STUTTGARTER BEITRÄGE ZUR PRODUKTIONSFORSCHUNG

David Albert Breunig

»Integrations- und Steuerungsframework für die Cyberphysische Industrieautomatisierung ISCI«

Fraunhofer IPA

Universität Stuttgart

STUTTGARTER BEITRÄGE ZUR PRODUKTIONSFORSCHUNG BAND 179

David Albert Breunig

»Integrations- und Steuerungsframework für die Cyberphysische Industrieautomatisierung ISCI«

Herausgeber

Univ.-Prof. Dr.-Ing. Thomas Bauernhansl[1,2]
Univ.-Prof. Dr.-Ing. Dipl.-Kfm. Alexander Sauer[1,3]
Univ.-Prof. Dr.-Ing. Kai Peter Birke[4]
Univ.-Prof. Dr.-Ing. Marco Huber[1,2]

[1]Fraunhofer-Institut für Produktionstechnik und Automatisierung IPA, Stuttgart
[2]Institut für Industrielle Fertigung und Fabrikbetrieb (IFF) der Universität Stuttgart
[3]Institut für Energieeffizienz in der Produktion (EEP) der Universität Stuttgart
[4]Institut für Photovoltaik (*ipv*) der Universität Stuttgart

Kontaktadresse:

Fraunhofer-Institut für Produktionstechnik und Automatisierung IPA
Nobelstr. 12
70569 Stuttgart
Telefon 0711 970-1101
info@ipa.fraunhofer.de
www.ipa.fraunhofer.de

Bibliographische Information der Deutschen Nationalbibliothek

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

Zugl.: Stuttgart, Univ., Diss., 2024

D 93

2024

Druck und Weiterverarbeitung:

Fraunhofer Verlag, Mediendienstleistungen, Stuttgart, 2024
Für den Druck des Buches wurde chlor- und säurefreies Papier verwendet.

Dieses Werk steht, soweit nicht gesondert gekennzeichnet,
unter folgender Creative-Commons-Lizenz:
Namensnennung – Nicht kommerziell – Keine Bearbeitungen
International 4.0 (CC BY-NC-ND 4.0).

Integrations- und Steuerungsframework für die Cyberphysische Industrieautomatisierung *ISCI*

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

Vorgelegt von

David Albert Breunig
aus Freiburg im Breisgau

Hauptberichter: Prof. Dr.-Ing. Thomas Bauernhansl
Mitberichter: Prof. Dr.-Ing. Jörg Krüger

Tag der mündlichen Prüfung: 8.7.2024

Institut für Industrielle Fertigung und Fabrikbetrieb (IFF)
der Universität Stuttgart

2024

Inhaltsverzeichnis

Abkürzungsverzeichnis VI

Abbildungsverzeichnis XI

Tabellenverzeichnis XV

Kurzfassung XVII

1 Einleitung 1

1.1 Ausgangssituation 1

1.2 Problemstellung 4

1.3 Thema und Ziele der Arbeit 5

1.4 Aufbau der Arbeit 10

1.5 Verortung 11

1.6 Vorgehen, Methodik und Forschungsfragen 15

1.7 Cyberphysische Systeme 19

2 Grundlagen der Industrieautomatisierung 23

2.1 Verständnis 23

2.1.1 Aufgabe der Industrieautomatisierung 23

2.1.2 Adaptierbarkeit der Automatisierung 24

2.1.3 Determinismus und Echtzeitfähigkeit 25

2.1.4 Umsetzungsvorgehen 27

2.2 Abstraktionsmodelle und ihre technischen Systeme 28

2.2.1 Organisationsebene 29

2.2.2 Steuerungsebene 31

2.2.3 Feldebene 33

2.2.4 Ebenenintegration 33

2.3 Technologien der Feld- und Steuerungsebenen 35

2.3.1 Steuerungen nach der IEC 61131 36

2.3.2 Steuerungen nach der DIN 66253 38
2.3.3 Kommunikation über Feldbusse 40
2.3.4 OPC Unified Architecture (IEC 62541) 40
2.3.5 Programmierung und Modellierung 42
2.4 Zusammenfassung und Bewertung 44

3 Entwicklung der Industrieautomatisierung 49
3.1 Abstraktionssysteme 49
3.2 Technische Frameworks 52
3.2.1 IEC 61499 53
3.2.2 Field Device Integration 56
3.2.3 Verwaltungsschale 60
3.2.4 Distributed Object Model Environment 64
3.2.5 AUTOSAR 66
3.3 Adaptierte allgemeine Ansätze 70
3.3.1 Modularisierung über Serviceorientierung 70
3.3.2 Einbezug externer Plattformen 73
3.3.3 Containerarchitekturen 74
3.3.4 CPS- und System-Engineering 75
3.4 Diskussion 77

4 Das Integrations- und Steuerungsframework *ISCI*: Konzept 87
4.1 Anforderungen 88
4.2 Struktur 89
4.2.1 Lösungsansätze 89
4.2.1.1 Dynamischer, modellbasierter Aufbau verteilter Systeme . 89
4.2.1.2 Gemeinsame Datenmodelle und datenbasierte Integration 90
4.2.1.3 Determinismus über Zustandsorientierung 92
4.2.1.4 Funktions- und Ereignisschnittstellen 93
4.2.1.5 Aufbau auf Standardtechnologien 94
4.2.1.6 Kartierbarkeit auf die Verwaltungsschale 94

4.2.2 Technische Artefakte 96
4.2.2.1 Automatisierungssystem 96
4.2.2.2 Automatisierungsressource 96
4.2.2.3 Datenstruktur 97
4.2.2.4 Module 99
4.2.2.5 Handler 99
4.2.2.6 Administrationsplattform 102
4.2.2.7 Quelle 103
4.2.3 Logische Artefakte 104
4.2.3.1 Anwendungsmodelle 104
4.2.3.2 Dateneinträge und Datenmodelle 105
4.2.3.3 Ereignis- und Funktionsmodelle 107
4.2.3.4 Konfigurationspakete 108
4.2.3.5 Beschreibung 109
4.2.3.6 Abbild 109
4.2.3.7 Schnittstelle 110
4.3 Anwendung 111
4.3.1 Aufbau und Ausführung von Modulen 111
4.3.2 Aufbau einer Automatisierungsanwendung 114
4.3.3 Validierung von Verhalten und Rechtzeitigkeit 117
4.4 Konzept zur VWS-Kartierung 120

5 Das Integrations- und Steuerungsframework *ISCI*: Umsetzung 125
5.1 ISCI-Klassenstruktur 125
5.1.1 Allgemein 126
5.1.2 Anwendungen 128
5.1.3 Beschreibung 130
5.1.4 Daten 134
5.1.5 Konfiguration 137
5.2 ISCI-Softwarestruktur 139
5.2.1 Automatisierungsressource 139

5.2.2 Modulimplementierungen 142
5.2.3 Handler und Handlerintegration 144
5.2.4 Administrationsplattform und Quellen 146
5.3 Implementierung 148

6 Das Integrations- und Steuerungsframework *ISCI*: Anwendung und Evaluation 149
6.1 Funktionale Validierung 149
6.1.1 Implementierung 150
6.1.2 Modulstruktur 154
6.2 Szenariovalidierung 157
6.2.1 Szenario 157
6.2.2 Herkömmliche Automatisierungsstruktur 157
6.2.3 Vorstellung der Automatisierungsstruktur mit ISCI 158
6.3 Ergebnisse 166
6.3.1 Einhaltung der generellen Anforderungen 167
6.3.2 Vergleich mit anderen Frameworks 169
6.3.3 Untersuchungspotentiale 177
6.3.4 Betrachtung der Forschungsfragen 178

7 Zusammenfassung 185

Literatur 191

A Technische Grundlagen des Frameworks 223
A.1 Verteilte Systeme 223
A.1.1 Paradigmen im verteilten System 224
A.1.2 Ausführungsmodelle 226
A.1.3 System- und Datenaggregation 227
A.1.4 Echtzeitfähigkeit und Determinismus 230
A.1.5 Integration mit physischen Anwendungen 231
A.1.6 Entwicklung und Auslieferung 232
A.2 Betriebssysteme 236
A.2.1 Aufgaben und Bestandteile 237

A.2.2 Taskparadigma und Bestandteile 239
A.2.3 Echtzeitfähigkeit 240

B Bewertung der technischen Frameworks 245
B.1 IEC 61499 245
B.2 Field Device Integration 246
B.3 Verwaltungsschale 247
B.4 Distributed Object Model Environment 248
B.5 AUTOSAR 249

C Kernelemente der ISCI-Bibliothek 251

D Implementierte Modulelemente auf ISCI-Basis 263
D.1 Modulskelett für die Modulerstellung 263
D.2 Modul Datenverknüpfung 264
D.3 Modul Abbilderstellung in Datenbank 266
D.4 Modul zur Zykluszeitermittlung 268
D.5 Modul Datenbereitstellung via OPC UA 269
D.6 Modul Interne Kommunikation via UDP 279
D.7 Modul für Kunbus Revolution Pi EA 283
D.8 Modul für Manufacturing Service Bus 288
D.9 Anwendung für die funktionale Validierung 293
D.10 Anwendungen für das Evaluationsszenario 294

E Definitionen 303
E.1 RAMI4.0 303
E.2 SoA 304

Abkürzungsverzeichnis

AP	AUTOSAR Adaptive Platform
API	Application Programming Interface, Programmierschnittstelle
ARM	Advanced RISC Machines
ARMv7	CPU-Architektur von ARM in Version 7
CD	Continuous Delivery
CI	Continuous Integration
CP	AUTOSAR Classic Platform
CPS	Cyberphysisches System
DOME	Distributed Object Model Environment
EA	Ein-/Ausgänge
EDD	Electronical Device Descriptions
ERP	Enterprise Ressource Planning
ESB	Enterprise Service Bus
EVA	Eingabe-Verarbeitung-Ausgabe
FB	Funktionsblock
FDI	Field Device Integration
FDT	Field Device Tool
FTP	File Transfer Protocol
GUI	Graphical User Interface
HMI	Human Machine Interface

HTTP	Hypertext Transfer Protocol
I4.0	Industrie 4.0
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IIoT	Industrial Internet of Things
IoT	Internet of Things
IPC	Industrie-PC
ISCI	Integrations- und Steuerungsframework für die Cyberphysische Industrieautomatisierung
ISO	Internationale Organisation für Normung
JSON	JavaScript Object Notation
M2M	Machine-to-machine
MDE	Model-driven Engineering
MES	Manufacturing Execution System
MIME	Multipurpose Internet Mail Extensions
MQTT	MQ Telemetry Transport (bis v3.1)
MSB	Manufacturing Service Bus
NCK	Numerical Control Kernel
NTP	Network Time Protocol
OPC UA	OPC Unified Architecture
PAS	Publicly Available Specification
PTP	Precision Time Protocol
RAMI4.0	Referenzarchitekturmodell Industrie 4.0
REST	Representational State Transfer

SoA	Serviceorientierte Architektur
SoS	System of Systems
SPS	Speicherprogrammierbare Steuerung
SSH	Secure Shell
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TSN	Time-Sensitive Networking
UDP	User Datagram Protocol
UML	Unified Modeling Language
VDI	Verein Deutscher Ingenieure e.V.
VM	Virtuelle Maschine
VWS	Verwaltungsschale

Abbildungsverzeichnis

1.1 Beschreibung einer Software-Architektur nach ISO 42010 7
1.2 4+1-Sichtenmodell 16
1.3 Wissenschaftsmethodik 18
1.4 CPS-Grundprinzip 19
1.5 Auflösung der Automatisierungspyramide 22

2.1 Prinzip Latenz und Jitter 27
2.2 V-Modell im Erstellungsvorgehen 28
2.3 Automatisierungspyramide 30
2.4 Endlicher deterministischer Automat 38
2.5 AutomationML-Erstellung 44

3.1 Erweiterung der Automatisierungspyramide 50
3.2 Modell der RAMI4.0 51
3.3 IEC-61499-Systemmodell 53
3.4 IEC-61499-Funktionsblock 55
3.5 IEC-61499-Anwendungsmodelle 56
3.6 FDI-Struktur 57
3.7 Aufbau einer Verwaltungsschale 62
3.8 Kommunikation der Verwaltungsschalen 63
3.9 Schichten einer DOME-Architektur 65
3.10 AUTOSAR Classic Modelle 69
3.11 MES-Auflösung nach VDI 5600-7 71

4.1 Grundüberlegung gemeinsame Datenhaltung 92
4.2 Verteilter endlicher Automat 93
4.3 Anordnung um gemeinsame Datenstruktur 94
4.4 Kartierung der Verwaltungsschale 95

4.5 Automatisierungssystem und Automatisierungsressource 96
4.6 Datenstruktur 98
4.7 Funktionsprinzipien der Handler 101
4.8 Integration durch den Handler 102
4.9 Bestandteile der Administrationsplattform 103
4.10 Quellenregister 104
4.11 Zusammenhänge um den Handler 105
4.12 Beschreibungserstellung 110
4.13 Ablage von Abbildserien 110
4.14 Programmablaufplan Modul 112
4.15 Ablauf Ausführung 115
4.16 Lebenszyklusansatz im Framework 116
4.17 Beispiel zur Wahrscheinlichkeitsdichte 119
4.18 Zeitstrecken im verteilten System 120
4.19 Auszug VWS-Metamodell 121
4.20 Automatisierungsressource im VWS-Metamodell 122
4.21 Automatisierungssystem im VWS-Metamodell 123

5.1 Überblick Klassenstruktur 126
5.2 Dateneinträge in Logik und Speicher 137
5.3 Ablagestruktur auf der Automatisierungsressource 141
5.4 Dateneinträge im Arbeitsspeicher 142
5.5 Positionierung eines Handlers 145
5.6 Ablauf Verbindungserstellung 146

6.1 Systemstruktur, funktionale Validierung 150
6.2 RAM-Dateisystem 151
6.3 Angelegte Dateneinträge im RAM-Dateisystem 152
6.4 Systemstruktur, funktionale Validierung 152
6.5 Lokale Verläufe der Dateneinträge 153
6.6 Lokale Verläufe der EA-Dateneinträge auf EA-System 154
6.7 CPS-Strukturprinzip Validierungsszenario 158

6.8 CNC-Automatisierungselemente 158
6.9 Anwendungsmodelle Validierungsszenario 159
6.11 Verbesserungen bei Zeit, Kosten und Qualität 175
6.10 Funktionen Szenario-Automatisierungssystem in MSB 176
6.12 Beschreibung Szenario-Automatisierungssystem in MSB 182

7.1 CPS-Grundprinzip 186

A.1 mdaa-Architektur 229
A.2 Struktur um gemeinsames Informationsmodell 230
A.3 Softwareartefakt im verteilten System 234
A.4 Deploymentsoftware im Prinzip 237
A.5 Bestandteile des Linux-Kernels 239

Tabellenverzeichnis

1.1 Konzeptuelle Beiträge des Frameworks 8

3.1 Frameworks, Elemente und Eigenschaften 83
3.2 Frameworks, Beschreibungen und Ansätze 84
3.3 Frameworks, Normen und Technologien 85

5.1 Grundbefehle der HTTP/REST-API 148

6.1 Bewertung des ISCI gegen Referenzen an Kriterien 171
6.2 Frameworks und ISCI im Vergleich (1/2) 183
6.3 Frameworks und ISCI im Vergleich (2/2) 184

E.1 Asset-Bekanntheitsklassen 303
E.2 VWS-Kommunikationsfähigkeiten 303
E.3 Merkmalklassen 303
E.4 VWS-Implementierungen 304

Kurzfassung

Automatisierung ist fester Teil der Produktion und wichtig für Produktqualität und Produktionszahlen. Ihre technische Basis, die Automatisierungstechnik, ist stärker denn je softwaredefiniert, benötigt aber große Initialaufwände und fachspezifische Kompetenzen, weil prinzipielle Vorteile von Softwaresystemen durch etablierte Architekturen nicht genutzt werden. Heutige Automatisierungssoftware baut auf isolierten statischen Strukturen mit schwachen Schnittstellen und großer Herstellerbindung auf. Neue Paradigmen bleiben unberücksichtigt. Modernisierungsansätze konnten mangels technischer Unterstützung, Komplexität und eingeschränktem Lösungsraum nicht etabliert werden. Der Software der Automatisierung fehlt ein Ansatz, um seine Systeme in moderne *Cyberphysische Systeme* mit adaptionsfähigen Strukturen und offenen Technologien zu überführen.

Die Arbeit begegnet der Problemstellung mit der Entwicklung eines Frameworks für die Software verteilter Automatisierungssysteme gemäß den Paradigmen cyberphysischer Systeme. Als Ausgangsbasis werden das heutige Automatisierungsvorgehen und die verwendete Automatisierungstechnik, automatisierungsspezifische Anforderungen sowie neue Ansätze und Frameworks für verteilte Systeme betrachtet. Darauf aufbauend wird verbunden mit festgelegten Paradigmen eine modellbasierte Struktur aus logischen und technischen Artefakten definiert und in ein Framework zur Konfiguration, Erstellung und zum Betrieb verteilter Automatisierungssysteme integriert. Beachtet wird die Nutzung mit etablierten Mittel- und Hochsprachen, verbreiteten technischen Paradigmen und Architekturen. Das entstandene ***I**ntegrations- und **S**teuerungsframework für die **C**yberphysische **I**ndustrieautomatisierung* (*ISCI*) wird funktional und gegen ein Szenario validiert, sowie erfolgreich bezüglich definierter Anforderungen und anderer Modernisierungsansätze geprüft.

ISCI-Automatisierungssysteme sind CPS, modellbasiert, verteilt, ereignisfähig, zustandsorientiert und mit bewertbarem Verhalten, um verteilte Steuerungsanwendungen deterministisch auszuführen. *ISCI*-Modelle und -Artefakte sind standardbasiert und frei erweiterbar, womit sich *ISCI* als Ausgangspunkt für die Weiterentwicklung von Automatisierungssoftware, -komponenten und -systemen eignet.

Abstract

Automated processes are an integral part of production and important for product quality and production figures. Automation technologies as their basis are more software-defined than ever but require great initial efforts and specialist expertise because the fundamental advantages of software systems are not used by established automation architectures. Today's automation software is based on isolated static structures with weak interfaces and strong vendor lock-ins. New paradigms remain unused. Modernization approaches could not be established due to the lack of technical support, complexity, and limited technical possibilities. Automation software lacks an approach to transform its systems into modern *Cyberphysical Systems* with adaptive structures and open technologies.

This work addresses the problem by developing a framework for the software of distributed automation systems according to the paradigms of cyberphysical systems. As a starting point, current automation procedures and technology, automation-specific requirements, and new approaches and frameworks for distributed systems are considered. Based on this, a model-based structure of logical and technical artifacts is defined considering established paradigms and integrated into a framework for the configuration, creation and operation of distributed automation systems. Attention is paid to the use of established middle and high-level languages, common technical paradigms and architectures. The resulting *Integration and Control Framework for Cyberphysical Industrial Automation* (*ISCI*) is validated functionally and against a use case and successfully tested against defined requirements and other modernization approaches.

ISCI automation systems are CPS, model-based, distributed, event-driven, stateful, and with assessable behavior to deterministically execute distributed control applications. *ISCI* models and artifacts are standards-based and freely extensible, making *ISCI* a suitable starting point for further development of automation software, components, and systems.

1 Einleitung

1.1 Ausgangssituation

Automatisierung ist ein fester Bestandteile der Fabrik und ihres Betriebs. Die Automatisierung erstreckt sich von der Bewegungssteuerung in Maschinen über die Ablaufsteuerung in Produktionslinien hin zu instationären, selbstfahrenden Transportsystemen (Nof 2009). Sie leistet durch die ermöglichte Kostensenkung einen wichtigen Beitrag zum heutigen Lebensstandard und erlaubt bei minimiertem menschlichen Aufwand gleichbleibende Produktqualität und große Produktionsmengen (Nof 2023; Lamnabhi-Lagarrigue & Samad 2023; Chobanov & Hardalov 2022; Nof 2009). Gleichzeitig benötigt Automatisierung große Initialaufwände und erfordert umfangreiche Kompetenzen, die heute mehr denn je im Softwarebereich liegen: Die mechanisch, elektrisch oder fluidisch definierten Automatisierungen sind von digitalen Systemen verdrängt worden, die sich durch Programmierbarkeit und Kommunikationsfähigkeiten eindeutig hervorheben (Ellwein et al. 2023; Neumann et al. 2022; Vogel-Heuser et al. 2014).

Zukünftige Produktionsprozesse müssen angesichts individualisierter Produkte, sich dauernd wandelnder Nachfrage und Konkurrenzdrucks die Forderung nach ausreichender Adaptierbarkeit erfüllen (Napoleone et al. 2023; Bahtat et al. 2023; Dafflon et al. 2021; ElMaraghy & Wiendahl 2009). Auf Ebene physischer Produktionssysteme soll die Adaptierbarkeit zukünftig vor allem über eine Rekonfigurierbarkeit der Produktionssysteme geschehen (Bahtat et al. 2023; ElMaraghy et al. 2021). Das Konzept der Rekonfigurierbarkeit basiert auf dem Austausch oder der Neuzusammenstellung virtueller und physischer Systemteile während oder nahe der Laufzeit. So soll ein Produktionssystem gezielt und schnell auf eine neue Aufgabe ausgerichtet werden. Als Engineering- und Simulationsbasis dienen dabei virtuelle Begleitungen wie der digitale Zwilling. (Bahtat et al. 2023; ElMaraghy et al. 2021)

Die Adaptierbarkeit von Produktionssystemen via rekonfigurierbarer Software ist ein aktu-

eller und vielversprechender Untersuchungsgegenstand (Ellwein et al. 2023; ElMaraghy et al. 2021), da Softwaresysteme, verglichen mit anderen technischen Systemen, in ihren Eigenschaften leicht änderbar sind: Ihre Gestalt ist nicht an physische Körper gebunden, sondern wird textuell oder graphisch bestimmt und durch Übersetzung in Maschinensprache maschinell ausführbar. Für die Erstellung moderner Softwaresysteme existieren zudem zahlreiche frei verfügbare und standardisierte Sprachen und Werkzeuge, Referenzarchitekturen, Rahmenwerke und Bibliotheken. Diese Vorteile moderner Software stünden für Automatisierungssysteme bereit, werden allerdings mangels Technologien und Werkzeuge nicht genutzt (Feichtinger et al. 2022; Vogel–Heuser et al. 2022; Neumann et al. 2022).

In der Automatisierungsprojektierung wird ein Automatisierungssystem mit einem initial bestimmten Ziel und Anwendungsfall geplant (Seitz 2021). Diese Ausrichtung wird an die Automatisierungssoftware, die als ein fester und unveränderlicher Systemteil betrachtet wird, weitergegeben. So werden in der Praxis physische und virtuelle Konfigurationen und Programmierungen fest auf einen definierten Prozess oder Anwendungsfall ausgerichtet (Vogel–Heuser et al. 2022). Der Fokus liegt auf einmalig eingerichteter und nicht oder selten umdefinierter Automatisierung. Dieses Vorgehen wird durch die Beschaffenheit der Soft- und Hardware der industriell geprägten Komponenten unterstützt: Konfigurations- und Programmänderungen zur oder nahe der Laufzeit sind in ihren Architekturen nicht berücksichtigt und die Integratoren werden in veraltete zeitaufwändige Entwicklungsumgebungen gezwungen. Dazu zählen insbesondere die durch Industrieentwicklungen getriebenen IEC-61131-Systeme, die den Feldstandard in der Automatisierungssteuerung stellen (Neumann et al. 2022; Vogel–Heuser et al. 2022; Rothhöft 2022; Rothhöft 2018). Proprietäre geschlossene Architekturen und mangelnde Interoperabilität sind große Probleme der heutigen Feld- und Steuerungskomponenten (Vogel–Heuser et al. 2022; Rothhöft 2022; Rothhöft 2018). Software- und Kommunikationsänderungen verursachen Maschinen- und Anlagenstillstände, weil sie nicht zur Laufzeit stattfinden können (Lesi et al. 2023). Standardtechnologien bei Programmiersprachen und Hardware, die stärkere Trennung von Hard- und Software, die umfassende Nutzung digitaler Technologien und die Transformation zur dezentralen Systemsteuerung sind unter anderem notwendig um rekonfigurierbare Produktionssysteme zu ermöglichen (Napoleone et al. 2023; Ellwein et al. 2023).

Einschränkungen proprietärer und geschlossener Technologien oder Rahmenwerke wirken nicht nur auf der unmittelbaren Maschinen- und Anlagenebene. Wie gut, in welcher Art und Weise und mit welchen Funktionen vertikale Integration zwischen Geschäfts- und Produktionssystemen ausgeführt werden kann, variiert und unterliegt ebenfalls den Einschränkungen proprietärer und geschlossener Hard- und Software der Feld- und Steuerungsebenen (Schnicke et al. 2022; Ye et al. 2022).

Aufgrund mangelnder Interoperabilität beschränkt sich die vertikale Integration von Feld- und Steuerungssystem bislang auf den Austausch einfacher Daten, zum Beispiel das Setzen von Variablen, oder die Weitergabe von Identifikationsinformationen. Den höheren Ebenen bleibt die Architektur der Automatisierung verborgen. (Schnicke et al. 2022; Tabim et al. 2021) Konzepte zur unternehmens- oder standortübergreifenden vertikalen Integration existieren (Nof 2023; Körner et al. 2019), haben aber keine nennenswerte Praxisadaption.

Bisherige Ansätze zur Neustrukturierung, beispielsweise mit neuen Modellier- und Programmiersystemen sowie konfigurationsbasierten oder selbstintegrierenden Architekturen, scheiterten unter anderem an der mangelnden Unterstützung von Komponentenherstellern, nicht beherrschbarer Komplexität, der Bindung an variierende physische Systeme und nicht vorab bekannte Zielprozesse oder zu stark eingeschränktem Zielraum (Feichtinger et al. 2022; Vogel–Heuser et al. 2022). Mit dieser Arbeit wird ein Architektur- und Softwareframework für die Automatisierung vorgestellt, das die Softwaredefinition der Automatisierung, ihre tiefe Integration über Systeme hinweg, ihre Verteilung im Netzwerk und ihre bedarfweise Rekonfiguration gezielt unterstützt. Dafür wird eine technische und logische Grundstruktur umgesetzt, die darauf ausgelegt ist, modellbasiert instanziiert zu werden, mit freien Sprachen und Entwicklungsumgebungen auf generalisierter Hardware zu arbeiten, Automatisierungsanforderungen zu erfüllen und Vernetzung und Verknüpfung mit externen Systemen zu unterstützen.

1.2 Problemstellung

Den digitalen Systemen in der Industrieautomatisierung fehlt eine Grundstruktur, um die heute nur punktuell integrierten und in ihrer Beschaffenheit starren Inselsysteme der Produktion in wandelbare *Cyberphysische Systeme* (CPS) zu überführen und die Potentiale moderner Software zu nutzen (Zhang et al. 2023). Es existieren keine ganzheitlichen CPS-Frameworks für industrielle Anwendungen (Feichtinger et al. 2022).

Mit den in der Automatisierung verwendeten Normen und Frameworks werden abgetrennte, hierarchische Ebenen geschaffen, zwischen denen eingeschränkte Kommunikation stattfindet (Schnicke et al. 2022). Die Systeme tauschen Daten aus, interagieren und kooperieren aber nicht. Sie bilden kein gemeinsames verteiltes Produktionssystem. Verteilte und cyberphysische Architekturen gelten jedoch als Zukunft der Automatisierung (Dafflon et al. 2021; Eckert 2015), bedingt durch die wachsende Komplexität der zu automatisierenden Maschinen und Anlagen und der zunehmenden Zahl vernetzter, datenliefernder und intelligenter Feldkomponenten. Diese Architekturen sind mit den heutigen Modellier- und Programmiersystemen wie der IEC 61131 nicht realisierbar, wodurch sie keine praktische Relevanz haben (Rothhöft 2022; Rothhöft 2018).

Um individualisierte Produkte mit automatisierten Systemen fertigen zu können, muss die Industrieautomatisierung adaptierbar werden (Dafflon et al. 2021; Zhong et al. 2017). Die Softwaresysteme zur Erfassung und Steuerung physischer Produktionseinrichtungen werden statisch konfiguriert und programmiert und nutzen die durch den Softwareeinsatz gegebenen Potentiale nicht (Vogel–Heuser et al. 2022). CPS sind ein nutzbarer Ansatz für die adaptierbare Produktion, müssen aber entwickelt werden (Dafflon et al. 2021). Die rekonfigurierbare und generalisierbare Integration von Software und physischen Einrichtungen ist eine Hürde für die CPS-Automatisierung (Rikalovic et al. 2022).

Etablierte Softwaresysteme nutzen herstellerspezifische Hardware und stellen für das IIoT notwendige Technologien nicht oder nur als proprietäre Erweiterungen bereit, wodurch die Umsetzbarkeit von Lösungen eingeschränkt wird und Herstellerwelten entstehen (Vo-

gel–Heuser et al. 2022; Rothhöft 2022; Rothhöft 2018). Die verbreiteten Standard-, Open-Source- und Virtualisierungstechnologien der Web-, Cloud- und PC-Welten stehen der Automatisierung dadurch nicht zur Verfügung. Durch die Geschlossenheit der Softwareanwendungen ist die Automatisierung auch fabrik- oder unternehmensintern intransparent.

