modells auf der Robotersteuerung	122
12.1 VR der Betriebsmitteln.....	123
12.2 Programmierung von CPDs und Funktionsbaugruppen	126
12.2.1 Programmierung einzelner CPDs	126
12.2.2 Programmierung einer CPDC	127
12.2.3 Programmierung der Funktionsbaugruppe Förderband.....	130
12.3 Automatische Konfiguration der CPRC	131
12.4 Identifikation und Zuweisung von Primär- und Sekundärprozessen	135
13 Validierung.....	137
14 Ausblick	140
14.1 Implementierung kraftgeregelten Betriebs.....	140
14.2 Entwicklung und Implementierung eines Lego-Stein-Zuführsystems mit höherer Flexibilität.....	143
14.3 Untersuchungen zur Implementierung aktiv kommunizierender CPDs	144
15 Zusammenfassung	145
16 Literaturverzeichnis	146
17 Anhänge	155
17.1 Funktionaler Aufbau der SPS	155
17.2 Integration eines zweiten Projekts in Visual Studio	155
17.3 UML-Diagramme.....	158
17.3.1 Aktivitätsdiagramme zur Abarbeitung der Targets 1 bis 7 ...	158
17.3.2 Aktivitätsdiagramm zur Funktion FctGripperSelection	166
17.4 Anleitungen zu Bedienung des Demonstrators	167
17.4.1 Rahmenbedingungen:.....	167
17.4.2 Start des Demonstrators	168
17.5 Übersichtstabelle zur Verwendung der P-Variablen.....	169
17.6 Protokoll zum Test der Funktion DoDataset	171

Inhaltsverzeichnis

17.7 Übersichtstabelle zu den validierten Baugruppen	172
18 Verzeichnis verwendeter Software.....	173
19 Inhalt der Daten-CD.....	174
20 Eidesstattliche Erklärung	175

Abstract

This paper presents the development and implementation of a software for task-oriented programming of Cyber-Physical-Systems (CPS) in the robot-based assembly. CPS are systems with computational elements, which reflect the inherent abilities of a physical entity. These embedded systems are working as a network of autonomously interacting elements. Through the high cross-linking level in combination with their virtual description CPS can autonomously work together. They can reconfigure themselves in case of changing boundary conditions. MICHNIEWICZ & REINHART (2015A) propose an architecture, which allows the utilization of CPS in robot-based assembly. However, this method still needs to be validated using a real robot cell.

Necessary hardware has already been configured in a previous work in form of a robot cell. Moreover, a CAD software has been programmed that allows the construction and validation of a product to be assembled by the robot cell. This software provides a virtual representation of the product in form of a so called *Augmented-Assembly-Priority-Plan*.

This work presents a software which allows the control of the robot cell based on this plan. Therefore a Graphical User Interface (GUI) has been implemented. By the help of the implemented software and the robot cell Lego modules can be assembled in batch size 1. In addition, further investigation for the use of CPS in robotics is made.

Abkürzungsverzeichnis

ADL	Architecture Description Language
API	Application Program Interface
AutomationML	Automation Markup Language
BK	Bezugskanten
bt	Bricktype
CAN	Controller Area Network
CAR	Computer Aided Robotics
CPD	Cyber-Physical Device
CPDC	Cyber-Physical-Device-Combination
CPP	Cyber-Physisches Produkt
CPRC	Cyber-Physical-Robot-Cell
CPS	Cyber-Physical Systems
CRC	Cyclic Redundancy Check
CyPros	Cyber-Physische Produktionssysteme
DLC	Data Length Code
DLL	Dynamic Link Library
eMVG	erweiterter Montagevorranggraph
FIFO	First In First Out
FP	Funktionsprimitiva
GRAPHCET	GRAphe Fonctionnel de Commande Etapes/Transitions
GUI	Graphical User Interface
I/O	Input / Output
ISO	International Standardization Organisation
KIF	Knowledge Integration Framework
KMS	Kraft-Momenten-Sensor
KOP	Kontaktplan
MP	Manipulationsprimitiv
MTM	Methods-Time Measurement

OSI	Open Systems Interconnection
PDDL	Planning Domain Definition Language
PG	Production Graph
PHG	Programmierhandgerät
PMMA	Polymethylmethacrylat
RDF	Resource Description Framework
SCARA	Selective Compliance Assembly Robot Arm
SDK	Software Development Kit
SysML	Systems Modeling Language
TCP	Tool Center Point
TCP/IP	Transmission Control Protocol / Internet Protocol
TMU	Time Measurement Unit
to	Toolorientation
tp	Toolposition
tt	Tooltype
UDP	User Datagram Protocol
UML	Unified Modeling Language
VR	Virtuelle Repräsentanz
XML	Extensible Markup Language

Verzeichnis der Formelzeichen

Formelzeichen – [Physikalische Einheit] – Beschreibung inkl. Indizes und Abkürzungen

Lateinische Zeichen

$A_{AngleBase}$	[$-$]	Transformationsmatrix vom Koordinatensystem Base ins Koordinatensystem Angle
$A_{BaseUK(j)}$	[$-$]	Transformationsmatrix vom Userkoordinatensystem j ins Koordinatensystem Base
$A_{MB(oi)}$	[$-$]	Rotationsmatrix zur Darstellung eines Vektors aus dem Koordinatensystem des Lego-Steins B im User-Koordinatensystem M
$A_{PulseAngle}$	[$-$]	Transformationsmatrix vom Koordinatensystem Angle ins Koordinatensystem Pulse
\vec{a}	[μm]	Annäherungsvektor
D_i	[mm]	Durchmesser des Körpers i
D_{i-1}	[$-$]	Rotationsmatrix zwischen dem (i-1)-ten und dem i-ten Koordinatensystem
h_i	[mm]	Höhe des Körpers i
iHeight	[μm]	Höhe der bereits gefügten Lego-Baugruppe
k_M	[$-$]	Skalierungsfaktor
M	[kg]	Gesamtmasse
M_i	[μm]	Messpunkt i
m_i	[kg]	Masse des Körpers i
\vec{n}	[μm]	Normalenvektor
\vec{o}	[μm]	Orientierungsvektor
${}_B\vec{r}_{BG}$	[μm]	Offsetvektor im Koordinatensystem des Lego-Steins B vom Steinursprung zum Angriffspunkt des Greifers

$M\vec{r}_{0L}$	[μm]	Vektor im User-Koordinatensystem M zum Zentrum der Ursprungs-Lego-Noppe
$M\vec{r}_{LB}$	[μm]	Vektor im User-Koordinatensystem M zum Ursprung des Lego-Steins
$Pulse\vec{r}_{0Pi}$	[PULSE]	Vektor im Koordinatensystem Pulse zum Punkt P_i mit $i \in \{1,2,3,4,5\}$
$UK(0)\vec{r}_{0P1}$	[μm]	Vektor zum Punkt P1
$UK(0)\vec{r}_{OffsetFeeder}$	[μm]	Vektor zur Steinaufnahme in der Zuführeinheit
$UK(0)\vec{r}_{OffsetTool}$	[μm]	Vektor des nötigen Werkzeugoffsets
$UK(0)\vec{r}_{S.UK0}$	[μm]	Verschiebung des User-Koordinatensystems 0
$UK(1)\vec{r}_{OffsetAssembly}$	[μm]	Vektor zum Montagepunkt auf dem WT
$UK(1)\vec{r}_{S.UK1}$	[μm]	Verschiebung des User-Koordinatensystems 1
$UK(j)\vec{r}_{0Pi}$	[μm]	Vektor im User-Koordinatensystem j zum Punkt P_i mit $i \in \{1,2,3,4,5\}$
U_{AC}	[V]	Wechselspannung
T_{i-1}	[\cdot]	Homogene Transformationsmatrix zwischen dem (i-1)-ten und dem i-ten Koordinatensystem
t_{i-1}	[μm]	Translationsvektor zwischen dem (i-1)-ten und i-ten Koordinatensystem im (i-1)-ten Koordinatensystem

Griechische Zeichen

$_{si}\theta_{jji}$	[kg mm ²]	Massenträgheitsmoment um die j-Achse des Körpers i im körperfesten Koordinatensystem des Körpers i mit $j \in \{x, y, z\}$
---------------------	-----------------------	--

1 Einleitung

Die Produktion unterliegt gegenwärtig einem Wandlungsprozess aufgrund von Megatrends wie Globalisierung, Dynamisierung der Produktlebenszyklen oder steigender Mobilität (ABELE & REINHART 2011). Mit Bestrebungen unter dem Schlagwort *Industrie 4.0* sollen diese Veränderungen beherrscht werden können. Technologische Grundlagen hierfür sind Cyber-Physical Systems (CPS) und das Internet der Dinge (SENDLER 2013). Diese sollen dazu beitragen, Herausforderungen wie kürzer werdende Produktlebenszyklen, steigende Variantenvielfalt, Produktindividualisierung und sinkende Losgrößen bewältigen zu können (RUßWURM 2013).

Der Einsatz von Industrierobotern bei niedrigen Stückzahlen ist gegenwärtig gering (BACKHAUS 2014). Die Gründe hierfür sind fehlende technische Lösungen und hohe Kosten für automatisierte Robotersysteme (ZÄH ET AL. 2004). Nach BARTSCHER (2011) können die Kosten zur Integration eines Industrieroboters in eine Automatisierungslösung das bis zu zehnfache der Anschaffungskosten des Roboters betragen. Ursache sind hohe manuelle Aufwände zur Betriebsmittelauswahl, Produktionsplanung und Programmierung (LOTTER 2012). Diese Tatsache steht im Gegensatz zu der inhärenten Flexibilität von Robotern. Aufgrund ihrer kinematischen Freiheitsgrade, freier Programmierbarkeit und einem großen Spektrum an Peripheriegeräten können diese an sich ändernde Produktionsbedingungen angepasst werden. Wegen der notwendigen manuellen Programmierung sind der Aufwand und die Stillstandszeiten der Anlage weiterhin hoch, weshalb die Anlagenflexibilität aus betriebswirtschaftlichen Gründen nur selten genutzt werden kann.

Die Montage ist gegenwärtig der größte Kostenverursacher im Produktentstehungsprozess, da diese bis zu 70% der Gesamtfertigungszeit einnimmt (LOTTER 2012). Im Hinblick auf Bestrebungen zur Flexibilitätssteigerung beim Einsatz von Robotern im Fertigungsprozess bietet die variantenreiche Montage daher großes Potential (MICHNIEWICZ & REINHART 2015B). Um dieses abzurufen, existieren diverse Systemarchitekturen zur aufgabenorientierten Programmierung, bei denen der Anwender die vom Roboter auszuführende Aufgabe beschreiben muss. Wie Abbildung 1-1 entnommen werden kann, ist ein Überblick über den *Stand der Technik und Forschung* zu dieser Thematik ist in Kapitel 2 zu finden. Im Kapitel *Kritik am Stand der Technik* wird aufgezeigt, welche Mängel die bereits entwickelten Forschungsansätze aufweisen.

1	Einleitung
2	Stand der Technik und Forschung
3	Kritik am Stand der Technik
4	Wissenschaftlicher Kontext dieser Arbeit
5	Rahmenbedingungen
6	Zielsetzung und erwartete Vorteile dieser Arbeit
7	Anforderungen an die zu entwickelnde Software
8	Die entwickelte Softwarearchitektur im Überblick
9	Notwendige Anpassungen des Gesamtsystems
10	Realisierung der Softwarekomponente für den Leitrechner
11	Realisierung der Softwarekomponente für die FS100
12	Untersuchung zur Implementierung des iwb-Modells auf der Robotersteuerung
13	Validierung
14	Ausblick
15	Zusammenfassung

Abbildung 1-1: Aufbau dieser Arbeit (Quelle: Eigene Ausarbeitung).

Diese Arbeit steht im Kontext zum Forschungsprojekt CyPros (Cyber-Physische Produktionssysteme), das die Steigerung von Produktivität und Flexibilität produzierender Unternehmen auf der Grundlage Cyber-Physischer Produktionsysteme ermöglichen soll. Dazu wird eine Methode entwickelt, mit der die Montage variantenreicher Produkte in modularen Roboterzellen automatisiert durchgeführt werden kann (MICHNIEWICZ & REINHART 2015A). Im Gegensatz zu den in Kapitel 2 vorgestellten Systemarchitekturen umfasst dieser Ansatz die automatische Extrahierung der Produktanforderungen aus den CAD-Daten und die Ableitung valider Montagereihenfolgen. Des Weiteren beinhaltet diese Methode die Selbstbeschreibung der Betriebsmittelfähigkeiten Cyber-Physischer Roboterzellen sowie einen automatischen Abgleich derer mit den Produktanforderungen. Ergebnis ist die automatisierte, aufgabenorientierte Programmierung des Roboters. Kapitel 4 dient dazu, einen Überblick über dieses Gesamtsystem geben zu können.

Ziel dieser Arbeit ist es, eine Software zu entwickeln und zu implementieren, mit deren Hilfe eine Cyber-Physische Roboterzelle zur roboterbasierten Montage auf Grundlage des in Kapitel 4 vorgestellten Systementwurfs gesteuert werden

kann. Die dazu notwendige Hardware wurde bereits ausgewählt, das Anlagenlayout festgelegt und eine entsprechende Roboterzelle konzipiert. Diese stellt gewisse Rahmenbedingungen an die Entwicklung der Software, was in Kapitel 5 ausführlich erläutert wird. In Kapitel 6 werden die Ziele dieser Arbeit und die zu erwartenden Vorteile, die aus dieser Arbeit hervorgehen, dargelegt. Im Hinblick auf die Softwareentwicklung werden einleitend die Anforderungen konkretisiert. Kapitel 7 ist dieser Thematik gewidmet. Die daraus abgeleitete und entwickelte Softwarearchitektur wird in Kapitel 8 dargelegt. Diese macht einige Anpassungen an der bestehenden Hardware erforderlich, welche in Kapitel 9 behandelt werden. Der Softwareentwurf fußt auf einem Client-Server-Modell, bestehend aus Robotersteuerung und Leitrechner. In Kapitel 10 wird daher die entwickelte Software für den Leitrechner, in Kapitel 11 die für die Robotersteuerung erläutert. Im Anschluss daran finden sich weitergehende Untersuchungen zur Implementierung von CPS in der Robotik. Die implementierte Software sowie die Funktionsweise der Roboterzelle werden in Kapitel 13 validiert. Ein Ausblick hinsichtlich weiterer Forschungsarbeiten zur Cyber-Physischen Robotik ist in Kapitel 14 gegeben. Das letzte Kapitel fasst die Ergebnisse dieser Arbeit abschließend zusammen.

2 Stand der Technik und Forschung

Einleitend wird in diesem Kapitel ein Überblick über die bestehenden Methoden zur Programmierung von Roboterzellen gegeben. Im Anschluss daran finden sich Ausführungen zur aufgabenorientierten Programmierung. Bestehende Forschungsansätze und Modelle werden dargelegt. Das Potential Cyber-Physischer Systeme (CPS) kann auch in der Robotik genutzt werden. Dahingehend werden einleitend Grundlagen zu CPS erläutert. Im Anschluss daran finden sich Ausführungen zu Forschungsprojekten, die sich mit CPS in der Robotik befassen.

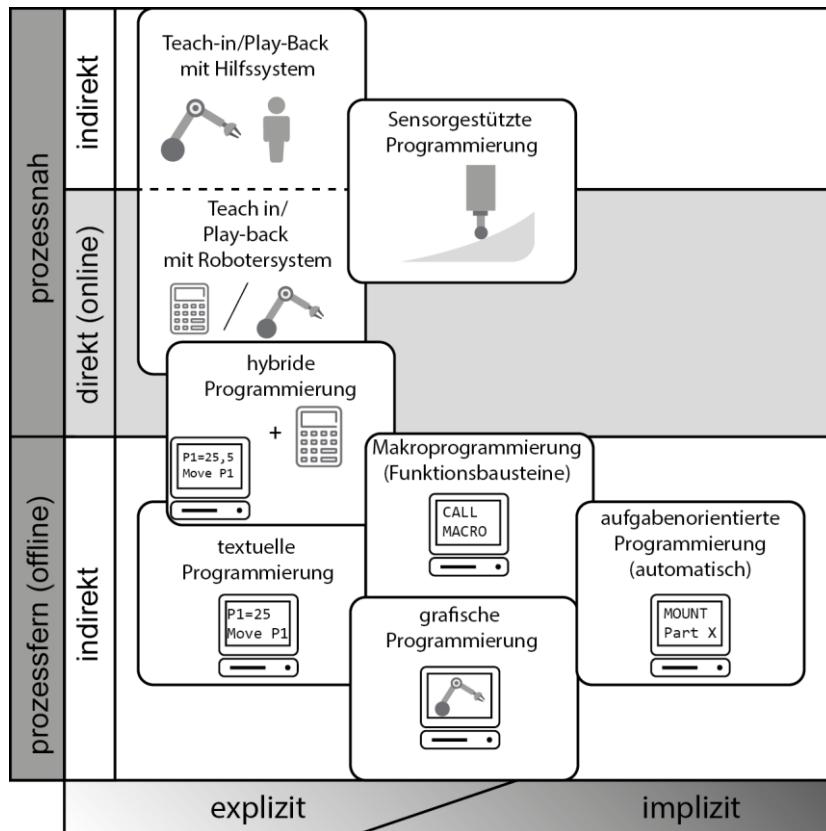
2.1 Einordnung bestehender Methoden zur Programmierung von Roboterzellen

Zur Programmierung von Roboterzellen existiert eine Vielzahl von Methoden, die nach HUMBURGER (1998) wie folgt eingeteilt werden können.

Als häufig in Betracht gezogene Eigenschaft zur Einteilung von Programmierverfahren dienen der Ort und die Inanspruchnahme des Produktionssystems während der Programmierung des Roboters. Erfolgt diese mit dem Roboter, so kann die Vorgehensweise als Online-Verfahren kategorisiert werden. Nachteil dieser Methode ist, dass der Roboter während der Programmierung für die Produktion ausfällt (HAUN 2013). Wird das Programm hingegen prozessfern, ohne Verwendung der Roboterzelle erstellt, so kann dieses Verfahren als Offline-Programmierung bezeichnet werden. Offline erstellte Programme müssen im Normalfall allerdings an die realen Gegebenheiten angepasst werden (KUGELMANN 1999). Nach KRUG (2013) können die Roboterprogrammiermethoden zusätzlich in prozessnah und prozessfern eingeteilt werden. Diese Einteilung wird in Abbildung 2-1 wiedergegeben. (HUMBURGER 1998)

Daneben können Programmierverfahren in explizite und implizite Verfahren eingeteilt werden. Erstere sind solche, bei denen der Programmierer alle Weginformationen sowie die Steuerung der Peripherie von Hand spezifizieren muss. Der Prozess wird dazu vom Programmierer in Einzelschritte unterteilt, die durch den verfügbaren Befehlsvorrat der Robotersteuerung abgearbeitet werden können. Implizite Verfahren hingegen sind durch ein höheres Abstraktionsniveau gekennzeichnet. Die erforderlichen Roboterbewegungen werden auf Grundlage einer abstrakten Aufgabenbeschreibung automatisch generiert. Der Übergang

zwischen expliziter und impliziter Roboterprogrammierung ist fließend, da beispielsweise einzelne Funktionen in Form von Makros den Prozess bereits abstrahieren. (HUMBURGER 1998)



*Abbildung 2-1: Einteilung bestehender Roboterprogrammierverfahren
(In Anlehnung an HUMBURGER 1998).*

Teach-In- und Playback-Verfahren zählen zu den expliziten, prozessnahen Verfahren. Dabei können beide entweder den direkten, als auch den indirekten Verfahren zugeordnet werden. Dies ist abhängig davon, ob ein Hilfssystem oder die reale Roboterzelle zum Programmieren verwendet wird. Beim Teach-In-Verfahren werden relevante Punkte im Arbeitsraum angefahren und gespeichert. Diese Vorgehensweise kann den direkten Verfahren zugeordnet werden, sofern der Roboter hierzu genutzt wird. Wird hingegen ein Hilfssystem verwendet, das beispielsweise den kinematischen Aufbau des Roboters modelliert, so handelt es sich um ein indirektes Verfahren. Beim Playback-Verfahren hingegen sollen komplexe Bahnkurven reproduziert werden können. Hierzu wird die Position des Roboters in einem meist fest vorgegebenen Zeitintervall gespeichert (WECK 2001). Beim sensorbasierten Verfahren wird die vom Roboter zu beschreibende Bahn nur grob vorgegeben. Mit Hilfe geeigneter Sensorik wird im Anschluss daran die exakte Bahn eingelernt (MATTHIAS ET AL. 2004). (HUMBURGER 1998)

Die textuelle Programmierung ist den prozessfernen Offline-Programmiermethoden zugeordnet. Hierbei wird der Roboter explizit in der zumeist steuerungsspezifischen Programmiersprache mit Hilfe eines rechnergestützten Systems programmiert (BACKHAUS 2014). Die Makro-Programmierung kann ebenfalls zu den Offline-Programmierverfahren gezählt werden. Zur Programmerstellung stehen dem Anwender vordefinierte Makros bzw. Funktionsmodule zur Verfügung. Diese beinhalten Programme, die auf den Grundfunktionen der Roboterprogrammiersprache basieren und vom Benutzer lediglich parametrisiert werden müssen. Dies ist besonders in Fertigungsbereichen mit wiederkehrenden Prozessen hilfreich, etwa beim Schweißen variantenreicher, ähnlicher Produkte (MITSI ET AL. 2005). Der Aufruf der Makros kann mit Hilfe eines vom Anwender zu spezifizierenden graphischen Programmablaufplans erfolgen (MACKENZIE & ARKIN 1998).

In Anwendungsbereichen, bei denen eine hohe Positioniergenauigkeit erforderlich ist, kommen hybride Programmierverfahren zum Einsatz. Hierbei wird das Programmgerüst mit Hilfe eines textuellen Programmeditors offline erstellt. Im Anschluss daran werden die exakten Positionen und Orientierungen unter Verwendung prozessnaher Programmierverfahren geteacht. (HUMBURGER 1998)

Bei der graphischen Roboterprogrammierung kommen entweder erweiterte CAD-Systeme oder herstellerspezifische Lösungen mit 3D-Funktionalität zum Einsatz. Mit diesen häufig als CAR-Systeme (Computer Aided Robotics) bezeichneten Programmiersystemen wird der zu programmierende Ablauf auf Grundlage eines dreidimensionalen Modells der Roboterzelle simuliert. Hierdurch können Programmfehler aufgedeckt und Abläufe optimiert werden (HUMBURGER 1998). Unter Verwendung von Postprozessoren wird im Anschluss das Roboterprogramm erstellt und auf die Robotersteuerung übertragen. (WECK 2001)

Da der Programmieraufwand von Roboterapplikationen in der Regel hoch ist, steigen infolge von Produktindividualisierung und sinkenden Losgrößen die Kosten (CEDERBERG ET AL. 2005). Mit Hilfe von aufgabenorientierten Programmiersystemen kann dieser Problematik entgegengetreten werden, da diese je nach Ansatz die teil- oder vollautomatisierte Programmerstellung ermöglichen (HUMBURGER 1998). Dahingehend existiert eine Vielzahl von Forschungsansätzen zur aufgabenorientierten Roboterprogrammierung für Schweiß-, Klebe- oder Montagetätigkeiten in der Fertigung. (LÜDEMANN-RAVIT 2005)

Nach FREUND & HECK (1990) werden beim aufgabenorientierten Programmierdogma ganze Bewegungsfolgen des Roboters spezifiziert. Demnach muss der

Programmierer oftmals nur Anfangs- und Endzustand beschreiben. Dies ist auf unterschiedlichen Abstraktionsebenen möglich, beispielsweise durch implizite Befehle. Ein Beispiel hierfür ist der Befehl „Füge Stift X in Bohrung 1 von Platte“ (HAUN 2013). Von höherem Abstraktionsniveau gekennzeichnet ist die automatische Generierung solcher Befehle aus dem Montagevorranggraphen nach CUIPER (2000). In der höchst denkbaren Abstraktionsebene wird die Roboterapplikation aus der Produktspezifikation generiert (HUMBURGER 1998). Allen Ansätzen liegt ein Umweltmodell zugrunde, auf Basis dessen die abstrakten Befehle spezifiziert werden können (WECK 2001). Aufgrund der großen Varianz aufgabenorientierter Programmieransätze werden im nachfolgenden Kapitel Systeme betrachtet, die für die Montage geeignet sind.

2.2 Aufgabenorientierte Roboterprogrammierung in der Montage

Nach WECK (2001) basieren alle aufgabenorientierten Ansätze zur Roboterprogrammierung für die Montage auf der Aufteilung der Prozesse in Teilbereiche. Jeder Teilbereich arbeitet die ihm zugeordnete Aufgabe ab. Abbildung 2-2 veranschaulicht die von WECK (2001) beschriebene Struktur.

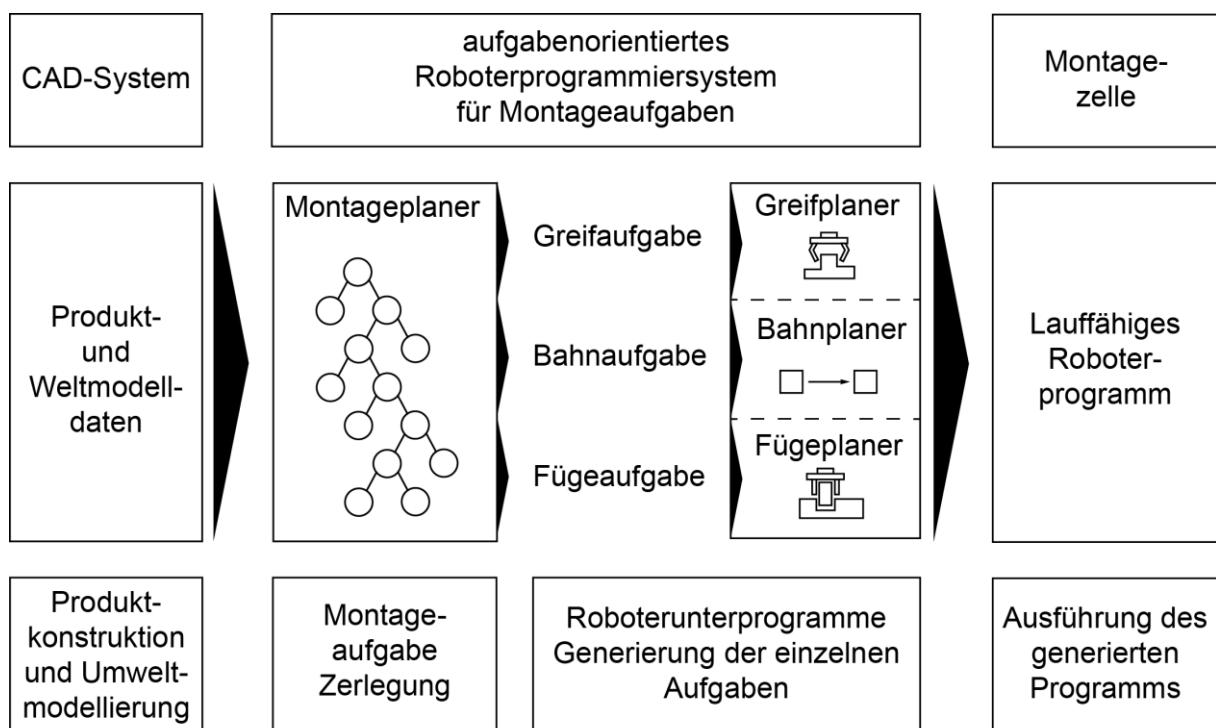


Abbildung 2-2: Struktur eines aufgabenorientierten Programmiersystems für die Montage (Quelle: WECK 2001).

Demnach beinhaltet das CAD-System die Produkt- und Weltmodelldaten. Aus diesen kann mithilfe des Montageplaners die Aufgabe in einzelne Schritte zerlegt werden. Zur Montage einzelner Bauteile müssen diese durch den Roboter aufgenommen werden. Diese Aufgabe wird durch den Greifplaner erledigt. Die zum Fügen für die Roboterzelle relevanten Informationen werden durch den Fügeplaner aus den CAD-Daten extrahiert. Um die vom Roboter zu beschreibende Bahn zum Fügen von Bauteilen ermitteln zu können, ist in aller Regel ein Bahnplaner implementiert. Die Ergebnisse dieser drei Planungselemente werden im lauffähigen Roboterprogramm vereinigt, das auf die Robotersteuerung übertragen und ausgeführt wird. (WECK 2001)

Nach dieser allgemeinen Beschreibung eines aufgabenorientierten Programmiersystems werden im Folgenden konkrete Projekte zu dieser Thematik vorgestellt. Aufgrund der Systemkomplexität beschäftigen sich einige Forschungsprojekte lediglich mit einzelnen Komponenten eines aufgabenorientierten Roboterprogrammiersystems.

KAUFMAN ET AL. (1996) beispielsweise konzentrieren sich hauptsächlich auf die Montageplanung. Das vorgestellte System basiert auf zwei Teilen. Im ersten Programmteil werden Fügesequenzen anhand geometrischer Informationen aus den CAD-Daten sowie den Greiferdaten ermittelt. Diese beschreiben geometrisch zulässige Fügeoperationen ohne Kollision. Als Input hierfür ist neben einer CAD-Datei des montierten Produkts eine weitere Datei erforderlich. In dieser ist beschrieben, wie die einzelnen Bauteile zusammengefügt sind, welche Subbaugruppen zuerst gefügt werden müssen und in welcher Richtung die Bauteile vorzugsweise zu fügen sind. Im zweiten Programmteil wird mittels Standardsuchalgorithmen sowie einer implementierten Kostenfunktion der Montageplan erstellt. Die Planung basiert auf der Assembly-by-Disassembly-Strategie (BACKHAUS 2014). Dies bedeutet, dass das montierte Produkt im Endzustand betrachtet wird. Ausgehend von diesem werden nacheinander Bauteile demonstriert, sofern dies ohne Verletzung von Randbedingungen wie z.B. Kollisionsfreiheit möglich ist. In dem von KAUFMAN ET AL. (1996) vorgestellten System werden auf Grundlage des Montageplans vordefinierte Funktionen zur Steuerung eines Parallelroboters aufgerufen. Der Montagevorgang lässt sich in nacheinander durchzuführende Einzelschritte zerlegen. Für jeden dieser Einzelschritte ist eine parameterbasierte Funktion implementiert. Die notwendigen Parameter zum Aufruf der Funktionen werden den CAD-Daten entnommen. An dieser Stelle sei angemerkt, dass zur Systemvalidierung lediglich die Montage eines einzigen Produkts herangezogen wurde. Hinsichtlich der Qualität der automatisiert programmierten Montage machen KAUFMAN ET AL. (1996) keine Aussage.

Die Arbeit von WEEKS (1997) konzentriert sich auf die Entwicklung eines aufgabenorientierten Greif- und Bahnplanungssystems für die automatisierte Montage mittels SCARA-Robotern (Selective Compliance Assembly Robot Arm). Zur Greifplanung werden dazu parallele, kontaktfreie Greifflächenpaare des zu greifenden Werkstücks analysiert. Zur Bahnplanung wird der diskretisierte Konfigurationsraum des SCARA-Roboters durch zwei Potentiale beschrieben. Das anziehende Potential sinkt bei der Annäherung des TCPs (Tool Center Point) zum Zielpunkt. Das abstoßende Potential steigt im Umfeld von Bauteilen im Konfigurationsraum, mit denen eine Kollision möglich ist. Durch die Überlagerung der beiden Potentiale kann mit Hilfe des Best-First-Verfahrens eine mögliche Bahn des Roboters ermittelt werden. Die Erzeugung des resultierenden Roboterprogramms wird in der vorliegenden Arbeit nicht beschrieben. Darüber hinaus wird die Problemstellung durch den kinematischen Aufbau von SCARA-Robotern stark vereinfacht.

CUIPER (2000) nutzt Petri-Netze zur Entwicklung eines aufgabenorientierten Montagesystems. Abbildung 2-3 gibt den hierarchischen Aufbau der Montagevorgangsbeschreibung wieder. Der Ablaufplan wird in dem implementierten System mit Hilfe eines GUI (Graphical User Interface) vom Anwender erstellt. Hierzu sind die montagerelevanten Grundfunktionalitäten hinterlegt und können vom Anwender zum Ablaufplan hinzugefügt werden. In Abbildung 2-3 wird dies durch das Modell auf der linken Seite der Darstellung in Abstraktionsstufe II wiedergegeben. Der Ablaufplan wird mit der systemhinterlegten betriebsmittelbezogenen Arbeitsgangfolge verglichen. Ergebnis ist ein objektbezogenes Petri-Netz mit dessen Hilfe der Steuerungscode für die Anlage generiert werden kann. Die für den Montagevorgang relevanten Punkte im Arbeitsraum müssen vom Anwender spezifiziert werden. Die CAD-Daten werden in diesem Ansatz nicht in Betracht gezogen.

MOSEMANN & WAHL (2001) beschreiben eine Methode, mit deren Hilfe komplexe Montagefolgen in Aktionsprimitive zerlegt werden. Dieser Ansatz wird von THOMAS (2009) erweitert. Basis für die Ermittlung der Montagereihenfolge sind montagerelevante Kongruenzbedingungen. Diese müssen vom Anwender im CAD-Programm festgelegt werden. Bei einer Schraubverbindung beispielsweise muss die Schraube als „Shaft“, die zugehörige Bohrung im Bauteil als „Hole“ spezifiziert werden (THOMAS 2009). Aus diesen Angaben lassen sich Montageplan und Aktionsprimitivnetz ermitteln. Als Aktionsprimitiv bezeichnen MOSEMANN & WAHL (2001) elementare, sensorbasierte Roboterfunktionen. So kann beispielsweise spezifiziert werden, welches Drehmoment an der z-Achse

maximal auftreten darf. Bei der zur Systemvalidierung herangezogenen Montageaufgabe soll eine Schaltschrank-Steckdose auf einer Hutschiene angebracht werden (THOMAS 2009). Neben der fehlenden Varianz der Montageaufgabe ist zu kritisieren, dass die Bewertung der Prozessqualität fehlt.

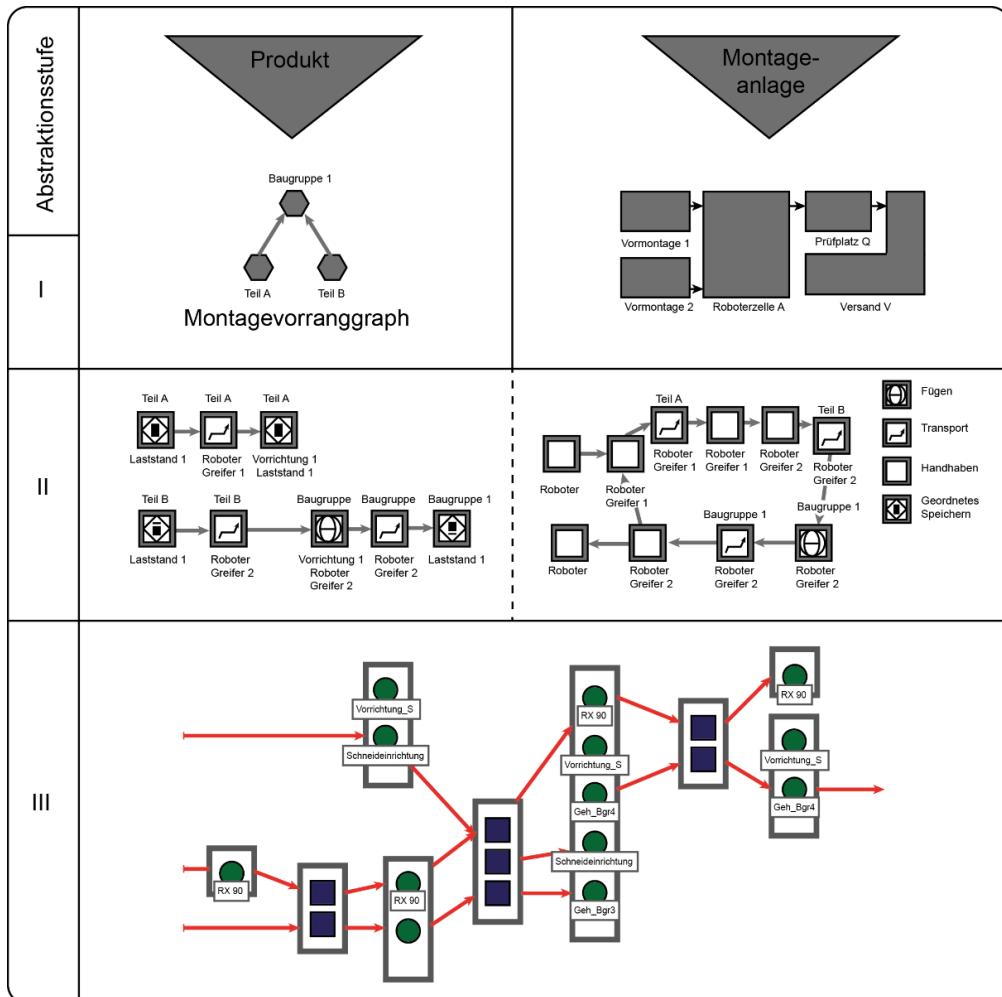


Abbildung 2-3: Hierarchische Darstellung der Montagevorgangsbeschreibung (In Anlehnung an CUIPER 2000).

LÜDEMANN-RAVIT (2005) stellt ein System vor, mit dessen Unterstützung die Programmierung von Roboter-Zellen automatisiert werden kann. Hierzu ist eine roboterunabhängige Sprache implementiert, mit der das System auf die zu erwartenden Anwendungsfälle zugeschnitten werden kann. Dies erfolgt mittels vordefinierter, parametrisierbarer Schablonen. Auf Grundlage derer wird unter Verwendung maschinenlesbarer Daten das Roboterprogramm in einem neutralen Zwischencode erstellt. Dieser wird mit Hilfe einer Simulationsumgebung validiert. Sofern das Ergebnis zufriedenstellend ist, wird der Code durch einen Postprozessor in die herstellerabhängige Roboterprogrammiersprache transferiert und auf die Robotersteuerung übertragen. Das von LÜDEMANN-RAVIT (2005) er-

fordert Expertenwissen, da das System mit Hilfe der systemeigenen Spezifikationssprache zur Definition der Programmschablonen konfiguriert werden muss. Darüber hinaus fehlt dieser Softwarearchitektur eine Schnittstelle zu vorhandenen CAD-Daten. Die inhärente Flexibilität von Roboterzellen ist daher nur teilweise nutzbar.

Der von BRECHER ET AL. (2004) im Rahmen des Forschungsprojekts Porthos entwickelte Ansatz ermöglicht die aufgabenorientierte Programmierung von Roboterzellen. Für alle Komponenten einer Roboterzelle sind die jeweils relevanten Daten modelliert. Für einen Greifer wird beispielsweise hinterlegt wie dieser anzusteuern ist oder welche TCP-Verschiebung berücksichtigt werden muss. Die Positionsdaten der einzelnen Elemente werden nicht aus einem CAD-Modell der Roboterzelle ausgelesen, sondern müssen vom Programmierer von Hand eingegeben werden. Auch dieser Ansatz basiert auf vordefinierten implementierten Funktionen, die den Elementarfunktionen der Handhabungsaufgabe entsprechen. Beispiele hierfür sind Funktionen zum Entnehmen von Bauteilen oder zum Bestücken von Paletten. Zur Programmierung des Systems muss der Anwender mit Hilfe eines GUI einen Programmablaufplan erstellen. Hierbei stehen die angesprochenen Elementarfunktionen zur Verfügung. Die notwendigen Parameter für die Funktionen werden soweit möglich aus den modellierten Komponenten der Roboterzelle abgeleitet. Fehlenden Informationen müssen von Hand durch den Anwender spezifiziert werden. Auf Grundlage des Programmablaufplans wird im Anschluss daran das Roboterprogramm automatisch erstellt. Die vorhandenen Produktinformationen in Form von CAD-Daten werden durch den von BRECHER ET AL. (2004) vorgestellten Ansatz nicht in Betracht gezogen. Darüber hinaus fehlt die Validierung der Systemarchitektur an industriell relevanten Anwendungsfällen.

Der Ansatz von MEDELLIN ET AL. (2010) basiert auf der Zerlegung komplexer CAD-Modelle in primitive Geometrien. Dazu wird das CAD-Modell eines Einzelkörpers durch Quader verschiedener Größen approximiert. Im Anschluss wird die Reihenfolge ermittelt, in der die Würfel zu fügen sind. Hierzu sind Methoden der MTM-Analyse (Methods-Time Measurement) implementiert. Jede Einzelaktion wird in TMUs (Time Measurement Unit) bewertet. Basierend auf den Ergebnissen kann die optimale Fügereihenfolge bestimmt und das Programm für die Roboterzelle erzeugt werden. Die Quader werden schichtweise mit Hilfe eines SCARA-Roboters durch Klebeverbindungen gefügt. Hinsichtlich dieser Softwarearchitektur sei angemerkt, dass die Zerlegung eines Einzelkörpers in Quader verschiedener Größen und die anschließende Montage derer keinen relevanten industriellen Anwendungsfall darstellt.

MAAß ET AL. (2008) stellen ein Konzept vor, bei dem der Roboter durch den Anwender aufgabenorientiert programmiert werden kann. Hierzu sind sogenannte Manipulationsprimitive (MP) implementiert. Diese bestehen aus Bewegungs-, Transitions- und Werkzeugbefehl. Mit Hilfe der Transitionsbefehle legt der Anwender fest, wann zum nächsten MP in der Prozesskette übergegangen wird. Die Prozesskette besteht aus einem MP-Netzwerk, das mit Hilfe eines GUI vom Anwender erstellt wird. Hierbei werden auch die relevanten Eigenschaften und Parameter der einzelnen MPs definiert. Im Anschluss daran wird das Programm für den Roboter erzeugt. Es sei darauf hingewiesen, dass das von MAAß ET AL. (2008) vorgestellte System sensorgeführt ist. Mit Hilfe eines Kraft-Momenten-Sensors und eines implementierten Regelkreises werden die auftretenden Kräfte und Momente gemessen und beim Montagevorgang berücksichtigt. Die Verwendung von CAD-Daten ist in dem vorgestellten Softwareentwurf nicht vorgesehen. Zur Validierung wird die bereits von THOMAS (2009) verwendete Montage einer Elektro-Komponente auf einer Hutschiene herangezogen.

HUCKABY & CHRISTENSEN (2012) stellen eine Klassifizierungs-Systematik zur vereinfachten Roboterprogrammierung vor. Dazu werden die Aufgaben sowie die Fähigkeiten des Roboters mittels SysML (Systems Modeling Language) modelliert. Die Fähigkeiten werden dabei als Skill Primitives bezeichnet. Da bei deren Verwendung oftmals physikalische Einschränkungen berücksichtigt werden müssen, sind diesen die systembedingten Randbedingungen zugeordnet. Skills, wie zum Beispiel *Fixieren* sind die Fähigkeiten *Schrauben*, *Kleben* und *Nieten* hierarchisch untergeordnet. Die Aufgabenbeschreibung erfolgt in einer hierfür entwickelten Planning Domain Definition Language (PDDL). Mit dieser werden beispielsweise auch die Roboterfähigkeiten oder die Erreichbarkeit der Devices im Arbeitsraum definiert (HUCKABY ET AL. 2013). Im Anschluss wird, basierend auf der SysML-Klassifikation und den in PDDL beschriebenen Systemeigenschaften das Programm für den Roboter erzeugt und dieses auf die Steuerung übertragen (HUCKABY 2014). Auch dieser Entwurf lässt die CAD-Daten außer Betracht. Die Software wird anhand einer einzelnen, komplexen Montagebaugruppe validiert. Es ist nicht ersichtlich, wie hoch der erforderliche Aufwand zu bewerten ist, der notwendig ist, um ein anderes Produkt montieren zu lassen.

Ein modell-basiertes Konzept zur Programmierung einer Roboterzelle in der Montage stellen HERFS ET AL. (2013) vor. Grundlage des Entwurfs ist eine dreistufige Struktur. Siehe dazu Abbildung 2-4. In der Modell-Ebene werden die notwendigen Informationen durch den Anwender formalisiert. Dies beinhaltet Ele-

mentarfunktionen der Roboterzelle, zur Montage notwendige Einzelschritte sowie eine Beschreibung der Zellenstruktur und der geometrischen Beziehungen. Die Control-Ebene besteht aus einem Planungsmodul, einem Zellenmodell und dem Aktions-/Eventmodul. Aufgabe des Planungsmoduls ist es, anhand der Elementarfunktionen und der hinterlegten Montageschritte eine Reihenfolge von Aktionen zu generieren, mit deren Hilfe die Montageaufgabe erledigt werden kann. Das Zellenmodell gibt den aktuellen Systemzustand der Roboterzelle wieder. Auf Treiber-Ebene sind die hardware-spezifischen Eigenheiten eines jeden Devices der Roboterzelle hinterlegt.

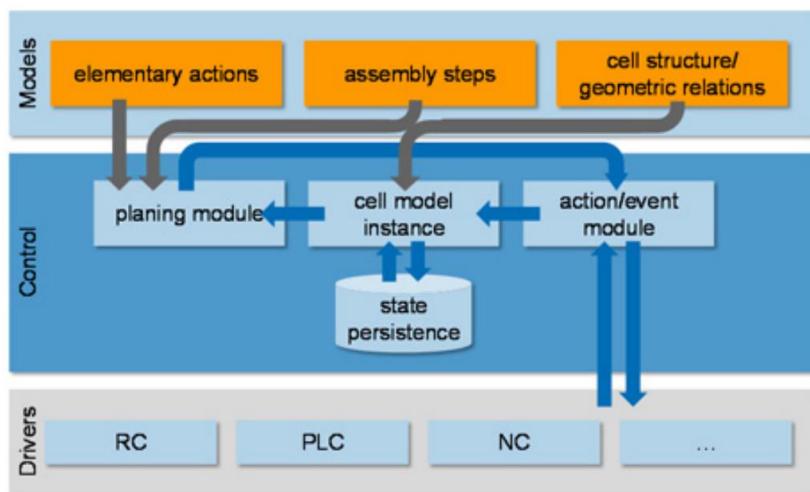
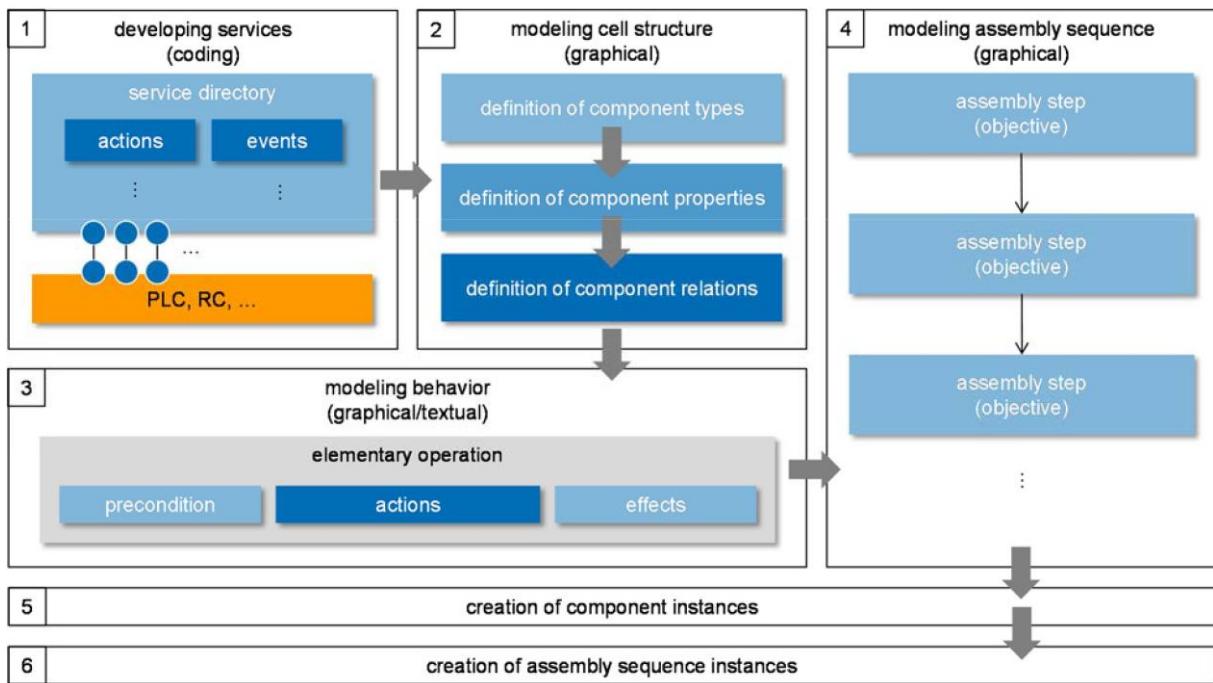


Abbildung 2-4: Systemarchitektur des Konzepts (Quelle: HERFS ET AL. 2013).

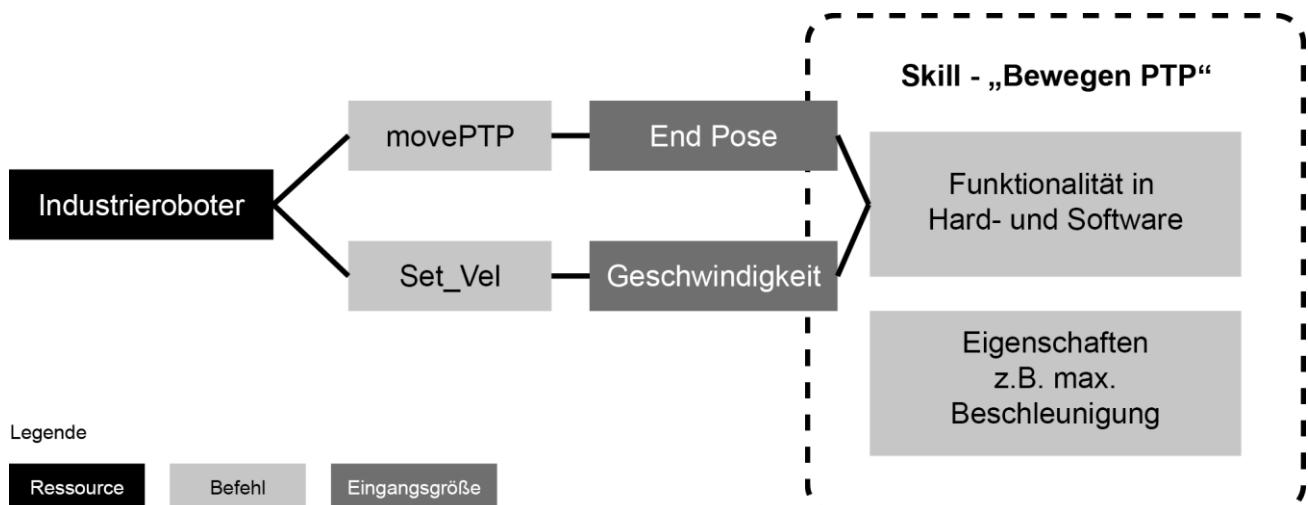
Die zum Betrieb durchzuführenden Schritte des von HERFS ET AL. (2013) vorgestellten Systems werden durch Abbildung 2-5 verdeutlicht. Im ersten Schritt müssen die Sensoren und Aktoren der Roboterzelle der jeweils übergeordneten SPS (Speicherprogrammierbare Steuerung) zugeordnet werden. Im Anschluss daran müssen das Anlagenlayout und geometrische Relationen modelliert werden. Hierzu steht ein GUI zur Verfügung. Der dritte Schritt umfasst die Beschreibung der verfügbaren Elementarfunktionen der Roboterzelle. Dazu müssen die jeweils notwendigen Vorbedingungen, die Aktion sowie die Auswirkungen der Elementarfunktion im System hinterlegt werden. Weitere Aufgabe des Anwenders ist die Modellierung der Montagereihenfolge. Hierfür steht ebenfalls ein GUI zur Verfügung. Im fünften Schritt müssen die Instanzen der Komponenten erstellt werden. Dabei müssen die Relationen zwischen den Elementen der Roboterzelle definiert werden. Die letzte durchzuführende Aktion dieser Systemarchitektur umfasst die Zuweisung der modellierten Instanzen zur Montagereihenfolge. Auf Grundlage dieser Benutzereingaben wird im Anschluss eine Roboterzelle automatisiert programmiert. Zur Validierung des Systems nutzen HERFS ET AL. (2013) eine Simulationsumgebung, in der die Montage von Slablasern modelliert wird. Hierbei wird allerdings stets dasselbe Produkt montiert.



*Abbildung 2-5: Durchzuführender Engineering-Prozess
(Quelle: HERFS ET AL. 2013).*

CAVIN ET AL. (2013) schlagen im Rahmen der Forschungsarbeit zum Themengebiet Plug and Produce ein agentenbasiertes System für Pick-And-Place-Aufgaben vor. Die Autoren konzentrieren sich auf ein Entwurfsmodell zur Beschreibung der Fähigkeiten (Skills) der Prozessteilnehmer. Diese sind von unterschiedlichem Funktionsumfang gekennzeichnet. Atomare Skills beispielsweise beinhalten elementare Grundfunktionen eines Elements. Composite Skills hingegen bestehen aus einer Anzahl atomarer Skills und können hierdurch komplexere Fähigkeiten abbilden. Die Prozessanforderungen (Skill Requirements) werden vom Programmierer nach einem vorgegebenen Schema spezifiziert (CAVIN & LOHSE 2014). Die Agenten des Systems und deren Skills werden mit den benötigten Skills Requirements abgeglichen und der Prozesskette zugewiesen. Hinsichtlich dieses Entwurfs sei angemerkt, dass die CAD-Daten nicht in Betracht gezogen werden. Darüber hinaus ist die zur Validierung herangezogene Aufgabe zu kritisieren, da hierbei Schüttgut unterschiedlicher Art in Behältnisse abgefüllt wird. Das Potential dieses Ansatzes kann anhand eines Abfüllprozesses nur schwer bewertet werden, da dies keine Montageaufgabe darstellt.

BACKHAUS (2014) beschreibt ein adaptierbares aufgabenorientiertes Programmiersystem für Montageaufgaben. Grundlage für diesen Ansatz ist die Beschreibung von Skills. Als solche sind Operationen definiert, die einer Ressource zugeordnet sind. Dies wird durch Abbildung 2-6 veranschaulicht.



*Abbildung 2-6: Beispiel für die Modellierung von Skills
(In Anlehnung an BACKHAUS & REINHART 2015).*

Die Ressource Industrieroboter kann beispielsweise mit Hilfe des Befehls *movePTP* bewegt werden. Notwendige Eingangsgröße hierfür ist die End-Pose des Roboters. Diese ist dem Skill *Bewegen PTP* zugeordnet. Da die Geschwindigkeit der Bewegung definiert sein muss, ist auch dies Eingangsgröße des Skills. Die einzelnen Skills werden in Abhängigkeit ihrer Funktion in Klassen eingeteilt, beispielsweise zur Kommunikation oder zur Verwaltung. Darüber hinaus modellieren BACKHAUS & REINHART (2015) die Architektur für eine Wissensbasis, aus der die für den Montageprozess notwendigen Informationen generiert werden können. In der realisierten Applikation müssen vom Anwender die Primär- und Sekundärprozesse festgelegt werden. Erstere beschreiben wertschöpfende Vorgänge. Sekundärprozesse sind solche, die zur Durchführung der Primärprozesse unterstützend erforderlich sind. Diese Prozesse werden mit den implementierten Skills verglichen und die Anlage dementsprechend programmiert. Auch diese Architektur verfügt über keine Schnittstelle zur Nutzbarmachung der Informationen aus den CAD-Daten. Darüber hinaus wurde der Entwurf im Hinblick auf die Robotik lediglich simulativ validiert.

Der Einsatz kognitiver Systeme zur automatischen Montageplanung und Durchführung wird von BÜSCHER ET AL. (2013) untersucht. Unter Kognition im technischen Bereich ist die Informationsverarbeitung eines künstlichen Systems zu verstehen, das vergleichbar mit dem zentralen Nervensystem eines Menschen ist. Zentrale Einheit der Softwarearchitektur ist eine kognitive Planungs- und Steuerungseinheit. Dieser Einheit werden die CAD-Daten der Baugruppe zur Verfügung gestellt. Darüber hinaus müssen Informationen durch den Anwender ergänzt werden. Mit Hilfe der Assembly-by-Disassembly-Strategie wird ein Zustandsgraph erzeugt. Dieser beschreibt alle, während der

Montage möglichen Systemzustände. Durch Kopplung der kognitiven Planungs- und Steuerungseinheit mit einer Simulationsumgebung kann der Zustandsgraph bewertet und die optimale Montagereihenfolge ermittelt werden. Aus dieser werden die notwendigen Steuerungsbefehle für den Roboter abgeleitet. Zusammenfassend stellen BÜSCHER ET AL. (2013) fest, dass sich der Einsatz kognitiver Systeme für automatische Montageplanungen gegenwärtig noch im Bereich der Grundlagenforschung bewegt, da der Transfer von kognitiven Systemen hin zu technischen Anwendungen noch nicht ausreichend vollzogen ist. Die Eignung für reale Produktionsprozesse ist daher kaum gegeben.

BENGEL (2010) verfolgt den Ansatz ontologiebasierter Datenbanken. Dazu werden die Eigenschaften aller Devices der Roboterzelle, deren Skills und Kommunikationsanbindung zu anderen Teilnehmern vom Anwender spezifiziert und in einer Datenbank gespeichert. Darüber hinaus muss das Anlagenlayout der prozessrelevanten Zellenelemente definiert werden. Die notwendigen Informationen über das zu montierende Produkt werden mit Hilfe eines Plug-Ins für ein kommerzielles CAD-Programm generiert. Dieses Plug-In hat Zugriff auf die Datenbank. Der Konstrukteur kann hiermit beispielsweise die erforderlichen Skills zur Montage einer Baugruppe im CAD-Programm zuweisen. Auf Basis dieser Informationen kann ein Montagevorranggraph erstellt werden, aus dem eine Sequenz von Basisanweisungen für die Robotersteuerung generiert werden kann. Dieser Ansatz wird durch BENGEL (2010) mit Hilfe einer Roboterzelle zur Durchführung von Pick-and-Place-Aufgaben bewertet. Die Montageaufgabe und die dazu verwendete Hardware vereinfachen den Prozess stark. Darüber hinaus ist hinsichtlich der auftretenden Produktvarianz und der daraus resultierenden Nutzung der Anlagenflexibilität keine Aussage möglich.

Auch der Ansatz von BJÖRKELUND ET AL. (2012) fußt auf einem ontologiebasierten Wissensserver. In einer als KIF (Knowledge Integration Framework) bezeichneten Einheit werden Skills und Elemente der Roboterzelle mittels Ontologien modelliert. Die Struktur der Skills basiert auf der Beschreibung der Zusammenhänge von Produkt, Ressourcen und Prozess. Grundlage für die Datenerhebung ist die in der Modellierungssprache AutomationML (Automation Markup Language) gehaltene Beschreibung der Zellenelemente (BJÖRKELUND ET AL. 2011). Diese werden in der Device Library im RDF-Format (Resource Description Framework) abgespeichert. Das KIF als zentrales Element der Softwarearchitektur ist mit einer Engineering Station verbunden. Dieses Programm dient als Systeminterface, mit Hilfe dessen die Aufgabe spezifiziert werden muss. Im

konkreten Anwendungsfall des Projekts wird als Engineering System die Off-line-Programmierumgebung ABB Robot-Studio verwendet. Durch ein Eingabemodul kann die auszuführende Aufgabe in natürlicher Sprache beschrieben werden. Im Anschluss wird die Bedeutung eines jeden Wortes mit Hilfe einer Software analysiert und maschinenlesbare Datensätze daraus abgeleitet (STENMARK & MALEC 2015). In einer als Task-Execution bezeichneten Softwareeinheit werden diese im Folgenden analysiert und ein Montagevorranggraph erstellt. Grundlage für die Analyse sind die im KIF modellierten Ontologien. Fehlende Informationen, beispielsweise über den zum Einsatz kommenden Greifer müssen vom Anwender ergänzt werden. Mit Hilfe eines nativen Controllers wird der Steuerungscode für die einzelnen physikalischen Elemente der Roboterzelle erzeugt (STENMARK ET AL. 2015). Die Positionen der zu fügenden Bauteile werden nach STENMARK ET AL. (2015) aus den CAD-Daten extrahiert.

2.3 Cyber-Physische Systeme

Im Folgenden werden einleitend die Grundlagen von CPS erläutert. Im Anschluss daran folgt ein Überblick über Forschungsprojekte von CPS im Bereich der Robotik.

2.3.1 Grundlagen

Als Cyber-Physische Systeme werden im Allgemeinen System mit erweiterter Kommunikationsfähigkeit bezeichnet (BAUERNHANSL 2014). Gemäß ACATECH (2011) können CPS als Systeme definiert werden, die:

- „über Sensoren unmittelbar physikalische Daten erfassen und durch Aktoren auf physikalische Vorgänge einwirken,
- erfasste Daten auswerten und speichern und aktiv oder reaktiv mit der physikalischen sowie der digitalen Welt interagieren,
- über digitale Kommunikationseinrichtungen untereinander sowie in globalen Netzen verbunden sind (drahtlos und / oder drahtgebunden, lokal und / oder global),
- weltweit verfügbare Daten und Dienste nutzen
- über eine Reihe dedizierter, multimodaler Mensch-Maschine-Schnittstellen verfügen.“

Nach BAUERNHANSL (2014) ist der Entwicklungsprozess Cyber-Physischer Systeme kein revolutionärer Vorgang, sondern vielmehr ein evolutionärer, stetiger

Prozess, der in vier grundlegende Stufen eingeteilt werden kann. In der ersten Stufe der Entwicklung zeichnen sich CPS durch Passivität aus. Die Geräte und Elemente sind mit Technologien zur eindeutigen Identifikation versehen. Die Systemintelligenz ist auf zentralen Einheiten implementiert. Wesentliches Unterscheidungsmerkmal der zweiten Stufe ist das aktive Verhalten der Sensoren und Aktoren. In der dritten Stufe wird das System um intelligente, netzwerkfähige Sensoren und Aktoren erweitert. Darüber hinaus wird über intelligente Schnittstellen die Kommunikation mit anderen Systemen ermöglicht. In der höchsten Stufe des Entwicklungsprozesses können die CPS-Elemente ihre Einzelfähigkeiten selbstständig, dezentral und intelligent kombinieren. Auf dieser Grundlage können neue Fähigkeiten entwickelt und autonom Dienste zur Verfügung gestellt werden. (BAUERNHANSL 2014)

Cyber-Physische Systeme bieten das Potential Produktionssysteme entwickeln zu können, die durchgängige Datenintegrität über die unterschiedlichen betriebswirtschaftlichen Ebenen eines Unternehmens anbieten können. Darüber hinaus können Wertschöpfungsnetzwerke erzeugt werden, mit deren Hilfe die Daten des Engineering-Prozesses über den Produkt-Lebenszyklus hinweg verwendet werden können. (HUBER 2013)

2.3.2 Cyber-Physische Robotik

Das Potential Cyber-Physischer Systeme kann auch auf die Robotik übertragen werden. Nach MIKUSZ & CSISZAR (2015) bieten die fünf größten Roboterhersteller mit einer Marktdeckung von 81% derzeit keine Lösung zur Implementierung Cyber-Physischer Systeme. Dem gegenüber steht das Potential, das service-orientierte Cyber-Physische Systeme mit sich bringen. Anlagen können automatisch rekonfiguriert und an neue Gegebenheiten angepasst werden. Durch hohe Vernetzungsgrade kann die Fertigung flexibilisiert und durch den Endanwender mit geringem Aufwand adaptiert werden. (MIKUSZ & CSISZAR 2015)

Das CPS-Potential in der Robotik wird durch ein von STEEGMÜLLER & ZÜRN (2014) vorgestelltes Projekt deutlich, bei dem 45 Industrieroboter kooperieren. Die Roboter sind miteinander starr vernetzt und bilden ein CPS-Netzwerk der ersten Stufe. Auf konventionelle Transportbänder und Werkstückträger wird innerhalb dieser Anlage verzichtet. Die Roboter arbeiten Hand in Hand wodurch nichtwertschöpfende Nebenzeiten minimiert werden können. Dieses Projekt zeigt die Vorteile hoher Vernetzungsgrade in der roboterbasierten Montage auf.

KIM ET AL. (2015) untersuchen ein adaptives, aufgabenorientiertes Modell für autonome Robotersysteme. Abbildung 2-7 zeigt das vorgestellte Architekturmödell. Grundlage für das Verhalten eines jeden Roboters ist ein implementiertes Zustandsmodell. Dieses kann zur Laufzeit verändert werden. So können das Zustandsverhalten einzelner Zustände und die Transitionen zwischen Zuständen abgewandelt werden. Darüber hinaus können Zustände und Transitionen bei Bedarf entfernt werden. Der Anstoß zur Rekonfiguration des Zustandsmodells resultiert aus der Kommunikation der teilnehmenden Roboter. Diese tauschen Meldungen, Anfragen und Befehle miteinander aus. Ein jeder Roboter analysiert die ausgetauschten Daten und übersetzt diese in eine mathematische Beschreibung, die bei Bedarf dem Zustandsmodell hinzugefügt wird. Das neue Modell wird mit Hilfe eines Modell-Checkers validiert. Dazu wird die text-basierte Beschreibung des Modells analysiert und mit der gegenwärtigen Konfiguration verglichen. Darüber hinaus werden Gegenbeispiele zur Soll-Konfiguration mit dem Modell abgeglichen, um unzulässiges Verhalten und Transitionen zu verhindern. Dieser Softwareentwurf dient lediglich zur automatisierten Rekonfiguration von Zustandsmodellen autonomer Robotersysteme.