Schnittstellen und die Funktionen dahinter sind bei Automatisierungsanwendungen vorgegeben oder programmatisch definiert. Dadurch sind Anpassung oder Erstellung, egal ob manuell oder automatisch, nicht möglich oder erschwert. Mit der OPC Unified Architecture (OPC UA) sind teilweise vordefinierte, konfigurierbare Schnittstellen bei Komponenten aus verschiedenen Ebenen eingeführt worden, die aber nur Repräsentation und Integration des eigenen Systems vorsehen und keine anderen Systeme ohne gesonderte Programmierung miteinbeziehen können. Für eine umfassendere Automatisierung, die über standardisierte native Schnittstellen auch unternehmens- oder standortexterne Systeme oder Nutzer einbeziehen kann, existiert kein bekannter Ansatz.

1.3 Thema und Ziele der Arbeit

Die Arbeit begegnet der Problemstellung durch Entwicklung eines grundlagenorientierten Frameworks, das adaptierbare, ineinandergreifende Strukturen zur Architekturdefinition von Automatisierungssystemen bereitstellt und ein auf den Strukturen aufbauendes generalisiertes Softwareframework implementiert. Das Framework zielt auf die softwaredefinierte Automatisierung mit verteilten, kooperierenden Systemen nach CPS-Paradigmen ab. Das Framework übernimmt die Bereitstellung von Strukturen und Modellen für ein verteiltes System selbst als auch für die auf den einzelnen Systemen umgesetzten Softwarekomponenten, wobei die aufgestellten Strukturen und Modelle während Aufbau, Betrieb und Rekonfiguration wirksam sind. Über die Softwarekomponenten werden einzelne technische Systeme allgemein und anwendungsgerichtet konfiguriert und zu verteilten Systemen integriert. Das Framework löst keine Steuerungsaufgaben, sondern gibt ein Grundgerüst vor, mit dem diese Aufgaben als verteilte, rekonfigurierbare Anwendungen mit fähigen Schnittstellen realisierbar sind. Die Frameworkdefinition bezieht die wissenschaftlichen

Überlegungen zu CPS ein. Folgende generelle Anforderungen sind für die Entwicklung zentral:

1. Aufarbeitung *Cyberphysischer Systeme* für die Automatisierung, siehe Abschnitt 1.7
2. (Re)Konfiguration als native Funktion in die Automatisierung einführen
3. Automatisierungsanforderungen (u.a. Zeit, Interoperabilität) abdecken
4. Transparenz und Verständlichkeit zur Entwicklungs- und Laufzeit
5. Konfigurierbare logische Integration verteilter Systeme ermöglichen
6. Standardparadigmen und freie Software nutzen und fördern
7. Implementierungs- und Anwendungsnähe wahren

Die Arbeit unterstützt durch die Einführung von Konfigurierbarkeit, Zugang zu Software und Daten, großer Transparenz, Adaption an Standardtechnologien, Adaptierbarkeit und des CPS-Ansatzes den Wandel zur softwaredefinierten Produktion. Traditionell wird die Beschaffenheit von Produktionssystemen durch physische Komponenten bestimmt. Physische Komponenten sind durch harte Eigenschaften wie Geometrie oder Parametergrenzen beschränkt, aber auch unbedingter Bestandteil von Produktionssystemen. Um trotz physischer Einschränkungen dynamische Produktionssysteme zu erzeugen muss der Softwareteil bestimmend werden. In der softwaredefinierten Produktion werden die Produktionsteile, die nicht auf eine physische Definition angewiesen sind, von physischen Teilen abgetrennt. (Xu et al. 2021) Durch diese Virtualisierung können Logik und Abläufe über Schnittstellen konfigurierbar gemacht werden. Die Konfigurationen können bei Gewährleistung von Konsistenz bei Formaten und Schnittstellen auch automatisiert erfolgen. Die Produktionsteile, die auf eine physische Definition angewiesen sind, können mit konsistent trennbaren Softwareschichten in eine ganzheitliche Konfigurierbarkeit überführt werden. Der Ansatz der Softwaredefinition ist auf gemeinsame Grundstrukturen und automatisierungsfähige Lösungs- und Abstraktionsansätze angewiesen. Er wird mit dem Framework in der Industrieautomatisierung ermöglicht.

Begriffe Architektur und Architekturframework

Das zu entwickelnde Konzept beschreibt ein Framework zur Implementierung und zum Betrieb von verteilten Systemen, die das Ziel haben, physische Prozesse unter CPS-Paradigmen zu steuern. Das Begriffsverständnis zu Architekturen und zugehörigen (Architektur)Frameworks orientiert sich an der ISO/IEC/IEEE 42010 (siehe ISO/IEC/IEEE 42010, Abbildung 1.1). Ein entstehendes Automatisierungssystem entspricht in der Abbildung 1.1 dem System. Das entwickelte Automatisierungsframework liefert Standardteile für die Systemarchitektur und die Architekturbeschreibung sowie für die Komponenten, die Systemzwecke erfüllen und durch Teilhaber implementiert werden können. Es ist ein erweiterbares, komponenten- und modularisierungsfähiges Architekturframework, das für die Entwicklung Softwarekomponenten für Automatisierungssysteme, für die Konfiguration Modelle und Erstellungslösungen und für die Laufzeit Funktionen zur Zielerfüllung, Überwachung, Bewertung und Administration der Automatisierungssysteme bereitstellt.

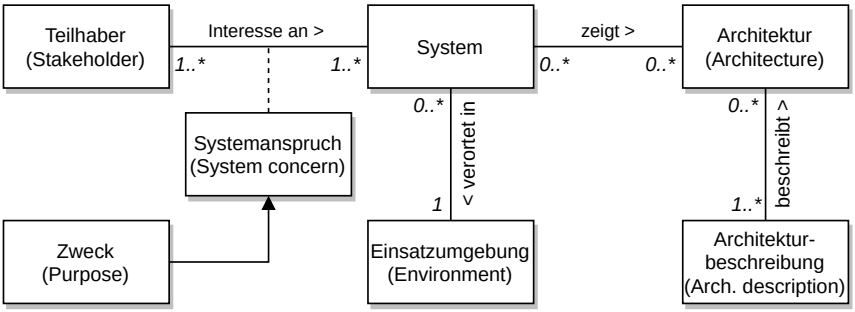


Abbildung 1.1: Verhältnisse einer technischen Architektur bezogen auf seine Umgebung nach der ISO 42010 (ISO/IEC/IEEE 42010)

Wirkzeiten

Für ein Automatisierungsframework sind Vorgehensweisen wie zum Beispiel das V-Modell (siehe Abschnitt 2.1.4 und Abbildung 2.2), an denen sich die Umsetzung eines Automatisierungsprojekts üblicherweise orientiert (Eckert 2015), nicht unmittelbar geeignet. Ein Framework ist nicht an ein praktisches Projekt gebunden, sondern stellt ein konsistentes Rahmenwerk aus wiederverwendbaren Elementen für Projekte aus einer Domäne bereit (Riehle 2000). Es besteht dazu aus generalisierten Lösungskonzepten und technischen Kom-

ponenten, mit denen während der Umsetzung eines Automatisierungsprojekts und dem anschließenden Betrieb Probleme gelöst werden. Es wird separat von Automatisierungsprojekten entwickelt. Gezielte Entwicklungsabfolgen wie das V-Modell sind deshalb nicht geeignet. Das Framework und seine Elemente werden in Automatisierungsprojekten, die nach dem V-Modell oder ähnlichen Modellen organisiert sind, genutzt, um Automatisierungen zu erstellen. Deshalb werden die Aufgaben des Frameworks an den Lebenszyklusphasen eines Automatisierungssystems, für das es verwendet wird, orientiert. Nach den Definitionen für Automatisierungsanwendungen nach Meyer 2002 sind dies:

1. **Entwicklung:** Komponentenerstellung, Standardsysteme
2. **(Re)Konfiguration:** Systemdefinition, Systemerstellung, Systemänderung
3. **Betrieb:** Funktionserfüllung, Administration

Die Bereiche bauen aufeinander auf, das heißt, Artefakte des Frameworks, die in der Entwicklung entstehen, sind auch zur Konfigurations- und zur Laufzeit aktiv, allerdings dann als Bestandteil eines übergeordneten technischen Systems.

	Frameworkbeiträge	**Zielergebnisse**
Entwicklung	Modelle, Paradigmen, Ansätze, Funktionen, Softwaredefinitionen	frameworkfähige Modellierungsbasen, wiederverwendbare Software- und Konfigurationskomponenten
(Re)Konfiguration	Modelle, Abläufe, Analysefunktionen, Softwaredefinitionen	Systemdefinitionen, Erstellungsfunktionen für verteilte Systeme, Analyseergebnisse zu Verhalten gebauter Systeme
Betrieb	Modelle, Analysefunktionen, Plattform	automatisierter, abgebildeter Betrieb verteilter Automatisierungssysteme

Tabelle 1.1: Die Beiträge des Frameworks in den drei Bereichen und die Ergebnisse in abstrahierter Form, die mit Frameworkbeiträgen erstellt werden

Entstehende Artefakte

Mit der Arbeit entsteht ein Framework, das aus technischen und logischen Artefakten besteht. Technische Artefakte sind aktive Softwarekomponenten oder Strukturen für aktive Softwarekomponenten, die Frameworkteil sind oder bei Frameworkverwendung entstehen. Logische Artefakte sind Elemente zur Informationswiedergabe und Struktur- sowie Integrationsmodellierung. Sie werden mit Basiselementen in einer Softwarebibliothek aggregiert. Die Artefakte sind Modelle und Softwarestrukturen zur Implementierung einer CPS-basierten Automatisierung. Mit *Einzelsystem* sind Teilsysteme eines verteilten Systems gemeint. Die Artefakte sind:

1. **Technische Artefakte**
    - Automatisierungssystem und Automatisierungsressourcen: entstehende verteilte Systeme zur Automatisierung und daran beteiligte Einzelsysteme
    - Datenstruktur: lokale Softwarestrukturen auf Automatisierungsressourcen zum Datenaustausch zwischen Softwarekomponenten und Systemen
    - Module und Handler: Softwarestrukturen zur Implementierung von Automatisierungsfähigkeiten auf Systemen und zur Administration und Einrichtung der Automatisierungsressourcen
2. **Logische Artefakte**
    - Anwendungsmodelle: Modellstrukturen zur Definition des Aufbaus und der Konfiguration von verteilten Systemen
    - Dateneinträge und Datenmodelle: Grundlagen für die Einrichtung der automatisierungsgerechten zustandsorientierten Integration der verteilten Systeme
    - Ereignisse und Funktionen: Objektbasierte Definitionsmöglichkeit zur Berücksichtigung von Schnittstellen zwischen verteiltem System und Umgebung
    - Konfigurationsmodelle: Definitionsklassen für die Vorgabe von Softwareverteilungen und Einrichtungsvorgängen im verteilten System
    - Beschreibungen: Modelle als Wiedergabebasis für die Repräsentation von Systemen und Systembestandteilen für Administration und Integration

1.4 Aufbau der Arbeit

Im weiteren Teil dieses Kapitels wird die Arbeit thematisch und in ihren Zielbereichen verortet (Abschnitt 1.5), die Forschungsinhalte vorgestellt (Abschnitt 1.6) sowie das Hintergrundthema CPS beleuchtet (Abschnitt 1.7).

Im **Kapitel 2** erfolgt eine Erläuterung der speziellen Aspekte und spezifischen Technologien der Industrieautomatisierung, die für das Verständnis der Arbeit notwendig sind. Das Kapitel wird mit einer Zusammenfassung und Bewertung geschlossen.
Mit dem **Kapitel 3** werden existierende automatisierungsspezifische Entwicklungsansätze für Abstraktion, Frameworks und einzelne Technologien gezeigt, die im Rahmen der Arbeit bedeutend sind. Am Kapitelende folgt eine Diskussion der vorgestellten Ansätze.
Mit dem **Konzeptkapitel 4** beginnt der Kern der Arbeit. Es werden Anforderungen an das Framework definiert, die Struktur in ihren Lösungsansätzen und Artefakten festgelegt und die vorgesehene Frameworkanwendung besprochen. Zudem erfolgt eine Bezugnahme auf die Integration mit der Verwaltungsschale.
Das **Kapitel 5** zeigt die Umsetzung und den Aufbau des Frameworks, aufgeteilt in die logische Klassenstruktur und die implementierungsorientierte Softwarestruktur. Das Implementierungsvorgehen wird ebenfalls vorgestellt.
Den Kern der Arbeit schließt das **Kapitel 6** mit der Vorstellung von Frameworkanwendung und -evaluation. Das Kapitel zeigt die funktionale und die szenariobasierte Validierung, vergleicht das Framework mit Referenzansätzen, führt identifizierte Potentiale auf und beleuchtet die Ergebnisse aus Anforderungs- und Forschungssicht.
Geschlossen wird die Arbeit mit einer Zusammenfassung aller Inhalte im **Kapitel 7**.

Im Anhang werden die technischen Grundlagen für die Arbeit erläutert (Anhang A), die Detailwertung der Referenzframeworks für die Frameworkentwicklung (Anhang B), Texte der Modulprogramme und Modelle aus Umsetzung und Evaluation (Anhang D) sowie im Rahmen der Kapitel anfallende Begriffsdefinitionen für das Verständnis (Anhang E).

1.5 Verortung

Im Folgenden wird die Arbeit bezüglich den adressierten Gebiete in der Automatisierung und den anvisierten technischen Plattformen positioniert, eine Bezugnahme zum *Referenzarchitekturmodell Industrie 4.0* und der *Verwaltungsschale* hergestellt, sowie eine Verortung zur IEC 61131 und zur Automatisierungspyramide durchgeführt.

Adressierte Gebiete in der Automatisierung

Die Arbeit stellt ein Framework zur Realisierung verteilter Systeme vor, die gemeinsam und mit dauerndem Datenaustausch physische Produktionsaufgaben realisieren. Das direkte Einsatzziel des verteilten Systems liegt auf den äußerst physisnahen Systemen, die Maschinen oder Anlagen und die Abläufe darin überwachen, steuern und regeln. Das verteilte System, das mit dem Framework aufgebaut wird, ist dabei im Sinne der gesteuerten physischen Vorgänge echtzeitfähig und in der Lage, Anwendungen zur Steuerung physischer Vorgänge deterministisch[1] auszuführen. In einer Instanz werden im gebildeten verteilten System alle technischen Systeme integriert, denen der physische Zustand im Prozesssinn in Echtzeit bekannt sein muss. Dies betrifft alle physiserfassenden (Sensorik), physissteuernden (Steuerung), physisstellenden (Aktorik) und physisabbildenden (Datenverarbeitung) Systeme. Externe Systeme, die nicht echtzeitfähig beteiligt sein müssen, werden über mit Frameworkfunktionen definierten Schnittstellen zum verteilten System einbezogen.

Das Framework ist anwendungsinvariant und demnach nicht an eine Form von Prozess oder Ablauf gebunden. Es wird für eine Anwendung konfiguriert und instanziiert. Die genaue Qualität der verteilten Systeme richtet sich nach der umgesetzten Anwendung.

Anvisierte technische Plattformen

Das Framework wird auf technische Plattformen ausgelegt, mit und auf denen eigene Software oder Softwareteile mit großer Gestaltungsfreiheit betrieben werden können. Abgeschlossene Systeme, wie beispielsweise Feldgeräte mit fester Firmware, werden nicht unmittelbar adressiert. Für solche Systeme sind Schnittstellen umsetzbar. Mit dem Framework wird ein Beitrag zur Hardware- und Herstellerunabhängigkeit sowie zur Weiterentwicklung der Industrieautomatisierung weg von ihren bislang eingeschränkten spezifischen

[1] Zu Determinismus und Echtzeitfähigkeit siehe Anhang A.1.4

Systemen geleistet. Dementsprechend ist die Gestaltung auf verbreitete Prozessor- und Speicherarchitekturen wie x64 oder ARM mit Zugriff auf die Dateisysteme sowie Standardnetzwerkschnittstellen ausgerichtet. Erstellte verteilte Systeme des Frameworks sind CPS und setzen als solche eine ständige Vernetzung über die Schnittstellen voraus, nach innen und außen. Für kritische, prozessrelevante Kommunikation innerhalb eines mit dem Framework erstellten verteilten Systems sind hinreichend fähige Technologien bei den technischen Plattformen notwendig. Die prozessrelevante Kommunikation kann über die Standardnetzwerkschnittstellen ausgeführt werden, wenn sie ausreichend fähig sind und deterministisch arbeiten können. Zur Erfüllung der Anforderungen setzt das Framework deterministische Betriebssysteme voraus.

Die technischen Plattformen müssen nicht statisch oder an eine physische Entität gebunden sein. Virtuelle Maschinen oder Containerumgebungen sind als Ausführungsplattformen möglich, wenn die Anforderungen des automatisierten Prozesses eingehalten werden. Technische Plattformen, die keinen vollständigen Frameworkumfang implementieren können, beispielsweise Mikrocontroller oder IEC-61131-Systeme, können über Brückenlösungen (externe befähigte Handler und manuelle Anbindungen an die Datenstrukturen in einem verteilten System) mitberücksichtigt werden, sind aber nachrangig behandelt.

Für die technischen Plattformen der Begleitsysteme zur Administration, Modellierung und Softwarebereitstellung werden keine speziellen oder besonderen technischen Systeme vorgesehen. Da sie im Framework zur Laufzeit außerhalb prozesskritischer Anforderungen laufen, müssen sie über ihre Eignung zur Erfüllung ihrer funktionalen Aufgaben (Vernetzung, Dateibereitstellung und -ablage, Erreichbarkeit) aufgebaut werden.

Verhältnis zu Referenzarchitekturmodell I4.0 und Verwaltungsschale

Das Referenzarchitekturmodell Industrie 4.0 (RAMI4.0) gibt der Arbeit eine bedeutende Orientierung, da es zum Zeitpunkt der Erstellung der größte akademisch und industriell unterstützte Ansatz zur Weiterentwicklung der Automatisierung und genormt ist. Das RAMI4.0 wird in Abschnitt 3.1 näher erläutert. Das RAMI4.0 konzentriert sich auf ein unternehmensübergreifendes Verständnis des Themas *Industrie 4.0*, das hauptsächlich in der Entwicklungszeit von Systemen für die Industrie 4.0 wirkt. Das Verständnis umfasst unter anderem ein Abstraktionsmodell sowie Deskription-, Sprach- und Semantikmodelle.

Die im Kontext des RAMI4.0 wichtige *Verwaltungsschale* (VWS, siehe Abschnitt 3.2.3) ist für diese Arbeit inhaltlich relevant. Die VWS ist eine Softwarestruktur, die verschiedene Ausbaustufen aufweisen kann und ein *Produktionsasset* für die Einbettung in digitalen Strukturen der Produktion befähigt (Schnicke et al. 2022). Der Mindestumfang einer VWS ist eine passive Beschreibung in einem Standardformat. Im größten Umfang ist eine VWS ein aktives Softwaresystem, mit dem ein Asset technisch integriert wird und das Funktionen und Kommunikation des Assets zugänglich macht. Die VWS-Struktur kann so als Strukturgrundlage für ein erweiterbares Softwaremodell für Automatisierungssysteme adaptiert werden (Breunig et al. 2020), sodass allgemeingültige Beschreibungsformate einhaltbar sind. Entsprechend wird die VWS und eine Adaption der VWS an das zu entwickelnde Framework diskutiert. Der VWS-Ansatz wird durch die Adaption zur aktiven und rekonfigurierbaren Softwarestruktur erweiterbar, die den Aufbau und den Betrieb verteilter Automatisierungssysteme ermöglicht.

Verhältnis zur Automatisierungspyramide

Mit dem Ansatz wird der Aufbau verteilter Automatisierung mit gleichwertig vernetzten Systeme unterstützt, die kooperierend arbeiten, keiner Hierarchie unterliegen sind und deren Architektur jederzeit rekonfiguriert werden kann. Damit steht der Ansatz in Kontrast zur hierarchischen und statischen Automatisierungspyramide. Die zweidimensionale Abstraktion der Automatisierung in der Automatisierungspyramide ist für CPS-Ansätze nicht geeignet, weil sie die um physische Prozesse angeordneten CPS nicht wiedergeben kann. Da im Ansatz der Arbeit zudem keine statische Integration vorliegt, sondern Systeme als Ressourcen gesehen und nach Bedarf in einem verteilten, softwaredefinierten System integriert werden, sind die festen horizontalen und vertikalen Integrationswege der Automatisierungspyramide nicht passend. Im Folgenden wird mit Eckpunkten eine Verortung hergestellt, um eine Bezugnahme zur gegenwärtigen Automatisierung zu erleichtern. Mit *Ebenen* sind die Ebenen der Automatisierungspyramide gemeint.

- Das Framework ist an keine Ebene gebunden, realisiert keine ebenenspezifische Anwendung und verfügt nicht über ebenenspezifische Ausprägungen.
- Das Framework sieht keine Hierarchien für die beteiligten Systeme vor.

- Für die Anwendung müssen die beteiligten Systeme eines verteilten Systems Teil einer gemeinsamen Vernetzung sein.
- Durch die Datenintegration liegen beteiligte Systeme aus beliebigen Ebenen in einer gemeinsamen Logikebene.
- Das Framework integriert eine bestimmte Menge aus Systemen aus beliebigen bisherigen Ebenen zur gemeinsamen Lösung einer Automatisierungsaufgabe.

Die Arbeit konzentriert sich auf die Systeme der unteren Ebenen (Feldebene, Steuerungsebene, Prozessleitebene), da diese Ebenen die physische Welt miteinbeziehen und die harten Anforderungen hinsichtlich logischer Konsistenz, Zeitanforderungen und Determinismus aufstellen. Da die oberen Ebenen, Betriebs- und Unternehmensleitebene, leichter zu bewältigende technische Anforderungen stellen, können diese Anforderungen über die Anforderungen der unteren Ebenen mit abgedeckt werden.

Verhältnis zur IEC 61131

Die IEC 61131 ist die Norm, nach der heute der größte Teil der Steuerungssysteme in der Automatisierung gestaltet sind. Sie definiert Grundmodelle für Steuerungssysteme und deren Programme und Datenstrukturen sowie spezifische Programmiersprachen für die Automatisierungssteuerungen. Sie wird in Abschnitt 2.3.1 genauer betrachtet. Die IEC 61131 ist ein Rahmen- und Regelwerk für den graphischen oder textuellen Bau von prozessspezifischen Steuerungsprogrammen, die auf einem Gerät mit einer singulären Datenstruktur zur Steuerung eines abgeschlossenen Prozesses ausgeführt werden.

Das Framework liegt in der Anwendung eine Ebene unter der IEC 61131: Es führt selbst keine Steuerungsaufgaben aus, sondern stellt eine Instanziierungs- und Ausführungsgrundlage für Steuerungsanwendungen und eine Daten- und Kommunikationsstruktur, auf die Steuerungsanwendungen zugreifen können. Auf Frameworksystemen können IEC-61131-Systeme ausgeführt werden, solange diese keine spezifische Hardware voraussetzen. Das Framework zielt darauf ab, Komponenten für rein virtuelle verteilte Systeme bereitzustellen und mit diesen verteilten Systemen die heutigen IEC-61131-Systeme zu ersetzen. Die bereitgestellten Komponenten basieren dabei nicht auf automatisierungsspezifischen

Technologien, sondern auf verbreiteten Standardprogrammiersprachen und werden auf Standardhardware ausgeführt.

1.6 Vorgehen, Methodik und Forschungsfragen

Bei der Konzepterstellung und -umsetzung werden Grundsätze beachtet, die das Entwickeln eines stringenten und qualitativ hochwertigen Frameworks unterstützen. Diese Grundsätze wirken auch nach der initialen Definition, weil über das Framework auch aufbauende Software und ihre Nutzung geprägt werden. Allgemein beachtet wird der Grundsatz *einfach, eindeutig, sicher* (Feldhusen & Grote 2013): Problemlösung und Systemanwendung müssen einfach gestaltet und umsetzbar sein; die Systemgestaltung (insbesondere die Programmierung) und das Verhalten müssen eindeutig erkenn- und deutbar sein; in der Umsetzung und Nutzung müssen Risiken durch Fehlverhalten oder Fehlanwendung ausgeschlossen sein. Es werden Forschungsfragen aufgestellt, die in der Arbeit inhaltlich verfolgt werden. Die Beantwortung der Forschungsfragen erfolgt im Ergebnisteil in Abschnitt 6.3.4.

Wissenschaftsmethodik

Das Vorgehen der Arbeit richtet sich nach dem 4+1-Sichtenmodell nach Kruchten 1995. Die vier Sichten um die Anwendungsfälle und -szenarien in Abbildung 1.2 bestehen aus

- **Logische Sicht:** Nötige Funktionalität einzelner und verknüpfter Komponenten
- **Prozesssicht:** Verhalten der Einzelsysteme und des Gesamtsystems
- **Entwicklungssicht:** Integration und Verwaltung von Soft- und Hardware, für initiale Erstellung und spätere Änderung
- **Physische Sicht:** Topologie, Infrastruktur, Orchestrierung von Soft- und Hardware

Die vier Sichten gruppieren sich um die eigentliche Anwendung. Betrachtet wird die Logik, der Prozess, die Entwicklung und die Physis des Systems. In jeder Sicht wird ein in der Anwendung umzusetzendes Ziel betrachtet. Jede Sicht kann sowohl auf den Entstehungsprozess von Konzept und Umsetzung angewandt werden als auch rein auf das Ziel *Architekturframework* gerichtet sein. Zentral für das Vorgehen ist der Kern *Forschung* der

in Abbildung 1.3 am Ende des Kapitels vorgestellten Methodik. Mit der Arbeit werden nutzungsnahe Erkenntnisse angestrebt, weswegen die Rückführung der aufgestellten Theorien (*Konstruktion*) in die praktische Validierung (*Evaluation*) den Methodenschwerpunkt stellt.

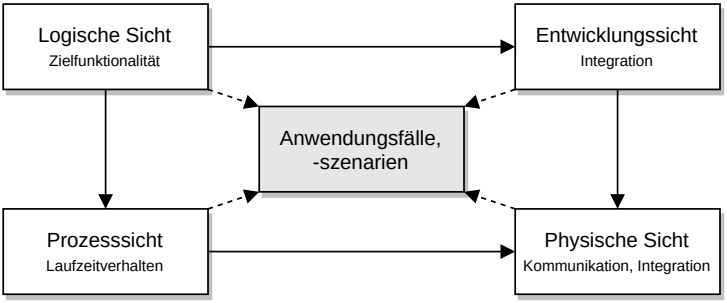


Abbildung 1.2: Vorgehen nach dem 4+1-Sichtenmodell nach Kruchten 1995

Forschungsfragen

Die Untersuchung geht von der Kernfragestellung aus, wie ein Architekturframework für die Automatisierung von Produktionsvorgängen beschaffen sein muss, um mittels dynamisch verknüpfter und koordinierter cyberphysischer Komponenten Produktionsaufgaben erfüllen zu können. Die generelle Fragestellung wird in fünf Komponenten präzisiert:

- **Definition:** Wie müssen cyberphysische und virtuelle Komponenten im Automatisierungsnetz grundsätzlich beschaffen sein? Wie können Fähigkeiten wie Self-X-Funktionen (Selbstorganisation, Selbstkonfiguration etc.) verhaltensdeterminierbar und regelgerecht integriert werden?

- **Struktur:** Wie sind Modellierung, Organisation und Verwaltung eines Automatisierungsnetzes zu gestalten und mit kollektiven Self-X-Funktionen (z.B. Fähigkeitsbeschreibung) zu vereinen?

- **Kommunikation:** Können Vermittelungs- und Beschreibungsfähigkeiten im Automatisierungsnetz semantikoffen und generisch integriert werden?

- **Hierarchie:** Wie ist ein verhaltensdeterminierbarer Aufbau und Ablauf im Automatisierungsnetz beschaffen? Von wem und wie werden Aufgaben und Abläufe abstrahiert, abgestimmt und verteilt?

- **Koordination:** Wie werden parallel stattfindende Vorgänge und Abläufe auf verschiedenen Systemen, insbesondere bei zeit- und materialkritischen Aufgaben, koordiniert?

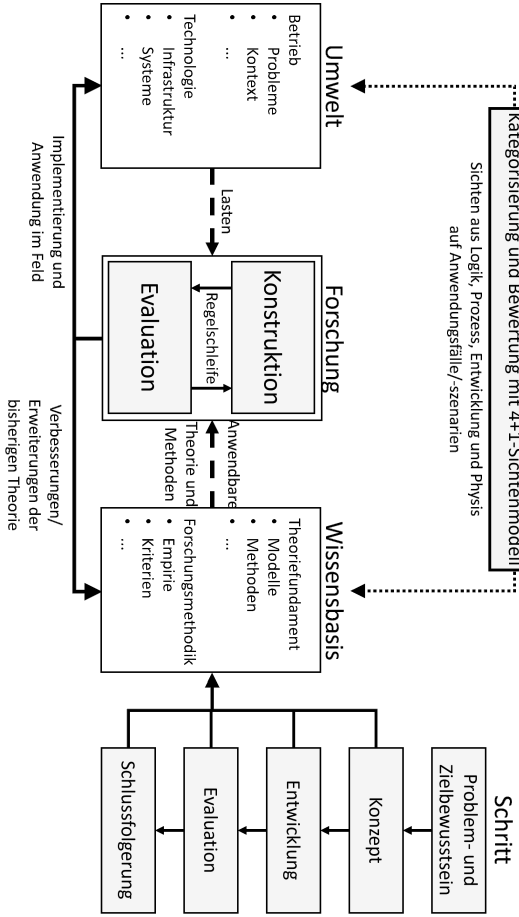


Abbildung 1.3: Das Modell der wissenschaftlichen Methodik der Arbeit

1.7 Cyberphysische Systeme

In der ersten Dekade des 21. Jahrhunderts prägte Helen Gill[2] den Begriff *Cyber-physical system* (Lee 2008; Lee & Seshia 2012), zu deutsch *Cyberphysisches System* (CPS). Die US-amerikanische Wissenschaft formte den Begriff in der darauffolgenden Zeit als Bezeichnung für Systemgruppen, in denen mehrere vernetzte digitale und virtuelle Systeme gemeinsam physische Prozesse steuern und regeln, für die in den beteiligten Systemen komplexe Rechenfähigkeit notwendig ist. Die grundlegende Beschreibung von CPS, ergänzt durch einen Erstellungsprozess von eingebetteten Systemen als CPS, erfolgte durch Lee & Seshia 2012. In der Abbildung 1.4 wird das CPS-Prinzip gezeigt: zwei Systeme *Plattform 1* und *Plattform 2* sind durch eine Netzwerkinfrastruktur (*Netzwerk*) fest integriert. Beide Systeme verfügen über Rechenfähigkeit (*Verarbeitung 1, 2, 3*) und Schnittstellen zur realen Welt *Eingabe 1, 2*, *Ausgabe 1*) und arbeiten mit ihnen gegen einen Prozess (*Physischer Prozess*), den sie gemeinsam steuern. (Lee & Seshia 2012) CPS sind ein *System-of-systems*-Ansatz (Hoffmann et al. 2021). Hervorzuheben ist das *National Institute of Standards and Technology*, das Standards für CPS in Arbeitsgruppen erarbeitet und die vorrangig allgemein gehaltenen Ergebnisse über öffentliche Berichte (siehe Griffor et al. 2017) verbreitet.

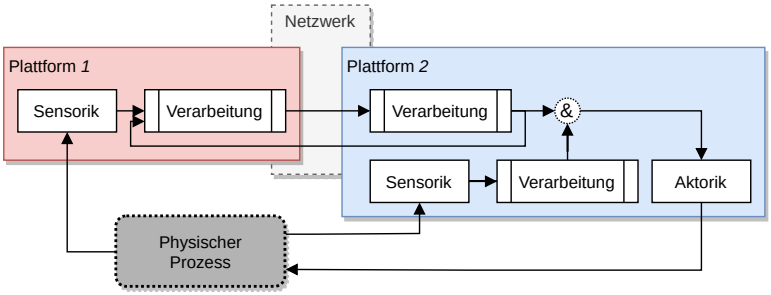


Abbildung 1.4: Beispiel für ein cyberphysisches System nach Lee & Seshia 2012

Der CPS-Begriff wird in verschiedenen Branchen genutzt, um Systeme zu bezeichnen, die über komplexe digitale und virtuelle Funktionen verfügen, umfassend netzwerk- und plattformintegriert sind und physische Prozesse erfassen, beeinflussen oder steuern (Hamzah et al. 2023). Die Softwarestruktur und Parametrierung cyberphysischer Systeme werden als

[2] siehe auch https://cps-vo.org/node/3966

leicht und schnell wandelbar verstanden, um die Abhängigkeit von festen Eigenschaften zu mindern und durch Softwaredefinition zu ersetzen. CPS begründen keine bestimmten neuen Technologien oder Geräte, sondern Initiativen zur Verbindung bisheriger und neuer Paradigmen, ihrer bekannten technischen Ansätze sowie punktuelle technische Weiterentwicklungen (Lee 2006; Lee 2008). Die Systeme, auf die der CPS-Ansatz zur Weiterentwicklung angewendet wird, variieren stark (Hamzah et al. 2023; Zhang et al. 2023), wodurch sich die CPS-Vielfalt auch auf die Entwicklung auswirkt (Mohamed et al. 2021). CPS erhalten ihre funktionale Prägung auf der Softwareebene. Durch die Verschiebung der Definitionslast von statischen physischen Komponenten zu wiederverwendbaren, generischen und schnell änderbaren Softwarekomponenten wird eine Steigerung und Vereinfachung von Adaption und Flexibilität erhofft. Es existiert keine normierte Begriffsdefinition. Als CPS verstandene Arbeiten nutzen teils deutlich vor dem Aufkommen des CPS-Begriffs eingeführte Softwaretechnologien (siehe beispielsweise Riedl et al. 2014). Monostori et al. 2016 stellt eine frühe Begriffsverortung auf Basis wissenschaftlicher Veröffentlichungen vor, die zeigt, wie CPS früh mit verschiedenen Kontexten verbunden werden. Erkennbar ist, dass die Forschung CPS als Grundlage für neue, außerordentlich befähigte Geräte sieht („Real time communication“, „Intelligent X“ et cetera), CPS als hochvernetzte Geräte betrachtet und mit dem Internet der Dinge verbindet sowie in der Anwendung besonders mit Produktion und Energieerzeugung untersucht. CPS werden als ein Haupttreiber von Industrie-4.0-Umsetzungen gesehen (Hamzah et al. 2023; Rikalovic et al. 2022). CPS-Entwicklungen sind modell-, analyse- und simulationsgetrieben (Lee 2008; Mohamed et al. 2021), wobei alle technischen Domänen eines technischen Systems Entwicklungsbedarf aufweisen, wenn CPS-Paradigmen wirken sollen (Lee 2006). Wesentliche Eigenschaften eines CPS sind die umfassende und dauernde Vernetzung der beteiligten Systeme sowie die Kooperation der Systeme in der Prozesssteuerung zur Erfüllung der Prozessaufgaben (Lee & Seshia 2012). Um ein technisches System als CPS zu entwickeln, sind diese Eigenschaften als Paradigmen während der Systemgestaltung einzubringen.

CPS in der Automatisierung

Die Einführung der CPS-Paradigmen in die Automatisierung werden vorrangig in der Forschung im deutschsprachigen Raum als Grundlage gesehen, um die Automatisierung

weiterentwickeln zu können (beginnend mit Monostori et al. 2016; Bangemann et al. 2016; Riedl et al. 2014; Herterich et al. 2015). CPS sollen die Implementierungsbasis für horizontale und vertikale Integration in der Produktion, die Integration umfassender Engineeringgrundlagen und die horizontale Integration von Unternehmen entlang der Wertschöpfungskette bereitstellen (Monostori et al. 2016).

Der VDI stellt mit VDI/VDE-Fachausschuss 7.20 2013 eine bedeutende Grundüberlegung für den DACH-Raum, in dem CPS-Paradigmen für die Industrieautomatisierung betrachtet werden. Ein wesentliches vorgestelltes Konzept liegt in der Nutzung von CPS zur Auflösung der statischen Automatisierungsstrukturen (Abbildung 1.5). Dieses Konzept ist ein Beispiel für die Transformation statischer hin zu dynamisch festgelegten und per Software definierten Eigenschaften. Automatisierungsgeräte werden ähnlich wie (virtuelle) Maschinen in der Informationstechnik nicht mehr zweckgebunden, sondern als Ausführungsplattformen für beliebig zuweisbare Aufgaben gesehen, deren Integration jederzeit geändert werden kann. VDI/VDE-Fachausschuss 7.20 2013 beschreibt den Einsatzzweck für CPS in der Produktion vorrangig in der Automatisierung. In anderen Abstraktionen werden CPS als erfassende, analysierende und handelnde physische Endpunkte virtueller Plattformen gesehen, die Fabriken, Unternehmen und Unternehmensketten integrieren (Pallasch et al. 2018a). CPS sollen durch die Verwendung von plattformgestützten Daten und Softwarekomponenten Produktionsaufgaben schneller, effizienter und flexibler realisieren, wobei sie sich jederzeit an neue und andersartige Aufgaben adaptieren können. (Zhang et al. 2023; Dafflon et al. 2021)

Die *knotenbasierte* Automatisierung mittels CPS wird in verschiedenen frühen Arbeiten als zukünftige Form der Industrieautomatisierung gesehen[3]. Hinter diesem Konzept steht die Einführung einer Abstraktionsschicht zwischen Automatisierungssystemen und Automatisierungsaufgaben, um die Automatisierung rekonfigurierbar gestalten zu können. Die Systeme werden in ihrer technischen Beschaffenheit und logischen Integration auf kleinstmögliche statische Anteile optimiert, um Fähigkeiten zur Adaption an Aufgaben erweitert und in dieser Form als *Knoten* gesehen, die zur Laufzeit ein Automatisierungsnetzwerk knüpfen.

[3]Unter anderem VDI/VDE-Fachausschuss 7.20 2013, Riedl et al. 2013, Riedl et al. 2014, Colombo et al. 2017, Gräßler et al. 2016, Monostori et al. 2016

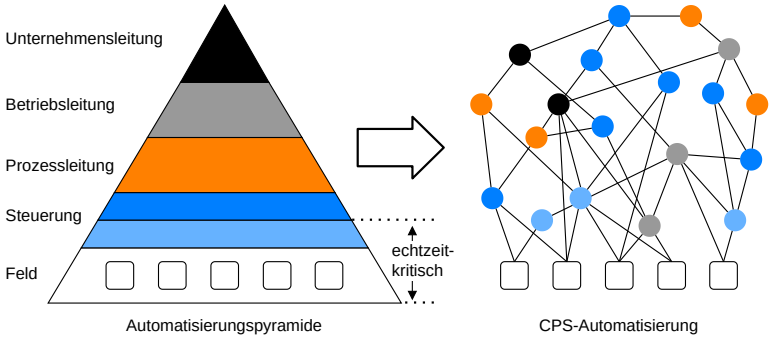


Abbildung 1.5: Auflösung der Automatisierungspyramide in ein CPS-Netz nach VDI/VDE-Fachausschuss 7.20 2013

Automatisierungsaufgaben in der Produktion und die ausführenden Automatisierungssysteme sind - bedingt durch die Bindung an physische Prozesse, spezielle Netzwerke und Hardware - nur bedingt voneinander trennbar. Die Konzeptentwickler der knotenbasierten Automatisierung gehen von einer Gesamtabstraktion des Automatisierungsprozesses von der Hardware aus, was unrealistisch ist. Deswegen muss ihr Konzept als *Idealziel* betrachtet werden. Der Begriff *Knoten* ist irreführend, da er aus der Netzwerkwelt stammt und beim Betrachter einen einseitigen Fokus auf die Netzwerkbildung erzeugt. Es wird zwar ein Netzwerk aufgespannt; die mit *Knoten* bezeichneten Stellen sind aber keine Verbindungsstellen, sondern Mittelpunkte des tatsächlichen Automatisierungsgeschehens: Sie sind die technischen Systeme, auf denen alle Automatisierungsdaten platziert sind und die Automatisierungslogik ausgeführt wird.

2 Grundlagen der Industrieautomatisierung

2.1 Verständnis

2.1.1 Aufgabe der Industrieautomatisierung

Ein automatisiertes System ist ein technisches System, das festgelegte Vorgänge ohne menschliches Zutun in einer bestimmten Weise durchführt und sich dabei selbst steuert. Die Automatisierung eines technischen Systems kann aus verschiedenen Gründen erfolgen, beispielsweise um Kosten durch den Entfall von menschlicher Arbeitskraft zu sparen oder um Arbeiten durchzuführen, die ein Mensch nicht erledigen kann. (Nof 2009)

Mit Automatisierung wird der Gesamtvorgang der Änderung eines manuellen technischen Systems hin zum automatisierten System bezeichnet. Automatisierung erzeugt zusätzliche Aufwände bei der Systemerstellung. Diese Aufwände müssen durch den Automatisierungsnutzen ausgeglichen werden. Die Aufwände sind abhängig von zahlreichen Faktoren, zu denen vor allem die Prozess- und Qualitätsanforderungen und die Ausgestaltung der Automatisierungstechnik und ihrer Funktionen gehören, sowie vom Automatisierungsgrad. Der Automatisierungsgrad beschreibt qualitativ das Automatisierungsausmaß an einem System und ob dieses teil- oder vollautomatisiert ist. (Nof 2009)

Für die Automatisierung eines Systems sind meist mehrere technische Ansätze möglich, wobei diese Ansätze wiederum Anteile aus mehreren technischen Domänen beinhalten können. Heutige Automatisierung ist elektronik- und softwaregetrieben und hat gegenüber vergangener starrer Automatisierung, die durch Elektrik und Mechanik fest vorgegebene Zusammenhänge aufwies, Vorteile in der Adaptierbarkeit des automatisierten Systems. Neben der reinen automatisierten Bearbeitung eines bestimmten Vorgangs, können softwaregetriebene Automatisierungssysteme den Leistungsumfang eines Systems durch ihre Adaptierbarkeit vergrößern.

2.1.2 Adaptierbarkeit der Automatisierung

Die Grundlage eines anpassbaren Gesamtprozesses ist die Adaptierbarkeit der Prozesssysteme an die Prozessschritte. Adaptierbarkeit kann mit unterschiedlichen Ansätzen integriert werden. Nach ElMaraghy & Wiendahl 2009 lassen sich zwei Ansätze aufgreifen:

- **Flexibilität:** Adaptierbarkeit über initial vorhandene Eigenschaften
- **Rekonfigurierbarkeit:** Adaptierbarkeit über Änderung von Komponenten

Flexibilität beruht damit auf den immanenten Fähigkeiten eines Systems, sich an geänderte Anforderungen anzupassen, während im Rekonfigurationsansatz das bestehende System verändert oder erweitert wird. Flexibilität kann auf das Produkt, auf Prozessausführung oder Stückzahlen bezogen werden. (Chryssolouris 2006, ElMaraghy & Wiendahl 2009) Für automatisierte Systeme sind zwei Systemflexibilitäten relevant (Nof 2009):

- **Applikationsflexibilität:** Flexibilität bei der Prozessausführung
- **Adaptionsflexibilität:** Flexibilität bei Status- bzw. Prozessänderungen

Die umgesetzte Flexibilität eines Systems wird an zwei Eigenschaften bewertbar (Nof 2009):

- **Anzahl der Status**, die automatisch angenommen werden können
- **Kosten und Dauer für** eine **Systemantwort und** eine **Statusänderung**

Die Adaptierbarkeit eines Produktionssystems wird von der Gesamtstruktur bestimmt, ausgehend von der physischen Beschaffenheit bis zur Software der Organisationsebene. Die technischen Systeme der Organisationsebene sind allen Produktionssystemen übergeordnet, wodurch sie für die spezifische Betrachtung eines Produktionssystems entfallen. In der spezifischen Betrachtung wiederum kann die Adaptierbarkeit in einen Physis- und einen Automatisierungsteil unterteilt werden. Die Adaptierbarkeit des Automatisierungsteils wird durch die Gesamtheit seiner Software, der Hardware, der Integration und der Aktorik und Sensorik bestimmt. Alle Systeme dieser Gruppen besitzen eingeprägte Adaptionsfähigkeiten. Die Software eines Produktionssystems besitzt, verglichen mit den anderen technischen Teilen, die größte inhärente Gestaltungsfreiheit für die Adaptierbarkeit des Produktionssystems, wird aber durch die Fähigkeiten von Aktorik, Sensorik, Integration und Hardware begrenzt.

2.1.3 Determinismus und Echtzeitfähigkeit

Automatisierungssysteme müssen ein vorhersehbares Laufzeitverhalten besitzen, die Prozesse zu jedem Zeitpunkt beherrschen, sowie auf Prozessänderungen hinreichend schnell reagieren können. So ist gewährleistet, dass die automatisierten physischen Prozesse anforderungsgerecht durchgeführt werden. Automatisierungssysteme müssen deterministisch und echtzeitfähig im Prozesssinne sein. Determinismus ist ein inhärentes Merkmal echtzeitfähiger Systeme (Benra & Halang 2009), er wird hier gesondert genannt.

Hält ein System eine festgelegte Anforderung für die Dauer zwischen einem relevanten Ereignis und der eigenen Reaktion ein, kann es als Echtzeitsystem bezeichnet werden. (Benra & Halang 2009) Da Automatisierungssysteme physische Prozesse bearbeiten und diese Prozesse in zeitlichen Zusammenhängen erfüllt werden müssen, müssen die Automatisierungssysteme als Echtzeitsysteme konzipiert werden. Die Größe der Zeitanforderungen wird je nach Einsatzzweck unterschieden; eine bewegungsregelnde Komponente muss eine andere Zeitanforderung erfüllen als eine Verwaltungsanwendung. Das Kriterium für die Bewertung der Echtzeitfähigkeit ist nicht die Verarbeitungsgeschwindigkeit, sondern die Zuverlässigkeit, in einem festgelegten Zeitraum auf ein Ereignis zu reagieren. (Benra & Halang 2009)

Die Anforderungen an die Funktionen eines prozesssteuernden Echtzeitsystems werden in zwei Gruppen aufgeteilt (Benra & Halang 2009):

- **Rechtzeitigkeit** ist die Einhaltung einer maximalen Zeitdauer zwischen Ereignis und Reaktion, um die Zeitanforderung bei der Prozesssteuerung zu erfüllen. Mit der Rechtzeitigkeit werden Prozesseigenschaften in der geforderten Qualität realisierbar.

- **Gleichzeitigkeit** bedeutet für das Echtzeitsystem die nebenläufige Bearbeitung von Aufgaben zur parallelen Bearbeitung von Ein- und Ausgangssignalen, Prozessverarbeitung und systeminterner Verwaltung.

Die Unterteilung der Anforderungsgüte erfolgt in (Benra & Halang 2009):

- **Harte Echtzeit** ist gefordert, wenn das Gesamtsystem auf die unbedingte Einhaltung der Zeitanforderung angewiesen ist, weil bei Überschreitung der zulässigen Antwortzeit der Prozess für Produktionssystem, Produkt oder Mensch schädigend

oder gefährdend abläuft. Eine Unterart der harten Echtzeit ist die feste Echtzeit, bei der die Antwortzeit auf einen Wert festgelegt ist, der zusätzlich nicht unterschritten werden darf. Harte Echtzeit ist beispielsweise bei der parallelen Regelung von Einzelbewegungen oder Sicherheitskreisen nötig.

- **Weiche Echtzeit** ist nötig, wenn der korrekte Ablauf des Gesamtprozesses zwar die Einhaltung einer Zeitanforderung erfordert, es bei Nichteinhaltung aber nicht zu Schäden oder Gefährdung, sondern zu Zeit- oder Geldverlusten kommt. Ein Beispiel für eine weiche Echtzeitanforderung ist die Durchführung von Bestellungen durch eine Anwendung zur automatisierten Organisation der Unternehmensressourcen.

Um echtzeitfähige Systeme zu realisieren, sind die Einhaltung von bestimmten Gestaltanforderungen und die Nutzung passender Technologien auf Hard- und Softwarebene vonnöten; auf beiden Ebenen muss verhindert werden, dass nicht-prozessrelevante oder außerordentliche Vorgänge das System blockieren oder in Ausnahme- oder Fehlerzustände versetzen. Die Realisierung wird insbesondere dann anspruchsvoll, wenn das System aus Komponenten oder einer änderbaren Zusammenstellung bestehen soll. In diesem Fall müssen die Komponenten zusätzlich zur Zeitanforderung ihres jeweiligen Prozessteils die Zeitanforderungen des Gesamtprozesses erfüllen und sicher gegenüber Wechselwirkungen sein. Keine Komponente darf eine andere Komponente in ihrem Zeitverhalten negativ beeinflussen. (Boelmann et al. 2016) Durch die Aggregation und Wechselwirkungen der Komponentenzustände erhält das Gesamtsystem eine Menge an zu beherrschenden Systemzuständen, die dem Entwickler und dem Betreiber bekannt sein müssen.

Der Aufbau eines echtzeitfähigen Systems als verteiltes System vergrößert die Schwierigkeit, eine ausreichende Echtzeitfähigkeit zu gewährleisten. Komponenten eines echtzeitfähigen Systems sind nicht nur Geräte oder Anwendungen, sondern auch die verbindenden Teile: Ist das System ein verteiltes, also aus mehreren rechenfähigen Einzelsystemen aufgebaut, die überlappend einen Prozess steuern, wachsen die Echtzeitanforderungen über die Grenzen der Einzelsysteme. In verteilten Systemen existieren Kommunikationsverbindungen oder Netzwerke zur Verbindung der Systeme. Die Kommunikations- und Netzwerktechnologien müssen im verteilten System als echtzeitrelevante Komponenten mitbetrachtet werden und selbst Zeitanforderungen erfüllen. Beachtete Zeitgrößen sind:

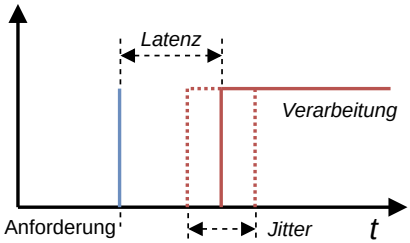


Abbildung 2.1: Prinzipdarstellung von Latenz und Jitter (Rauschen) über der Zeit bei Signal und zugehöriger Signalverarbeitung (National Institute of Standards & Technology 2002)

- **Reaktionszeit:** Zeit, die von einem auftretenden Ereignis bis zur ausgehenden Reaktion des Automatisierungssystems vergeht
- **Latenz:** Zeit, die zwischen auftretendem Ereignis und dem Beginn der Verarbeitung vergeht (siehe Abbildung 2.1)
- **Jitter:** Variationsbreite, innerhalb derer die Latenz schwankt

Die Einhaltung der Zeitgrößen kann theoretisch bewiesen oder statistisch untersucht werden (Kweon et al. 1999). In der statistischen Untersuchung werden Methoden der deskriptiven Statistik und daraus abgeleitete Leistungsindikatoren verwendet. Im Bereich der Sensorik und Aktorik liegen die Reaktionsanforderungen um eine Millisekunde, die Jitteranforderungen bis in den Bereich einer Mikrosekunde (VDI 3687; IEEE Std 61158). Deterministische Kommunikationstechnologien können anhand ihrer Zykluszeit eingeordnet werden. Gundall et al. 2020a; Wollschlaeger et al. 2017 unterteilen sie in drei Klassen A, B und C mit zugehörigen Zykluszeiten jeweils kleiner 100, 10, 1 ms.

2.1.4 Umsetzungsvorgehen

Im industriellen Maschinen- und Anlagenbau sind domänenweise aufgeteilte Sequenzen im Erstellungsprozess üblich (Eckert 2015). In der Sequenz ist die Automatisierungsdomäne (die Einrichtung von Automatisierungskomponenten, ihre Konfiguration, Vernetzung und Programmierung) üblicherweise die letzte der beteiligten Domänen. Nach ihr erfolgen die Inbetriebnahme- und Prüfungsschritte für das Gesamtsystem. In den Domänen arbeiten

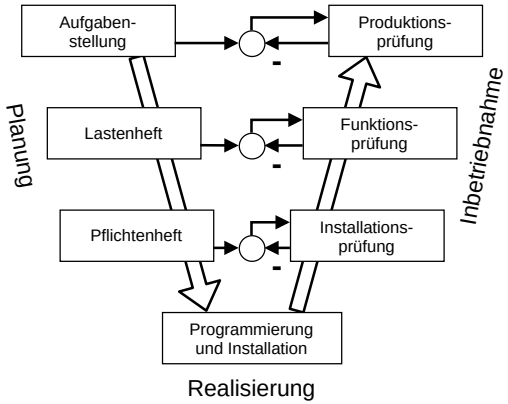


Abbildung 2.2: Vorgehen bei der Erstellung eines automatisierten Systems in der Industrieautomatisierung nach dem V-Modell (nach Seitz 2015)

meist domänenspezifische Experten. Insbesondere Anlagen sind zudem häufig einzigartige Systeme, die auftragsspezifisch entstehen, was Generalisierungen erschwert. (Wiesmayr et al. 2021; Eckert 2015) Für die Erstellung werden domäneneigene Softwarewerkzeuge verwendet, die zum Teil frei verwendbar sind und zum Teil durch in der Anlagenstruktur festgelegte Hard- und Softwarekomponenten vorgegeben werden. In der Automatisierungssoftwareentwicklung sind manuelle, aufwändige Methoden für die Generalisierung und Wiederverwendung von Implementierungen üblich. (Fischer et al. 2020)

Erstellungsprozesse eines automatisierten technischen Systems in der Industriepraxis werden mit sequentiellen Modellen abstrahiert. Bekannt ist beispielsweise das V-Modell (siehe Abbildung 2.2). Die Modelle sind durch eine Festlegung einer Abfolge bestimmter Schritte und Rückführungspunkte gekennzeichnet. Schritte nach einem solchen Modell werden nicht einzeln durchgeführt. Nach Überführung in den Produktivbetrieb wird ein automatisiertes System als fertiggestellt betrachtet. Modelle wie das V-Modell wirken zur Entwicklungs- und Validierungszeit, nicht zur Laufzeit. (Seitz 2015; Eckert 2015)

2.2 Abstraktionsmodelle und ihre technischen Systeme

Die Standardabstraktion der industriellen Automatisierung ist die Automatisierungspyramide, ausgehend von der Definition der ISA-95. Es existieren verschiedene Variationen

der Pyramide, die aber ihre wesentlichen Eigenschaften teilen: Die Einzelsysteme einer Gesamtautomatisierung werden in mehreren hierarchisch organisierten, aufeinander aufbauenden Ebenen eingeordnet, die vom unmittelbaren physischen Produktionsprozess hoch zur strategischen Unternehmensleitung gehen. Üblich sind drei bis fünf Ebenen. Die Darstellung der Pyramide in Abbildung 2.3 vereinfacht die Definition auf drei Ebenen. Von der Feldebene des Pyramidenfundaments aus hin zur Spitze reduziert sich die Bedeutung der ausführenden Hardwareplattform für das jeweilige System. Im Verständnis der Automatisierungspyramide kommunizieren Systeme aus einer Ebene ausschließlich mit Systemen aus der oder den angrenzenden Ebenen. Steuerungs- und Feldebenen oder Prozessleit- und Steuerungsebenen werden bei einigen Definitionen jeweils zusammengefasst. Die internationale Weiterentwicklung der US-amerikanischen ISA-95 ist die IEC 62264. Die Normierung nach der IEC 62264 abstrahiert die Automatisierungspyramide zu fünf Hierarchieebenen, um den Eindruck der gegenseitigen Abhängigkeit zu reduzieren, da die Systeme innerhalb der Ebenen üblicherweise autark funktionieren können. Die Hierarchieebenen sind Sammlungen ebenenspezifischer Aktivitäten. Für die Ebenen werden ungefähre Zeitdauern für die Aktivitäten angegeben. Für alle Ebenen existieren Softwareanwendungen und Gerätetypen, die als ebenentypisch angesehen werden. In den Organisationsebenen (meist getrennt in Betriebs- und Unternehmensleitebene) sind ERP-Systeme und MES (siehe Abschnitt 2.2.1) bestimmend, die als reine Softwareanwendungen von keiner besonderen Hardware oder Technologie abhängig sind. In den Steuerungs- und Feldebenen ist die Hardware ein bestimmendes Merkmal, sowohl systemintern hinsichtlich der Beschaffenheit der ausführenden Plattform als auch in der Systemvernetzung hinsichtlich der Kommunikationstechnologien. Dabei ist diese Hardware auf spezifische, meistens herstellerdefinierte und -gebundene Software angewiesen.

2.2.1 Organisationsebene

In der Organisationsebene arbeiten zwei Kernsysteme: *Enterprise-Ressource-Planning-* (ERP) und *Manufacturing-Execution-Systeme* (MES). ERP sind Softwaresysteme zur Digitalisierung und Automatisierung von organisatorischen und kaufmännischen Aufgaben in Unternehmen. ERP realisieren Prozesse, die nicht unmittelbar mit der Erzeugung eines Produkts ver-

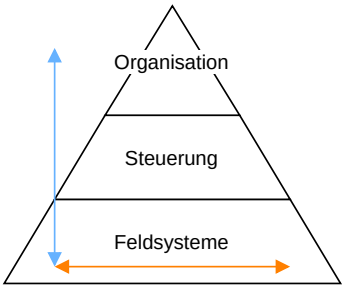


Abbildung 2.3: Kernebenen der Automatisierungspyramide, reduziert aus der ISA-95 und IEC 62264 mit horizontaler (orange) und vertikaler Integration (blau)

bunden sind, beispielsweise zur Kosten- oder Personalplanung. Die industrielle Anwendung von ERP ist Standard. Einführungsaufwände in Unternehmen sind sehr groß, weswegen ERP selten geändert werden. (Aggarwal et al. 2021; Prasetyo & Soliman 2021) Weitere Systeme (z.B. Fertigungssteuerung, Qualitätsmanagement) können gesondert vorhanden sein, wenn sie nicht Teil des MES sind (Subramanian et al. 2019).