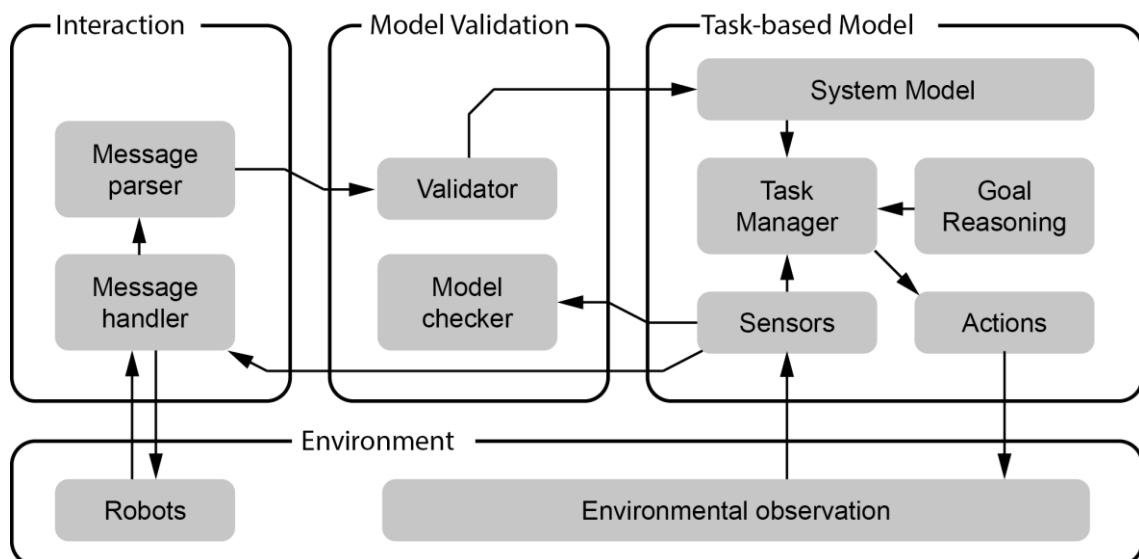


Abbildung 2-7: Softwarearchitektur eines Systems adaptiven, aufgabenorientierten CPS-Multi-Roboter-Systems (In Anlehnung an KIM ET AL. 2015).

Ein weiterer Forschungsansatz zur Cyber-Physischen Robotik ist es, eine Modellierungssprache zu schaffen, mit deren Hilfe das Verhalten einzelner CPS-Komponenten definiert werden kann (RINGERT ET AL. 2014). Jede Komponente wird mit Hilfe einer Architecture Description Language (ADL) modelliert. Diese können über definierte Schnittstellen miteinander kommunizieren. Eine genaue Beschreibung der Funktionsweise dieser Modellierungssprache kann dem Paper nicht entnommen werden.

3 Kritik am Stand der Technik

Im Rahmen dieses Kapitels werden die im Stand der Technik und Forschung vorgestellten Ansätze bewertet. Dazu wird der Zielerreichungsgrad vier herangezogener Kriterien in Tabelle 3-1 wiedergegeben. Diese sind:

1. **Extraktion der Informationen aus den CAD-Daten:** Dieses Kriterium dient zur Bewertung, inwieweit Informationen aus den CAD-Daten des Produkts zur Programmierung der Roboterzelle herangezogen werden. Anhand derer könnte beispielsweise die Montagereihenfolge sowie die zur Montage notwendige Fügebewegung abgeleitet werden.
2. **Programmierung realer Steuerungskomponenten:** In diesem Bereich wird beurteilt ob die Systemarchitektur zur Programmierung realer Steuerungskomponenten herangezogen wurde. Bei rein simulativ bewerteten Systemen wird die Komplexität der Realität zumeist nur unzureichend betrachtet, weshalb deren Praxistauglichkeit kritisch zu hinterfragen ist.
3. **Nutzung der inhärenten Anlagenflexibilität:** Der notwendige Aufwand zur Konfiguration der Anlage durch den Programmierer ist Teil dieses Betrachtungspunktes. Hierbei soll der Umfang der Benutzereingaben bewertet werden, der erforderlich ist um den Steuerungscode zu generieren, sodass ein neues Produkt automatisiert montiert werden kann.
4. **Eignung für den realen Produktionsprozess:** Dieses Kriterium dient dazu, Rückschlüsse über die Eignung der Architektur in der realen Umgebung zu ziehen. Einige Softwarearchitekturen sind für die Montage komplexer Produkte in Ziellosgröße Eins gänzlich ungeeignet, da der Softwareentwurf die Realität grundsätzlich zu stark vereinfacht. Dahingegen gibt dieses Kriterium Aufschluss darüber, ob der Systementwurf für den realen Anwendungsfall überhaupt geeignet ist.

Es fällt auf, dass nur wenige Ansätze die vorhandenen CAD-Daten zur aufgabenorientierten Programmierung der Roboterzelle heranziehen. Architekturen, die die Verwendung der CAD-Daten beinhalten, sind aufgrund stark vereinfachter Randbedingungen oftmals kaum für reale Produktionsprozesse geeignet. Einige Softwaresysteme wurden lediglich simulativ oder anhand stark vereinfachter, realer Steuerungskomponenten validiert. Deren Allgemeingültigkeit ist zu hinterfragen. Des Weiteren wird die inhärente Anlagenflexibilität oftmals nur teilweise genutzt, da durch den Anwender weiterhin hoher Aufwand betrieben werden muss, um das System an neue Gegebenheiten anzupassen.

*Tabelle 3-1: Bewertung der im Stand der Technik erwähnten Ansätze
(Quelle: Eigene Ausarbeitung).*

		Extraktion der Informationen aus den CAD-Daten	Programmierung realer Steuerungskomponenten	Nutzung der inhärenten Anlagenflexibilität	Eignung für den realen Produktionsprozess
Kategorie	Autor				
Aufgabenorientierte Ansätze	KAUFMAN ET AL. (1996)	○	○	○	○
	WEEKS (1997)	●	●	●	○
	CUIPER (2000)	○	●	●	●
	MOSEMANNE & WAHL (2001)	●	○	○	●
	LÜDEMANN-RAVIT (2005)	○○	●	○	●
	BRECHER ET AL. (2004)	○	○	●	○
	MEDELLIN ET AL. (2010)	●	●	○	○
	MAAß ET AL. (2008)	●	○	○	●
	HUCKABY & CHRISTENSEN (2012)	○○	●	○	●
	HERFS ET AL. (2013)	○○	○	●	○
	CAVIN ET AL. (2013)	○○	○	●	●
Ontologiebasierte Ansätze	BACKHAUS (2014)	○	○	●	●
	BÜSCHER ET AL. (2013)	●	○	○	○
Ontologiebasierte Ansätze	BENGEL (2010)	●	○	●	○
	BJÖRKELUND ET AL. (2012)	○	●	●	●

Anhand der vorgestellten Methoden zur Roboterprogrammierung wird deutlich, dass die inhärente Flexibilität von Robotern aufgrund der Aufgabenkomplexität gegenwärtig kaum genutzt wird. Darüber hinaus sind die Programmierzeiten zur Anpassung bestehender Programme an neue Produkte weiterhin hoch. Durch die Ansätze zur aufgabenorientierten Programmierung können diese Kosten teilweise gesenkt werden. Im Allgemeinen sind die vorgestellten Architekturen jedoch zumeist auf einen konkreten Anwendungsfall zugeschnitten oder vereinfachen die Montageaufgabe so stark, dass ein Einsatz im industriellen Umfeld irrelevant ist. Des Weiteren ist anzumerken, dass keines der vorgestellten Systeme eine automatisierte Anlagenrekonfiguration bei Ausfall einzelner Betriebsmittel beinhaltet.

Ein ganzheitlicher Ansatz zur automatisierten, aufgabenorientierten Programmierung von Robotern ist nicht erkennbar. Dieser beinhaltet die automatische Extraktion relevanter Bauteilinformationen und Produktanforderungen aus den CAD-Dateien des Produkts, die Generierung von Montagereihenfolgen und die Beschreibung von Betriebsmittelfähigkeiten. Die Informationen müssen miteinander verglichen und der Programmcode für die Betriebsmittel daraus generiert werden. Einige Systeme beinhalten Teilautomatisierungslösungen, mit deren Hilfe die Programmentwicklungszeiten verkürzt werden können. Ein gutes Beispiel hierfür ist das vorgestellte System von CUIPER (2000). Dieses beinhalten den Abgleich von Produktanforderungen und Betriebsmittelfähigkeiten sowie die automatische Erzeugung des Programmcodes. Die Produktanforderungen müssen allerdings vom Anwender in Form eines Ablaufplans spezifiziert werden. (CUIPER 2000)

Zum Einsatz von CPS in der Robotik kann festgehalten werden, dass bisher lediglich Forschungsansätze zu einigen Teilbereichen der Robotik bestehen. Diese konzentrieren sich beispielsweise auf die Schaffung einer CPS-konformen Modellierungssprache oder auf die Kommunikation von CPS-Netzwerken. Es existiert kein Ansatz für eine aufgabenorientiere Softwarearchitektur, die den Gedanken von CPS aufgreift, um Roboterprogramme auf Basis eines Anforderungs-Fähigkeiten-Abgleichs automatisiert zu erstellen.

4 Wissenschaftlicher Kontext dieser Arbeit

Diese Arbeit steht im Kontext zu einer von MICHNIEWICZ & REINHART (2015A; 2015B) vorgestellten Methode zur aufgabenorientierten Programmierung von Robotern, die auf der Verwendung Cyber-Physischer Systeme fußt. Diese soll die automatische Montageplanung und -durchführung variantenreicher Produkte ermöglichen. Dazu soll dieser Ansatz, basierend auf der lösungsneutralen Beschreibung der Produkteigenschaften, die automatische Konfigurationsauswahl, Betriebsmittelauswahl und Programmierung ermöglichen. Die Produktanforderungen sollen automatisiert aus den CAD-Daten des montierten Endprodukts extrahiert werden. Ziel dieser Architektur ist darüber hinaus, das Potential Cyber-Physischer Robotik zu veranschaulichen, da die einzelnen modularen Cyber-Physischen Betriebsmittel gemäß den Produktanforderungen rekonfiguriert werden sollen. Die Systemübersicht der Architektur ist in Abbildung 4-1 dargestellt.

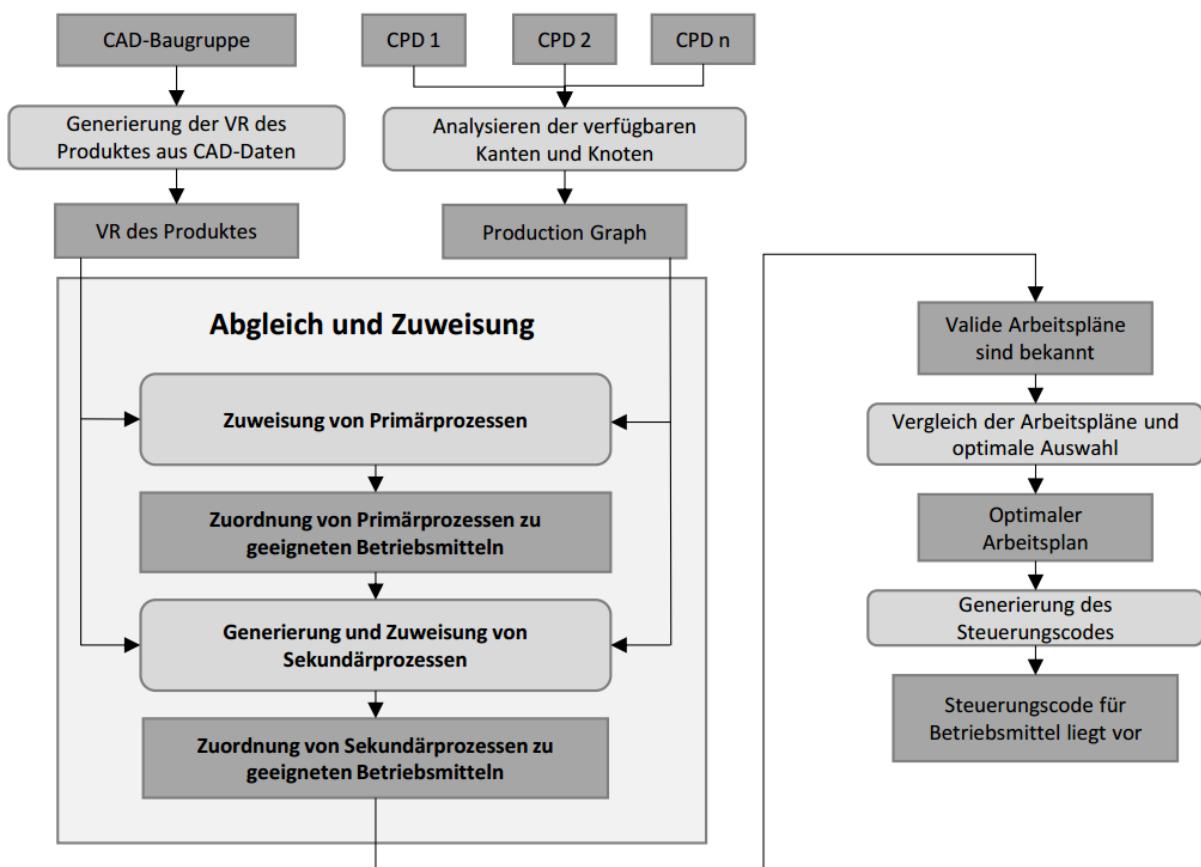


Abbildung 4-1: Systementwurf nach MICHNIEWICZ & REINHART (2015B)
(Quelle: KRAUS 2015).

Einleitend werden aus dem CAD-Modell der Baugruppe die virtuelle Repräsentanz (VR) des Produktes generiert. Diese beinhaltet relevante Bauteilinformationen sowie verschiedene valide Montagereihenfolgen. Weiterer Teil der VR ist

die lösungsneutrale Beschreibung der notwendigen Montageschritte. Diese werden mit Hilfe von sogenannten Funktionsprimitiva (FP) beschrieben. Beispielhaft seien an dieser Stelle die FPs *Halten*, *Lösen*, *Bewegen*, und *Kraft messen* genannt. Die Kombination von Produkt und dessen VR bilden das CPP (Cyber-Physisches Produkt).

Als CPD (Cyber-Physical Device) werden Kombinationen realer Betriebsmittel mit deren virtuellen Repräsentanzen bezeichnet. Letztere beschreiben die individuellen Eigenschaften, Fähigkeiten und Restriktionen der Betriebsmittel der CPRC (Cyber-Physical-Robot-Cell). Einzelne CPDs können Verbindungen eingehen und CPDCs (Cyber-Physical-Device-Combination) erzeugen. Diese bestehen aus mindestens zwei CPDs und bieten erweiterte Funktionalitäten an. Bestes Beispiel hierfür ist ein Roboter, der nur in Kombination mit einem Greifer Werkstücke handhaben und bewegen kann. Die Fähigkeiten der CPDs werden ebenfalls durch Funktionsprimitiva beschrieben. Basierend auf den VRs wird im Anschluss mit Hilfe eines Knoten-Kanten-Modells der Production Graph (PG), ein digitales Fähigkeitsmodell der Anlage erstellt. Dieser dient zur Beschreibung der Gesamtfähigkeiten der CPRC sowie der möglichen Materialflüsse im Produktionssystem.

Der nächste Schritt beinhaltet einen Abgleich der VR des Produkts mit dem PG. Hierbei werden die zur Montage der Baugruppe notwendigen Primär- und Sekundärprozesse identifiziert und überprüft, ob das Produkt durch die modellierte CPRC gefertigt werden kann. Primärprozesse beschreiben Vorgänge, die zum Vervollständigen eines Produkts und damit direkt zu Wertschöpfung beitragen. Sekundärprozesse sind Aufwendungen, die notwendig sind, aber wertschöpfungstechnisch keinen Beitrag leisten. Im Anschluss werden die Primär- und Sekundärprozesse den geeigneten Betriebsmitteln zugeordnet und valide Arbeitspläne erstellt. Dabei werden die unterschiedlichen Pfade durch das System berücksichtigt. Mit Hilfe eines Algorithmus wird der optimale Arbeitsplan ausgewählt und auf dessen Grundlage der Steuerungscode für die Anlage erzeugt.

Dieser Systementwurf soll die Potentiale von CPS in der Robotik aufzeigen. Im Gegensatz zu den in Kapitel 2 vorgestellten Systemen sollen mit diesem Entwurf die Produktanforderungen automatisch aus den CAD-Daten extrahiert werden. Darüber hinaus soll dieser Entwurf die automatische Rekonfiguration modular aufgebauter Produktionssysteme ermöglichen. Des Weiteren soll durch diesen Entwurf die inhärente Flexibilität von Robotern nutzbar gemacht werden, da auf Grundlage der virtuellen Repräsentanzen von Produkt und CPDs die Roboterzelle automatisiert programmiert werden soll.

5 Rahmenbedingungen

Grundlage für diese Arbeit ist eine bereits existierende CPRC, die das automatisierte Fügen von Lego-Steinen ermöglichen soll. Darüber hinaus ist ein SPS-Programm zur Steuerung der Anlagen-Peripherie vorhanden. Zum Designen von Lego-Baugruppen liegt ein hierfür entwickeltes CAD-Programm vor, das die Informationen, die für einen automatisierten Montageprozess notwendig sind, durch eine Schnittstelle in Form einer Textdatei zur weiteren Nutzung bereitstellt. Abbildung 5-1 zeigt die vorhandene Roboterzelle.



Abbildung 5-1: Die vorhandene CPRC (Quelle: Eigene Ausarbeitung).

Innerhalb dieses Kapitels werden die Rahmenbedingungen für die zu entwickelnden Software dargelegt. Einleitend wird in Kapitel 5.1 die vorhandene CPRC beschrieben. Dabei wird zuerst ein Überblick über das Gesamtsystem gegeben. Anschließend werden der Aufbau, die Elektrik sowie das auf der Robotersteuerung implementierte SPS-Programm erläutert. Kapitel 5.2 umfasst die Darlegung der Funktion des CAD-Programms BrickCAD sowie die aus dem Programm für die Entwicklung der Software abzuleitenden Rahmenbedingungen.

5.1 Die vorhandene CPRC

Um einen Überblick über die vorhandene CPRC aufzuzeigen, wird im Folgenden die Systemstruktur der Roboterzelle dargestellt. Siehe dazu Abbildung 5-2. Aufbauend darauf werden die einzelnen Zellenelemente, die Elektrik sowie die SPS dargelegt.

5.1.1 Überblick über das Gesamtsystem

Zentrales Bauelement der CPRC ist ein Delta-Roboter des Typs MOTOMAN MPP3 des Herstellers Yaskawa Electric Corporation. Dieser Roboter ist an eine Steuerung des Typs FS100 angeschlossen, welche die Servo-Motoren des Roboters sowie die peripheren Devices der CPRC durch eine interne speicherprogrammierbare Steuerung steuert. Betrieben wird die FS100 unter Verwendung eines Netzgeräts, das die erforderlichen 200 V bereitstellt. An die FS100 ist darüber hinaus ein Programmierhandgerät (PHG) angeschlossen, mit dessen Hilfe die Roboterzelle gesteuert werden kann. Daneben ist der Leitrechner der CPRC per Ethernet mit der Robotersteuerung verbunden.

Am Handgelenksflansch des Roboters ist ein Kraft-Momenten-Sensor (KMS) FTC-050-80 der Firma Schunk angebracht. Dieser dient dazu, die auftretenden Kräfte und Momente zu ermitteln und per CAN-Bus (Controller Area Network) zur weiteren Verwendung an den Leitrechner zu übermitteln. Am Kraft-Momenten-Sensor wiederum ist ein Greiferwechselsystem angebracht, welches den automatisierten Greiferwechsel ermöglicht. Dies ist notwendig, da die in der CPRC zu fügenden Lego-Bausteine unterschiedliche Abmessungen aufweisen, weshalb drei verschiedene Greifer zum Einsatz kommen. Sowohl Greifer als auch Greiferwechselsystem werden durch eine Ventilinsel angesteuert. Um den Greiferwechsel zu ermöglichen, ist in der CPRC ein Greiferbahnhof angebracht. Die drei Buchten des Greiferbahnhofs sind mit jeweils einem Sensor ausgestattet.

Weiterer Teil der Roboterzelle ist ein Förderband, welches dazu dient leere Werkstückträger bereitzustellen und Werkstückträger mit fertig montierten Lego-Baugruppen aus dem Gefahrenbereich des Roboters abzutransportieren. Bestandteile des Förderbands sind zwei Umsetzer sowie zwei Stopper, welche mit Näherungsschaltern ausgestattet sind. Darüber hinaus besteht das Förderband aus zwei GTS-Antrieben, denen der Transport der Werkstückträger zu kommt. Um die exakte Position der Werkstückträger bei der Montage gewährleisten zu können, werden diese durch eine Spannvorrichtung fixiert.

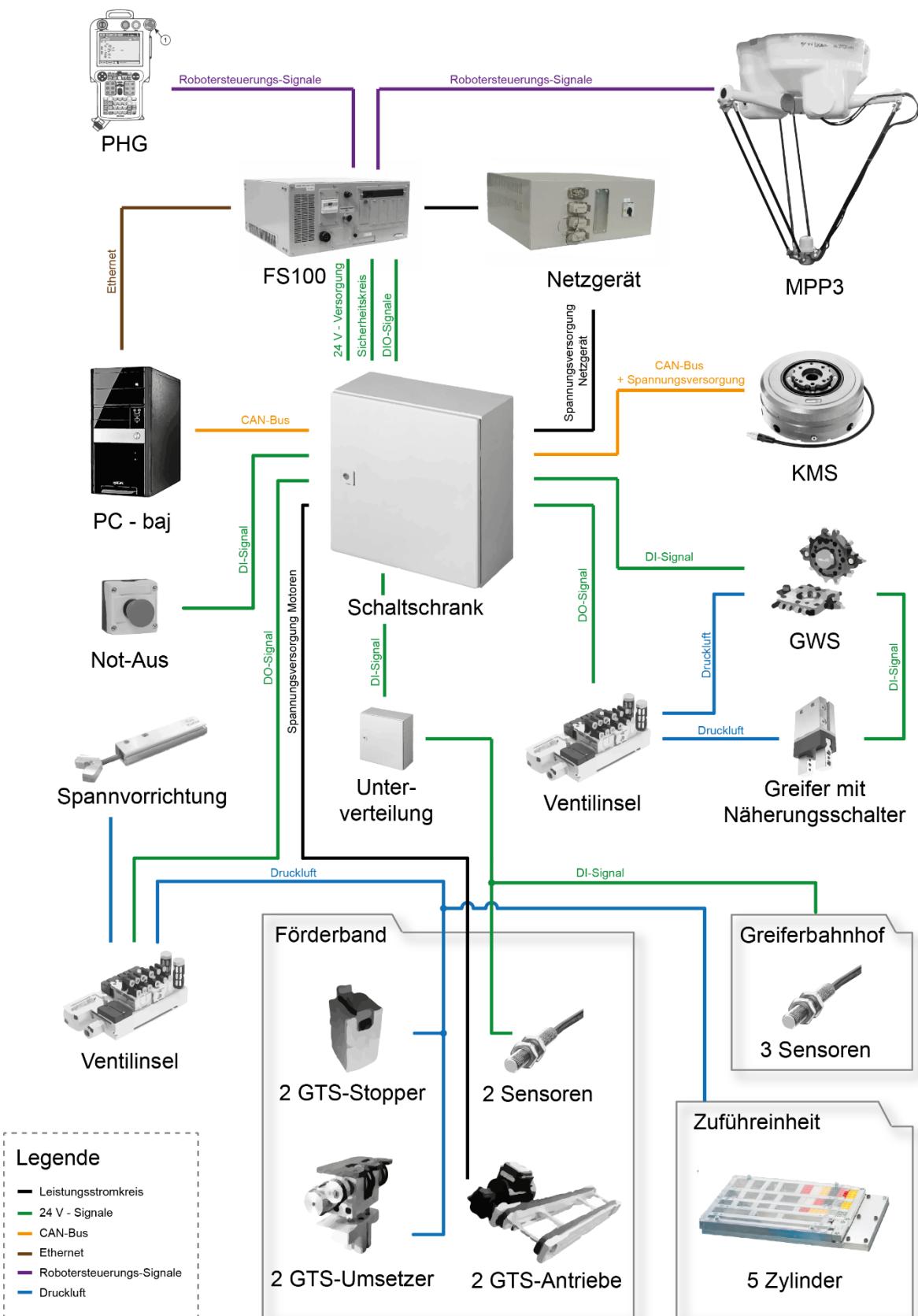


Abbildung 5-2: Systemstruktur der Roboterzelle
(Quelle: Eigene Ausarbeitung).

Durch eine Zuführeinheit werden die zu montierenden Lego-Steine bereitstellt. Diese werden bei Bedarf durch einen pneumatischen Hubzylinder ausgestoßen. Die Steuerung der pneumatischen Bauelemente des Förderbands, der Spannvorrichtung sowie der Zuführeinheit erfolgt ebenfalls durch eine Ventilinsel, die mit zehn 4/2-Wegeventilen ausgestattet ist. Die elektrische Spannungsversorgung aller Bauteile sowie die Übermittlung von elektrischen Signalen erfolgt durch einen hierfür konzipierten und aufgebauten Schaltschrank.

5.1.2 Relevante Baugruppen der CPRC

Im Folgenden werden Baugruppen der CPRC beschrieben, aus denen sich Rahmenbedingungen ergeben, die für die Entwicklung der Software zu beachten sind. Hierunter fallen die Robotersteuerung FS100, der Greiferbahnhof sowie das Zuführsystem. Für eine detaillierte Beschreibung aller weiteren Elemente der Roboterzelle sei auf SCHMITT (2014) verwiesen.

5.1.2.1 Der Motoman MPP3

Der Motoman MPP3 ist ein Delta-Roboter der Yaskawa Motoman GmbH. Roboter dieses Typs sind durch eine spezielle Parallelkinematik gekennzeichnet. Drei um 120° versetzte Armsysteme sind an einer fest montierten Basis angebracht. Diese Armsysteme können mit Hilfe von Servomotoren bewegt werden und bestehen zumeist aus Kohlefaser. Zur Identifikation sind die Arme mit den Buchstaben S, L und U gekennzeichnet.

Jeder Arm ist mit der Arbeitsplatte verbunden, die durch die kinematische Verkettung stets parallel zur Basis bewegt wird. Darüber hinaus werden rotatorische Bewegungen der Arbeitsplatte unterbunden. An dieser wiederum ist die T-Achse angebracht. Mit dieser kann der am Roboter montierbare Greifer um die Hochachse verdreht werden.

5.1.2.2 Die Robotersteuerung FS100

Die FS100 ist eine Robotersteuerung des Herstellers Yaskawa Electric Corporation mit deren Hilfe unter anderem Roboter des Typs Motoman MPP3 gesteuert werden können. Unter Einsatz eines Programmierhandgerätes kann der Roboter bewegt werden. Darüber hinaus können unter Verwendung der auf der Robotersteuerung implementierten Programmiersprache Inform II Jobs programmiert werden. Diese *Jobs* sind mit der Dateiendung .JBI versehen.

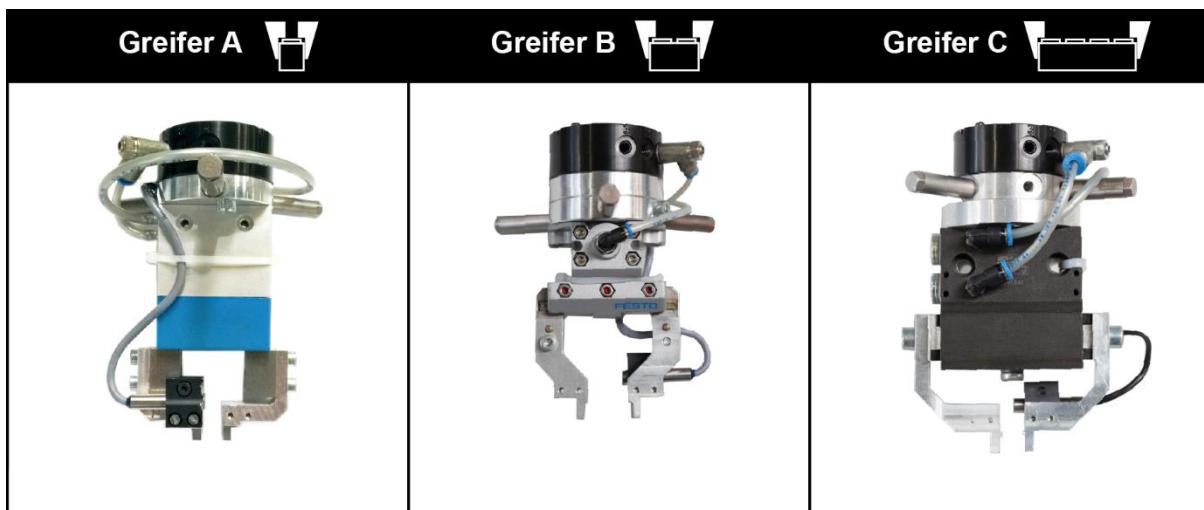
Teil der FS100 ist eine interne SPS, welche die Signale der angeschlossenen Sensoren ausliest, gemäß des implementierten SPS-Programms abarbeitet und die angeschlossenen Aktoren dementsprechend steuert. Bezuglich der SPS sei auf das hierfür eingeführte Unterkapitel 5.1.4 verwiesen.

Die Robotersteuerung basiert auf dem proprietären Echtzeitbetriebssystem VxWorks. Mit Hilfe des Software Development Kits (SDK) MotoPlusSDK können anwendungsspezifische Programme in der Programmiersprache C geschrieben werden, die nach Einschalten der Versorgungsspannung der FS100 parallel als Task des Betriebssystems ausgeführt werden. Hierdurch können beispielsweise die Kommunikation der FS100 per Ethernet, die Steuerung der SPS-IO oder der Roboterantriebe realisiert werden. (YASKAWA ELECTRIC CORPORATION 2013)

5.1.2.3 Verfügbare Greifer

In der Roboterzelle sind ein Greiferwechselsystem sowie ein Greiferbahnhof verbaut. In diesem sind drei unterschiedliche Greifern gespeichert. Tabelle 5-1 zeigt die vorhandenen Greifer.

Tabelle 5-1: Verfügbare Greifer (Quelle: Eigene Ausarbeitung).



Jeder der Greifer kann unterschiedlich breite Lego-Steine greifen. Der Greifer vom Typ A kann Lego-Steine greifen, die eine Lego-Noppe breit sind. Mit Hilfe von Typ B können Steine mit einer Breite von zwei Lego-Basiseinheiten gegriffen werden. Typ C ist dazu geeignet, Lego-Steine mit drei oder vier Noppen aufzunehmen.

5.1.2.4 Die Lego-Zuführeinheit

Die Zuführeinheit stellt die in der Roboterzelle zu handhabenden Lego-Bausteine zur Verfügung. Diese ist in Abbildung 5-3 illustriert. Lego-Steine gleichen Typs werden dabei in einem Magazin gespeichert und durch eine Feder in der vorderen Endlage gehalten. Unter der vorderen Endlage ist jeweils ein Hubzylinder angebracht, welcher dazu dient, die Lego-Steine nach oben auszustoßen. Hierdurch wird die Aufnahme der Steine durch Zweibacken-Parallelgreifer ermöglicht. Auf der Oberseite der Zuführeinheit ist eine Platte aus PMMA (Polymethylmethacrylat) angebracht. Um alle, durch die verschiedenen Greifer möglichen Greifpositionen abdecken zu können, sind in die Platte Aussparungen eingebracht. Durch diese Aussparungen sind die Positionen des Greifers bei Aufnahme eines Steins beschränkt. Diese Restriktion wird in Kapitel 11.4.4.3 nochmals aufgegriffen und durch Abbildung 11-13 verdeutlicht.

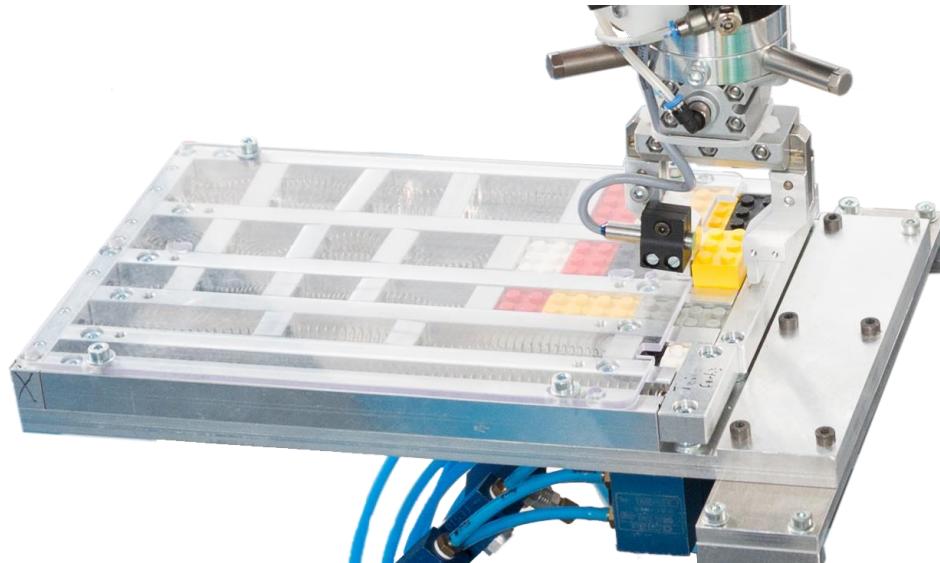


Abbildung 5-3: Zuführeinheit der CPRC mit Greifer B, der einen Lego-Stein des Typs D greift (Quelle: Eigene Ausarbeitung).

5.1.3 Beschreibung der Elektrik

Im Folgenden wird die Elektrik der CPRC beleuchtet. Hierbei werden sowohl die Sensoren als auch die Aktoren in den Vordergrund gestellt, da diese für die zu entwickelnde Software von Belang sind. Hinsichtlich des Leitungsstromteils sowie weiteren detaillierteren Ausarbeitungen zur Elektrik sei an dieser Stelle auf SCHMITT (2014) verwiesen.

5.1.3.1 Relevante Sensoren und Eingangssignale

Für die Entwicklung einer Software zum automatisierten Fügen von Lego-Steinen sind die in Tabelle 5-2 dargestellten Sensoren und Eingangssignale von Bedeutung.

Um Informationen über den Zustand des Not-Aus-Kreises sowie über die Motorschutzschalter des Förderbands gewinnen zu können, sind die Eingangssignale von Relay 20030 und 20031 der SPS relevant. Darüber hinaus geben die bereits erwähnten Näherungsschalter des Greiferbahnhofs darüber Rückmeldung, ob eine Bucht besetzt ist oder nicht. Dies ist beispielsweise vor einem Greiferwechsel von Belang, um feststellen zu können, ob der einzuwechselnde Greifer zur Verfügung steht. Des Weiteren sind die Eingangssignale der Greifer relevant. Durch diese drei Signale kann festgestellt werden, welcher der Greifer momentan eingespannt ist.

*Tabelle 5-2: Relevante Eingangssignale der CPRC
(Quelle: Eigene Ausarbeitung).*

Signalart	Klemme	Funktion	Relay
Eingang	X2.2	NOT-Aus-Kreis i.O.	20030
Eingang	X2.3	Motorschutzschalter Förderband	20031
Eingang	X2.8	Signal 1 Greiferbahnhof	20036
Eingang	X2.9	Signal 2 Greiferbahnhof	20037
Eingang	X2.11	Signal 3 Greiferbahnhof	20040
Eingang	X2.14	Greifer A Aktiv	20043
Eingang	X2.15	Greifer B Aktiv	20044
Eingang	X2.16	Greifer C Aktiv	20045

5.1.3.2 Relevante Aktoren

Im Folgenden werden die Aktoren der CPRC, die für die zu programmierenden Fügevorgänge essenziell sind, näher erläutert. Tabelle 5-3 fasst diese zusammen. Zur Durchführung eines Greiferwechsels ist das hierfür zu verwendende Ventil, welches Relay 30034 zugeordnet ist, von Bedeutung. Das Öffnen und Schließen des Greifers wird mittels Relay 30035 gesteuert. Weiterhin sind die Ventile der Zuführeinheit relevant. Unter Verwendung dieser werden die Zylinder zum Ausstoßen der einzelnen Lego-Steine aus dem Magazin gesteuert.

Eine weitere Baugruppe in der CPRC ist das Förderband. Diese, aus mehreren Aktoren bestehende Baugruppe dient dazu, leere Werkstückträger in Montageposition zu bringen und Werkstückträger mit fertigen Baugruppen aus der Roboterzelle zu transportieren. Dabei ist das Förderband als eine zusammengehörige Einheit anzusehen, da das Ansteuern eines einzelnen Umsetzers oder eines einzelnen Stoppers ohne das Zusammenspiel mit den anderen Aktoren des Förderbands als wertlos zu erachten ist. Zur Steuerung des Förderbands wurde bereits ein Programm auf der internen SPS der FS100 implementiert. Dieses Programm ermöglicht es, alle Werkstückträger um eine Position zu verschieben. Zur Steuerung der Roboterzelle reicht daher die Verwendung dieses Programms. Auf die Beschreibung der einzelnen Aktoren des Förderbands wird daher an dieser Stelle verzichtet. Stattdessen sei auf die Ausführungen zum SPS-Programm zum Steuern des Förderbands in Kapitel 5.1.4.2 verwiesen.

Tabelle 5-3: Relevante Aktoren der CPRC (Quelle: Eigene Ausarbeitung).

Signalart	Klemme	Funktion	Relay
Ausgang	X2.23	Ventil Greiferwechselsystem	30034
Ausgang	X2.24	Ventil Greifer	30035
Ausgang	X2.25	Ventil Zuführeinheit – Schacht 1	30036
Ausgang	X2.26	Ventil Zuführeinheit – Schacht 2	30037
Ausgang	X2.28	Ventil Zuführeinheit – Schacht 3	30040
Ausgang	X2.29	Ventil Zuführeinheit – Schacht 4	30041
Ausgang	X2.30	Ventil Zuführeinheit – Schacht 5	30042

5.1.4 SPS und implementierte SPS-Programme

Die Register der internen SPS der FS100 sind in logische Gruppen eingeteilt. Dieser funktionale Aufbau wird im Rahmen dieses Unterkapitels dargestellt. Weiterer Teil dieses Kapitels ist eine Beschreibung der bereits auf der SPS implementierten Programme.

5.1.4.1 Funktionaler Aufbau der SPS

Die SPS arbeitet parallel zur CPU der FS100 mit einer Zykluszeit von 1 ms (YASKAWA ELECTRIC CORPORATION 2011). Der byteweise organisierte Speicher ist in logische Gruppen unterteilt, denen unterschiedliche Aufgabenbereiche zuge-

ordnet sind. Die Speicheradressen 10010 bis 11287 sind beispielsweise für Output-Befehle innerhalb eines Jobs oder einer anwendungsspezifischen Applikation reserviert.

Das SPS-Programm der FS100 ist in die Bereiche System Ladder Section und User Ladder Section unterteilt. Innerhalb der User Ladder Section können die einzelnen Speicheradressen der SPS vom Anwender logisch miteinander verknüpft werden. In der System Ladder Section sind systemrelevante Register vom Hersteller miteinander verknüpft. Dieser Bereich der SPS ist schreibgeschützt und kann nicht beliebig verändert werden.

Zur Steuerung des Ventils zum Öffnen und Schließen des Greifers muss innerhalb eines Jobs beispielsweise die Speicheradresse 10015 angesteuert werden. Mittels Inform-Befehlssatz kann dies durch auch durch den Befehl OT(#6) zur Steuerung von Output Number 6 realisiert werden. Die Speicheradresse 10015 ist wiederum mit der Adresse 30035 der Gruppe „External Output“ verknüpft, welche das 24 V-Ausgangssignal schaltet. Eine Übersicht über den funktionalen Aufbau der SPS ist mit Abbildung 17-1 im Anhang zu finden.

Für die zu entwickelnde Software stellt dieser Aufbau insofern eine Randbedingung dar, da die in Tabelle 5-2 und Tabelle 5-3 genannten Speicheradressen nicht direkt ausgelesen oder angesteuert werden können. Zu den angegebenen Speicheradressen muss die jeweils in der User Ladder Section des SPS-Programms zugeordnete Adresse der General I/O Area ermittelt werden.

5.1.4.2 Implementierte SPS-Programme

Die auf der FS100 bereits implementierten Netzwerke umfassen die Verknüpfung der Ausgänge der logischen Gruppe *General Output* mit der Gruppe *External Output*.

Darüber hinaus ist auf der Steuerung ein SPS-Programm implementiert, mit dessen Hilfe das Förderband gesteuert werden kann. Dieses Programm wurde unter Einsatz von GRAPHCET (GRAphe Fonctionnel de Commande Etapes/Transitions), einer Spezifikationssprache für die Darstellung von Ablaufbeschreibungen, modelliert und als Schrittkette realisiert. Abbildung 5-4 zeigt den GRAPHCET zur Steuerung des Förderbands. Durch Betätigung des Tasters *Durchtakten*, der dem logischen Eingang 20041 zugeordnet ist, wird der Durchlauf der Schrittkette gestartet. Bezüglich detaillierten Ausführungen zum Ablauf der Schrittkette sei auf SCHMITT (2014) verwiesen.

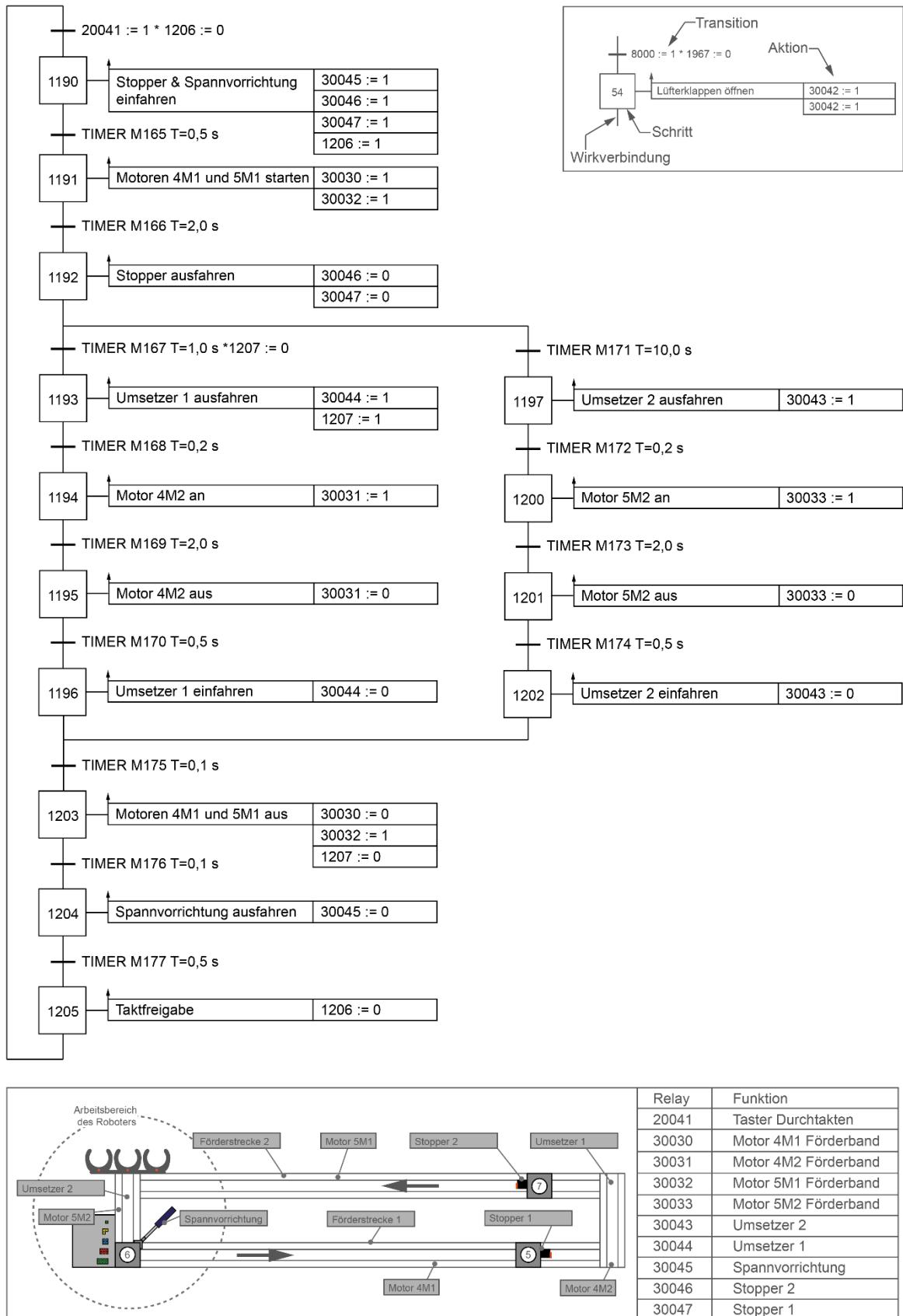


Abbildung 5-4: GRAPHCET zur Steuerung des Förderbands
(Quelle: Eigene Ausarbeitung).

5.2 Das CAD-Tool BrickCAD

Im Rahmen dieses Kapitels wird das CAD-Tool BrickCAD zur Montage von Lego-Baugruppen einleitend kurz erläutert. Im Anschluss daran ist eine Schnittstellenbeschreibung des Tools zu finden.

5.2.1 Funktionsweise

Mit Hilfe des Programms BrickCAD können auf einfache und intuitive Weise virtuelle Lego-Baugruppen erstellt werden. Dazu stehen die in der Zuführeinheit der Roboterzelle vorhandenen Lego-Steine zur Verfügung. Diese Lego-Baugruppen können im Anschluss daran dahingehend überprüft werden, ob diese durch die in Kapitel 5.1 vorgestellte Roboterzelle montierbar sind. Hierzu sind die in der Roboterzelle vorhandenen Greifer in BrickCAD modelliert. Nach Prüfung der Montierbarkeit kann der erweiterte Montagevorranggraph (eMVG) der Baugruppe erstellt werden. Dieser beinhaltet eine valide Montagereihenfolge der Lego-Steine sowie weitere prozessrelevante Informationen. Der eMVG wird in eine Text-Datei exportiert. Diese befindet sich im Verzeichnis von BrickCAD und trägt den Dateinamen build.txt. Eine exakte Schnittstellenbeschreibung zu dieser Datei findet sich im nachfolgenden Abschnitt. Abbildung 5-5 zeigt das GUI von BrickCAD.

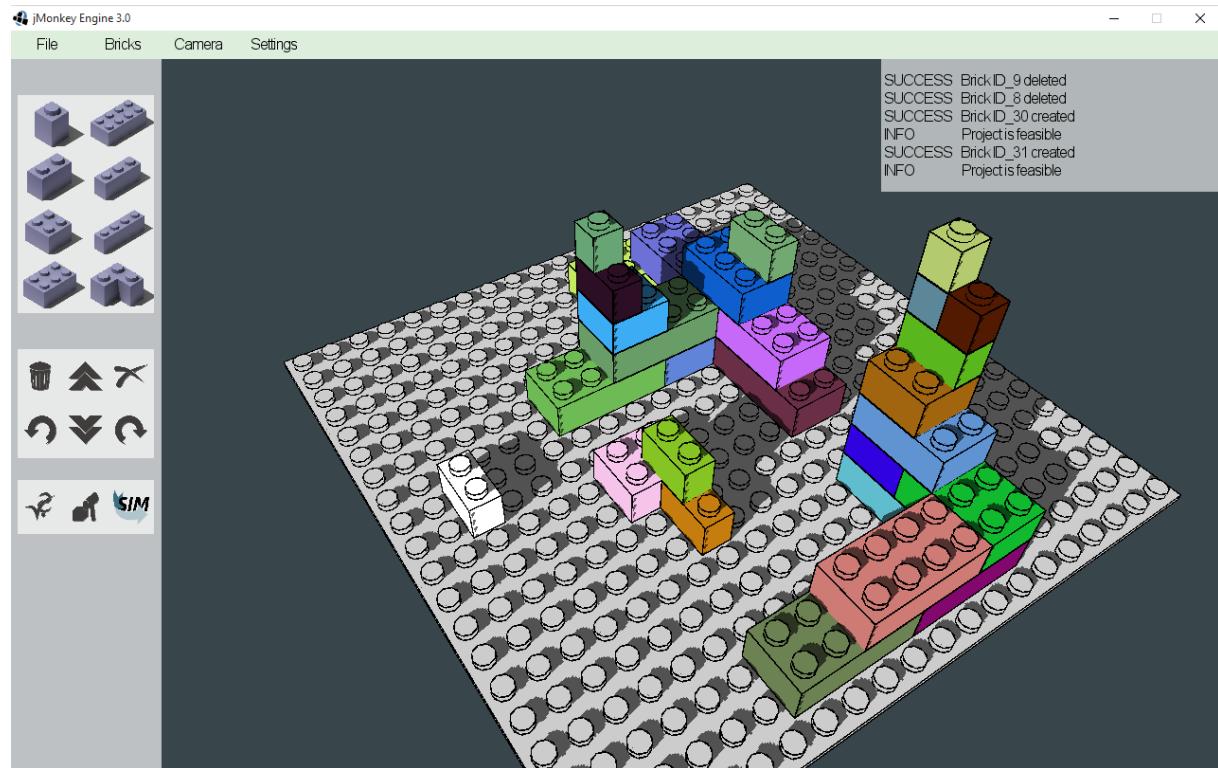
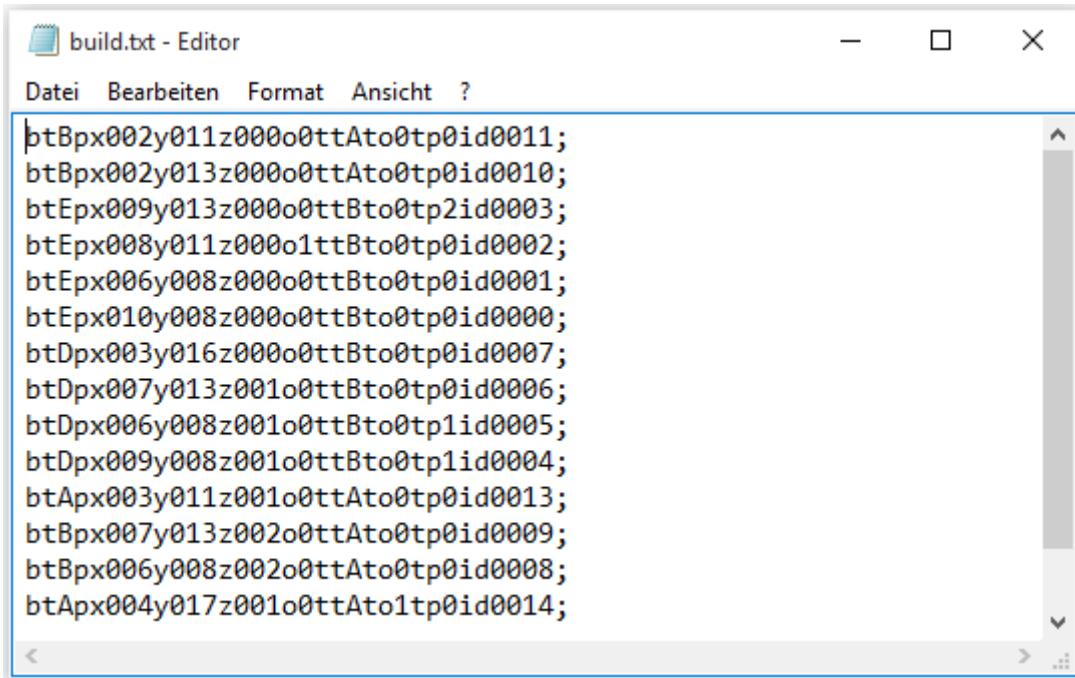


Abbildung 5-5: Das CAD-Programm BrickCAD (Quelle: Eigene Ausarbeitung).

5.2.2 Schnittstellenbeschreibung

Abbildung 5-6 zeigt den Inhalt einer mit BrickCAD erstellten Textdatei. Der Aufbau dieser Datensätze wird im nachfolgenden Unterkapitel erläutert. Zur Veranschaulich folgt im Anschluss daran ein Beispiel zum Aufbau der Datensätze



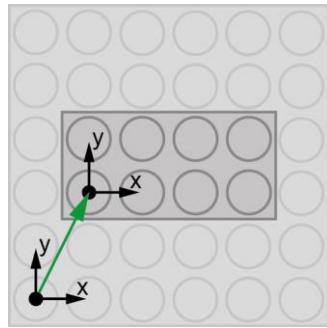
```
btBpx002y011z000o0ttAto0tp0id0011;
btBpx002y013z000o0ttAto0tp0id0010;
btEpx009y013z000o0ttBto0tp2id0003;
btEpx008y011z000o1ttBto0tp0id0002;
btEpx006y008z000o0ttBto0tp0id0001;
btEpx010y008z000o0ttBto0tp0id0000;
btDpx003y016z000o0ttBto0tp0id0007;
btDpx007y013z001o0ttBto0tp0id0006;
btDpx006y008z001o0ttBto0tp1id0005;
btDpx009y008z001o0ttBto0tp1id0004;
btApx003y011z001o0ttAto0tp0id0013;
btBpx007y013z002o0ttAto0tp0id0009;
btBpx006y008z002o0ttAto0tp0id0008;
btApx004y017z001o0ttAto1tp0id0014;
```

Abbildung 5-6: Inhalt einer build.txt-Datei (Quelle: Eigene Ausarbeitung).

5.2.2.1 Bedeutung der Parameter

Der erste Parameter, gekennzeichnet durch den Identifier *bt*, gibt die Art des Lego-Steins wieder. Aufgrund der Bereitstellung von fünf verschiedenen Lego-Steintypen durch die Zuführeinheit können dem Parameter die Werte A bis E zugewiesen werden. Eine Zuordnung, welche Bausteine welchem Buchstaben zugewiesen sind, kann aus Tabelle 5-4 abgelesen werden. Selbiges gilt für alle weiteren Parameter.

Im Anschluss an den Bricktype findet sich die Angabe der Positionskoordinaten des Lego-Steins. Die Position wird nicht in absoluten Maßangaben wiedergegeben sondern in Lego-Einheiten. Eine Lego-Einheit entspricht den Abmaßen eines Basis-Legosteins mit einer Noppe mit der quadratischen Grundfläche von 7,8 mm und einer Höhe von 9,6 mm. Ursprungspunkt des Koordinatensystems zur Positionsangabe ist der Mittelpunkt der Lego-Noppe in der linken, unteren Ecke der Lego-Montageplatte. Der Positionsvektor bezieht sich immer auf die linke untere Ecke des Lego-Steins in Grundorientierung. Abbildung 5-7 zeigt das Prinzip der Positionsangabe des Lego-Steins.



*Abbildung 5-7: Prinzip zur Angabe der Position eines Lego-Steins
(Quelle: Eigene Ausarbeitung).*

Nach der Angabe der Positionskoordinaten folgt ein Parameter, der durch den Identifier *o* gekennzeichnet ist und Auskunft über die Orientierung des Lego-Steins relativ zur Koordinatensystem gibt. Dieser Parameter kann Werte von 0 bis 3 annehmen, wobei 0 der Grundstellung entspricht. Ist dem Parameter der Wert 1 zugewiesen, so muss der Lego-Stein um 90° um die z-Achse verdreht montiert werden. *o2* entspricht einer Verdrehung von 180° um die z-Achse, *o3* einer Verdrehung von 270° .

Im Anschluss an den Parameter zur Orientierung folgt der Parameter zur Angabe des zu verwendenden Werkzeugtyps. Da in der CPRC drei verschiedene Greifer zur Verfügung stehen, folgt auf den Identifier *tt* der zugehörige Parameter, der mit den Werten A, B oder C beschrieben werden kann.

Weiterer Parameter ist die Angabe der Orientierung des Werkzeugs, da bei quadratischen Bausteinen die Werkzeugorientierung nicht aus der Orientierung des Bausteins abgeleitet werden kann. Hierzu gibt BrickCAD hinter dem Kürzel *to* die Toolorientation an. Diese kann die Zustände 0 oder 1 annehmen. Hinsichtlich der Ausrichtung des Werkzeugs sei auf Tabelle 5-4 verwiesen.

Der letzte relevante Parameter eines Datensatzes gilt der Beschreibung der Werkzeugposition. Ausgehend vom Koordinatensystem eines Lego-Steins gibt die Toolposition *tp* den Versatz des Werkzeugs vom Steinnullpunkt nach rechts an. *tp0* entspricht beispielsweise einem Versatz um 0 Lego-Noppen, *tp2* einem Versatz um 2 Lego-Noppen.

*Tabelle 5-4: Parameterzuordnung der BrickCAD-Datensätze
(Quelle: Eigene Ausarbeitung).*

Variable	Parameter	Bricks-Parameter	Symbol
Bricktype	A	btA	
	B	btB	
	C	btC	
	D	btD	
	E	btE	
Orientation	0	o0	
	1	o1	
	2	o2	
	3	o3	
Tooltype	A	ttA	
	B	ttB	
	C	ttC	
Toolorientation	0	to0	
	1	to1	
Toolposition	0	tp0	
	1	tp1	
	2	tp2	

5.2.2.2 Beispiel-Datensatz

Anhand eines Beispiels soll die Datensatz-Konvention von BrickCAD verdeutlicht werden. Abbildung 5-8 zeigt einen möglichen Datensatz.

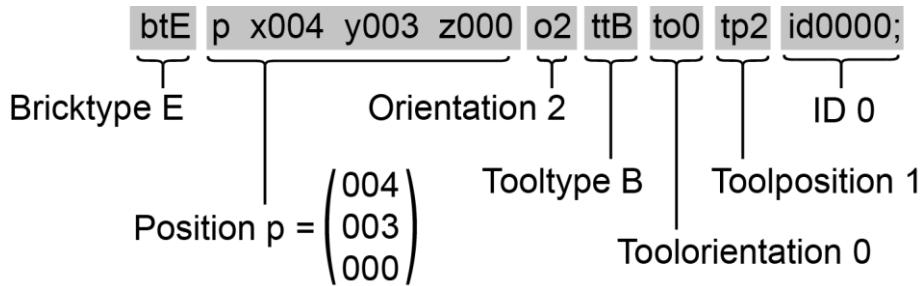


Abbildung 5-8: Beispiel-Datensatz (Quelle: Eigene Ausarbeitung).

Gemäß Tabelle 5-4 soll ein Lego-Stein des Typs E gefügt werden. Der Ortsvektor zum Ursprungspunkt des Stein-Koordinatensystems in der linken unteren Ecke des Lego-Steins in Grundstellung lautet:

$$_{iwBricks}\vec{r}_{0B} = \begin{pmatrix} 4 \\ 3 \\ 0 \end{pmatrix} \quad (1)$$

Der Lego-Stein ist um 180° um die z-Achse gedreht zu fügen. Als Fügewerkzeug ist Greifer B mit der Werkzeugorientierung 0 zu verwenden. Darüber hinaus ist der Greifer im Stein-Koordinatensystem um zwei Lego-Basiseinheiten nach rechts zu verschieben. Die aus dem Beispieldatensatz ableitbaren Informationen über den zu fügenden Lego-Stein werden in Abbildung 5-9 dargestellt. Die Position des Greifers wird durch rote Dreiecke beschrieben.

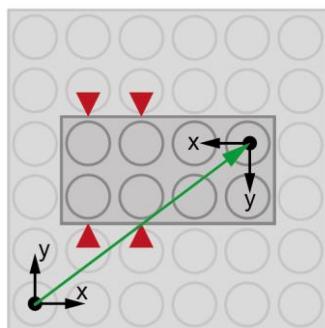


Abbildung 5-9: Darstellung der aus dem Datensatz ablesbaren Informationen. (Quelle: Eigene Ausarbeitung).

6 Zielsetzung und erwartete Vorteile dieser Arbeit

Mit Hilfe dieser Arbeit soll das Potential von CPS in der Montage demonstriert werden. Es ist eine Software-Architektur zu entwickeln, mit deren Hilfe die aufgabenorientierte Programmierung der Roboterzelle auf Basis des eMVG durchgeführt werden kann. Dazu muss ein Konzept zur Kommunikation der einzelnen Systemkomponenten erstellt und umgesetzt werden. Im Anschluss soll die entworfene Lösung implementiert werden.

Nach der Entwicklung der Software soll die CPRC in Betrieb genommen werden. Darüber hinaus sollen mit Hilfe von BrickCAD und der im Rahmen dieser Arbeit zu entwickelnden Software Lego-Baugruppen sowohl virtuell als auch real erstellt werden. Auf Basis dieser Modelle können die postulierten Vorteile von CPR nach dem Modell von MICHNIEWICZ & REINHART (2015B) validiert werden. Um gewährleisten zu können, dass die implementierte Software im Zuge nachfolgender Forschungsarbeiten weiterverwendet werden kann, soll die erstellte Software mit UML2 (Unified Modeling Language) dokumentiert werden. Des Weiteren sollen Untersuchungen zur Implementierung von CPS unter Berücksichtigung der vorliegenden Hardware durchgeführt werden.

Mit Hilfe von BrickCAD und der zu implementierenden Software kann die von MICHNIEWICZ & REINHART (2015B) vorgeschlagene Systemarchitektur validiert werden. Dieser Ansatz bietet gegenüber den bisher implementierten Architekturen zur aufgabenorientierten Programmierung den Vorteil, dass die Produktionsdaten automatisiert aus den CAD-Dateien extrahiert werden können. Des Weiteren wird erwartet, dass mit Hilfe der zu entwickelnden Software gezeigt werden kann, dass die automatische Konfigurationsauswahl sowie Rekonfiguration des Produktionssystems möglich ist. Die effektive Nutzbarmachung der inhärenten Anlagenflexibilität, basierend auf einem aufgabenorientierten Cyber-Physischen Roboterprogrammiersystem, soll veranschaulicht werden.

7 Anforderungen an die zu entwickelnde Software

Ziel dieser Arbeit ist eine Softwarelösung, mit deren Hilfe die in Kapitel 5.1 vorgestellte Roboterzelle vom Leitrechner aus gesteuert werden kann. Eingangsgröße zur Steuerung der Zelle ist dabei die durch das CAD-Tool BrickCAD erzeugte Text-Datei, die unter anderem Informationen zur Montagereihenfolge, zur Position der zu fügenden Lego-Steine und zu dem zum Einsatz kommenden Greifer beinhaltet.

Um Benutzereingaben zur Laufzeit der Softwareapplikation ermöglichen und berücksichtigen zu können, ist daneben ein Graphical User Interface notwendig. Unter Verwendung des GUI soll es dem Benutzer möglich sein, beliebige, mit BrickCAD erzeugte Build-Dateien auswählen zu können und die darin beschriebenen Lego-Baugruppen durch die Roboterzelle montieren zu lassen. Der User soll den Start des Montagevorgangs selbst bestimmen und gegebenenfalls auch abbrechen können. Darüber hinaus sollen über das GUI auch Informationen zum Systemzustand der Roboterzelle angezeigt werden.

In Anbetracht der in Kapitel 5.1 vorgestellten Hardwarearchitektur muss die Datenübertragung zwischen Leitrechner und FS100 per Ethernet erfolgen. Teil der zu entwickelnden Software muss darüber hinaus eine Funktion sein, mit deren Hilfe der am Roboter montierte Kraft-Momenten-Sensor per CAN-Bus angesteuert werden kann. Teilnehmer des CAN-Bus-Netzwerks sind der verbaute KMS sowie der Leitrechner unter Einsatz der verbauten zweikanaligen CAN-Bus-Karte. Die am Leitrechner ausgewerteten Daten müssen im Anschluss daran per Ethernet an die Robotersteuerung gesendet werden.

Um die per Ethernet an die Robotersteuerung gesendeten Daten- und Sensorsignale verarbeiten zu können, ist die Entwicklung einer auf der FS100 ablaufenden Applikation notwendig. Hierzu steht die in Kapitel 11.1 näher beschriebene Entwicklungsumgebung MotoPlusSDK zur Verfügung. In Abhängigkeit der Eingangssignale muss bei jedem zu fügenden Stein überprüft werden, ob ein Greiferwechsel notwendig ist. Im Anschluss daran muss der zu fügende Stein von der Zuführeinheit ausgestoßen, vom Roboter aufgenommen und auf dem Werkstückträger montiert werden. Diese Funktionsreihe muss so lange durchgeführt werden, bis alle Lego-Steine einer Baugruppe durch die Roboterzelle gefügt worden sind.

8 Die entwickelte Softwarearchitektur im Überblick

Bevor die Software entwickelt werden kann, muss aufgrund der Systemkomplexität ein Architekturmodell erstellt werden. Mit Hilfe dieses Modells kann die Struktur der Software und deren Zusammenspiel aufgezeigt werden. Darüber hinaus kann während des feingranularen Entwicklungsprozesses sichergestellt werden, dass stets klar ist, welche Funktion die sich gerade in Entwicklung befindende Softwarekomponente erfüllen muss.

Um ein Architekturmodell entwickeln zu können, müssen neben den in Kapitel 7 aufgezeigten Anforderungen auch die zu beachtenden Restriktionen bekannt sein. Letztere ergeben sich aus den im System verbauten Hardwarekomponenten. Im Rahmen dieses Kapitels werden daher einleitend die für die Softwareentwicklung relevanten Hardwarekomponenten behandelt. Darüber hinaus werden die Bussysteme analysiert, über die die Komponenten miteinander interagieren können. Im Anschluss daran findet sich in Abschnitt 8.2 das Architekturmodell des Softwareentwurfs. Dieses ist als Kompositionssstrukturdiagramm nach UML2-Standard ausgeführt.

8.1 Für die Entwicklung relevante Hardwarekomponenten und Bussysteme

Bei der Entwicklung der Softwarearchitektur sind Restriktionen zu beachten, die sich durch die für die Software relevanten Hardwarekomponenten und Bussysteme ergeben. Ausgehend von der in Abbildung 5-2 dargestellten Systemstruktur kann hierzu ein vereinfachtes Modell erstellt werden, um einen ersten Überblick zu erhalten. Abbildung 8-1 veranschaulicht dieses vereinfachte Modell.

Zu den für die Softwareentwicklung relevanten Hardwarekomponenten zählen der KMS, der Leitrechner sowie die Robotersteuerung FS100. In Abbildung 8-1 sind neben den Hardwarekomponenten auch die Aufgaben angedeutet, die diesen Komponenten zukommen. Diese können aus der Beschreibung der Anforderungen an die Lösung abgeleitet werden.

Der Kraft-Momenten-Sensor dient zur Erfassung der während des Fügeprozesses auftretenden Kräfte und Momente. Aufgrund der CAN-Bus-Schnittstelle des KMS werden die erhobenen Daten per CAN-Bus an den Leitrechner übertragen. Gemeinsame Aufgabe von Leitrechner und KMS ist somit die Bewältigung der CAN-Bus-Kommunikation.