In Abstraktionsmodellen sind ERP mit Fertigungssystemen nicht verbunden. Um Fertigungsaufträge anzuleiten, werden diese von der ERP-Ebene über MES verteilt. In der industriellen Praxis kommt es zu Überschneidungen zwischen ERP und MES. Teilweise werden Fertigungssysteme an ERP angebunden, teilweise besteht keine automatisierte Verbindung und der Informationsaustausch wird durch einen Bediener über Mensch-Maschine-Schnittstellen vorgenommen. Die Basis eines ERP ist eine Datenbankenmenge, um die lokale oder internetbasierte Softwarekomponenten zur Implementierung von Geschäftslogik und Schnittstellen angeordnet sind. (Prasetyo & Soliman 2021)

MES sind Softwaresysteme zur automatisierten Anweisung, Ausführung, Überwachung und Abnahme von Fertigungsvorgängen auf Fabrikebene. MES agieren auf Auftragsebene (VDI 5600-1) und sind in der Organisations-, teilweise in der Steuerungsebene der Automatisierungspyramide zu sehen. Sie sind in der VDI 5600 relevant standardisiert. MES-Funktionen sind beispielsweise (VDI 5600-1; Arica & Powell 2017):

- Planung und Zuweisung von Arbeitsaufträgen
- Betriebsmittelüberwachung und Instandhaltungsplanung

- Leistungsanalyse von Produktionssystemen

MES konzentrieren sich auf die Produktionsorganisation innerhalb eines Produktionsstandortes, wobei die Aufgaben aus der Datensammlung, der Unterstützung bei der Entscheidungsfindung und der Automatisierung von organisatorischen oder dokumentierenden Vorgängen bestehen. Ein MES erhält Informationen von allen relevanten Geräten, Programmen und Personen. Es ist nicht an eine bestimmte Kategorie von Betriebsmitteln gebunden. (VDI 5600-1; Arica & Powell 2017) Es existieren Ansätze für fabrik-und unternehmensübergreifende MES und MES-Teile (Helo et al. 2014; VDI 5600-7). Aus Fertigungssystemsicht verteilt ein MES Aufträge und überwacht deren Ausführung sowie das Fertigungssystem. MES-Zeitanforderungen bewegen sich im Sekunden- bis Schichtenbereich. Ihr Kommunikationsinhalt ist auftragsbezogen und findet änderungs- oder abfragebezogen statt. Die nach der VDI 5600-3; DIN 62264-5 standardisierte MES-Logikschnittstelle zu Fertigungssystemen basiert auf Datenpunkten mit OPC-Datentypen, ist hierarchisch und auf IEC-61131-3-Variablen kartierbar.

MES ragen in die Steuerungsebene hinein. Da MES stark geschlossene Systeme sind, richtet sich die Kommunikationsimplementierung der Steuerungsebene zur Organisationsebene nach den Schnittstellenbeschaffenheiten der Organisationsebene oder es müssen Kommunikationsadapter verwendet werden. Ähnlich den ERP ist die technische Basis von MES eine Datenabbildung auf Basis einer Menge von Datenbanken. (Li et al. 2020) In der technischen Struktur unterscheiden sich MES von ERP vorrangig durch zusätzliche Schnittstellen zur Steuerungstechnik. Für die Interaktion sind Normfunktionen beschrieben, die auf ereignisorientierte HTTP/REST-Schnittstellen kartiert werden können (DIN 62264-5).

2.2.2 Steuerungsebene

Die Steuerungsebene und ihre technischen Systeme sind im Verständnis auf zwei Ebenenstufen abstrahierbar. Die obere Ebenenstufe ist die Leittechnik, mit der eine oder mehrere Maschinen und Anlagen gesteuert und überwacht werden und der die tatsächliche Steuerungstechnik zur Detailausführung untergeordnet ist. In einer tatsächlichen automatisierten Umgebung müssen die beiden Stufen nicht getrennt voneinander existieren, wobei ins-

besondere bei kleineren Anwendungen oder geschlossenen, einfachen Maschinen ohne Anlageneinbettung die Leittechnik weitestgehend entfallen kann.

Leittechnik

Mit der (Prozess)Leittechnik werden Produktionsprozesse angeleitet und überwacht. Zum Einsatz kommt sie insbesondere bei kontinuierlicher Produktion und bei größeren Anlagen. Die Leittechnik beschäftigt sich nicht mit der unmittelbaren Prozessregelung. Auf der Leitebene definiert sind zum Beispiel *Supervisory-Control-and-Data-Acquisition*-Systeme (SCADA). Ein übliches SCADA-System besteht aus einer Mensch-Maschine-Schnittstelle (Human-Machine-Interface, HMI), rechnergestützte Datensammlung und Überwachung, Fernbedienungsterminals (Remote Terminal Units, RTU) oder universelle Steuerungssysteme (Speicherprogrammierbare Steuerungen, SPS) zur Prozesserfassung, -regelung und -steuerung, sowie einer entsprechenden Netzwerkinfrastruktur zur Verbindung aller Systeme. (Bangemann et al. 2014) Leittechnik betreut im Regelfall steuernde Geräte, Aktorik und Sensorik eines geschlossenen Anlagenteils oder eine gesamte Anlage. Leit- und Steuerungstechnik können sich überschneiden. Leittechnik-Systeme sind anlagenspezifisch.

Steuerungstechnik

Aus Produktionsprozesssicht führt die Steuerungstechnik die prozessspezifische Automatisierungslogik aus. Sie erfasst über Sensorik die Zustände eines physischen Prozesses und steuert ihn über Aktorik entsprechend der programmierten Automatisierungslogik zur automatisierten Prozessdurchführung. Zur Umsetzung werden die unprogrammiert ausgelieferten Steuerungssysteme maschinen- oder anlagenspezifisch programmiert und integriert. In der Steuerungstechnik sind IEC-61131-Systeme der Standard (siehe Abschnitt 2.3.1). Teilweise werden Steuerungen oder Industrie-PC, die mit Mittel- und Hochsprachen programmierbar sind, verwendet. Weitere normierte Systeme sind nach der DIN 66253 oder DIN EN 61499-1 gestaltet, werden allerdings nur wenig genutzt. (Seitz 2021; Seitz 2015; Bangemann et al. 2014) Die Steuerungstechnik besitzt programmierte oder konfigurierte Schnittstellen zu LT oder zur Organisationsebene. Für einen Prozess können mehrere Steuerungssysteme zuständig sein, die getrennt oder redundant arbeiten. (Seitz 2021; Seitz 2015)

2.2.3 Feldebene

Auf der Feldebene, teilweise mit eigenem Steuerungsumfang, befindet sich mit der Sensorik und Aktorik die Schnittstellenperipherie der Automatisierungssysteme in Richtung der physischen Produktionseinrichtungen. Sensorik und Aktorik sind der Ein- und Ausgabeteil, mit dem die Steuerungssysteme physische Prozesse erfassen und antreiben. Bei beiden Peripherietypen ist der Raum möglicher Einsatzzwecke, Fähigkeitsumfänge und Komplexitätsgrade groß. Bei beiden Peripherietypen werden neben digitalen Systemen auch analoge über eigenständige oder in die Steuerungstechnik integrierte Wandler verwendet. Digitale Feldsysteme sind in der modernen industriellen Praxis der Standard, besonders in der Aktorik, da bei Bewegungssystemen viele Parameter anfallen. Die Feldebene ist fest verwoben mit der Steuerungstechnik, außer in Ausnahmefällen, in denen Aktorik und Sensorik über eigene Steuerungs- und Kommunikationstechnik verfügt und direkt mit der Leittechnik zusammenarbeiten kann.

2.2.4 Ebenenintegration

Unterschieden werden in der Abstraktion vertikale und horizontale Integration, wie in der Abbildung 2.3 eingezeichnet. Die vertikale Integration bezeichnet die technische und logische Verknüpfung von Systemen aus verschiedenen Ebenen. Sie soll möglichst durchgängig und umfassend sein. Mit horizontaler Integration wird die technische und logische Verknüpfung von Systemen innerhalb einer Ebene bezeichnet. Die vertikale Integration berücksichtigt hierarchische und leitende Kommunikation um Vorgänge anzuleiten, während die horizontale Integration die Verknüpfung von Systemen zur Durchführung von ebeneninternen Vorgängen realisiert. (Nof 2009)

Vertikale Integration

In der hierarchischen Automatisierung variieren Systeme und Technologien auf allen Ebenen, insbesondere in den unteren Ebenen, deren Teilsysteme kleiner und zahlreicher sind als auf den oberen. Hinzu kommt, dass in allen Ebenen kontinuierlich Änderungen stattfinden, die in Zeitpunkten, Ansätzen und Vorgehensweisen nicht abgestimmt sind oder

sein müssen. Durch die resultierende Variation entstehen Hürden bei der Integration der Systeme. Betroffen ist besonders die vertikale Integration, da bei dieser Systeme aus unterschiedlichen Ebenen verbunden werden müssen, die entsprechend größere technologische Unterschiede aufweisen als Systeme aus einer gemeinsamen Ebene. Diese Hürden stehen der Durchgängigkeit und der Konsistenz des Datenaustauschs und der automatisierten Abläufe entgegen. (Tabim et al. 2021)

Die Hürden, die durch die verschiedenen Umsetzungs- und Anwendungsansätze der beteiligten Systeme entstehen, sind technologischer und logischer Natur. Die technologischen Hürden entstehen durch fehlende technische Verbindungen und Unterschieden in den Kommunikationsprotokollen sowie technischen Kommunikationsformaten. Die logischen Hürden ergeben sich durch Systemparadigmen (Kommunikationsgründe, Kommunikationsmuster) und die Beschaffenheit von Nachrichteninhalten. (Tabim et al. 2021)

Die vertikale Integration innerhalb der Organisationsebene und zwischen Organisations- und Steuerungsebenen setzt auf allgemeine Vernetzung über drahtbasierte und drahtlose Standardnetzwerke auf. Entsprechend basieren die genutzten Protokolle auf Ethernet oder IEEE 802.11 und den dafür verfügbaren Standardtechnologien wie HTTP/REST oder TCP. Zwischen Steuerungs- und Feldebenen sind Standardtechnologien oberhalb der physischen Schicht aufgrund der Determinismusanforderungen wenig vertreten. Es werden vorrangig Feldbusverbindungen genutzt, die über Konfiguration oder Programmierung bei den Steuerungs- und Feldsystemen angelegt werden. (Bangemann et al. 2014)

Horizontale Integration

Die Gestaltung der horizontalen Integration variiert mit den Ebenen. Auf den physisunabhängigen oberen Ebenen werden Netzwerkstandardtechnologien eingesetzt, über die standardisierte oder software- und herstellerspezifische Schnittstellen arbeiten. Die Umsetzung der horizontalen Integration der Industrieautomatisierung ist in der Organisationsebene reinen IT-Systemen gleich und baut auf Standardtechnologien wie TCP oder HTTP auf, bestimmt durch die verwendeten ERP, MES und weiteren Systeme.

Die horizontale Integration der Steuerungs- und Feldtechnologien besitzt besondere technische Ansätze aufgrund der speziellen Technologien. Sie stützt sich bei allen relevanten Ansätzen auf Drahtverbindungen und überschneidet sich mit der vertikalen Integration

zwischen Steuerungstechnik und Feldkomponenten. Ansätze für industrieorientierte Drahtloskommunikation (siehe beispielsweise WIA-FA nach DIN 62948) sind nicht etabliert. Zugrundeliegend sind in der Feld- und Steuerungsebene drei Vernetzungsarten:

- **Hartverdrahtung:** Hartverdrahtete Komponenten sind fest installiert und können sich üblicherweise nicht digital repräsentieren oder verbinden. In die Automatisierung werden sie über EA-Schnittstellen anderer Geräte elektrisch eingehängt und angesteuert. Hartverdrahtete Integration ist technisch einfach und günstig. Ihre Rekonfiguration erfordert physische Eingriffe und Arbeit am Integrationsgerät. Hartverdrahtete Systeme können Determinismusanforderungen erfüllen. Beispiele für betroffene Systeme sind Sensorik wie Induktiv- oder Kapazitivgeber.

- **Feldbusintegration:** Feldbusintegrierte Systeme sind über ein (Feld)Bussystem angebunden. Der zugehörige Feldbus kann exklusiv an einem übergeordneten Gerät sein oder ein Netzwerk mit mehreren gleich eingeordneten Geräten aufspannen. Je nach Feldbustechnologie können verschiedene Topologien (Linie, Stern, Ring) unterstützt werden. Einige Feldbustechnologien bauen auf Ethernet auf (beispielsweise Profinet, Ethercat), wodurch parallel Standardprotokolle verwendet können. Feldbusse unterstützen deterministische Kommunikation und sind der Standard für komplexe Sensorik und Aktorik wie beispielsweise Servoantriebe, und können funktional sichere Kommunikation umsetzen. Feldbusprofile sind über die IEC 61158 standardisiert.

- **Standardnetzwerkintegration:** Automatisierungskomponenten können über nicht-deterministische Standardprotokolle über Ethernet- und Wifi-Standardnetzwerke integriert werden. Verwendet werden beispielsweise MQTT, TCP oder OPC UA über TCP. Diese Integration kommt bei nicht-kritischer Kommunikation zum Einsatz, die zudem oft parallel vertikal und horizontal integriert.

2.3 Technologien der Feld- und Steuerungsebenen

Genannt werden die spezifischen Technologien der Feld- und unteren Steuerungsebenen in der Industrieautomatisierung, die nicht zu IT-Standardtechnologien gezählt werden können. Dazu gehören zunächst die Steuerungssysteme der IEC 61131, die den Quasistandard bilden,

sowie die Steuerungssysteme der DIN 66253 für spezielle Echtzeit- und Regelmechanismen, die Feldbus-Kommunikationstechnologien für die automatisierungsfähige Kommunikation´ sowie die Beschreibungs- und Modellierungssysteme für die Entwicklung.

2.3.1 Steuerungen nach der IEC 61131

Steuerungen, die nach der IEC-61131-Norm programmiert und betrieben werden, sind der etablierte Standard bei speicherprogrammierbaren Steuerungen für Maschinen und Anlagen. Die Programmiersprachen der IEC 61131 wurden mit dem Ziel der Virtualisierung der zuvor verwendeten hartverdrahteten Logikschaltungen entworfen. Sie sind darauf ausgerichtet, logische Zusammenhänge zwischen Ein- und Ausgaben herzustellen. IEC-61131-Steuerungen stellen den Feldstandard bei Maschinen- und Anlagen. (Seitz 2021)

Steuerungshersteller unterstützen die IEC 61131 mindestens im Grundumfang, oft proprietär abgewandelt oder erweitert. Die Programmiersprachen der IEC 61131-3 sind auf Logikabläufe und Ein-/Ausgabeoperationen ausgerichtet. Die geschriebenen Programme werden gebaut und über einen Schreibvorgang der Steuerungsumgebung fest zugewiesen, die diese Programme nach dem Starten kontinuierlich ausführt. Änderungen der Programme oder Struktur erfordern einen Bauvorgang und ein neues Schreiben auf die Steuerung. Programme sind in der IEC 61131 als geschlossene Entität ausgelegt, die auf einem abgeschlossenen System platziert und ausgeführt wird. (DIN 61131-1) Steuerungssysteme nach der IEC 61131 sind für den Einsatz in IIoT-Umgebungen untersucht und oft proprietär erweitert (Seitz 2021), wobei die Erweiterung mit modernen netzwerkorientierten Kommunikationsprotokollen wie OPC UA und MQTT sowie entsprechender Adaptionsfähigkeit an Programme und Daten des Systems die Haupterweiterung des Funktionsumfangs ist.

Modelle

Die IEC 61131 beschreibt drei Kernmodelle (Seitz 2015; DIN 61131-3):

- **Softwaremodell:** Die Zusammenstellung eines IEC-61131-Systems aus den verwendeten Ressourcen (Steuerungen, EA, Module) und den darauf platzieren Tasks (Task: $1..n$ Programme(n), Funktionen, Funktionsbausteine) und Variablen.

- **Kommunikationsmodell:** Der Austausch von Daten bzw. die Übergabe von Werten zwischen Programmen sowie Programmteilen (Funktionen, Funktionsbausteine) über globale Variablen, lokale Variablen in Programmen und Funktionsbausteinen und Funktionsparameter.

- **Programmiermodell:** Umfasst die Grundlagen für die Programmierung der spezifischen Nutzerprogramme, die auf einem IEC-61131-System ausgeführt werden sollen. Es umfasst Programmiersprachen, die Erstellung eigener Programme und Strukturen für Nutzerdatentypen, -funktionen und -bausteinen.

Daten- und Programmstruktur werden in IEC-61131-Systemen programmiert. Je nach Hersteller können Datenstrukturen auch konfiguriert werden.

IEC 61131-3: Programmiersprachen

In der IEC 61131-3 sind fünf Programmiersprachen definiert: Anweisungsliste (AWL, textuell), Strukturierter Text (ST, textuell), Ablaufsprache (AS, graphisch), Funktionsbausteinsprache (FBS, graphisch) und Kontaktplan (KOP, graphisch). Je Funktion oder Funktionsbaustein kann eine Sprache verwendet werden. Dadurch können in einem Programm insgesamt mehrere Programmiersprachen verwendet werden. In der Praxis werden anwendungsspezifische Abfolgen oft über graphische Sprachen modelliert, während Funktionen und Funktionsbausteine für generische und komplexe Funktionen textuell erstellt werden. Es gibt herstellerspezifische Erweiterungen, die einzelne Sprachen der IEC 61131 zu objektorientierten Sprachen aufwerten. (DIN 61131-1; DIN 61131-3)

Anwendersoftware auf IEC-61131-Systemen

Programmiert werden IEC-61131-Steuerungen nach Seitz 2015 als Ablauf- oder Verknüpfungssteuerung. Bei der Ablaufsteuerung ist die Programmlogik in Schritten organisiert, wobei das aktuelle Programm in einem Schritt gehalten wird, bis Bedingungen zum Übergang in einen anderen erfüllt sind. In der Verknüpfungssteuerung wird über eine Schleife das gesamte Programm durchgeführt, in dem alle Systemzustände abgebildet sind.

IEC-61131-Programme, die physische Prozesse steuern und bei denen Prozess- und Physiszustände wichtig sind, werden als Verknüpfungssteuerung in Form von Moore-Automaten,

Abbildung 2.4: Endlicher deterministischer Automat nach Moore

also deterministischen endlichen Automaten erstellt. Der Moore-Automat ist in Abbildung 2.4 skizziert. Das Eingangsnetzwerk erfasst das Eingangsbild $\underline{u}(k)$ und den gespeicherten Zustand $\underline{z}(k)$, woraus der neue Zustand $\underline{z}(k+1)$ entsteht, der im Zustandsspeicher abgelegt wird. Als neuer Istzustand wird $\underline{z}(k)$ an das Ausgangsnetzwerk gegeben, das dann nach relevanter Logik das Ausgangsbild $\underline{v}(k)$ ausgibt. Der Zustand $\underline{z}(k)$ wird für die nächste Zustandsbildung zurückgeführt. Eine Programmlogik befindet sich jederzeit in einem definierten Zustand aus einer bekannten Zustandsmenge. Entsprechend der Eingaben und interner Logik geht der Automat von Zustand zu Zustand. (Hromkovic 2011; Lee & Seshia 2012) IEC-61131-Programme können Programmteile als Tasks gebunden an Daten- oder Zeitänderungen ausführen. (DIN 61131-3)

2.3.2 Steuerungen nach der DIN 66253

Die DIN 66253 beschreibt mit der *Process and Experiment Automation Realtime Language* (PEARL) eine Programmiersprache für deterministische, funktional kritische Echtzeitanwendungen in der Automatisierung, Verfahrenstechnik und für Fahr- und Flugzeugsystemen (DIN 66253). PEARL ist trotz früher Normung nicht etabliert. Dedizierte technische Plattformen sowie Entwicklungsumgebungen sind nicht verbreitet. Die Programmiersprache besitzt relevante Charakteristika (DIN 66253; Benra & Halang 2009):

- PEARL sieht ein ereignisbasiertes Vorgehen vor, bei dem abhängig von Werten oder Eingaben Tasks in einer Multitaskingumgebung ausgelöst werden. Eine Zyklusorientierung wie in IEC-61131-3-Programmen ist nicht beabsichtigt.
- Tasks in PEARL deklarieren unmittelbar Anweisungen.
- PEARL verfügt über Funktionen zur Implementierung von verteilten Steuerungsprogrammen, um verteilte Steuerungssysteme umzusetzen.

  - In einem PEARL-Programm können Zielumgebungen für Tasks definiert werden, wodurch eine Nutzung von Mehrprozessorarchitekturen möglich ist.
  - Innerhalb eines PEARL-Programms werden Verbindungen zwischen Systemen abstrakt beschrieben, wodurch keine Kommunikationsimplementierung im Steuerungsprogramm enthalten ist.

- Über Orientierung an Zustandskombinationen von Eingängen ist ein zustandsorientiertes Vorgehen in der Programmlogik möglich, wobei sich Implementierungen in PEARL nicht an Zuständen, sondern an Transitionsroutinen orientieren.

Ein PEARL-Programm ist aus Modulen aufgebaut, die keine Abhängigkeiten voneinander aufweisen. Es können globale Strukturen (Variablen) deklariert werden, die modulübergreifend erreichbar sind. Prozeduren, die Funktionen oder Methoden in anderen Sprachen entsprechen, werden innerhalb von Modulen definiert. Ein Modul verfügt über einen *Main*-Teil, der Kern und Einstiegspunkt der Modullogik bildet. Aus diesem Teil oder auslösergebunden werden definierte Tasks ausgeführt, die über eigene Anweisungssequenzen verfügen und Prozeduren aufrufen können. Tasks haben Prioriäten und werden wie bei einem multitaskingfähigen Betriebssystem über Starts, Unterbrechungen, Abbrüche und Fortsetzungen verwaltet, wodurch eine ausreichende Reaktionsfähigkeit des Systems sichergestellt werden kann. (DIN 66253; Benra & Halang 2009)

Verteilte Systeme in PEARL

PEARL unterstützt die Erstellung von Programmen, die auf ein System bestehend aus mehreren physischen Teilen entsprechend einer Modellierung verteilt werden. In diesen Programmen wird eine textuelle Architekturbeschreibung eingeführt, die nicht in laufzeitaktive Software übersetzt wird, sondern nur zur Konfiguration beim Bauen der Software dient. Die Geräte des entstehenden verteilten Systems werden als Stationen bezeichnet. Verbindungswege zwischen den Stationen werden explizit beschrieben. Den Stationen werden über die Beschreibung Systemteile und Gerätezugriffe zugewiesen. Die Kommunikation wird über Signale definiert, die auf Ports gelegt werden. Ports abstrahieren als Struktur die tatsächliche Kommunikationsschnittstelle und das darübergelegte Protokoll. Signale abstrahieren die Übermittlung von Datenwerten. (DIN 66253; Benra & Halang 2009)

2.3.3 Kommunikation über Feldbusse

Feldbusse sind netzwerkorientierte, digitale Kommunikationssysteme, die Feld- und Steuerungssysteme für Industrieautomatisierungsanwendungen unter Einhaltung der speziellen Anforderungen prozesskritischer Kommunikation untereinander vernetzen. Sie sind wesentlich normiert in der IEC 61158. Beispiele hierfür sind Profinet und Ethercat. Feldbusse gewährleisten eine verlässliche Nachrichtenübermittlung innerhalb bekannter und kürzestmöglicher Zeitfenster. Sie erfüllen bei Normeinhaltung Determinismus- und Echtzeitanforderungen (siehe DIN 61784-2). Aus Sicht der Schichtenabstraktion für Netzwerksysteme gesehen, implementieren sie den Bereich von der physischen Schicht bis an die Anwendungsschicht, teilweise auch Elemente der Anwendungsschicht. Sie übernehmen auch Transport- und Sitzungsschichten und die Formatierung zu übermittelnder Daten. (DIN 61158-1) Feldbusse können bei Normerfüllung Kommunikation für Sicherheitsfunktionen übertragen. (DIN 61158-3 und Unterprofile) Freie Kommunikationsreserven eines Feldbusses dürfen für nicht-zeitkritischen Datenaustausch genutzt werden. (DIN 61158-1; DIN 61158-3) Bei ethernetbasierten Feldbussen ermöglicht dies den zusätzlichen Betrieb herkömmlicher Kommunikationsprotokolle wie TCP über die physische Verbindung.

Feldbusfähige Sensorik und Aktorik werden als abgeschlossene Komponenten ausgeliefert, die über eine feste, über den Feldbus verfügbare Parameterstruktur verfügen. Die Parameter folgen einer festen Definition, die über Feldbusfunktionen ausgelesen oder über eine Beschreibung in einem digitalen oder Papierformat an den Steuerungsintegrator übergeben wird. Auf Steuerungsseite werden in den zugehörigen Entwicklungsumgebungen unter Nutzung der abgerufenen Parameterdefinition oder der eingebundenen Beschreibung Topologiebeschreibungen erstellt, die als Programm- und Kompiliergrundlage dienen. Beispiele für entsprechende Beschreibungsformate sind GSD (*Gerätestammdaten* oder *General Station Description*) bei Profibus und GSDML (*GSD Markup Language*) bei Profinet.

2.3.4 OPC Unified Architecture (IEC 62541)

OPC Unified Architecture (OPC UA, IEC 62541) beschreibt eine Beschreibungs-, Zugriffs- und Kommunikationsarchitektur für vernetzte Industriesysteme. Die zwei wesentlichen Teile

einer OPC-UA-Implementierung sind das Informationsmodell, das den Zugriff auf die Daten und Funktionen einer Anwendung oder eines Geräts ermöglicht und das Kommunikationsmodell, das getrennt vom Informationsmodell einen standardisierten Zugriff auf die Daten und Funktionen organisiert. (OPC Foundation 2017a) Das OPC-UA-Informationsmodell ist ein beliebig verzweigbarer Graph, der neben Daten auch Funktionen abbilden kann. OPC UA ist keine ebenenspezifische Technologie, besitzt integrierte Sicherheitsfunktionen (siehe OPC Foundation 2015) und domänen- oder prozessspezifische Grundmodelle über öffentliche Standards (*Companion Specifications*). OPC UA ist auf die Schaffung eines digitalen Abbilds eines technischen Systems ausgerichtet und stützt eine konfigurationsbasierte Kommunikation als Ersatz für programmierte Kommunikation. Das technische System besitzt ein Informationsmodell, das von einem OPC-UA-Server anhand eines Kommunikationsmodells instanziiert und für die Außenwelt bereitgestellt wird.

Informationsmodell

Grundlage des OPC-UA-Informationsmodellsystems sind *Nodes*. Ein Informationsmodell besteht aus einer beliebigen Menge Nodes. Jeder Node beschreibt einen Datenpunkt (meist äquivalent einer Variable) in einem standardisierten Format, wobei die Bestandteile des Formats so wie Ausprägungen und Relationen ebenfalls über Nodes beschrieben werden. Alle Nodes sind einem *Namespace* zugewiesen, der über einen Index identifizierbar ist und seine Nodes mit zusätzlichen Metainformationen über Referenzen erweitert. Namespaces sind innerhalb eines Informationsmodells eindeutig identifizierbar, ebenso wie Nodes. Die Nodeidentifikation erfolgt über die Kombination aus der numerischen Namespaceidentifikation und einem von vier möglichen Nodeidentifikatoren (numerisch, String, Guid nach RFC 4122, Base64-String). Standardnodes, die von einem OPC-UA-Server bereitzustellen sind, sind fest an einen Standardnamespace und Standard-Nodeidentifikationen gebunden.(OPC Foundation 2017c; OPC Foundation 2017b) Das Informationsmodell ist aggregationsfähig (siehe Abschnitt A.1.3). OPC UA gibt einen Standardsatz von Datentypen vor, die ein *Variablen*-Node annehmen kann, die den Typen üblicher Programmiersprachen entsprechen.

Kommunikationsmodell

Das Standard-OPC-UA-Kommunikationsmodell folgt einem Server-Client-Muster über TCP. Der Server baut sein Nodesystem anhand des Informationsmodells auf. Informationsmodelle können über eine Browsing-Funktion von Clients abgerufen werden. Clients lesen über ihre stehende Verbindung Nodes nach eigener Logik. Auf das Server-Client-Muster kann ein nodefeines Monitoringparadigma gelegt werden, wodurch ein Client bei Änderung eines Nodewertes durch den Server mit dem neuen Nodewert benachrichtigt wird. Ist OPC UA über Ethernet/TCP implementiert, ist es nicht deterministisch. (Schleipen et al. 2018) Neben dem Server-Client-Muster kann OPC UA auch mit einem Publish-Subscribe-Muster (*OPC UA PubSub*) über UDP-Broadcast, MQTT oder andere Protokolle implementiert werden (Yuan et al. 2021). OPC UA PubSub verfügt damit über keine technisch vollständige interne Publish-Subscribe-Logik, sondern lagert diese aus.

OPC UA kann bei Verwendung mit Ethernet TSN deterministische Kommunikation ermöglichen. (Yuan et al. 2021; Eymüller et al. 2020; Luo et al. 2021) Eymüller et al. 2020 zeigt die Implementierung eines explizit deterministischen OPC-UA-Kommunikationsmusters auf Basis von OPC UA PubSub über UDP-Broadcast und validiert dieses mit physisch getrennten IPC, die jeweils ein OPC-UA-Programm in C sowie C++ ausführen. Der Datenaustausch ist für eine einstellige Zahl Gleitkommazahlen mit kleiner Abweichung unter einer Millisekunde durchführbar. Eine verlässliche Latenz kleiner einer Millisekunde mit geringem Jitter wird für OPC UA über TSN auch in Luo et al. 2021 gezeigt. Die Kommunikationspartner sind hier in C# implementiert. OPC UA ist ein verbreiteter und empfohlener Ansatz für die Umsetzung agentenbasierter Automatisierungssysteme (Leitão et al. 2021).

2.3.5 Programmierung und Modellierung

Programmierung und Modellierung sind Vorgänge mit spezifischen Werkzeugen in der Entwicklung von technischen Systemen. In breiter Anwendung finden sie in der Automatisierung von Maschinen und Anlagen auf den Feld- und Steuerungsebenen statt. Die technischen Systeme der Organisationsebene sind als abgeschlossene proprietäre Softwareanwendungen intern nicht von Programmierungs- und Modellierungsvorgängen außerhalb

des Herstellers betroffen. Sie verfügen über Schnittstellen (beispielsweise OPC UA nach VDI 5600-3), die in der Entwicklung auf Steuerungsseite eingebunden werden.

Zwar ist die Programmierung mittels IEC 61131-3-Sprachen eine Gemeinsamkeit etablierter Steuerungssysteme und der an sie gebundenen Entwicklungswerkzeuge, allerdings sind diese durch Eigen- und Geschlossenheit der Implementierungen und der Kompilierungs- und Schreibvorgänge voneinander abgeschlossen. Entwicklungsumgebungen, gebaute Programme, Bibliotheken, Hardwaremodelle, Netzwerktopologien et cetera sind in der aktuellen Steuerungstechnik nicht interoperabel - auch nicht innerhalb von Herstellerangeboten, beispielsweise aufgrund von Architekturunterschieden. (Rothhöft 2022; Fischer et al. 2020)

Die universale Repräsentation und eine zugrundeliegende Modellierung von IEC-61131-Systemen zur Laufzeit sind Forschungsgegenstand (siehe Cavalieri & Salafia 2020; Gampig et al. 2021; Tantik & Anderl 2017). Für die Entwicklungszeit steht PLCopen XML (DIN 61131-10) zur Verfügung, um Programme und Programmelemente auszutauschen. Für die Abbildung von Komponenten in Topologiemodellen, Parametrierungen et cetera sind feldbusgebundene Formate (beispielsweise GSDML (Profinet)) vorgesehen. Durch die Normierung sind die Feldkomponenten eindeutig an herstellerunabhängige Beschreibungsformate gebunden. Für Firmwareprogrammierung und Komponentenbeschreibungen wurden FDT und EDD (siehe Abschnitt 3.2.2) aufgegriffen (Shan et al. 2010).

Für die Beschreibung von Automatisierungsprojektdaten ist AutomationML vorhanden, ein Standardformat, das von Herstellern über Entwicklungsumgebungen unterstützt wird. (Drath et al. 2021; Gangoiti et al. 2021) AutomationML ist normiert (DIN 62714-1) und auf die Nutzung in Automatisierungsprojekten ausgelegt. Ein vergleichbares, nicht automatisierungsspezifisches Format, welches für Automatisierungsprojekte ebenfalls genutzt werden kann, ist SysML. (Berardinelli et al. 2016) Ein wesentlicher Unterschied ist die feste Berücksichtigung von relevanten Dateiformaten für Referenzen in AutomationML: die Norm sieht die Einbettung von COLLADA-Daten für 3D-Informationen nach der ISO/PAS 17506:2012 sowie die Einbettung von IEC-61131-3-Projektdaten vor. Letzte werden mittels PLCopen XML gespeichert. Die Norm lässt die spätere Erweiterung mit weiteren Formaten zu und berücksichtigt die Referenzierung von Dokumenten, die in keinem aktuell von AutomationML explizit zugelassenen Format vorliegen. (DIN 62714-1) Für die Nutzung von AutomationML

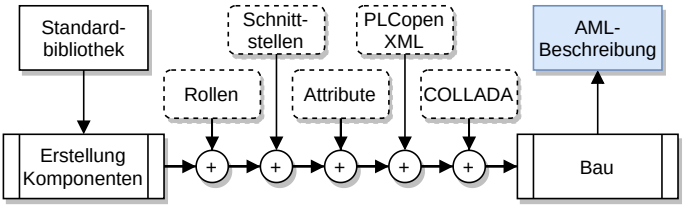


Abbildung 2.5: Bau einer AutomationML-Beschreibung (nach Drath et al. 2021)

für die Entwicklung von Prozessleittechnik existieren standardisierte Vorgehensweisen (VDI 3697-1; VDI 3697-2). Die Herstellerinteroperabilität des AutomationML-Formats liegt im eingebetteten IEC-61131-3-Austausch über PLCopen XML und ist an dessen Nutzung gebunden (Gangoiti et al. 2021). Zur Beschreibungsbildung (verkürzt in Abbildung 2.5) werden komponentenbezogene Instanzen aus Standardbibliotheken gebildet, die in einer festgelegten Reihenfolge mit Informationen angereichert aggregiert werden.

Die Norm DIN 61131-10 definiert das Beschreibungsformat PLCopen XML zur standardisierten Abbildung von IEC-61131-3-Elementen in XML-Dateien. Ziel ist die Bereitstellung eines hersteller- und softwareunabhängigen Standardformats, wodurch die Bindung an bestimmte Entwicklungsumgebungen und andere Entwicklungswerkzeuge entfällt und generalisierte Elemente systemunabhängig ausgetauscht werden können.

2.4 Zusammenfassung und Bewertung

Automatisierte technische Systeme führen unabhängig physische Vorgänge nach einem definierten Muster durch, wodurch Prozesse ohne menschliches Zutun durchgeführt werden können. So können Kosten gespart oder Vorgänge durchgeführt werden, die ein Mensch nicht erledigen kann. In der Industrie sind teil- oder vollautomatisierte Systeme über elektronische und softwaregetriebene Automatisierungsansätze heutiger Standard. Bewertungskriterien für die Leistungsfähigkeit eines automatisierten technischen Systems sind seine Adaptierbarkeit und sein Automatisierungsgrad. Die Adaptierbarkeit beschreibt, wie gut sich ein existierendes System an Prozessänderungen selbst anpassen oder angepasst werden kann. Der Automatisierungsgrad beschreibt den Umfang der automatisierten Teilprozesse am Gesamtprozess. Die Fusion von Adaptierbarkeit und Automatisierungsgrad ist

eine besondere Schwierigkeit, der mit der großen Gestaltungsfreiheit softwaregetriebener Ansätze wie dem CPS-Ansatz begegnet werden kann.

Technische Systeme zur Automatisierung, insbesondere die Systeme mit Schnittstellen zum physischen Prozess, unterliegen besonderen Anforderungen, durch deren Nichteinhaltung physische Einrichtungen in fehlerbehaftete oder gefährliche Zustände übergehen können. Die Automatisierungssysteme müssen deswegen deterministisch agieren und Grenzen sowie Zeitanforderungen des Prozesses jederzeit einhalten.

In der industriellen Praxis werden automatisierte Systeme in sequentiellen Prozessen erstellt, in denen die technischen Systeme für die Automatisierung oft als letztes entstehen. Maschinen und Anlagen sowie ihre Automatisierungstechnik sind oft Einzelstücke. (Eckert 2015) Die Entwicklung von Automatisierungssystemen und ihrer Software findet zu einem großem Teil mit herstellerspezifischen Werkzeugen statt. Automatisierungskomponenten sind dabei zumeist an die vorgegebene Software ihrer Hersteller gebunden. Es existieren herstellerübergreifende, modellbasierte Formate zur Beschreibung von Automatisierungssystemen und ihren Teilen, die verschieden gut unterstützt werden. Durch herstellerspezifische Elemente in den Formaten ist die herstellerübergreifende Nutzung eingeschränkt. Die Software der Steuerungs- und Feldsysteme ist besonders eingeschränkt. Interoperabilität ist in der Praxis auf die Kommunikation begrenzt. (Poștovei et al. 2020)

Die Industrieautomatisierung, ihre Struktur und ihre Systeme werden heute in Ebenen abstrahiert, die in einer Automatisierungspyramide hierarchisch aufeinander aufsetzen (DIN 62264-1). Sie unterteilt die an der Automatisierung beteiligten technischen Systeme in leitende (Organisationssysteme wie MES und ERP), in Steuerungs- und Feldsysteme auf. Die Feldsysteme befinden sich unmittelbar an der automatisierten physischen Einrichtung. Die Steuerungssysteme steuern über die Feldsysteme die physische Einrichtung und kommunizieren mit den übergeordneten Organisationssystemen. Systeme der Automatisierungspyramide sind nach Bedarf horizontal und vertikal integriert. Im Organisationsbereich der Automatisierung herrschen vertikal wie horizontal Standardtechnologien aus der IT zur Integration vor. In der vertikalen und horizontalen Integration der Steuerungs- und Feldebenen werden teilweise Standardtechnologien, vorrangig allerdings deterministische, automatisierungsspezifische Technologien eingesetzt.

Prägend für die heutige Industrieautomatisierung sind ihre spezifischen Technologien, die in anderen Domänen nicht verwendet werden. Diese Technologien sind bei der Erstellung oder beim Betrieb von Automatisierungslösungen aktiv und vorwiegend Teil der Steuerungs- und Feldebenen. Zentraler Teil heutiger Automatisierungssysteme sind Steuerungssysteme nach der DIN 61131-1, die Softwareprogramme in normierten Sprachen zur automatisierten Prozesssteuerung ausführen. Sie können als reine Softwaresysteme auf Standardumgebungen umgesetzt sein, sind aber im Regelfall auf dedizierten technischen Plattformen auf Mikrocontroller- (SPS) oder IPC-Basis ausgeführt, um Determinismus- und Zeitanforderungen einhalten zu können. Programme werden kompiliert, gebaut und auf die technische Plattform geladen. Die Sprachen (IEC 61131-3) sind auf die programmatische Abbildung von physischen Zuständen in einem Programm ausgelegt. Ein erstelltes Programm wird in einer Endlosschleife ausgeführt, wobei globale Daten außerhalb der Zykluslogik liegen können, und über eigene oder externe Ein- und Ausgänge gegen die physische Welt gearbeitet wird. Die IEC 61131 konzentriert eine Automatisierungslogik auf einzelne, abgeschlossene Systeme, wobei die Logik als starres Programm in einer der vorgegebenen Sprachen implementiert werden muss, um anschließend auf das System ausgeliefert werden zu können. Verteilte Systeme müssen auf mehrere Programme mit mehreren Systemen und implementierter Kommunikation ausgestreckt werden. Adaptierbarkeit ist über implementierte Flexibilität oder über Rekonfiguration mittels neuer Programme umzusetzen. Die programmatischen Möglichkeiten sind auf die Abbildung physischer Prozessschritte, Logik- und einfache Operationen beschränkt.

Neben den IEC-61131-Steuerungssystemen werden mit kleinem Anteil andere technische Ansätze verwendet. Dazu zählen SPS und IPC, die in Mittel- und Hochsprachen (beispielsweise C, C++) programmiert werden sowie Steuerungssysteme nach der DIN 66253. (Rothhöft 2022) Bei ersteren gelten ähnliche Einschränkungen wie bei dedizierten IEC-61131-SPS, wobei die programmatischen Möglichkeiten größer sind. DIN-66253-Steuerungssysteme sind wenig verbreitet, verfügen aber über relevante Elemente zur Definition von verteilten, multitaskingfähigen Echtzeitsystemen, die für zukünftige Entwicklungen aufgreifbar sind. Für die Weiternutzung von bestehenden Steuerungssystemen in moderneren Umgebungen existieren Adaptionsansätze als eigenständige Softwareadapter auf dedizierter Hardware oder als unabhängige Softwareadapter auf allgemeiner oder Fremdhardware (Rupprecht

et al. 2021). Eine Adapterlösung stellt eine Übersetzung zwischen Steuerungstechnik und Umgebung her, wofür sie beide Seiten mit ihrer Logik integrieren können muss. Die Adaptionsfähigkeit entsteht bereits mit der Bereitstellung einer OPC-UA-Kommunikationsfähigkeit. (Givehchi et al. 2017)

Die Prozessbeherrschung durch Steuerungs- und Feldsysteme erfordert deterministische und ausreichend schnelle Kommunikation, um Informationen zwischen Steuerungssystemen, Aktorik und Sensorik prozessgerecht zu verteilen. Hierfür werden Feldbussysteme genutzt, insbesondere Feldbusse nach der DIN 61158-1 und mit Ethernet als physischer Grundlage. Diese Feldbussysteme reichen aus Netzwerkschichtensicht bis in die Applikationsschicht und stellen Standardformate für den Datenaustausch. Ihre Kommunikation ist auf Komponentenebene stark vordefiniert und über Begleitdokumente/-daten oder über die Systeme selbst beschrieben. Auf Steuerungsseite wird die Kommunikation programmatisch oder konfigurativ eingebettet. Ethernet-Feldbusse erlauben die Nutzung freier Übertragungskapazitäten für andere Protokolle wie OPC UA über TCP.

Wertung im Kontext

Aus technischer Sicht sind die Systeme der bestehenden Industrieautomatisierung separiert, voneinander abgeschlossen und werden punktuell und nach Bedarf statisch verbunden (Poștovei et al. 2020). Ihre verschiedenen Ebenen weisen getrennte Netzwerke und eigene Schnittstellen auf, welche durch die spezifischen Systeme definiert und vorgegeben werden. Integration und technische Systeme sind nicht auf die Bildung verteilter oder cyberphysischer Systeme ausgelegt. Der Aufbau der Einzelsysteme und ihrer Integrationen läuft spezifisch, in isolierten systembezogenen Modellen und Programmen, nach hierarchischen und statischen Paradigmen und mit programmatisch definierter Kommunikation.

Die dedizierten IEC-61131-Steuerungssysteme, die den Großteil der Steuerungen in der Industrieautomatisierung stellen, sind für die Automatisierungsumsetzung unter CPS-Paradigmen nicht hinreichend fähig. Die Modellierung und Programmierung in der IEC 61131 sind manuell durchzuführen, auf Einzelsysteme und eine in einer kontrollierten Laufzeitumgebung abgeschlossenen Logik ausgelegt. Die programmatischen Möglichkeiten sind auf die IEC-61131-Sprachen beschränkt, die verglichen mit Mittel- und Hochsprachen geringe Gestaltungsfreiheiten aufweisen. Insbesondere Kommunikationsimplementierun-

gen sind auf die durch den jeweiligen Hersteller implementierten Möglichkeiten beschränkt. IEC-61131-Systeme implementieren keine Umgebung für die Ausführung von Software außerhalb der Steuerungsprogramme.

Die Normierung in der Steuerungstechnik kennt mit der DIN 66253 eine Norm für Steuerungssysteme und ihre Programmierung, die zwar in der Praxis nicht etabliert wurde, aber validierte Eigenschaften besitzt, die eine Umsetzung des CPS-Ansatzes im Bereich unterstützen können. Diese sind: textuelle Beschreibung eines verteilten Systems an einem Punkt für die Instanziierung, integrierter Aufbau eines Programmmodells für alle beteiligten Systeme an einem Punkt, Textbasierung der Programmierung, Abstraktion der unter Ausführung und Kommunikation liegenden Soft- und Hardware sowie unabhängige Bindung von Vorgängen an Zustände aus EA und Datenpunkten.

Um die CPS-Paradigmen Vernetzung und Kooperation bei zusätzlicher dynamischer Bildung von CPS für den bedarfgerechten Aufbau von automatisierten Systemen zu ermöglichen, muss der Integrationsaufgabe, die bei der Verbindung von Systemen anfällt, mit einem generischen Ansatz begegnet werden. Aus der heutigen Industrieautomatisierung lassen sich ethernetbasierte Integration und OPC UA adaptieren. Über ethernetbasierte Integration können alle modernen relevanten Kommunikationsprotokolle aus Internet-, Netzwerk- und Feldbusbereichen, auch parallel, in Systemen implementiert werden. Über Ethernet-Netzwerke lässt sich eine allgemeine Verbindungsfähigkeit zwischen technischen Systemen aller Kategorien vorhalten. Die Adaption von OPC UA im Gesamten oder in Ansätzen löst Repräsentations-, Datentypen- und Datenformatprobleme, wodurch die notwendige Interoperabilität für die universale Verbindung von Systemen geschaffen und zudem durch die Typ- und Formatangleichung die Kommunikation aus spezifischen programmierten in konfigurierte Strukturen überführt werden kann.

Aktuelle standardisierte Modellierungssysteme für die Automatisierung sind für CPS in Teilen aufgreifbar (beispielsweise AutomationML für Systembeschreibungen), aber nicht gestaltgebend. Für die Entwicklung von CPS-fähigen Modellierungssystemen gibt es Ansätze (Mohamed et al. 2021; Juhlin et al. 2021), die allerdings nicht umfassend genug sind. Für die technische Modellierung sind Elemente aus IEC 61131 und DIN 66253 als Grundlage oder Orientierung nutzbar.

3 Entwicklung der Industrieautomatisierung

3.1 Abstraktionssysteme

Die Automatisierungspyramide, die als Standardabstraktion für den Aufbau der Automatisierung gilt, dient als Verständnisgrundlage für die Hierarchien und Abfolgen in der Automatisierung. Die Pyramide ist organisationsintern: alle Ebenen befinden sich innerhalb einer organisatorischen Einheit. Die Übergänge zu anderen Organisationen (Kunde, Lieferant, externe Logistik) werden nicht abgebildet. Wird der Kunde in die Produktentwicklung einbezogen oder das Produkt in einem anderen Unternehmen weiterbearbeitet, erfolgt die physische und digitale Übergabe außerhalb der Automatisierungspyramide. Auch nach unten zeigt sich die Begrenzung der Pyramide auf bleibende organisationsinterne Systeme: eine Interaktion mit einem intelligenten Produkt ist nicht abbildbar.

Eine Überlegung zur Erweiterung der Automatisierungspyramide nach oben wurde durch Körner et al. 2019 vorgestellt. Der Ansatz richtet sich an den Energiebereich. Verallgemeinert für die Automatisierung wird er in Abbildung 3.1 dargestellt. Im Ansatz wird die Pyramide nicht mit den internen kaufmännischen/organisatorischen Ebenen abgeschlossen, sondern bezieht externe Entitäten darüberliegend ein. Es wird eine abstrakte gemeinsame Integrationsschicht angenommen, die Systeme aller Ebenen integriert.

Referenzarchitekturmodell Industrie 4.0

Durch Arbeitsgruppen und Beteiligte aus verschiedenen Verbänden und Normungsgremien wurde das *Referenzarchitekturmodell Industrie 4.0* (RAMI4.0) als I4.0-Grundmodell erarbeitet, und als DIN SPEC 91345 sowie IEC PAS 63088 standardisiert. Das RAMI4.0 erweitert das Abstraktionsverständnis und bezieht Erweiterungen der Automatisierungspyramide ein. Es gilt als zukünftiges Abstraktionssystem. (Heidel et al. 2017)

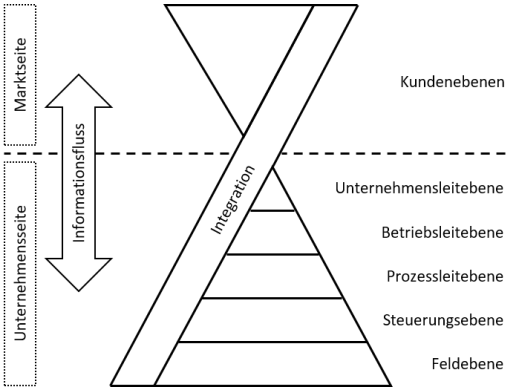


Abbildung 3.1: Die um Kundenebenen und eine gemeinsame Integration erweiterte Automatisierungspyramide nach Körner et al. 2019, verallgemeinert

Durch das RAMI4.0 sollen Betriebsmittel in einer I4.0-fähigen Abstraktion beschrieben und für eine gemeinsame digitale Integration gleichartig repräsentiert und integriert werden. Dadurch soll vollständige Durchgängigkeit innerhalb eines Unternehmens sowie über Unternehmen (Partner, Lieferanten und Kunden) und Komponentenarten (Produkt, Werkzeug, Software, Fabrik et cetera) hinweg geschaffen werden. Ein physischer oder virtueller Gegenstand, der für die Industrie relevant ist, wird im RAMI4.0 *Asset* genannt. Ein Asset ist durch einen *Begriff*, bestehend aus Begriffsbezeichnung, Begriffsdefinition und Eigenschaften gekennzeichnet und ist auf einer oder mehreren Ebenen verortet. Diese Eigenschaften geben *Merkmale* des Assets wieder. Jedes Asset wird durch Einordnungen seiner *Bekanntheit* über eine Bekanntheitsklasse und seiner *Kommunikationsfähigkeit* über eine Kommunikationsklasse charakterisiert. (DIN SPEC 91345; IEC PAS 63088, siehe Abschnitt E.1) Im RAMI4.0 soll für jedes Asset ein vollständiges digitales und für jeden verständliches Abbild existieren. (Heidel et al. 2017) Dafür wird für das RAMI4.0 die *Verwaltungsschale* (VWS) definiert. Sie ist eine Informations- und Integrationshülle, die um einen I4.0-relevanten Gegenstand aufgebaut wird (siehe Abschnitt 3.2.3, Abbildung 3.8). Die Kombination aus kommunikationsfähigem Asset (siehe Tabelle E.2) und VWS wird als I4.0-Komponente bezeichnet.

Das RAMI4.0 stellt ein mehrdimensionales Abstraktionsmodell für Produktionssysteme vor, das die Automatisierungspyramide ersetzt. Die Abbildung 3.2 zeigt das Abstraktionsmodell

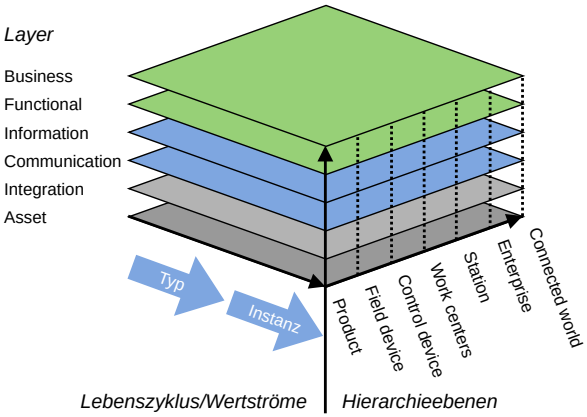


Abbildung 3.2: RAMI4.0-Modell zur Asset-Abstraktion (nach Plattform Industrie 4.0 2016)

mit den folgenden Achsen und Ebenen (Heidel et al. 2017; DIN SPEC 91345; IEC PAS 63088):

- **Lebenszyklus und Wertströme**: Lebenszyklusphase über Asset-Typ und -Instanz
- **Hierarchie-Achse**: Einordnung in hierarchische Systemordnung; basiert auf Erweiterung der Automatisierungspyramide
- **Architektur-Achse** (Layer): Sechs Schichten zur Wiedergabe der Asset-Beschaffenheit:
  - **Business:** Beschreibung von rechtlichen, geschäftlichen und organisatorischen Bedingungen, Vorgängen und ähnlichem
  - **Functional:** I4.0-konforme, fachliche Funktionen und Dienste des Assets
  - **Information:** von der Funktionsschicht genutzte, erzeugte und notwendige Daten und deren Weiterverarbeitung zu besonderen Informationen oder Wissen
  - **Communication:** I4.0-konformer Zugriff auf das Asset durch andere Assets
  - **Integration:** Integration mit physischer Welt; „Industrie-3.x"-Eigenschaften
  - **Asset:** Wiedergabe der physischen Qualität

3.2 Technische Frameworks

Arbeiten zu Frameworks für verteilte oder konfigurierbare Softwaresysteme in der Industrieautomatisierung sind bereits vorhanden. Die vorhandenen Arbeiten verfolgen unterschiedliche Ansätze, besitzen aber wiederkehrende Schnittpunkte wie die Einführung einer starken Hardwareabstraktion, modellbasierte Systemdefinitionen, tiefe Vernetzung oder Verwaltung aller Systeme in allen Ebenen von einem Punkt aus. Im Rahmen der Arbeit werden fünf technische Frameworks identifiziert, die genauer vorgestellt werden. Gewählt werden ausschließlich Frameworks, die

- Paradigmen für verteilte, konfigurierbare Systeme integrieren,
- für Automatisierungsanwendungen vorgesehen oder adaptiert sind
- und zu denen vollständige Beschreibungen oder Implementierungen verfügbar sind

Besprochen werden:

3.2.1 IEC 61499: eine Norm zur modellbasierten Erstellung von Steuerungsprogrammen für verteilte Automatisierungssysteme

3.2.2 Field Device Integration: eine Norm für die Architektur von Automatisierungsgeräten und dem Aufbau von verteilten Systemen aus diesen Automatisierungsgeräten

3.2.3 Verwaltungsschale: die Softwarestruktur für I4.0-Komponenten im RAMI4.0 nach DIN SPEC 91345, IEC PAS 63088

3.2.4 Distributed Model Object Environment: eine objekt- und modellbasierte Definitionsumgebung für den Aufbau von verteilten Automatisierungssystemen

3.2.5 AUTOSAR: ein Framework zum plattformunabhängigen Aufbau von Steuergerätesoftware im Automotivebereich, mehrfach adaptiert für die Industrieautomatisierung

In den drei Tabellen 3.1, 3.2, 3.3 am Kapitelende erfolgt eine Übersicht der Frameworks und eine Charakterisierung über ihre wesentlichen Ansätze, Elemente und Technologien. Im Anhang B erfolgen gesonderte, einzelne Bewertungen der Frameworks.

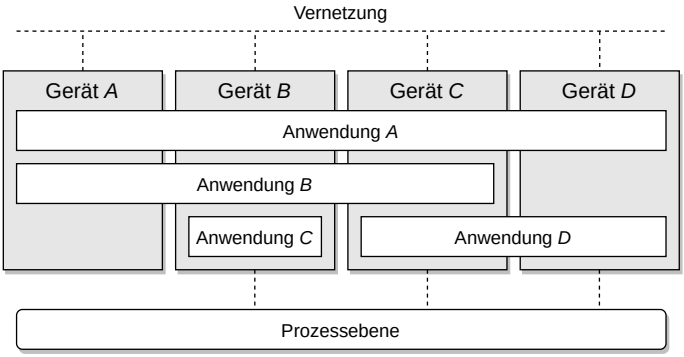


Abbildung 3.3: Ein IEC-61499-Systemmodell (nach DIN EN 61499-1)

3.2.1 IEC 61499

Die Norm IEC 61499 (DIN EN 61499) definiert ein modellbasiertes System zur Ein-Punkt-Erstellung von geräteübergreifenden Steuerungsprogrammen für verteilte Steuerungssysteme (Sieffert & Dorofeev 2021; Zoitl & Strasser 2016). Die Norm besteht aus:

- IEC 61499-1:2012 - Architektur
- IEC 61499-2:2012 - Anforderungen an Software-Werkzeuge
- IEC 61499-4:2014 - Regeln für normgerechte Profile

In einer IEC-61499-Steuerungsmodellierung wird ein Systemmodell (Abbildung 3.3) aus Geräten, Anwendungen, einem oder mehreren Kommunikationsnetzwerken und einer Prozessanbindung aufgebaut. Graphisch modellierte Anwendungsmodelle aus Funktionsblöcken (FB) stellen die eigentlichen Steuerungsprogramme. In den Anwendungsmodellen sind FB über ihre Ereignisein- und -ausgänge sowie ihre Datenein- und -ausgänge miteinander verbunden. Jeder FB erhält über das Verteilungsmodell eine Bindung an ein bestimmtes Gerät. Beim Durchführen der Verteilung werden die FB entsprechend auf die Ressourcen der beteiligten Geräte verteilt und Verbindungen zwischen den FB aufgebaut. Die Geräte einer Anwendung müssen nicht in einem Gesamtnetzwerk verbunden sein. Da die Kommunikation über Kommunikation-FB modelliert wird, die parametrisierbar sind, kann die

Integration der Geräte über verschiedene Netzwerke parallel stattfinden. Ein Gerät muss nicht am physischen Prozess teilnehmen, sondern kann rein virtuelle Funktionen beitragen.

Die Norm ist kein allein verwendbarer Standard, sondern muss mit weiteren Technologien erweitert werden. (Zoitl & Lewis 2014) Die Steuerungssystementwicklung mit der IEC 61499 ist so ausgelegt, dass die tatsächliche Implementierungstechnologie eines FB verborgen bleibt. Die Abstraktion erfasst auch die Kommunikations-FB, die Kommunikationsverbindungen zwischen Geräten einer Applikation umsetzen. Implementierungstechnologien werden durch die Norm nicht vorgegeben. Die Implementierung von verschiedenen, auch steuerungsuntypischen Kommunikationsprotokollen ist möglich (siehe beispielsweise Breunig & Götz 2018). Die Norm wird als Nachfolge- oder Ergänzungsnorm zur IEC 61131 gesehen, allerdings auch kritisiert (Thramboulidis 2013). Sie wurde hauptsächlich von der akademischen Fachwelt aufgegriffen und ist industriell nicht etabliert (Sehr et al. 2021; Lyu & Brennan 2021). Durch Entwicklung eigener FB können auch adaptive oder Systemfunktionen, beispielsweise zur Softwareverteilung, eingebracht werden, ebenso wie FB, die durch den Zugriff auf externe Software Definitionslogik aus dem IEC-61499-Programm nehmen und in beliebige Umgebungen auslagern können. (Grüner et al. 2018)

Struktur, Anwendung

Das durch die Norm festgelegte System beschreibt Steuerungsstrukturen mit acht Modelltypen (DIN EN 61499-1). Die Modellinstanzen entstehen teilweise in Automatisierungsprojekten und sind daran gebunden, teilweise sind sie generalisiert und wiederverwendbar.