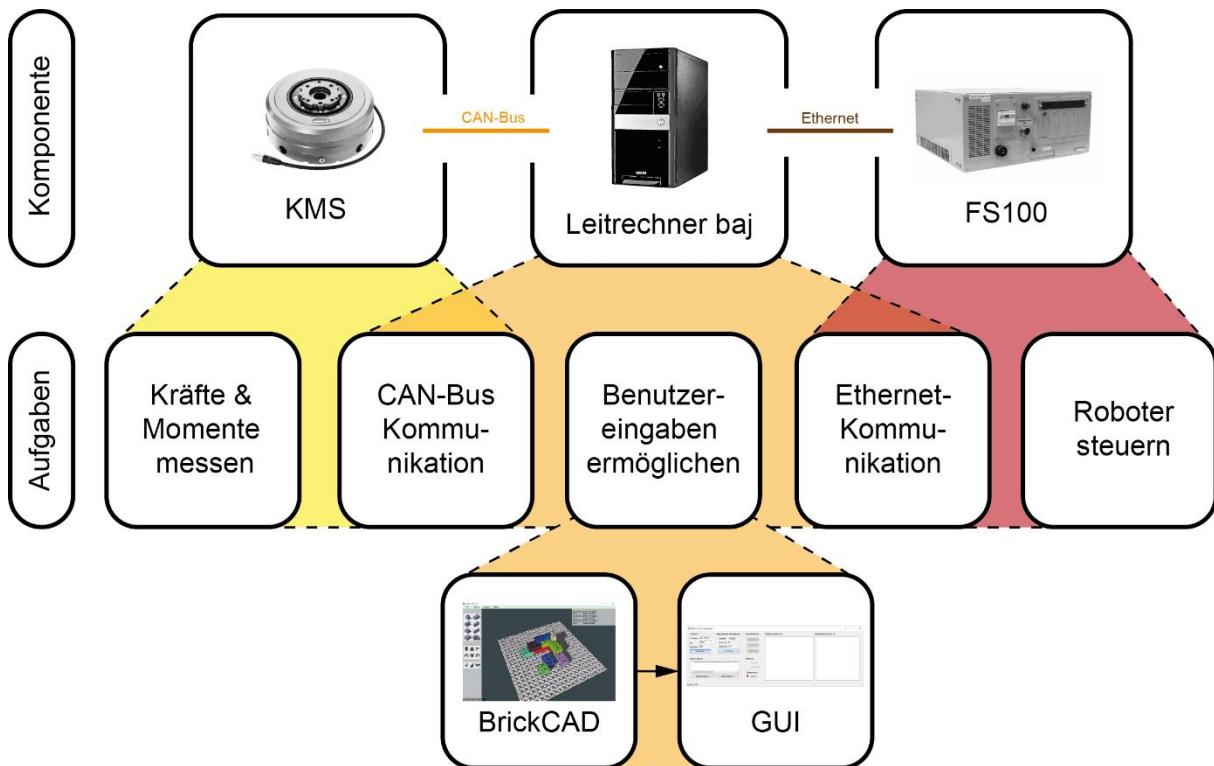


Abbildung 8-1: Vereinfachtes Modell der für den Software-Entwurf relevanten Komponenten und deren Aufgaben. (Quelle: Eigene Ausarbeitung).

Unter Verwendung des Leitrechners soll der User den Demonstrator starten können und die mit BrickCAD entwickelten Lego-Baugruppen zur Montage an die FS100 übertragen können. Um eine möglichst benutzerfreundliche Bedienung der Software sicherzustellen, soll diese ein GUI beinhalten, mit deren Hilfe Benutzereingaben per Mausklick möglich sind. Die erhobenen Steuerungsdaten sowie die Sensordaten des KMS müssen per Ethernet an die FS100 übertragen werden. Dahingehend ist die Ethernet-Kommunikation zwischen Leitrechner und FS100 eine Aufgabe, die auf beiden Hardwarekomponenten umgesetzt werden muss. Auf Basis dieser per Ethernet versendeten Daten soll die Roboterzelle gesteuert werden.

Jede der Hardwarekomponenten ist durch inhärente Restriktionen hinsichtlich der Programmierung gekennzeichnet. Im Folgenden werden diese für jede Komponente erläutert.

8.1.1 Rahmenbedingungen durch die FS100

Auf der FS100 ist das proprietäre Echtzeitbetriebssystem VxWorks installiert. Hierdurch ist sichergestellt, dass die Anforderungen an die Rechtzeitigkeit der Verarbeitung erfüllt werden. Zur Entwicklung von Applikationen für die FS100

steht die Entwicklungsumgebung MotoPlusSDK des Herstellers Yaskawa Motoman zur Verfügung. Diese wird, wie bereits erwähnt, in Kapitel 11.1 ausführlich behandelt. Die Programme sind in der Programmiersprache C zu entwickeln. Einschränkungen hinsichtlich der Programmierung ergeben sich durch die vom MotoPlusSDK vorgegebenen Funktionen zur Steuerung der Hardware. Es liegen verschiedene Funktionen beispielsweise zur Speicherallokation oder zur Kommunikation via TCP/IP (Transmission Control Protocol / Internet Protocol) oder UDP (User Datagram Protocol) vor. Darüber hinaus können Multitask-Prozesse programmiert werden, so dass mehrere Aufgaben gleichzeitig abgearbeitet werden können. Die IP-Adresse der FS100 lautet 192.168.0.2. Die Subnetzmaske lautet 255.255.255.0, dem Standartgateway ist Adresse 0.0.0.0 zugewiesen.

8.1.2 Der Leitrechner

Betriebssystem des Leitrechners „baj“ ist Windows 7. Somit können keine harten Echtzeitbedingungen garantiert werden. Es ist nicht sichergestellt, dass eine Aufgabe in einer festgelegten Zeitspanne abgearbeitet wird. Hinsichtlich aller weiteren Softwareeigenschaften müssen seitens des Betriebssystems nahezu keine Restriktionen beachtet werden. Die zu programmierende Software kann in einer Vielzahl höherer Programmiersprachen erstellt werden. In Anbetracht des zu erstellenden GUI erscheinen C++/CLI, C# oder Visual Basic.NET als besonders geeignete Programmiersprachen. Dem PC ist die IP-Adresse 192.168.0.3 zugeordnet. Die Subnetzmaske lautet ebenfalls 255.255.255.0, das Standartgateway 0.0.0.0.

Im Leitrechner verbaut ist eine CAN-Bus-Karte CAN-AC2-PCI, hergestellt von der Softing Industrial Automation GmbH. Mit Hilfe dieser Schnittstelle kann die CAN-Bus-Kommunikation mit Transferraten im Bereich zwischen 10 und 1000 kBit/s ermöglicht werden (SOFTING INDUSTRIAL AUTOMATION GMBH 2012A). Zur Programmierung dieser CAN-Bus-Karte stellt die Softing Industrial Automation GmbH ein API zur Verfügung. Dieses ist in zwei DLL-Dateien (Dynamic Link Library) gegliedert. Der eine Teil dient zur Implementierung eines Programms in der Programmiersprache C. Der andere Teil des API basiert auf dem objektorientierten Programmierdogma und kann mittels der Programmiersprachen C# sowie Visual Basic verwendet werden. Darüber hinaus existiert ein in der Programmiersprache C gehaltenes Beispielprojekt für Visual Studio, in Rahmen dessen ein Bus-System vollständig initialisiert wird. Anschließend können per Tastatureingabe verschiedene Funktionen angesteuert werden. Aufgrund der Komplexität der Schnittstelle ist dieses Beispielprojekt als Grundlage für die zu

implementierende CAN-Bus-Kommunikation zu empfehlen. Die Dokumentation des API ist durch SOFTING INDUSTRIAL AUTOMATION GMBH (2012B) gegeben.

8.1.3 Der Kraft-Momenten-Sensor

Der Kraft-Momenten-Sensor ist eine reaktive Komponente, da die Übertragung der Messdaten keineswegs automatisch erfolgt, sondern stets per Anfrage durch einen anderen CAN-Bus-Teilnehmer veranlasst werden muss. Das auf dem KMS implementierte Funktionsspektrum kann grundsätzlich in vier Bereiche untergliedert werden. Nach Eingang eines **Daten-Befehls** sendet der KMS die angeforderten Daten im Hexadezimalsystem per CAN-Bus an den anfragenden Teilnehmer. Rückgabewerte von **Informations-Befehlen** sind Informationen über den Zustand des Sensors. Mit Hilfe von **Einstellungs-Befehlen** können Einstellungen auf dem Sensor vorgenommen werden, beispielsweise zur Baudrate oder zur CAN-ID des Sensors, die als Adresse zur Ansteuerung des Sensors dient. Weitere, nicht kategorisierbare Sensor-Befehle sind im Bereich **Sonstige Befehle** zusammengefasst. Bezuglich den zur Verfügung stehenden Befehlen sei an dieser Stelle auf SCHUNK GMBH & Co. KG (2009) verwiesen. Dem am MPP3 verbauten KMS ist die CAN-ID 5 zugewiesen. Die Baudrate ist auf 1000 kBit/s eingestellt.

8.1.4 Kommunikation via Ethernet

Das von der ISO (International Standardization Organisation) entwickelte OSI-Referenzmodell (Open Systems Interconnection) beschreibt sieben abgeschlossenen Schichten, auf deren Grundlage Computernetzwerke basieren. Tabelle 8-1 beschreibt diese Schichten. (SCHREINER 2009)

Wie in Kapitel 8.1.1 erwähnt, bietet MotoPlusSDK Funktionen zur Kommunikation unter Verwendung der Netzwerkprotokolle TCP und UDP. Beide Protokolle sind Teil der Transportschicht und basieren auf dem Client-Server-Modell.

Letzteres ist ein Konzept zur Verteilung von Aufgaben innerhalb eines Netzwerkes. Der Server ist ein Netzwerkteilnehmer, der Services oder Daten zur Verfügung stellt. Der Client nimmt Services des Servers in Anspruch. Die Kommunikation zwischen beiden Teilnehmern erfolgt immer in der festgelegten Reihenfolge, dass der Client zuerst eine Anforderung sendet. Auf diese Anforderung antwortet der Server mit einer Rückantwort. (BENGEL 2004)

Tabelle 8-1: Schichten des OSI-Modells (In Anlehnung an SCHREINER 2009).

Layer	Bezeichnung
Layer VII	Anwendungsschicht
Layer VI	Darstellungsschicht
Layer V	Kommunikationsschicht
Layer IV	Transportschicht
Layer III	Vermittlungsschicht
Layer II	Sicherungsschicht
Layer I	Physikalische Schicht

8.1.4.1TCP-IP

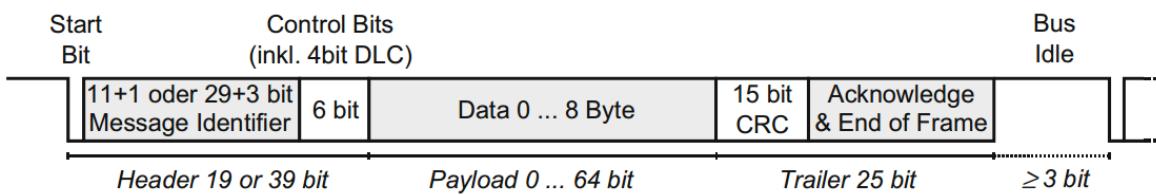
Das Transmission Control Protocol ermöglicht den fehlerfreien Datentransport via Ethernet. Da die Datenpakete eines Datenstroms während der Übertragung unterwegs unter Umständen verändert werden können, ist TCP die Kontrollinstanz, unter deren Verwendung die Datenpakete wieder korrekt zusammengezettet werden. Darüber hinaus ist mittels TCP Multiplexing möglich, so dass über eine Verbindung mehrere Applikationen gleichzeitig kommunizieren können. Einleitend wird bei jeder TCP-Verbindung zunächst eine virtuelle Verbindung zwischen Sender und Empfänger aufgebaut. Erst wenn der Kommunikationspartner erreicht werden kann, können auch Daten übertragen werden. (SCHREINER 2009)

8.1.4.2UDP

Kennzeichen des User Datagram Protocols ist die fehlende Überprüfung der vom Empfänger empfangenen Daten. Im Gegensatz zum TCP-IP-Protokoll werden Daten versendet und vom Zielport empfangen. Auf die Überprüfung der versendeten Daten durch das Netzwerkprotokoll wird verzichtet. Dies kann angewendet werden, wenn der Empfänger in der Lage ist, die empfangenen Daten selbst zu kontrollieren oder wenn der Datenverlust keine gravierenden Auswirkungen hat. Unnötiger Overhead kann dadurch reduziert werden. (SCHREINER 2009)

8.1.5 CAN-Bus

CAN-Bus ist das gegenwärtig am häufigsten eingesetzte Bus-System im Kfz-Bereich. Das Botschaftsformat von CAN-Bus-Nachrichten ist durch den, in Abbildung 8-2 veranschaulichten Aufbau geprägt. Ein Identifier dient zur Adressierung des Bus-Teilnehmers, an den die Nachricht gerichtet ist. Um eine größere Anzahl von Teilnehmern adressieren zu können, wurde der Identifier mit der zweiten Generation von CAN-Bus von 11 Bit auf 29 Bit abwärtskompatibel erweitert. Anhand des Identifiers kann jeder Bus-Teilnehmer entscheiden, ob die Nachricht weiterverarbeitet oder ignoriert wird. Der Aufbau einer Verbindung wie bei TCP-IP findet nicht statt. Die eigentliche Botschaft einer CAN-Nachricht steckt in den Nutzdatenbytes (Payload). In Abhängigkeit der Nachrichtenlänge können zwischen 0 und 8 Nutzdatenbytes versendet werden. Die Anzahl der Nutzdatenbytes wird durch das Feld DLC (Data Length Code) wiedergeben. Zur Fehlererkennung beinhaltet eine CAN-Nachricht darüber hinaus eins 15 Bit lange Prüfsumme CRC (Cyclic Redundancy Check). (ZIMMERMANN & SCHMIDGALL 2014)



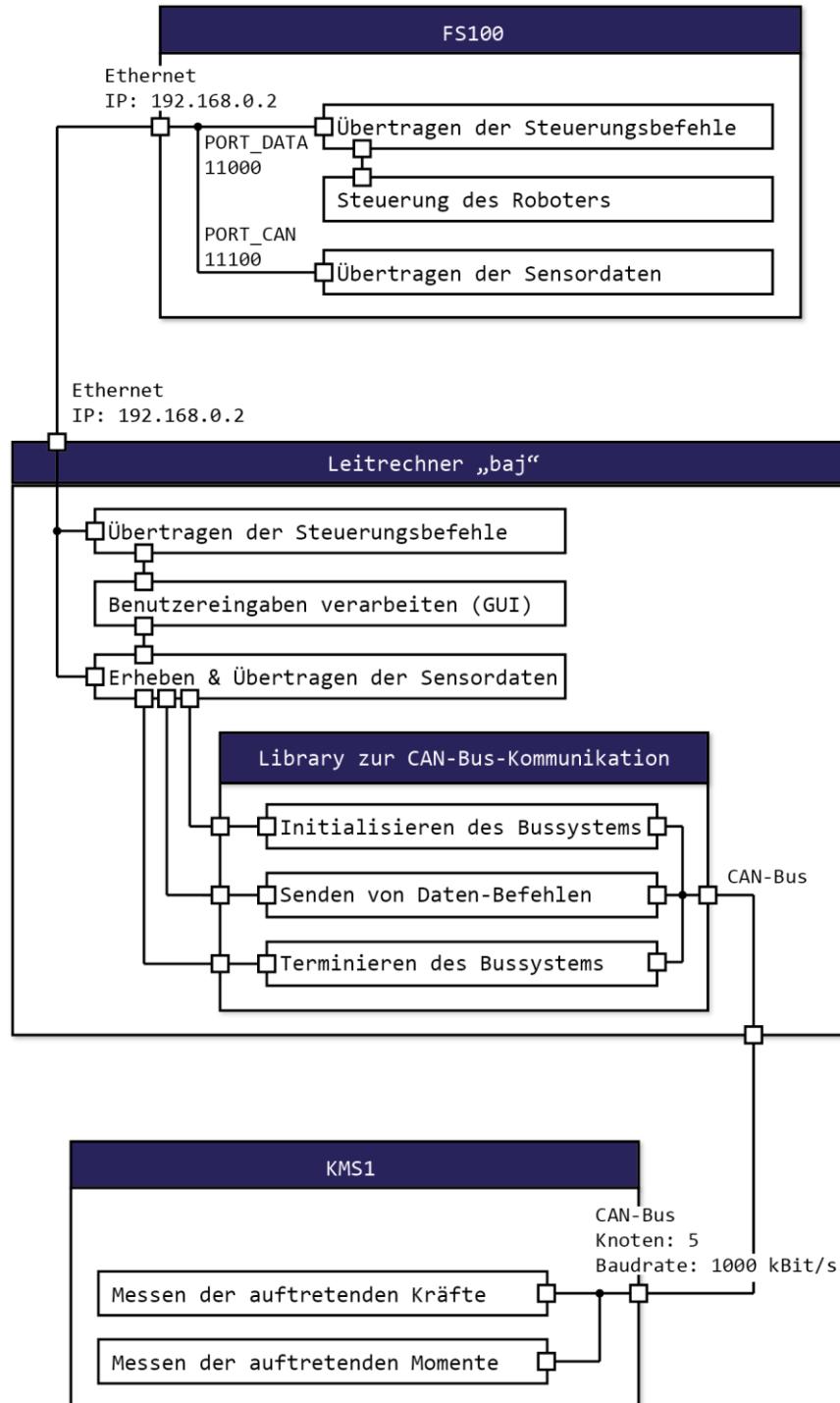
*Abbildung 8-2: Aufbau von CAN-Bus-Nachrichten
(Quelle: ZIMMERMANN & SCHMIDGALL 2014).*

8.2 Architekturmodell des Softwareentwurfs

Ausgehend von den bereits aufgezeigten Restriktionen und unter Berücksichtigung der für die Softwareentwicklung relevanten Hardware-Komponenten wurde im Rahmen dieser Arbeit das in Abbildung 8-3 gezeigte, als Kompositionsdigramm ausgeführte Architekturmodell entwickelt. Die implementierte Software basiert auf diesem Modell.

Aus Abbildung 8-1 geht bereits hervor, dass die Software als verteiltes System auszuführen ist. Eine Komponente der Software muss als Steuerungssystem auf den Leitrechner ablaufen. Teil dieses Programms ist die Berücksichtigung von Benutzereingaben per GUI und die Übertragung der Steuerungsbefehle per Ethernet. Weiterer Bestandteil ist die CAN-Bus-Kommunikation und Übertragung der Messdaten an die FS100 per Ethernet. Diese drei Prozesse müssen

zeitgleich auf dem Leitrechner ausgeführt werden und sind daher jeweils in einem eigenen Thread zu behandeln. Der Thread zur Behandlung von Benutzeingaben ist mit dem Thread zur Übertragung der Steuerungsbefehle verknüpft. Da per GUI die Datenerhebung der Kräfte und Momente zu- oder abgeschaltet werden soll, steht der Thread zur CAN-Bus-Kommunikation ebenfalls mit dem GUI-Thread in Verbindung.



*Abbildung 8-3: Kompositionsstrukturdiagramm des Softwareentwurfs
(Quelle: Eigene Ausarbeitung).*

Hinsichtlich der CAN-Bus-Kommunikation ist dem Programm eine eigene, in der Programmiersprache C geschriebene Library hinzuzufügen, auf Basis derer das API (Application Program Interface) der am PC verbauten CAN-Bus-Karte verwendet werden kann. Diese Library muss Funktionen beinhalten, mit denen das Bus-System initialisiert, Daten-Befehle an den KMS ausgesendet und die Antworten ausgewertet werden können. Darüber hinaus müssen geöffnete CAN-Bus-Kanäle nach Beendigung der Kommunikation wieder terminiert werden können.

Für die Steuerung des Roboters ist eine auf der FS100 ablaufende Applikation zu entwickeln, die die vom Leitrechner per Ethernet versendeten Befehle empfängt. Darüber hinaus müssen die empfangenen Befehle verarbeitet werden. Hierzu muss ein parallel dazu arbeitenden Task zur Steuerung des Roboters programmiert werden. Elemente dieses Tasks sind beispielsweise die Analyse eines BrickCAD-Datenstrings, die Durchführung von Initialisierungsabläufen oder von Montagevorgängen. Da die FS100 während den geschilderten Prozessen auch auf Benutzereingaben reagieren soll, ist Multitasking unabdingbar. Andernfalls könnten neue, per Ethernet versendete Befehle immer erst verarbeitet werden, nachdem beispielsweise der komplette Montagevorgang beendet wurde. Der dritte Task der Applikation für die FS100 behandelt den Empfang der Messdaten des KMS. Diese werden zwar auch per Ethernet übertragen, allerdings über Port 11100 an die FS100 übergeben.

Die Kommunikation zwischen Leitrechner und FS100 kann als Client-Server-Modell implementiert werden. Da zwischen beiden Teilnehmern eine sichere Datenübertragung gewährleistet sein muss, kommt nur TCP-IP als Netzwerkprotokoll in Frage. Bereits durch Vertauschen zweier Bytes im Koordinatenbereich eines BrickCAD-Datenstrings könnte zur Kollision von Roboter und bereits platzierten Lego-Steinen führen. Die korrekte Übertragung muss daher sicher gestellt sein, weshalb UDP nicht verwendet werden kann.

Als Client dient der Leitrechner. Server ist die FS100. Hierbei ist anzumerken, dass nicht immer scharf zwischen Client und Server getrennt werden kann. Element der GUI soll beispielsweise auch die Rückmeldung des Status der FS100 sein. Wird beispielsweise der Not-Aus betätigt, so soll dies dem Benutzer angezeigt werden. In diesem Fall müsste der Server allerdings als aktiver Kommunikationspartner eintreten, was für die TCP-IP-Programmierung nicht vorgesehen ist und auch nicht implementiert werden kann. Um diesem Mangel entgegenzutreten, muss der Client zyklisch Anforderungen an den Server stellen und die entsprechenden Rückantworten verarbeiten.

9 Notwendige Anpassungen des Gesamtsystems

Um die in Kapitel 8 vorgestellte Softwarearchitektur umsetzen und den autonomen Montageprozess frei designbarer Lego-Modelle ermöglichen zu können, wurden vor Entwicklung der Software einige Veränderungen an der bestehenden Hard- und Software vorgenommen. Erst diese Anpassungen ermöglichen beispielsweise die Nutzung der von SCHMITT (2014) implementierten SPS-Programme durch mit MotoPlusSDK entwickelte Applikationen oder tragen zur vereinfachten Steuerung des Roboters aufgrund zweckmäßig geteachter Benutzerkoordinatensysteme bei.

9.1 Anpassung des SPS-Programms

Die bereits auf der FS100 implementierte Schrittkette zur Steuerung des Förderbands ist Teil der im Rahmen dieser Arbeit entwickelten Software. Unter Verwendung der Schrittkette kann ein leerer Werkstückträger in den Arbeitsraum des Roboters bewegt werden. Nach erfolgreicher Montage kann der Werkstückträger, auf dem die gefügte Lego-Baugruppe montiert ist, aus dem Gefahrenbereich des Roboters bewegt werden.

Die SPS ist in der Programmiersprache KOP zu programmieren. Diese ist nach IEC 61131-3 eine genormte graphische Programmiersprache, die an die Darstellung in Stromlaufplänen erinnert. In Reihe geschaltete Elemente entsprechen einer Und-Verknüpfung, parallel geschaltete Elemente einer Oder-Verknüpfung. (JOHN & TIEGELKAMP 2000)

Um die Schrittkette mit Hilfe des Programmierhandgeräts (PHG) starten zu können, ist Schließer kontakt 20041 im Netzwerk der Startmarke 1190 parallel zu Kontakt 10030. Per PHG können grundsätzlich nur Register der General I/O-Area manipuliert werden. Kontakt 10030 ist Teil der General I/O-Area und ab Werk Ausgang 30050 zugeordnet. Da aufgrund nicht eingebauter zusätzlicher I/O-Gruppen dieser Ausgang ungenutzt bleibt, kann Kontakt 10030 als zusätzliche Starttransition für die Schrittkette verwendet werden. Somit kann die Schrittkette zum Durchtakten des Förderbands durch den mit Kontakt 10030 verknüpften Inform-Befehl OT(#17) gestartet werden.

Wie in Abbildung 9-1 zu sehen, ist neben Kontakt 10030 auch Kontakt 10567 im Netzwerk der Startmarke 1190 verschalten. Dieser Kontakt, der ebenfalls Teil der General I/O-Area ist, dient dazu die Schrittkette aus der mit MotoPlusSDK entwickelten Applikation starten zu können. Abbildung 9-1 zeigt das geänderte Netzwerk. Alle weiteren Netzwerke des SPS-Programms wurden im

Rahmen dieser Arbeit nicht verändert. Für eine ausführliche Dokumentation zum implementierten SPS-Programm sei an dieser Stelle auf SCHMITT (2014) verwiesen.

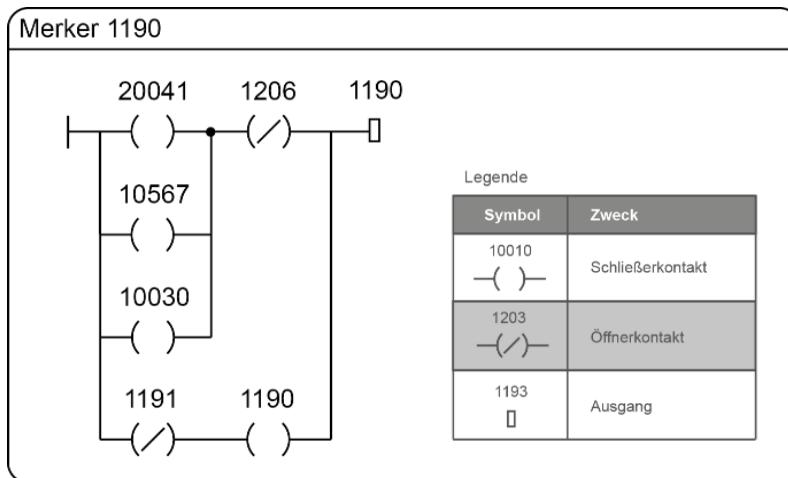


Abbildung 9-1: Das geänderte Netzwerk zu Merker 1190, der Startmerker der modellierten Schrittstrecke ist. (Quelle: Eigene Ausarbeitung).

9.2 Adaption der Pneumatik

Die Pneumatikverschaltung von Greifer und Greiferwechselsystem wurde im Rahmen dieser Arbeit dahingehend verändert, dass ein Greiferwechsel möglich ist, ohne dass Druckluft bei nicht angeschlossenem Greifer ungehindert über das Greiferwechselsystem ausströmen kann.

In der Ausgangssituation war das System so verschalten, dass mit Hilfe von Ventil 19V1-CH1 ein Greiferwechsel durchgeführt werden konnte. Am Greiferwechselsystem angeschlossen war das Ventil 19V1-CH2 zur Steuerung des Greifers. Sobald der Greifer vom Greiferwechselsystem abgenommen wurde, konnte Druckluft unabhängig von der Schaltstellung des Ventils 19V1-CH2 über die Anschlüsse des Greiferwechselsystems ausströmen, da kein Aktor mehr angeschlossen war und das Ventil 19V1-CH2 keine Sperrmittelstellung besitzt.

Um diesem Mangel entgegenzuwirken und Werkzeugwechselvorgänge ohne unnötige Lärmbelästigung zu ermöglichen, sind nun zwei Ventile zwischen Ventil 19V1-CH2 und Greiferwechselsystem geschalten. Die Ansteuerung beider Ventile erfolgt pneumatisch, sobald Ventil 19V1-CH1 zur Durchführung eines Greiferwechsels in Schaltstellung 1-4 schaltet. Hierbei werden beide Arbeitsleitungen, welche von Ventil 19V1-CH2 zum Greiferwechselsystem führen, gesperrt. Somit kann keine Luft mehr entweichen, solange sich Ventil 19V1-CH1 in Schaltstellung 1-4 befindet. Sobald das Ventil in Schaltstellung 1-2 schaltet,

werden die zwischengeschalteten Ventile durch die Federrückstellung wieder in Ruhestellung gebracht. Der am Greiferwechselsystem angebrachte Greifer kann somit wieder über Ventil 19V1-CH2 geschlossen und geöffnet werden. Abbildung 9-2 veranschaulicht die Verschaltung der Pneumatik.

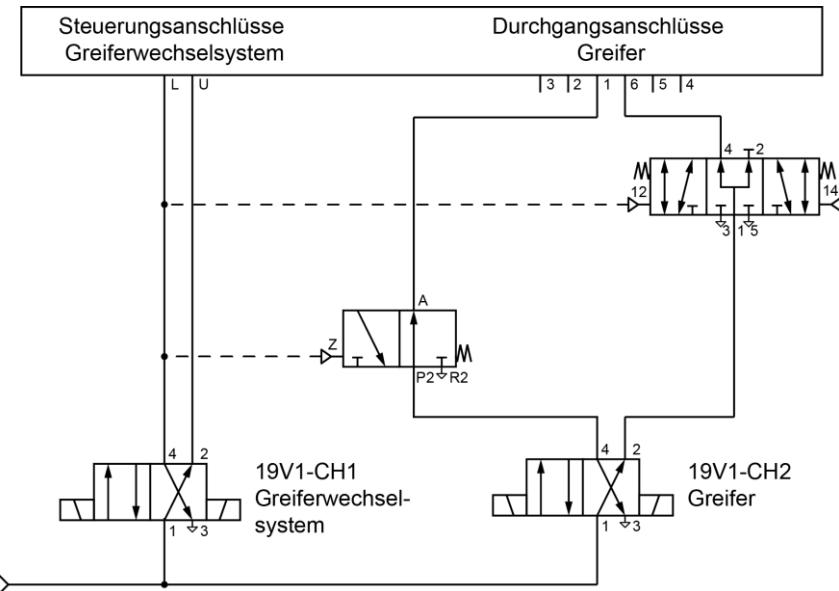
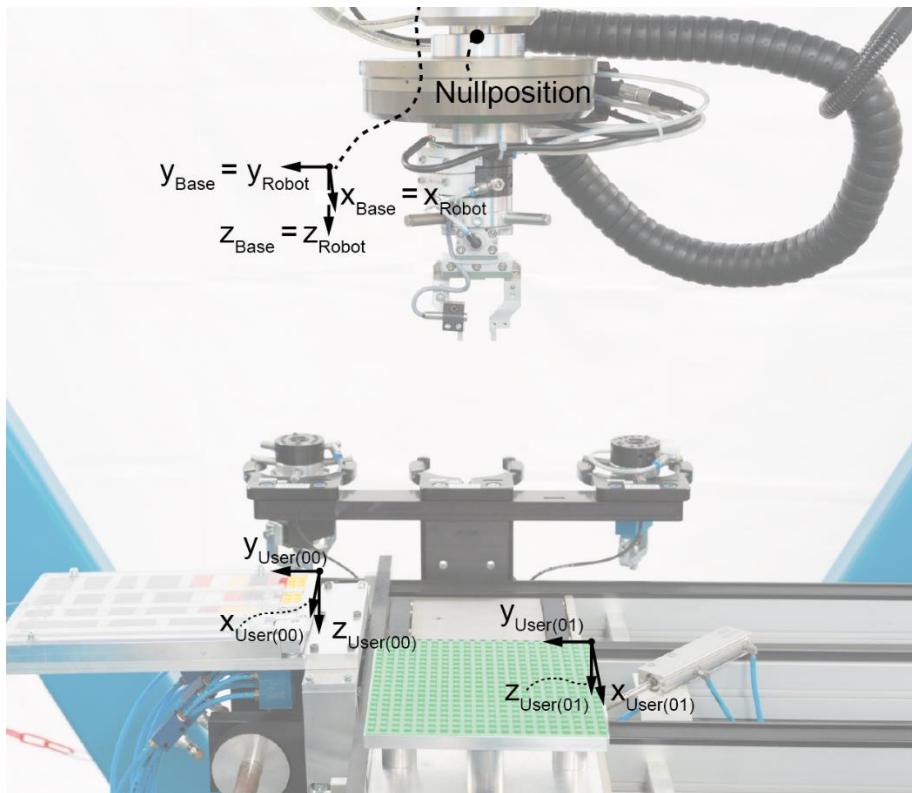


Abbildung 9-2: Angepasste Pneumatikverschaltung von Greifer und Greiferwechselsystem (Quelle: Eigene Ausarbeitung).

9.3 Koordinatensysteme des Roboters

Zur Steuerung des MPP3 sind ab Werk vier Koordinatensysteme implementiert. Im **achsbezogenen Koordinatensystem** Pulse kann die Stellungen einer jeden Achse des Roboters in Bezug auf den Rückgabewert des Inkrementalgebers des jeweiligen Servomotors angegeben werden. Die drei Achsen des Armsystems werden dabei als S-, L- und U-Achse bezeichnet. Die in der Arbeitsplatte verbaute Achse zur Verdrehung des Greifers um die Hochachse ist als T-Achse festgelegt. Innerhalb des in Kapitel 11.1 erläuterten und im Rahmen des zur Entwicklung der Software verwendeten Software-Development-Kits kommt darüber hinaus ein weiteres **achsbezogenes Koordinatensystem** zum Einsatz, welches die Stellung der Achsen in Grad wiedergibt.

Neben diesen beiden Koordinatensystemen sind darüber hinaus zwei kartesische Koordinatensysteme vorhanden. Diese liegen deckungsgleich in der Basis des Deltaroboters und werden als *Base-* und *Robot-Koordinatensystem* bezeichnet. Die Orientierung dieser beiden Koordinatensysteme und die Nullposition des Roboters werden durch Abbildung 9-3 wiedergegeben.



*Abbildung 9-3: Die Koordinatensysteme der Roboterzelle
(Quelle: Eigene Ausarbeitung).*

Um die Handhabung der Bauteile im Arbeitsraum zu erleichtern, sind zwei User-Koordinatensysteme geteacht. Das User-Koordinatensystem 00 ist zur Arbeit im Bereich des Zuführsystems zweckdienlich. Hierzu wurden mit Hilfe eines Kalibrierstifts der Ursprungspunkt des Koordinatensystems, ein Punkt auf der X-Achse sowie ein Punkt in der XY-Ebene geteacht. Das zweite User-Koordinatensystem ist auf die einzelnen Werkstückträger referenziert. Alle Werkstückträger weisen zwei Bezugskanten (BK) auf, auf welche die Lego-Montageplatte ausgerichtet ist. Der Eckpunkt eines jeden Werkstückträgers, an dem die beiden Bezugskanten aufeinandertreffen, ist an der Unterseite mit BK gekennzeichnet. Alle Werkstückträger müssen stets so auf das Förderband gelegt werden, dass der Eckpunkt mit der Markierung BK dem Eckpunkt entspricht, in dem die Spannvorrichtung die Kraft einleitet, da in diesem Eckpunkt der Ursprung des User-Koordinatensystems 01 liegt. Abbildung 9-3 veranschaulicht neben der Orientierung des Base-Koordinatensystems auch die Lage und Orientierung der beiden User-Koordinatensysteme.

Beim Einmessen der User-Koordinatensysteme mit dem Kalibrierstift ist Werkzeug 01 mit Hilfe des PHGs zu wählen. Diesem Werkzeug ist eine TCP-Verschiebung von 61,4 mm zugeordnet. Der Kalibrierstift kann aufgrund fehlender Zentrierungen exzentrisch am Greiferwechselsystem montiert sein. Somit

muss die Stellung der T-Achse bei Kalibervorgängen definiert sein, da ansonsten Abweichungen aufgrund der Exzentrizität des Kalibrierstifts einfließen. Für Messvorgänge ist daher die Stellung 18974 im Koordinatensystem Pulse vorgegeben.

9.4 Festlegung der Speicherbereiche

Die Robotersteuerung bietet Variablen verschiedener Datentypen, die aufgrund des vergleichsweise kleinen Speichers der Steuerung nur in begrenzter Anzahl verwendet werden können. Zum Teachen von einzelnen Positionen sind als P-Variablen bezeichnete Strukturen hilfreich, mit denen Positionen im Arbeitsraum gespeichert und bei Bedarf reproduziert werden können. Um größtmögliche Struktur im Aufbau der Software gewährleisten zu können, wurden die verwendeten P-Variablen in Gruppen eingeteilt. Eine Übersicht kann durch Tabelle 9-1 gegeben werden. Eine weitere Tabelle, welche den Zweck einer jeden einzelnen P-Variable wiedergibt, findet sich im Anhang in Abschnitt 17.5.

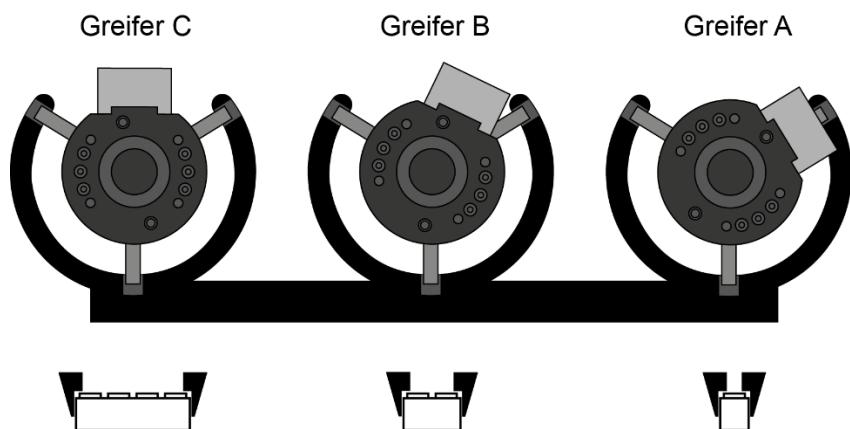
Tabelle 9-1: Einteilung der P-Variablen (Quelle: Eigene Ausarbeitung).

P-Variable	Zweck	Koordinaten-System
P000	Nullposition	Base
P001 - P009	unbenutzt	-
P010 - P029	Punkte der Offline-Legomontage	Pulse
P030 - P039	Aufnahme Greifer A	Base
P040 - P049	Aufnahme Greifer B	Base
P050 - P059	Aufnahme Greifer C	Base
P060 - P071	Punkte der Offline-Legomontage	Pulse

Der P-Variable P000 ist die Nullposition zugewiesen. Die Speicherbereiche P010 bis P029 sowie P060 bis P071 sind Punkte eines geteachten Programms, mit Hilfe dessen elf Legosteine auf dem Werkstückträger gefügt werden können. Zweck dieses Programms war es, die Fähigkeiten der Roboterzelle zu einem Zeitpunkt aufzeigen zu können, an dem die im Rahmen dieser Arbeit implementierte Software noch nicht einsatzfähig war. Bezugskoordinatensystem für diese Variablen ist das *Pulse*-Koordinatensystem. Die Speicherbereiche P030 bis P039 dienen zur Aufnahme des Greifers A. Hierbei kommt das *Base*-Koordinatensystem zum Einsatz, da die Steuerung des Roboters in kartesischen Koordinaten weitaus einfacher ist als die Steuerung im *Pulse*-Koordinatensystem. Für die Aufnahmevergänge der Greifer B und C wurden jeweils neun P-Variablen reserviert.

9.5 Werkzeugwechselvorgänge

Um automatisierte Greiferwechselvorgänge durchführen zu können, muss die Position und Lage der Greifer im Greiferbahnhof definiert sein. Grundsätzlich muss die Zuordnung von Greifer zu Haltebucht des Greiferbahnhofs bekannt sein. Da die Greifer von drei um 120° versetzt angebrachte Bolzen im Greiferbahnhof gehalten werden und da jedem Bolzen eine Nut in der Haltebacke des Greiferbahnhofs zukommt, kann jeder Greifer in drei um die Hochachse um 120° verdrehten Lagen im Greiferbahnhof abgelegt werden. Daher sind die in Abbildung 9-4 aus der Draufsicht dargestellten Positionen und Lagen für die drei Greifer einzuhalten. Der reale Greiferbahnhof mit den Greifern A und C in den jeweils zugehörigen Haltebuchten ist in Abbildung 9-3 illustriert.



*Abbildung 9-4: Position und Lage der drei Greifer aus der Draufsicht
(Quelle: Eigene Ausarbeitung).*

Greifer C ist in der linken Bucht zu lagern, Greifer B in der mittleren und Greifer A in der rechten. Hinsichtlich der korrekten Lage ist darauf zu achten, dass die Kontaktstecker der drei Greifer nach hinten, weg vom Greiferbahnhof zeigen.

Die Werkzeugwechselvorgänge der drei Greifer werden durch eine Abfolge von linearen Bewegungen durchgeführt. Diese Bewegungen sind relativ komplex, da die Greifer teilweise breiter sind als die Öffnungen der Haltebuchten. Um dieser Problematik entgegenzutreten werden die Greifer während der Linearbewegung über die Endposition im Greiferbahnhof um die Hochachse verdreht. Die Entwicklung von Algorithmen zum Generieren automatischer Werkzeugwechsel steht außer Frage, da der hierzu notwendige Aufwand nicht in Relation mit dem ziehbaren Nutzen steht.

Die Greiferwechselvorgänge sind daher von Hand geteacht, wozu die in Tabelle 9-1 erläuterten P-Variablen verwendet werden. Hierzu sind auf der Robotersteuerung sechs Motoman-Job-Dateien gespeichert, gekennzeichnet durch die Da-

teiendung .JBI. Die Programmierung dieser Dateien erfolgt in der auf der Steuerung implementierten objektorientierten Programmiersprache INFORM III. Pro Greifer finden sich jeweils eine Datei zur Aufnahme des Greifers und eine Datei zur Ablage des Greifers. Abbildung 9-5 veranschaulicht das Programm TAKEGA.JBI zur Aufnahme von Greifer A.

```
NOP                                //Programmstart
DOUT OT#(5) ON                      //Greiferwechselsystem lösen
DOUT OT#(6) OFF                     //Greifer schließen
MOVL P030 V=200.0                   //Linearbewegung zu Punkt P030, v=200 mm/s
MOVL P031 V=75.0                    //Linearbewegung zu Punkt P031, v=75 mm/s
DOUT OT#(5) OFF                     //Greiferwechselsystem spannen
TIMER T=0.500                       //Timer zur Kompensation der Systemtotzeit
MOVL P032 V=40.0                    //Linearbewegung zu Punkt P032, v=40 mm/s
MOVL P033 V=200.0                   //Linearbewegung zu Punkt P033, v=200 mm/s
MOVL P034 V=200.0                   //Linearbewegung zu Punkt P034, v=200 mm/s
END                               //Programmende
```

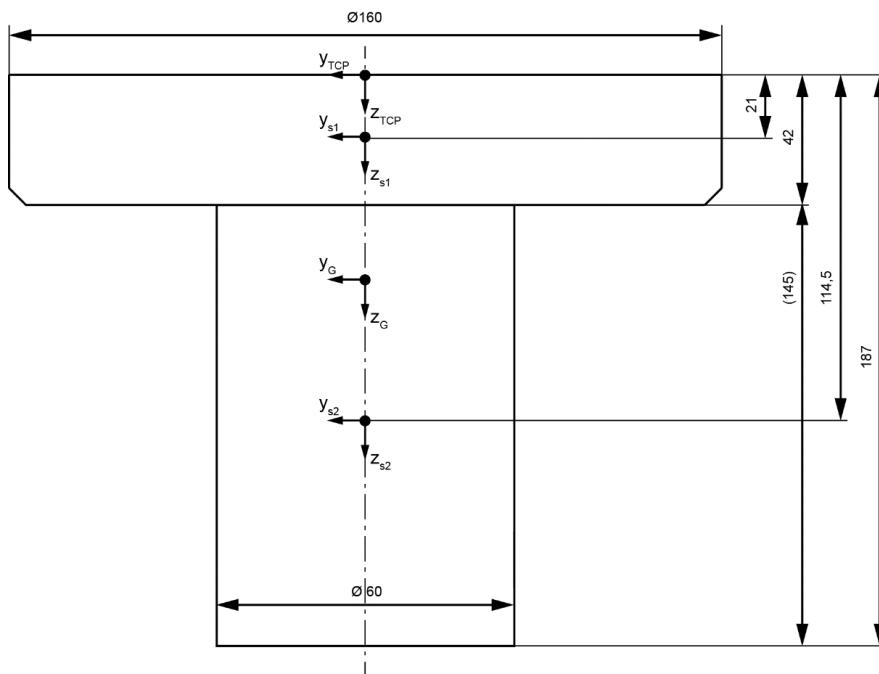
Abbildung 9-5: Auf der Robotersteuerung implementiertes JBI-Programm zur Aufnahme des Greifers A (Quelle: Eigene Ausarbeitung).

Die Namensgebung der Job-Dateien unterliegt folgender Nomenklatur: Programme mit dem Präfix *PUTAWAY* dienen zum Ablegen von Greifern. Hinter diesem Präfix ist gekennzeichnet, welcher Greifer G abgelegt werden soll. Mit Hilfe der Job-Datei *PUTAWAYGA.JBI* wird beispielsweise Greifer A in die Haltebucht bewegt. Die Dateien zum Aufnehmen von Greifern unterliegen derselben Nomenklatur, allerdings mit dem Präfix *TAKE*.

Um die Programmaufrufe für den User möglichst angenehm zu gestalten, sind die Programme zum Ablegen und Aufnehmen einzelner Greifer in übergeordneten Programmen zusammengefasst. Im Hinblick auf die Namensgebung wird zuerst angegeben welcher Greifer G abgelegt werden soll. Im Anschluss folgt das Kürzel *WW*, welches Aufschluss darüber gibt, dass diese Job-Datei einen Werkzeugwechsel veranlasst. Die letzten beiden Buchstaben geben an welcher Greifer aufgenommen werden soll. Das Programm *GAWWGB.JBI* beispielsweise ruft zuerst das Unterprogramm *PUTAWAYGA.JBI* auf. Im Anschluss daran wird das Programm *TAKEGB.JBI* aufgerufen. Somit erfolgt ein Greiferwechsel, in dessen Verlauf der Greifer A im Greiferbahnhof angelegt wird und B im Anschluss aufgenommen wird. Start- und Endpunkt eines jeden Greiferwechselvorgangs ist immer die Nullposition P000.

9.6 Einstellen des Massenträgheitsmoments

Für eine möglichst exakte Steuerung des Roboters durch die FS100 sind neben der Vorgabe des Zielpunkts im Arbeitsraum auch die Berücksichtigung des Massenträgheitsmoments und des Schwerpunkts des Werkzeugs erforderlich. Wie in Abschnitt 11.4.4.7 näher erläutert, wird bei mit MotoPlusSDK entwickelten Applikationen lediglich das Standardwerkzeug berücksichtigt. Eine greiferabhängige Spezifikation von Massenträgheitsmoment und Werkzeugschwerpunkt ist nicht möglich. Eine korrekte Ermittlung des Massenträgheitsmoments ist allerdings ohnehin unmöglich, da die CAD-Modelle des KMS und der drei Greifer nicht vorliegen und der Roboter nicht in der Lage ist dieses zu vermessen. Grundlage für die Berechnung des Massenträgheitsmoments ist daher ein aus zwei Zylindern bestehendes Modell. Abbildung 9-6 veranschaulicht dieses.



*Abbildung 9-6: Technische Zeichnung des Modells
(Quelle: Eigene Ausarbeitung).*

Der obere als Zylinder 1 bezeichnete Körper dient als Substitut für den KMS. Die Abmessungen des Zylinders entsprechen denen des Sensors, die der Berechnung zu Grunde gelegte Masse ebenso. Es wird davon ausgegangen, dass der Massenmittelpunkt im Volumenschwerpunkt des Sensors liegt. Der untere Zylinder entspricht dem Greifer. Da die drei Greifer mit unterschiedlichen Verdrehungswinkeln am Greiferwechselsystem angebracht sind, ist ein Quader als Modell für den Greifer nicht zweckdienlich. Daher wird ebenfalls mit einem Zylinder gerechnet, der als Hüllkörper in etwa den durchschnittlichen Greiferabmessungen entspricht und dessen Gewicht dem schwersten der drei Greifer

gleichkommt. Bei Zylinder 2 wird ebenfalls davon ausgegangen, dass dieser Körper homogen ist, so dass Massenmittelpunkt und Volumenschwerpunkt zusammenfallen. Tabelle 9-2 fasst die Abmessungen und die Massen der beiden Zylinder zusammen.

Tabelle 9-2: Relevante physikalische Größen der Körper des Modells zur Berechnung des Massenträgheitsmoments (Quelle: Eigene Ausarbeitung).

Zylinder i	Physikal. Größe	Formelzeichen	Wert
1	Durchmesser	D_1	160 mm
	Höhe	h_1	42 mm
	Masse	m_1	1,56 kg
2	Durchmesser	D_2	60 mm
	Höhe	h_2	145 mm
	Masse	m_2	1,40 kg

Einleitend muss zunächst der Massenmittelpunkt $_{TCP}\vec{r}_{0G}$ des Gesamtmodells berechnet werden. Dieser berechnet sich wie folgt:

$$_{TCP}\vec{r}_{0G} = \frac{1}{M} \sum_i m_i \, _{TCP}\vec{r}_{0si} \quad (2)$$

$_{TCP}\vec{r}_{0G}$:= Vektor zum Massenmittelpunkt des Gesamtmodells im Koordinatensystem TCP am Tool-Center-Point des Roboters

M := Masse des Gesamtmodells

m_i := Masse des Zylinders i

$_{TCP}\vec{r}_{0si}$:= Vektor zum Massenmittelpunkt des Zylinders i im Koordinatensystem TCP

Die Masse des Gesamtmodells ergibt sich aus der Summe der Massen der Einzelkörper.

$$M = \sum_i m_i \quad (3)$$

Mit den gegebenen Zahlenwerten ergibt sich somit für die Gesamtmasse aus Formel (3) folgende Berechnung:

$$M = 1,56 \text{ kg} + 1,40 \text{ kg} = 2,96 \text{ kg}$$

Die Vektoren zu den Schwerpunkten der beiden Zylinder im Koordinatensystem TCP können aus der Zeichnung abgelesen werden und lauten:

$${}_{TCP}\vec{r}_{0s1} = \begin{pmatrix} 0 \\ 0 \\ 21,0 \end{pmatrix} \text{ mm}$$

$${}_{TCP}\vec{r}_{0s2} = \begin{pmatrix} 0 \\ 0 \\ 114,5 \end{pmatrix} \text{ mm}$$

Der Vektor zum Massenmittelpunkt des Gesamtkörpers im Koordinatensystem TCP errechnet sich unter Berücksichtigung dieser beiden Vektoren sowie der Gesamtmasse bei Anwendung von Formel (2) zu:

$${}_{TCP}\vec{r}_{0G} = \begin{pmatrix} 0 \\ 0 \\ 65,2 \end{pmatrix} \text{ mm.}$$

Das Massenträgheitsmoment im körperfesten Koordinatensystem eines Zylinders kann mit Hilfe der Formeln (4) und (5) berechnet werden.

$${}_{si}\Theta_{xxi} = {}_{si}\Theta_{yyi} = \frac{m_i}{12} \left(3 \left(\frac{D_i}{2} \right)^2 + h_i^2 \right) \quad (4)$$

$${}_{si}\Theta_{zzi} = \frac{1}{2} m_i \left(\frac{D_i}{2} \right)^2 \quad (5)$$

${}_{si}\Theta_{xx}$:= Massenträgheitsmoment der x-Achse des Körpers i im körperfesten Koordinatensystem des Körpers i, für die y- und z-Achse analog.

D_i := Durchmesser des Zylinders i

h_i := Höhe des Zylinders i

Damit ergibt sich für die beiden Zylinder folgende Rechnung:

$${}_{s1}\Theta_{xx1} = {}_{s1}\Theta_{yy1} = 2725 \text{ kg mm}^2$$

$${}_{s1}\Theta_{zzi} = 4993 \text{ kg mm}^2$$

$${}_{s2}\Theta_{xx2} = {}_{s2}\Theta_{yy2} = 2768 \text{ kg mm}^2$$

$${}_{s2}\Theta_{zzi} = 630 \text{ kg mm}^2$$

Das Massenträgheitsmoment des Gesamtmodells errechnet sich aus der Summe der Trägheitsmomente der beiden Zylinder. Hierbei muss allerdings die Verschiebung des Bezugspunktes vom körperfesten Koordinatensystem der Einzelkörper zum Körperschwerpunkt des Gesamtmodells miteinbezogen werden. Hierzu finden die Sätze von Huygens-Steiner Anwendung.

$${}_G\Theta_{xxi} = {}_{si}\Theta_{xxi} + (b_i^2 + c_i^2) m_i \quad (6)$$

$${}_G\Theta_{yyi} = {}_{si}\Theta_{yyi} + (a_i^2 + c_i^2) m_i \quad (7)$$

$${}_G\Theta_{zzi} = {}_{si}\Theta_{zzi} + (a_i^2 + b_i^2) m_i \quad (8)$$

$${}_G\vec{r}_{Gsi} = {}_{TCP}\vec{r}_{0si} - {}_{TCP}\vec{r}_{0G} = \begin{pmatrix} a_i \\ b_i \\ c_i \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ c_i \end{pmatrix} \quad (9)$$

Da die Achsen beider Zylinder sowie der Massenschwerpunkt auf einer Linie liegen, sind die x- und y-Komponenten des Ortsvektors

$$a_i = b_i = 0$$

woraus folgt

$${}_G\Theta_{xxi} = {}_G\Theta_{yyi} = {}_{s1}\Theta_{xxi} + c_i^2 m_i \quad (10)$$

$${}_G\Theta_{zzi} = {}_{s1}\Theta_{zzi}. \quad (11)$$

Für die beiden Zylinder ergibt sich damit folgende Rechnung:

$$\begin{aligned} {}_G\vec{r}_{Gs1} &= \begin{pmatrix} 0 \\ 0 \\ 21,0 \end{pmatrix} mm - \begin{pmatrix} 0 \\ 0 \\ 65,2 \end{pmatrix} mm = \begin{pmatrix} 0 \\ 0 \\ -44,2 \end{pmatrix} mm \\ {}_G\Theta_{xx1} &= {}_G\Theta_{yy1} = 5775 \text{ kg mm}^2 \\ {}_G\Theta_{zz1} &= {}_{s1}\Theta_{zz1} = 4993 \text{ kg mm}^2 \\ {}_G\vec{r}_{Gs2} &= \begin{pmatrix} 0 \\ 0 \\ 114,5 \end{pmatrix} mm - \begin{pmatrix} 0 \\ 0 \\ 66,2 \end{pmatrix} mm = \begin{pmatrix} 0 \\ 0 \\ 48,3 \end{pmatrix} mm \\ {}_G\Theta_{xx2} &= {}_G\Theta_{yy2} = 6034 \text{ kg mm}^2 \\ {}_G\Theta_{zz2} &= {}_{s1}\Theta_{zz1} = 630 \text{ kg mm}^2 \end{aligned}$$

Zur Berechnung der Massenträgheitsmomente des Gesamtmodells werden im Anschluss die Massenträgheitsmomente der Einzelkörper addiert.

$${}_G\Theta_{xx\Sigma} = {}_G\Theta_{yy\Sigma} = {}_G\Theta_{xx1} + {}_G\Theta_{xx2} = 11809 \text{ kg mm}^2 = 0,012 \text{ kg m}^2$$

$${}_G\Theta_{zz\Sigma} = {}_G\Theta_{zz1} + {}_G\Theta_{zz2} = 5623 \text{ kg mm}^2 = 0,0056 \text{ kg m}^2$$

Die hier berechneten Ergebnisse des Massenschwerpunkts und der Massenträgheitsmomente des oben beschriebenen Modells sind auf der FS100 als Parameter des Standardwerkzeugs 0 eingetragen.

10 Realisierung der Softwarekomponente für den Leitrechner

Hinsichtlich der Realisierung der Softwarekomponente für den Leitrechner werden die Anforderungen an den Client nochmals beleuchtet. Im Anschluss daran wird die gewählte Entwicklungsumgebung kurz beschrieben. In den darauf folgenden Kapiteln wird jeder der drei parallel arbeitenden Threads des Clients ausführlich erläutert.

10.1 Anforderungen an den Client

Der Client dient zur Steuerung der Roboterzelle durch den Benutzer. Dies soll durch ein zu implementierendes GUI ermöglicht werden. Darüber hinaus müssen die notwendigen Steuerungsbefehle per Ethernet an die FS100 übertragen werden. Die versendeten Befehle sollen durch das GUI dargestellt werden.

Parallel zu diesen Vorgängen soll der am TCP verbaute KMS per CAN-Bus angesteuert werden. Hierzu dient eine im PC verbaute CAN-Bus-Karte. Der PC muss die vom KMS gesendeten Sensordaten empfangen, verwerten und per Ethernet an die FS100 weiterleiten.

10.2 Beschreibung der gewählten Entwicklungsumgebung

Das .NET-Framework ist eine Softwareplattform, mit deren Hilfe Anwendungsprogramme entwickelt und ausgeführt werden können. Bestandteil dieses Frameworks sind beispielsweise Klassenbibliotheken zum Erstellen von Windows Forms (WILLMS 2011). Daher empfiehlt sich die Verwendung einer Sprache, mit der auf die Microsoft-Klassenbibliotheken des .NET-Frameworks zugegriffen werden kann. Zur Umsetzung eines GUI kommen daher die höheren Programmiersprachen C++/CLI, C# und Visual Basic in Betracht.

Als Programmiersprache wurde C++/CLI gewählt. Diese Sprache ist eine Erweiterung von C++ um das .NET-Framework. Die CAN-Bus-Library basiert auf dem in der Programmiersprache C gehaltenen Beispielprojekt der Softing Industrial Automation GmbH. Dieses wurde für die Zwecke dieses Projekts entsprechend modifiziert. Somit finden innerhalb des Projekts sowohl die Programmiersprache C als auch C++/CLI Anwendung. Entwicklungsumgebung des Projekts ist Visual Studio Premium 2013.

10.3 Das Windows Form als Main-Thread

Main-Task der Softwarekomponente für den Leitrechner ist das Windows Form. Hierdurch wird sichergestellt, dass die beiden anderen Threads beendet werden, sobald das Windows Form geschlossen wird.

Einleitend wird im Rahmen dieses Unterkapitels das erstellte GUI beschrieben. Daran schließt ein Kapitel zum strukturellen Aufbau des Threads an. Da von den anderen Threads auf Elemente des Main-Threads zugegriffen wird, muss die Threadsicherheit gewährleistet sein. Dies wird in einem weiteren Abschnitt dargelegt.

10.3.1 Die graphische Benutzeroberfläche

Abbildung 10-1 zeigt die programmierte graphische Benutzeroberfläche der Softwarekomponente für den Leitrechner.

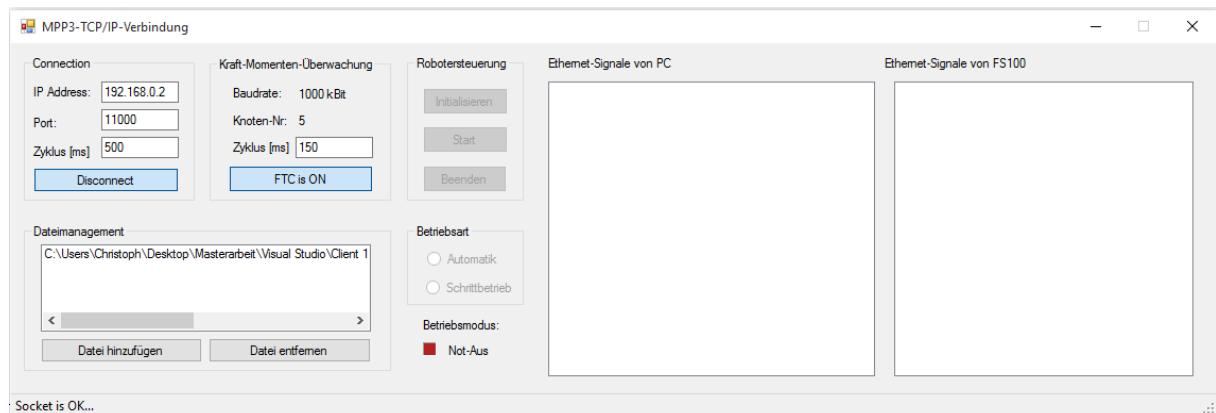


Abbildung 10-1: GUI der auf dem Leitrechner laufenden Softwarekomponente (Quelle: Eigene Ausarbeitung).

In der Gruppe *Connection* können IP-Adresse, Port und Zykluszeit in Millisekunden für die Ethernet-Kommunikation mit der FS100 parametriert werden. Die IP-Adresse der FS100 ist in den Systemeinstellungen fest vorgegeben. Die Port-Nummer ist Teil des Programmcodes des Servers und kann ebenfalls nicht beliebig verändert werden. Die Zykluszeit gibt wieder, in welchem Abstand der Client Anfragen an den Server versendet. Unter diesen drei Eingabefeldern befindet sich der Switch-Button *Connect*. Bei Betätigung dieses Buttons werden die untergeordneten Threads zur Ethernet-Kommunikation und zum CAN-Bus-Handling gestartet.

Daneben befindet sich die Gruppe *Kraft-Momenten-Übertragung*. Diese Gruppe beinhaltet neben Informationen zur eingestellten Baudrate und zur Knotennummer

mer des Kraft-Momenten-Sensors auch eine TextBox zur Vorgabe der Zykluszeit. Dieser Parameter gibt an, in welchem Zeitabstand die per CAN-Bus abgerufenen Sensordaten an die FS100 gesendet werden. Untergrenze für die Zykluszeit sind 200 ms. Bei geringeren Zeitabständen reicht die Verarbeitungsgeschwindigkeit der FS100 nicht aus, um die eingehenden Daten verarbeiten zu können.

Zur Steuerung der Roboterzelle beinhaltet das Windows Form drei in Abbildung 10-1 inaktive Buttons. Der Button *Initialisieren* dient zum Aufruf von Routinen auf der FS100, die nach dem Starten des Demonstrators ablaufen müssen, um den Roboter in einen definierten Systemzustand zu bringen. Mit dem Button *Start* kann ein Montagevorgang gestartet werden. Der Button *Beenden* dient dazu, den Montagevorgang vorzeitig abzubrechen. Es kann zwischen den Betriebsarten *Schrittbetrieb* und *Automatik* gewählt werden. Im Automatikmodus wird nach dem Fügen eines jeden Steins automatisch der nächste BrickCAD-Datenstring vom Client an den Server gesendet. Im Schrittbetrieb muss nach jedem Lego-Stein erneut der *Start*-Button betätigt werden.

Unter der Gruppe *Betriebsart* befindet sich eine Anzeige zur Rückmeldung des *Betriebsmodus* der FS100. Der Text sowie die Farbe des Feldes links daneben geben dynamisch zur Laufzeit den Betriebsmodus des Servers an.

Links neben dieser Anzeige befindet sich eine Gruppe zum *Dateimanagement*. Mit Hilfe des Buttons *Datei hinzufügen* kann eine mit BrickCAD generierte build-Datei in die Warteschlange der zu montierenden Lego-Baugruppen aufgenommen werden. Die Warteschlange wird nach dem FIFO-Prinzip (First In First Out) abgearbeitet. In der sich darüber befindlichen Listbox wird der Dateipfad aller build-Dateien in der Warteschlange angezeigt. Mit Hilfe des Buttons *Datei entfernen* können Dateien aus der Warteschlange entfernt werden, sofern diese nicht bereits zur Bearbeitung geöffnet sind.

Die beiden Listboxen auf der rechten Seite der GUI dienen zur Anzeige der Anfragen und Rückmeldungen zwischen Leitrechner und FS100. Auf eine Live-Anzeige der auftretenden Kräfte und Momente wird vorerst verzichtet.

10.3.2 Struktureller Aufbau des Main-Threads

Um den strukturellen Aufbau des Threads erläutern zu können, müssen zunächst die Attribute und Methoden der Klasse aufgezeigt werden. Auf Grundlage dieser Darstellung kann im Anschluss die Bedeutung des Threads im Hinblick auf die anderen Threads erörtert werden. Abbildung 10-2 zeigt die Attribute und Methoden der Klasse des Windows Forms.

```

Form1

- Elemente des WindowsForms
- strClientCommand: String^
- strServerResponse: String^
- strGuiCommand: String^
- strClientCommandBefore: String^
- strServerResponseBefore: String^
- strFileString: String^
- iOperatingMode: int
- iProcessMode: int
- iMeasureOn: int
- bFlagDeleteFinishedFile: bool
- bFlagAlreadyConnected: bool
- bFlagEndConnection: bool
- CommunicationThread: Thread^
- CanTransmissionThread: Thread^
- Operator: OperatingHandler

+ Form1()
- ~Form1()
- InitializeComponent()
- InitializeVariables()
- WriteToStatusBar(Message: String^)
- DoNetworkingConnection()
- ConnectTo()
- ConnectToCan()
- BreakButton_Click(sender: System::Object^ , e: System::EventArgs^)
- QuitButton_Click(sender: System::Object^ , e: System::EventArgs^)
- InitButton_Click(sender: System::Object^ , e: System::EventArgs^)
- PlayCheckBox_CheckedChanged(sender: System::Object^ , e: System::EventArgs^)
- AddFileButton_Click(sender: System::Object^ , e: System::EventArgs^)
- RemoveFileButton_Click(sender: System::Object^ , e: System::EventArgs^)
- ConnectButton_CheckedChanged(sender: System::Object^ , e: System::EventArgs^)
- FTConCheckbox_CheckedChanged(sender: System::Object^ , e: System::EventArgs^)
- SetListboxClientSent(text: String^)
- SetListboxServerResponse(text: String^)
- SetButtons(iOperatingMode: int, iProcessMode: int)
- SetOperatingLabel(int iOperatingMode: int)
- DeleteFinishedFileFromQueue(iProcessMode: int)
- SetConnectButton(text: String^)
- FctMakeEntryToLogFiles(strForces: String^, strMoments: String^_
_, forces: StreamWriter^, moments: StreamWriter^)
```

Abbildung 10-2: Attribute und Methoden der Klasse Form1 des Windows Forms (Quelle: Eigene Ausarbeitung).

Die aufgelisteten Attribute dienen größtenteils als globale Variablen. Welche Rolle diesen Variablen zukommt, wird in den nachfolgenden Unterkapiteln erläutert. Hinsichtlich den Methoden der Klasse fällt auf, dass alle Elemente zur Benutzerinteraktion eine Methode aufrufen. Von besonderer Relevanz für die beiden anderen Threads ist die Methode `ConnectButton_CheckedChanged([...])`. Diese wird ausgeführt, sobald eine Veränderung des Zustands des Switch-Buttons `Connect` auftritt. Wird der Button aktiviert, so wird die Methode `DoNetworkingConnection()` aufgerufen. Diese Methode wiederum startet den `CommunicationThread` und den `CanTransmissionThread`. Bei Start des `CommunicationThread` wird die Methode `ConnectTo()` aufgerufen. Diese ist für die Datenübertragung zur FS100 per Ethernet zuständig. Darüber hinaus wird mit Start des `CanTransmissionThreads` die Methode `ConnectToCan()` aufgerufen, der die CAN-Bus-Kommunikation und die

Datenübertragung der Sensordaten zukommt. Die beschriebenen Abläufe werden durch Abbildung 10-3, ein Interaktionsübersichtsdiagramm nach UML-Standard aufgezeigt.

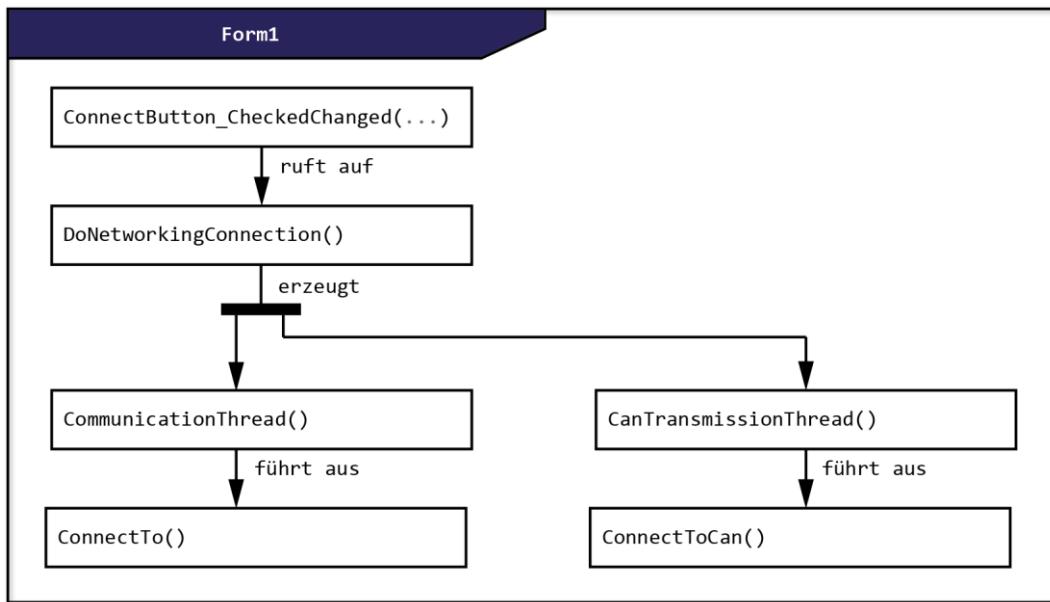


Abbildung 10-3: Interaktionsübersichtsdiagramm zu den Prozessen bei Aktivierung des Switch-Buttons Connect (Quelle: Eigene Ausarbeitung).

10.3.3 Umsetzung der Threadsicherheit

Die für den Leitrechner bestimmte Softwarekomponente basiert, wie bereits erläutert, auf einem Multithreading-Prozess. Diese Architektur führt dazu, dass die Threadsicherheit beachtet werden muss, da die Threads theoretisch gleichzeitig auf dieselben Methoden und Attribute zugreifen können.

Bestes Beispiel hierfür ist die Integer-Variable `iOperatingMode`. Diese Variable ist Teil eines im Thread zur Datenübertragung implementierten Zustandsautomaten. Der Zustand, in dem sich der Zustandsautomat befindet, kann einerseits durch Benutzereingaben per GUI, andererseits durch Rückantworten des Servers verändert werden. Betätigt der User beispielsweise den Button *Initialisieren*, so geht der Zustand Idle in den Zustand Init über. Um dies bewerkstelligen zu können, greifen Methoden des Main-Threads auf die Variable `iOperatingMode` zu. Gleichzeitig kann diese Variable aber auch durch den `CommunicationThread` verändert werden, da bei Rückgabe eines Not-Aus-Signals vom Server in den Zustand EmergencyOff gesprungen wird. An dieser Stelle sei angemerkt, dass der angesprochene Zustandsautomat in Kapitel 10.4 ausführlich erläutert wird.

Die angesprochene Problematik und Garantie der Threadsicherheit kann mit Hilfe von Delegaten bewerkstelligt werden. Ein Delegat ist ein Verweis auf eine Methode. Die Funktion der Delegaten wird anhand von Abbildung 10-4 verdeutlicht. Die Methode des Programmauszugs dient dazu, die vom Client versendeten Anfragen in eine in der GUI enthaltene ListBox einzutragen.

Sofern der die Methode aufrufende Thread derselbe ist, wie derjenige der das Objekt erzeugt hat, so wird der übergebene String in Kombination mit der aktuellen Systemzeit in die Listbox eingetragen. Stimmen aufrufender und erzeugender Thread nicht überein, so erzeugt die Methode einen SetTextDelegaten. Dieser ruft sich selbst mittels der Invoke-Eigenschaft asynchron auf und stellt sicher, dass der Eintrag in die Listbox threadsicher erfolgt. Bezüglich weiterer Ausführungen zu threadsicherer Programmierung mit Windows Forms sei auf SELLS (2002) und WILLMS (2011) verwiesen.

```
private: void SetListboxClientSent(String^ text)
{
    if (this->ClientSentListBox->InvokeRequired)
    {
        SetTextDelegate^ d = gcnew SetTextDelegate(this, &Form1::SetListboxClientSent);
        this->Invoke(d, gcnew array < Object^ > { text });
    }
    else
    {
        String ^ ListboxEntry = DateTime::Now.ToString("T") + ":" + text;
        this->ClientSentListBox->Items->Insert(0, ListboxEntry);
    }
}
```

Abbildung 10-4: Verwendung eines Delegaten in der Methode SetListboxClientSent (Quelle: Eigene Ausarbeitung).

10.4 Thread zur Datenübertragung per Ethernet

Dieses Kapitel ist der Datenübertragung der Steuerungsbefehle an die FS100 per Ethernet gewidmet. Einleitend wird dazu der strukturelle Aufbau des Threads dargelegt. Die festgelegten Anfragen und Rückantworten von Client und Server sowie deren Bedeutung findet sich in Unterkapitel 10.4.2. Weiterer Teil dieses Kapitels sind Ausführungen zweier implementierter Zustandsautomaten, auf denen die Steuerung der Client-Server-Kommunikation basiert.

10.4.1 Struktureller Aufbau des Threads

Die Übertragung der Anfragen an den Server erfolgt im `CommunicationThread()`. Dieser wiederum ruft, wie bereits erläutert, die Methode `ConnectTo()` auf. Um die für diesen Thread verwendeten Klassen, deren Attribute und Methoden aufzeigen zu können, wurde das in Abbildung 10-5 dargestellte Klassendiagramm entwickelt.

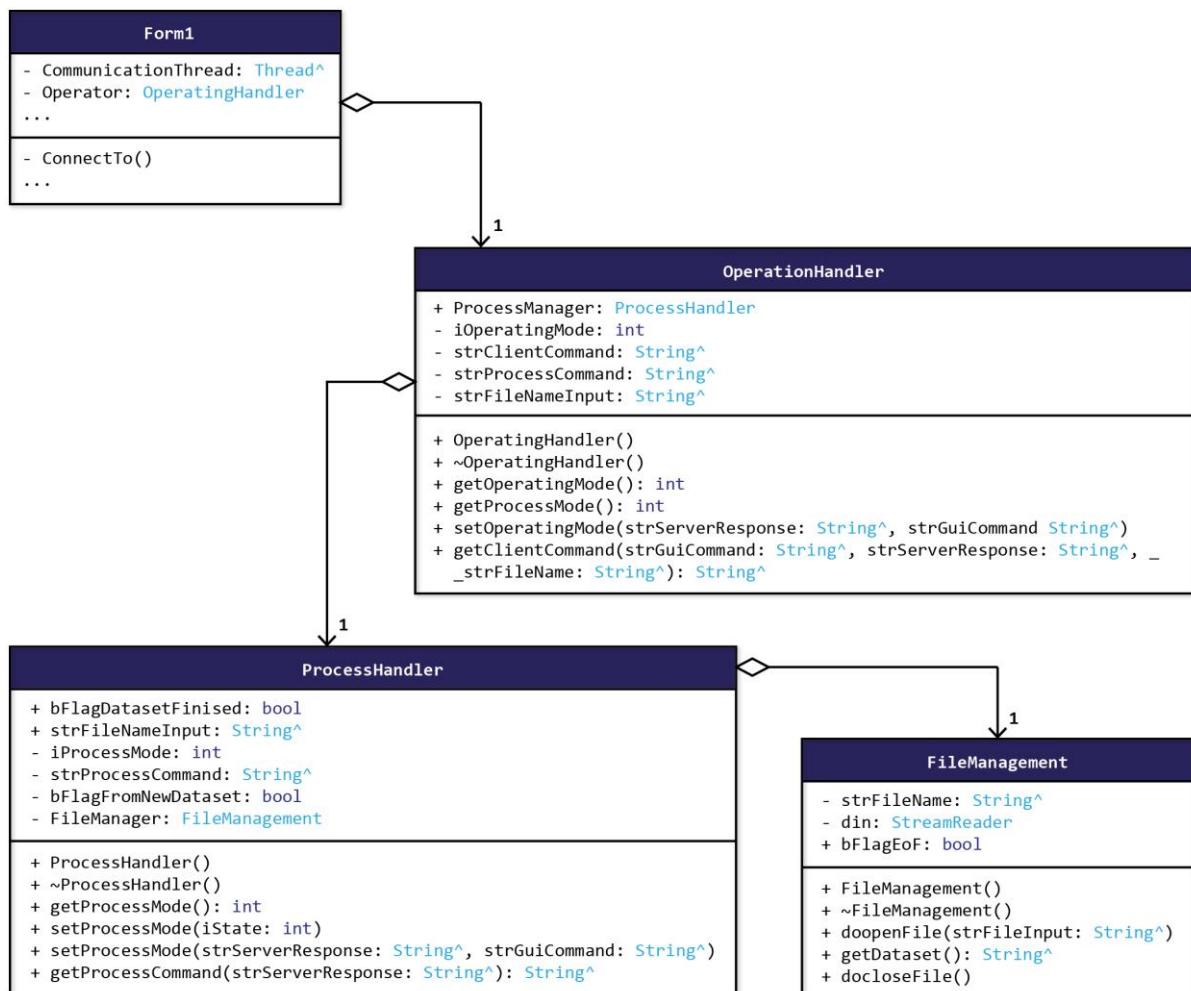


Abbildung 10-5: Auszug aus dem Klassendiagramm des Gesamtsystems mit Fokus auf den Thread zur Datenübertragung (Quelle: Eigene Ausarbeitung).

Wie bereits dargestellt, ist das Windows Form die Klasse, welche allen anderen übergeordnet ist. Da in Rahmen dieses Kapitels die Elemente dieser Klasse bereits dargelegt wurden, wird auf eine vollständige Darstellung an dieser Stelle verzichtet. Teil der Klasse `Form1` ist der `OperationHandler`. Hierbei handelt es sich um den bereits angedeuteten Zustandsautomaten, der die Grundzustände der Software behandelt. Mit der Klasse `ProcessHandler` ist dieser Klasse ein weiterer Zustandsautomat hierarchisch untergeordnet. Dieser dient dazu, die Zustände während der Montage einer Lego-Baugruppe handhaben zu können.

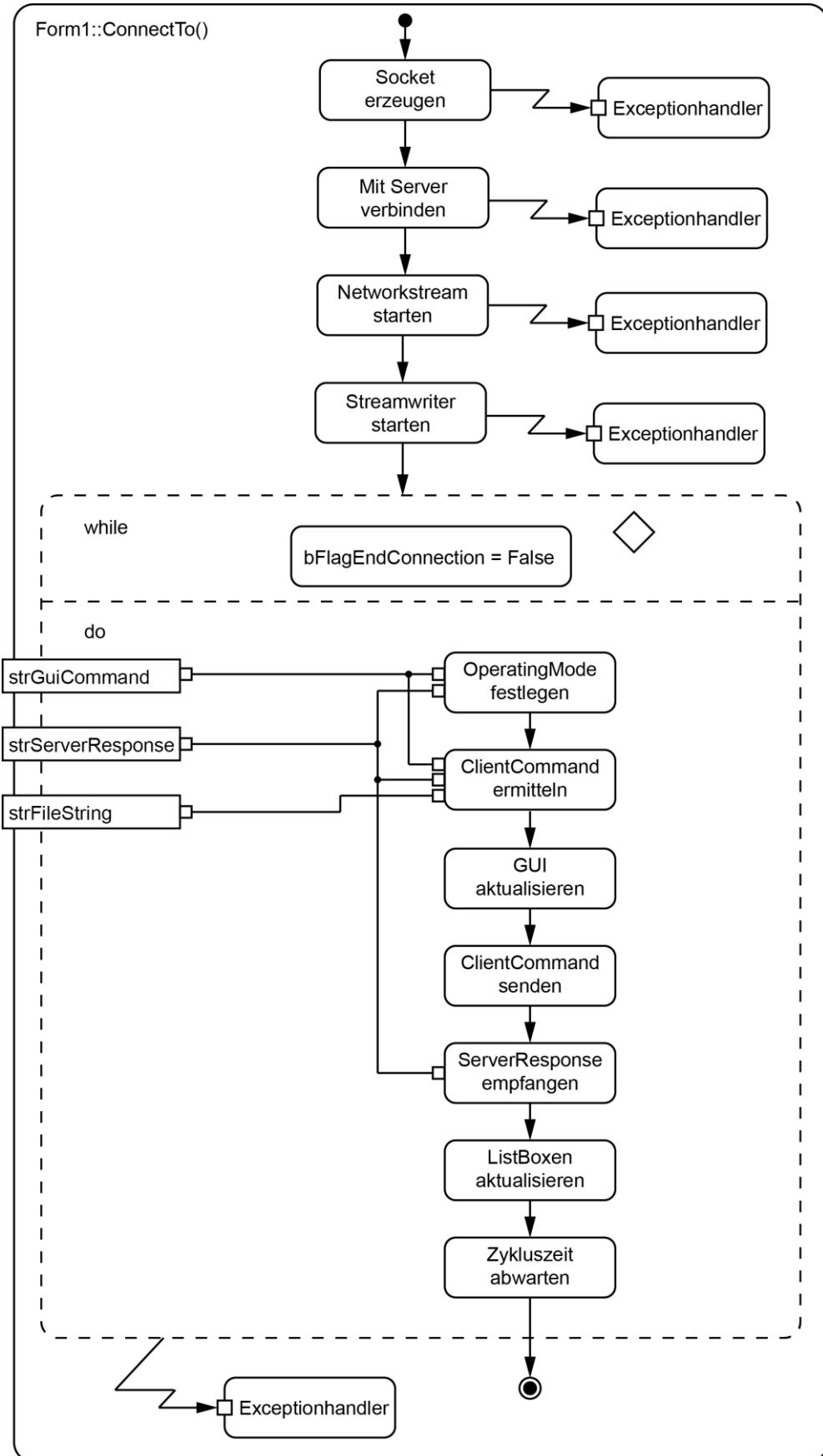
Da der ProcessHandler Zugriff auf die build-Datei der zu montierenden Lego-Baugruppe haben muss, ist diesem wiederum die Klasse FileManagement zugeordnet. Bestandteil dieser Klasse sind unter anderem Methoden um eine Text-Datei öffnen, einen BrickCAD-Datenstring auslesen und die Datei wieder schließen zu können. Hinsichtlich des ProcessHandlers fällt auf, dass die Methode setProcessMode überladen ist. Der ProcessMode kann auf zwei Arten festgelegt werden. Einerseits durch Vorgabe des Zielzustandes mit Hilfe einer Integer-Zahl, andererseits durch Analyse zweier Eingangsstrings. Deren Bedeutung sowie die implementierten Klassen Operating- und Processhandler werden in den nachfolgenden Abschnitten dieses Kapitels erläutert.

Die Methode ConnectTo() aus Form1 ist zentrales Element des CommunicationThreads. Zur Verdeutlichung der Abläufe dieser Methode dient ein Aktivitätsdiagramm. Siehe hierzu Abbildung 10-6.

Einleitend muss die TCP-IP-Verbindung hergestellt werden. Hierzu muss ein Socket erzeugt werden. Sockets sind vom Betriebssystem bereitgestellte Objekte, die als Kommunikationsendpunkte dienen (TANENBAUM & WETHERALL 2012). Im Anschluss kann die Verbindung hergestellt und der Networkstream gestartet werden. Letzterer beinhaltet die zu übertragenden Daten in einer geordneten Anzahl von Bytes (KÜHNEL 2010). Mit Hilfe der Klasse Streamwriter kann ein Zeichenstring an den Networkstream übergeben werden, der auf Basis des Sockets die Anfrage an den Server versendet. Tritt bei einer der genannten Aktionen ein Fehler in Erscheinung, so wird ein entsprechender Exceptionhandler aufgerufen.

In der auf diese Aktionen folgenden while-Schleife werden die für die Datenübertragung notwendigen Schritte zyklisch abgearbeitet, bis die Verbindung beendet werden soll. Einleitend wird der OperatingMode ermittelt. Diese Variable gibt den aktuellen Zustand des implementierten Zustandsautomanten wieder. Eingangsgröße hierfür ist der String strServerResponse, welcher die Rückantwort des Servers vom vorherigen Zyklusdurchlauf enthält. Weitere Eingangsgröße ist der String strGuiCommand. Diese Variable wird im Main-Thread beschrieben und ist abhängig von den Benutzereingaben. Wird beispielsweise der Button *Start* betätigt, so wird der String strGuiCommand mit der Zeichenfolge *Play* beschrieben.

Im Anschluss daran wird die zu sendende Anfrage an den Server ermittelt. Im Programmcode wird diese Anfrage als strClientCommand bezeichnet. Eingangsgröße für diese Methode ist, sofern ein BrickCAD-Datenstring übergeben werden muss, der String strFileString, erzeugt von der Klasse FileManager, welcher die benötigte Zeichenfolge enthält.



*Abbildung 10-6: Aktivitätsdiagramm der Methode `ConnectTo()`
(Quelle: Eigene Ausarbeitung).*

Auf Grundlage des OperatingModes wird daraufhin die GUI aktualisiert. Danach wird die Anfrage an den Server gesendet und auf die Rückantwort gewartet. Nach deren Empfang wird diese in den global verfügbaren String strServerResponse kopiert. Sofern sich die Anfrage oder die Rückantwort von denen des vorherigen Zyklus unterscheiden, werden die Listboxen des GUI geupdatet. Abschließend wird die über das GUI parametrierbare Zykluszeit abgewartet bis ein neuer Zyklus gestartet wird. Zum Auffangen auftretender Fehlersituationen ist der while-Schleife ebenfalls ein Exception-Handler zugeordnet.

10.4.2 Anfragen und Rückantworten von Client und Server

Um die Kommunikation zwischen Client und Server zu ermöglichen, muss definiert sein, welche Anfragen der Client an den Server stellen kann. Daneben müssen die möglichen Antworten des Servers ebenfalls bekannt sein, damit der Client diese verarbeiten kann. Für den Client sind folgende, in Tabelle 10-1 aufgelistete Anfragen festgelegt.

Tabelle 10-1: Anfragen des Clients (Quelle: Eigene Ausarbeitung).

Anfrage	Intention
DoNothing	Stillstand des Roboters erwünscht
DoInitialize	Initialisiere die Roboterzelle
DoBreak	Pausiere während des Montagevorgangs
Start	Startmarke Montagevorgang
Dataset	Datensatz z.B. „btEpx002y002z000o0ttBto0tp0id0001“
Transcomp	Endmarke Montagevorgang
State?	Anfrage zum Bearbeitungszustand eines Datensatzes

Es sei angemerkt, dass ein Großteil der Anfragen einem Befehl gleichkommt, der festlegt, welche Aktion die Roboterzelle auszuführen hat. Soll die Roboterzelle keine Aktion durchführen, so wird Befehl *DoNothing* übermittelt. Die Initialisierung der Roboterzelle erfolgt durch den Befehl *DoInitialize*. Der Befehl *DoBreak* dient dazu, die Roboterzelle während der Montage pausieren zu lassen. Der Befehl *Start* dient als Startmarke, welche den auf der FS100 implementierten Zustandsautomaten signalisiert, dass ein neuer Montageprozess durchgeführt werden soll. Als *Dataset* ist in der Tabelle ein BrickCAD-Datensatz

bezeichnet. Nach Montage eines Steins wird der nächste Datensatz der geöffneten build-Datei versendet. Wurde der letzte Datensatz übermittelt und der Stein gefügt, so wird der Befehl *Transcomp* übertragen. Dieser dient dem Zustandsautomaten der Softwarekomponente auf der FS100 als Endmarke, die signalisiert, dass alle Datensätze der zu montierenden Lego-Baugruppe übertragen wurden.

Die Anfrage *State?* dient dazu zur Überprüfung, ob ein übertragener Datensatz fertig bearbeitet wurde. Wie lange ein Montagevorgang eines Lego-Steins dauert, kann aufgrund unterschiedlicher Prozesszeiten vom Leitrechner aus nicht abgeschätzt werden. Nach Übertragen eines Datensatzes wird daher stetig der Zustand abgefragt. Erst wenn der Server signalisiert, dass der übertragene Datensatz abgearbeitet wurde, kann ein neuer Datensatz versendet werden. Die möglichen Antworten des Servers werden durch Tabelle 10-2 wiedergegeben.

Tabelle 10-2: Rückantworten des Servers (Quelle: Eigene Ausarbeitung).

Rückantwort	Intention
Emergency	Notaus oder Motorschutzschalter wurde ausgelöst
EmergencyHandled	Notaus und Motorschutzschalter OK
InitReady	Initialisierungsvorgang Beendet
InProgress	Datensatz in Bearbeitung
DatasetProcessed	Datensatzbearbeitung Beendet
EndOfAssembly	Montage beendet
DoingBreak	Pause

Der Rückgabewert *Emergency* gibt Auskunft darüber, dass der Not-Aus betätigt ist oder dass die Freigabe nach Betätigung dessen noch nicht durchgeführt worden ist. Ein weiterer Grund für diese Antwort kann ein ausgelöster Motorschutzschalter sein. Wird dieser Zustand verlassen, so sendet der Server die Rückantwort *EmergencyHandled*. Not-Aus-Relais und Motorschutzschalter sind im Zustand, der für den Betrieb der Zelle erforderlich ist und führen somit ein 24 V-High-Signal, was in der bitweisen Verarbeitung der SPS einer logischen 1 entspricht. Nachdem der Initialisierungsvorgang beendet wurde, sendet der Server den Response *InitReady*. *InProgress* gibt sowohl darüber Auskunft, dass der Initialisierungsvorgang momentan durchgeführt wird, als auch, dass sich ein übertragener Datensatz momentan noch in Bearbeitung befindet. Nachdem dieser Datensatz erfolgreich abgearbeitet wurde, der Lego-Stein also gefügt wor-

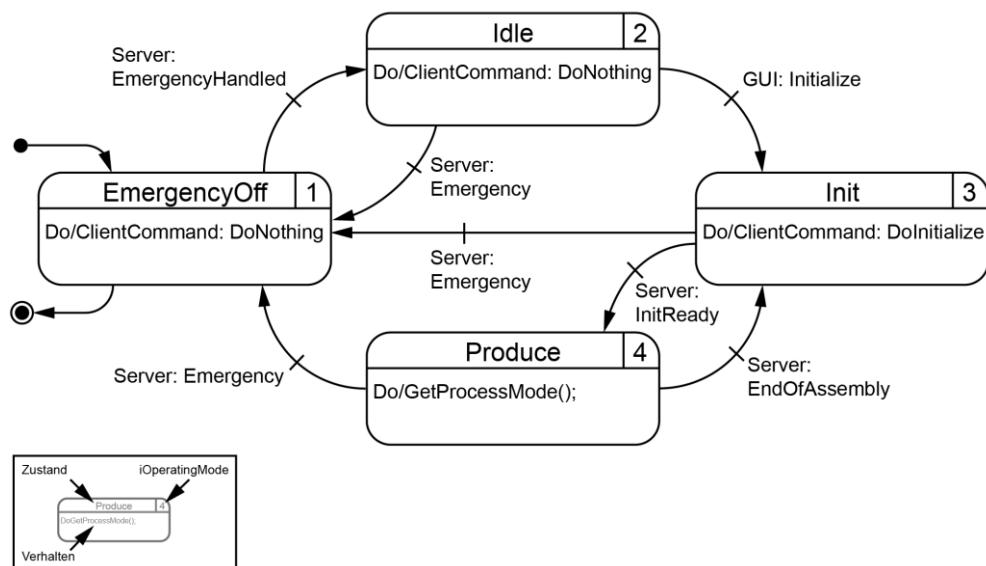
den ist, sendet der Server den Rückgabewert *DatasetProcessed*. Als Bestätigung der Endmarke *Transcomp*, die eine Anfrage des Clients ist, dient der Befehl *EndOfAssembly*. Wird als Rückgabe der String *DoingBreak* empfangen, so befindet sich die Roboterzelle im Pausen-Modus.

10.4.3 Operation- und Process-Handler

Im Folgenden werden die Zustandsautomaten der Klassen *OperatingHandler* und *ProcessHandler* vorgestellt. Zustände der Zustandsautomaten sind zur Verdeutlichung fett hervorgehoben. Übertragene Anfragen und Rückantworten sind kursiv dargestellt.

10.4.3.1 Der OperationHandler

Der Zustandsautomat des *OperationHandlers* dient dazu, die Grundzustände der Anlage modellieren zu können. Das Prinzip wird durch Abbildung 10-7 deutlich.



*Abbildung 10-7: Zustandsdiagramm des OperationHandlers
(Quelle: Eigene Ausarbeitung).*

Der *OperatingHandler* besteht aus vier Zuständen. Grundzustand ist der Zustand **EmergencyOff**. Dieser kann von jedem anderen Zustand des Automaten erreicht werden. Transition hierfür ist immer die Rückantwort *Emergency* seitens des Servers. Wird der Server aus dem Not-Aus-Zustand gelöst, so sendet dieser die Nachricht *EmergencyHandled*. Der Zustandsautomat geht in den Zustand **Idle** über. Dieser kann als Ruhezustand betrachtet werden. Der Name

röhrt daher, dass sich die Roboterzelle dabei in einem aus Clientsicht unbestimmten Zustand befindet. Das implementierte Zustandsverhalten der Zustände **EmergencyOff** und **Idle** ist beiderseits die Rückgabe der Clientanfrage *DoNothing*.

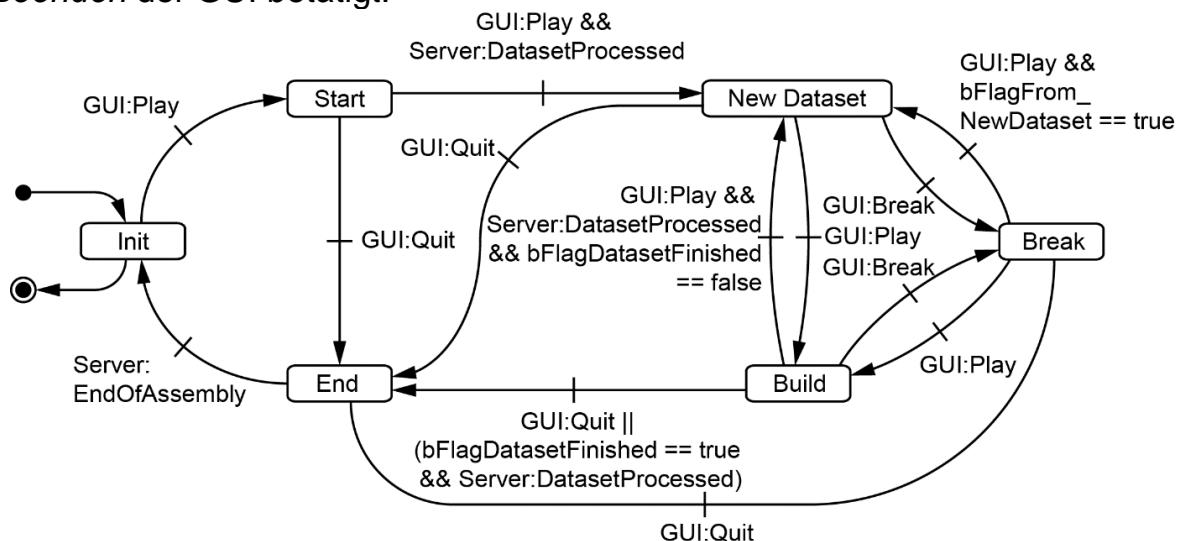
Befindet sich der Zustandsautomat im Zustand **Idle** so kann in den Zustand **Init** übergegangen werden, sofern der Button *Initialisieren* der GUI betätigt wird. Dieser Zustand kennzeichnet den Initialisierungsprozess der Roboterzelle. Ausgegebener Befehl des Clients ist *DoInitialize*, woraufhin die Roboterzelle die Initialisierung durchführt. Nach deren Abschluss sendet der Server die Rückantwort *InitReady*, wodurch der Zustand **Produce** erreicht wird. Diesem Zustand ist der Zustandsautomat der Klasse ProduceHandler untergeordnet. Nachdem die Montage einer Baugruppe vollständig abgeschlossen ist und der Server die Rückantwort *EndOfAssembly* sendet, geht der Zustandsautomat erneut in den Zustand **Init** über. Im Regelbetrieb werden die beiden Zustände **Init** und **Produce** somit zyklisch aufgerufen. Dieser Zyklus wird im Falle der Rückantwort *EmergencyOff* unterbrochen. Der Zustand des Automaten wird durch die Intervariable *iOperatingMode* wiedergegeben. Die Zahl in der rechten oberen Ecke eines jeden Zustands des Diagramms gibt an, wie Zustände des Automaten und der Inhalt der Variable *iOperatingMode* einander zuzuordnen sind. Entspricht die Variable beispielsweise dem Wert 3, so befindet sich der Zustandsautomat im Zustand **Init**.

10.4.3.2Der ProcessHandler

Der ProcessHandler ist, wie bereits dargestellt, dem OperationHandler hierarchisch untergeordnet und dient dazu, die Abläufe während des Zustands **Produce** modellieren und steuern zu können. Abbildung 10-7 zeigt das zugehörige Zustandsdiagramm. Das Zustandsverhalten eines jeden Zustands ist aus Übersichtsgründen nicht in der Abbildung enthalten. Diese werden stattdessen durch Tabelle 10-3 wiedergegeben. Hinsichtlich den in der Abbildung enthaltenen Transitionen sei angemerkt, dass diese sich an die Syntax der Programmiersprache C anlehnen. Eine logische Und-Verknüpfung wird durch zwei &-Zeichen symbolisiert. Oder-Verknüpfungen werden durch ||-Operator dargestellt.

Grundzustand des Zustandsautomaten ist der Zustand **Init**, in dem der Client die Anfrage *DoNothing* sendet. Wird der Button *Start* des GUI betätigt, so ist die Transition hin zum Zustand **Start** erfüllt, da so lange dieser Button aktiviert ist die Zeichenfolge *Play* in einem String zwischengespeichert wird. Während die-

ses Zustands wird dem Client der String *Start* als zu sendende Anfrage zugewiesen. Darüber hinaus wird die sich in der Warteschlange an erste Stelle befindliche build-Datei durch die Methode *OpenFile()* des *FileManager*s geöffnet. Als Rückantwort der vom Client gesendeten *Start*-Marke wird vom Server *DatasetProcessed* gesendet. Sobald diese Bestätigung empfangen und der *Play*-Button der GUI weiterhin aktiviert ist, wird der erste Datensatz übertragen. Diese Aktion ist Teil des Zustands **NewDataset**. Von diesem Zustand aus können in Abhängigkeit der erfüllten Transition die Zustände **Break**, **Build** und **End** erreicht werden. In letzteren wird übergegangen, sofern der User den Button *Beenden* der GUI betätigt.



*Abbildung 10-8: Das Zustandsdiagramm des in der Klasse *ProcessHandler* implementierten Zustandsautomaten (Quelle: Eigene Ausarbeitung).*

*Tabelle 10-3: Zustände, Zuordnung derer zur Variable *iProcess* sowie das Zustandsverhalten eines jeden Zustandes (Quelle: Eigene Ausarbeitung).*

Zustand	iProcess	Zustandsverhalten
Init	1	Do/ClientCommand: DoNothing;
Start	2	Do/ClientCommand: Start; Do/FileManager.OpenFile();
New Dataset	3	Do/ClientCommand: Dataset; Do/bFlagFromNewDataset = False;
Break	4	Do/ClientCommand: DoBreak;
Build	5	Do/ClientCommand: State?;
End	6	Do/ClientCommand: Transcomp; Do/FileManager.CloseFile();

Der Zustand **Break** wurde in den Zustandsautomaten aufgenommen, um während den Montagevorgang per GUI stoppen und bei Bedarf wieder fortsetzen zu können. Der notwendige Button zum Erreichen dieses Zustands ist momentan allerdings nicht implementiert. Da der Zustand **Break** von zwei unterschiedlichen Zuständen aus erreicht werden kann, muss festgehalten werden, aus welchem Zustand dieser erreicht wurde. Hierzu dient die boolsche Variable `bFlagFromNewDataset`. Sobald der Zustand **NewDataset** eintritt, wird dieser Variable der Wert `False` zugewiesen. Wird der Zustand **NewDataset** verlassen und der Zustand **Break** aktiviert, so wird diese Variable auf `True` gesetzt.

Nachdem ein Datensatz während des Zustands **NewDataset** übertragen worden ist, wird sofern der `Start`-Button weiterhin betätigt ist, direkt in den Zustand **Build** übergegangen. Dieser dient dazu, mittels der Client Anfrage `State?` stetig abzufragen, wie weit der Server mit der Verarbeitung des übertragenen Datensatzes ist. Wurde der Stein gefügt, so sendet der Server die Rückantwort `DatasetProcessed`. Um zurück zu Zustand **NewDataset** zu springen und den nächsten Datensatz übertragen zu können, muss neben dem weiterhin aktiven `Start`-Button der Variable `bFlagDatasetFinished` der Value `False` zugewiesen sein. Diese Variable ist Teil des `FileManagers`. Sobald dieser den letzten Datensatz ausgelesen hat, wird diese Variable auf `True` gesetzt. In diesem Fall wird in den Zustand **End** übergegangen und die Anfrage `Transcomp` gesendet.

Im Regelfall wird somit zwischen den Zuständen **NewDataset** und **Build** so lange hin und her gesprungen, bis alle Datensätze durch den Server verarbeitet worden sind. Anschließend wird in den **End**-Zustand übergegangen. Nach Rückgabe des Strings `EndOfAssembly` wird der **Init**-Zustand aktiviert. Dies wird vom `OperatorHandler` registriert, woraufhin dieser den **Produce**-Zustand verlässt und die notwendige Initialisierung durchführt.

10.5 Thread zur Umsetzung des CAN-Bus-Netzwerks

Wie schon im vorherigen Kapitel, wird im Rahmen der Ausführungen zum Thread zur Implementierung des CAN-Bus-Netzwerks einleitend der strukturelle Aufbau des Threads dargelegt. Im Anschluss daran werden die CAN-Bus-Library sowie deren elementare Funktionen erläutert. Da zur Umsetzung der CAN-Bus-Kommunikation ein zweites Projekt in das bestehende Visual-Studio-Projekt eingefügt werden muss, sind Ausführungen zur dafür notwendigen Vorgehensweise im Anhang zu finden.

10.5.1 Struktureller Aufbau des Threads

Wie in Abbildung 10-3 veranschaulicht, wird bei Aktivierung des Buttons *Connect* der *CanTransmissionThread* gestartet, welcher wiederum die Methode *ConnectCan()* startet. Die Struktur dieser Softwarekomponente wird durch das Klassendiagramm aus Abbildung 10-9 veranschaulicht.

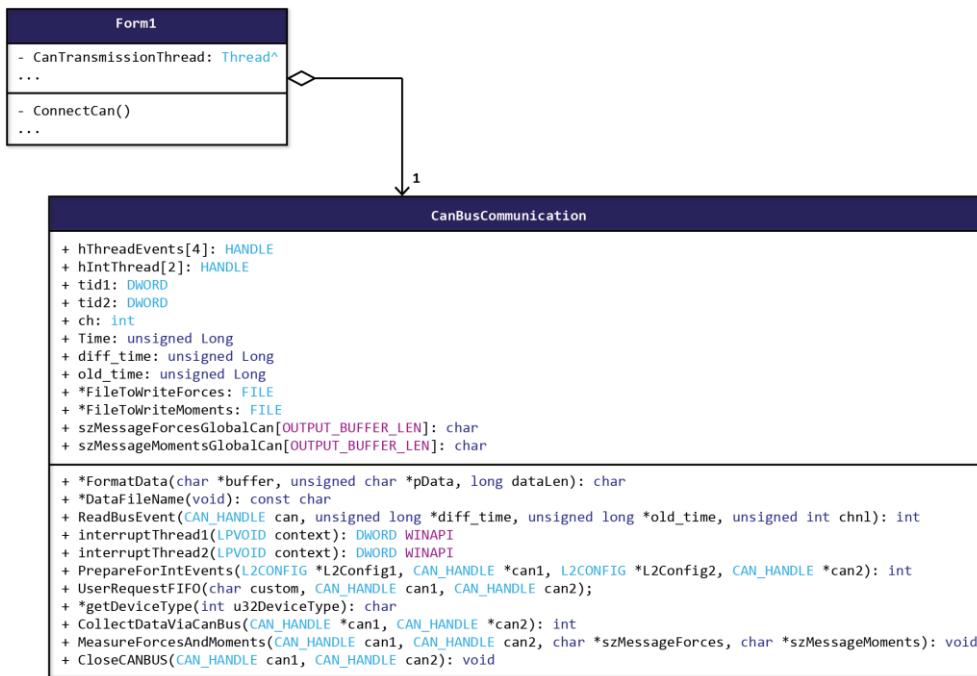


Abbildung 10-9: Auszug aus dem Klassendiagramm des Gesamtsystems mit Fokus auf den Thread zur CAN-Bus-Verwaltung (Quelle: Eigene Ausarbeitung).

Die Header-File *CanBusCommunication.h* gibt die in Programmiersprache C gehaltene Library zur Steuerung des CAN-Bus-Netzwerks wieder. Diese Library ist als zweites Projekt in die Visual Studio Projektmappe eingebunden. Die im Rahmen dieser Betrachtung relevanten Methoden finden sich am unteren Ende der Klasse *CanBusCommunication*. Mit Hilfe der Methode *CollectDataViaCanBus([...])* kann ein CAN-Bus-Netzwerk initialisiert werden. Rückgabewert der Methode *MeasureForcesAndMoments([...])* sind zwei Strings, die die aktuell gemessenen Kräfte und Momente am KMS beinhalten. Durch die Funktion *CloseCANBUS([...])* kann ein bestehendes CAN-Bus-Netzwerk terminiert werden. Die Logik hinter den einzelnen Methoden wird in Kapitel 10.5.2 ausführlich erläutert.

Das Klassendiagramm aus Abbildung 10-9 gibt die für die Funktion des Threads benötigten Attribute und Methoden wieder. Um die bei Aufruf der Methode *ConnectCan()* ablaufenden Prozesse veranschaulichen zu können, ist ein Aktivitätsdiagramm zu dieser Methode hilfreich. Siehe dazu Abbildung 10-10.

Realisierung der Softwarekomponente für den Leitrechner

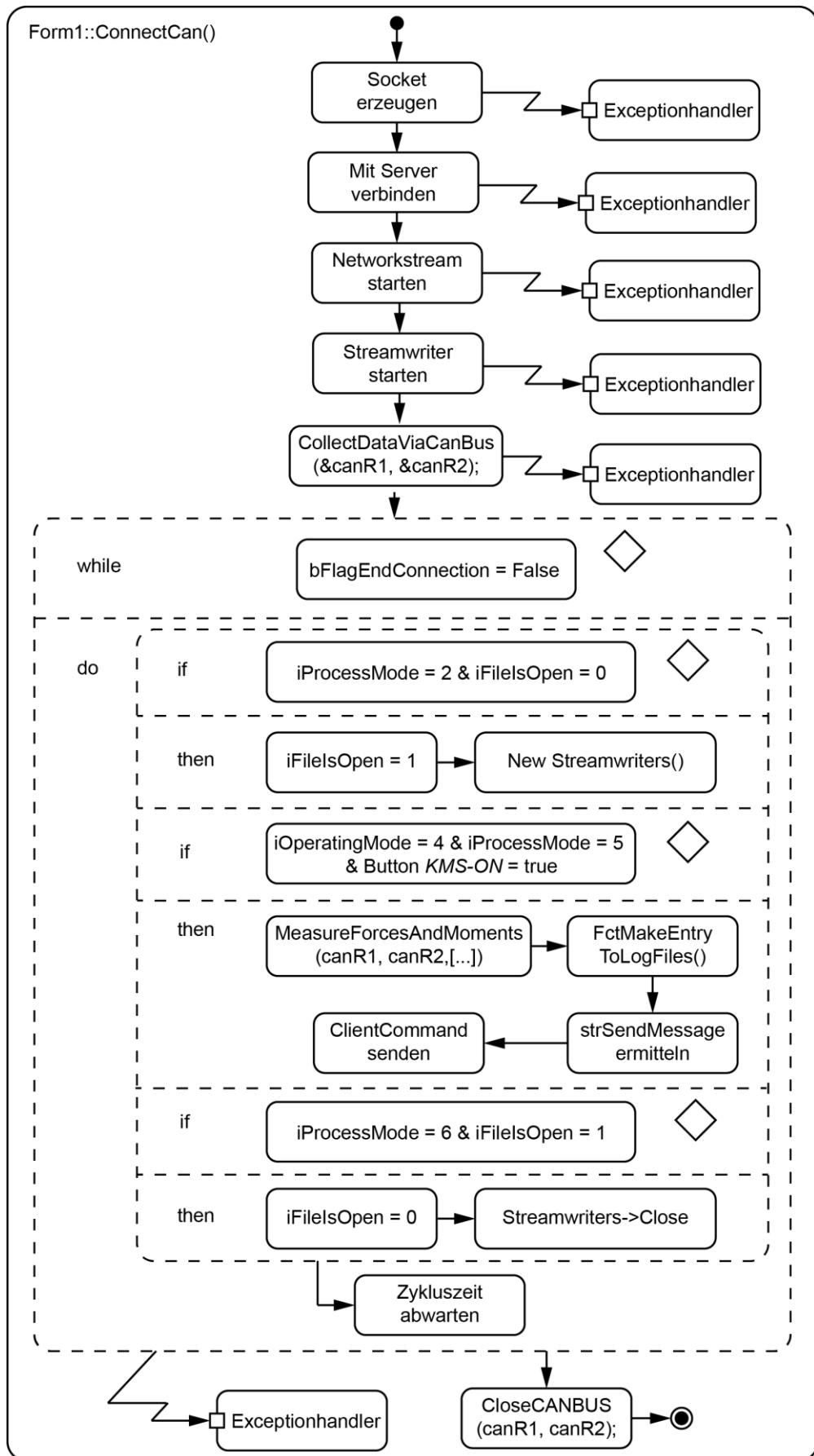


Abbildung 10-10: Aktivitätsdiagramm der Methode `ConnectCan()`
(Quelle: Eigene Ausarbeitung).

Analog zum Aufbau der TCP-IP-Verbindung zum Übertragen der Steuerungsbefehle muss auch hier einleitend ein Socket erzeugt werden, der mit dem Server verbunden werden muss. Darüber hinaus müssen Networkstream und Streamwriter gestartet werden. Im Anschluss daran muss das CAN-Bus-Netzwerk initialisiert werden. Dies übernimmt die Methode `CollectDataViaCanBus()` mit den Rückgabewerten `canR1` und `canR2`.

Nachdem das CAN-Bus-Netzwerk initialisiert ist, wird wie bei der Methode `ConnectTo()` eine while-Schleife betreten, die ebenfalls so lange durchlaufen wird, bis die boolsche Variable `bFlagEndConnection` den Wert False aufweist. Innerhalb der while-Schleife werden drei if-Anweisungen geprüft. Befindet sich der Zustandsautomat des ProcessHandlers (siehe Tabelle 10-3) im Zustand **Start** (`iProcessMode = 2`) so wird der Variable `iFileIsOpen` der Wert 1 zugewiesen. Darüber hinaus werden zwei Streamwriter erzeugt, mit deren Hilfe zwei Text-Dateien beschrieben werden. Diese dienen zum Speichern der gemessenen Kräfte und Momente auf dem Leitrechner. Die Variable `iFileIsOpen` dient als Rückgabewert, um überprüfen zu können, ob die Text-Dateien erfolgreich erstellt und geöffnet worden sind.

Befindet sich der Zustandsautomat des OperatorHandlers im **Produce**-Zustand (`iOperatingMode = 4`) und ist der Zustandsautomat des ProcessHandlers im Zustand 5 (= Build), so kann mit der Datenübertragung gestartet werden. Hintergedanke dieser Logik ist, dass die Übertragung erst erfolgen soll, sobald ein Montageprozess gestartet worden ist. Andernfalls würden irrelevante Daten direkt nach Beendigung der Initialisierung übertragen werden, die keinerlei Aussagekraft besitzen. Weitere Bedingung zum Eintritt dieser If-Klausel ist, dass der Switch-Button *KMS-ON* der GUI betätigt ist. Tritt diese Klausel ein, so wird einleitend die Methode `MeasureForcesAndMoments()` ausgeführt. Rückgabewert dieser Methode sind zwei Datenstrings im Hexadezimalsystem.

Diese enthalten die gemessenen Kräfte und Momente und werden mit Hilfe der Methode `FctMakeEntryToLogFile()` in Gleitkommazahlen übersetzt und in den beiden Text-Files gespeichert. Der Aufbau der Nutzdatenbytes der zurückgegebenen Can-Bus-Nachrichten sowie eine Erläuterung der erforderlichen Rechenschritte zur Umrechnung sind im nachfolgenden Unterkapitel zu finden.

Im Anschluss wird die zu sendende Nachricht an den Server erzeugt. Hierzu werden drei Identifier eingefügt. An die Nachricht vorangestellt wird „F-“. Dahinter folgt der String der gemessenen Kräfte. Durch das F kann identifiziert werden, dass im Anschluss der String `strFTCForces` folgt. Hinter diesen wiederum wird der Identifier „M-“ angefügt, welcher darüber Aufschluss gibt, dass danach

String strFTCMoments folgt. Als Identifier zum Kennlichmachen des Nachrichtenendes wird ein „X“ angehängt. Dieser zusammengesetzte String wird an den Server übertragen. Auf die Trennung von Kräften und Momenten in einzelne Nachrichten wurde verzichtet um Bandbreite einsparen zu können.

Die dritte If-Klausel behandelt den Fall, dass die Montage beendet ist. Der ProcessMode ist im Zustand **End**. Sofern zuvor die StreamWriter zum Speichern der Daten erfolgreich erzeugt worden sind, werden diese nun geschlossen.

Nachdem die If-Klauseln abgearbeitet worden sind, wird die per GUI eingestellte Zykluszeit zur KMS-Datenübertragung abgewartet. Im Anschluss wird der Zyklus erneut durchlaufen.

10.5.2 Methoden der CAN-Bus-Library

Hinsichtlich den Methoden der CAN-Bus-Library sei an dieser Stelle auf SOFTING INDUSTRIAL AUTOMATION GMBH (2012B) verwiesen. Die Abläufe zum Initialisieren eines CAN-Bus-Netzwerks sind darin hinreichend dokumentiert. Im Rahmen dieser Arbeit wird daher lediglich auf die Methode MeasureForcesAndMoments() eingegangen.

An dieser Stelle sei darüber hinaus angemerkt, dass die Funktionen des API der Softing Industrial Automation GmbH in der Dynamic Link Library „canL2.dll“ implementiert sind. Die Methoden der CAN-Bus-Library finden sich in komplizierter Form in der Datei „CAN-Bus-Kommunikation-Lib 1.0.lib“. Beide Dateien müssen stets im selben Verzeichnis der Execute-Datei der Softwarekomponente für den Leitrechner auffindbar sein.

Die Methode MeasureForcesAndMoments ruft die Methode UserRequestFIFO auf. Rückgabe dieser Methode ist bei Übergabe des Parameters 't' ein String, der die aktuell gemessenen Kräfte enthält. Wird der Funktion als Eingangsparameter ein 'm' übergeben, so liefert die Methode die auftretenden Momente als Rückgabewert.

Bei Aufruf der Methode UserRequestFIFO mit Parametrisierung zur Rückgabe der Kräfte, wird der Datenbefehl 48 per CAN-Bus an den KMS versendet. Als Antwort sendet dieser eine Nachricht, die beispielsweise die in Abbildung 10-11 dargestellten Nutzdatenbytes enthält.

Als Identifier auf Befehl 48 enthält die Antwort an erster Stelle ein Bestätigungs-Byte mit dem Inhalt 49. Darauf folgen die Datenbytes der Kräfte in X-, Y- und Z-Richtung. Bei erfolgreicher Messung folgt am Ende der Nachricht eine 0 zur Bestätigung. Jeweils zwei Bytes sind einer Messrichtung zugeordnet. Die Daten

werden in der Byte-Reihenfolge Little Endian abgespeichert. Dies bedeutet, dass das niederwertigere Byte auf in der niedrigeren Speicheradresse gespeichert wird. Der dargestellte String wird von der Funktion zurückgegeben und in dieser Form an die Robotersteuerung durch die Methode ConnectCan übertragen. Die einzelnen Nutzdatenbytes werden innerhalb des Strings durch Bindestriche getrennt.

CAN-Nachricht:	Bestätigung 49	Fx e7	Fy ff	Fz ab	ff	39	3	Status-Byte 0
Hex-Zahl:		ffe7		ffab		0339		
Dezimalzahl:		65511		65451		825		
Vorzeichen:		-25		-85		825		
Normieren [N]:		-0,7813		2,6563		25,7813		

Abbildung 10-11: Übertragene Nutzdatenbytes und daran anschließende Berechnungen (Quelle: Eigene Ausarbeitung).

Um die Kraft in Newton zu erhalten, muss zunächst das Hexadezimalformat erzeugt werden. Hierzu müssen die beiden Bytes in ihrer Reihenfolge vertauscht und aneinandergefügt werden. Im Anschluss kann die Zahl in das Dezimalsystem umgerechnet werden. Ist die Dezimalzahl größer als 2^{15} , so muss die Zahl 2^{16} von der Dezimalzahl subtrahiert werden. Dies ist eine Folge der Tatsache, dass im hexadezimalcodierten Datensatz kein Vorzeichen dargestellt werden kann. Aus der Dezimalzahl 65511 beispielsweise wird somit -25. Die errechneten Integer-Zahlen müssen im Anschluss daran noch normiert werden. Zur Berechnung der gemessenen Kraft in Newton müssen die Ergebnisse daher noch durch 32 dividiert werden.

Der Datenbefehl zum Anfordern der Momente lautet 4a, die dazugehörige Bestätigung 4b. Die Umrechnung der Daten in Gleitkommazahlen erfolgt analog zu den Berechnungen der Kräfte. Divisor zum Errechnen der Kraft in Nm ist 1024.

11 Realisierung der Softwarekomponente für die FS100

Hinsichtlich der Realisierung der Software für die Robotersteuerung FS100 wird einleitend die Entwicklungsumgebung MotoPlusSDK vorgestellt. Im Anschluss daran wird die Architektur der entwickelten Applikation in Kapitel 11.2 erläutert. Diese besteht aus drei parallel ablaufenden Tasks. Der Task zur Übermittlung der Befehlsdaten findet sich in Kapitel 11.3. Auf Grundlage der übermittelten Daten wird der Roboter gesteuert. Die Ausführungen hierzu sind in Kapitel 11.4 zu finden. Parallel zu diesen Aufgaben müssen die vom Client per TCP-IP versendeten Sensor-Daten des KMS empfangen und verarbeitet werden. Hierzu dient der in Kapitel 11.5 erläuterte Task.

11.1 Die Entwicklungsumgebung MotoPlusSDK

Im Rahmen dieses Abschnitts wird einleitend das Funktionsprinzip des MotoPlusSDK erläutert. Darüber hinaus wird in Kapitel 11.1.2 behandelt, welche APIs zum Entwickeln von Applikationen zur Verfügung stehen und wie diese debuggt werden können.

11.1.1 Funktionsweise

Das MotoPlusSDK ist eine von Yaskawa Motoman zur Verfügung gestellte Entwicklungsumgebung. Mit diesem SDK können Applikationen für die Robotersteuerung FS100 in der Programmiersprache C entwickelt werden.

Die kompilierten out-Dateien die mit Hilfe des SDKs erzeugt werden, können per USB-Stick auf die FS100 übertragen werden. Hierzu muss die Datei im Maintanance-Modus vom USB-Stick geladen werden. Bei Neustart der FS100 wird die out-Datei vom Bootloader gestartet und ausgeführt.

Wird von der entwickelten Applikation aus auch auf die Servo-Motoren des Roboters zugegriffen, so muss sich der Schlüsselschalter des PHGs in der Remote-Stellung befinden.

11.1.2 Entwickeln und Debuggen mit MotoPlusSDK

Das MotoPlusSDK bietet Zugriff auf eine Reihe von APIs. Mit deren Hilfe können beispielsweise Sensordaten ausgelesen oder der Roboter bewegt werden. Für

die im Rahmen dieser Arbeit entwickelte Applikation sind die Interfaces Task Control API, Network API und Motion Control API von Bedeutung. Aufgrund notwendiger Koordinatentransformationen findet darüber hinaus das Kinematics API Anwendung. Bezuglich weiterer Ausführungen zu den verfügbaren APIs sei an dieser Stelle auf YASKAWA ELECTRIC CORPORATION (2012B) verwiesen. Erläuterungen zu den Funktionen eines jeden Interfaces sind in YASKAWA ELECTRIC CORPORATION (2012A) zu finden.

Mit Hilfe des MotoPlusSDKs können die Applikationen lediglich programmiert und kompiliert werden. Eine Fehlersuche, wie sie beispielsweise bei der Entwicklung von Programmen mit Visual-Studio möglich ist, wird vom MotoPlusSDK nicht angeboten. Debugging wird stattdessen durch Telnet ermöglicht. Hierzu muss die kompilierte Datei auf die Robotersteuerung übertragen werden. Die kritischen Passagen im Code müssen mit printf-Befehlen versehen sein. Mit Hilfe derer können Systemzustände im Telnet-Fenster des MotoPlusSDK angezeigt werden. Nach dem Neustart der Robotersteuerung wird dazu Telnet gestartet. Zuvor müssen in den Telnet-Einstellungen allerdings die IP-Adresse der FS100, Username und Passwort spezifiziert werden. Zum Debuggen via Telnet lauten sowohl Username als auch Passwort MOTOMANrobot.

11.2 Architektur des Servers

Der Server besteht aus drei parallel ablaufenden Task. Welche Aufgaben den einzelnen Tasks zukommen, wird im Folgenden kurz erläutert. Da zwei dieser Tasks auf gleiche Speicherbereiche zugreifen, muss der Prozess synchronisiert werden. Kapitel 11.2.2 veranschaulicht das dahinter stehende Prinzip.

11.2.1 Aufgabenzuordnung der einzelnen Tasks

Bei Start der FS100 werden drei parallel ablaufende Tasks gestartet. Der Task mpTaskCommunication dient zur Kommunikation mit dem Client, um die Anfragen dessen per Ethernet entgegenzunehmen und eine Rückantwort senden zu können. Der zweite Task, mpTaskRobotControl übernimmt die Steuerung des Roboters auf Grundlage der vom Client ausgesendeten Befehle. Der Empfang der vom Client gesendeten Sensordaten des KMS und das Abspeichern dieser Daten in einer globalen Variable wird durch den dritten Task, mpTaskCanCommunication erledigt.

11.2.2 Umsetzung der Prozesssynchronisation

Die Tasks `mpTaskCommunication` und `mpTaskRobotControl` greifen teilweise auf dieselben globalen Variablen zu. Da beide Tasks parallel ausgeführt werden, kann es sein, dass gleichzeitig auf die gleichen Speicherbereiche lesend oder schreibend zugegriffen wird. Der Zugriff auf einen Speicherbereich darf allerdings immer nur exklusiv durch einen Task erfolgen, da andernfalls fehlerhafte Ergebnisse oder inkonsistente Speicherzustände auftreten können.

Um Multitaskprozesse synchronisieren zu können, wird im Allgemeinen mit Semaphoren gearbeitet. Semaphoren stellen prinzipiell Integer-Variablen dar. Mit Hilfe derer können die Prozesse gegenseitig Freigaben erteilen oder Ressourcen reservieren. Wird eine Freigabe erteilt, so erhöht sich der Wert der Semaphore um 1. Innerhalb des MotoPlusSDK steht hierfür die Funktion `mpSemGive()` zur Verfügung. Zur Reservierung einer Ressource dient die Funktion `mpSemTake()`. Dabei wird der Wert der Semaphore um 1 verringert.

Die Steuerung der beiden Tasks erfolgt in der entwickelten Applikation mit Hilfe der beiden Variablen `semid1` und `semid2` vom Datentyp `SEM_ID`. Die Funktionsweise wird durch das in Abbildung 11-1 dargestellte Ablaufdiagramm deutlich

Herzstück beider Tasks ist je eine Schleife, die erst beim Beenden der FS100 wieder verlassen wird. Vor Beginn dieser Schleife wird im Task `mpTaskCommunication` SemID 2 freigegeben. Erst nach Freigabe derer kann die Schleife des Tasks `mpTaskRobotControl` starten. Einleitend werden in diesem Clones der gemeinsam benötigten Variablen angelegt. Im Anschluss wird SemID 1 freigegeben. `mpTaskCommunication` wartet auf die Freigabe durch den Task `mpTaskRobotControl` und wird im Anschluss fortgesetzt. Parallel dazu kann die Prozessverarbeitung des Tasks `mpTaskRobotControl` erfolgen, da diese auf Basis der Clone-Variablen und nicht auf die gemeinsamen Speicherbereiche beider Tasks zugreift. Hat der Server die Rückantwort an den Client gesendet, so wird SemID 2 durch den Task zur Befehls-Datenübertragung wieder freigegeben. Dieser wird im Anschluss daran für 50 ms in Ruhe versetzt. Der Task zur Robotersteuerung kann nach Freigabe von SemID2 die Rückgabewerte nach Prozessverarbeitung in den gemeinsam genutzten globalen Variablen abspeichern.

Zusammenfassend lässt sich der Vorgang zur Prozesssynchronisation wie folgt beschreiben: Der Task zur Kommunikation mit dem Client erteilt dem Task zur Steuerung der Zelle zyklisch die nötige Semaphore SemID 2, um die gemeinsam genutzten Variablen lesen, bzw. nach Verarbeitung beschreiben zu kön-

nen. Währenddessen ist durch Task `mpTaskRobotControl` mittels SemID 1 sichergestellt, dass Task `mpTaskCommunication` nicht auf die diese Speicherbereiche zugreifen kann.

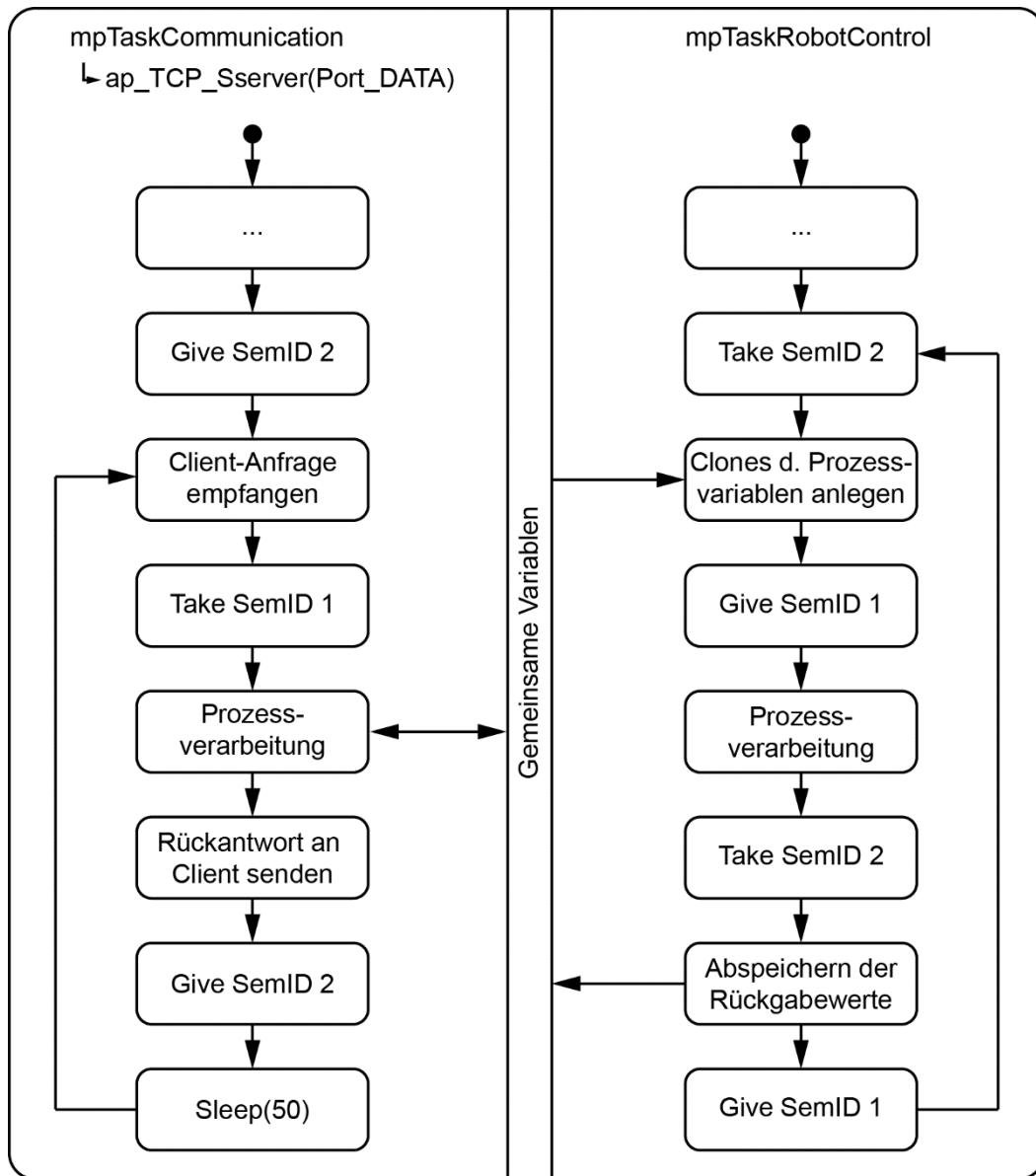


Abbildung 11-1: Ablaufdiagramm zur Veranschaulichung der Prozesssynchro-nisation mittels Semaphoren (Quelle: Eigene Ausarbeitung).

11.3 Der Task zur Befehls-Datenübertragung via Ethernet

Innerhalb dieses Abschnitts finden sich einleitend Ausführungen zum Ablauf der TCP-IP-Kommunikation. Diese werden mittels eines Ablaufdiagramms verdeutlicht. Die zwei implementierten und bereits angedeuteten Zustandsautomaten werden in Kapitel 11.3.2 beschrieben.

11.3.1 TCP-IP-Kommunikation

Die ablaufenden Prozesse zur TCP-IP-Kommunikation sind in Abbildung 11-2 dargestellt.

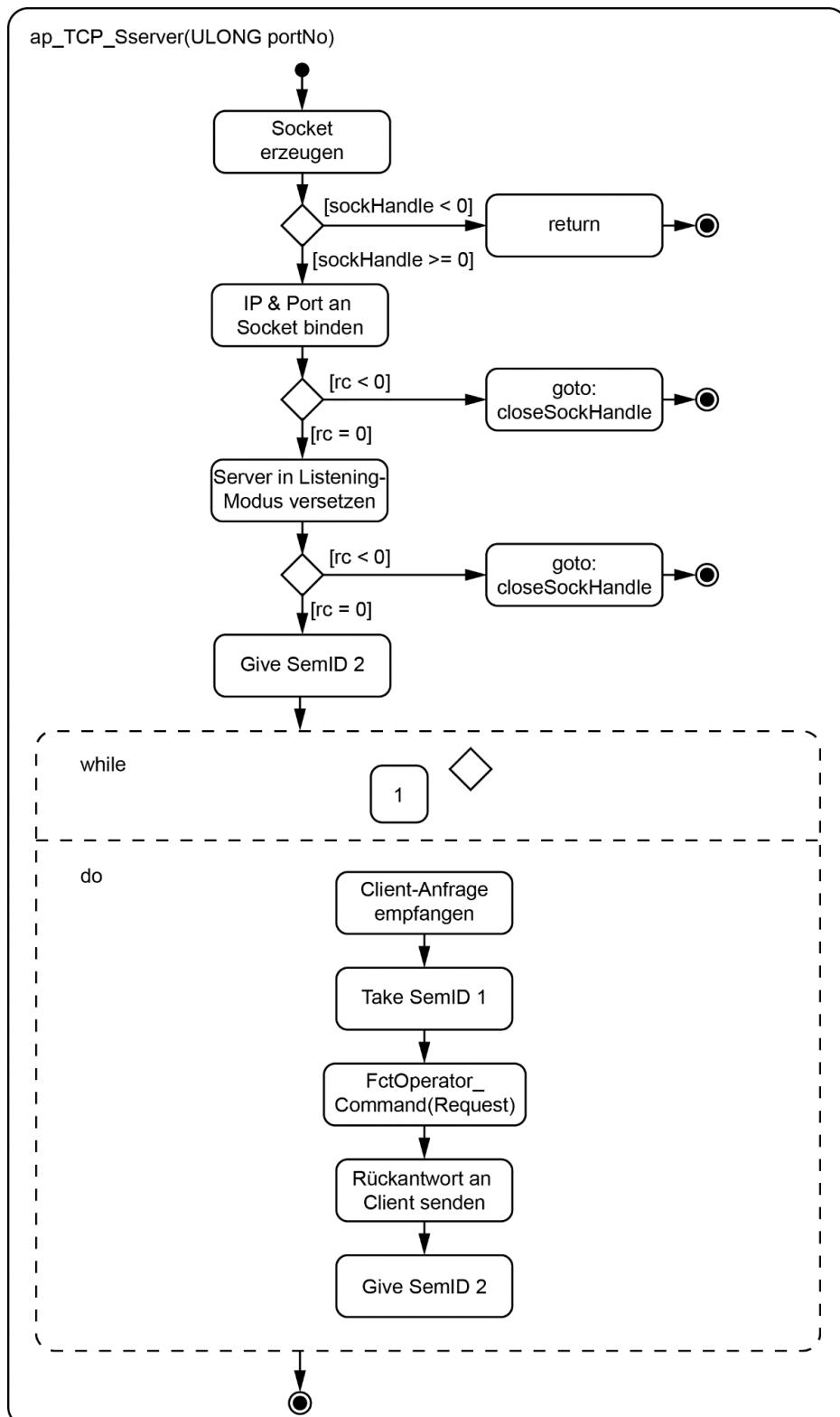


Abbildung 11-2: Ablaufdiagramm des Tasks `mpTaskCommunication`
(Quelle: Eigene Ausarbeitung).