- **Systemmodell:** Beschreibung eines Steuerungssystems (siehe Abbildung 3.3)
- **Gerätemodell:** Beschreibung eines Geräts und seiner Ressourcen (Verteilungsziele)
- **Ressourcenmodell:** Beschreibung einer Ressource, die zur Ausführung eines Anwendungsmodells bzw. des zugewiesenen Teils des Anwendungsmodells bereitsteht
- **Anwendungsmodell:** Steuerungsprogramm bestehend aus Funktionsblöcken
- **Funktionsblockmodell:** Grundaufbau der FB. Die funktionalen Grundelemente, aus denen eine Anwendung aufgebaut wird. FB implementieren endliche Automaten

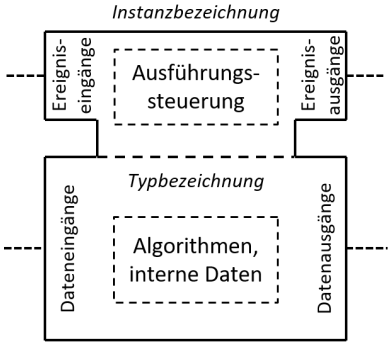


Abbildung 3.4: Aufbau eines IEC-61499-Funktionsblocks (DIN EN 61499-1)

(Yoong et al. 2015) mit Datenein- und -ausgängen, werden mit Ereignissen angestoßen und erzeugen Ereignisse (siehe Abbildung 3.4). Ein FB ist ein atomares Element (DIN EN 61499-1) und nicht teilbar.

- **Verteilungsmodell:** Verteilung des Anwendungsmodells auf die Ressourcen
- **Managementmodell:** Modelle für allgemeine System- oder Ressourcendienste
- **Betriebszustandsmodell:** Beschreibung der Betriebszustände

Anwendungen werden aus FB zusammengestellt, deren Ereignis- und Datenschnittstellen gezielt verbunden werden (siehe Abbildung 3.5). Die grafische Zusammenstellung und Verbindung von FB ist das Standardentwicklungsprinzip in einer IEC-61499-Entwicklungsumgebung. Ein in einer Anwendung genutzter FB ist eine Instanz eines FB-Typs. Zustände, Algorithmen und interne Daten des FB sind verborgen. Die FB sind in ihrer internen Logik nicht begrenzt. Sie können sowohl einfache Funktionen als auch ganze Programme implementieren. Ein FB kann aus einer verschalteten Menge von FB bestehen. FB verbergen Komplexität vor dem Steuerungsprogrammierer. Die FB können intern Zustände und Daten halten. Über die Ereigniseingänge wird die Ausführungssteuerung eines FB betätigt; analog kann sie über Ereignisausgänge andere Ausführungssteuerungen bedienen. Die FB-Algorithmen können über Dateneingänge bedatet werden; über Datenausgänge können Ergebnisse ausgegeben werden. Es gibt keine Standardumfänge hinsichtlich Ein- und Ausgängen bei Ereignis- und Datenschnittstellen, wobei üblicherweise

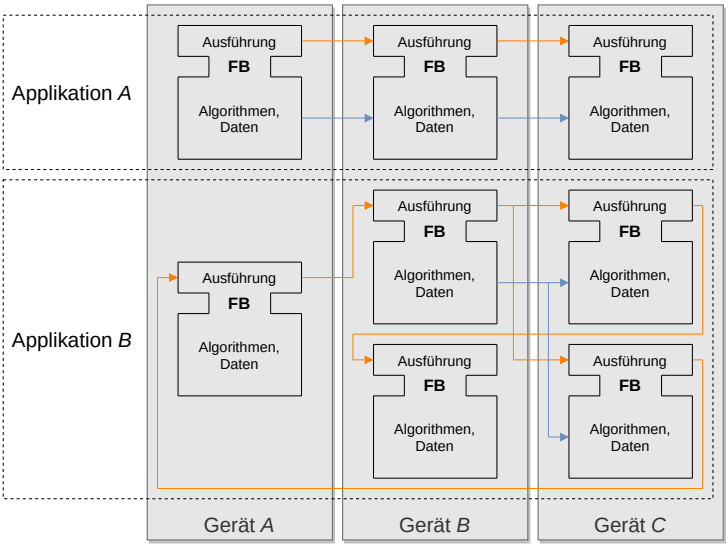


Abbildung 3.5: Anwendungsmodelle der IEC 61499 bestehend aus Funktionsblöcken

mindestens ein Ereigniseingang zur Auslösung sowie ein Ereignisausgang zur Anzeige des Bearbeitungsendes vorhanden sind. Das FB-Konzept lässt sich auf die IEC 61804 legen. Die Ausführungssteuerung differiert (DIN 61804-2). Abfolgen entstehen durch Anordnung und Verknüpfung von FB. Das Aufteilen einer Anwendung erfolgt zwischen FB. Geräte können an mehreren Anwendungen gleichzeitig beteiligt sein. Durch die FB-Basierung besitzt eine IEC-61499-Anwendung keine offenen Daten, wie sie beispielsweise als lokale oder globale Variablen in IEC-61131-Programmen vorhanden sind.

3.2.2 Field Device Integration

Die Field Device Integration (FDI) ist ein entwicklungs- und laufzeitaktives Framework für die Software auf und um Automatisierungsgeräte und deren Anordnung in einem gemeinsamen System. FDI baut auf drei Ausgangsnormen auf (DIN EN 62769-1): Field Device Tool (FDT, IEC 62543), Electronic Device Description (Language) (EDD(L), IEC 61804) und OPC UA (IEC 62541). Alle Ausgangsnormen sind eigenständige Welten, die mit der FDI adaptiert werden. EDD(L) und FDT werden im Folgenden beschrieben, OPC UA im Abschnitt

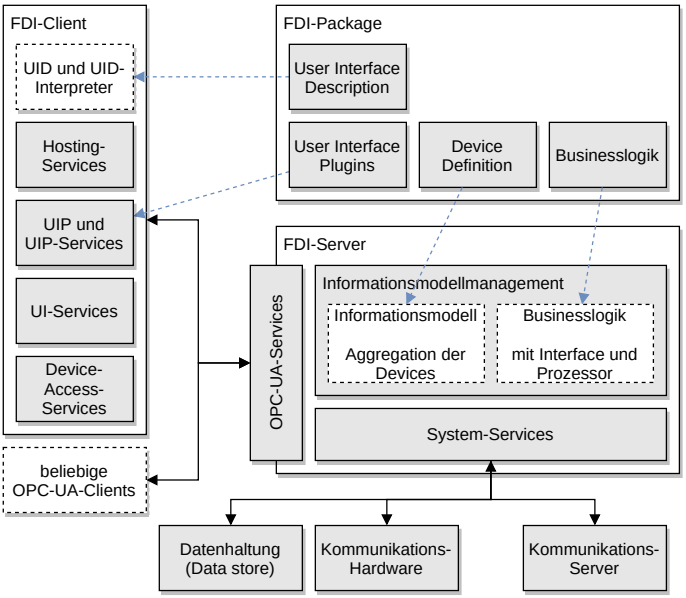


Abbildung 3.6: Struktur der Field Device Integration (nach DIN EN 62769-1)

2.3.4. Die FDI definiert Client-Server-Strukturen, in der Automatisierungsgeräte angeordnet werden, eine Grobstruktur für die Software der Clients (Automatisierungsgeräte) und der Server (Administrationsgeräte), ein Paketsystem für das Softwaredeployment sowie ein Servicesystem für die Integration. Das Framework berücksichtigt eine konzentrierte Datenhaltung, einen servicebasierten Zugriff auf aufgebaute Architekturen und die Integration von anwendungsfallgebundenen Businessapplikationen. Pakete sind mehrteilig und können mehrere Ziele sowie Anwendungen gleichzeitig bedienen.

FDI setzt die Einzelsysteme im Automatisierungssystem in definierte Beziehungen zueinander. Es baut dafür auf anderen technischen Standards auf und implementiert ein Profilsystem für Automatisierungsgeräte. Kommunikation und Integration innerhalb einer FDI-Architektur sind über OPC UA realisiert. Funktionsaufrufe über OPC UA werden als Services vorgesehen, also eine funktions- und nicht datenorientierte Logik. Der FDI-Server übernimmt über eine Datenhaltung eine Datenpersistierung. Über OPC-UA-Clients oder System-Services werden andere Kommunikationsprotokolle eingehängt.

Komponenten der Field Device Integration (FDI)

Die Kernbestandteile (Abbildung 3.6) sind drei Artefakte:

- **Server:** Der FDI-Server verwaltet das Gesamtinformationsmodell, bestehend aus instanzgebundenen Teilmodellen zur Gerätepräsentation und der Businesslogik, und realisiert die Anbindung an Clients (OPC UA Services), Datenhaltung (Data Store) und genereller Kommunikationsbereitstellung (Communication Server).
- **Client:** Der Client ist das Device, das die Beschreibungen, Interfaces und Services, die durch die Packages bereitgestellt werden, interpretiert und ausführt, und den Zugriff auf die Feldebene implementiert. Vom Client aus können Businesslogikvorgänge auf dem Server ausgelöst und auf Informationen auf dem Server zugegriffen werden.
- **Package:** Über Packages werden Devicebeschreibungen ausgerollt, die vom Server wiedergegeben und Clients ausgeführt werden. Die Packages beschreiben und implementieren Interfaces, beschreiben Geräte und liefern zugehörige Businesslogik.

User Interface Plugins sind als einziger Bestandteil an eine technische Plattform bindbar.

Packages und Package Deployment

Packages realisieren die Implementierung und Beschreibung von technischen Funktionen und Businesslogik in einem FDI-Gesamtsystem. Sie werden bei der Installation auf dem FDI-Server gespeichert und bei der Implementierung im System in Unterelemente aufgetrennt, die dann teilweise auf Clients (z.B. User-Interface-Plugins), teilweise auf dem Server (z.B. Datenpunkte) instanziiert werden. (DIN EN 62769-4)

- **Device Package**: Beschreibt einen einzelnen Gerätetypen für das Gesamtsystem
- **Communication Package**: Beschreibt ein FDI-fähiges Kommunikationsgerät (siehe DIN EN 62769-7) in den kommunikationsnotwendigen Eigenschaften
- **UIP Package**: Liefert *1..n* User-Interface-Plugins an den Server für den Nutzerzugriff
- **Profile Package**: Ist eine profilhafte Abstraktion eines Device Package in immer vorhandene Grundelemente und erlaubt eine vorläufige Integration, wenn das tatsächliche Device Package (also die tatsächliche Geräteausprägung) nicht vorliegt

Device, Communication und Profile Package bestehen mindestens aus einem Katalog und einer oder mehreren EDD. Diese Packagetypen können zudem User-Interface-Plugins (UIP) und freie Anhänge enthalten. Ein UIP-Package besteht aus einem Katalog und einem oder mehreren UIP.

Es gibt keine Normvorgabe für Dateitypen. Im Anhang der DIN EN 62769-4 werden für eine Beispielimplementierung Standard-MIME-Typen wie PDF und XML vorgeschlagen. Eine Aussage für die Deploymenttechnologien wird nicht unmittelbar getroffen. Für die Verbindung zwischen Client und Server ist nach Abbildungen OPC UA vorgesehen (siehe Abbildung 3.6), welches einen Datentypen für die Abbildungen von und den Zugriff auf Ordner und Dateien vorsieht (Anhang C, OPC Foundation 2017c).

Packages werden aus einem Dateisystem gelesen, in dem die Struktur des Packages unmittelbar in der Dateisystemstruktur in Ordnern und Dateien abgebildet ist. Für das Deployment wird das Package verifiziert und geprüft, ob es schon bekannt ist. Das Package wird gesamt und in den Subteilen in den jeweiligen Katalogen referenziert und in ein eigene Ablage kopiert. Nach dem Einlesen der Package-EDD und Erfolg bei allen Schritten werden die Datenpunkte des Packages zum Gesamtmodell hinzugefügt.

FDI sieht Lebenszyklusfunktionen für die Packagebetreuung in Form von Aktualisierungen, Erweiterungen bzw. Hochstufungen, Austausch und Deinstallationen von Packages vor. Aktualisierungen und Hochstufungen müssen rückwärtskompatibel sein und die vergangene Systemgestaltung unterstützen (Beibehaltung von Schnittstellen und Logik). Hochstufungen dürfen den Funktionsumfang und das allgemeine Funktionsziel eines Packages ändern. Die Lebenszyklusfunktionen sind nach der DIN EN 62769-4 nicht detailliert definiert, sondern dem Package-Lieferanten weitestgehend freigestellt.

Informationsmodelle und Abbildungen

Das Informationsmodell eines auf der FDI aufgebauten Systems wird in einer Instanz auf dem zugewiesenen FDI-Server des Systems gehalten. Der FDI-Server baut das Informationsmodell unter Einbezug der für das System hinterlegten Device-Packages. Datengrundtypen im Informationsmodell folgen der OPC UA.

Über das Informationsmodell wird die Systemtopologie in den einbezogenen Geräten und Netzwerken abgebildet. Die Geräte und Netzwerke können auch verschachtelt sein.

Die in der Topologie abgebildeten Geräte und Netzwerke, die Teil des FDI-Systems sind, müssen keine direkte Verbindung zum FDI-Server unterhalten. (DIN EN 62769-5)

Electronical Device Description und Field Device Tool

Die EDD ist eine elektronische Gerätebeschreibung mittels der EDD-Language und Teil der IEC 61804. Eine EDD beschreibt unter anderem Geräteparameter, Leitgeräteinteraktionen und Gerätefunktionen. (DIN EN 61804-4) EDD sind formatierungsstandardisierte Systembeschreibungen von Automatisierungsgeräten, die mit universalen Softwareapplikationen interpretiert werden und keine Implementierungen auf der Geräteebene vorsehen. Die Bindung an eine Systemarchitektur ist auf die interpretierende Applikation beschränkt. Geräte-EDD sind beschreibende Modelle für die FDI-Systemmodellierung, die in der Entwicklung, im Betrieb (durch aufbauende Applikationen) und in der Systemänderung genutzt werden können. (DIN EN 61804-4) Das Field-Device-Tool-Schnittstellenspezifikation nach der IEC 62453 beschreibt ein Schnittstellensystem für Feldgeräte und Feldgerätekomponenten, um interoperablen Zugriff auf Daten und Funktionen zu realisieren. Kernbestandteil des FDT ist ein Device-Type-Manager (DTM), der als Handler Software oder Geräte entsprechend der Norm verwaltet. Der DTM ist zur Anbindung an Entwicklungs- und Betreuungssoftware für das Automatisierungsgerät vorgesehen. Über die FDT-Schnittstelle kann die Software eines Automatisierungsgeräts über seinen Lebenszyklus betreut werden. Mit FDT sollen einheitliche herstellerunabhängige Werkzeuge für die Konfiguration, Betreuung und die Überwachung von Automatisierungsgeräten möglich sein. (DIN EN 62453-1)

3.2.3 Verwaltungsschale

Die Verwaltungsschale (VWS, auch *Asset Administration Shell*) ist das verwaltende Element für Logik und Daten zu Assets im RAMI4.0. Plattform Industrie 4.0 2019. Sie bündelt assetbezogen Daten und Logik und kann auf einer dem Asset zugehörigen Plattform (zum Beispiel eines IPC) oder einer allgemeinen Verwaltungsplattform (beispielsweise in einer Cloud) gespeichert und je nach Beschaffenheit auch ausgeführt werden. VWS-Formen sind:

- **passiv:** beschreibende Struktur, beispielsweise als AASX-Paket, XML oder JSON; kann eingelesen werden und beschreibt ein System
- **reaktiv** oder **passiv mit API:** Systembeschreibung wie passive VWS, mit zusätzlicher Schnittstellenspezifikation für HTTP/REST, OPC UA oder andere; Schnittstelle wird von der lesenden Anwendung instanziiert
- **aktiv:** aktive Softwarestruktur mit Steuerungsfähigkeit und protokollbasierter, komplexer Kommunikation und Integration mit der Umgebung

Die VWS ist in der passiven Form eine Repräsentation und lebenszyklusbegleitende Merkmalsammlung eines Gegenstands. Sie ist dann eine Informationsablage mit datei- oder schnittstellenbasiertem Zugriff. Sie kann mit einer dedizierten Schnittstelle zur passiven VWS mit API aufgewertet werden. Als passives System führt die passive VWS nicht eigenständig Funktionen, Abläufe et cetera durch, sondern ist auf eine ausführende Anwendung angewiesen. Die aktive VWS verfügt über eine eigene *I4.0-konforme* Kommunikation zur Integration mit anderen VWS. Sie verwaltet selbstständig ihre Daten und stellt Funktionen und Vorgänge bereit. Abbildung 3.7 löst die Verwaltungsschale abstrakt auf. Die halbseitige Trennung in einen deskriptiven und einen funktionalen Teil ist nicht Teil der VWS-Definition, sondern dient der Verständlichkeit in dieser Arbeit. Der deskriptive Teil der VWS besteht aus den statischen Informationen (eindeutige Identifikation des Assets und der VWS und weitere) und den Beschreibungen der *Teilmodelle* und wird in einem VWS-Register abgebildet. Der deskriptive Teil unterliegt Sichten, die den Zugang zu den Informationen regulieren und wird in einem standardisierten, einheitlichen Format abgespeichert. Der funktionale Teil realisiert über die Funktionsteile der Teilmodelle die Dienste und Funktionen, die in der VWS eingebettet sind. Der funktionale Teil kann über eine API oder Kommunikationsprotokolle zugänglich sein. Sie sind generisch oder gegenstandsspezifisch und können teilweise oder ganz standardisiert sein sowie ausgetauscht oder erweitert werden. Eine VWS besitzt einen Komponenten-Manager, der die Teilmodelle verwaltet. Manifest und Teilmodelle müssen nicht am selben Ort abgelegt sein. (Plattform Industrie 4.0 2019; Plattform Industrie 4.0 2022; DIN SPEC 91345)

Die VWS dient als Grundlage für Kommunikation und Interaktion von Automatisierungskomponenten nach dem RAMI4.0, wobei VWS in beliebiger Komplexität ausgestaltet und

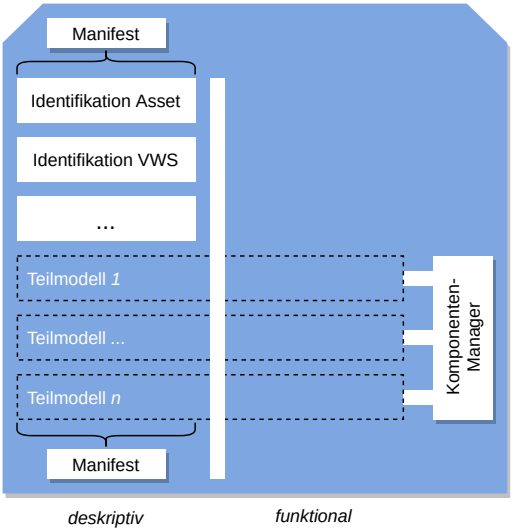


Abbildung 3.7: Struktur der Verwaltungsschale (DIN SPEC 91345)

kaskadiert werden können. Eine VWS kann weitere VWS als Teilmodelle aufnehmen. Teilkomponenten mit eigenen VWS können somit aggregiert in einer Gesamt-VWS gehalten werden. (Plattform Industrie 4.0 2019; Plattform Industrie 4.0 2022; DIN SPEC 91345)

Eine aktive VWS besitzt eine I4.0-konforme Kommunikation, über die sie mit anderen VWS und ihrer Verwaltung kommuniziert (siehe Abbildung 3.8). Das durch die VWS integrierte Asset kann nebenläufige Kommunikationsarchitekturen betreiben. Über diese Kommunikationsarchitekturen können auch VWS-lose Assets angebunden werden. (Plattform Industrie 4.0 2019; Plattform Industrie 4.0 2022; DIN SPEC 91345)

Es existiert eine Bandbreite an Arbeiten, die Teilaspekte der VWS oder die VWS im Gesamten implementieren (Ye & Hong 2019; Sakurada et al. 2021, sowie Tabelle E.4). Die VWS erfährt breite Aufmerksamkeit aus der Forschung. In Standardisierungsliteratur für MES ist die VWS bereits als CPS-Kooperationspartner von MES zur gemeinsamen Prozessdurchführung vorgesehen (VDI 5600-7).

Tantik & Anderl 2018 griffen die VWS-Struktur als Konzept für eine softwaredefinierte technische Plattform für Automatisierungsanwendungen auf und zeigt einen entsprechenden Ansatz, bei dem die VWS als Ausführungshülle für Automatisierungsanwendungen

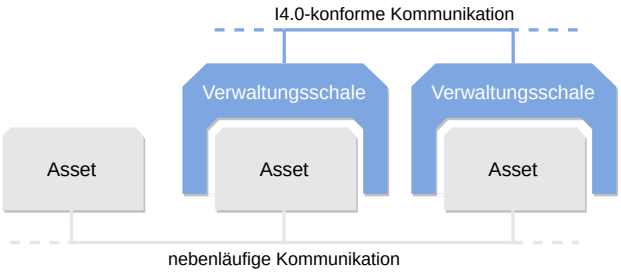


Abbildung 3.8: Kommunikation der VWS und des Assets (DIN SPEC 91345)

dient. Die Ausführungshülle besitzt im Konzept Schnittstellen nach innen (Asset/Betriebsmittel) und nach außen (VWS-Kommunikation). Eine zustandsorientierte generalisierte Softwarearchitektur für Steuerungssysteme, deren Bestandteile auf die der VWS-Struktur kartiert werden können, wird in Breunig et al. 2020 vorgestellt. VWS-Teilmodelle werden durch Softwarekomponenten auf dem Steuerungssystem repräsentiert. Diese Komponenten arbeiten im Konzept auf einer gemeinsamen Datenbasis, um eine Zustandsorientierung zu erleichtern. Tantik & Anderl 2018 und Breunig et al. 2020 setzen auf der ersten VWS-Struktur auf, in der die VWS in Body- und Header unterteilt wurden.

Sakurada et al. 2021 diskutierten eine Reihe von bisherigen Arbeiten zur VWS und untersucht sie auf I4.0-Tauglichkeit. Die größte Gemeinsamkeit der untersuchten Arbeiten liegt in der Nutzung von OPC UA als gemeinsame Technologie. Die VWS wird vorrangig in der passiven Form zur Beschreibung genutzt. Implementierungen der aktiven VWS, beispielsweise in Form eines Multi-Agent-Systems, werden als Forschungsfeld benannt. Die Verbindung mit Multi-Agent-Systemen über Adapter in Agenten oder in die VWS integrierten Agenten wurde erfolgreich untersucht (beispielsweise Karnouskos et al. 2019; López et al. 2020). Die Verbindung von agentenbasierten Systemen und VWS wurde als Integrationsansatz automatisierter Altsysteme in moderne Umgebungen vorgeschlagen (López et al. 2020).

VWS können isoliert von der RAMI-4.0-Abstraktion in der Pyramidenabstraktion weitergenutzt werden, indem die technischen Artefakte der Ebenen mittels eigener VWS repräsentiert und integriert werden. Auch ERP und MES sind einhüllbar in der VWS. (Ye et al. 2021) VWS-Elemente sind mittels AutomationML beschreibbar (Liu et al. 2020). Der

VWS-Aufbau kann als Schritt in die Systementwicklung integriert werden: Informationen werden aus Werkzeugen automatisiert in das VWS-Modell eingebettet und begleiten dann als Grundmodell die I4.0-Komponente ab Auslieferung. (Göppert et al. 2021)

Eine Integration der VWS mit bekannten Steuerungssystemen ist möglich, zum einen durch die Einbettung von Steuerungssystemen in VWS, zum anderen durch die Herstellung von Verbindungsfähigkeit zwischen Steuerungssystemen und VWS-Strukturen. Cavalieri & Salafia 2020 zeigten die Nutzung der VWS zur Repräsentation von IEC-61131-Systemen. Steuerungsprogramme werden als Teilmodelle abgebildet. IEC-61131-Steuerungssysteme sind dadurch im VWS-Format digital abgebildet und können in RAMI-4.0-Gesamtarchitekturen eingebettet werden. Gampig et al. 2021 zeigten die Implementierung von VWS-gerechter Konnektivität in IEC-61131- und IEC-61499-Funktionsblöcke. Ein Repräsentations- und Datenaustausch zwischen Steuerungssystemen und VWS kann dadurch in Steuerungsprogrammen implementiert werden. Zugrundeliegende Protokolle sind OPC UA und HTTP/REST.

3.2.4 Distributed Object Model Environment

Die *Distributed Object Model Environment*, kurz DOME, ist ein objektorientiertes und modellgetriebenes System zur Engineering verteilter Automatisierungssysteme (Riedl 2005). Im DOME-Modell werden für die Automatisierung relevante Betriebsmittel wie Rechner oder Steuerungen als Knoten betrachtet. Die Prozesse, die Teil einer modellbasiert und objektorientiert entwickelten Gesamtapplikationen sind, werden auf die beteiligten Knoten verteilt und dort durch eine Laufzeitumgebung verwaltet und ausgeführt. (Riedl 2005; Riedl et al. 2011; Riedl et al. 2004) Es ist keine industrielle Verwendung bekannt.

Automatisierungssysteme heben sich gegenüber anderen digitalen Systemen durch ihre stark verteilte Entwicklung ab: in einem System wird eine große Zahl Grundkomponenten verschiedener Hersteller eingesetzt, die sich in Technologien und ihrem Aufwandsabstand zur Anwendung unterscheiden und ihre gemeinsam realisierte Prozessfähigkeit erst durch Integration und Programmierung nach Auslieferung und ohne Herstellerbeteiligung erhalten. DOME begegnet dieser Eigenheit mit einer objektorientierten Beschaffenheit. Dadurch können Komponenten als Objekte kartiert direkt von der Systemtopologie in das Softwaremodell übernommen werden und Konfiguration und Programmierung sind auftrennbar,

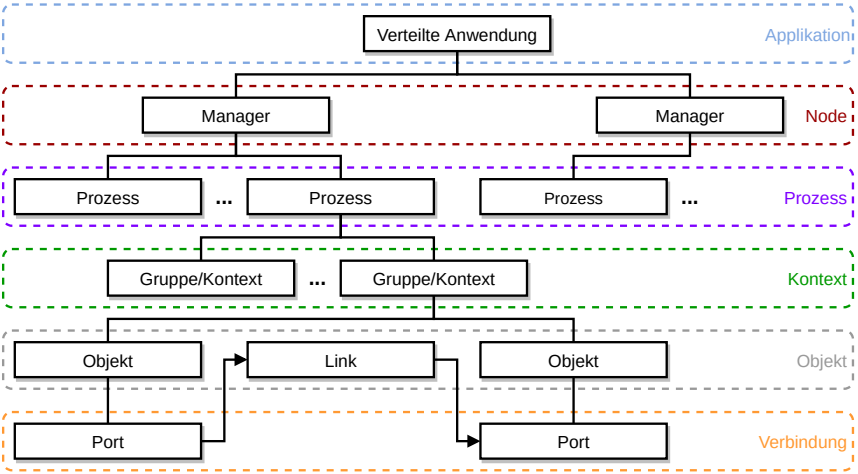


Abbildung 3.9: Die Schichten einer implementierten DOME-Architektur (nach Riedl et al. 2011)

sodass sie bei der Verteilung auf die Objekte des Gesamtsystems aufgeteilt werden können. Die Kommunikation zwischen Objekten im DOME-Modell kann synchron oder asynchron erfolgen und ist in beiden Fällen ereignisgetrieben (Riedl et al. 2013), wobei ein Ereignis auch zyklisch anfallen kann, was zur Datenweitergabe ohne vorherige Wertänderung führt. Eine datenbasierte Integration oder Repräsentation über ein Informationsmodell existiert nicht explizit. Die Repräsentation nach außen muss über ein entsprechend befähigtes Programmobjekt durch den Steuerungsprogrammierer implementiert werden.

DOME besitzt keine speziellen Funktionen für die Planung der Ausführung ihrer Softwareobjekte. Die vorstellende Dissertation von Riedl 2005 erläutert die Ansätze zur Ausführungssteuerung gesondert, welche allerdings nur beschränkt für ein verteiltes System anwendbar sind. Objekte, aus denen ein Programm besteht, sind gruppierbar. Innerhalb einer Gruppe basiert die Ausführungsfolge auf dem schrittweisen Ablauf, der sich durch die sequentielle Verknüpfung der Objekte ergibt. Bei der parallelen Ausführung mehrerer Gruppen wird auf das darunterliegende Betriebssystem verwiesen, über das eine parallele Ausführung der Gruppen zu realisieren ist, die extern jeweils notwendige Prioritäten zugewiesen bekommen müssen. Alternativ kann über ein allen gemeines Objekt über Objektdaten oder -ausgänge eine sequentielle Aktivierung in das Programm implementiert werden.