Der Task ruft nach Start die Funktion ap_TCP_Sserver auf. Durch das Aktivitätsdiagramm wird deutlich, dass einleitend ein Socket erzeugt werden muss. Da in der Programmiersprache C kein Exception-Handler implementiert ist, mit dessen Hilfe nicht erfolgreiche Funktionsaufrufe abgefangen werden können, wird nach jedem Funktionsaufruf der Rückgabewert abgefragt. Ist dieser kleiner Null, so wird zu einem hierfür programmierten Abschnitt gesprungen.

Nachdem der Socket erzeugt wurde, werden IP-Adresse und Port-Nummer an den Socket gebunden. Im Anschluss wird der Server in den Listening-Modus versetzt. Hierbei wartet der Server auf den Verbindungsaufbau durch den Client. Sobald die Verbindung besteht wird die Semaphore SemID 2 freigegeben.

Innerhalb der darauf folgenden while-Schleife wird einleitend die Anfrage des Clients empfangen und in einem Zwischenspeicher abgespeichert. Nach Freigabe durch den Task zur Robotersteuerung wird dieser Befehl verarbeitet. Dies erfolgt durch zwei Zustandsautomaten, deren Aufbau ähnlich zu denen des Clients ist. Die beiden Automaten sind ebenfalls hierarchisch aufgebaut, weshalb innerhalb der Funktion ap_TCP_Sserver zur Verarbeitung der Anfrage des Clients lediglich die Funktion FctOperatorCommand() aufgerufen wird. Die durch diese Funktion zurückgegeben Antwort wird im Anschluss an den Client gesandt. Danach erfolgt die Freigabe von Sem ID2.

11.3.2 Operator- und Process-Command

Der in der Funktion OperatorCommand implementierte Zustandsautomat wird durch Abbildung 11-3 dargestellt. Dieser Zustandsautomat ist mit den Zustandsautomaten des Clients gekoppelt, da diese miteinander kommunizieren. Siehe dazu Abschnitt 10.4.3. Die Kommunikation zwischen Client und Server, wird durch die Zustandsautomaten gesteuert. Ein Beispiel zur Verdeutlichung derer ist in Kapitel 11.3.3 zu finden.

Grundzustand des Automaten ist der Zustand **EmergencyOff**. Dieser kann von allen anderen Zuständen aus erreicht werden, sobald entweder der Not-Aus betätigt oder einer der Motorschutzschalter auslöst. Dies kann durch die I/O-Register 20030 und 20031 der internen SPS der FS100 abgefragt werden. Führen beide Register High-Signal, so ist die Transition hin zum Zustand **Idle** erfüllt. Rückgabe der Funktion ist der Bereits in 10.4.2 angesprochene Befehl **EmergencyHandled**. Wird im Folgenden die Anfrage **DoInitialize** durch den Client versendet, so wird der Zustand **Init** aktiviert. So lange dieser Zustand erhalten bleibt, sendet der Client den Rückgabewert **InProgress**. Nachdem der durch

den Task `mpTaskRobotControl` ausgeführte Initialisierungsprozess abgeschlossen ist, weist dieser der Variable `iFlagInitReady` den Wert 1 zu. Damit ist die Transition zum Zustand **InitReady** erfüllt. Der Rückgabewert lautet `InitReady`.

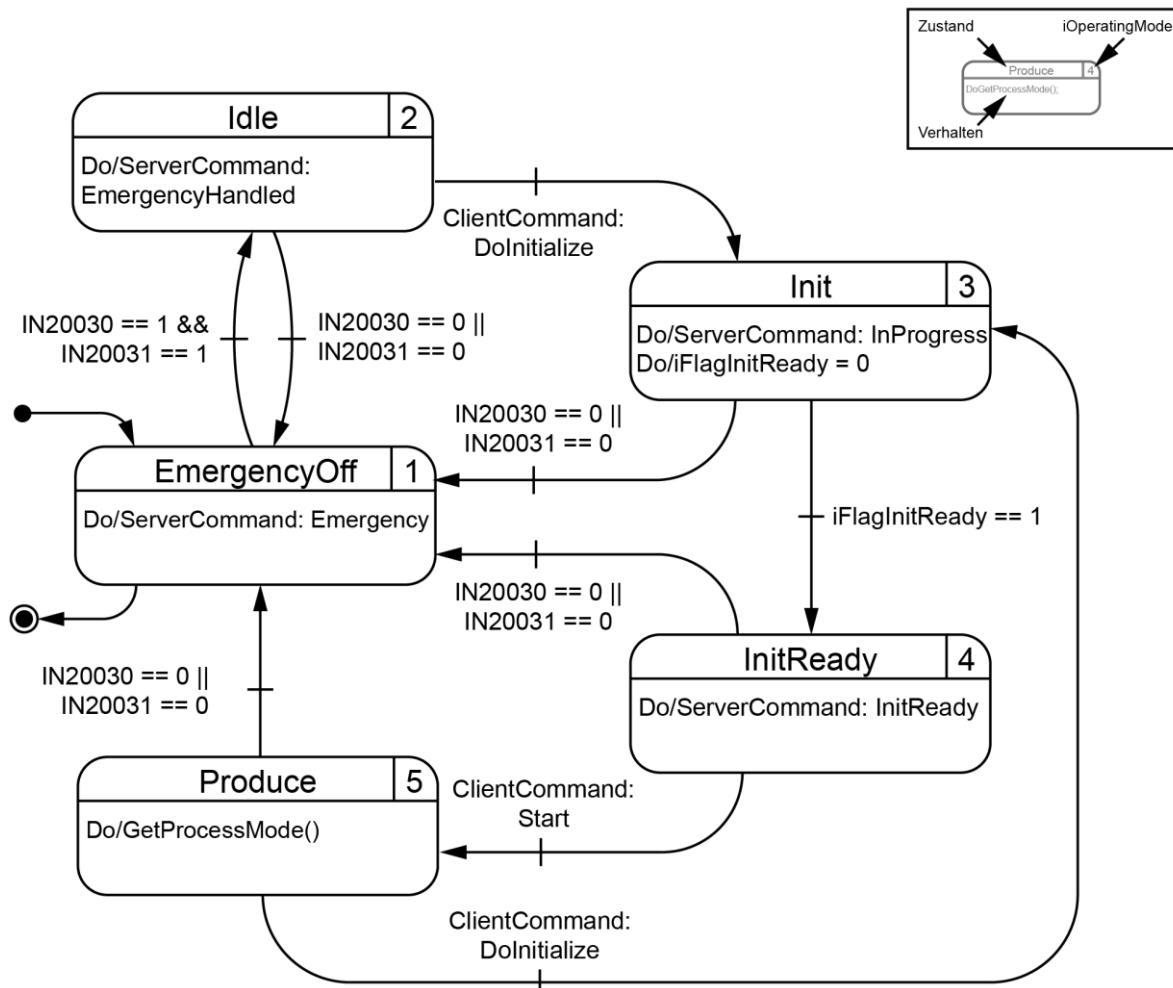


Abbildung 11-3: Zustandsdiagramm des Zustandsautomaten der Funktion `OperatorCommand` (Quelle: Eigene Ausarbeitung).

Nach erfolgreicher Initialisierung wird durch Betätigen des `Start`-Buttons auf der Softwarekomponente des Leitrechners die Anfrage `Start` an den Server gesandt. Der Zustand **Produce** wird aktiviert. Diesem Zustand ist ein zweiter, in der Funktion `FctProcessCommand` implementierte Zustandsautomat hierarchisch untergeordnet, der beim Aktivieren des Zustands **Produce** aufgerufen und abgearbeitet wird. Vom Zustand **Produce** aus kann bei Eingang der Anfrage `DoInitialize` in den Zustand **Init** gesprungen werden. Nach Fertigstellung einer Lego-Baugruppe wird die Roboterzelle erneut initialisiert um definierte Systemzustände gewährleisten zu können. Auf die Vermessung der Koordinatensysteme kann dabei verzichtet werden. Im Regelbetrieb werden somit die

Zustände **Init**, **InitReady** und **Produce** abgearbeitet. Die angesprochene Funktion `FctProcessCommand` wird im Folgenden erläutert. Abbildung 11-4 veranschaulicht den implementierten Zustandsautomaten.

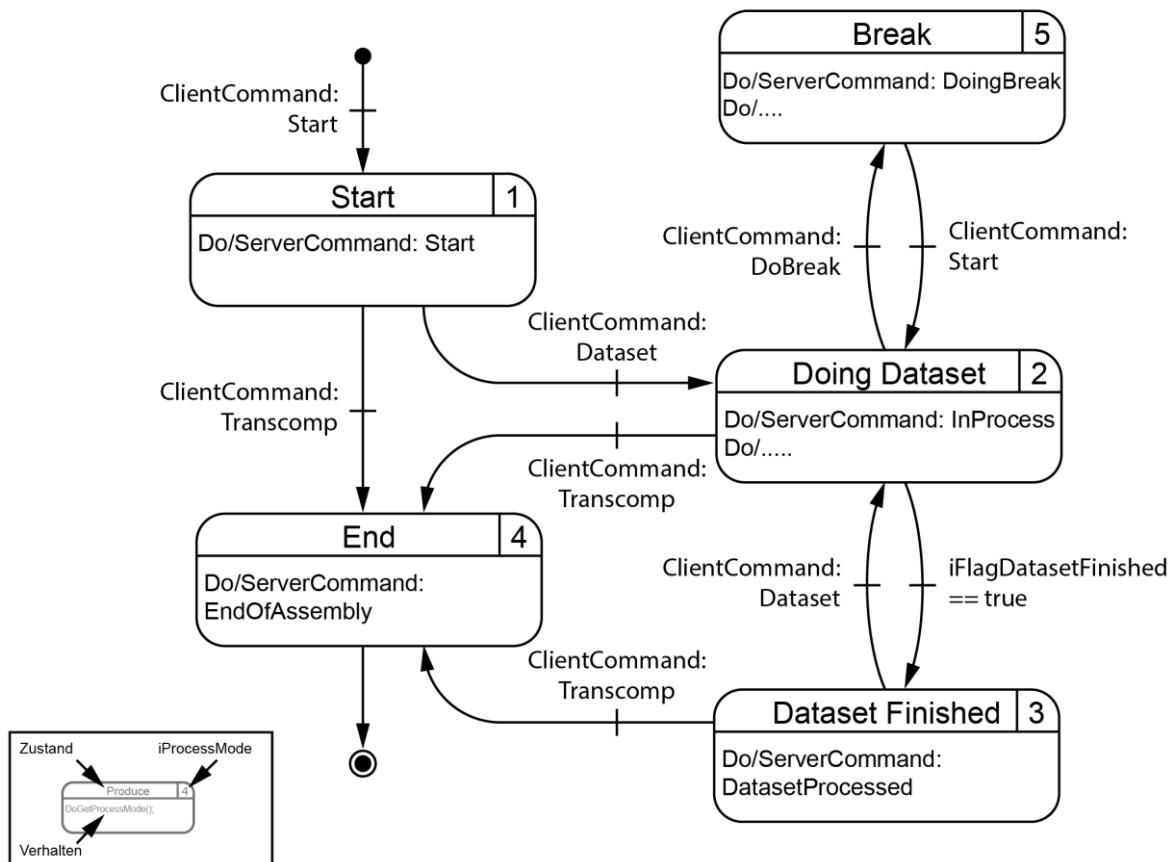


Abbildung 11-4: In der Funktion `FctProcessCommand` implementierter Zustandsautomat (Quelle: Eigene Ausarbeitung).

Nach Aufruf des Automaten ist der Zustand **Start** aktiviert. Der Rückgabewert des Servers lautet ebenfalls *Start*. Geht in diesem Zustand der Befehl *Transcomp* ein, der vom Client gesendet wird sobald alle Datensätze abgearbeitet wurden, so wird direkt in den **End**-Zustand gesprungen und der Zustandsautomat wird beendet. Geht ein BrickCAD-Datensatz als Client-Anfrage ein, so wird in den Zustand **Doing Dataset** gesprungen. Solange dieser Zustand aktiviert ist, werden die zur Montage des Lego-Steins notwendigen Abläufe durch den Task `mpTaskRobotControl` ausgeführt. Während dieser Zustand aktiviert ist, wird der String *InProcess* vom Server zurückgegeben.

Ist der Montageprozess abgeschlossen, so wird der Variable `iFlagDatasetFinished` der Wert 1 zugewiesen. Im Zustand **Dataset Finished** sendet der Server den Befehl *DatasetProcessed*. Geht nun ein neuer BrickCAD-Datensatz ein, so wird zurück in den Zustand **Doing Dataset** gesprungen und die Anfrage wird durch den Task `mpTaskRobotControl` abgearbeitet. Wurden

alle Datensätze übertragen oder der Button *Beenden* betätigt, so sendet der Client *Transcomp*. Die Transition hin zum Zustand **End** ist somit erfüllt. Der Server sendet *EndOfAssembly* und beendet den Zustandsautomaten der Produce-Funktion. Grundsätzlich kann vom Zustand **Doing Dataset** auch in den Zustand **Break** gesprungen werden. Die zur Steuerung der Zelle in diesem Zustand notwendigen Funktionen sind allerdings nicht implementiert.

11.3.3 Beispiel zur Client-Server-Kommunikation

Nachdem die Kommunikationsprinzipien und Befehlssätze von Client und Server bekannt sind, kann nun die Befehls-Datenübertragung an Hand eines Beispiels erläutert werden. In dem Sequenzdiagramm in Abbildung 11-5 werden User, Client und Server als Teilnehmer betrachtet. Für die Befehls-Datenübertragung irrelevante Threads und Tasks sind nicht dargestellt.

Einleitend werden Client und Server gestartet. Die Zustandsautomaten beider Teilnehmer befinden sich im Grundzustand **EmergencyOff**. Wird vom User nun der Button *Connect* des Windows-Forms betätigt, so wird die TCP-IP-Verbindung hergestellt. Der Client sendet die Anfrage *DoNothing*, die vom Server mit der Rückantwort *EmergencyOff* beantwortet wird. Löst der User den Not-Aus, so gibt der Server nach der nächsten Anfrage durch den Client den String *EmergencyHandled* zurück. Der Operation-Handler des Clients geht in den Zustand **Idle** über.

Wird in diesem Zustand vom User der Button *Initialisieren* betätigt, ist die Transition zum Zustand **DoInitialize** des Operation-Handlers erfüllt. Der Client sendet den Befehl *DoInitialize* an den Server, der bis zur Vollendung der Initialisierung die Rückantwort *InProgress* aussendet. Ist die Initialisierung abgeschlossen, lautet die Rückantwort *InitReady*. Der Operation-Handler des Clients geht in den Zustand **Produce** über. Wird anschließend durch den User der *Start*-Button gedrückt, sendet der Client die *Start*-Marke aus, die signalisiert, dass eine neue Lego-Baugruppe montiert werden soll. Nach deren Bestätigung durch den String *DatasetProcessed* wird der erste Datensatz versendet. Während dessen Bearbeitung lautet die Rückantwort des Servers *InProgress*. Ist diese abgeschlossen, lautet diese *DatasetProcessed* und der nächste Datensatz wird ausgesendet. Dies wird so lange vollzogen, bis alle Datensätze versendet und die Lego-Baugruppe vollendet ist. Danach wird die Roboterzelle erneut initialisiert und eine neue Baugruppe kann montiert werden.

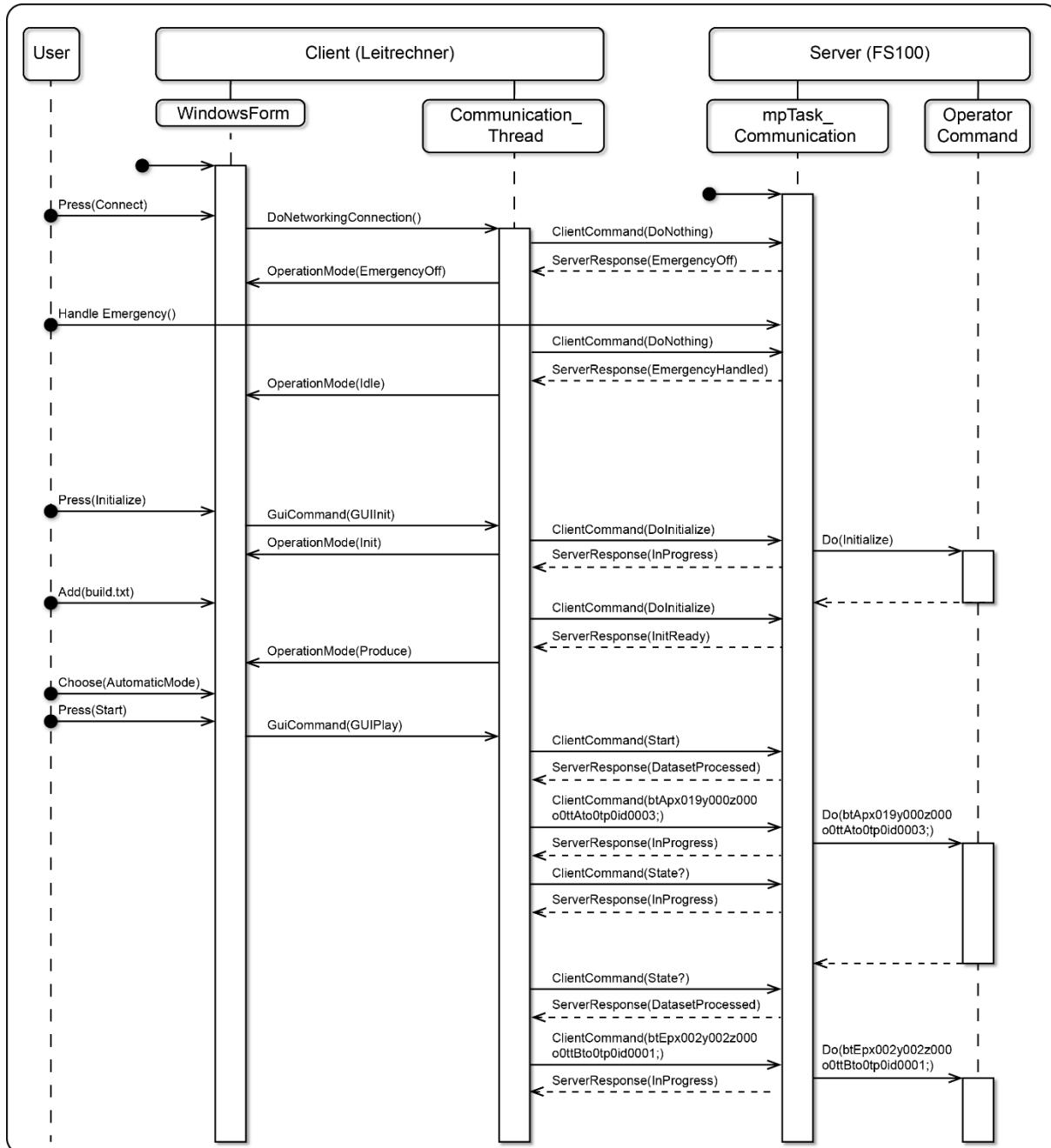


Abbildung 11-5: Sequenzdiagramm als Beispiel zur Client-Server-Kommunikation (Quelle: Eigene Ausarbeitung).

11.4 Der Task zur Steuerung des Roboters

Herzstück des Tasks `mpTaskRobotControl` sind die implementierten Funktionen zur Initialisierung der Roboterzelle und zur Verarbeitung eines übertragenen BrickCAD-Datensatzes. Bevor der Aufbau dieser beiden Funktionen in den Kapiteln 11.4.2 und 11.4.4 erläutert wird, folgt zunächst ein Kapitel zum strukturellen Aufbau des Tasks, um den Funktionsaufruf darlegen zu können.

11.4.1 Struktureller Aufbau des Tasks

Die Abläufe innerhalb des Tasks mpTaskRobotControl können durch ein Aktivitätsdiagramm dargestellt werden. Abbildung 11-6 zeigt dieses Diagramm. Aufgrund des selbsterklärenden Charakters der Darstellung wird an dieser Stelle auf eine vollständige Beschreibung der Abläufe verzichtet. Wesentliche Aufgabe dieses Tasks ist der Aufruf der Funktion FctDoInitialize sofern die Roboterzelle initialisiert werden muss. Zur Abarbeitung eines Datensatzes muss darüber hinaus die Funktion FcDoDataset ausgeführt werden.

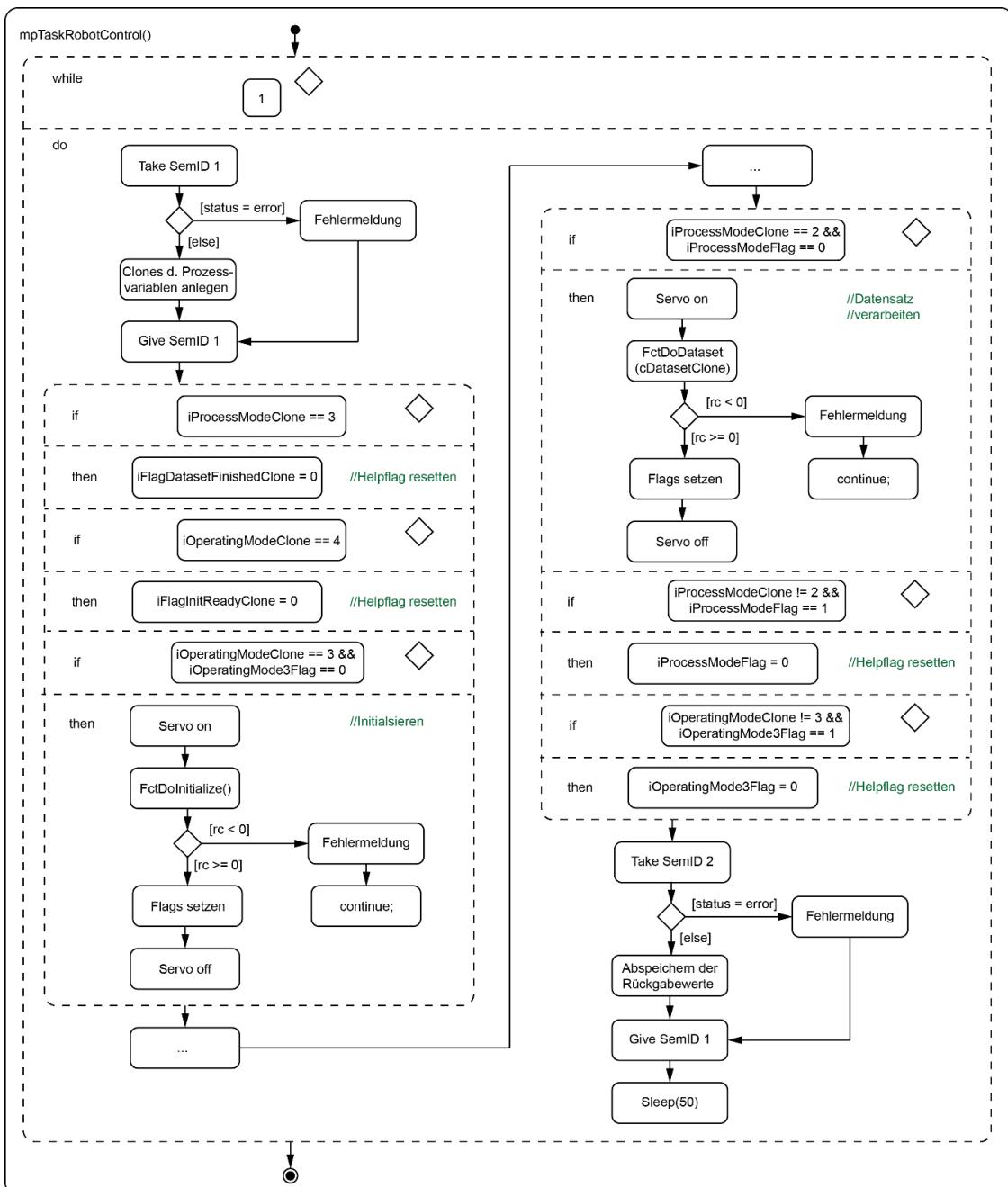


Abbildung 11-6: Ablaufdiagramm zum Task mpTaskRobotControl
(Quelle: Eigene Ausarbeitung).

Die Flag `iFlagDatasetFinishedClone` dient als Merker um anzuzeigen, ob ein vom Client übermittelter Datensatz bereits abgearbeitet worden ist. Ist dem so, nimmt der Process-Zustandsautomat den Zustand **DatasetFinished** ein. Die Flag kann zurückgesetzt werden. Eine ähnliche Funktion hat die Variable `iFlagInitReadyClone`. Diese dient als Merker zum Zustand der Initialisierung. Ist der Zustand **Produce** der Funktion `OperatorCommand` aktiv, kann auch diese Flag resettet werden. Mit Hilfe der Variablen `iProcessModeFlag` und `iOperatingMode3Flag` wird sichergestellt, dass solange ein Zustand des Zustandsautomaten besteht, jede Operation nur einmal durchlaufen wird.

Die Initialisierung der Roboterzelle wird durch die Funktion `FctDoInitialize` abgearbeitet. Bevor diese verwendet werden kann, müssen die Servomotoren des Roboters angeschaltet werden. Nach Beendigung der Initialisierung werden diese wieder abgeschaltet. Zur Verarbeitung eines BrickCAD-Datensatzes dient die Funktion `FctDoDataset`. Übergabeparameter ist der übertragene und in die Variable `cDatasetClone` kopierte Datensatz.

11.4.2 Initialisieren der Roboterzelle mittels `FctDoInitialize`

Die Funktion `FctDoInitialize` dient dazu, die Roboterzelle zu initialisieren. Im Rahmen der Initialisierung werden dazu alle Aktoren in eine definierte Lage gebracht. So wird beispielsweise der Greifer geöffnet und alle Zylinder der Zuführeinheit werden abgesenkt. Der Roboter wird zur Nullposition bewegt und das Förderband wird einmal durchgetaktet. Durch diese Maßnahmen wird gewährleistet, dass sich die Anlage in dem für den Betrieb notwendigen Zustand befindet.

Abbildung 11-7 veranschaulicht den durch die Funktion ausgeführten Prozess. Zum Erhalt der Übersicht dieses Ablaufdiagramms wird auf die Darstellung der Abfrage der Rückgabewerte aufgerufener Funktionen verzichtet.

Wird die Funktion `FctDoInitialize` aufgerufen, so wird als erstes die Schritt-kette zum Durchtakten des Förderbands gestartet. Hierzu wird die Funktion `FctPulseHelpFlagCycle` aufgerufen, die Register 10567 der SPS für 1 s High-Signal zuweist. Im Anschluss daran werden I/O-Signale, die die Greifer betreffen, eingelesen und zwischengespeichert. Wird die Roboterzelle nach einem Neustart zum ersten Mal initialisiert, so müssen die in Kapitel 9.3 erläuterten User-Koordinatensysteme vermessen und die zugehörige Transformationsmatrix bestimmt werden. Dies erfolgt durch die Funktion `FctDetermineFrame_Base_UserCosy()`. Deren Funktionsweise wird in Kapitel 11.4.3 beschrieben. Bevor diese Funktion aufgerufen wird, muss überprüft werden, ob

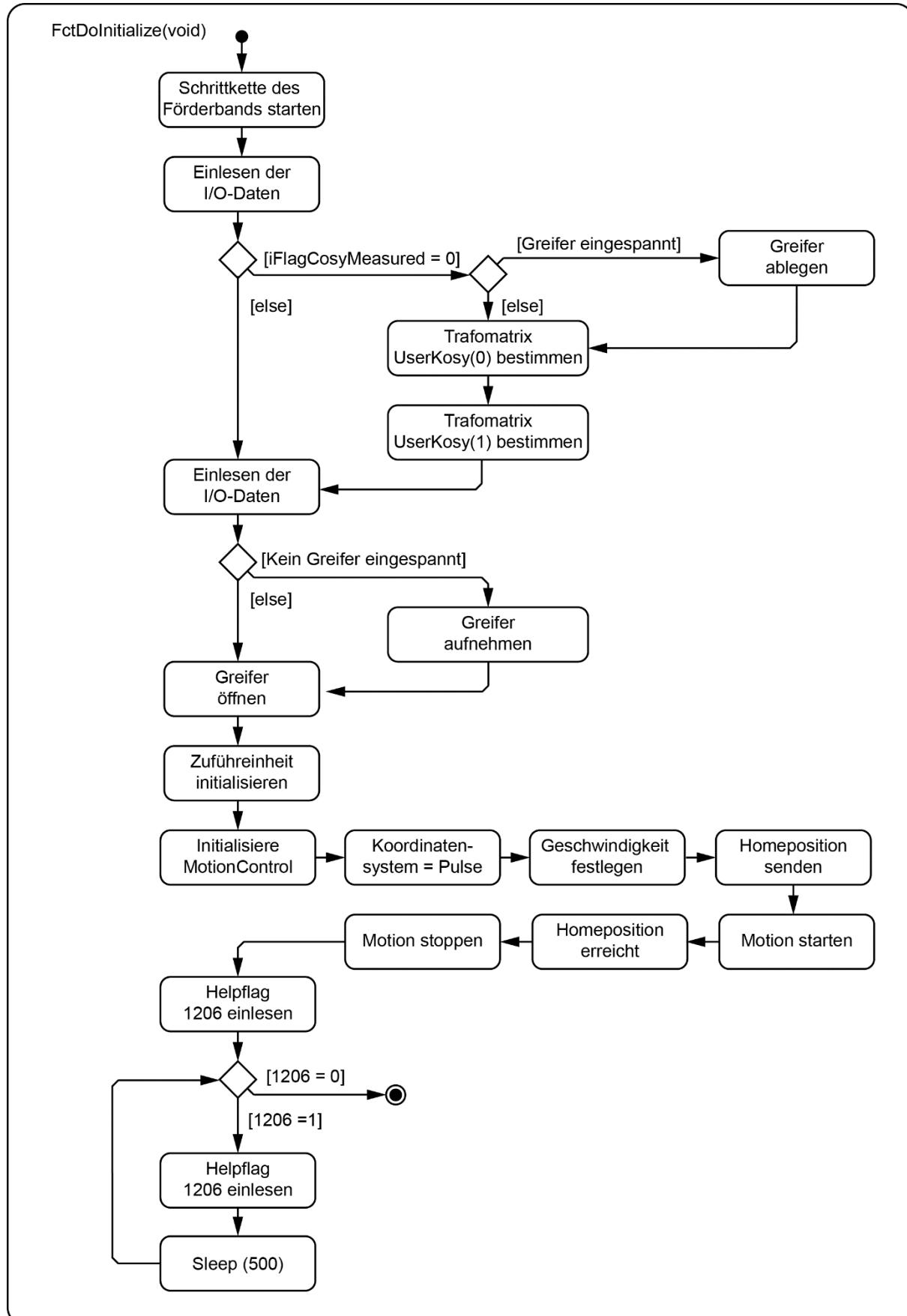
ein Greifer eingespannt ist. Falls dem so ist, muss dieser Greifer im Greiferbahnhof abgelegt werden. Zur Erkennung ob ein Greifer eingespannt ist dienen die in Tabelle 5-2 aufgelisteten Eingangssignale. Jedem Greifer ist ein Eingangssignal der SPS zugeordnet, deren Status abgefragt wird. Führt einer dieser Eingänge eine logische 1, so kann daraus ermittelt werden welche Job-Datei zur Greiferablage aufgerufen werden muss.

Bei einem Greiferwechsel wird der MPP3 zunächst von der aktuellen Position auf einer linearen Bahn zur Homeposition bewegt. Daher ist vor der Initialisierung zu überprüfen, ob diese Bahn ohne Kollision beschrieben werden kann. Ist dem nicht so, so muss der Roboter mit Hilfe des PHGs von dieser Position zu einem neutralen Punkt bewegt werden, bei dem sichergestellt ist, dass keine Kollision stattfinden kann.

Nach erfolgreicher Vermessung der Koordinatensysteme werden die I/O-Daten erneut eingelesen. Sofern kein Greifer eingespannt ist, wird vorzugsweise Greifer C eingewechselt. Hierzu wird entsprechend der auf der FS100 geteachte Job aufgerufen. Die Durchführung von Werkzeugwechselaufgaben mittels Job-Dateien findet sich in Kapitel 9.5. Steht Greifer C nicht zur Verfügung, wird Greifer B aufgenommen. Ist dies ebenfalls nicht möglich, so kommt Greifer A zum Einsatz. Der Greifer wird im Anschluss geöffnet. Als nächstes wird die Zuführeinheit initialisiert. Hierzu werden alle Zylinder der Einheit abgesenkt.

Da nach dem Start der Robotersteuerung der Roboter an einem beliebigen Punkt im Arbeitsraum steht, wird dieser in die Homeposition gebracht. Hierzu werden das Motion-Control-API initialisiert, das Koordinatensystem sowie die Geschwindigkeit festgelegt. Danach wird die Homeposition beschrieben. Die Abarbeitung dieses festgelegten Ziels wird mittels `mpMotStart` gestartet. Nachdem die Homeposition erreicht ist, wird die Operation gestoppt.

Im Anschluss daran muss überprüft werden, ob die Schrittkette zur Steuerung des Förderbands bereits abgearbeitet ist. Wenn dem so ist, führt Register 1206 der SPS Low-Signal. Andernfalls wird das Signal solange im Zeitabstand von 500 ms eingelesen, bis die Schrittkette abgearbeitet und die Initialisierung der Roboterzelle beendet ist.



*Abbildung 11-7: Aktivitätsdiagramm der Funktion FctDolInitialize
(Quelle: Eigene Ausarbeitung).*

11.4.3 Einmessen von User-Koordinatensystemen

Wie in Abschnitt 9.3 erläutert, sind auf der FS100 zwei User-Koordinatensysteme geteacht. MotoPlusSDK beinhaltet keine Funktion, mit deren Hilfe Informationen über die geteachten User-Koordinatensysteme ausgelesen werden können. Daher müssen diese Koordinatensysteme nach dem Neustart der Robotersteuerung eingemessen werden. Die Abläufe zur Bestimmung der homogenen Transformationsmatrix eines User-Koordinatensystems werden durch das, in Abbildung 11-8 dargestellte Aktivitätsdiagramm zusammengefasst.

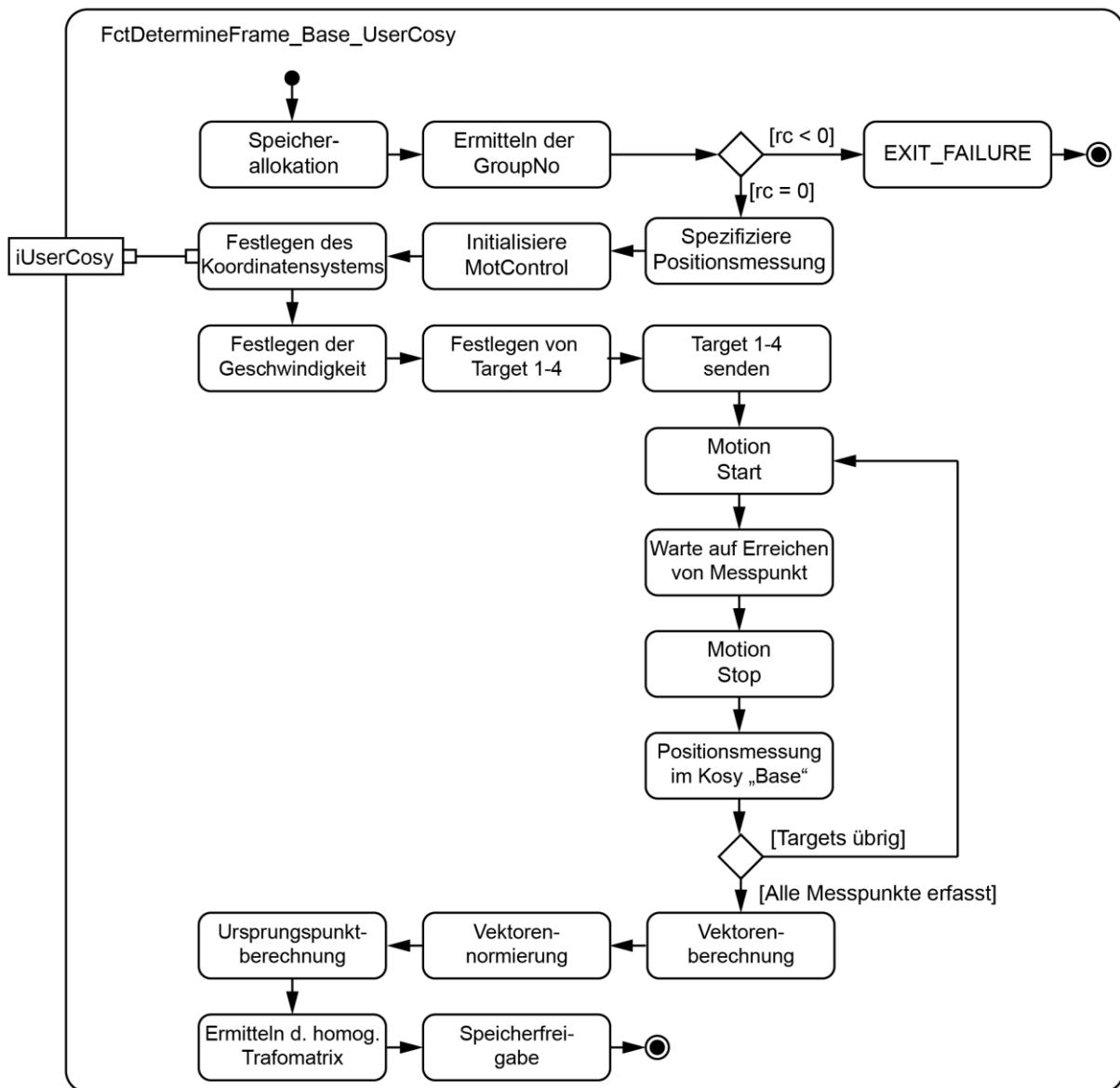


Abbildung 11-8: Aktivitätsdiagramm der Funktion FctDetermineFrame_Base_UserCosy (Quelle: Eigene Ausarbeitung).

Zur Verwendung dieser Koordinatensysteme muss eine Transformationsvorschrift bekannt sein. Mittels der Funktion FctDetermineFrame_Base_UserCosy

wird in Abhängigkeit des Eingabeparameters iUserCosy die homogene Transformationsmatrix eines User-Koordinatensystems ermittelt.

Das MotoPlusSDK stellt die Funktion `mpMakeFrame` zur Verfügung, mit deren Hilfe eine homogene Transformationsmatrix aus drei Messpunkten berechnet werden kann. Im Rahmen dieser Arbeit wurde einleitend mit dieser Funktion gearbeitet. Da für diese Funktion der Ursprungspunkt des Koordinatensystems als Parametereingabe benötigt wird, dieser ohne Wissen über die Orientierung des User-Koordinatensystems im Base-Koordinatensystem allerdings nicht hinreichend genau ermittelt werden kann, liefert die Funktion `mpMakeFrame` eine homogene Transformationsmatrix als Rückgabewert, mit der im Rahmen der Montage von Lego nicht gearbeitet werden kann. Die Abweichungen bei Koordinatentransformation unter Einsatz dieser Matrix betragen in der Realität bis zu 1400 µm, was eine zu große Abweichung darstellt. Um diesem Mangel entgegenzuwirken, ist die Funktion `FctDetermineFrame_Base_UserCosy` implementiert. Deren Funktionsweise wird im Folgenden erläutert.

Homogene Transformationsmatrizen sind im Allgemeinen wie folgt aufgebaut:

$$T_{i-1} = \begin{pmatrix} D_{i-1} & t_{i-1} \\ 0 & k_M \end{pmatrix} \in \mathbb{R}^{4 \times 4} \text{ mit} \quad (12)$$

D_{i-1} := Rotationsmatrix der Koordinatentransformation

t_{i-1} := Translationsvektor zwischen dem (i-1)-ten und i-ten Koordinatensystem im (i-1)-ten Koordinaten- system

k_M := Skalierungsfaktor

Der Skalierungsfaktor k_M ist bei maßstabsgereuer Abbildung zu $k_M = 1$ zu setzen, wovon bei Koordinatentransformationen in der Robotik ausgegangen werden kann. Dies erklärt, weshalb die homogenen Transformationsmatrizen innerhalb des MotoPlusSDK durch Structs des Typs `MP_FRAME` beschrieben werden, die aus vier Vektoren bestehen:

$$T_{AB} = \begin{pmatrix} n_x & o_x & a_x & P_x \\ n_y & o_y & a_y & P_y \\ n_z & o_z & a_z & P_z \end{pmatrix} \quad (13)$$

Die Rotationsmatrix wird durch den Normalenvektor \vec{n} , den Orientierungsvektor \vec{o} sowie den Annahrungsvektor \vec{a} beschrieben. Der Translationsvektor \vec{P} gibt die Verschiebung des Ursprungs des Koordinatensystems im Raum wieder. Abbildung 11-9 veranschaulicht die durch diese Matrix dargestellten Größen.

Zur Ermittlung der Elemente der homogenen Transformationsmatrix werden im zu ermittelnden Koordinatensystem vier Punkte angefahren. Drei dieser Punkte liegen in einer zur xy-Ebene des Koordinatensystems parallelen Ebene mit einem Abstand von 100 mm in negativer z-Richtung. Dieser ist notwendig, da die am Roboter angebrachten Bauelemente nicht zulassen, dass die Messpunkte in der xy-Ebene des Koordinatensystems durch den TCP des Roboters ohne Kollision angefahren werden können.

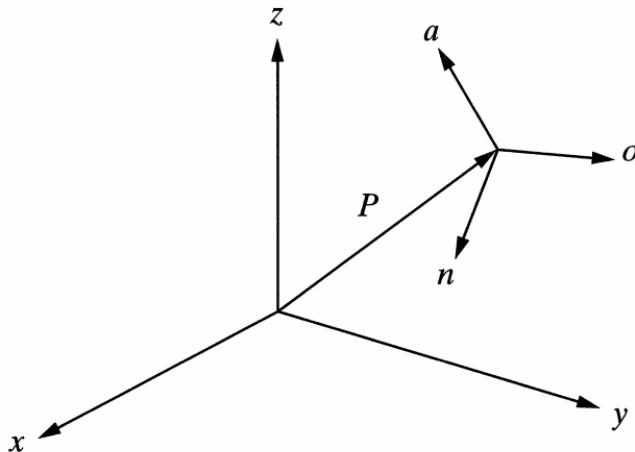


Abbildung 11-9: Darstellung der Elemente einer homogenen Transformationsmatrix (Quelle: UYGUROĞLU 2011).

Nach Erreichen eines jeden Punktes werden die zugehörigen kartesischen Koordinaten im Basis-Koordinatensystem durch die Funktion `mpGetCartPos` ermittelt. Abbildung 11-10 beschreibt die Lage der Punkte im Koordinatensystem des Werkstückträgers, das auf der Robotersteuerung als User-Koordinatensystem(01) hinterlegt ist.

Im Anschluss an die Ermittlung der Messpunkte M_i müssen die für die homogene Transformationsmatrix notwendigen Vektoren ermittelt werden. Aufgrund der nur geringfügig abweichenden Orientierung der User-Koordinatensysteme in Relation zum Basis-Koordinatensystem kann die Richtung der Vektoren ohne Berücksichtigung weiterer Umrechnungsvorschriften durch Subtraktion der einzelnen Messpunkte ermittelt werden.

$$\vec{r}_x = M_3 - M_1 \quad (14)$$

$$\vec{r}_y = M_4 - M_1 \quad (15)$$

$$\vec{r}_z = M_3 - M_2 \quad (16)$$

Im Anschluss daran müssen die ermittelten Richtungsvektoren mit Hilfe der euklidischen Norm auf Länge 1 normiert werden, welche den Richtungsvektoren der Drehmatrix zur Koordinatentransformation entsprechen.

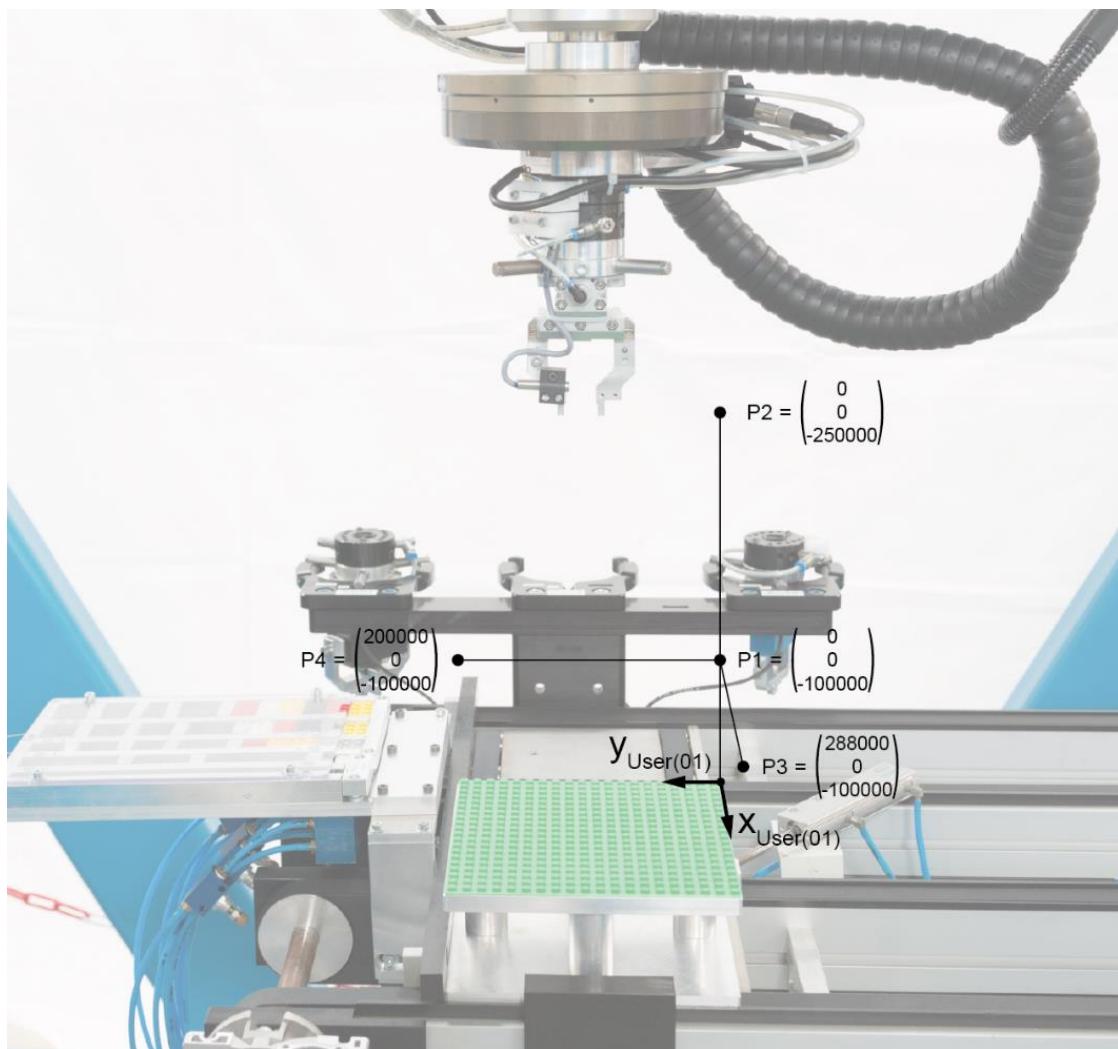


Abbildung 11-10: Lage der Messpunkte im User-Koordinatensystem(01)
(Quelle: Eigene Ausarbeitung).

$$\vec{n} = \frac{\vec{r}_x}{\|\vec{r}_x\|} \quad (17)$$

$$\vec{o} = \frac{\vec{r}_y}{\|\vec{r}_y\|} \quad (18)$$

$$\vec{a} = \frac{\vec{r}_z}{\|\vec{r}_z\|} \quad (19)$$

Unter Verwendung dieser Vektoren kann darüber hinaus der Ursprungspunkt des User-Koordinatensystems im Base-Koordinatensystem ermittelt werden.

$$\vec{P}_0 = M_1 + 100000 \vec{a} \quad (20)$$

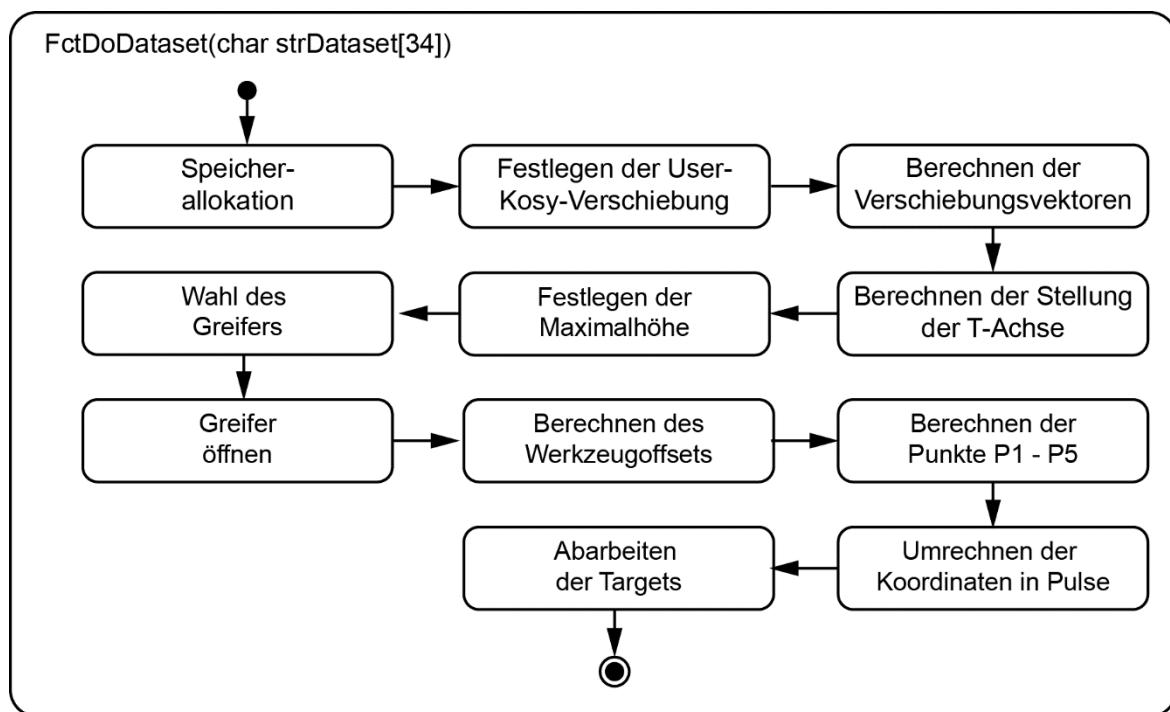
Dieser Vorgang wird im Rahmen der Initialisierung für beide User-Koordinaten-systeme durchgeführt.

11.4.4 Verarbeiten eines Datensatzes

Übermittelte Datensätze werden mit Hilfe der Funktion FctDoDataset abgearbeitet. Um die Struktur der Funktion deutlich machen zu können, wird diese im Rahmen dieses Kapitels einleitend behandelt. Im Anschluss daran finden sich die von dieser Funktion aufgerufenen Unterfunktionen, beispielweise zur Berechnung der relevanten Punkte im Arbeitsraum oder zur Koordinatentransformation.

11.4.4.1 Struktur der Funktion

Der strukturelle Aufbau der Funktion FctDoDataset wird durch das Aktivitätsdiagramm aus Abbildung 11-11 deutlich.



Nachdem der für die Funktion notwendige Speicher alloziert ist, wird zunächst die Verschiebung der User-Koordinatensysteme festgelegt. Dies dient dazu, entstandene Abweichungen zwischen der realen Hardware und den geteachten User-Koordinatensystemen auszugleichen. Dazu wird ein Vektor zur Beschreibung des dreidimensionalen Raums vom Programmierer im Code spezifiziert. Dieser wird später bei Ermittlung der Punkte im Arbeitsraum addiert. Im Anschluss daran werden die Verschiebungsvektoren berechnet. Diese Vektoren beschreiben die Lage der für den Fügeprozess notwendigen Punkte im Arbeits-

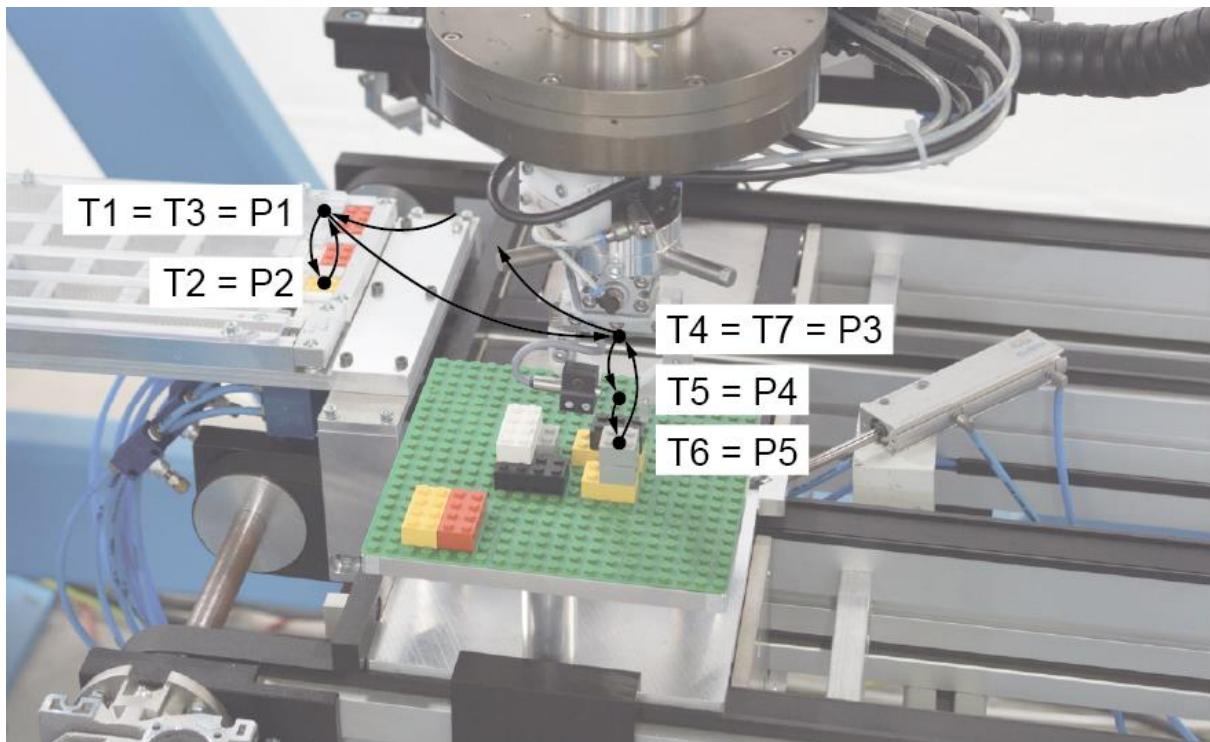
raum. Darüber hinaus muss die Stellung der T-Achse berechnet und die momentane Maximalhöhe der Lego-Baugruppe ermittelt werden. Letzteres ist notwendig, um den erforderlichen Abstand in z-Richtung zwischen Greifer und bereits gefügter Lego-Baugruppe beim Verfahren des Roboters minimieren und gleichzeitig Kollisionsfreiheit gewährleisten zu können.

Im Anschluss daran wird der Greifer mittels der Funktion FctGripperSelection aufgerufen. In Abhängigkeit des momentan eingespannten Greifers und des zum Fügen des Legosteins notwendigen Greifers wird ein Greiferwechsel durchgeführt. Hierzu werden die in Kapitel 9.5 erläuterten JOB-Dateien aufgerufen. Nach dem Greiferwechsel wird der Greifer geöffnet. Durch das MotionControlAPI kann kein Werkzeug spezifiziert werden. Die verwendeten Greifer weisen jedoch unterschiedliche Abmessungen auf und sind mit unterschiedlichem Verdrehungswinkel am Greiferwechselsystem montiert. Daher muss in Abhängigkeit des eingespannten Werkzeugs das Werkzeugoffset und die notwendige Verdrehung der T-Achse berechnet werden. Sind diese ermittelt, so können die für den Fügeprozess notwendigen Punkte P1 bis P5 berechnet werden. Deren Lage wird im nachfolgenden Kapitel 11.4.4.2 beschrieben.

Die Koordinaten der Punkte im Arbeitsraum werden im jeweiligen User-Koordinatensystem angegeben. Diese müssen im Folgenden in das Koordinatensystem Pulse umgerechnet werden. Hierzu dient die implementierte Funktion FctCartUserCosyToPulse. Auf Basis der errechneten Punkte im Arbeitsraum im Koordinatensystem Pulse können die notwendigen Targets ermittelt und abgearbeitet werden.

11.4.4.2 Punkte und Targets im Arbeitsraum

Zum Fügen eines Legosteins durch den Roboter ist eine Reihe von Linearbewegungen zwischen fünf Punkten nötig. Das API des MotoPlusSDK zum Steuern des Roboters arbeitet mit Targets, die vom Roboter nach dem FIFO-Prinzip abgearbeitet werden. Da einige Punkte mehrfach angefahren werden müssen, sind sieben Targets zum Fügen eines Lego-Steins notwendig. Abbildung 11-12 veranschaulicht die notwendigen Punkte und Targets zum Fügen eines Lego-Steins.



*Abbildung 11-12: Punkte (P) und Targets (T) zum Fügen eines Lego-Steins
(Quelle: Eigene Ausarbeitung).*

Zuerst muss der Roboter mit geöffnetem Greifer aus unbestimmter Position zu einem Punkt P1 über der Zuführeinheit bewegt werden. Im Anschluss muss der Roboter in positiver z-Richtung des User-Koordinatensystems der Zuführeinheit bewegt werden, um an Punkt P2 einen Lego-Stein aufnehmen zu können. Der Greifer wird geschlossen und der Roboter bewegt sich erneut zum Punkt P1. Von diesem Punkt aus wird eine lineare Bewegung zum Punkt P3 ausgeführt. P3 befindet sich über der Montageposition auf dem Werkstückträger. Nach Erreichen dessen bewegt sich der Roboter zum Punkt P4. Dieser Zwischenpunkt dient dazu, den Roboter möglichst lange mit möglichst hoher Geschwindigkeit verfahren lassen zu können. Vom Punkt P4 wird der Roboter mit reduzierter Geschwindigkeit zum Punkt P5 bewegt. Dies ist notwendig, da Kollisionen mit bereits montierten Lego-Steinen auftreten können. Nach Erreichen der Montageposition wird der Greifer geöffnet und der Roboter bewegt sich erneut zum Punkt P3. Da die Punkte P1 und P3 zwei Mal angesteuert werden müssen müssen sieben Targets in der Reihenfolge P1-P2-P1-P3-P4-P5-P3 abgearbeitet werden.

Aus Darstellungsgründen sind die Verbindungslien zwischen den Targets gekrümmmt. Entgegen dieser Darstellung werden alle Bewegungen linear ausgeführt.

11.4.4.3 Berechnung des Vektors IOffsetvectorFeeder

Der Verschiebungsvektor 10ffsetvectorFeeder gibt die notwendige Position zur Aufnahme eines Lego-Steins an der Zuführeinheit wieder. Dieser wird mit Hilfe der Funktion FctUptaktePosition aus dem BrickCAD-Datensatz bestimmt. Abbildung 11-13 zeigt eine Skizze der Zuführeinheit aus der Draufsicht. Durch deren Bauweise sind sieben Aufnahmepunkte vorgegeben. Diese können in der Abbildung durch die Bezeichner P1 bis P7 identifiziert werden. Die eingefügten roten Dreiecke symbolisieren die Position des Greifers. Stein D beispielsweise kann im Punkt P4 von Greifer B gegriffen werden. Der Greifer muss dabei so ausgerichtet sein, dass der Öffne- und Schließprozess in y-Richtung ausgeführt wird. Daneben kann der Stein auch mittels Greifer C aufgenommen werden, wobei der Roboter hierzu zu Punkt P5 bewegt werden muss. Der Greifer muss im Vergleich hierbei um 90° verdreht zu der Orientierungsrichtung von Aufnahmepunkt P4 stehen.

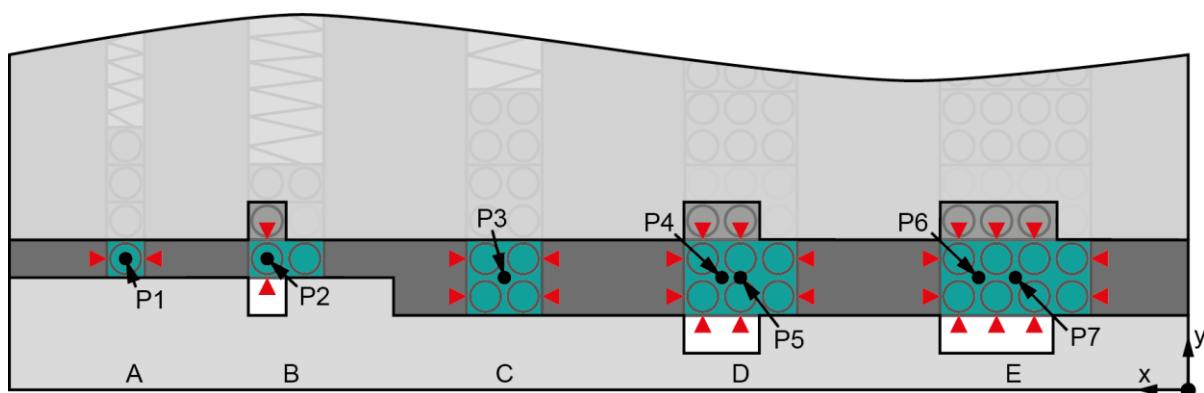


Abbildung 11-13: Aufnahmepunkte der Lego-Steine in der Zuführeinheit und zugehörige Stellung der Greifer (Quelle: Eigene Ausarbeitung).

Die Aufnahmepunkte können als Vektoren im User-Koordinatensystem der Zuführeinheit dargestellt werden. Anhand der aufzunehmenden Steinart, der angegebenen Werkzeugposition sowie dem zu verwendenden Greifer können die Aufnahmepunkte ermittelt werden. Tabelle 11-1 gibt eine Zuordnung dieser drei Parameter zu den sieben Aufnahmepunkten wieder. Darüber hinaus enthält diese Tabelle eine Spalte, die Auskunft über den Bezeichner der zugehörige Variable gibt. Die Vektoren vom Ursprungspunkt des Koordinatensystems zu den einzelnen Aufnahmepunkten finden sich in der letzten Spalte der Tabelle. Diese sind in der Einheit μm angegeben.

Tabelle 11-1: Verschiebungsvektoren bei Aufnahme der Lego-Steine an der Zuführeinheit (Quelle: Eigene Ausarbeitung).

Bricktype	Toolposition	Tooltype	Variable	Symbol	Vektor
A	0	A	IzrP1		$\begin{pmatrix} 142700 \\ 62300 \\ 0 \end{pmatrix}$
B	0	A	IzrP2		$\begin{pmatrix} 124300 \\ 62300 \\ 0 \end{pmatrix}$
B	1	A			
C	0	B	IzrP3		$\begin{pmatrix} 94200 \\ 57800 \\ 0 \end{pmatrix}$
D	0	B	IzrP4		$\begin{pmatrix} 68900 \\ 57800 \\ -500 \end{pmatrix}$
D	1	B			
D	0	C	IzrP5		$\begin{pmatrix} 60000 \\ 57800 \\ 0 \end{pmatrix}$
E	0	B	IzrP6		$\begin{pmatrix} 33400 \\ 57800 \\ 0 \end{pmatrix}$
E	2	B			
E	1	B	IzrP7		$\begin{pmatrix} 25600 \\ 57800 \\ 0 \end{pmatrix}$
E	0	C			

11.4.4.4 Berechnung des Vektors IOffsetvectorAssembly

Der Verschiebungsvektor `10ffsetvectorAssembly` beschreibt den Punkt, zu dem der Greifer im Koordinatensystem des Werkstückträgers zum Fügen eines Lego-Steins bewegt werden muss. Dieser Vektor wird mit der Funktion `FctGripperpoint` ermittelt und kann ebenfalls aus dem BrickCAD-Datensatz abgeleitet werden.

Abbildung 11-14 veranschaulicht den Verschiebungsvektor sowie die Vektoren, die zur Berechnung von `10ffsetvectorAssembly` addiert werden müssen. Das dargestellte Koordinatensystem M ist deckungsgleich mit dem geteachten User-Koordinatensystem (01) des Werkstückträgers.

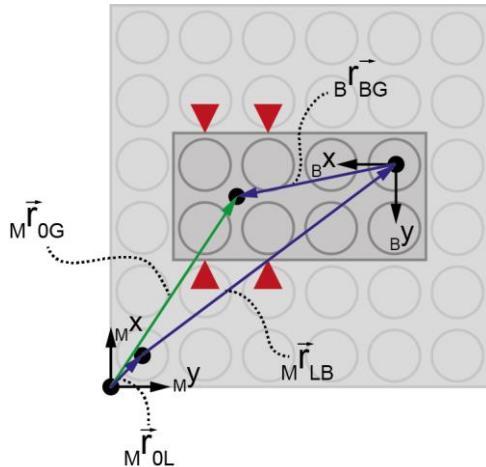


Abbildung 11-14: Prinzipskizze zur Errechnung des montagerelevanten Punktes im Koordinatensystem des Werkstückträgers
(Quelle: Eigene Ausarbeitung).

Der Vektor zum Angriffspunkt des Greifers ist in der Abbildung als Vektor $M\vec{r}_{0G}$ gekennzeichnet. Dieser kann folgendermaßen berechnet werden:

$$M\vec{r}_{0G} = M\vec{r}_{0L} + M\vec{r}_{LB} + A_{MB(oi)}\vec{r}_{BG} \quad \text{mit } i = 1, 2, 3, 4 \quad (21)$$

$M\vec{r}_{0L}$:= Vektor vom User-Koordinatensystem M zum Zentrum der Lego-Noppe mit den BrickCAD-Koordinaten (0/0/0)

$M\vec{r}_{LB}$:= Vektor des BrickCAD-Datensatzes zum Ursprung des Lego-Steins im User-Koordinatensystem M

$A_{MB(oi)}$:= Rotationsmatrix zur Darstellung eines Vektors aus dem Koordinatensystem des Lego-Steins B im User-Koordinatensystem M

$B\vec{r}_{BG}$:= Offsetvektor im Koordinatensystem des Lego-Steins B vom Steinursprung zum Angriffspunkt des Greifers

Wie in Kapitel 5.2.2 beschrieben, entsprechen die in den BrickCAD-Datensätzen enthaltenen Koordinaten dem Vektor vom Ursprungspunkt der Lego-Platte, auf dem die Steine montiert werden, zum Ursprungspunkt des Koordinatensystems eines jeden Lego-Steins. Der Ursprungspunkt des Vektors aus BrickCAD liegt in der Mitte der Lego-Noppe an der Ecke des Werkstückträgers, an der auch die Spannvorrichtung die Haltekraft einleitet. Der Ursprungspunkt des User-Koordinatensystems hingegen liegt nicht in der Mitte dieser Lego-Noppe sondern auf dem Eckpunkt des Werkstückträgers. Mit Hilfe des Vektors $M\vec{r}_{0L}$ kann diese Abweichung ausgeglichen werden. Aufgrund der quadratischen

Grundfläche einer Lego-Einheit mit einer Seitenlänge von 7800 µm kann dieser Vektor zu

$${}_{M}\vec{r}_{0L} = \begin{pmatrix} 3900 \\ 3900 \\ 0 \end{pmatrix} \quad (22)$$

bestimmt werden. Alle Vektoreneinträge sind in der Einheit µm angegeben. Der Vektor ${}_{M}\vec{r}_{LB}$ kann unter Berücksichtigung der Koordinaten im BrickCAD-Datensatz wie folgt berechnet werden:

$${}_{M}\vec{r}_{LB} = \begin{pmatrix} 7800x \\ 7800y \\ 9600z \end{pmatrix} \quad (23)$$

Durch den Vektor ${}_{B}\vec{r}_{BG}$ wird der Offsetvektor vom Ursprungspunkt B des Lego-Steins zum Angriffspunkt des Greifers G wiedergegeben. Zur Umrechnung dieses Vektors in das User-Koordinatensystem muss der Vektor ${}_{B}\vec{r}_{BG}$ mit einer Drehmatrix multipliziert werden. Diese kann in Abhängigkeit der Orientierung des Lego-Steines ermittelt werden. In Tabelle 11-2 sind die möglichen Zustände der Drehmatrix den im BrickCAD-Datensatz enthaltenen Orientierungsrichtungen zugeordnet.

*Tabelle 11-2: Auflistung der der Drehmatrizen $A_{MB(oi)}$
(Quelle: Eigene Ausarbeitung).*

Orientation	Variable	Symbol	Matrix
o0	A_{MBo0}		$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
o1	A_{MBo1}		$\begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
o2	A_{MBo2}		$\begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$
o3	A_{MBo3}		$\begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Der Offset-Vektor ${}_B\vec{r}_{BG}$ kann für jeden Stein und jeden möglichen Greiferangriffspunkt in Abhängigkeit des zum Einsatz kommenden Greifers und der zugehörigen Werkzeugposition ermittelt werden. Diese sind in Tabelle 11-3 aufgelistet.

Tabelle 11-3: Mögliche Offsetvektoren ${}_B\vec{r}_{BG}$ (Quelle: Eigene Ausarbeitung).

Bricktype	Toolposition	Tooltype	Variable	Symbol	Vektor
B	1	A	lBrbtBtp1ttA		$\begin{pmatrix} 7800 \\ 0 \\ 0 \end{pmatrix}$
C	0	B	lBrbtCtp0ttB		$\begin{pmatrix} 3900 \\ 3900 \\ 0 \end{pmatrix}$
D	0	B	lBrbtDtp0ttB		$\begin{pmatrix} 3900 \\ 3900 \\ 0 \end{pmatrix}$
D	1	B	lBrbtDtp1ttB		$\begin{pmatrix} 11700 \\ 3900 \\ 0 \end{pmatrix}$
D	0	C	lBrbtDtp0ttC		$\begin{pmatrix} 7800 \\ 3900 \\ 0 \end{pmatrix}$
E	0	B	lBrbtEtp0ttB		$\begin{pmatrix} 3900 \\ 3900 \\ 0 \end{pmatrix}$
E	1	B	lBrbtEtp1ttB		$\begin{pmatrix} 11700 \\ 3900 \\ 0 \end{pmatrix}$
E	2	B	lBrbtEtp2ttB		$\begin{pmatrix} 19500 \\ 3900 \\ 0 \end{pmatrix}$
E	0	C	lBrbtEtp0ttC		$\begin{pmatrix} 11700 \\ 3900 \\ 0 \end{pmatrix}$

Anhand des in Kapitel 5.2.2.2 erläuterten Beispiel-Datensatzes soll die Berechnung des montagerelevanten Punktes nun veranschaulicht werden. Der diesem Beispiel zu Grunde gelegte und vom Client übermittelte BrickCAD-Datensatz lautet btEpx004y003z000o2ttBto0tp2. Unter Berücksichtigung von Tabelle 11-2 und Tabelle 11-3 ergibt sich somit folgende Rechnung:

$${}^M\vec{r}_{0G} = \begin{pmatrix} 3900 \\ 3900 \\ 0 \end{pmatrix} + \begin{pmatrix} 7800 \times 4 \\ 7800 \times 3 \\ 9600 \times 0 \end{pmatrix} + \begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} 19500 \\ 3900 \\ 0 \end{pmatrix} = \begin{pmatrix} 15600 \\ 23400 \\ 0 \end{pmatrix}.$$

11.4.4.5 Berechnung der Stellung der T-Achse

Sowohl bei der Aufnahme eines Lego-Steins in der Zuführeinheit als auch bei der Montage des Steins muss die dafür notwendige Stellung der T-Achse ermittelt werden.

Zu deren Ermittlung bei Aufnahme eines Lego-Steins steht die Funktion FctToolOrientationUptake zur Verfügung. Analog zur Ermittlung des Verschiebungsvektors kann die Stellung der T-Achse aus der Art des zu fügenden Lego-Steins und dem zu verwendenden Greifer abgeleitet werden. Tabelle 11-4 stellt die dazugehörige Zuordnung dar. Die Angabe zur Stellung der T-Achse bezieht sich auf das Koordinatensystem Base.

Tabelle 11-4: Zuordnungstabelle zur Stellung der T-Achse im Koordinatensystem Base bei Aufnahme eines Steins (Quelle: Eigene Ausarbeitung).

Bricktype	Tooltype	Position of T-Axis
A	A	1300000
B	A	400000
C	B	1300000
D	B	400000
D	C	1300000
E	B	400000
E	C	1300000

Nach dem gleichen Prinzip ist die Funktion FctToolOrientationAssembly aufgebaut. Diese Funktion gibt die Stellung der T-Achse im Pulse-Koordinatensystem bei der Montage an. Die notwendigen Eingangsparameter sind Art und Orientierung des zu fügenden Lego-Steins sowie Orientierung des Werkzeugs. Darüber hinaus sind die Position des Werkzeugs und der verwendete Greifer

von Belang. Die Zuordnung der Stellung der T-Achse und der einzelnen Parameter wird durch Tabelle 11-5 verdeutlicht. Beinhaltet ein Parameterfeld keinen Eintrag, so ist dieser für die Stellung der T-Achse unerheblich. Da die Steine A und B nur mit Greifer A und die Steine C und D nur mit Greifer B gegriffen werden können, ist der Parameter *ToolType* für diese Lego-Steine irrelevant.

Tabelle 11-5: Zuordnungstabelle zur Stellung der T-Achse im Koordinatensystem Pulse beim Montieren eines Steins (Quelle: Eigene Ausarbeitung).

Bricktype	Orientation	Tool-Orientation	Toolposition	ToolType	Position of T-Axis
A	-	0	-	-	1300000
C					
A	-	1	-	-	400000
C					
B	0	-	0	-	3100000
	1				2200000
	0		1		1300000
	1				400000
D	0	-	0	-	3100000
	1				2200000
	0		1		1300000
	1				400000
E	0	-	0	-	3100000
	1		1		2200000
	0		0		1300000
	1		1		400000
	0	-	2	C	1300000
	1				400000
	0				2200000
	1				1300000

11.4.4.6 Wahl eines Greifers

Die Auswahl eines Greifers wird durch die Funktion FctGripperSelection realisiert. Das in Abbildung 11-15 dargestellte Aktivitätsdiagramm zeigt die implementierten Abläufe für den Fall, dass Greifertyp A benötigt wird. Ein vollständiges Aktivitätsdiagramm ist im Anhang zu finden.

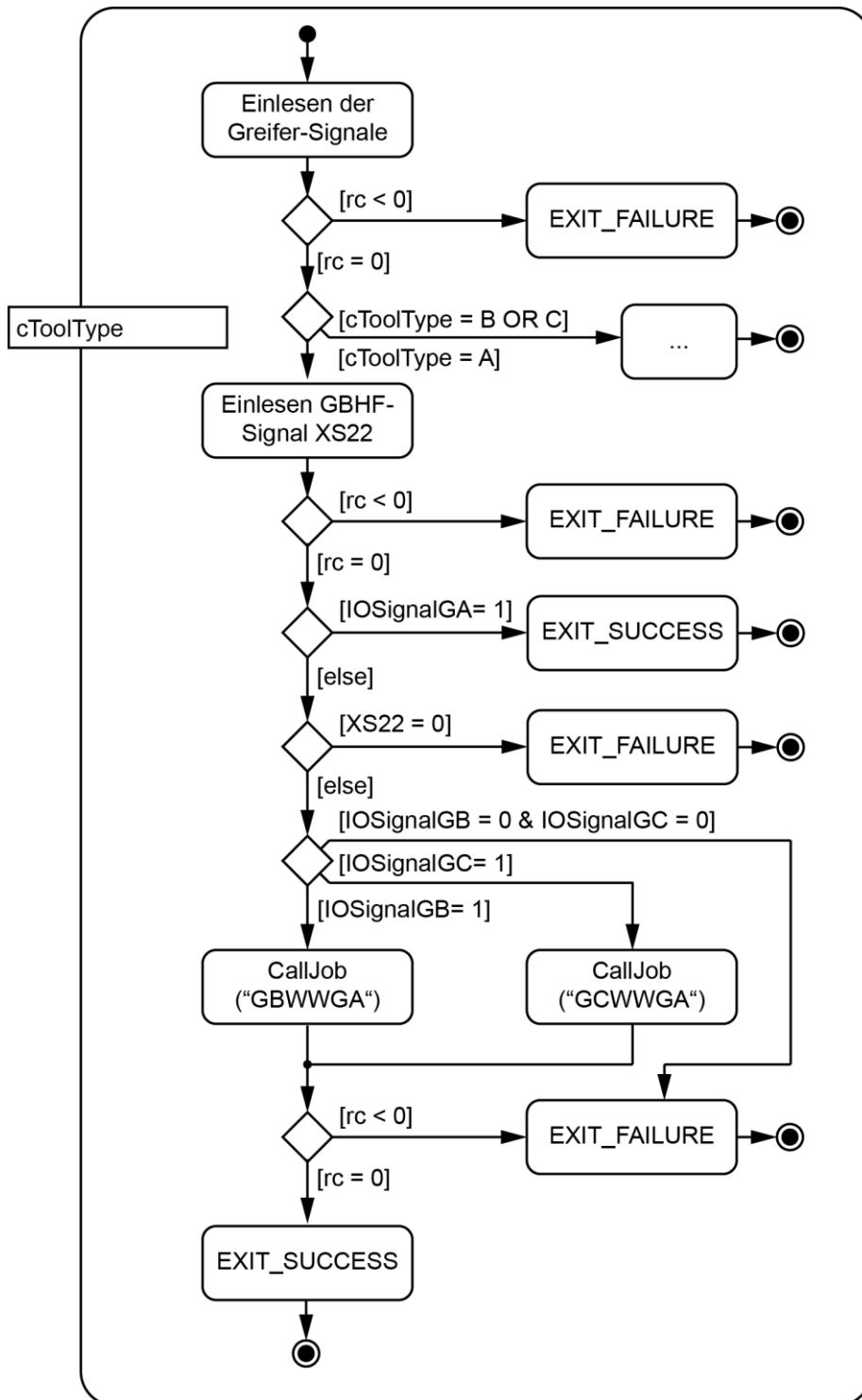


Abbildung 11-15: Ausschnitt des Aktivitätsdiagramms zur Funktion FctGripperSelection (Quelle: Eigene Ausarbeitung).

Einleitend werden die Greifer-Signale aus dem Register der SPS eingelesen und die Variablen beschrieben. Für jeden Greifer und jeden Sensor des Greiferbahnhofs existiert eine boolsche Variable, die dem Zustand des zugehörigen SPS-Eingangssignals entspricht. Für Greifer A ist die Variable *IOSignal/GA* deklariert, für den Sensor des Greiferbahnhofs *XS22*. Ist dieser Aufruf erfolgreich, so wird selektiert, welcher Greifer benötigt wird. Im dargestellten Aktionsstrang ist der Greifer vom Typ A erforderlich. Im Anschluss wird der Sensor der Bucht des Greiferbahnhofs eingelesen, in der Greifer A gespeichert ist. Sofern die Sensordatenerfassung erfolgreich verläuft, wird im nächsten Schritt überprüft, ob ein Greiferwechsel überhaupt notwendig ist. Ist Greifer A bereits eingespannt, so kann die Funktion beendet werden. Ist dem nicht so, wird anschließend geprüft, ob der Greifer im Greiferbahnhof liegt. Wenn der Greifer vorhanden ist, wird in Abhängigkeit des momentan eingespannten Greifers der entsprechende Job zum Werkzeugwechsel aufgerufen. Die im Diagramm abgefragte Variable *rc* dient als Speicher der Rückgabewerte verwendeter Funktionen.

11.4.4.7 Ermittlung des Werkzeugoffsets

Jeder der drei Greifer hat unterschiedliche Abmessungen. Darüber hinaus sind die Greifer mit unterschiedlicher Verdrehung um die Hochachse am Greiferwechselsystem angebracht. Um diese Unterschiede berücksichtigen zu können, wird mit Hilfe der Funktion *vectorToolOffset* ermittelt, welcher Greifer zum Einsatz kommt. In Abhängigkeit davon werden die in Tabelle 11-6 aufgelisteten Offsetvektoren und Verdrehungen zurückgegeben.

Tabelle 11-6: Werkzeugoffset der drei Greifer (Quelle: Eigene Ausarbeitung).

Variable	Greifer GA	Greifer GB	Greifer GC	Einheit
Δx	0	1000	0	μm
Δy	0	0	0	μm
Δz	-132000	-125500	-139500	μm
$\Delta \varphi_x$	0	0	0	$10^2 \mu^\circ$
$\Delta \varphi_y$	0	0	0	$10^2 \mu^\circ$
$\Delta \varphi_z$	682154	-104987	-103600	$10^2 \mu^\circ$

11.4.4.8 Berechnung der Punkte P1 bis P5

Auf Grundlage der vorangegangenen Berechnungen können im Folgenden die im Arbeitsraum relevanten Punkte ermittelt werden. Zur Verdeutlichung derer sei nochmals auf Abbildung 11-12 verwiesen. Punkt P1 im User-Koordinatensystem der Zuführeinheit errechnet sich wie folgt:

$${}_{UK(0)}\vec{r}_{0P1} = {}_{UK(0)}\vec{r}_{OffsetFeeder} + {}_{UK(0)}\vec{r}_{S.UK0} + {}_{UK(0)}\vec{r}_{OffsetTool} - \begin{pmatrix} 0 \\ 0 \\ 15000 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ iHeight \end{pmatrix} \quad (24)$$

mit

- ${}_{UK(0)}\vec{r}_{0P1}$:= Vektor im User-Koordinatensystem 0 (User-Koordinatensystem der Zuführeinheit) zum Punkt P1
- ${}_{UK(0)}\vec{r}_{OffsetFeeder}$:= Verschiebungsvektor 10ffsetvector-Feeder (siehe Kapitel 11.4.4.3)
- ${}_{UK(0)}\vec{r}_{S.UK0}$:= Verschiebung des User-Koordinatensystems (10ffsetvectorFeederOrigin)
- ${}_{UK(0)}\vec{r}_{OffsetTool}$:= Werkzeugoffset vectorToolOffset (siehe Kapitel 11.4.4.7)
- $iHeight$:= Höhe der bereits gefügten Lego-Baugruppe in μm

Neben den aufgeführten Variablen wird in Formel (24) ein Vektor subtrahiert, dessen z-Komponente $15000 \mu\text{m}$ groß ist. Dies ist der Mindestabstand der so zwischen den Punkten P1 und P2 gewährleistet ist.

Durch Abbildung 11-16 zur Berechnung des Punkts P2 wird verdeutlicht, wie sich die genannten Vektoren zusammensetzen. Dabei ist auch ersichtlich, wie der reale Greifpunkt an der Zuführeinheit virtuell durch Berücksichtigung des Werkzeugoffsets verschoben wird.

Punkt P2 errechnet sich somit wie folgt:

$${}_{UK(0)}\vec{r}_{0P2} = {}_{UK(0)}\vec{r}_{OffsetFeeder} + {}_{UK(0)}\vec{r}_{ShiftUK0} + {}_{UK(0)}\vec{r}_{OffsetTool}. \quad (25)$$

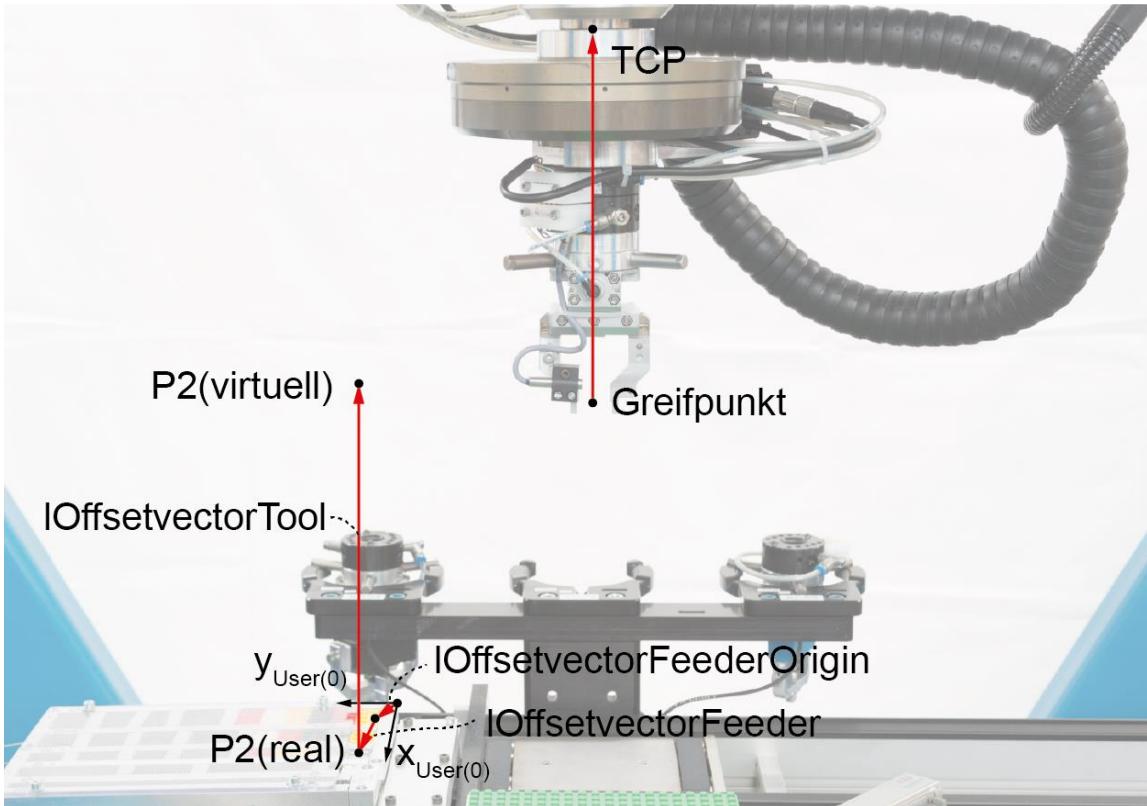


Abbildung 11-16: Ermittlung des Punkts P2 durch Vektoraddition der relevanten Größen (Quelle: Eigene Ausarbeitung).

Die Punkte P3, P4 und P5, allesamt Punkte im User-Koordinatensystem 1, dem Koordinatensystem des Werkstückträgers, können wie folgt berechnet werden:

$${}_{UK(1)}\vec{r}_{0P3} = {}_{UK(1)}\vec{r}_{\text{OffsetAssembly}} + {}_{UK(1)}\vec{r}_{S.UK1} + {}_{UK(1)}\vec{r}_{\text{OffsetTool}} - \begin{pmatrix} 0 \\ 0 \\ 20000 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ iHeight \end{pmatrix} \quad (26)$$

$${}_{UK(1)}\vec{r}_{0P4} = {}_{UK(1)}\vec{r}_{\text{OffsetAssembly}} + {}_{UK(1)}\vec{r}_{S.UK1} + {}_{UK(1)}\vec{r}_{\text{OffsetTool}} - \begin{pmatrix} 0 \\ 0 \\ 15000 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ iHeight \end{pmatrix} \quad (27)$$

$${}_{UK(1)}\vec{r}_{0P5} = {}_{UK(1)}\vec{r}_{\text{OffsetAssembly}} + {}_{UK(1)}\vec{r}_{S.UK1} + {}_{UK(1)}\vec{r}_{\text{OffsetTool}} \quad (28)$$

mit

${}_{UK(1)}\vec{r}_{0Pi}$:= Vektor im User-Koordinatensystem 1 zum Punkt P_i mit $i \in \{3,4,5\}$

${}_{UK(1)}\vec{r}_{\text{OffsetAssembly}}$:= Verschiebungsvektor $\text{IOffsetvector-Assembly}$ (siehe Kapitel 11.4.4.4)

${}_{UK(1)}\vec{r}_{S.UK1}$:= Verschiebung des User-Koordinatensystems (Variable $\text{IOffsetvector-Origin}$)

11.4.4.9 Koordinatentransformation ins Pulse-Koordinatensystem

Die Koordinaten der berechneten Punkte sind in den User-Koordinatensystemen angegeben. Werden diese Koordinaten als Grundlage für den Montagevorgang verwendet, so tritt ein Fehler bei der Positionierung der T-Achse in Erscheinung. Wird für die T-Achse ein Winkel von 0° als Zielposition festgelegt, so kann es passieren, dass die T-Achse zu den Positionen -180° oder $+180^\circ$ bewegt wird. Grund hierfür ist vermutlich ein Fehler in der Berechnung der inversen Kinematik. Da die durch das SDK bereitgestellten Funktionen gekapselt sind, kann über die Ursache für diesen Fehler keine sichere Aussage getroffen werden. Der Fehler ist insofern gravierend, da die T-Achse aufgrund der angebrachten Messkabel nur in einem eingeschränkten Bereich bewegen kann. Positionen kleiner -30° von der Nullposition können nicht erreicht werden ohne, dass Schäden an den angebrachten Mess- und Versorgungsleitungen entstehen.

Diesem Fehler kann entgegengetreten werden, indem die anzufahrenden Koordinaten im Koordinatensystem Pulse angegeben werden. Dahingehend ist eine Koordinatentransformation notwendig. Diese ist in der Funktion FctCartUserCosyToPulse implementiert und basiert auf folgender Formel:

$$Pulse \vec{r}_{0Pi} = A_{PulseAngle} A_{AngleBase} A_{BaseUK(j)} UK(j) \vec{r}_{0Pi} \quad (29)$$

mit

- $Pulse \vec{r}_{0Pi}$:= Vektor im Koordinatensystem Pulse zum Punkt P_i mit $i \in \{1,2,3,4,5\}$
- $A_{PulseAngle}$:= Transformationsmatrix vom Koordinatensystem Angle ins Koordinatensystem Pulse
- $A_{AngleBase}$:= Transformationsmatrix vom Koordinatensystem Base ins Koordinatensystem Angle
- $A_{BaseUK(j)}$:= Transformationsmatrix vom Userkoordinatensystem j ins Koordinatensystem Base
- $UK(j) \vec{r}_{0Pi}$:= Vektor im User-Koordinatensystem j zum Punkt P_i mit $i \in \{1,2,3,4,5\}$

Die Transformationsmatrix vom Koordinatensystem Angle ins Koordinatensystem Pulse ist eine systeminterne Größe und wird vom SDK zur Verfügung gestellt.

11.4.4.10 Ausführen der Targets

Da die für den Montagevorgang erforderlichen Punkte P_i berechnet sind, können die sieben Targets nun abgearbeitet werden. Jedes Target wird in einer hierfür implementierten Funktion behandelt. Exemplarisch wird im Folgenden die Funktion FctDoTarget1 an Hand des in Abbildung 11-17 gezeigten, vereinfachten Aktivitätsdiagramms erläutert. Dabei wird der Roboter lediglich über die Zuführeinheit bewegt und ein Stein ausgestoßen. Die Aktivitätsdiagramme aller weiteren Funktionen zum Abarbeiten der Targets sind im Anhang zu finden.

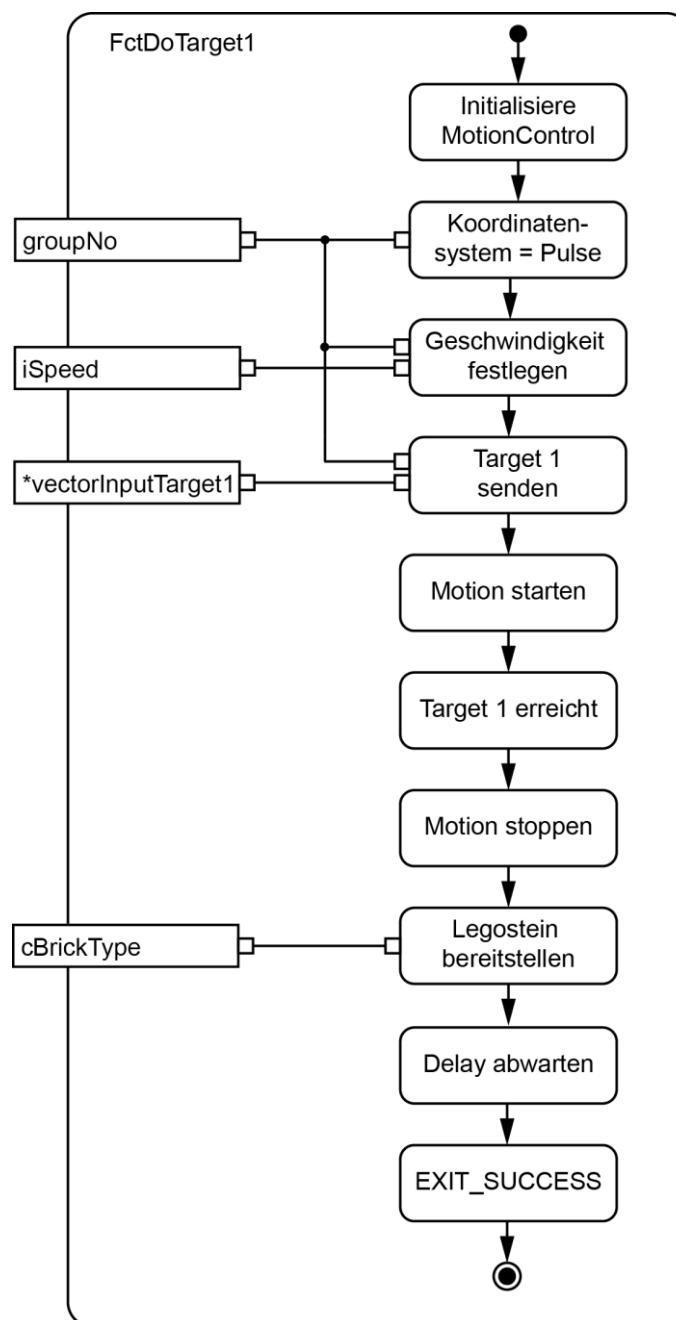


Abbildung 11-17: Aktivitätsdiagramm zur Funktion FctDoTarget1
 (Quelle: Eigene Ausarbeitung).

Nach der Initialisierung des Motioncontrol-API muss das Koordinatensystem spezifiziert werden. Anschließend wird die Geschwindigkeit angegeben, mit der die lineare Bewegung durchgeführt wird. Die Pulse-Koordinaten P1 des Target 1 werden daraufhin übermittelt. Nach dem Start-Befehl bewegt sich der Roboter zu Target 1. Ist dieser Punkt erreicht, kann das MotionControlAPI beendet werden. Im Anschluss daran kann der Lego-Stein durch die Zuführeinheit ausgestoßen werden. Aufgrund langer Wege zwischen Ventilinsel und Aktorik weisen die pneumatischen Bauelemente eine hohe Systemtotzeit auf. Um dieses Problem zu lösen wird nach dem Ansteuern des Zylinders der Zuführeinheit noch ein Delay von 500 ms abgewartet.

11.5 Der Task zur Übertragung der Sensordaten via Ethernet

Die Übertragung der Sensordaten via TCP-IP im Task mpTaskCanCommunication wird nach dem in Abbildung 11-18 veranschaulichtem Schema durchgeführt.

Wie bei der Befehls-Datenübertragung wird einleitend ein Socket erzeugt. Dieses wird an IP und Port gebunden. Hierfür zugeordnet ist Port 11100. Im Anschluss daran wird der Server in den Listening-Modus versetzt. Nach Aufbau der TCP-IP-Verbindung durch den Client werden in der darauf folgenden while-Schleife die Anfragen vom Client empfangen und durch die Funktion FctProcessCANBUSData verarbeitet.

Zur Erinnerung wird durch Abbildung 11-19 der Aufbau der vom Client versendeten Daten veranschaulicht. Innerhalb der Funktion FctProcessCANBUSData werden einleitend die Positionen der Identifier ermittelt. Anhand derer kann der übermittelte String in zwei Teilstings aufgeteilt werden. Auf Grundlage dieser Zeichenfolgen werden die hexadezimalcodierten Kräfte in x- y- und z-Richtung ermittelt. Nach der Umrechnung ins Dezimalsystem folgt die in Abschnitt 10.5.2 erläuterte Vorzeichenbetrachtung. Das Ergebnis dieser Betrachtung muss im Anschluss daran noch normiert werden. Die berechneten Kräfte und Momente werden in einem global angelegtem Struct ActFTCSensorData vom Typ FTCSensorData gespeichert.

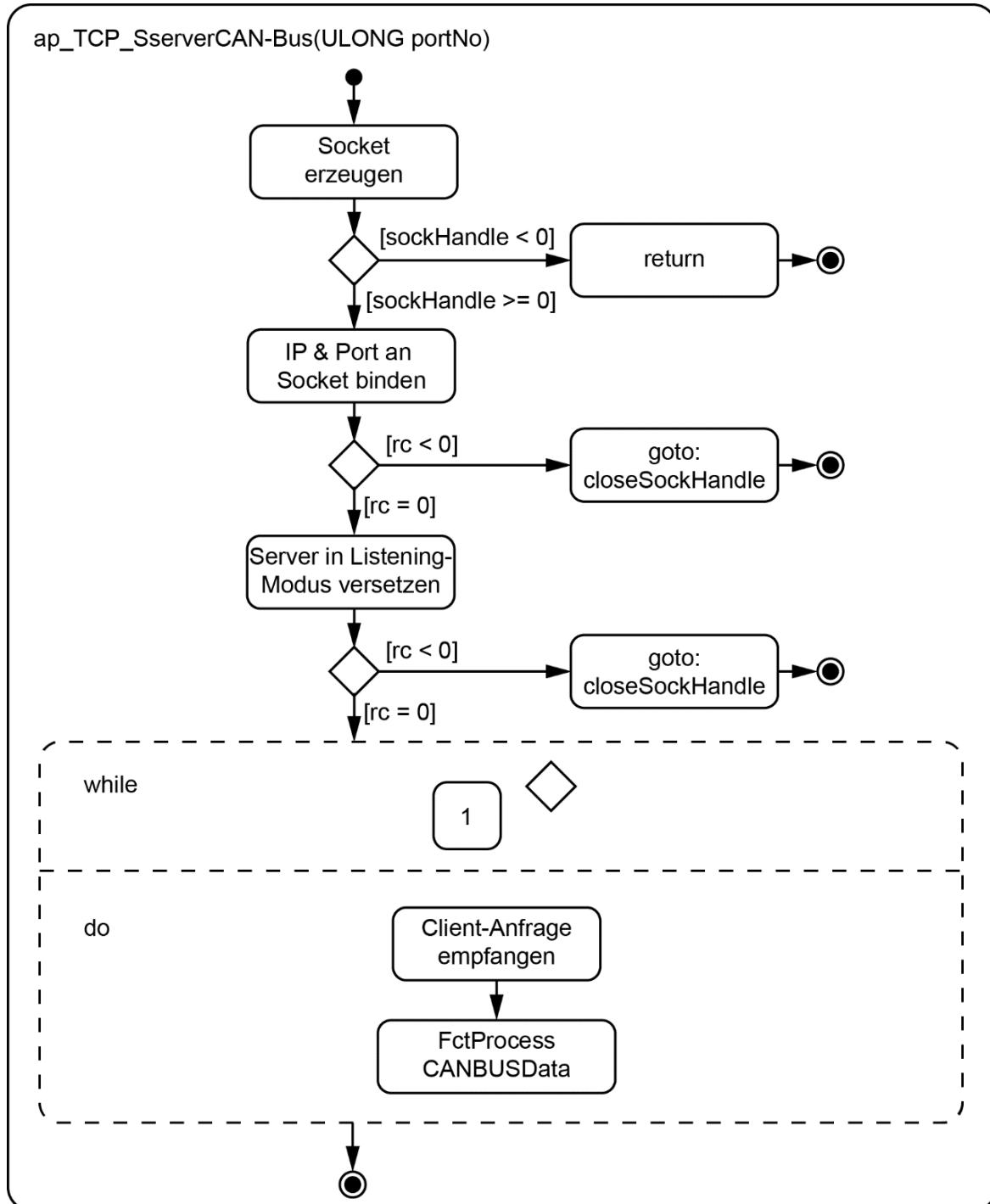


Abbildung 11-18: Aktivitätsdiagramm zum Thread `mpTaskCanCommunication`
 (Quelle: Eigene Ausarbeitung).

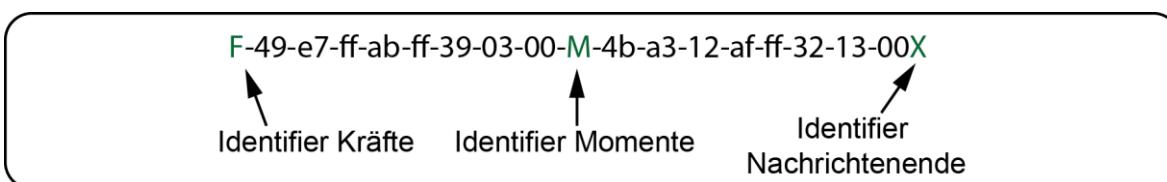


Abbildung 11-19: Beispiel einer vom Client übermittelten Nachricht mit den Sensordaten des KMS als Inhalt (Quelle: Eigene Ausarbeitung).

12 Untersuchung zur Implementierung des iwb-Modells auf der Robotersteuerung

Im Folgenden soll untersucht werden, inwieweit das von MICHNIEWICZ & REINHART (2015A) beschriebene Modell und dessen Erweiterung durch KRAUS (2015) in der vorliegenden CPRC implementiert werden kann. Grundlage für diese Untersuchung ist die in Kapitel 4 bereits vorgestellte Systemarchitektur. Auf Basis der bisherigen Arbeit soll betrachtet werden, wie die Steuerung der Roboterzelle mit Hilfe dieses Modells erfolgen kann und in welchem Grad dieses adaptiert werden muss.

Hierbei sei angemerkt, dass dieses Modell bereits in der vorgestellten Software BrickCAD enthalten ist. Dazu werden der Production Graph sowie der lösungsneutrale Montagevorranggraph erstellt. Im Anschluss daran wird der erweiterte Montagevorranggraph erzeugt. Dieser ist in der build-Textdatei beschrieben und wird zeilenweise durch den implementierten Client per TCP-IP an die Robotersteuerung übertragen.

Die angestellten Untersuchungen konzentrieren sich daher auf die Zuweisung der vom Produkt geforderten Prozesse zu den Betriebsmitteln des Produktionsystems. Die Robotersteuerung soll dazu als rekonfigurierbares System programmiert werden, dessen Steuerung auf den übergebenen BrickCAD-Datensätzen beruht. Dabei soll die Programmiermethodik der Funktionsprimitiva umgesetzt werden.

Die Untersuchung besteht aus einer zweistufigen Betrachtung. Im ersten Schritt soll analysiert werden, wie die VR der Betriebsmittel und der PG erstellt werden können, wobei davon ausgegangen wird, dass der Anlagenprogrammierer die Anlage weiterhin konfiguriert. Daher wird untersucht, inwieweit die Implementierung der virtuellen Repräsentanzen der verfügbaren Betriebsmittel auf der Robotersteuerung möglich ist und ob dies unter den gegebenen Randbedingungen zielführend ist. Im zweiten Schritt soll die vollständige, automatische Konfiguration der Roboterzelle auf Grundlage von CPS betrachtet werden. Ausführung hierzu finden sich in Kapitel 12.3. Auf Grundlage dieser Ergebnisse sollen im Anschluss die für die Montage der Lego-Baugruppe erforderlichen Primär- und Sekundärprozesse identifiziert und umgesetzt werden.

12.1 VR der Betriebsmitteln

Einleitend muss der Production Graph der Roboterzelle modelliert werden. Hierzu müssen die CPDs und CPRCs der Roboterzelle identifiziert und mit einer geeigneten Methodik beschrieben werden. Der Production Graph kann mit Hilfe verschiedener Vorgehensweisen erstellt werden. Im Folgenden werden drei denkbare Methoden erläutert.

MICHNIEWICZ & REINHART (2015B) schlagen vor, den Production Graph auf Grundlage des dreidimensionalen Simulationsmodells aller Cyber-Physischer Zellenelemente automatisch zu erstellen. Diese Vorgehensmethodik ist für die auf dem Leitrechner zu implementierende Software ein adäquates Mittel. Für die Robotersteuerung ist diese Methodik allerdings als zu rechenintensiv zu erachten. Darüber hinaus muss hierzu zuerst untersucht werden wie CAD-Dateien per TCP-IP auf die Robotersteuerung übertragen und dort analysiert werden können. Unabhängig vom Format dieser Dateien können diese weder als gekapselte Datei am Stück übertragen, noch auf der FS100 abgespeichert werden. Dahingehend müsste das Simulationsmodell beispielsweise im XML-Format (Extensible Markup Language) beschrieben sein. Dieses Dokument kann zeichenweise übertragen und als String abgespeichert werden. Der hierzu notwendige Aufwand erscheint in Relation mit dem ableitbaren Nutzen allerdings nicht zweckmäßig.

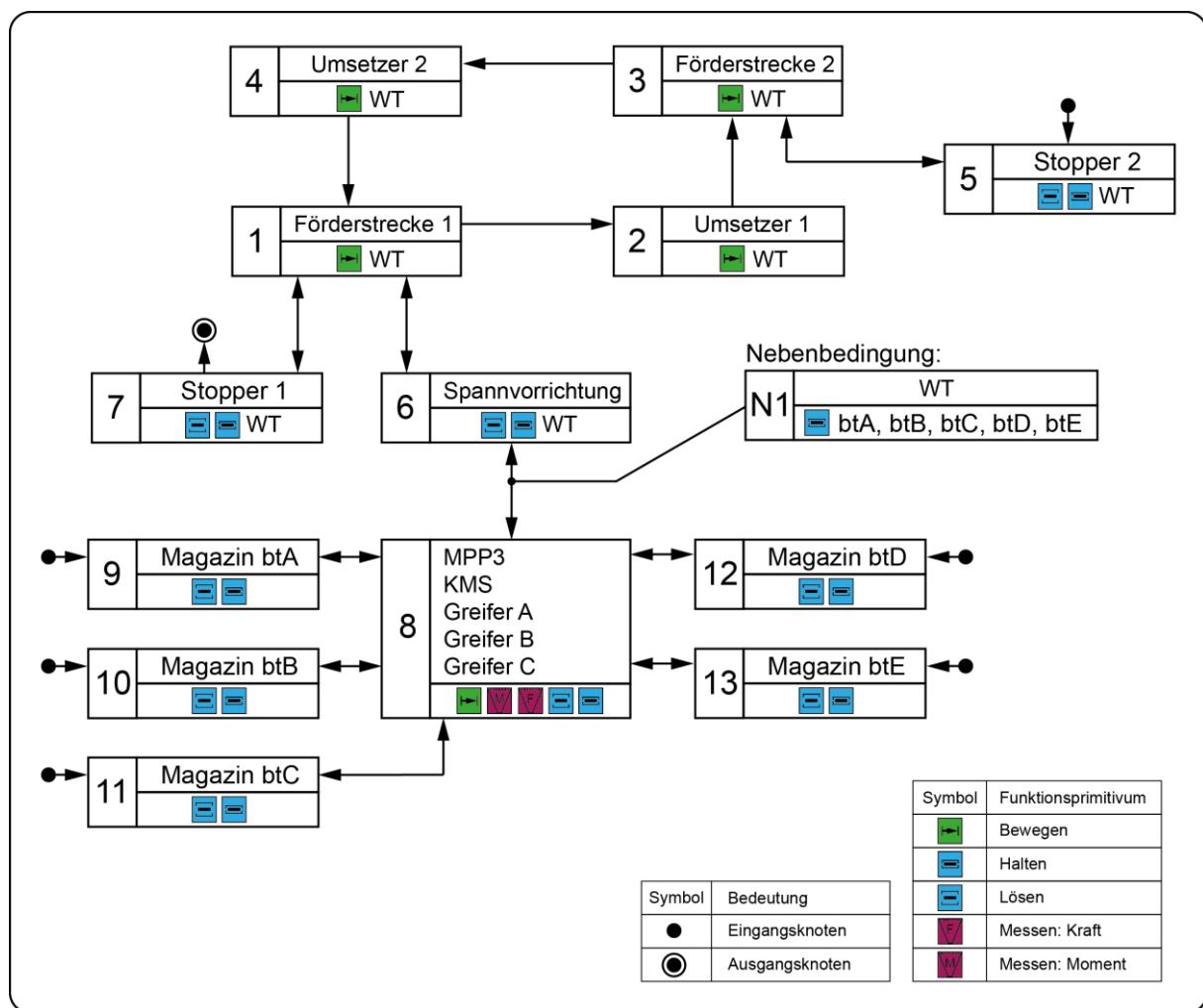
Alternativ kann der auf dem Leitrechner bereits erstellte Production Graph per TCP-IP an die Robotersteuerung übertragen werden. Auf Grundlage dessen können die Betriebsmittel mit den Produktionsprozess-Anforderungen verglichen und entsprechende Handlungsstrategien abgeleitet werden. Zur Beschreibung des Production Graph muss ein entsprechendes Schnittstellendokument erarbeitet werden, das im Anschluss per TCP-IP an die Robotersteuerung gesendet werden kann. Dieses Schnittstellendokument ist aus der in MICHNIEWICZ & REINHART (2015B) vorgestellten und bereits implementierten Klassenstruktur zur Beschreibung des Production Graph abzuleiten.

Eine weitere Möglichkeit zur Modellierung des Production Graph auf der FS100 ist die Programmierung der Zellenelemente durch den Programmierer vor Inbetriebnahme. In Abhängigkeit der SPS-Eingangssignale können einzelne CPDs aus dem PG entfernt oder diesem wieder hinzugefügt werden. Auf diese Weise kann der PG zur Laufzeit rekonfiguriert werden. Dies erscheint zum gegenwärtigen Entwicklungsstand als adäquate Lösung, da kein Schnittstellendokument zur Übertragung des auf dem Leitrechner erstellten Production Graphs existiert.

Untersuchung zur Implementierung des iwb-Modells auf der Robotersteuerung

Der hierzu notwendige Aufwand ist aufgrund der Überschaubarkeit der Roboterzelle vertretbar. Auf Grundlage dieses Modells kann im Anschluss die eigentliche Untersuchung zur Programmierung der Zelle mittels Funktionsprimitiva erfolgen.

Der zu erstellende PG des Demonstrators wird in Abbildung 12-1 veranschaulicht. Die Elemente des Förderbands werden durch die CPDs 1 bis 7 dargestellt. In der vorliegenden Grafik sind die Funktionsprimitiva, die Elemente des Förderbands betreffen mit der Kennzeichnung WT versehen. Hierdurch wird dargestellt, dass diese Elemente in der Lage sind, Werkstückträger handhaben zu können. Lego-Steine hingegen können von diesen Elementen nicht gehalten oder bewegt werden.



*Abbildung 12-1: Production Graph des Demonstrators
(Quelle: Eigene Ausarbeitung).*

Knoten 1 beinhaltet Förderstrecke 1. Diesem Knoten ist das Funktionsprimitivum *Bewegen* zugeordnet. Durch eine Kante wird dieser Knoten mit Knoten 2 verbunden, welcher Umsetzer 1 beinhaltet und ebenfalls Werkstückträger *Bewegen* kann. Darauf folgen mit Knoten 3 und 4 zwei weitere Förderelemente

Untersuchung zur Implementierung des iwb-Modells auf der Robotersteuerung

des Förderbandes. Die Kanten zwischen diesen Knoten sind gerichtet. Hierdurch wird modelliert, dass das Förderband nur in einer Förderrichtung betrieben werden kann. Darüber hinaus sind diesen vier Knoten die Knoten 5, 6 und 7 zugeordnet. Diese drei Knoten können die Werkstückträger *Halten*. Bei Stopper 1 können fertig montierte Lego-Baugruppen aus dem System entnommen werden. Dieser ist daher als Ausgangsknoten modelliert. Knoten 5 wiederum ist als Eingangsknoten dargestellt, da hier leere Werkstückträger in das System eintreten. Zur Verdeutlichung der Verschaltung der Elemente des Förderbands dient Abbildung 12-2. Hierbei ist die räumliche Anordnung der einzelnen CPDs aus der Draufsicht auf das Förderband dargestellt.

Knoten 6 ist mit Knoten 8 verbunden. Diese CPDC beinhaltet den Roboter MPP3, den Kraft-Momenten-Sensor sowie die drei Greifer A, B und C. Auf Grundlage dieser CPDs kann die CPDC Lego-Steine bewegen, Kräfte und Momente messen sowie Lego-Steine halten und auch lösen. Knoten 8 wiederum ist mit den fünf Magazinen der Zuführeinheit verbunden. Jedes Magazin ist als CPD modelliert und kann einen Lego-Stein sowohl halten als auch lösen.

Die Kante zwischen den Knoten 6 und 8 ist durch eine Nebenbedingung gekennzeichnet. Durch diese wird modelliert, dass Werkstückträger Lego-Steine halten können. Im Programm kann dies allgemein gehalten sein und muss keiner Kante zugeordnet werden, da diese Nebenbedingung allgemeingültig ist. Aus Darstellungsgründen ist diese im vorliegenden Fall allerdings dieser relevanten Kante zugeordnet.

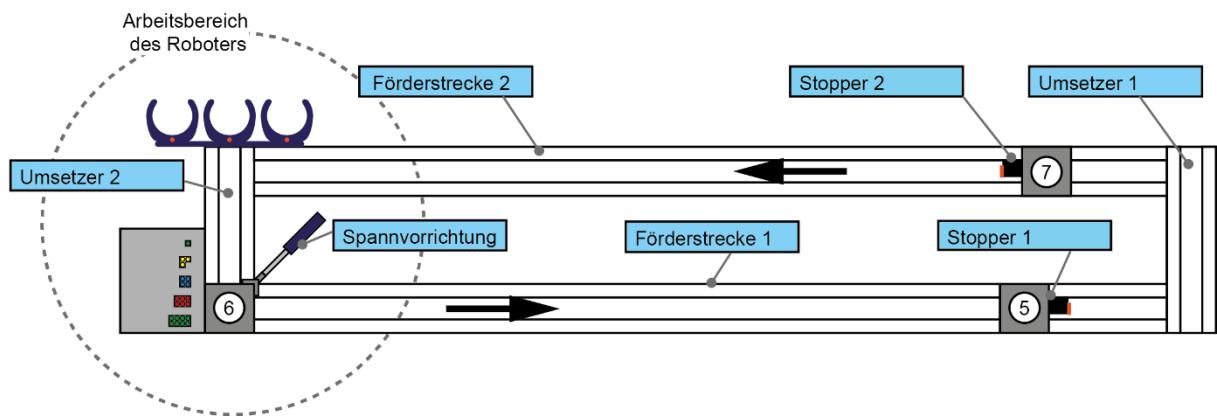


Abbildung 12-2: Das Förderband der Roboterzelle und dessen relevante CPDs aus der Draufsicht (Quelle: Eigene Ausarbeitung).

Der erweiterte Montagevorranggraph ist in der durch BrickCAD erzeugten Text-Datei beschrieben und wird zeilenweise auf die Robotersteuerung übertragen. Jeder eingehende Datenstring kann gespeichert und mit dem Production Graph verglichen werden. Darüber hinaus ist denkbar, ein virtuelles Modell der Lego-Baugruppe aufzubauen. Dies kann beispielsweise durch eine dreidimensionale

Binärmatrix erfolgen, bei der jeder Eintrag eine Lego-Noppe symbolisiert. Jeder vom Client versendete Datensatz wird analysiert und die Binärmatrix wird dementsprechend gefüllt. Auf Basis dieses Modells können beispielsweise die zurückzulegenden Wege der Sekundärprozesse minimiert werden.

12.2 Programmierung von CPDs und Funktionsbaugruppen

Im Folgenden wird betrachtet, wie ausgewählte CPDs und CPDCs programmiert werden können, damit diese in der Lage sind, die modellierten Funktionsprimitiva anzubieten. Dies wird noch vor der Identifikation der Primärprozesse betrachtet, da einleitend geklärt sein muss, ob der Production Graph, der der Betrachtung zu Grunde gelegt ist, in dieser Form umgesetzt werden kann.

12.2.1 Programmierung einzelner CPDs

Abbildung 12-3 veranschaulicht, wie beispielsweise Knoten 9 programmiert werden kann. Das Magazin zur Bereitstellung von Lego-Steinen des Typs A muss diese sowohl halten als auch lösen können. Als Halten wird dabei die obere Endlage des Zylinders definiert. Angesteuert werden die zugehörigen Adressbereiche des General I/O-Registers der SPS. Diese werden entweder mit einer logischen 0 oder einer logischen 1 beschrieben. Rückgabewert der Funktion ist ein Statusbit, das darüber Auskunft gibt, ob der Ausgang erfolgreich gesetzt worden ist.

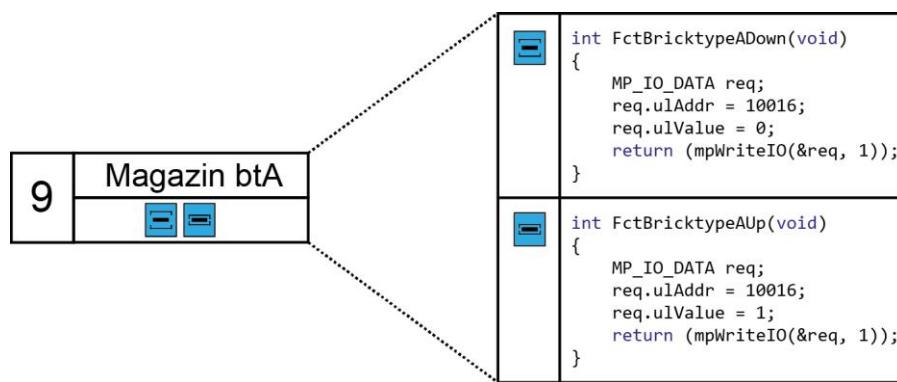


Abbildung 12-3: Implementierung der Funktionsprimitiva von Knoten 9
(Quelle: Eigenen Ausarbeitung).

In ähnlicher Weise können auch die beiden Umsetzer programmiert werden. Siehe hierzu Abbildung 12-4. Beide bieten das Funktionsprimitivum *Bewegen* von Werkstückträgern an. Die Steuerung dieser CPDs setzt sich aus zwei zeitlich aufeinander abzustimmenden Prozessen zusammen. Zuerst muss das

Untersuchung zur Implementierung des iwb-Modells auf der Robotersteuerung

Ventil des Umsetzers, das durch Ausgang 30044 der I/O-Baugruppe ansteuerbar ist, geschalten werden. Der Zylinder des Umsetzers bewegt sich dadurch in die vordere Endlage. Im Anschluss kann der Motor gestartet werden. Hat der Werkstückträger das Ende der Förderstrecke des Umsetzers erreicht, kann der Motor gestoppt und der Zylinder abgesenkt werden. In Abbildung 12-4 wird deutlich, wie diese Prozesse zeitlich aufeinander abzustimmen sind. Diese können in Abhängigkeit des zurückzulegenden Wegs und der Fördergeschwindigkeit angepasst werden.

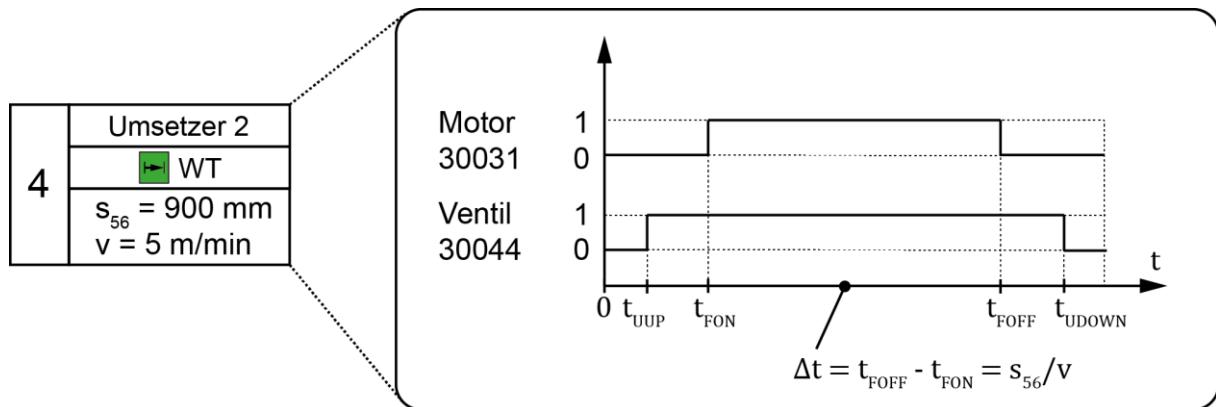


Abbildung 12-4: Zu implementierende Funktion zur Realisierung des Funktionsprimitiva Bewegen von Umsetzer 2 (Quelle: Eigene Ausarbeitung).

12.2.2 Programmierung einer CPDC

Die im Production Graph modellierte CPRC setzt sich aus dem Roboter, den drei Greifern und dem KMS zusammen. Mit Hilfe der Greifer können die Funktionsprimitiva *Halten* und *Lösen* angeboten werden. Abbildung 12-5 verdeutlicht dies.

Die Programmierung dieser beiden Funktionen erfolgt durch Ansteuerung des zugehörigen SPS-Registers 10015. Durch den am Roboter montierten KMS können die Funktionsprimitiva zum Messen der auftretenden Kräfte und Momente implementiert werden. Es ist zu hinterfragen, ob es zweckmäßig ist, den KMS von der FS100 anzusteuernd. Im gegenwärtigen Softwareentwurf ist der Leitrechner der aktive Teilnehmer, mit dessen Hilfe die Sensordaten per CAN-Bus abgefragt und per TCP-IP an die FS100 gesandt werden.

Im Production Graph hingegen ist die FS100 der aktive Teilnehmer, da Knoten 8 anbietet, dass Kräfte und Momente gemessen werden können. Dies impliziert, dass die Kraft- und Momentenmessung erst aktiviert wird, sobald das zugehörige Funktionsprimitivum benötigt wird. Die Softwarearchitektur müsste somit

Untersuchung zur Implementierung des iwb-Modells auf der Robotersteuerung

dahingehend verändert werden, dass per TCP-IP eine Anfrage an den Leitrechner geschickt wird, der diese Nachricht wiederum per CAN-Bus an den KMS weiterleitet. Die resultierende Antwort muss in umgekehrter Richtung an die FS100 übermittelt werden.

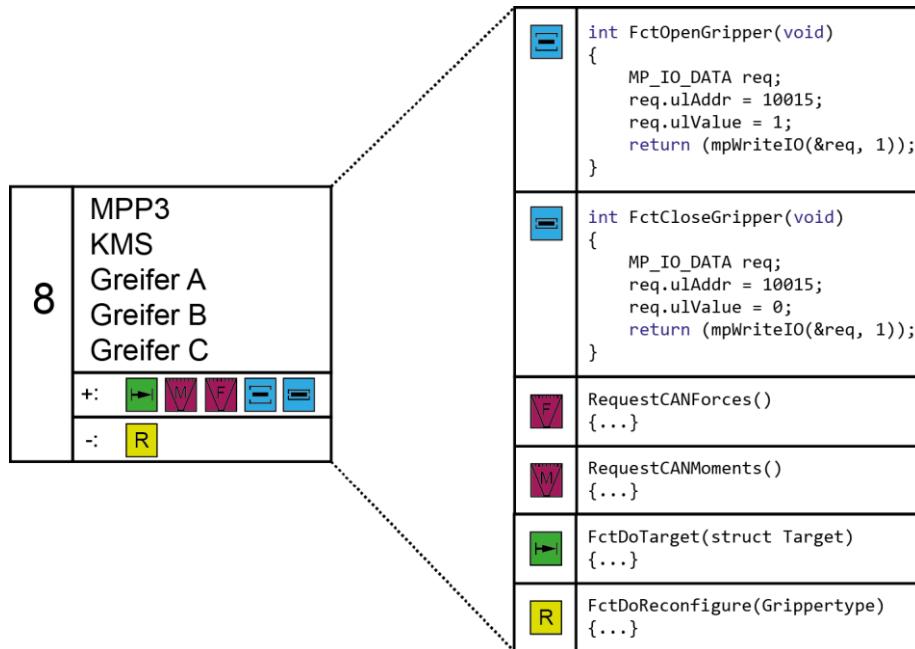


Abbildung 12-5: Die CPDC aus MPP3, KMS und den drei Greifern sowie die zu implementierenden Funktionsprimitive (Quelle: Eigene Ausarbeitung).

Dieser Entwurf ist im Gegensatz zum bestehenden Architekturmodell deutlich komplexer. Aus diesem Grund wird an dieser Stelle empfohlen, bei Bedarf auf die global verfügbaren Structs, in denen die aktuellen Kräfte und Momente gespeichert sind, zuzugreifen. Hierdurch kann der effizientere und bereits implementierte Softwareentwurf beibehalten werden. Sobald die Verbindung zwischen FS100 und Leitrechner aktiv ist, werden die auftretenden Kräfte und Momente übertragen und bei Bedarf dieser aus der globalen Variable ausgelesen.

Das Funktionsprimitivum *Bewegen* kann auf Basis der in Kapitel 11.4.4.10 erläuterten Funktion *FctDoTarget* ebenfalls ohne nennenswerten Aufwand implementiert werden. Als Parameter müssen hierzu lediglich die Geschwindigkeit des Roboters, das Bezugskoordinatensystem sowie der Vektor zum Zielpunkt spezifiziert werden.

Beim Vergleich des Production Graphs mit den Anforderungen eines Datensatzes des eMVG sind die fünf Funktionsprimitive *Halten*, *Lösen*, *Kraft messen*, *Moment messen* und *Bewegen* von Bedeutung. In Abhängigkeit des notwendigen Greifers muss die CPDC rekonfiguriert werden. Ein entsprechendes Funktionsprimitivum ist in Abbildung 12-5 vorgesehen. Da die Rekonfiguration eines

Untersuchung zur Implementierung des iwb-Modells auf der Robotersteuerung

CPDC nicht mit anderen Knoten in Verbindung treten muss, ist dies in der Darstellung als CPDC-internes Funktionsprimitivum eingetragen. Dies wird durch das vorangestellte Minus-Zeichen kenntlich gemacht. Diese Darstellungsform ist an die Verfügbarkeit von Methoden in UML-Klassenmodellen angelehnt.

Die Rekonfiguration der Roboterzelle beschränkt sich im gegebenen Fall auf die Durchführung notwendiger Greiferwechsel. Dies kann mit der in Kapitel 11.4.4.6 vorgestellten Funktion FctGripperSelection durchgeführt werden. Alternativ ist auch ein erweitertes Petri-Netzwerk denkbar. Dieses ist in Abbildung 12-6 dargestellt.

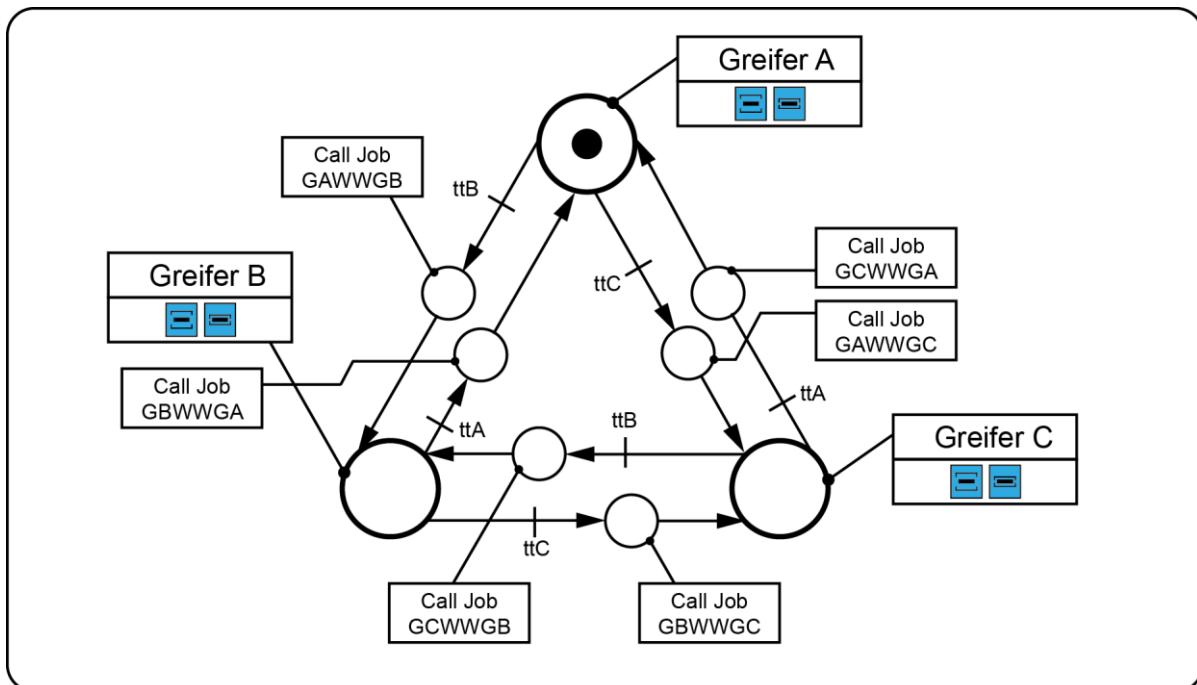


Abbildung 12-6: Petri-Netz zur Rekonfiguration der CPDC 8
(Quelle: Eigene Ausarbeitung).

Den drei Hauptstellen des Netzwerks sind die Funktionsprimitiva *Halten* und *Lösen* zugeordnet. Jeder Stelle ist dabei ein Greifer als CPD zugewiesen. Darüber hinaus kann jede Stelle von jedem Punkt im Petri-Netz erreicht werden. Transition ist immer die jeweils zugeordnete Angabe des notwendigen Greifertyps im eMVG-Datensatz. Auf jeder Kante zwischen zwei Hauptstellen ist eine Nebenstelle zu finden. In dem dargestellten Petri-Netz sind diese als Kreise mit geringeren Durchmesser dargestellt.

Nebenstellen beinhalten den notwendigen Job-Aufruf zum durchzuführenden Greiferwechsel. Dabei wird weiterhin auf die in Kapitel 9.5 erläuterten Job-Daten zugegriffen. Es erscheint nicht zweckmäßig, die zum Greiferwechsel not-

wendigen Pfade auf Basis eines auf der FS100 implementierten Modells automatisch zu generieren. Dies ist einerseits dem hohen Aufwand geschuldet, der nicht in Relation zum Teachen der sechs benötigten Job-Dateien steht. Andererseits ist nach wie vor unklar, inwieweit die Übertragung von CAD-Modellen auf die Steuerung überhaupt möglich ist. Des Weiteren sind die Greifer mit flexiblen Bauelementen, wie Pneumatikschläuchen und Sensorkabel versehen. Diese stehen frei im Raum und müssen beim Werkzeugwechsel berücksichtigt werden. Deren Verhalten während der Roboterbewegung kann insbesondere bei hohen Geschwindigkeiten nur schwer modelliert werden.