Ein DOME-Anwendungsmodell hat sechs Schichten (siehe Abbildung 3.9):

1. **Application Layer:** Gesamte verteilte Applikation als einzelnes Objekt
2. **Node Layer:** Die Manager zur Verwaltung der technischen Systeme (Nodes), die von der Applikation adressiert werden
3. **Process Layer:** Die einer Applikation zugehörigen Prozesse auf einem Node (beispielsweise Programminstanzen)
4. **Context Layer:** Organisation von Kontextdaten und -objekten für Prozesse
5. **Object Layer:** Objekte, die Teil eines Kontexts sind. Dazu zählen Verbindungsinstanzen (*Links*) und einfache sowie komplexe Prozessdaten (*ClassObjects*, *Groups*)
6. **Connection Layer:** Objekte zur Beschreibung der Verbindungen zwischen Verbindungs- und Prozessobjekten

DOME wurde für Unix- und Windowsssysteme implementiert und ist auf Anfrage erhältlich[1].

3.2.5 AUTOSAR

AUTOSAR ist eine Sammlung offener Standards für die modellbasierte Entwicklung von Steuerelektronik im Kraftfahrzeugbereich. An AUTOSAR sind unter anderem Kraftfahrzeughersteller und -zulieferer, aber auch Entwickler aus dem akademischen Bereich beteiligt. Ziel ist es, standardisierte Software und dazugehörende Technologien und Methoden für Steuergeräte zu erstellen und zu beschreiben. AUTOSAR gab 2005 eine erste Version frei. (Martínez-Fernandez et al. 2015) Inzwischen existieren zwei verschiedene AUTOSAR-Frameworks: *AUTOSAR Classic Platform* (AUTOSAR-CP) und *AUTOSAR Adaptive Platform* (AUTOSAR-AP). Bei beiden Frameworks sind für Implementierungen C und C++ vorgesehen, wobei für die AUTOSAR-AP C++11 angestrebt wird. AUTOSAR-CP ist auf die Nutzung für kleine, hart echtzeitfähige Steuergeräte ausgelegt; AUTOSAR-AP auf die Nutzung als Basis komplexer Softwaresysteme für automatisiertes Fahren auf großen Steuergeräten.

[1] siehe https://www.ifak-ts.com/pf/ifak-dome/, Abruf am 20.9.2023

Ein wesentlicher Unterschied liegt in der Laufzeitrekonfigurierbarkeit der AUTOSAR-AP, die bei der AUTOSAR-CP nicht vorhanden ist. (AUTOSAR 2021) AUTOSAR über Anpassung der Laufzeitumgebung und anderer Systemteile, auch auf Mehrkernprozessoren, deterministisch ausgeführt werden. Dazu werden deterministische Paradigmen zur Kommunikation und zur Ausführungsplanung eingeführt (Gemlau et al. 2021; Menard et al. 2020).

AUTOSAR ist aufgrund seiner grundsätzlichen Determinismusfähigkeit, der Rekonfigurierbarkeit, Interoperabilität, der modellgetriebenen Entwicklung und der Ausrichtung von AUTOSAR-CP auf kostengünstige Systeme interessant für die Anwendung in der Industrieautomatisierung (Schmidt et al. 2016). Für die Nutzung von AUTOSAR-Implementierungen ohne automatisierungsspezifischen Neuaufbau können AUTOSAR-Middlewares mit Automatisierungsprotokollen wie OPC UA verbunden werden (Ioana & Korodi 2019). Etablierte Implementierungen in der Automatisierung sind nicht bekannt.

Martínez-Fernandez et al. 2015 stellten eine Studie unter AUTOSAR-Nutzern und -Beteiligten vor. Als vorrangige Vorteile der AUTOSAR-CP werden die eingeführte Standardisierung, die darüber realisierbare Interoperabilität und die Wiederverwendbarkeit der Entwicklungen aufgeführt. Dem gegenüber stehen besonders stark die Komplexität, der Initialaufwand und die Abhängigkeit von AUTOSAR-CP. Die AUTOSAR-Dokumentation umfasst eine zweistellige Anzahl Dokumente mit mehreren hundert Seiten, was ein umfassendes Verständnis erschwert. Neue Designansätze, die durch AUTOSAR-CP möglich würden, werden von den Befragten als Vorteil nur selten genannt.

Classic Platform

Ein AUTOSAR-CP-System besteht aus drei Softwareschichten (AUTOSAR 2017a):

- Die **Basissoftware** ist hardwaregebunden und stellt abstrahierte Schnittstellen und eine standardisierte Umgebung für die darüberliegende Laufzeitumgebung bereit. Über die Basissoftware wird auf Ein- und Ausgänge und Kommunikationsschnittstellen zugegriffen. Für die Ausführung und Synchronisation der Basissoftware ist ein Modul zur Ausführungsplanung (Scheduler) vorgesehen, das allerdings nicht eigenständig sein muss, sondern je nach Hardware Teil der Laufzeitumgebung sein kann.

- Die **Laufzeitumgebung** wird für jedes Steuergerät anhand der in der Konfiguration definierten Modelle erzeugt und enthält Funktionen und Ausführungsplanung.

- **Softwarekomponenten** sind funktionserfüllende, eigenständige Softwaremodule, die von der Laufzeitumgebung ausgeführt und gesteuert werden. Sie werden in *Application software-components* und *Sensor-actuator components* unterschieden. Erstere greifen nicht auf Hardwarefunktionen zu, sondern erfüllen rein virtuelle Funktionen. Zweitere implementieren EA-Funktionen.

Eine entstehende Architektur besteht aus integrierten technischen Teilen: Auf einem Steuergerät stellt hardwarespezifische Basissoftware eine Ausführungsplattform für die AUTOSAR-Classic-Laufzeitumgebung bereit. Die Laufzeitumgebung implementiert standardisierte Schnittstellen, auf der Softwarekomponenten aufbauen. Mit den Softwarekomponenten werden die Steuergerätefunktionen realisiert. Über ein Bussystem kommuniziert das Steuergerät mit anderen Geräten.

Bei einer Systemmodellierung werden bei der AUTOSAR-Classic-Plattform aus einzelnen Softwarekomponenten Lösungen zusammengestellt und über ein virtuelles Bussystem verknüpft (Abbildung 3.10). Das Bussystem existiert nur zur Programmierzeit und definiert unter anderem Datenverknüpfungen und Datentypumwandlungen. Die Softwarekomponenten werden den verschiedenen Steuergeräten eines Kraftfahrzeugs zugewiesen und bei der Verteilung auf die zugewiesenen Geräte installiert. Die im Bussystem definierten Verbindungen werden aufgelöst und je nach Bedarf über die geräteinterne und plattformeigene Laufzeitumgebung realisiert oder über Basissoftwaremodule (z.B. für CAN-Kommmunikation) zu anderen Geräten aufgebaut. Geräteinterne Komponentenverbindungen funktionieren nach dem Sender-Empfänger- oder dem Client-Server-Modell, über gemeinsamen Speicher oder über Moduszuweisungen. Komponentenverbindungen nach außen richten sich nach dem jeweiligen Basissoftware-Kommunikationsstack. (AUTOSAR 2017a; AUTOSAR 2017b)

AUTOSAR-CP sieht keine Veränderung der Anwendungs- oder Gerätestruktur zur Laufzeit vor. Gestaltung oder Änderung finden während der Systemkonfiguration statt. Zum einen dient es als standardisierte Plattform zur Ausführung von portierbaren Softwarekomponenten auf Steuerungssysteme, die aus mehreren Geräten bestehen. Zum anderen besteht die Motivation für AUTOSAR-CP in der Bereitstellung einer modellgetriebenen Entwicklungsplattform für Steuerungssysteme, in der abstrahiert und modifizierbar Steuerungsanwendungen gestaltet werden können. Da AUTOSAR-CP taskbasiert arbeitet, können deterministische Systeme erstellt werden. (Menard et al. 2020)

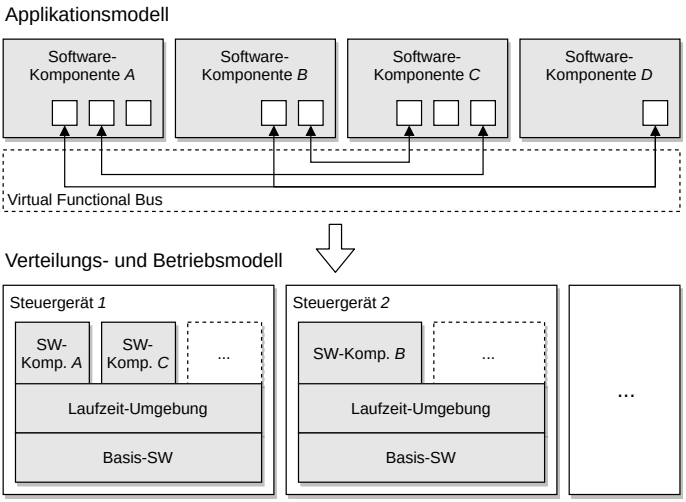


Abbildung 3.10: AUTOSAR-Classic-Plattform-Strukturmodelle zur Definitions- und Laufzeit nach AUTOSAR 2017b; die Modellbezeichnungen sind für die Arbeit gewählt

Adaptive Platform

Die AUTOSAR-AP ist eine SoA-Plattform, die auf die Nutzung von Threading und POSIX ausgelegt ist. Sie baut auf der früher veröffentlichten AUTOSAR-CP auf. Aufgrund der POSIX-Nutzung ist AUTOSAR-AP zur Nutzung in Verbindung mit oder auf Unix-Systemen ausgerichtet. Die Standardisierung macht keine Vorgabe zur technischen Umsetzung der Kommunikation zwischen Softwaremodulen. (Fürst & Bechter 2016; Menard et al. 2020) Softwareteile der AUTOSAR-CP sind mit der AUTOSAR-AP gemein („Foundation") AUTOSAR 2021. Die Laufzeitumgebung und die Softwarebasis werden zur rekonfigurierbaren Ausführung von Softwareservices und Applikationen erweitert. Die Instanziierung eines AUTOSAR-AP-Systems auf einer virtuellen technischen Plattform wird vorgesehen, was einen bedeutenden Unterschied zum AUTOSAR-CP-Ansatz darstellt, weil die 1-1-Bindung zwischen Hardwareplattform und AUTOSAR-Instanz aufgebrochen wird. (Wagner et al. 2014) Modellierung und Grundstruktur sind gleich zur AUTOSAR-CP, die Softwarekomponenten werden aber zur Laufzeit integriert.

AUTOSAR-AP ist im Gegensatz zu AUTOSAR-CP nichtdeterministisch, kann aber entsprechend angepasst werden. Aufgrund der nicht-statischen Systemstruktur bei Softwarekom-

ponenten, Ausführungsplanung und Kommunikationsdurchführung muss das Framework erweitert werden, beispielsweise mit Reaktionsfunktionen, um Zustandsänderungen von Eingängen oder Daten zu verarbeiten, und mit festen Ausführungskapazitäten für nichtvorhersehbare Ereignisse. (Menard et al. 2020)

3.3 Adaptierte allgemeine Ansätze

Die Industrieautomatisierung entwickelt sich aufgrund stärkerer Vernetzung, Komponentendigitalisierung sowie softwaredefinierter Abläufe und Integration hin zu den Softwaresystemen der Informatik und allgemeinen Datenverarbeitung, deren Paradigmen und Technologien im Umkehrschluss für die Automatisierung untersucht und genutzt werden können. Besondere Ansätze, die im Rahmen der Arbeit hierbei einfließen, sind die Modularisierung von Systemen mittels SoA, der Einbezug externer Plattformen im Automatisierungskontext, die Verwendung von Softwarecontainern zur Aufteilung und Auslieferung von Software, sowie die Softwareentwicklung mittels *DevOps*, *Model-driven Engineering* und *System-of-Systems-Engineering*. Die Ansätze werden nicht direkt eingebettet, sondern dienen als Orientierung.

3.3.1 Modularisierung über Serviceorientierung

Die Umsetzung von technischen Systemen als Serviceorientierte Architekturen (SoA) ist ein Ansatz zur funktionsorientierten Aufteilung der Logik monolithischer Softwarelösungen in voneinander abgeschlossene Komponenten, die als *Service* oder *Dienst* einzelne, anwendungsunabhängige technische Funktionen implementieren. Die Implementierung der Software für einen Anwendungsfall entsteht in einer SoA durch die Verschaltung der Dienste über ihre Schnittstellen in einer für den Anwendungsfall spezifischen Kombination. SoA sind ein bedeutender und produktiv etablierter Ansatz für die Gestaltung verteilter Systeme. Sie sind aufgrund ihrer Verbreitung, der für Software starken Definitionen und der demonstrierten Adaptierbarkeit für Automatisierungssysteme bedeutend.

Ein Dienst ist der kleinste Teil einer SoA. Dienste besitzen grundsätzliche Eigenschaften (Dai et al. 2014): Wiederverwendbarkeit, formalisierte Beschreibung, lose Kopplung, Komplexitätsabstraktion, Kombinierbarkeit, Autonomie/Autarkie, Zustandslosigkeit und

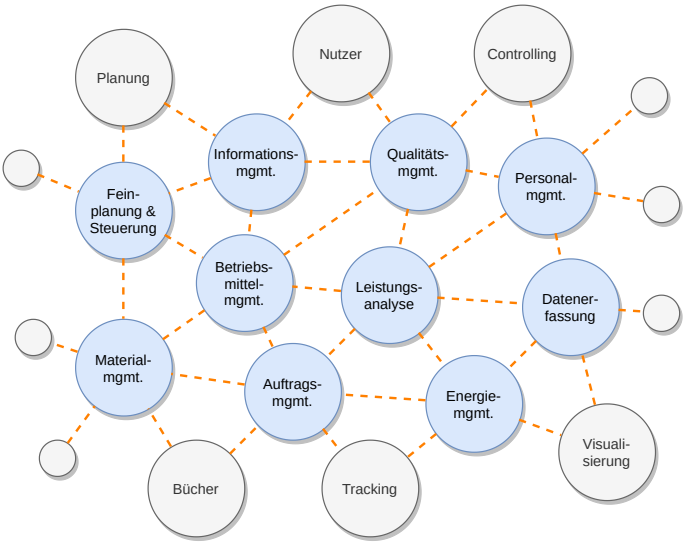


Abbildung 3.11: MES aufgelöst in applikationsorientierte Dienste (blau) nach VDI 5600-7

Entdeckbarkeit (siehe Anhang E.2). Die Schnittstellen der Dienste sind innerhalb einer SoA standardisiert. Diese Eigenschaften sind aufgreifbar für die Architekturentwicklung verteilter Systeme. Es sind nicht immer alle der genannten Eigenschaften vorhanden. Im Falle, dass (steuernde) Applikationen als Dienste eingebunden werden, ist die Zustandslosigkeit nicht immer gegeben. Implementierungen, die Laufzeitumgebungen als ausführende Basis nutzen (beispielsweise IEC-61499-SoA-Konzepte), können neben der Zustandslosigkeit auch die lose Kopplung nicht garantieren. Um die Umsetzung der Eigenschaften zu erleichtern oder zu ermöglichen, sind SoA meistens asynchron und ereignisorientiert (siehe Abschnitt A.1.2).

Aufgrund der unkritischen Zeitanforderungen ist der Ersatz monolithischer ERP und MES durch SoA etabliert (VDI 5600-7; Bloch et al. 2016). Der servicebasierte Aufbau eines MES nach VDI 5600-7 wird in Abbildung 3.11 gezeigt. Für SoA existieren dedizierte Integrationsplattformen, über die verteilte Dienste einen zentralen Nachrichtenaustausch und eine logische Integration erhalten, beispielsweise als *Enterprise Service Bus* (Schmidt et al. 2005; Schill & Springer 2012). Diese Integrationsplattformen sind für den Steuerungsbereich untersucht worden (Ashiwal & Zoitl 2021).

SoA wurden für die hardwaregebundene Nutzung adaptiert, um auf Steuer- und Automatisierungssystemen verwendet werden zu können. Durch die technischen Randbedingungen (Leistungsfähigkeit) und funktionalen Anforderungen (Zugriff auf Ein- und Ausgänge) unterliegen Geräte-SoA zusätzlichen Anforderungen. Kernziel der Geräte-SoA ist die Einführung einer laufzeitorientierten Rekonfigurierbarkeit des Geräts. (Delamer & Lastra 2007; Lam et al. 2022) Erhalten bleiben die SoA-Kernparadigmen hinsichtlich loser Kopplung, Selbstbeschreibung, Entdeckbarkeit, abgeschlossener Struktur, Zustandslosigkeit et cetera. SoA ist ein möglicher Ansatz zur Rekonfigurierbarkeitserweiterung von normalerweise zur Laufzeit starr definierter Architekturen wie AUTOSAR-Systemen (Wagner et al. 2014; Schmidt et al. 2016). Dediziert gerätebezogene SoA sind SIRENA und Arrowhead. Bei beiden SoA wird auf einem Parallelbetrieb von administrationsorientierter Kommunikation auf Standardnetzwerkbasis und deterministischen Kommunikationsprotokollen auf Sondertechnologien gebaut. Sie umfassen in der Implementierung, neben Softwarerahmenstrukturen für Dienste Werkzeuge für Entwicklung, Verwaltung und Deployment der Dienste sowie Basisdienste, die im Sinne einer lokalen Dienste-Cloud immer betrieben werden müssen. (Jammes & Smit 2005; Bohn et al. 2006; Hästbacka et al. 2022) Auf den deterministischen und prozessnahen Betrieb ist die Aufteilung in Dienste nicht ausgelegt. In den Geräte-SoA werden für kritische Anwendungen zumeist deterministische Umgebungen (üblicherweise nach IEC 61131) in einem Dienst gekapselt.

Die IEC 61499 (siehe Abschnitt 3.2.1) wurde mit SoA integriert. Durch die graphische Gestaltung von Programmen mit ereignisorientierter Verknüpfung, abgeschlossenen funktionalen Entitäten (Funktionsblöcke) und der nativen Verteilung einer Anwendung auf verschiedene Maschinen, besitzt die IEC 61499 direkt auf SoA kartierbare Ansätze. (Dai et al. 2015; Dai et al. 2014; Breunig & Götz 2018). Funktionsblöcke repräsentieren hierbei Dienste einer SoA direkt oder über eine Dienstablage (beispielsweise Dai et al. 2014) oder implementieren Schnittstellen zu Diensten (beispielsweise Breunig & Götz 2018).

3.3.2 Einbezug externer Plattformen

Über Endverbrauchersysteme wurden externe, über das Internet global verfügbare Plattformen etabliert. Über softwaredefinierte Strukturen werden Server an einem oder mehreren Standorten zu digitalen Plattformen, unter anderem für die Bereitstellung von Speicherplatz, Webseiten, Datenbanken und allgemeinen Ausführungsumgebungen für Software, zusammengeschlossen. Die Nutzung solcher externer Plattformen wird ebenfalls für die Produktion ausgeweitet, um betriebswirtschaftliche und allgemeine Anwendungen für die Betriebsleittechnik zu stützen, und um Produktionssysteme um die wesentlich größeren Fähigkeiten der Plattformen bei Speicherplatz und Rechenkapazität sowie um rein virtuelle Systeme zu erweitern. Cloudplattformen bieten aufgrund ihrer üblichen Grundtechnologien eine Umsetzungsmöglichkeit rekonfigurierbarer, spezifischer Softwaresysteme für Produktionsumgebungen (Stock et al. 2014). Bei ERP sind cloudbasierte Systeme oder cloudbasierte Erweiterungen etabliert und eine wesentliche Weiterentwicklungsgrundlage (Prasetyo & Soliman 2021; Križanic et al. 2020). Die Erweiterung von CPS mit Cloudsystemen wurden früh im Kontext mit Systemen aus der Unternehmens- und Betriebsleitung verfolgt (Bangemann et al. 2016), noch ohne direkt produktionsbezogene Systeme einzubeziehen. Bei Cloudplattformen, ebenso wie bei der Bereitstellung von einzelnen technischen Plattformen wie Servern, erfolgt die Kommunikation stark anlassorientiert.

Die einfachste Integration eines lokalen Systems in der Produktionstechnik mit einer externen Plattform ist die pauschale Übermittlung von Daten ohne Steuerungsbezug über ein definiertes, gemeinsames Protokoll, die vorrangig zur Datensammlung, Überwachung und teilweise zur Parameterdefinition dient, wie in Pallasch et al. 2018b gezeigt. Arbeiten zur weiteren Integration, etwa von Brecher et al. 2019, zeigen die Verknüpfung von einfachen EA-Systemen mit externen Plattformen über plattformfähige Protokolle mit kleinen Systemfähigkeiten seitens der EA-Systeme. Die Ein- und Ausgänge wurden hierbei vorrangig oder ausschließlich durch Software in der Plattform gelesen und gestellt. Technisch sind komplexe Steuerungsvorgänge aus Cloudplattformen untersucht worden, mit dedizierten eigenen Ansätzen (siehe Lyu et al. 2019) oder unter Verwendung bekannter Technologien (Virtualisierung von IEC-61131-Systemen, siehe Goldschmidt et al. 2015). Vorgänge mit weichen Echtzeitanforderungen sind realisierbar (Goldschmidt et al. 2015).

3.3.3 Containerarchitekturen

Containerarchitekturen sind ein bedeutsamer neuer Ansatz für die Unterstützung der Modularisierung von Software und der Auslieferung von Software über Deploymentfunktionen. Container bezeichnen Softwarekapseln, die eine oder mehrere Zielanwendungen, notwendige Basissoftware (beispielsweise Bibliotheken) sowie für ein ausgewähltes Betriebssystem spezifische Softwareteile enthalten. Im Gegensatz zu Standardsystemen sowie virtuellen Maschinen enthält ein Container kein eigenes Betriebssystem. Container erhalten Rechenzeit über den Kernel des Plattformbetriebssystems. Sie gewährleisten die Begleitung von Zielanwendungen mit notwendiger Software und führen eine Separierung zwischen Zielanwendungen ein. Containerbasierte Anwendungen benötigen weniger Ressourcen als Anwendungen, gekapselt in virtuellen Maschinen und sind im Gegensatz zu virtuellen Maschinen kaum langsamer als Anwendungen direkt auf einem Betriebssystem (Felter et al. 2015; Gundall et al. 2020a). Docker auf Linuxsystemen ist der zentrale Ansatz zur Umsetzung von containerbasierten Systemen (Gundall et al. 2020a; Sollfrank et al. 2021). Zur Erstellung von Containerimages wird eine Betriebssystemreferenzumgebung festgelegt, die innerhalb des Containers bereitgestellt wird. Anschließend werden datei- und befehlsbasiert Anwendungen in das Containerimage eingebettet. Das Image kann dann in Containerumgebungen instanziiert werden.

Die Containermodularisierung wird für die Steuerungstechnik der Industrieautomatisierung aufgegriffen (Gundall et al. 2021; Gundall et al. 2020a; Goldschmidt et al. 2018; Tasci et al. 2018; Sollfrank et al. 2021; Dupont et al. 2017). Container sind grundsätzlich echtzeitfähige Technologie im Sinne der Industrieautomatisierung (Gundall et al. 2020a). Die aufwandsarme Migration, die auch zwischen stark unterschiedlichen Ausführungsplattformen möglich ist, ist ein Hauptvorteil der Containertechnologien (Dupont et al. 2017), zudem erleichtert sie potentiell Konfiguration und Rekonfiguration der Steuerungstechnik über die Einführung der Deploymentfähigkeit. Im Fokus der Forschung steht die Untersuchung der Determinismusmöglichkeiten. Tasci et al. 2018 demonstrierten, dass Containerarchitekturen auf Linuxsystemen mit für Determinismus angepasstem Standardkernel und auf Linuxsystemen mit zusätzlichem Echtzeitkernel ausführbar sind. Möglichkeiten zur Implementierung zum Hardwarezugriff durch Containeranwendungen werden besprochen, eine Festlegung aber nicht gemacht.

Goldschmidt et al. 2018 stellten eine Prinziparchitektur zur Nutzung von Containerumgebungen auf eingebetteten Systemen vor, als Ersatz für übliche SPS mit Zykluszeiten von 100 bis 1000 ms bei etwa zehn Prozent Schwankungen in der Zykluszeit (Jitter). Erzielte Zykluszeiten der gekapselten Anwendungen befinden sich im zweistelligen Mikrosekundenbereich, etwa zehn Prozent höher als bei Anwendungen direkt auf der technischen Plattform. Sollfrank et al. 2021 untersucht die Ausführung und Kommunikation von Instanzen einer C-Anwendung auf eingebetteten Systemen, direkt auf den Maschinen sowie gekapselt in Containern. Über die Ermittlung von mittleren Paketlaufzeiten und deren Standardabweichungen wird eine Ansatzbewertung vorgenommen. Die Kapselung verlangsamt die Ausführung und Kommunikation auf einzelnen und vernetzten Systemen in vertretbarem und abschätzbarem Maß.

3.3.4 CPS- und System-Engineering

CPS sind softwaregetriebene änderbare Systeme (Dafflon et al. 2021). Neben den Softwaretechnologien sind dadurch auch Entwicklungselemente der Softwarewelt interessant für die Weiterentwicklung der Industrieautomatisierung. Aufgrund immer kürzerer Produktentwicklungszeiten bei wachsender Softwarevielfalt werden für die Softwareentwicklung neue Vorgehensmodelle entwickelt. Aufgreifbar für die CPS-Automatisierung ist besonders die Integration von Softwareentwicklung (*Development*) und Softwarebetrieb (*Operations*) als *DevOps*, da ein zentrales Werkzeug des DevOps-Ansatzes die Rückführung von Laufzeiterfahrungen in die Softwareentwicklung ist (Stepien et al. 2022; Koren et al. 2023). Weitere bedeutende Werkzeuge des DevOps-Ansatzes sind die kontinuierliche Integration von neu- oder weiterentwickelten Softwarekomponenten in erstellte Software als *Continuous Integration* (CI) sowie die kontinuierliche Auslieferung von neu- oder weiterentwickelten Softwarekomponenten (CD) (Mowad et al. 2022). Mit DevOps kann die Softwarequalität trotz schneller und häufiger Änderungen der Softwarestruktur verbessert werden (Mowad et al. 2022; Ebert & Hochstein 2023). DevOps ist auf technologische Unterstützung angewiesen: Plattformen und Systeme für die Entwicklung müssen mit Plattformen und Systemen für den Betrieb zusammengeführt werden. Zudem muss Software auf die mit DevOps verwalteten Zielsysteme auslieferbar sein. DevOps ist für den hardwarenahen Einsatz

mit physischen Einrichtungen verwendbar, muss aber erweitert werden (Koren et al. 2023; Ebert & Hochstein 2023).

Erweiterungen müssen ein Vorgehen wie DevOps mit Simulations- und Validierungsmöglichkeiten für Softwareimplementierungen versehen, um das kombinierte Verhalten von Software und physischer Einrichtung transparent zu machen. Für Produktions-CPS ist diese Transparenz kritisch um Schäden zu verhindern und Qualitäten zu sichern. Zentraler Ansatz ist das Konzept des digitalen Zwillings. (Jones et al. 2020; Ejersbo et al. 2023; Zhang et al. 2023; Koren et al. 2023) Der Begriff wird eng verbunden mit CPS gesehen (Zhang et al. 2023), ist jedoch nicht fest definiert (Jones et al. 2020). Allen Ansätzen zum digitalen Zwilling gemein ist die Erstellung eines virtuellen Abbilds eines CPS. Umfang und Erstellungszeitpunkt (während Laufzeit, nach Auslieferung, während Entwicklung, ...) variieren je nach Ansatz. Ejersbo et al. 2023 sieht eine zur Laufzeit regelmäßige Abbilderstellung in Verbindung mit Rückführung von Änderungen auf das CPS zur Laufzeit vor, wodurch der digitale Zwilling neben Simulations- und Validierungsmöglichkeit auch Entwicklungsbasis ist. Für einen solchen und für die CPS-Automatisierung bedeutenden Ansatz müssen die zusammengehörenden Strukturen von Abbild und CPS bidirektional genutzt werden können (Jones et al. 2020).

CPS sind komplexe technische Systeme. Sie sind aufgrund der Verbindung von Hardware, Software, Vernetzung und physischem Prozess eine interdisziplinäre Systementwicklung angewiesen (Dafflon et al. 2021; Koren et al. 2023; Weyns et al. 2021). Sie integrieren Rekonfigurationsfunktionen und verfügen damit über eine änderbare Struktur (Dafflon et al. 2021). Bisherige Engineeringansätze müssen für CPS ersetzt oder weiterentwickelt werden (Weyns et al. 2021). Da CPS in ihren Architekturen als *Systems-of-systems* (SoS) gesehen werden können (Hoffmann et al. 2021), sind Ansätze für das SoS-Engineering verwendbar sowie zugrundeliegende Konzepte aus dem *Model-driven Engineering* (MDE) (Binder et al. 2023; Weyns et al. 2021). Das MDE für moderne softwaredefinierte Produktionssysteme muss bedingt durch die Interdisziplinarität ein breites Spektrum an Anforderungen mit Modellen und Modellentwicklung abgedeckt werden. Hinzu kommt die Notwendigkeit neue Erkenntnisse oder Anforderungen zur Laufzeit berücksichtigen zu können. (Masior et al. 2020) Modelle in MDE können mit Referenzarchitekturen für die Industrieautomatisierung verbunden werden (Binder et al. 2023), beispielsweise um den Modelltransfer

in standardisierte Repräsentationsformate zu ermöglichen. Das MDE deckt ein System vor allem in der technischen Beschaffenheit, aufgeteilt nach Domänen ab. In umfassenderen SoS-Engineeringansätzen wie dem *Advanced System Engineering* nach Albers & Lohmeyer 2012 erfährt ein CPS zusätzliche Architekturbetrachtungen, beispielsweise hinsichtlich des funktionalen oder strukturellen Aufbaus.