12.2.3 Programmierung der Funktionsbaugruppe Förderband

Die Programmierung des Förderbands mit Funktionsprimitiva ist ohne ein zugehöriges Erweiterungsmodell nicht möglich. Dies ist damit zu begründen, dass die einzelnen Elemente miteinander gekoppelt sind. Zum Transport eines Werkstückträgers von Knoten 5 zu Knoten 7 müssen die beiden Förderstrecken als auch Umsetzer 2 den Werkstückträger bewegen. Da Knoten 7 allerdings mit einem Werkstückträger belegt ist, muss dieser gleichzeitig durch Förderstrecke 1 zu Knoten 6 bewegt werden. Bei beiden, parallel ablaufenden Prozessen muss Förderstrecke 1 *Bewegen*.

Aus der beispielhaft dargelegten Problematik lassen sich Restriktionen ableiten, die mit Hilfe eines adaptierten Petri-Netzwerks modelliert werden können. Siehe dazu Abbildung 12-7.

Die CPDs 5, 6 und 7 sind den Stellen des Petri-Netzwerks zugeordnet. Alle Stellen sind durch einen Werkstückträger belegt. Darüber hinaus muss dem Modell als Nebenbedingung zu Grunde gelegt sein, dass jede Stelle nur einen Werkstückträger aufnehmen kann. Ein klassisches Petri-Netz nach LESZAK & EGGERT (1989) wäre in der vorliegenden Konstellation blockiert, da jede Stelle nur einen Werkstückträger aufnehmen und alle Stellen besetzt sind.

Das vorliegende Modell wird daher dahingehend erweitert, dass Kanten temporär Stellen aufnehmen können. Darüber hinaus sind die zum Transport notwendigen CPDs den Kanten zugeordnet. Soll ein Werkstückträger von Knoten 7 nach Knoten 5 bewegt werden, so kommen also die CPDs 1, 2 und 3 zum Einsatz. Als Übergabeparameter der CPDs sind sowohl die zurückzulegende Förderstrecke als auch die Geschwindigkeit des Förderbands hinterlegt. Diese werden, wie in Abschnitt 12.2.1 gezeigt, beispielsweise bei der Steuerung der Umsetzer berücksichtigt.

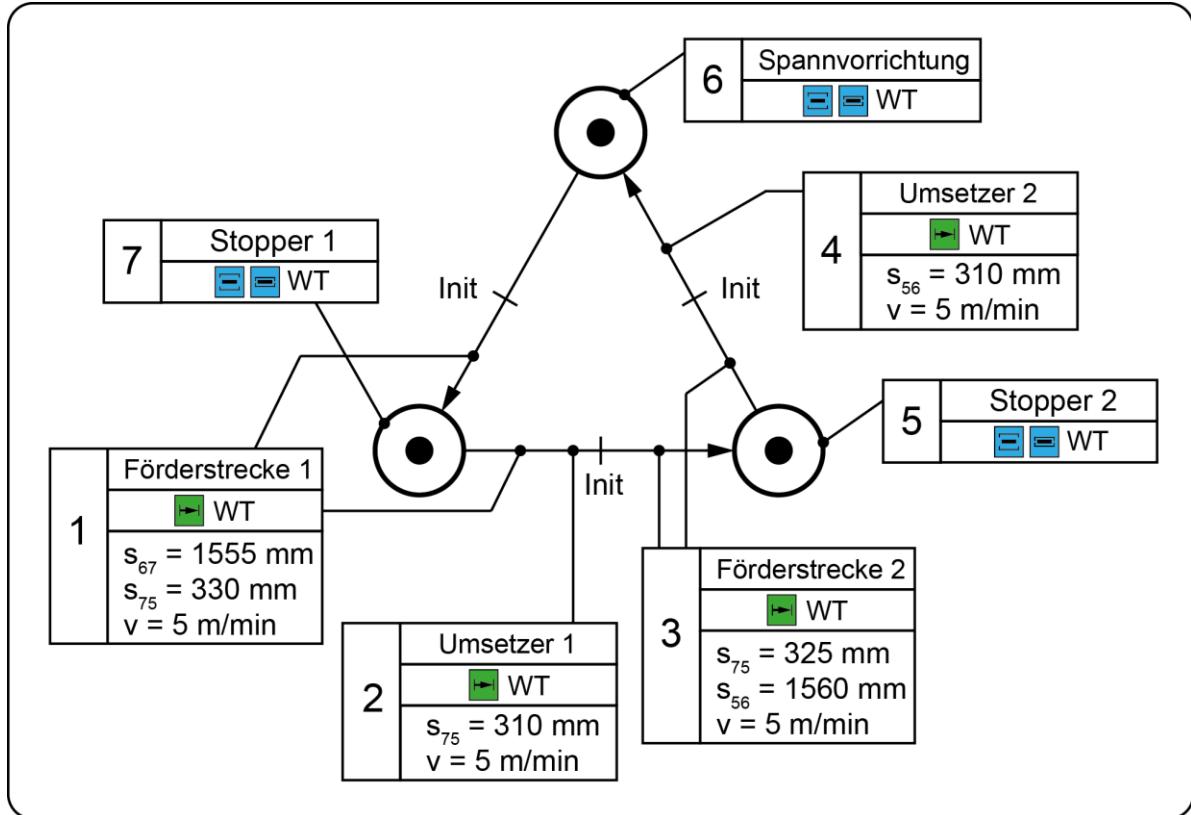


Abbildung 12-7: Petri-Netz des Förderbands zur Modellierung der Restriktionen (Quelle: Eigene Ausarbeitung).

Bei Übergang eines Werkstückträgers ist die Reihenfolge zu beachten, in der die CPDs der Kante zugeordnet sind. Ist ein CPD zwei Kanten zugeordnet, so muss betrachtet werden, zu welchem Zeitpunkt das CPD zum Einsatz kommt. Weiterer Betrachtungspunkt ist die Dauer, wie lange die Kante das CPD benötigt.

12.3 Automatische Konfiguration der CPRC

Wie in den Kapiteln 12.1 und 12.2 dargelegt, können die Fähigkeiten der CPDs anhand ihrer Funktionsprimitive beschrieben und der Production Graph modelliert werden. In den beiden vorangegangenen Kapiteln wurden die Funktionen für die einzelnen CPDs jedoch weiterhin von Hand programmiert. Dies wird beispielweise durch Abbildung 12-5 ersichtlich. Die Funktion FctOpenGripper ist speziell dazu vorgesehen, den Greifer am Roboter zu öffnen. Streng genommen ist dieser jedoch ein eigenes CPD. Der Steuerungscode hierfür muss allerdings auf der Robotersteuerung vorliegen, der zur Laufzeit nicht verändert werden kann, da dieser mit MotoPlusSDK kompiliert und händisch per USB-Stick auf die Robotersteuerung übertragen werden muss.

Untersuchung zur Implementierung des iwb-Modells auf der Robotersteuerung

Sollen sich die einzelnen CPDs möglichst ohne Zutun des Programmierers autonom vernetzen und den PG erstellen, so muss das bestehende CPD Modell erweitert werden.

Hierzu müssen neben der Beschreibung der Fähigkeiten des Devices mittels Funktionsprimitiva noch weitere Informationen hinterlegt werden. Für die Greifer muss im konkreten Anwendungsfall beispielsweise modelliert werden, welche Bauteile diese greifen können. Darüber hinaus ist eine Schnittstellenbeschreibung erforderlich. Die mechanische Schnittstelle des Greifers besteht aus einer Beschreibung des angebrachten Greiferwechselsystems. Das entsprechende Gegenstück muss auch am Roboter angebracht sein, damit die CPDs des Greifers und des Roboters zu einer CPDC kombiniert werden können. Hierzu bietet sich an, die mechanische Schnittstelle als String zu beschreiben, der die Typenbezeichnung des Greiferwechselsystems beinhaltet. Des Weiteren muss modelliert sein, dass der Greifer Druckluft als Arbeitsmedium benötigt. Diese kann und muss vom Roboter per Greiferwechselsystem bereitgestellt werden, was im VR des CPDs hinterlegt sein muss. Da das Programm der Robotersteuerung zur Laufzeit nicht mehr verändert werden kann, müssen Funktionen implementiert sein, mit der die Funktionalitäten der vorhandenen Schnittstellen angesteuert werden können. Diese Funktionen werden als Enabler bezeichnet. Für das Greiferwechselsystem bedeutet dies, dass zwei Funktionen vorhanden sind, mit denen Druckluft an den Anschlüssen 1 und 6 bereitgestellt werden kann. Um die Funktionsprimitiva des Greifers verwenden zu können, muss daher abgeglichen werden ob die Enabler zur Nutzung der Schnittstellen vorhanden sind.

Damit zwei CPDs automatisch miteinander verknüpft werden können, reicht somit nicht nur eine bloße Übereinstimmung der Schnittstellen. Im Beispiel der CPDs von Greifer und Roboter sind die Schnittstellen das Greiferwechselsystem und die zum Betrieb des Greifers benötigte Druckluft. Um die Funktionsprimitiva des Greifers verwenden zu können muss darüber hinaus sichergestellt werden, dass die Enabler der Schnittstellen durch den zugeordneten Knoten implementiert sind. Im konkreten Beispiel kann der Roboter Druckluft über Anschluss 1 des Greiferwechselsystems bereitstellen. Der Greifer benötigt Druckluft an Anschluss 1 um das Funktionsprimitivum *Halten* anbieten zu können. Ist der Abgleich der Schnittstellen und der Enabler beider CPDs erfolgreich, so können diese dem PG als CPDC hinzugefügt werden. Neben der Beschreibung der Schnittstellen und der Enabler müssen beispielsweise auch die relevanten CPD-Abmessungen hinterlegt sein. Für einen Greifer sind dies unter anderem die hinterlegten Werkzeug-Offsetvektoren. Die vollständige VR eines Greifers sowie ein Ausschnitt des VRs des MPP3s sind in Abbildung 12-8 illustriert.

Untersuchung zur Implementierung des iwb-Modells auf der Robotersteuerung

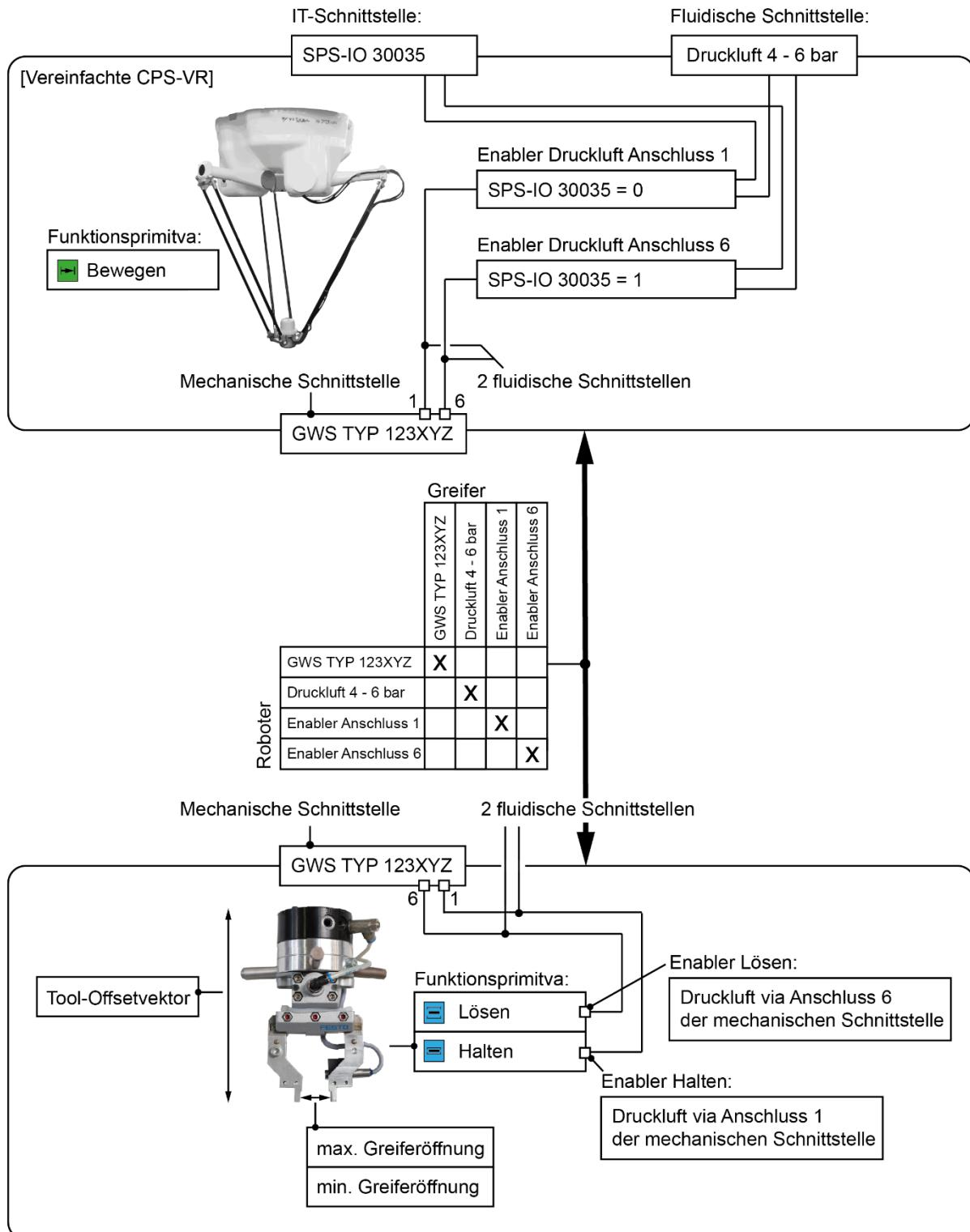


Abbildung 12-8: VR von Greifer und Roboter inklusive Abgleich der Schnittstellen und Enabler (Quelle: Eigene Ausarbeitung).

Die Programmierung von aktiven CPDs, mit deren Hilfe die Funktionsprimitive anderer CPDs nutzbar gemacht werden können, muss vorab erfolgen. Im Falle des MPP3 kann beispielsweise auf der Robotersteuerung hinterlegt werden, dass Druckluft an Anschluss 1 des Greiferwechselsystems dann zur Verfügung

Untersuchung zur Implementierung des iwb-Modells auf der Robotersteuerung

steht, wenn Ausgang 30035 der SPS eine logische 0 führt. Durch diese Modellierung kann jedes beliebige CPD das die Schnittstellenbeschreibung erfüllt mit dem MPP3 verknüpft und im PG konfiguriert werden. Der Programmcode muss nicht angepasst werden.

Hinsichtlich der Speicherung der erweiterten VRs der CPDs werden an dieser Stelle zwei Varianten vorgeschlagen. Zum einen können die digitalen Modelle der einzelnen Devices auf digitalen Speichern abgelegt werden, die an den einzelnen Baugruppen der CPRC angebracht sind. Wird die CPRC durch eine Baugruppe erweitert, so wird der Speicher ausgelesen und der PG um das Modell des hinzugekommenen CPDs erweitert. Dieser Ansatz ist an die Bestrebung hinsichtlich der Umsetzung von Plug & Produce angelehnt. Alternativ kann eine zentrale Datenbank erstellt werden, die die digitalen Modelle aller verfügbaren Baugruppen enthält. Wird per SPS-Eingangssignal festgestellt, dass beispielsweise ein Greifer zur Verfügung steht, so wird der PG um den Greifer erweitert. Das Vorhandensein von Baugruppen ohne SPS-Eingang wie die Zuführeinheit müsste vom Anwender angegeben werden.

Programmteile zu komplexen Funktionsbaugruppen mit Restriktionen, die nur schwer zu modellieren sind, müssen trotz vollständigen VRs weiter vom Programmierer geschrieben werden. Alternativ bietet sich die Programmierung von CPDCs an, die die Restriktionen beispielsweise durch Petri-Netz wie in Kapitel 12.2.3 gezeigt, bereits beinhalten.

Hinsichtlich der vollständigen automatischen Programmierung der Roboterzelle kann auf Basis eines autonom konfigurierten PGs mit den gegenwärtigen Methoden nur ein grobes Programmgerüst vorgegeben werden. Mit Hilfe der CAD-Daten und den virtuellen Repräsentanzen der Baugruppen können mit den vorgestellten Methoden die relevanten Punkte im Arbeitsraum ohne weiteres abgeleitet werden. Für die gegebene Roboterzelle und die Montage von Lego könne die Punkte P1 bis P5 identifiziert und berechnet werden. Diese Punkte unterliegen allerdings den Mängeln der Offline-Programmierung. Abweichungen der realen Roboterzelle gegenüber dem im PG beschriebenen Modell können bei der automatisierten Programmgenerierung nicht berücksichtigt werden, da diese nicht modellierbar sind. Nach der Programmerstellung muss dieses daher vom Betriebspersonal angepasst werden. Bestes Beispiel ist der Vektor zur Verschiebung der User-Koordinatensysteme. Dieser Vektor fließt in der implementierten Softwarelösung in die Berechnung der Punkte im Arbeitsraum mit ein. Wird die Zuführeinrichtung beispielsweise durch eine Kollision mit dem Ro-

bofer verschoben, so muss dieser Vektor vom Programmierer angepasst werden. Eine solche Anpassung muss auch für ein autonom erstelltes Programm vorgenommen werden.

12.4 Identifikation und Zuweisung von Primär- und Sekundärprozessen

Auf Grundlage der eingehenden Datensätze des eMVG in Verbindung mit dem Production Graph können im Anschluss die Primär- und Sekundärprozesse identifiziert werden. Hierbei müssen die implementierten Modelle zur Modellierung der Restriktionen berücksichtigt werden. Abbildung 12-9 veranschaulicht den Montageplan zum Fügen eines Lego-Steins.

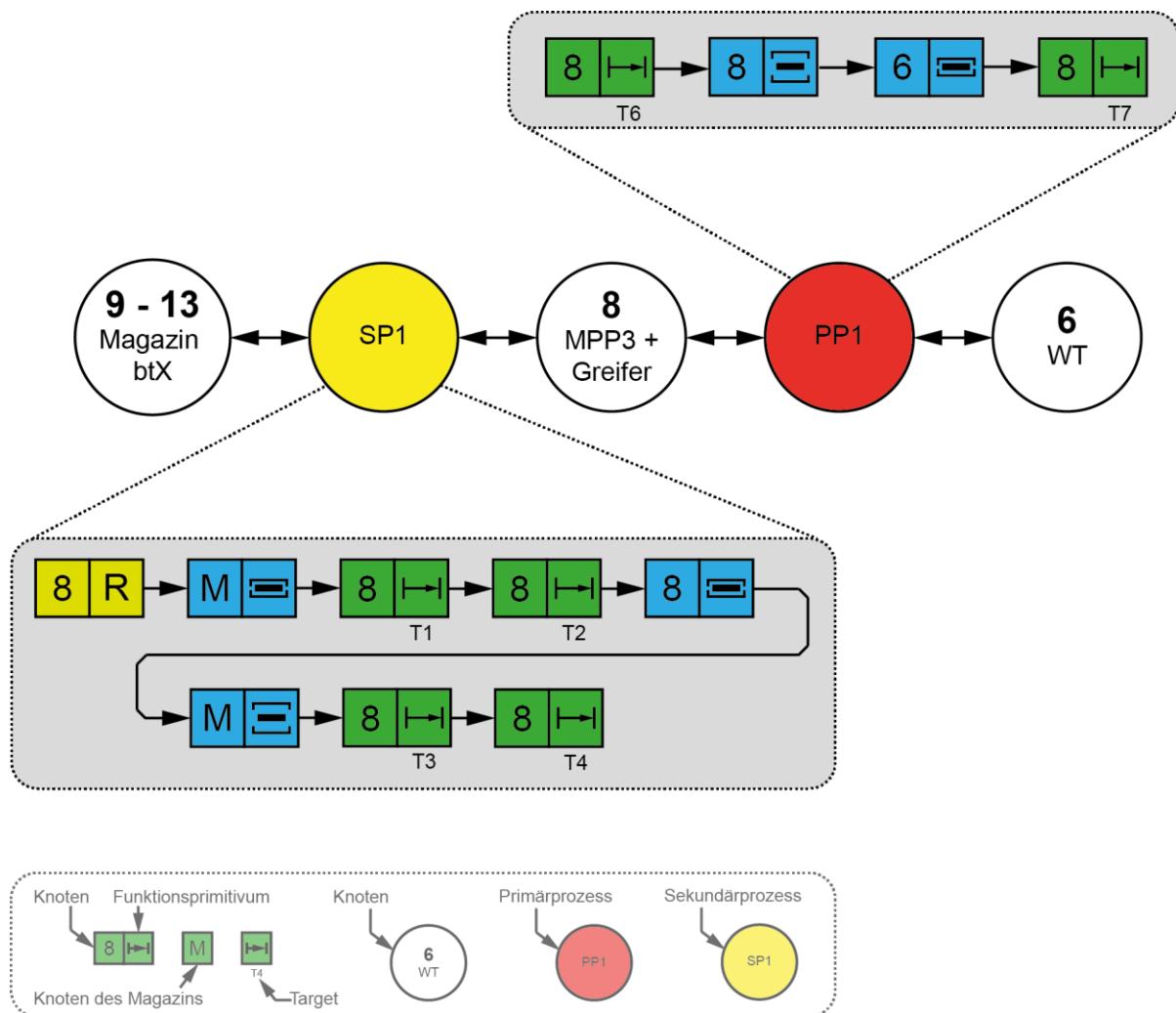


Abbildung 12-9: Montageplan und benötigte Funktionsprimitive zum Fügen eines Lego-Steins (Quelle: Eigene Ausarbeitung).

Untersuchung zur Implementierung des iwb-Modells auf der Robotersteuerung

Die zur automatisierten Identifikation dieser Prozesse notwendige Programmlogik kann auf Basis der von KRAUS (2015) entwickelten Methodik implementiert werden. Da Stein für Stein von der Roboterzelle gefügt wird, beinhaltet die zu betrachtende Prozesskette immer nur einen Primärprozess. Dieser beschreibt den Fügevorgang eines Lego-Steins. Am Primärprozess sind der Werkstückträger des Knoten 6 sowie die CPDC von Knoten 8 beteiligt. Die zum Fügen notwendigen Funktionsprimitiva können wie folgt identifiziert werden.

Knoten 6 muss die Lego-Steine aufnehmen und kann aufgrund der implementierten Nebenbedingung Lego-Steine halten. Diese Lego-Steine werden vom Greifer von Knoten 8 übergeben. Zusätzlich müssen Montage- und Retreat-Pfad ermittelt und bei Verwendung des Funktionsprimitivums *Bewegen* spezifiziert werden.

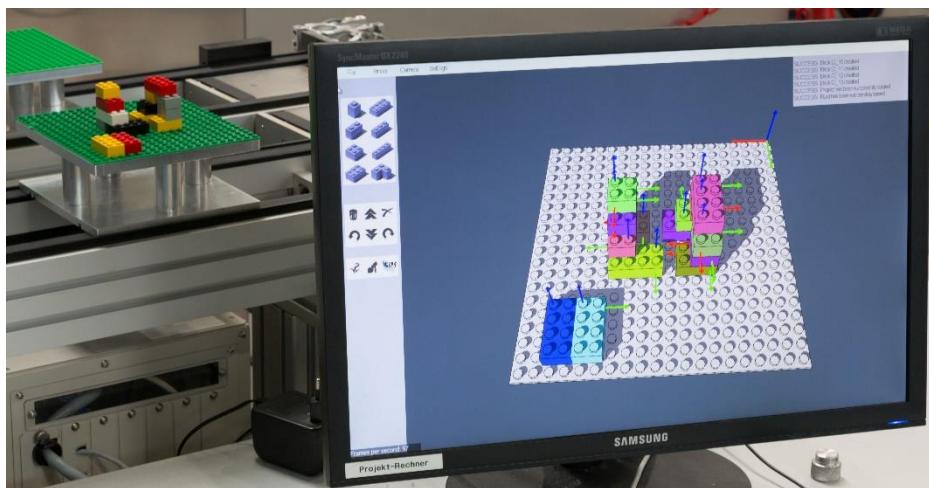
Die Pose bei Übergabe des Lego-Steins lässt sich anhand eines BrickCAD-Datensatzes auf Grundlage der in Kapitel 11.4.4.4 beschrieben Funktion ermitteln. Die dazu notwendigen Bewegungen können aus dieser Pose generiert werden, sofern hinterlegt ist, dass die zum Fügen notwendige Bewegung immer in positiver z-Richtung zu erfolgen hat. Der Retreat-Pfad muss dementsprechend in negativer z-Richtung berechnet werden. Wie weit sich der Roboter zurückzieht, kann durch eine Obergrenze ermittelt werden, die festlegt wie hoch eine Lego-Baugruppen maximal sein darf. Dies hat allerdings zur Folge, dass die zum Fügen notwendigen Wege nicht minimiert werden. Diesem Mangel kann entgegengetreten werden, indem die angesprochene Binärmatrix implementiert wird. Mit dieser können die bereits gefügte Lego-Baugruppe modelliert und die Montagewege minimiert werden.

Nach dem Primärprozess kann ein Sekundärprozess identifiziert werden. Dieser interagiert mit dem jeweiligen Knoten des Magazins und der CPRC von Knoten 8. Dazu muss zuerst geprüft werden, ob die Roboterzelle rekonfiguriert werden muss. Im Anschluss daran kann die Montageeinheit einen Legostein halten und diesem hierdurch der CPRC zur Verfügung stellen.

Die Übergabepose kann mit der in Kapitel 11.4.4.3 beschriebenen Funktion ermittelt werden. Auf Grundlage dieser Pose können die Targets T1 und T2 berechnet werden. Die Restriktion, dass lediglich Bewegungen in z-Richtung möglich sind, ermöglicht diese Berechnung. Nachdem der Greifer den Lego-Stein hält, kann das Magazin das Funktionsprimitivum *Lösen* ausführen. Im Anschluss daran kann der Roboter in negativer z-Richtung verfahren werden. Um die Lücke zum Primärprozess zu schließen, muss der Roboter als Abschluss des Sekundärprozesses zu Target 4 bewegt werden.

13 Validierung

Die implementierte Software wurde im Rahmen dieser Arbeit anhand zahlreicher Tests validiert. Dazu wurden einzelne, für sich stehende Funktionen in dafür erzeugten Testumgebungen, sowohl virtuell mittels Visual Studio als auch real mit Hilfe der Roboterzelle getestet. Des Weiteren wurden im Rahmen einer Versuchsreihe 15 Lego-Baugruppen mit BrickCAD erstellt und durch die Roboterzelle gefügt. Abbildung 13-1 zeigt eine solche Lego-Baugruppe sowie das von der Roboterzelle gefügte Produkt.



*Abbildung 13-1: Konstruktionsmodell einer Lego-Baugruppe in BrickCAD und das zugehörige, durch die CPRC gefügte Produkt im Hintergrund
(Quelle: Eigene Ausarbeitung).*

Im Folgenden werden die Ergebnisse anhand einer exemplarischen Baugruppe dargelegt. Abbildung 13-2 zeigt diese. Es kann festgehalten werden, dass die in Kapitel 7 formulierten Anforderungen an die Software erfüllt werden. Das Lego-Modell wurde mit BrickCAD erstellt. Der eMVG in Form der build-File wurde an die Robotersteuerung per Ethernet versendet. Darüber hinaus kann der Demonstrator mit Hilfe dem in Abbildung 10-1 dargestellten GUI gesteuert werden. Das Interface gibt wie gefordert den Systemzustand der Roboterzelle wieder. Parallel zum Fügeprozess werden die am KMS auftretenden Kräfte gemessen und per CAN-Bus an den Leitrechner versendet. Mit Hilfe des Clients werden diese einerseits als Text-Datei auf dem Leitrechner gespeichert, andererseits werden die Daten per Ethernet an die Robotersteuerung weitergeleitet.

Lego-Steine der Typen C, D und E werden durch die Roboterzelle grundsätzlich mit hinreichender Präzision gefügt. Aufgrund diverser Abweichungen können dabei teilweise hohe Spaltmaße zwischen den Lego-Steinen auftreten. Ein solcher Fehler ist in Abbildung 13-2 zu sehen.

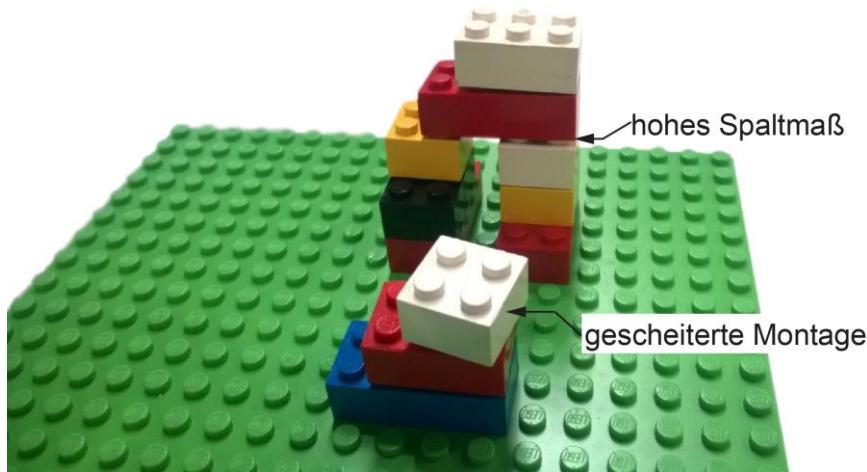


Abbildung 13-2: Lego-Baugruppen mit hohen Spaltmaßen und einem Stein, der nicht korrekt gefügt wurde (Quelle: Eigene Ausarbeitung).

Darüber hinaus scheitert die Montage dieser Bauteile in wenigen Fällen. Lego-Steine der Typen A und B können mit der Roboterzelle gegenwärtig nicht gefügt werden, da diese Bauteile relativ klein und weniger fehlertolerant hinsichtlich der Positioniergenauigkeit sind als die größeren Lego-Steine. Im Rahmen der Validierung wurde getestet, ob dieser Mangel aus den Unzulänglichkeiten der Offline-Programmierung herröhrt. Dazu wurde ein Programm online geteacht, optimiert und mehrfach wiederholt. Hierbei konnte beobachtet werden, dass selbst mit einem optimierten, online erstellten Programm Steine des Typs A und B nur selten erfolgreich gefügt werden können.

Ein Defizit, das unter anderem der Offline-Programmierung angerechnet werden kann, ist die nur selten erfolgreiche Montage von Lego-Steinen mit direktem Kontakt an den Seitenflächen. Sollen zwei Lego-Steine unmittelbar nebeneinander auf dem Werkstückträger gefügt werden, so scheitert die Montage oftmals. Abbildung 13-3 zeigt diese Problematik.

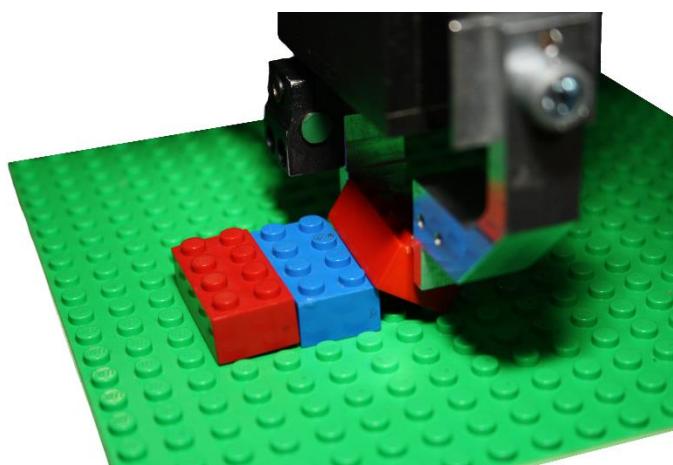


Abbildung 13-3: Fehlerhafte Montage beim Platzieren zweier Lego-Steine nebeneinander (Quelle: Eigene Ausarbeitung).

Der veranschaulichte Vorgang ist einerseits grundsätzlich problematisch, da die Lego-Steine keine Einführschrägen aufweisen, die diesen Prozess unterstützen können. Andererseits kommen Abweichungen der Form- und Lagetoleranzen der Lego-Steine, Ungenauigkeiten bei Steinaufnahme in der Zuführeinheit als auch bei der Montage auf dem Werkstückträger zum Tragen. Die Regelungsgrenzen und Rechengenauigkeit der Robotersteuerung spielen ebenso eine Rolle wie auch Einflüsse aufgrund von Erwärmung, Dynamik und weiteren Umwelteinflüssen.

Die bereits beschriebenen Systemdefizite wurden bei der durchgeführten Versuchsreihe beachtet. Bei der Konstruktion der 15 Baugruppen mit BrickCAD wurden nur Lego-Steine der Typen C, D und E verwendet. Des Weiteren wurden keine Lego-Steine nebeneinander platziert. Jede zur Validierung herangezogene Lego-Baugruppe besteht aus 10 Steinen. Im Rahmen der Versuchsreihe wurden daher 150 Lego-Steine gefügt. Die Montage dieser Steine war in 134 Fällen erfolgreich. Somit konnten 89,3 % aller Lego-Steine erfolgreich gefügt werden. Grundsätzlich kann festgehalten werden, dass Lego-Steine höherer Ebenen der Baugruppe öfter erfolgreich gefügt werden können, da die bereits montierten Steine die auftretenden Systemabweichungen kompensieren können. Alle notwendigen Greiferwechsel zur automatisierten Anlagenkonfiguration wurden korrekt durchgeführt. Die Montage vollständiger Lego-Baugruppen war in 8 von 15 Fällen zufriedenstellend, was einer Quote von 53,3 % entspricht. Zur Durchführung der Versuchsreihe waren insgesamt 17 Anläufe notwendig. Zwei Versuche wurden aufgrund von Kollisionen von Roboter und Lego-Stein bei Aufnahme des jeweils ersten zu fügenden Lego-Steins abgebrochen. Diese Problematik ist auf die unzureichende Anbringung der Zuführeinheit am Förderband zurückzuführen. Die Systemstabilität der CPRC liegt somit bei 88,3 %. Eine Übersichtstabelle zur durchgeführten Versuchsreihe ist im Anhang unter Abschnitt 17.7 zu finden.

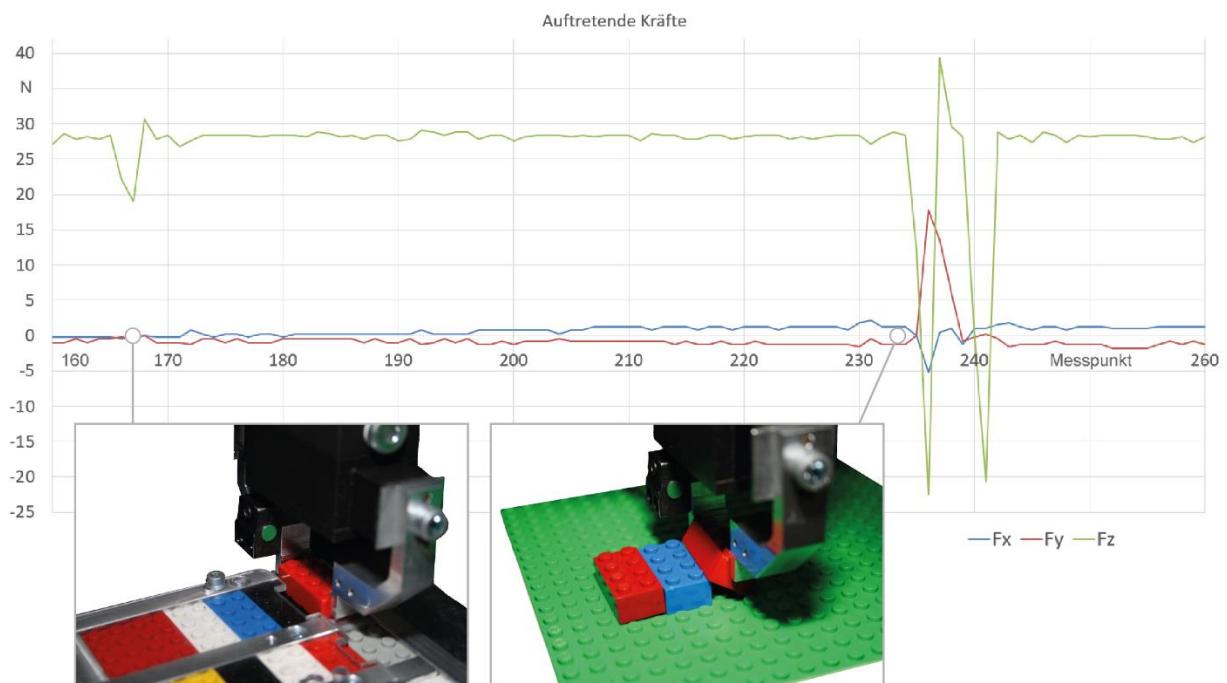
Anhand dieser Tests kann darüber hinaus nachgewiesen werden, dass die implementierte Architektur nach MICHNIEWICZ & REINHART (2015B) die formulierten Vorteile mit sich bringt. Mit Hilfe von BrickCAD wird anhand der CAD-Daten der montierten Lego-Baugruppe und anhand der VR der verfügbaren Greifer ein validier Montagevorranggraph erstellt. Die Daten werden dabei automatisch aus dem CAD-Programm extrahiert. Der Systementwurf gewährleistet hohe Flexibilität, da die Programmierung der Roboterzelle automatisiert und aufgabenorientiert erfolgt. Die Montage variantenreicher Lego-Baugruppen in Losgröße Eins wird auf der Grundlage Cyber-Physischer Systeme ermöglicht. Wie gefordert können die Potentiale von CPS anhand des Demonstrators aufgezeigt werden.

14 Ausblick

Zur Steigerung des inhärenten Potentials des Demonstrators bieten sich die nachfolgenden Erweiterungen an. Zum einen kann der kraftgeregelte Betrieb implementiert werden, mit dessen Hilfe auch Lego-Steine in unmittelbarer Nachbarschaft gefügt werden können. Darüber hinaus können mit Hilfe einer neu konzeptionierten Zuführeinheit die verfügbaren Lego-Steine hinsichtlich Varianz und Anzahl erhöht werden. Aus wissenschaftlicher Sicht gilt es ein Konzept zu untersuchen, wie die Softwarearchitektur umgestaltet werden kann, so dass die Cyber-Physischen Systeme der Roboterzelle aktiv kommunizieren.

14.1 Implementierung kraftgeregelten Betriebs

In der implementierten Softwarelösung werden die vom KMS erhobenen Sensordaten an die FS100 übertragen. Diese werden gegenwärtig noch nicht zur Steuerung oder Regelung des Fügeprozesses genutzt. Um die Fügegenauigkeit zu erhöhen und den inhärenten Abweichungen der Offline-Programmierung des Roboters entgegenzutreten, empfiehlt sich daher der sensorbasierte, kraftgeregelte Betrieb. Es gilt zu untersuchen ob mit Hilfe der gemessenen Kräfte und Momente die auftretenden Abweichungen ausgeglichen werden können. Abbildung 14-1 zeigt die gemessenen Kräfte eines gescheiterten Fügevorgangs.



*Abbildung 14-1: Gemessene Kräfte eines gescheiterten Fügevorgangs
(Quelle: Eigene Ausarbeitung).*

Auf der x-Achse sind die durchnummerierten Messpunkte, auf der y-Achse die Kraft in N angetragen. Die Zeitspanne zwischen zwei Messpunkten beträgt 80 ms. Die gemessene Kraft von ca. 25 N in der z-Achse entspricht dem Eigengewicht der am KMS montierten Bauelemente wie z.B. Greifer und Greiferwechselsystem. Der Peak bei Messpunkt 168 resultiert aus der Aufnahme des Lego-Steins in der Zuführeinheit. Der Ausschlag um Messpunkt 235 ist die Folge des gescheiterten Fügevorgangs. Hierbei wurde versucht zwei Steine des Typs E unmittelbar nebeneinander zu platzieren.

Die während dieses Vorgangs aufgetretenen Momente können Abbildung 14-2 entnommen werden. Auf der x-Achse des Diagramms sind erneut die Messpunkte, auf der y-Achse die aufgetretenen Momente in Nm angetragen.

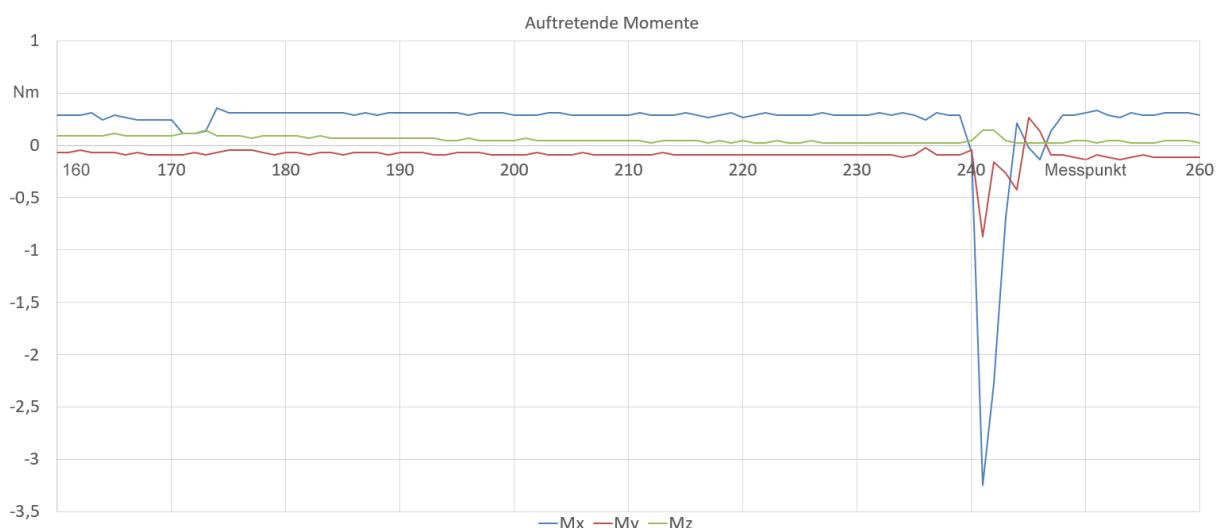


Abbildung 14-2: Gemessene Momente während des gescheiterten Fügevorgangs (Quelle: Eigene Ausarbeitung).

Hinsichtlich der Implementierung kraftgeregelten Prinzips werden im Folgenden drei zu prüfende Architekturen empfohlen.

Das Motion Control API von MotoPlusSDK funktioniert nach folgendem Prinzip: Die abzuarbeitenden Targets werden in einer FIFO-Warteschlange gespeichert. Mit Hilfe der Funktion `mpMotTargetReceive` wird während der Roboterbewegung geprüft, ob ein Target bereits erreicht wurde. Ist das Target erreicht, so wird die Abarbeitung des nachfolgenden Codes des Tasks fortgesetzt. Während die Funktion `mpMotTargetReceive` aufgerufen und das Target noch nicht erreicht ist, kann im Task keine andere Funktion aufgerufen werden. Die parallel zur Fügebewegung gemessenen Kräfte können in diesem Task somit nicht ausgewertet werden.

Hieraus ergibt sich, dass geprüft werden muss, in wie weit die Fügebewegung inkrementell vollzogen werden kann. Anstelle von Target 6 als Endposition des

Lego-Steins können Targets abgearbeitet werden, die inkrementell in z-Richtung der Endposition angenähert werden. Nachdem sich der Roboter um ein Inkrement verschoben hat, können die Kräfte und Momente im selben Task ausgewertet und eine Handlungsstrategie, beispielsweise in Form einer Ausgleichsbewegung, abgeleitet werden.

Neben diesem Ansatz muss überprüft werden, in wie weit die Funktion `mpMotTargetReceive` überhaupt notwendig ist. Nach der Übermittlung des Zieltargets und Starten der Abarbeitung des Targets in der Warteschlange mittels `mpMotStart` könnte zum Beispiel auch anhand der auftretenden Kräfte und Momente ermittelt werden, ob ein Lego-Stein platziert wurde. Sofern die auftretende Gegenkraft in z-Richtung einer definierten Fügekraft entspricht, gilt ein Target als abgearbeitet und der Zustandsautomat kann dazu übergehen, den nächsten Lego-Stein vom Client anzufordern. Diese Lösung bietet den Vorteil, dass die Fügebewegung als stetige Funktion mit einer parallelen Kraftregelung in einem Task durchgeführt werden kann. In wie weit dieser Ansatz durchführbar ist, muss allerdings erst validiert werden, da die Funktion `mpMotTargetReceive` bei Verwendung des Motion Control API grundsätzlich vorgesehen ist.

Als dritte Lösung zur Implementierung kraftgeregelten Betriebs wird an dieser Stelle vorgeschlagen, die Kraftregelung in einem vierten, unabhängigen Task zu modellieren. Hierdurch kann das Motion Control API gemäß dem vom Hersteller angedachten Prinzip verwendet und die Funktion `mpMotTargetReceive` weiter eingesetzt werden. In einem weiteren Task kann die Kraftregelung erfolgen, unabhängig von der Laufzeit des Codes zur Steuerung des Roboters. Dabei gilt es zu überprüfen, ob das Motion Control API taskübergreifend verwendet werden kann. Wird eine definierte Sollkraft überschritten, so muss der Task zur Kraftregelung die Abarbeitung der Targets in der Warteschlange beenden. Auf der FS100 können grundsätzlich drei Tasks normaler Priorität programmiert werden. Diese werden im gegebenen Softwareentwurf bereits verwendet. Zur Implementierung eines vierten, zur Kraftregelung bestimmten Tasks empfiehlt es sich beispielsweise den Task zur CAN-Bus-Kommunikation als prioren Task zu definieren. Dies hat zur Folge, dass dieser vom Scheduler bevorzugt abgearbeitet wird. Im Anschluss kann der Task zur Kraftregelung als Task normaler Priorität implementiert werden. Dieses Vorgehen empfiehlt sich, da Debugging via `printf`-Befehl im prioren Task nicht möglich ist.

Neben der Erweiterung der Softwarearchitektur muss auch geprüft werden, welche Regelstrategien auf Basis der erhobenen Sensordaten zur Verfügung stehen. Aufgrund zeitlicher Restriktionen seitens der Abarbeitung von Tasks auf der FS100 können die Sensordaten des KMS lediglich im Abstand von 200 ms

erhoben werden. Bei der momentan definierten Fügegeschwindigkeit von 20 mm/s werden somit zwischen zwei Messpunkten 4 mm zurückgelegt. Unter Berücksichtigung dieser Information, kann an dieser Stelle bereits vermutet werden, dass der kraftgeregelte Betrieb nur schwer umzusetzen ist. Um dieser Problematik entgegenzutreten, muss die Geschwindigkeit des Fügeprozesses verlangsamt werden. Für die Kraftregelung ist ohnehin lediglich die Bewegung des Roboters von Target 5 nach Target 6 relevant.

14.2 Entwicklung und Implementierung eines Lego-Stein-Zuführsystems mit höherer Flexibilität

Die Produktflexibilität der Roboterzelle ist gegenwärtig eher beschränkt. Dies ist der Tatsache geschuldet, dass mit Hilfe des Zuführsystems nur fünf verschiedene Lego-Steinarten zur Verfügung gestellt werden. Darüber hinaus können in jedem Magazin der Zuführeinheit maximal 8 bis 9 Lego-Steine gespeichert werden. Die Freiheitsgrade zur Produktgestaltung sind daher begrenzt. Darüber hinaus müssen die Magazine der Zuführeinheit relativ oft aufgefüllt werden.

Um diesem Mangel entgegenzutreten, bietet sich die Anbindung einer Zuführeinheit an, in der Lego-Steine variantenreich als Schüttgut gespeichert werden. Diese müssen dem Roboter lagerichtig bereitgestellt werden, da der Roboter aufgrund seines kinematischen Aufbaus die Steine lediglich um die Hochachse umorientieren kann. Die Steine müssen so vereinzelt sein, dass genug Platz zwischen den geförderten Steinen ist, um diese mittels Zweibacken-Parallelgreifer handhaben zu können.

Für eine Zuführeinheit, bei der die Steine lagerichtig auf einer definierten Fläche vereinzelt werden, ist ein Bilderkennungssystem notwendig. Mit Hilfe dessen können die benötigten Steine erkannt werden. Da durch BrickCAD vorgegeben wird, wie die Lego-Steine gegriffen werden müssen, ist hinsichtlich des Softwareentwurfs ein System notwendig, das die Bilderkennung, die build-File von BrickCAD und den Client auf dem Leinrechner miteinander in Verbindung setzt. Mit Hilfe der build-File müssen durch die Bilderkennung Greifpunkt und Steinorientierung ermittelt werden um die Steinaufnahme zu ermöglichen. Hierzu ist ein weiteres User-Koordinatensystem zweckdienlich. Die ermittelten Koordinaten müssen vom Client an die FS100 übertragen werden. Erst dann kann der Roboter den benötigten Lego-Stein aufnehmen. Hinsichtlich der Datenübertragung zwischen Client und Server empfiehlt es sich, den zu übertragenden Datensatz aus der build-File um den Aufnahmepunkt und die Steinorientierung zu erweitern.

14.3 Untersuchungen zur Implementierung aktiv kommunizierender CPDs

Der von MICHNIEWICZ & REINHART (2015A) vorgestellte Entwurf vereinfacht den CPS-Gedanken dahingehend, dass der Leitrechner als zentrale Systemkomponente dient. In nachfolgenden Arbeiten ist zu untersuchen, wie eine Systemarchitektur geschaffen werden kann, in der die einzelnen Devices als autonome Einheiten agieren. Nach dem bisherigen Softwareentwurf werden die Fähigkeiten der Zellenelemente vom Programmierer definiert. In Kooperation mit Forschungsansätzen von Plug & Play ist beispielsweise ein System denkbar, in dem jedes Device über einen Speicher verfügt, in dem die softwarespezifischen Eigenschaften modelliert sind. Sind dem Roboter die Greiferaufnahmepositionen beispielsweise bekannt, so ist denkbar, dass eine CPDC sich selbst erzeugt, indem der Roboter alle Greifer im Greiferbahnhof im Rahmen der Initialisierung einwechselt und dem PG ein Knoten als CPDC hinzugefügt wird. Dieser beinhaltet die CPS-Modelle der Greifer. Im Anschluss daran muss untersucht werden, wie die CPDs der Roboterzelle dem Leitrechner ihre Eigenschaften übermitteln können, sodass der Production-Graph durch aktive CPS-Elemente automatisch auf dem Leitrechner erzeugt wird. Auf Grundlage des autonom erzeugten Production-Graphs könnte in Verbindung mit BrickCAD der eMVG erzeugt und an die Robotersteuerung übertragen werden. Im Anschluss dran müssten, wie in Kapitel 12 dargelegt, die Primär- und Sekundärprozesse identifiziert und die CPDs auf Grundlage der verfügbaren Funktionsprimitiva gesteuert werden.

15 Zusammenfassung

Im Rahmen dieser Arbeit wurde eine Software entwickelt und implementiert, mit der die aufgabenorientierte Programmierung einer Cyber-Physischen-Roboter-Zelle möglich ist. Der von BrickCAD bereitgestellte, erweiterte Montagevorrang-graph wird mit Hilfe des für den Leitrechner programmierten Clients an die Robotersteuerung übermittelt. Die per Ethernet übermittelten Daten werden von der auf der Robotersteuerung implementieren Software analysiert und die Lego-Baugruppe gemäß des von BrickCAD modellierten eMVGs gefügt.

Die Steuerung der Roboterzelle erfolgt über ein adäquates Interface, das vom Leitrechner aus bedient werden kann. Darüber hinaus wird die Software im Rahmen dieser Arbeit mit Hilfe von UML2 dokumentiert.

Hinsichtlich der erwarteten Vorteile von CPS kann festgehalten werden, dass mit Hilfe von BrickCAD die Durchführbarkeit des Montageprozesses bereits während der Konstruktion geprüft und validiert wird. Des Weiteren kann anhand des Demonstrators die automatische Konfigurationsauswahl des Produktionsystems und die effektive Nutzung der Anlagenflexibilität veranschaulicht werden. Zusammenfassend lässt sich sagen, dass anhand des Demonstrators und mit Hilfe der entwickelten Software die Potentiale des von MICHNIEWICZ & REINHART (2015A) entwickelten Architekturmodells aufgezeigt werden können. Die Vorteile dieser aufgabenorientierten Roboterprogrammiermethode werden deutlich und können an einer realen Roboterzelle validiert werden.

16 Literaturverzeichnis

ABELE & REINHART 2011

Abele, E.; Reinhart, G.: Zukunft der Produktion. Herausforderungen, Forschungsfelder, Chancen. München: Carl Hanser 2011. ISBN: 978-3-446-42595-8.

ACATECH 2011

acatech (Hrsg.): Cyber-Physical Systems. Innovationsmotor für Mobilität, Gesundheit, Energie und Produktion. Berlin, Heidelberg: Springer 2011. ISBN: 978-3-642-29090-9.

BACKHAUS 2014

Backhaus, J.: Adaptierbares aufgabenorientiertes Programmiersystem für Montagesysteme. Diss. TU München (2014).

BACKHAUS & REINHART 2015

Backhaus, J.; Reinhart, G.: Digital description of products, processes and resources for task-oriented programming of assembly systems. Journal of Intelligent Manufacturing (2015), S. 1-14.

BARTSCHER 2011

Bartscher, S.: Mensch-Roboter-Kooperation in der Produktion – Chancen und Risiken. Robot World 6 (2011), S. 8–9.

BAUERNHANSL 2014

Bauernhansl, T.: Die Vierte Industrielle Revolution – Der Weg in ein wertschaffendes Produktionsparadigma. In: Bauernhansl et al. (Hrsg.): Industrie 4.0 in Produktion, Automatisierung und Logistik. Anwendung, Technologien und Migration. Wiesbaden: Springer Vieweg 2014, S. 5–35. ISBN: 978-3-658-04681-1.

BENGEL 2004

Bengel, G.: Grundkurs verteilte Systeme. Grundlagen und Praxis des Client-Server-Computing. 3. Aufl. Wiesbaden: Vieweg 2004. ISBN: 978-3-834-82150-8.

BENGEL 2010

Bengel, M.: Workpiece-centered Approach to Reconfiguration in Manufacturing Engineering. Diss. Universität Stuttgart, (2010). Heimsheim: ISBN: 978-3-939890-60-7.

BJÖRKELUND ET AL. 2011

Björkelund, A.; Malec, J.; Nilsson, K.; Nugues, P.: Knowledge and skill representations for robotized production. In: International Federation of Automatic Control (Hrsg.): Proceedings of the 18th IFAC Congress. Mailand, 2011. Kidlington: Elsevier Limited 2011, S. 8999 – 9004. ISBN: 978-3-902661-93-7.

BJÖRKELUND ET AL. 2012

Björkelund, A.; Malec, J.; Nilsson, K.; Nugues, P.; Bruyninckx, H.: Knowledge for Intelligent Industrial Robots. AAAI Spring Symposium: Designing Intelligent Robots. Palo Alto 2012.

BRECHER ET AL. 2004

Brecher, C.; Schröter, B.; Almeida, de O., C.; Dai, F.; Matthias, B.; Kock, S.: Intuitiv bedienbare Programmiersysteme zur effizienten Programmierung von Handhabungsaufgaben. Robotik 2004. Leistungsstand, Anwendungen, Visionen, Trends. München, 17. und 18. Juni 2004, S. 303–310.

BÜSCHER ET AL. 2013

Büscher, C.; Kuz, S.; Ewert, D.; Schilberg, D.; Jeschke, S.: Kognitive Planungs- und Lernmechanismen in selbstoptimierenden Montagesystemen. In: Jeschke, S. et al. (Hrsg.): Automation, Communication and Cybernetics in Science and Engineering 2011/2012. Berlin 2012. Berlin, Heidelberg: Springer 2012, S. 583–593. ISBN: 978-3-642-33388-0.

CAVIN ET AL. 2013

Cavin, S.; Ferreira, P.; Lohse, N.: Dynamic skill allocation methodology for evolvable assembly systems. In: Institute of Electrical and Electronics Engineers (IEEE) (Hrsg.): 11th IEEE International Conference on Industrial Informatics (INDIN). Bochum, 29. – 31. Juli 2013. Piscataway: IEEE 2013, S. 218–223. ISBN: 978-1-479-90752-6.

CAVIN & LOHSE 2014

Cavin, S.; Lohse, N.: Multi-level skill-based allocation methodology for evolvable assembly systems. In: Institute of Electrical and Electronics Engineers (IEEE) (Hrsg.): 12th IEEE International Conference on Industrial Informatics (INDIN). Porto Alegre RS, 27. – 30. Juli 2014. Piscataway: IEEE 2013, S. 532–537. ISBN: 978-1-4799-4905-2.

CEDERBERG ET AL. 2005

Cederberg, P.; Olsson, M.; Bolmsjö, G.: A semiautomatic task-oriented programming system for sensor-controlled robotised small-batch and one-off manufacturing. *Robotica* 23 (2005) 06, S. 743-754.

CUIPER 2000

Cuiper, R.: Durchgängige rechnergestützte Planung und Steuerung von automatisierten Montagevorgängen. Diss. TU München (2000). München: Utz 2000. ISBN: 978-3-89675-783-8.

FREUND & HECK 1990

Freund, E.; Heck, H.: Objektorientierte implizite Roboterprogrammierung mit graphischer Arbeitszellensimulation. In: Brauer, W. et al. (Hrsg.): Graphik und KI, Bd. 239. Berlin, 1990. Berlin, Heidelberg: Springer 1990, S. 113–125. ISBN: 3-540-52503-3.

HAUN 2005

Haun, M.: Handbuch Robotik. Programmieren und Einsatz intelligenter Roboter. Berlin: Springer 2005. ISBN: 3-540-25508-7.

HERFS ET AL. 2013

Herfs, W.; Malik, A.; Lohse, W.; Fayzullin, K.: Model-based assembly control concept. In: Institute of Electrical and Electronics Engineers (IEEE) (Hrsg.): 18th Conference on Emerging Technologies & Factory Automation (ETFA). Cagliari, 10. – 13. September 2013. Piscataway: IEEE 2013, S. 1–8. ISBN: 978-1-4799-0864-6.

HUBER 2013

Huber, A.: Das Ziel Digital Enterprise: die professionelle digitale Abbildung von Produktentwicklung und Produktion. In: Sendler (Hrsg.): Industrie 4.0. Beherrschung der industriellen Komplexität mit SysLM. Berlin: Springer-Vieweg 2013, S. 111–124. ISBN: 978-3-642-36916-2.

HUCKABY 2014

Huckaby, J.: Modeling Robot Assembly Tasks in Manufacturing Using SysML. In: International Symposium on Robotics (Hrsg.): Proceedings for the joint conference of ISR 2014. 8th German Conference on Robotics. München, 2. – 3. Juni 2014. Berlin, Offenbach: VDE-Verlag, S. 743–749. ISBN: 978-3-8007-3601-0.

HUCKABY & CHRISTENSEN 2012

Huckaby, J.; Christensen, H.: A Taxonomic Framework for Task Modeling and Knowledge Transfer in Manufacturing Robotics. In: American Association for Artificial Intelligence (Hrsg.): Cognitive robotics. Papers from the 2012 AAAI Workshop. Toronto, 22. – 23. Juli 2012. Palo Alto: AAAI Press, S. 94 – 101. ISBN: 978-1-57735-571-7.

HUCKABY ET AL. 2013

Huckaby, J.; Vassos, S.; Christensen, H.: Planning with a task modeling framework in manufacturing robotics. In: Institute of Electrical and Electronics Engineers (IEEE) (Hrsg.): IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2013). Tokyo, 3. – 7. November 2013. Piscataway: IEEE 2013, S. S. 5787–5794. ISBN: 978-1-46736-356-3.

HUMBURGER 1998

Humburger, R.: Konzeption eines Systems zur aufgabenorientierten Roboterprogrammierung. Diss. RWTH Aachen (1997). Aachen: Shaker 1998. ISBN: 3-8265-3427-1.

KAUFMAN ET AL. 1996

Kaufman, S.; Wilson, R.; Jones, R.; Calton, T.; Ames, A.: The Archimedes 2 mechanical assembly planning system. In: Institute of Electrical and Electronics Engineers (IEEE) (Hrsg.): IEEE International Conference on Robotics and Automation. Minneapolis, 22. - 28. April 1996. Piscattaway: IEEE 1996, S. 3361–3368. ISBN: 0-7803-2988-0.

KIM ET AL. 2015

Kim, Y.; Jung, J.; Matson, E.: An Adaptive Task-Based Model for Autonomous Multi-Robot Using HARMS and NuSMV. Procedia Computer Science 56 (2015), S. 127–132.

KRAUS 2015

Kraus, C.: Automatisierte Montageplanung für variantenreiche Produkte in modularen Produktionssystemen. Masterarbeit. Iwb. TU München (2015).

KRUG 2013

Krug, S.: Automatische Konfiguration von Robotersystemen (Plug & Produce). Diss. TU München (2013). München: Utz 2013. ISBN 978-3-8316-4243-4.

KUGELMANN 1999

Kugelmann, D.: Aufgabenorientierte Offline-Programmierung von Industrierobotern. Diss. TU München (1999). München: Utz 1999. ISBN: 978-3-89675- 615-2.

KÜHNEL 2010

Kühnel, A.: Visual C# 2010. Das umfassende Handbuch. 5. akt. und vollst. überarbeitete Aufl. Bonn: Galileo Press, 2010. ISBN: 978-3-83621-552-7.

LESZAK & EGGERT 1989

Leszak, M.; Eggert, H.: Petri-Netz-Methoden und -Werkzeuge. Hilfsmittel zur Entwurfsspezifikation und -validation von Rechensystemen. Berlin, Heidelberg: Springer 1989. ISBN: 978-3-54050-642-3.

LOTTER 2012

Lotter, B.: Montage in der industriellen Produktion. Ein Handbuch für die Praxis. Berlin, Heidelberg: Springer 2012. ISBN: 978-3-642-29060-2.

LÜDEMANN-RAVIT 2005

Lüdemann-Ravit, B.: Ein System zur Automatisierung der Planung und Programmierung von industriellen Roboterapplikationen. Diss. Universität Dortmund. Düsseldorf: VDI-Verlag 2005. ISBN: 3-18-340020-0.

MAAß ET AL. 2008

Maaß, J.; Molkenstruck, S.; Thomas, U.; Hesselbach, J.; Wahl, F.: Definition and execution of a generic assembly programming paradigm. Assembly Automation 28 (2008) 1, S. 61–68.

MACKENZIE & ARKIN 1998

MacKenzie, D.; Arkin, R. C.: Evaluating the Usability of Robot Programming Toolsets. The International Journal of Robotics Research 17 (1998) 4, S. 381-401.

MATTHIAS ET AL. 2004

Matthias, B.; Dai, F.; Knock, S.; Lau, A.; Merte, R.; Waldi, W.; Wittmann, J.; Behnisch, K.; Pühringer, A.; Hoffmann, S.; Prinz, S.; Fernholz, O.; Klebanowski, A.; Ahlberndt, N.; Neumann, A.; Almeida, C.; Brecher, C.; Kahmen, A.; Schröter, B.; Weck, M.; Krüger, P.; Görnemann, O.; Pfeil, H.; Herchel, M.; et al.: Ein flexibel einsetzbares Robotersystem für variierende Aufgaben in der Maschinenbeschickung. In: VDI (Hrsg.): VDI Berichte Bd. 1841. Düsseldorf: VDI-Verlag 2004, S. 567–574. ISBN 3-18-091841-1.

MEDELLIN ET AL. 2010

Medellin, H.; Corney, J.; Ritchie, J.; Lim, T.: Automatic generation of robot and manual assembly plans using octrees. *Assembly Automation* 30 (2010) 2, S. 173–183.

MICHNIEWICZ 2015

Während der Bearbeitung der Arbeit erfolgten zahlreiche Expertengespräche sowie inhaltliche Abstimmungen durch den Betreuer des *iwb*, J. Michniewicz.

MICHNIEWICZ & REINHART 2015A

Michniewicz, J.; Reinhart, G.: Cyber-Physical-Robotics – Modelling of modular robot cells for automated planning and execution of assembly tasks. *Mechatronics* (2015).

MICHNIEWICZ & REINHART 2015B

Michniewicz, J.; Reinhart, G.: Cyber-Physische Systeme in der Robotik - Automatische Planung und Durchführung von Montageprozessen durch Kommunikation zwischen Produkt und Produktionssystem. TU München (2015).

MIKUSZ & CSISZAR 2015

Mikusz, M.; Csiszar, A.: CPS Platform Approach to Industrial Robots: State of the Practice, Potentials, Future Research Directions. Pacific Asia Conference on Information Systems (PACIS). Singapur, 5. – 9. Juli 2015.

MITSI ET AL. 2005

Mitsi, S.; Bouzakis, K.-D.; Mansour, G.; Sagris, D.; Maliaris, G.: Off-line programming of an industrial robot for manufacturing. *The International Journal of Advanced Manufacturing Technology* 26 (2005) 3, S. 262 – 267.

MOSEMANN & WAHL 2001

Mosemann, H.; Wahl, F.: Automatic decomposition of planned assembly sequences into skill primitives. In: *IEEE Transactions on Robotics and Automation* 17 (2001) 5, S. 709 – 718.

RINGERT ET AL. 2014

Ringert, J. O.; Rumpe, B.; Wortmann, A.: A Requirements Modeling Language for the Component Behavior of Cyber Physical Robotics Systems. In: Seyff, N. & Koziolek, A. (Hrsg.): *Modelling and Quality in Requirements Engineering: Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday*. Münster: Monsenstein und Vannerdat, 2012, S. 143 – 155. ISBN: 978-3-86991-724-5

RUßWURM 2013

Rußwurm, S.: Software: Die Zukunft der Industrie. In: Sendler (Hrsg.): Industrie 4.0. Beherrschung der industriellen Komplexität mit SysLM. Berlin: Springer-Vieweg 2013, S. 21–36. ISBN: 978-3-642-36916-2.

SCHMITT 2014

Schmitt, C.: Entwicklung und Aufbau eines Cyber-Physischen Roboterzellen-Demonstrators zur aufgabenorientierten Montage von Lego. Semesterarbeit. iwb. TU München (2014).

SCHREINER 2009

Schreiner, R.: Computernetzwerke. Von den Grundlagen zur Funktion und Anwendung. 4., überarbeitete und erweiterte Auflage. München: Hanser 2009. ISBN: 978-3-446-41922-3.

SCHUNK GMBH & Co. KG 2009

Schunk GmbH & Co. KG: Kraft-Momenten-Sensor Typ FTC / FTCL. Montage- und Betriebsanleitung. Lauffen am Neckar: 2009. http://www.schunk.com/schunk_files/attachments/OM_AU_FTC__DE.pdf - 18.08.15.

SELLS 2002

Sells, C.: Sicheres und einfaches Multithreading in Windows Forms. <https://msdn.microsoft.com/de-de/library/ms951089.aspx> - 20.08.2015.

SENDLER 2013

Sendler, U.: Industrie 4.0 – Beherrschung der industriellen Komplexität mit SysLM (Systems Lifecycle Management). In: Sendler (Hrsg.): Industrie 4.0. Beherrschung der industriellen Komplexität mit SysLM. Berlin: Springer-Vieweg 2013, S. 1–19. ISBN: 978-3-642-36916-2.

SOFTING INDUSTRIAL AUTOMATION GMBH 2012A

Softing Industrial Automation GmbH: CAN-ACx-PCI. Hardware User Manual. München: 2012.

SOFTING INDUSTRIAL AUTOMATION GMBH 2012B

Softing Industrial Automation GmbH: Softing CAN Layer2 Manual. Software Description. Version 5.17. München: 2012.

STEEGMÜLLER & ZÜRN 2014

Stegmüller, D.; Zürn, M.: Wandlungsfähige Produktionssysteme für den Automobilbau der Zukunft. In: Bauernhansl et al. (Hrsg.): Industrie 4.0 in Produktion, Automatisierung und Logistik. Anwendung, Technologien und Migration. Wiesbaden: Springer Vieweg 2014, S. 103–119. ISBN: 978-3-658-04681-1.

STENMARK & MALEC 2015

Stenmark, M.; Malec, J.: Knowledge-based instruction of manipulation tasks for industrial robotics. *Robotics and Computer-Integrated Manufacturing* 33 (2015), S. 56–67.

STENMARK ET AL. 2015

Stenmark, M.; Malec, J.; Stolt, A.: From High-Level Task Descriptions to Executable Robot Code. In: Filev, D. et al. (Hrsg.): Proceedings of the 7th IEEE International Conference Intelligent Systems IS'2014. Warschau, 24. - 26. September 2014. Cham: Springer 2015, S. 189 – 202. ISBN: 978-3-319-11310-4.

TANENBAUM & WETHERALL 2012

Tanenbaum, A.; Wetherall, D.: Computernetzwerke. 5., aktualisierte Aufl. München: Pearson 2012. ISBN: 978-3-86894-137-1.

THOMAS 2009

Thomas, U.: Automatisierte Programmierung von Robotern für Montageaufgaben. In: Wagner, D. (Hrsg.): Ausgezeichnete Informatikdissertationen 2008. Bonn: Bonner Köllen Verlag 2009, S. 291–300. ISBN 978-3-88579-413-4.

UYGUROĞLU 2011

Uyguroğlu, M.: Robot Kinematics: Position Analysis. Eastern Mediterranean University. Gazimağusa.: <<http://opencourses.emu.edu.tr/mod/resource/view.php?id=596>> - 24.08.15.

WECK 2001

Weck, M.: Werkzeugmaschinen - Fertigungssysteme. 5., neu bearb. Aufl. Berlin [u.a.]: Springer 2001. ISBN: 3-5406-7613-9.

WEEKS 1997

Weeks, J.: Entwicklung eines aufgabenorientierten Greif- und Bahnplanungssystems für die automatisierte Montage mit SCARA-Robotern. Diss. RWTH Aachen. Aachen: Shaker 1997. ISBN: 3-826-52114-5.

WILLMS 2011

Willms, A.: Visual C++ 2010. Das umfassende Handbuch. 1. Aufl. Bonn: Galileo Press 2011. ISBN: 978-3-827-32901-1.

YASKAWA ELECTRIC CORPORATION 2011

Yaskawa Electric Corporation (Hrsg.): FS 100 Options - Instructions for concurrent I/O. Manual No. RE-CKI-A460. Kitakyushu: 2011.

YASKAWA ELECTRIC CORPORATION 2012A

Yaskawa Electric Corporation (Hrsg.): FS 100 Options – Instructions. Reference Manual for New Language Environment MotoPlus. API Function Specifications. Kitakyushu 2012.

YASKAWA ELECTRIC CORPORATION 2012B

Yaskawa Electric Corporation (Hrsg.): FS 100 Options – Instructions. Programmer's Manual for New Language Environment MotoPlus. API Function Specifications. Kitakyushu 2012.

YASKAWA ELECTRIC CORPORATION 2013

Yaskawa Electric Corporation (Hrsg.): FS 100 Options - Instructions. User's Manual for New Language Environment MotoPlus. Kitakyushu 2013.

ZÄH ET AL. 2004

Zäh, M. F.; Vogl, W.; Munzert, U.: Beschleunigte Programmierung von Industrierobotern. Augmented Reality-Einsatz zur intuitiven Mensch-Maschine-Interaktion. *wt Werkstattstechnik online* 94 (2004) 9, S. 438–441.

ZIMMERMANN & SCHMIDGALL 2014

Zimmermann, W.; Schmidgall, R.: Bussysteme in der Fahrzeugtechnik. 5., aktualisierte und erweiterte. Aufl. Wiesbaden: Springer Fachmedien 2014. ISBN 978-3-658-02418-5.

17 Anhänge

17.1 Funktionaler Aufbau der SPS

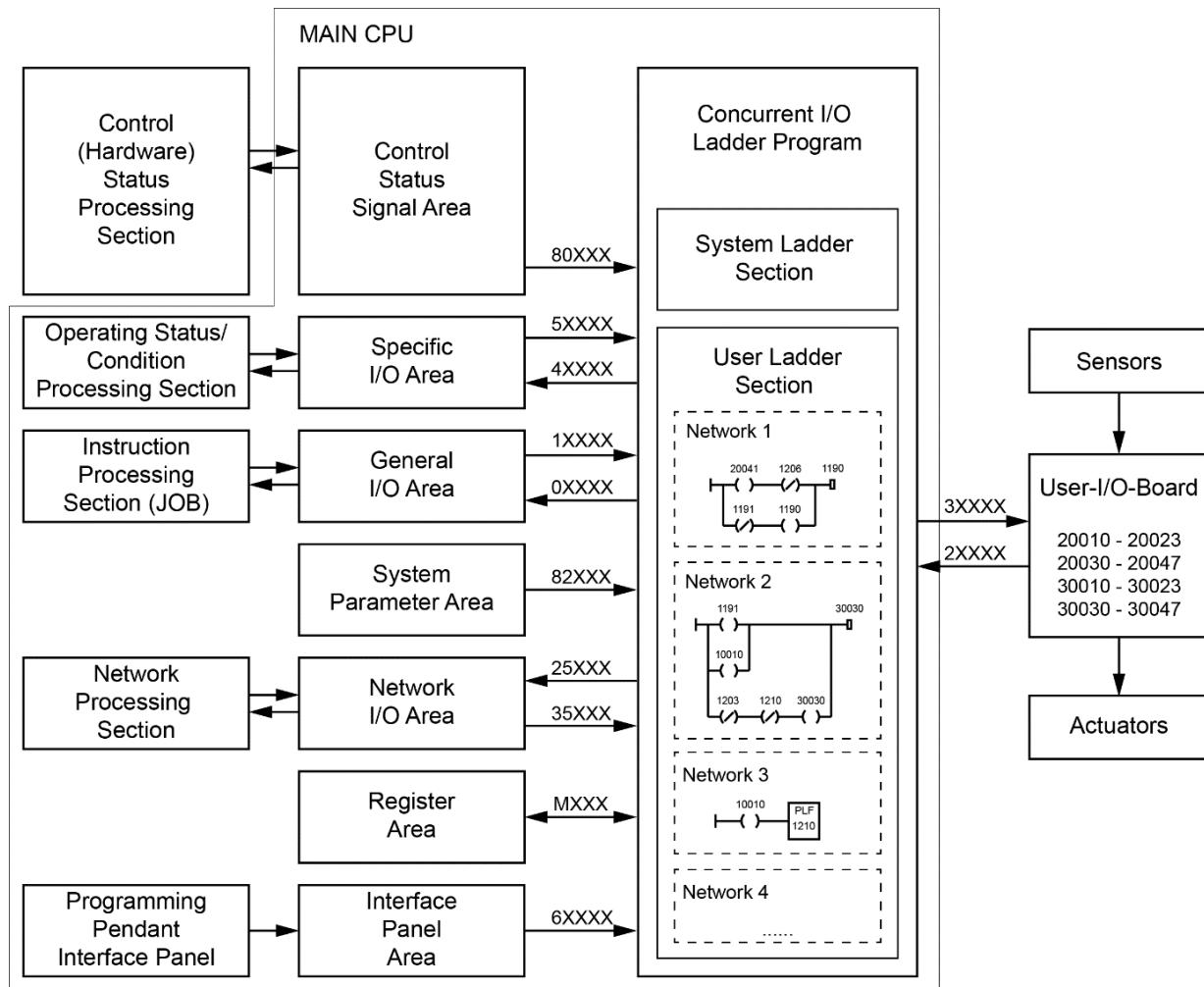


Abbildung 17-1: Funktionaler Aufbau des Registers der SPS
(In Anlehnung an YASKAWA ELECTRIC CORPORATION 2011).

17.2 Integration eines zweiten Projekts in Visual Studio

Um zu gewährleisten, dass die Software für den Leitrechner im Rahmen weiterer studentischer Arbeiten genutzt werden kann, wird im Folgenden der Aufbau des Projekts sowie die notwendigen Einstellungen zum Arbeiten mit Visual Studio bei Integration eines zweiten Projekts dargelegt.

Grundlage für das Projekt ist die Projektmappe WinNetworkIOCP. Diese befindet sich im Ordner Client 2.1, der auch auf der beigefügten Daten-CD auffindbar ist. Zu diesem Projekt hinzugefügt ist das Projekt CAN-Bus-Kommunikation-Lib

1.0, das im Ordner CAN-Bus-Kommunikation-Lib 1.3 zu finden ist. Wird dieses Projekt beim Start des Projekts WinNetworkIOPC nicht geladen, so muss selbiges entfernt, und über *Datei->Hinzufügen->Vorhandenes Projekt* eingefügt werden. Abbildung 17-2 zeigt die zugehörige Ansicht des Projektmappen-Explorers.

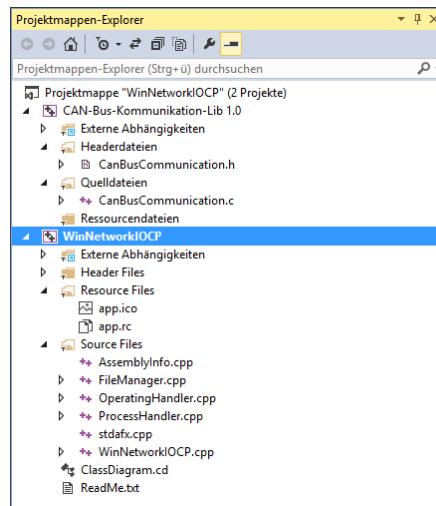


Abbildung 17-2: Projektmappen-Explorer des Projekts WinNetworkIOPC
(Quelle: Eigene Ausarbeitung).

Um die Methoden der in Programmiersprache C gehaltenen CAN-Bus-Library verwenden zu können, müssen im nächsten Schritt die nötigen Verweise gesetzt werden. Durch einen Rechtsklick auf das Projekt WinNetworkIOPC und Anwählen des Buttons *Eigenschaften* öffnet sich, das in Abbildung 17-3 auf der linken Seite befindliche Dialogfenster. Im diesem muss unter *Allgemeine Eigenschaften* ein neuer Verweis hinzugefügt werden. Im Anschluss öffnet sich das in Abbildung 17-3 dargestellte rechte Fenster. Hier muss das Projekt CAN-Bus-Kommunikation-Lib 1.0 ausgewählt und dies mit *OK* bestätigt werden.

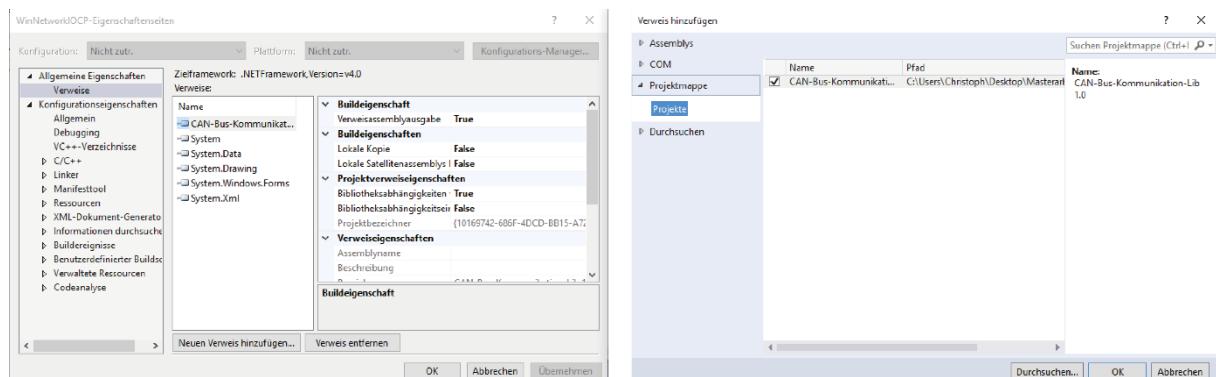
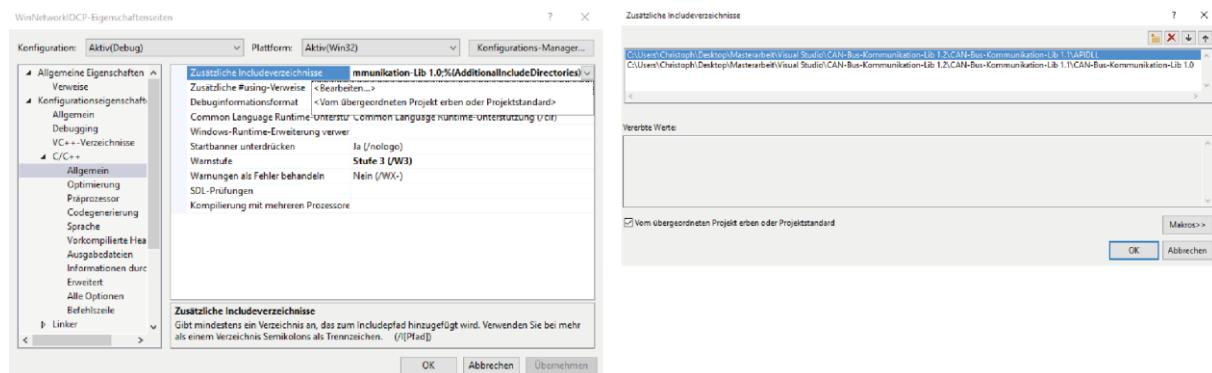


Abbildung 17-3: Verweis auf ein zweites Projekt in Visual Studio
(Quelle: Eigene Ausarbeitung).

Nach Hinzufügen des neuen Verweises müssen die Includeverzeichnisse der Programmiersprachen C/C++ ergänzt werden. Diese werden im Eigenschaftsfenster der Projektmappe WinNetworkIOPC in den Konfigurationseigenschaften angegeben. Abbildung 17-4 zeigt dies. Durch einen Klick auf das Drop-Down-Menü des Feldes *Zusätzliche Includeverzeichnisse* und Auswahl des Menüpunkts *Bearbeiten* wird das in Abbildung 17-4 gezeigte rechte Fenster geöffnet. Hier müssen die Dateipfade zum Ordner APIDLL und zum Projektordner des Projekts CAN-Bus-Kommunikation-Lib 1.0 spezifiziert werden.

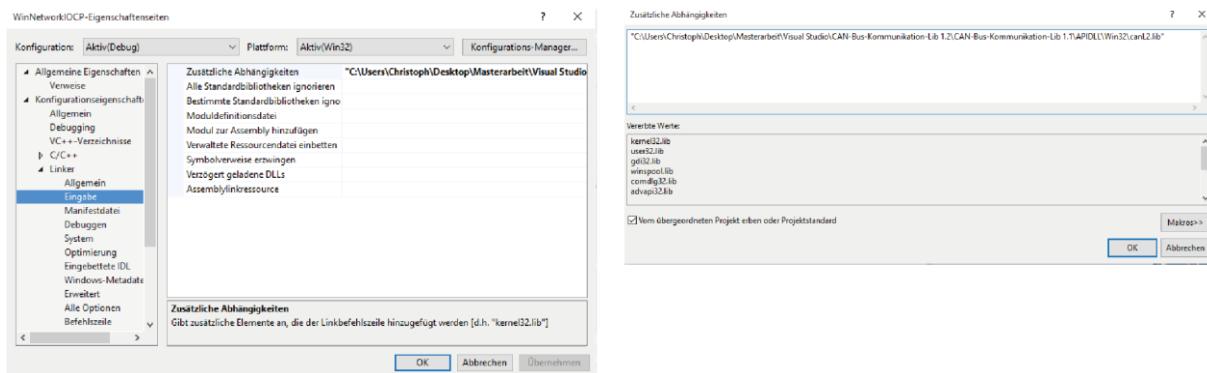


*Abbildung 17-4: Einstellung der Includeverzeichnisse
(Quelle: Eigene Ausarbeitung).*

Die Includeverzeichnisse der Projektmappe CAN-Bus-Kommunikation-Lib 1.0 müssen ebenfalls aktualisiert werden. Durch einen Rechtsklick auf dieses Projekt und durch Wahl des Menü-Punkts *Eigenschaften* öffnen sich die Eigenschaftenseiten. Die zusätzlichen Includeverzeichnisse des Projekts sind unter den Konfigurationseigenschaften anzugeben. Im vorliegenden Fall sind diese der Dateipfad zum Ordner APIDLL der Projektmappe.

Abschließend muss dem Linker des Projekts noch die zusätzliche Abhängigkeit zur canL2.lib der API zur Nutzung der CAN-Bus-Karte mitgeteilt werden. Abbildung 17-5 zeigt den dazu auszuwählenden Menüpunkt in den Projekteigenschaften. Der anzugebene Pfad muss von Anführungszeichen umschlossen sein.

Sind die beschriebenen Abhängigkeiten erfolgreich aktualisiert, so kann das Projekt kompiliert werden. Im Debug-Ordner des Projekts Client 2.0 sollten die Dateien WinNetworkIOPC.exe, canL2.dll und CAN-Bus-Kommunikation-Lib 1.0.lib zu finden sein. Diese drei Dateien sind stets im gleichen Ordner abzuspeichern. Erstere stellt die ausführbare Datei zur Steuerung der Roboterzelle dar. In der Datei canL2.dll sind die von der Softing Industrial Automation GmbH zur Verfügung gestellten Funktionen enthalten. Die lib-Datei enthält die in der Programmiersprache C entwickelten Funktionen der Projektmappe für die CAN-Bus-Kommunikation.



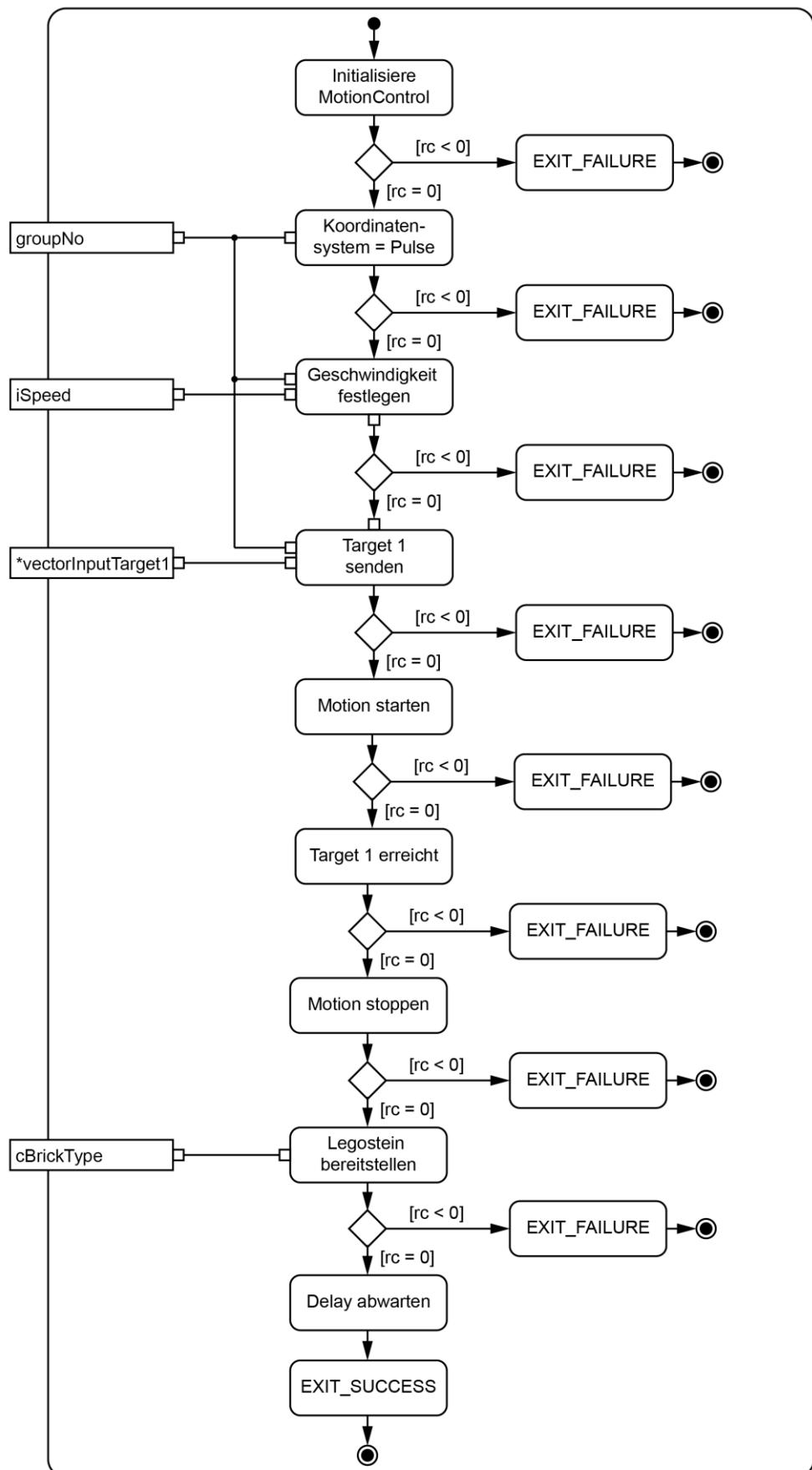
*Abbildung 17-5: Einstellung der zusätzlichen Abhängigkeiten des Linkers
(Quelle: Eigene Ausarbeitung).*

17.3 UML-Diagramme

Die Dokumentation der im Rahmen dieser Arbeit implementierten Software basiert vorrangig auf UML-Diagrammen nach dem UML2-Standard. Im Folgenden finden sich die Aktivitätsdiagramme zu den FctDoTarget-Funktionen. Diese Funktionen werden grundsätzlich in Kapitel 11.4.4 beschrieben. Darüber hinaus findet sich am Ende dieses Abschnitts das UML-Diagramm zur Funktion FctGripperSelection. Aufgabe dieser Funktion ist die Auswahl des richtigen Greifers.

17.3.1 Aktivitätsdiagramme zur Abarbeitung der Targets 1 bis 7

Übergabeparameter für die Funktion FctDoTarget ist immer die Gruppennummer GroupNo. Darüber hinaus kann die Geschwindigkeit, in der das Target angefahren werden soll, spezifiziert werden. Diese ist in der Einheit mm/s anzugeben. Der dritte Übergabeparameter dieser Funktionen ist der Vektor zum jeweiligen Punkt i. Dieser Vektor ist im Koordinatensystem Pulse zu übergeben.



*Abbildung 17-6: Aktivitätsdiagramm zur Funktion FctDoTarget1
(Quelle: Eigene Ausarbeitung).*

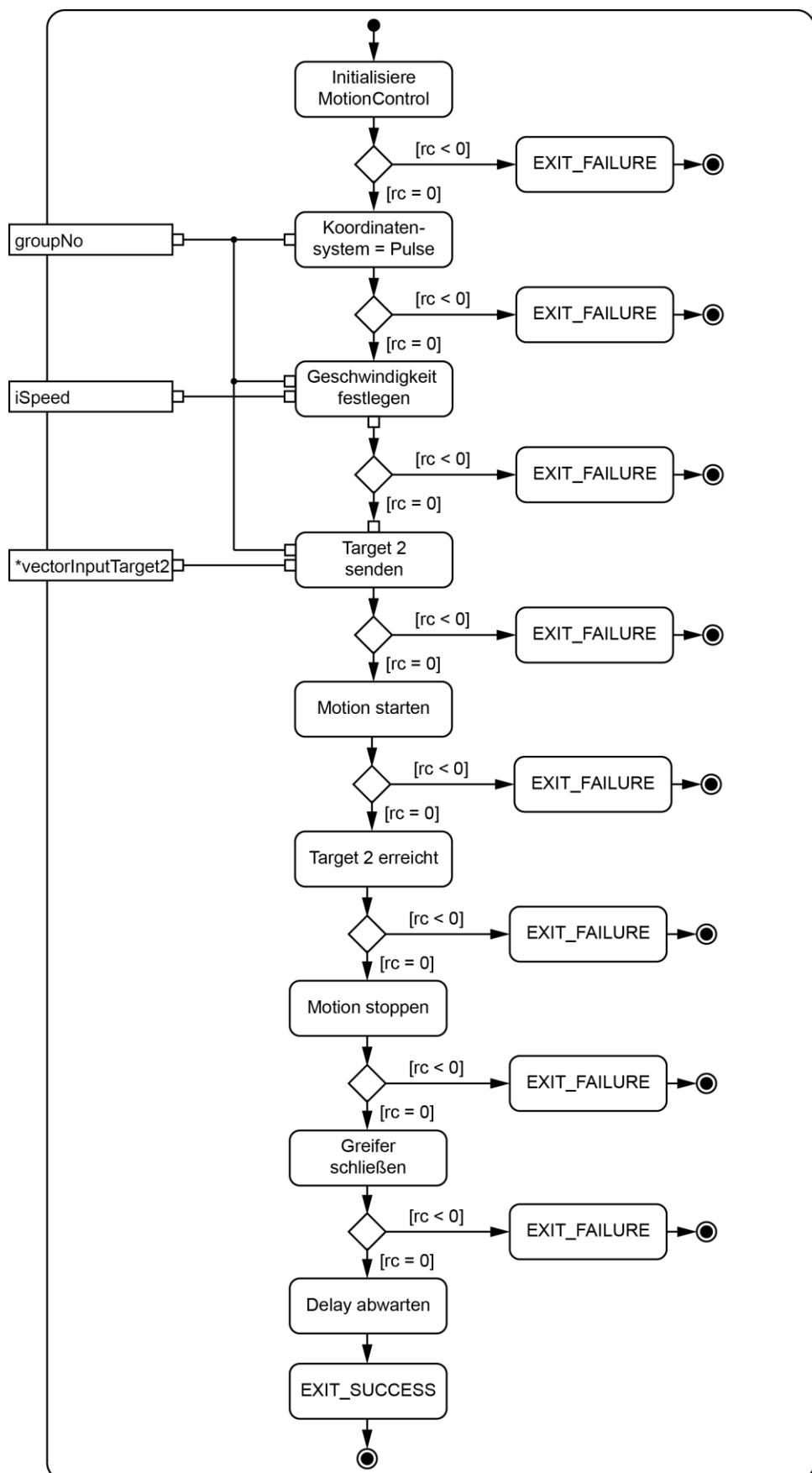


Abbildung 17-7: Aktivitätsdiagramm zur Funktion FctDoTarget2
 (Quelle: Eigene Ausarbeitung).

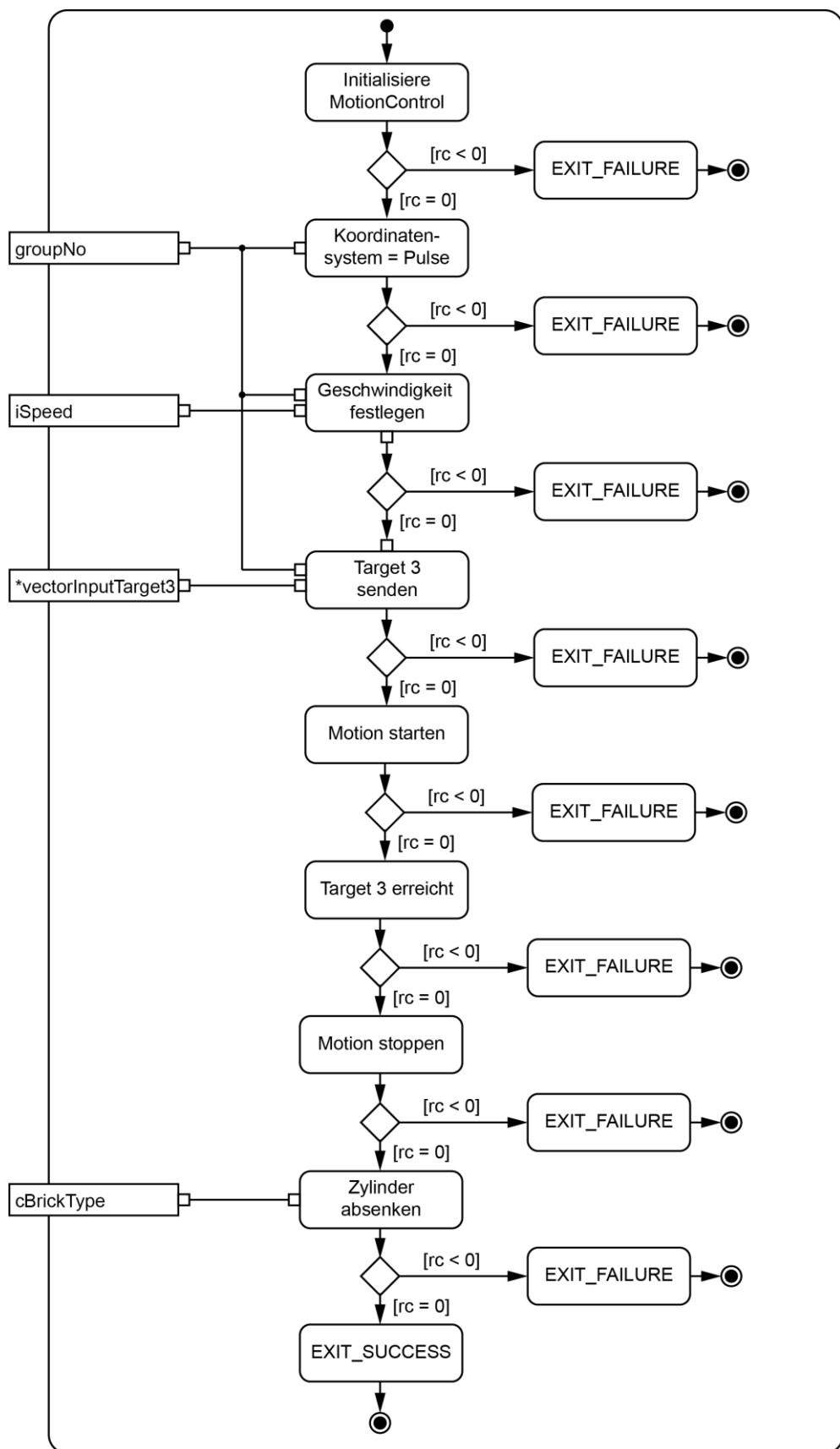


Abbildung 17-8: Aktivitätsdiagramm zur Funktion FctDoTarget3
 (Quelle: Eigene Ausarbeitung).

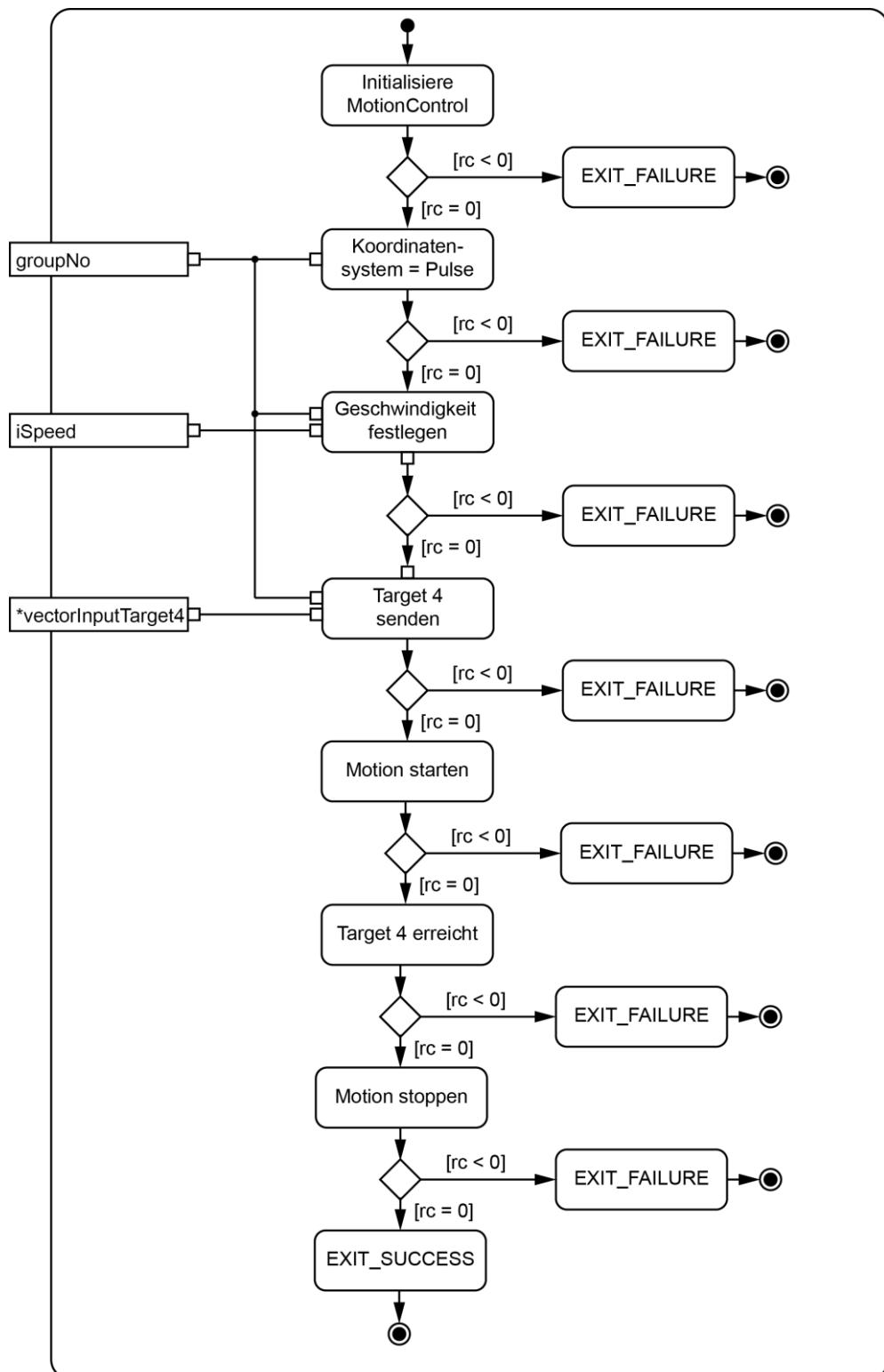


Abbildung 17-9: Aktivitätsdiagramm zur Funktion FctDoTarget4
 (Quelle: Eigene Ausarbeitung).

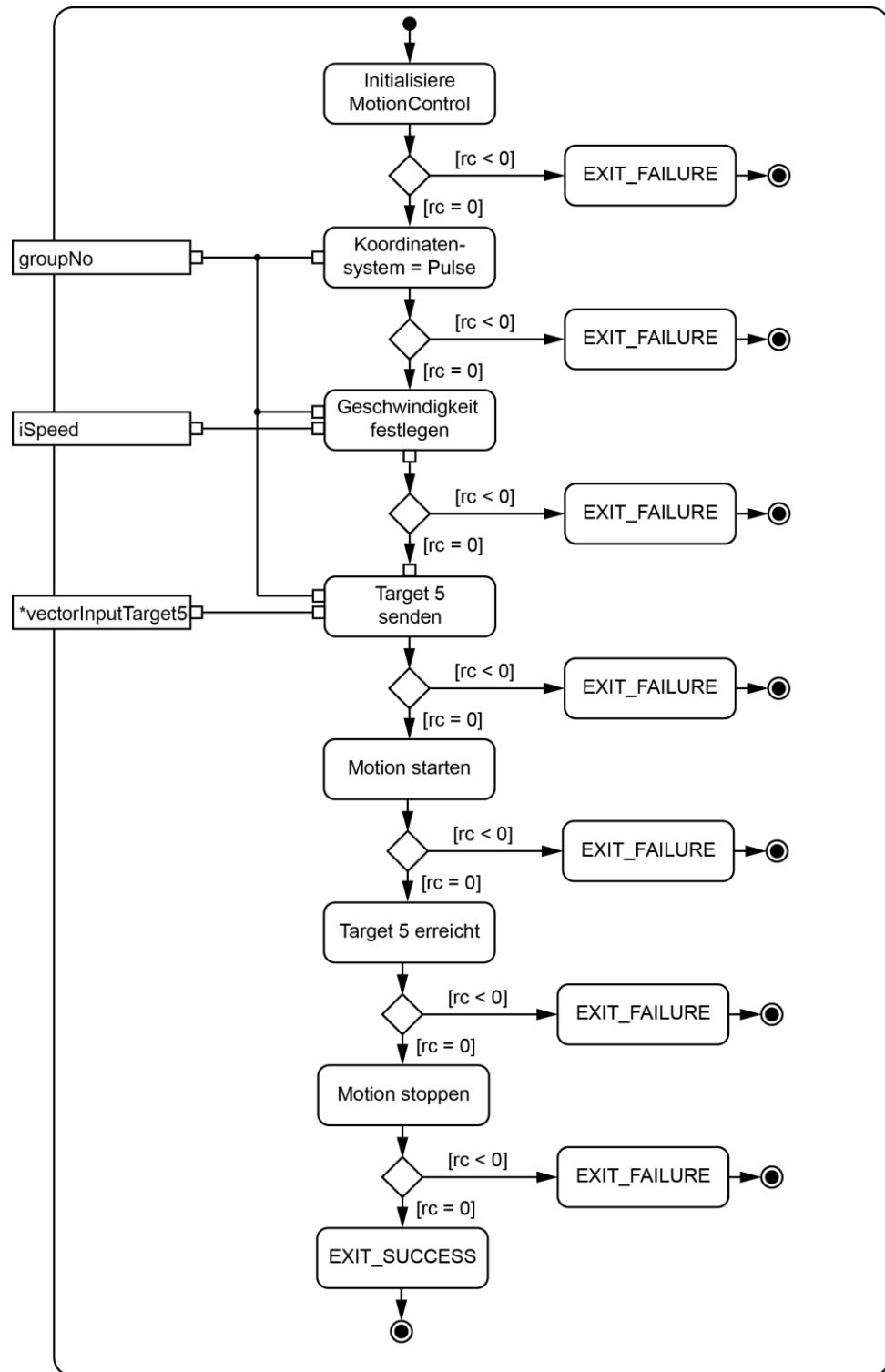


Abbildung 17-10: Aktivitätsdiagramm zur Funktion *FctDoTarget5*
 (Quelle: Eigene Ausarbeitung).

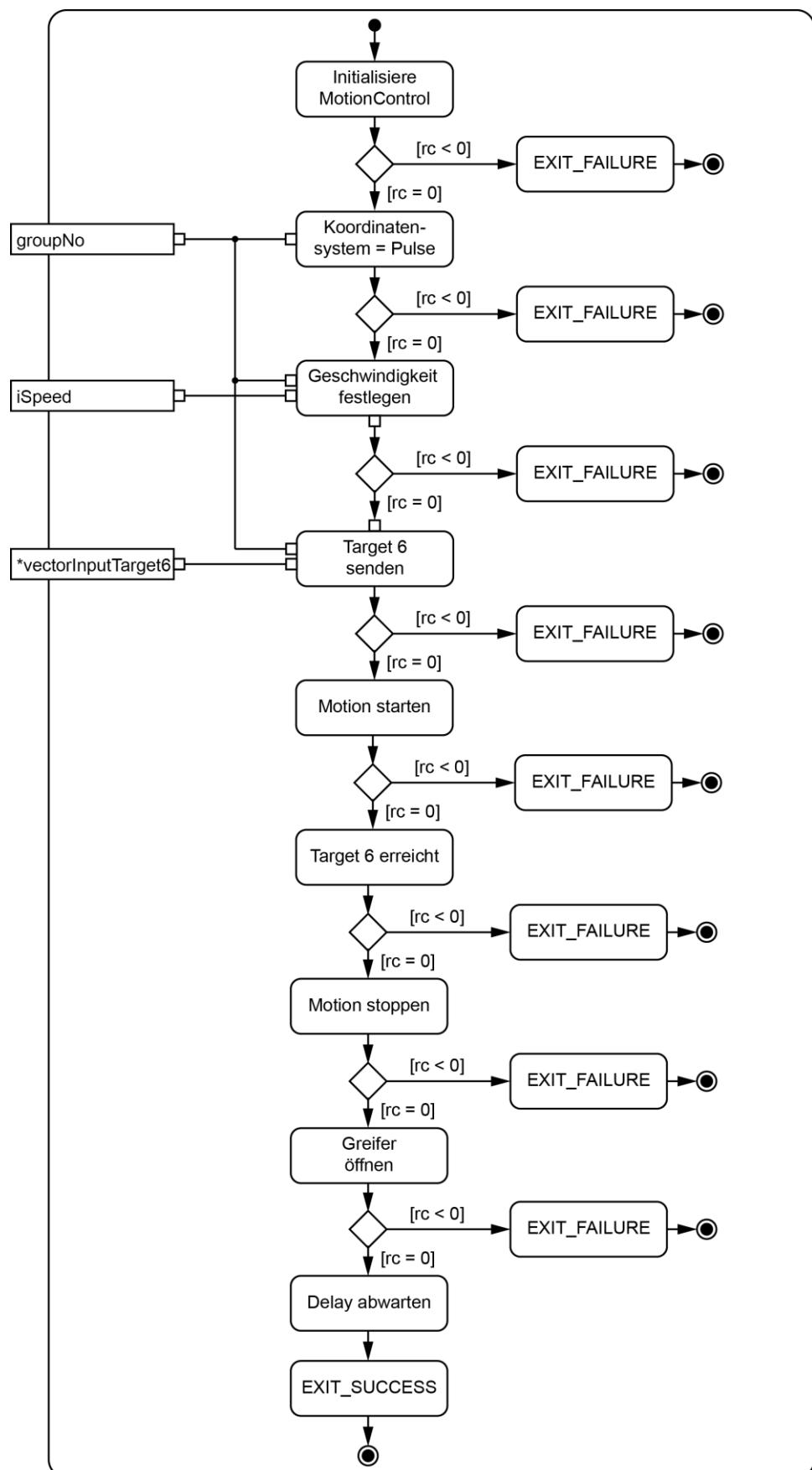


Abbildung 17-11: Aktivitätsdiagramm zur Funktion FctDoTarget6
 (Quelle: Eigene Ausarbeitung).

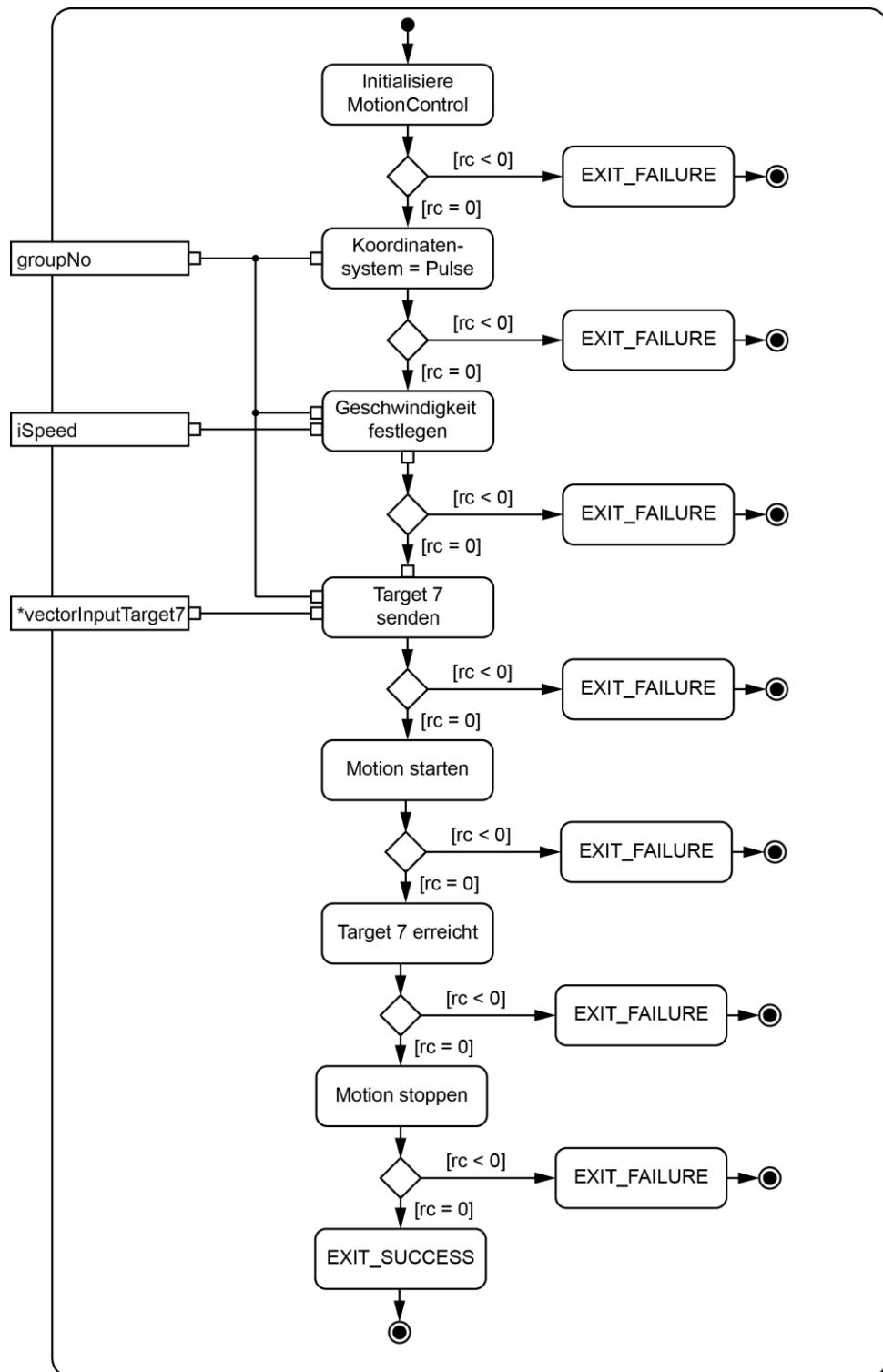


Abbildung 17-12: Aktivitätsdiagramm zur Funktion *FctDoTarget7*
 (Quelle: Eigene Ausarbeitung).

17.3.2 Aktivitätsdiagramm zur Funktion FctGripperSelection

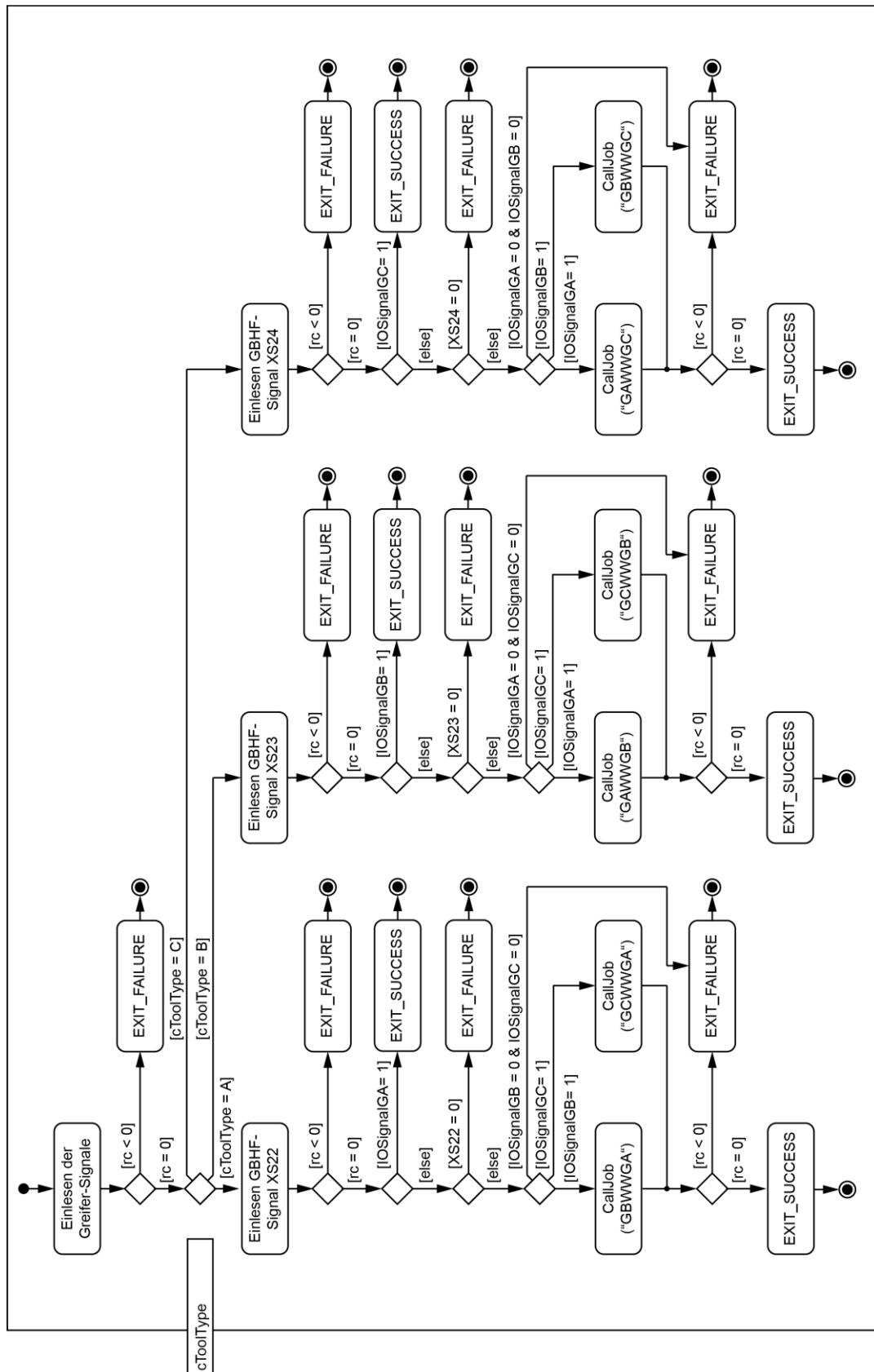


Abbildung 17-13: Aktivitätsdiagramm zur Funktion FctGripperSelection
(Quelle: Eigene Ausarbeitung).

17.4 Anleitungen zu Bedienung des Demonstrators

Zum Start des Demonstrators müssen einleitend die Rahmenbedingungen überprüft werden. Im Anschluss daran kann der Demonstrator gestartet werden. Beschreibungen hierzu finden sich in den folgenden beiden Unterkapiteln.

17.4.1 Rahmenbedingungen:

Bevor der Demonstrator gestartet wird, sollte sichergestellt sein, dass sich der Schlüsselschalter am PHG in Stellung *Remote* befindet. Daneben sollte der Not-Aus am PHG betätigt werden. Im Anschluss kann mit Hilfe des Hauptschalters am Schaltschrank die Spannungsversorgung hergestellt werden. Es ist zu überprüfen, ob der Betriebsdruck der Anlage 6 bar beträgt. Hierzu dient ein an der Pneumatik-Montageplatte angebrachtes Manometer. Bei Drücken unterhalb von 6 bar stimmen die im SPS-Programm zur Steuerung des Förderbands programmierten Zeiten nicht mit den realen Zeiten zum Ein- und Ausfahren der GTS-Umsetzer überein.

Als nächstes kann der Haupt-Not-Aus am Gestell gelöst werden, sofern dieser betätigt ist. Zum Quittieren des Not-Aus-Zustands muss im Anschluss der Freigabe-Taster am Schaltschrank betätigt werden. Zuvor ist sicherzustellen, dass sich keine Person im Arbeitsraum des Roboters befindet. Darüber hinaus muss der Not-Aus am PHG betätigt sein.

Ist dieser betätigt, so kann im Folgenden überprüft werden, ob die Zuführeinheit genügend Lego-Steine enthält. Ist dem nicht so, so sind die einzelnen Lego-Steine nachzufüllen. Darüber hinaus ist zu überprüfen, ob die Greifer im Greifbahnhof in der richtigen Lage eingelegt sind. Hierzu dient Abbildung 9-4. Des Weiteren ist zu kontrollieren, ob die Werkstückträger korrekt auf das Förderband aufgelegt sind. Die Ecke auf der Unterseite der Werkstückträger mit der Kennzeichnung BK muss stets die Ecke sein, in der die Spannvorrichtung die Kraft einleitet. Darüber hinaus ist sicherzustellen, ob die Werkstückträger leer sind. Andernfalls kann es beim Fügen einer Lego-Baugruppe zur Kollision kommen. Sind all diese Randbedingungen erfüllt, so kann der Not-Aus am PHG gelöst werden. Eine Auflistung der auszuführenden Tätigkeiten findet sich in Tabelle 17-1

*Tabelle 17-1: Auszuführende Tätigkeiten vor Start des Demonstrators
(Quelle: Eigene Ausarbeitung).*

Schritt-Nr.	Tätigkeit
1	Schlüsselschalter auf Remote stellen
2	Not-Aus am PHG betätigen.
3	Hauptschalter AN
4	Haupt-Not-Aus am Gestell lösen
5	Taster "Freigabe" am Schaltschrank drücken
6	Steinanzahl in der Zuführeinheit prüfen
7	Lage der Greifer im Greiferbahnhof prüfen
8	Orientierung der Werkstückträger prüfen
9	Prüfen ob alle Werkstückträger leer sind
10	Not-Aus am PHG lösen

17.4.2 Start des Demonstrators

Sind die beschriebenen Rahmenbedingungen erfüllt, so kann der Demonstrator in Betrieb genommen werden. Hierzu muss die ausführbare Datei MPP3-Controller.exe gestartet werden. Im Anschluss kann durch einen Klick auf den Button *Connect* die TCP-IP-Verbindung hergestellt werden. Hierzu sei angemerkt, dass die Robotersteuerung mindestens 30 s vor Verbindungsauftbau gestartet worden sein muss. Diese Zeit entspricht in etwa der Zeit die das auf der FS100 implementierte Betriebssystem zum Hochfahren benötigt.

Sofern der Betriebsmodus Not-Aus angezeigt wird ist zu überprüfen, ob die Not-Aus am Gestell gelöst ist. Sofern diese Bedingung erfüllt ist, muss der Taste Freigabe erneut betätigt werden. Sofern der Betriebsmodus Ruhezustand angezeigt wird, kann mit der Initialisierung der Zelle begonnen werden. Hierzu dient der Button *Initialisieren*.

Im Anschluss kann eine build.txt-Datei mittels *Datei hinzufügen* der Warteschlange des Controllers hinzugefügt werden. Sobald die Initialisierung beendet wurde, muss die Betriebsart gewählt werden. Durch einen Klick auf den Button Start beginnt die Montage der Lego-Baugruppe. Tabelle 17-2 fasst die beschriebenen Tätigkeiten zusammen.

*Tabelle 17-2: Tätigkeiten zum Starten des Demonstrators
(Quelle: Eigene Ausarbeitung).*

Schritt-Nr.	Tätigkeit
1	MPP3-Controller.exe starten
2	Button Connect betätigen
3	Button Initialisieren betätigen
4	Build-Datei mittels "Datei hinzufügen" dazufügen
5	Betriebsart wählen
6	Button Start betätigen

17.5 Übersichtstabelle zur Verwendung der P-Variablen

Tabelle 17-3: Verwendung der P-Variablen (Quelle: Eigene Ausarbeitung).

P-Variable	Zweck	Koordinaten-System
P000	Nullposition	Base
P001	unbenutzt	
P002	unbenutzt	
P003	unbenutzt	
P004	unbenutzt	
P005	unbenutzt	
P006	unbenutzt	
P007	unbenutzt	
P008	unbenutzt	
P009	unbenutzt	
P010	ÜberGE	Pulse
P11	GE	Pulse
P12	ÜberGD	Pulse
P13	GD	Pulse
P14	ÜberGC	Pulse
P15	GC	Pulse
P16	ÜberGB	Pulse
P17	GB	Pulse
P18	ÜberGA	Pulse
P19	GA	Pulse
P20	ÜberBrick1	Pulse
P21	Brick1	Pulse
P22	ÜberBrick2	Pulse
P23	Brick2	Pulse

P-Variable	Zweck	Koordinaten-System
P24	ÜberBrick3	Pulse
P25	Brick3	Pulse
P26	ÜberBrick4	Pulse
P27	Brick4	Pulse
P28	ÜberBrick5	Pulse
P29	Brick5	Pulse
P30	Aufnahmeposition GA + vertikales Offset	Base
P31	Aufnahmeposition GA	Base
P32	Hilfspunkt Aufnahmevergang GA	Base
P33	Hilfspunkt Aufnahmevergang GA	Base
P34	Endpunkt Aufnahmevergang GA	Base
P35	unbenutzt	
P36	unbenutzt	
P37	unbenutzt	
P38	unbenutzt	
P39	unbenutzt	
P56	Endpunkt Aufnahmevergang GC	Base
P57	unbenutzt	
P58	unbenutzt	
P59	unbenutzt	
P60	ÜberBrick6	Pulse
P61	Brick6	Pulse
P62	ÜberBrick7	Pulse
P63	Brick7	Pulse
P64	ÜberBrick8	Pulse
P65	Brick8	Pulse
P66	ÜberBrick9	Pulse
P67	Brick9	Pulse
P68	ÜberBrick10	Pulse
P69	Brick10	Pulse
P70	ÜberBrick11	Pulse
P71	Brick11	Pulse

17.6 Protokoll zum Test der Funktion DoDataset

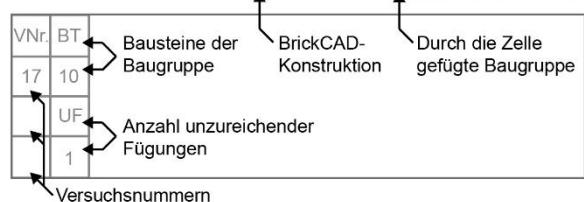
Tabelle 17-4: Testprotokoll zur Validierung der Funktion DoDataset des Servers (Quelle: Eigene Ausarbeitung).

Input	Erwartungs-wert	Über einstimmung	Input	Erwartungs-wert	Über einstimmung
x002y002z000 btE ttB tp0 o0	$\begin{pmatrix} 23400 \\ 23400 \\ 0 \end{pmatrix}$	Ja	x001y002z000 btC ttB tp0 o3	$\begin{pmatrix} 15600 \\ 15600 \\ 0 \end{pmatrix}$	Ja
x005y002z000 btE ttB tp1 o1	$\begin{pmatrix} 39000 \\ 31200 \\ 0 \end{pmatrix}$	Ja	x002y002z000 btB ttA tp0 o0	$\begin{pmatrix} 19500 \\ 19500 \\ 0 \end{pmatrix}$	Ja
x005y003z000 btE ttB tp2 o2	$\begin{pmatrix} 23400 \\ 23400 \\ 0 \end{pmatrix}$	Ja	x002y002z000 btB ttA tp1 o1	$\begin{pmatrix} 19500 \\ 27300 \\ 0 \end{pmatrix}$	Ja
x002y004z000 btE ttB tp1 o3	$\begin{pmatrix} 23400 \\ 23400 \\ 0 \end{pmatrix}$	Ja	x003y002z000 btB ttA tp1 o2	$\begin{pmatrix} 19500 \\ 19500 \\ 0 \end{pmatrix}$	Ja
x002y002z000 btD ttB tp0 o0	$\begin{pmatrix} 23400 \\ 23400 \\ 0 \end{pmatrix}$	Ja	x002y003z000 btB ttA tp0 o3	$\begin{pmatrix} 19500 \\ 27300 \\ 0 \end{pmatrix}$	Ja
x006y002z000 btD ttB tp1 o1	$\begin{pmatrix} 46800 \\ 31200 \\ 0 \end{pmatrix}$	Ja	x002y002z000 btA ttA tp0 o0	$\begin{pmatrix} 19500 \\ 19500 \\ 0 \end{pmatrix}$	Ja
x005y004z000 btD ttB tp0 o2	$\begin{pmatrix} 39000 \\ 31200 \\ 0 \end{pmatrix}$	Ja	x002y002z000 btA ttA tp0 o1	$\begin{pmatrix} 19500 \\ 19500 \\ 0 \end{pmatrix}$	Ja
x003y003z000 btC ttB tp0 o0	$\begin{pmatrix} 39000 \\ 23400 \\ 0 \end{pmatrix}$	Ja	x002y002z000 btA ttA tp0 o2	$\begin{pmatrix} 19500 \\ 19500 \\ 0 \end{pmatrix}$	Ja
x004y002z000 btC ttB tp0 o1	$\begin{pmatrix} 31200 \\ 31200 \\ 0 \end{pmatrix}$	Ja	x002y002z000 btA ttA tp0 o3	$\begin{pmatrix} 19500 \\ 19500 \\ 0 \end{pmatrix}$	Ja
x002y003z000 btC ttB tp0 o2	$\begin{pmatrix} 31200 \\ 23400 \\ 0 \end{pmatrix}$	Ja	x004y001z000 btE ttC tp0 o1	$\begin{pmatrix} 31200 \\ 23400 \\ 0 \end{pmatrix}$	Ja
x001y002z000 btC ttB tp0 o3	$\begin{pmatrix} 15600 \\ 23400 \\ 0 \end{pmatrix}$	Ja	x003y002z000 btD ttC tp0 o0	$\begin{pmatrix} 35100 \\ 23400 \\ 0 \end{pmatrix}$	Ja

17.7 Übersichtstabelle zu den validierten Baugruppen

*Tabelle 17-5: Übersichtstabelle der validierten Lego-Baugruppen
(Quelle: Eigene Ausarbeitung).*

VNr.	BT			VNr.	BT		
1	10			11	10		
	UF				UF		
	3				0		
VNr.	BT			VNr.	BT		
2	10			12	10		
	UF				UF		
	3				0		
VNr.	BT			VNr.	BT		
3	10			13	10		
	UF				UF		
	1				0		
VNr.	BT			VNr.	BT		
4	10			14	10		
	UF				UF		
	0				2		
VNr.	BT			VNr.	BT		
5	10			15	10		
	UF				UF		
	5				0		
VNr.	BT			VNr.	BT		
6	10			16	10		
	UF				UF		
	0				1		
VNr.	BT			VNr.	BT		
9	10			17	10		
	UF				UF		
	0				1		
VNr.	BT			VNr.	BT		
10	10			17	10		
	UF				UF		
	0				1		



18 Verzeichnis verwendeter Software

- CAN LAYER 2 API
Version 5.17
Hersteller: Softing AG
- MICROSOFT VISUAL STUDIO PREMIUM 2013
Version 12.0.30501.00 Update 3
Hersteller: Microsoft Corporation
- MOTOPLUSDK
Version 1.72.00
Hersteller: Yaskawa Electric Corporation
- SOFTING CAN INTERFACE MANAGER
Version 2.69
Hersteller: Softing AG

19 Inhalt der Daten-CD

- Eine digitale Version dieser Arbeit
- Alle, im Rahmen dieser Arbeit verwendeten Abbildungen in den Dateiformaten .eps und .png.
- Folgende Literatur:

ACATECH 2011	LÜDEMANN-RAVIT 2005
BACKHAUS 2014	MAAß ET AL. 2008
BAUERNHANSL 2014	MACKENZIE & ARKIN 1998
BENGEL 2010	MATTHIAS ET AL. 2004
BJÖRKELUND ET AL. 2011	MEDELLIN ET AL. 2010
BJÖRKELUND ET AL. 2012	MICHNIEWICZ & REINHART 2015A
BRECHER ET AL. 2004	MIKUSZ & CSISZAR 2015
BÜSCHNER ET AL. 2013	MITSI ET AL. 2005
CAVIN ET AL. 2013	MOSEMANN & WAHL 2001
CAVIN & LOHSE 2014	RINGERT ET AL. 2014.
CEDERBERG ET AL. 2005	SCHMITT 2014
CUIPER 2000	SCHUNK GMBH & Co. KG 2009
FREUND & HECK 1990	SENDLER 2013
HAUN 2005	SOFTING INDUSTRIAL AUTOMATION GMBH 2012A
HERFS ET AL. 2013	SOFTING INDUSTRIAL AUTOMATION GMBH 2012B
HUCKABY 2014	STENMARK & MALEC 2015
HUCKABY & CHRISTENSEN 2012	STENMARK ET AL. 2015
HUCKABY ET AL. 2013	STOJMENOVIC 2014
HUMBURGER 1998	THOMAS 2009
KAUFMAN ET AL. 1996	WEEKS 1997
KIM ET AL. 2015	YASKAWA ELECTRIC CORPORATION 2011
KRAUS 2015	YASKAWA ELECTRIC CORPORATION 2012A
KRUG 2013	YASKAWA ELECTRIC CORPORATION 2012B
KUGELMANN 1999	YASKAWA ELECTRIC CORPORATION 2013
LOTTER 2012	ZIMMERMANN & SCHMIDGALL 2014

20 Eidesstattliche Erklärung

Eidesstattliche Erklärung

Ich, Christoph Schmitt, erkläre hiermit eidesstattlich, dass ich die vorliegende Arbeit selbstständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt.

Garching, den 31.09.2015

(Schmitt, Christoph)