3.4 Diskussion

Es folgt eine Diskussion der betrachteten Ansätze für die Modernisierung der Industrieautomatisierung. Für die Arbeit relevante Eckpunkte sind fett hervorgehoben.

Abstraktionssysteme

Die Entwicklung der Automatisierung ist mit den Abstraktionssystemen verbunden, die an Stelle der Automatisierungspyramide treten sollen. Es wird die technische Auflösung der Hierarchie (siehe Abschnitt 1.7, Abbildung 1.5) in knotenbasierte Strukturen diskutiert, die grundsätzliche Erweiterung der Automatisierung nach außen (Abbildung 3.1) und neue Abstraktionen der Produktionsmittel mit Systemen wie der RAMI4.0-Abstraktion (Abbildung 3.7).

Für die automatisierungsbezogenen Systeme bedeuten die Abstraktionen und ihre Motivationen eine notwendige **Ausweitung der Kommunikations- und Integrationsfähigkeiten** für ihre Entwicklung, weil das Anforderungsfeld hinsichtlich Schnittstellen, Kommunikationsformaten und Integrationslogiken vergrößert wird. Für die Abstraktion des RAMI4.0 erweitert sich die bislang rein aufgabenbezogene Kommunikation um **systembezogene Metadaten**, um die Assetbeschreibung zu ermöglichen, beispielsweise durch Systembeschreibungen. Kommunikation und Integration mit externen Systemen der bislang höheren Ebenen der Automatisierungspyramide beziehen sich auf abstrakte, im Vergleich zu Prozessdaten unkritische Daten, ohne direkten technischen Bezug, wofür **anlassbezogene Kommunikation** gegenüber kontinuierlicher Kommunikation ausreichend ist. Im Falle der Auflösung der Pyramide und der notwendigen (Re)Konfigurationsfähigkeit der Kommunikation, abhängig von Produktions- und Automatisierungsaufgaben, ist

eine **adaptierbare Kommunikations- und Integrationsfähigkeit** über alle Systeme notwendig.

Technische Frameworks

Die abstrakten Kernparadigmen cyberphysischer Systeme sind die Kooperation und die ständige Vernetzung, mit denen technische Systeme, die zu einem CPS zusammengeschlossen sind, einen physischen Prozess überwachen, steuern und regeln. Ein Framework, das als Kernaufgaben Bau und Betrieb von CPS hat, muss diese Kernparadigmen in beiden Phasen ermöglichen und unterstützen. Neben den Kernparadigmen muss ein Framework weitere Paradigmen beachten und bei der Lösungsumsetzung ermöglichen, die sich aus dem Einsatzgebiet ergeben oder von der modernen Softwareentwicklung im Allgemeinen gestellt werden. Diese Paradigmen sind die Unterstützung verteilter Systeme, offene und standardisierte Software, Plattformunabhängigkeit, Adaptierbarkeit, Determinismus und Transparenz, interne und externe Schnittstellen, sowie Modell- und Objektorientierung.

Für die Betrachtung existierender Ansätze zur Modernisierung der Industrieautomatisierung werden fünf technische Frameworks (IEC 61499, FDI, VWS, DOME, AUTOSAR) betrachtet, die sich grundsätzlich für die CPS-orientierte Automatisierung eignen könnten, verteilte Systeme unterstützen und öffentlich beschrieben oder implementiert sind. Betrachtungen jedes einzelnen Frameworks sind im Anhang B aufgeführt.

Allen Frameworks gemein ist ein mehr oder weniger stark ausgeprägter **Alleinanspruch in der Nutzung**, den vor allem die Frameworks mit laufzeitaktiven Strukturen (IEC 61499, DOME, AUTOSAR sowie IEC 61131 und DIN 66253) haben. Der Alleinanspruch entsteht, da die Frameworks, um während der Prozessausführung aktiv sein zu können, Anforderungen bei Kommunikationsprotokoll, Programmierung, Ausführungsumgebung und Zusammenhängen zwischen den Systemen machen. Um Geräte, die diese Anforderungen nicht erfüllen und deswegen nicht direkt eingebunden werden können, mit Architekturen auf Basis solcher Frameworks zu integrieren, müssen technische und logische Schnittstellen vorhanden sein. Bei den genannten Frameworks sind solche **Schnittstellen kein nativer Bestandteil, sondern müssen im Format des jeweiligen Programmierelements implementiert** werden und ein solches Programmierelement in ein Programm eingebettet werden. Allgemein muss ein Framework ein Fundament für die Implementierung von Lösungen für

zukünftige, noch unbekannte Probleme bereitstellen. Es besteht immer die Gefahr, dass ein neues Framework die Implementierung bestmöglicher Lösungen nicht erleichtert oder gar nicht erlaubt. Der Ansatz, bereits bei der frühzeitigen Frameworkdefinition alle zukünftigen möglichen Aufgaben zu berücksichtigen, ist aufgrund größenbedingter Komplexität, Aufgabenvariation und -unbekanntheit nicht realisierbar. Die **Frameworkdefinition muss ausreichende Freiheitsgrade in der Anwendung und Erweiterung sowie Schnittstellen bieten**, um dem zu begegnen. Dabei muss trotz dynamischer Strukturen gewährleistet sein, dass hinreichende Betriebssicherheit und Prozesserfüllung umsetzbar sind. (Boelmann et al. 2016)

Die Frameworks besitzen **keine native Funktion zur Prüfung oder Untersuchung des Systemverhaltens**, insbesondere des Zeitverhaltens. Zwar sind IEC-61499-, DOME- und AUTOSAR-CP-Systeme prinzipiell deterministisch, allerdings erfordert die Verhaltensuntersuchung **immer die zusätzliche Erstellung eines Modells oder einer Prüfumgebung außerhalb des Funktionsumfangs** des jeweiligen Systems. Dadurch ergeben sich Zusatzaufwände und Fehleranfälligkeit, da die Entwicklungs- und Modellierungsumgebungen auf PC ausgeführt werden, die sich von den technischen Plattformen in der Feld- und Steuerungsebene unterscheiden. Die Verhaltensmodellierung ist nur möglich, wenn das Verhalten der verwendeten Softwareteile (beispielsweise FB in der IEC 61499) bekannt ist. Die Nutzung abgeschlossener Softwarekomponenten erschwert die Verhaltensvorhersage.

Die modellgetriebenen und laufzeitaktiven Systeme IEC 61499, AUTOSAR und DOME haben strukturelle Ähnlichkeiten bei ihren Modellen hinsichtlich schichtenweisem, hierarchischem Aufbau und der Verwendung von Programm- und Verteilungsmodellen. Ähnliche Strukturen finden sich auch bei der DIN 66253 (siehe Abschnitt 2.3.2). **Konzepte für die Programm- und Verteilungsmodelle** aus den Frameworks **sind aufgreifbar**, müssen aber für bessere Adaptierbarkeit erweitert werden. In den laufzeitaktiven Frameworks wird die Systemvernetzung im Programmmodell definiert, was bei Änderungen tiefe Eingriffe in oder direkt neben der Prozesslogik erfordert. Keines der laufzeitaktiven Frameworks berücksichtigt Umgebungsschnittstellen zwischen CPS und abstrakten Leit- oder anderen Umgebungssystemen. Alle Frameworks sind modellgetrieben, wobei sich die Verwendung und die Modellarchitekturen stark unterscheiden. Vorhanden sind beispielsweise **Modelle für Softwareteile** (hervorzuheben aufgrund des Umfangs ist das FDI-Paketsystem),

auszuführende Anwendungen (insbesondere IEC 61499 und DOME) **und Systemstrukturen** (FDI, AUTOSAR, VWS). Die VWS ist verfügt über eine ausgebaute Modellstruktur zur Systembeschreibung gegenüber externen Systemen. Da die VWS zum Zeitpunkt der Arbeitserstellung große Unterstützung erfährt, ist die VWS-Systembeschreibung als möglicher neuer etablierter Standard für die Repräsentation von Systemen bedeutend.

Standard- und offene Technologien sind essentiell für die Weiterentwicklung verteilter Systeme in der Automatisierung (Poștovei et al. 2020). Existierende Modernisierungsansätze greifen teilweise diese Technologien auf. Bei den Frameworks sind das die Nutzung von Standardprogrammiersprachen, OPC UA für die Informationshaltung und Kommunikation, offene Normen für Struktur und Verteilung (beispielsweise FDT, EDD, VWS), Objektorientierung bei Programmierung und Topologie, Generalisierung von Softwareteilen und Modularisierung sowie starke Modellbasierung. Die Integration dieser technologischen Ansätze variiert in der Ausprägung und ist insgesamt noch nicht ausreichend, weil keines der Frameworks bei allen aufgezählten Ansätzen ausreichend große Umsetzung aufweist. Implementierungen aller Frameworks basieren auf Standardprogrammiersprachen und sind auf Standardplattformen mit Betriebssystemen ausgelegt. Für die Verbindung mit Containern sind die VWS und FDI verwendbar, weil die Elemente ihrer Softwarestrukturen (Teilmodelle sowie Softwarepakete) auf Container kartierbar sind.

Adaptierte allgemeine Ansätze

SoA sind ein Ansatz zur Umsetzung verteilter, modularisierter und rekonfigurierbarer Systeme. Aufgrund ihrer fachlichen Akzeptanz und produktiven Nutzung sind SoA-Ansätze als verwendbar zu betrachten. Die Rekonfiguration und die Modularisierbarkeit in SoA stützen sich auf innerhalb der SoA **standardisierte Schnittstellen und Schnittstellenformate**. Um administriert und integriert zu werden sind SoA-Services automatisch **entdeckbar und gegenüber der Administrationsstelle dynamisch beschrieben** über Standardfunktionen. Services sind **abgeschlossene Softwareartefakte**, deren Beschaffenheit sich zur Laufzeit nicht ändert, um die Systemausführung verlässlich zu machen. Andere Ansätze, wie die Zustands- und Datenlosigkeit, sind für die Automatisierung nicht verwendbar, da sie die Erstellung kritischer deterministischer Systeme erschweren. In SoA wird anlassbezogen kommuniziert, üblicherweise auf Server- oder Cloudplattformen. Sie sind bei ERP und

MES etabliert und damit Zielstrukturen auf externen Plattformen. Die **Verknüpfung von lokalen Systemen** aus der Automatisierungstechnik **mit externen Plattformen ist eine notwendige Basis für Integrationskonzepte** von globalen Systemen über Ansätze wie Abbildung 3.1 oder des RAMI4.0. Weil die Umsetzung komplexer Vorgänge bei Produktionssystemen in der externen Plattform, insbesondere Steuerungsvorgänge, durch den Konflikt zwischen Kommunikationsweg und Zeitanforderung sowie die Determinismusanforderungen bei der Softwareausführung erschwert wird, ist die **Integration von externen Systemen** oder Plattformen und der Automatisierungstechnik vorrangig für eher abstrakte, **anlassbezogene, nicht kontinuierliche und unkritische Kommunikationsvorgänge** bedeutend. Eine **ereignisorientierte Kommunikation** in Protokollen aus dem Bereich (beispielsweise HTTP/REST oder ein SoA-Protokoll) **muss hierfür unterstützt werden**.

Die Modularisierung und Auslieferung von Software mittels Containersystemen bietet aufgreifbare Vorteile für die Automatisierung, da sie eine Anwendung kapseln und ihre technischen Anforderungen abstrahieren kann, sie eine einfache Anwendungsadministration erlauben und dabei im Vergleich mit virtuellen Maschinen sehr geringe zusätzliche Anforderungen oder Leistungseinbußen auf ihren ausführenden Systemen erzeugen. In Containern eingebettete Anwendungen sind ausreichend leistungsfähig für die Automatisierung und weisen ein stabiles, vorhersehbares Zeitverhalten auf (Sollfrank et al. 2021). **Automatisierungsfähige Software lässt sich in Containern einsetzen** (Sollfrank et al. 2021; Gundall et al. 2021; Gundall et al. 2020b). Sie sind zudem kompatibel mit weiteren Standardtechnologien, die für die Automatisierung interessant sind wie Ethernet TSN oder PTP (Sollfrank et al. 2021). Um Containerarchitekturen implementieren zu können, sind **technische Plattformen mit Betriebssystemen und containerfähigem Kernel sowie ein änderbares Dateisystem notwendig**. Um containerfähige Softwareteile zu bauen, ist eine **Dateibasierung** bei einzubettenden Anwendungen **nötig**.

Um CPS als verteilte und rekonfigurierbare Systeme in die Automatisierung zu tragen, wird eine neue Entwicklungs- und Engineeringbegleitung benötigt (Weyns et al. 2021). Aus der Softwareentwicklung ist das DevOps-Konzept adaptierbar, da es Rekonfiguration berücksichtigt, Softwareerstellung und -betrieb verbindet und die geordnete Rückführung von Laufzeiterfahrung in die Softwareentwicklung einführt (Stepien et al. 2022; Koren et al. 2023). DevOps-Ansätze **benötigen vereinte Plattformen und Funktionen für**

CPS-Erstellung und -Betreuung (Ebert & Hochstein 2023). DevOps setzt einen **komponentenbasierten Softwareaufbau** voraus, da es dauernden Austausch von Softwareteilen fordert (CI) sowie verteilungsfähige Technologien wie Container, da geänderte oder ersetzte Software sofort ausgeliefert werden soll (CD) (Mowad et al. 2022). Um Rekonfiguration mit physischen Systemen zu nutzen, müssen Validierung oder Simulation möglich sein (Koren et al. 2023). Lösungen aus dem Forschungsbereich des **digitalen Zwillings** und dem SoS-Engineering sind dafür nutzbar. Sie müssen ausreichend Informationen über alle Disziplinen am betroffenen CPS liefern **und bidirektional arbeiten** (Jones et al. 2020; Ejersbo et al. 2023; Hoffmann et al. 2021). SoS-Engineering erfordert MDE-Funktionen, um verwendet werden zu können (Albers & Lohmeyer 2012).

Framework	Modellumfänge	Automatisierungsfähigkeit	Interoperabilität	Adaptierbarkeit
IEC 61499	Modelle für Programme, beteiligte Ressourcen, Funktionsblöcke in Programmen	bedingt; Ereignisorientierung, graphische Programmerstellung, Daten- und Zustände verborgen	muss implementiert werden in Funktionsblöcken, direkt beteiligte Systeme müssen IEC 61499 unterstützen	über Deployment neuer Programm- und Verteilungsmodelle Rekonfiguration, Flexibilität muss implementiert werden in Programmen oder Funktionsblöcken
FDI	deployment- und informationsbezogene Modelle	keine Maßnahmen, da kein Logikbezug; muss durch Softwarepakete und System gewährleistet werden	Datenrepräsentation und -anbindung über einen OPC-UA-Server, Softwarepakete können verschiedene Inhalte haben	umfangreiche Funktionen für Softwareverteilung und Aufbau von Systemen für Rekonfigurierbarkeit, für Flexibilität Rückfall auf die nicht durch FDI logisch bestimmten Softwarepakete
VWS	Beschreibungsmodelle für Systemstrukturen und Funktionen	keine Maßnahmen, da kein Logikbezug; muss durch Softwarepakete und System gewährleistet werden	bei aktiver VWS mit kommunikationsfähigen Teilmodellen, keine gemeinsame Verteilung verschiedenartiger Systeme	über Austausch/Ergänzung von Teilmodellen oder Teilmodellparametern Rekonfigurierbarkeit bei aktiven VWS, Flexibilität muss in Teilmodellen implementiert werden
DOME	Modelle für Programme, beteiligte Ressourcen, Programmteile, Daten	bei entsprechender Implementierung der Kommunikation und Prozessobjekte, textuelle Programmierung	nur über kommunikationsfähige Objekte, direkt beteiligte Systeme müssen DOME unterstützen	über Austausch/Ergänzung von Teilmodellen Rekonfigurierbarkeit, Flexibilität muss implementiert werden in eigenen Programmen, Programmobjekten oder Parametern
AUTOSAR	Modelle für Komponenten, Struktur, Kommunikation	bei CP über bekannte Softwarekomponenten möglich, bei AP einzeln zu bewerten aufgrund änderbarer Struktur, schwache Transparenz	nur über eigene Softwarekomponentenimplementierung, direkt beteiligte Systeme müssen AUTOSAR besitzen und in Programm eingebettet sein	über Softwarekomponenten zu implementieren, Rekonfiguration bei AP durch Komponentendeployment

Tabelle 3.1: Elemente und abstrakte Eigenschaften der technischen Frameworks

Framework	Beschreibung	wesentliche Ansätze	Wirkungszeit und -ort
IEC 61499	modellbasiertes System zur grafischen Erstellung von Programmen für verteilte Steuerungssysteme; adaptiert Bausteinsprachen aus der IEC 61131; ein erstelltes Steuerungsprogramm wird definiert auf angegebene Systeme aufgebrochen und verteilt	Hardware- bzw. Plattformunabhängigkeit, Ereignisorientierung, Modellbasierung, verteilte Systeme	bei Maschinen- oder Anlagenbau durch Integrator bzw. Steuerungsprogrammierer
FDI	Normenbasiertes Integrationssystem zum Aufbau von digitalen Produktionssystemen aus Clients- und Servern; Grundstrukturen für Softwareverteilung, Grundsoftware auf Komponenten, Kommunikation	Paketdeployment, teilweise Modellbasierung (Profile)	Grundprofile bei Komponentenentwicklung, Gesamtsystem bei Maschinen- oder Anlagenbau durch Integrator bzw. Steuerungsprogrammierer
VWS	Softwarestruktur zur standardisierten Verwaltung eines Betriebsmittels und zugehöriger Informationen; kann passiv oder aktiv sein; mit Kommunikationsfähigkeit und integrierter Softwaremodularisierung	Kaskadierung, Aggregation, Anpassung über Teilmodelle im Gesamtsystem (Softwaremodularisierung), Lebenszyklusbegleitung	Standard-VWS bei Komponentenentwicklung; Gesamtsystem-VWS durch Integrator bei Maschinen- oder Anlagenbau
AUTOSAR	System zur modellbasierten Firmwareerstellung für Automotive-Steuergeräte und -Steuergerätenetz; zwei Grundversionen (Classic und Adaptive) mit verschiedenen Funktionsumfängen; Adaptive mit komplexem Kernel und Laufzeitrekonfiguration	Modellbasierung, Determinismus, Hardware- bzw. Plattformunabhängigkeit	hauptsächlich bei Geräteentwicklung durch Softwareentwickler, teilweise zur Laufzeit durch Systemhersteller (Adaptive)
DOME	Objekt- und modellorientierte Entwicklungs- und Ausführungsumgebung für verteilte Steuerungssysteme; Beschreibung eines gemeinsamen Steuerungsmodells über Objekte und Verteilung auf Laufzeitumgebungen auf eingebundenen Systemen	Objektorientierung, Modellbasierung, verteilte Systeme, Determinismus	bei Maschinen- oder Anlagenbau durch Integrator bzw. Steuerungsprogrammierer

Tabelle 3.2: Beschreibungen und Ansätze der technischen Frameworks

Framework	Normierung	Technologieunterstützung
IEC 61499	ja; DIN EN 61499, ISO IEC 61499	Systeme mit fähiger Ausführungsplattform (abstraktionsfähig für IEC-61499-Programme); Feldkomponenten, Maschinen- und Anlagensteuerungen; Implementierungen in Java, C++, teilweise Objektorientierung, keine Vorgabe von Kommunikationsprotokollen
FDI	ja; IEC 62543, IEC 62769	Feldkomponenten, Maschinen- und Anlagensteuerungen; Standard-Server-/PC-Umgebungen für begleitende und Verwaltungsfunktionen, OPC UA für Informationsmodelle und Kommunikation sowie Serviceschnittstellen
VWS	ja, frühes Stadium; DIN SPEC 91345	Automatisierungsgeräte allgemein, Schwerpunkt Feldkomponenten, intelligente Feldgeräte; Repräsentation von Maschinen- und Automatisierungskomponenten; OPC UA zur Informationswiedergabe, Implementierungen in C#, Java und weitere, insgesamt abhängig von Ausprägung (aktiv, passiv, passiv mit API), Objektorientierung, große Komplexität
AUTOSAR	ja; eigene Automotive-Normungen	Automotive-Steuergeräte; in Adaptionen für Feldkomponenten und Maschinen- und Anlagensteuerungen, Auslegung auf Kleinsysteme, ohne Betriebssysteme, wenige Standardtechnologien, (Embedded-)C++, Objektorientierung, sehr große Komplexität
DOME	nein	Feldkomponenten, Maschinen- und Anlagensteuerungen; C++, Objektorientierung, eigene Implementierung

Tabelle 3.3: Normierungen und zugrundeliegenden Technologien der technischen Frameworks

4 Das Integrations- und Steuerungsframework *ISCI*: Konzept

Technologische Lösungen für Automatisierungsprobleme in der Produktion sind heute gegeben, wodurch die Automatisierung von physischen Produktionsvorgängen an sich sowie die Vernetzung von Automatisierungssystemen keine Probleme mehr darstellen. Problematisch hingegen sind die heute notwendigen Umsetzungsaufwände, statische Beschaffenheiten, die Bindung an spezifische Technologien mit großen Problemen bei Verfügbarkeiten und Interoperabilität, die Aufwände bei Adaption bereits automatisierter Systeme an neue Produkte und der Umgang mit neuen Integrations- und Kommunikationsanforderungen.

Automatisierungslösungen sind stark anwendungsspezifische Lösungen, die nur beschränkt wiederverwendet werden können, wodurch sich bei neuen Automatisierungsaufgaben auch bei ähnlicher Aufgabenstellung Arbeiten oft wiederholen. Die spezifischen Technologien der Automatisierung und ihre Bindung an spezielle Programmiersprachen und Entwicklungsumgebungen schränken dabei mögliche Nutzung, implementierbare Funktionen, Wiederverwendung und Integration mit anderen Systemen weiter ein. Programmiersprachen und Entwicklungsumgebungen der Automatisierung erfordern spezifische Kenntnisse und sind verglichen mit Entwicklungssystemen der Consumer- oder Internetwelt aufwändig in der Anwendung. Wenn mehrere programmierbare Systeme zur Laufzeit gemeinsam einen Prozess automatisieren sollen, vervielfacht sich der Implementierungsaufwand mit den derzeit verfügbaren Systemen und Frameworks, obwohl die eigentlichen Prozessabläufe oft einfach sind. Für die ganzheitliche Implementierung einer verteilten Automatisierungslösung existiert insgesamt kein etablierter Ansatz. Dieser Problematik begegnet die Arbeit, indem ein nutzungsnahes Rahmenwerk definiert wird, das technische und logische Elemente und eine vereinigende Struktur für die zukünftige Umsetzung von Automatisierungssystemen bereitstellt und dabei die Paradigmen cyberphysischer, verteilter und standardisierter Systeme einbezieht. Dieses Framework wird nach seinem

Anwendungsbereich benannt: ***I**ntegrations- und **S**teuerungsframework für **C**yberphysische **I**ndustrieautomatisierung*, kurz *ISCI*. Das ISCI wird ausgelegt auf die Nutzung von verbreiteten Mittel- und Hochsprachen auf Standardsystemen, eine transparente und modellbasierte Modularisierung sowie größtmögliche Transparenz. Es soll als Grundlagensystem die technische Basis, Grundfunktionen und den funktionalen Rahmen für Automatisierungssysteme bereitstellen.

4.1 Anforderungen

Es sind abstrakte Anforderungen abzudecken, die aus den spezifischen Anforderungen verschiedener Gebiete folgen, die vom Framework berührt werden. Zu diesen Gebieten zählen CPS, kritische physische Systeme und Industrieautomatisierung, Softwareentwicklung, -modellierung und -implementierung.

- **Kooperation und Vernetzung:** Die Kerneigenschaften cyberphysischer Systeme sind Kooperation und Vernetzung. CPS-Teilsysteme müssen die Steuerung des gemeinsamen physischen Prozesses über ständige Vernetzung und kooperationsbasierte Logik implementieren.

- **Physis- und Zustandsnähe, Determinismus:** Automatisierungssysteme steuern physische Systeme und müssen diese jederzeit beherrschen. Der Zustand der gesteuerten Physis muss als zentrales Eingangsbild einfließen, um das physische System deterministisch als CPS zu steuern.

- **Generalisierung und beliebige Integration:** Technische Lösungen müssen wiederverwendbar und frei integrierbar sein, um Aufwände bei der Erstellung oder Änderung klein zu halten. Aufgaben für Softwareframeworks sind nicht im Detail vorhersehbar, was die Erstellung von Standardlösungen erschwert. Generalisierungen müssen ebenso wie Möglichkeiten für spezifische Lösungen unterstützt werden.

- **Technologieunabhängigkeit:** Die Funktionalitäten und die Weiterentwicklung der Automatisierungstechnik sind durch die Bindung an spezifische Hard- und Software beschränkt. Diese Bindungen schränken die Gestaltung von verteilten und besonders

netzwerk- und kooperationsorientierten Systemen ein und erschweren die Generalisierung von Automatisierungsanwendungen. Das Framework muss die Abstraktion von spezifischen Technologien unterstützen.

- **Vereinfachung und Transparenz:** Entwicklung und Betrieb eines technischen Systems werden von Menschen durchgeführt und das entstehende System vom Menschen betrieben. Transparenz von Teil- und Gesamtsystemen hinsichtlich Eigenschaften (Aufbau, Logik, Parameter, ...) sind für Entwicklung und Betrieb von deterministischen Systemen zentral. Trotz beliebiger und wachsender Systemumfängen muss eine beherrschbare Komplexität gewährleistet sein.

- **Implementierbarkeit**: Als Rahmenwerk ist die Arbeit auf eine Umsetzung in einem technischen System angewiesen, um entwickelt und validiert zu werden. Folglich sind Implementierbarkeit und Umsetzungsnähe zu beachten. Diese Eigenschaften sind auch für das in Abbildung 1.3 gezeigte Vorgehen nötig.

4.2 Struktur

4.2.1 Lösungsansätze

4.2.1.1 Dynamischer, modellbasierter Aufbau verteilter Systeme

Um die Änderungskosten in der Automatisierungstechnik bei Produktionsänderungen zu minimieren, muss ihre Struktur selbstorganisierend und änderbar werden, ohne die für die Automatisierung notwendige Anforderungen bei Determinismus und Konsistenz zu vernachlässigen. Da Automatisierungssysteme im Frameworkverständnis verteilte Systeme sind, nimmt die zu bewältigende Komplexität zu. Im Framework wird dieser Komplexität mit mehreren Elementen begegnet: Mehrstufige Modellbasierung der Automatisierungsanwendungen und ihrer Variablen- und Informationsstrukturen, selbstorganisierende beschreibungsbasierte Verbindungen, statische Kernmodelle für Automatisierungssysteme und dynamische, automatische Erweiterung durch Softwarekomponenten.

Die Automatisierungsanwendungen definieren die Teilnehmer des verteilten Systems und die Grundstrukturen für Daten, Funktionen und Ereignisse, die im gebauten verteilten Automatisierungssystem zu jedem Zeitpunkt vorhanden sind. Hierdurch ist ein fester logischer Grundaufbau im verteilten System gewährleistet. Die Softwareteile des verteilten Systems können diese Grundstrukturen zur Betriebszeit dynamisch erweitern. Die Beschaffenheiten der Teilnehmer des verteilten Systems werden in regelmäßigen, kurzen Abständen auf einer zentralen, allen bekannten Plattform in einem frameworkeigenen Standardformat beschrieben. Diese Beschreibung enthält Verbindungsinformationen, die von den anderen Teilnehmern des verteilten Systems für die Verbindungserstellung genutzt werden.

Feld- und feldnahe Automatisierungskomponenten sind, bedingt durch besondere Hardwareanforderungen, oftmals integrierte Systeme, bei denen Hardwareplattform, Logik und Parametrisierung (beispielsweise in Form eines Programms und einer Netzwerkkonfiguration) und Schnittstellen zur Außenwelt untrennbar aneinandergebunden sind. Sie können im Framework durch vordefinierte Module und die automatische Aggregation ihrer Modellteile zur Betriebszeit und damit dynamisch integriert werden. Alle Modelle im Framework folgen Grundstrukturen und besitzen eindeutige Identifikationen. Ihr Aufbau orientiert sich an Implementierungsklassen ohne auf eine bestimmte Sprache ausgelegt zu sein. Ihr Austausch erfolgt lesbar serialisiert. Jedes gegebene Modellformat des Frameworks ist ein Grundformat und kann spezifisch erweitert werden. Ein Grundverständnis ist für die beteiligten Einzelsysteme somit immer gegeben, während spezifische Erweiterungen berücksichtigt und möglich sind.

4.2.1.2 Gemeinsame Datenmodelle und datenbasierte Integration

Für rekonfigurierbare Automatisierungslösungen mit CPS muss die in Ebenen unterteilte, hierarchische Organisation aufgelöst werden. Statt hierarchisch angeordneter Ketten von Einzelsystemen werden dazu verteilte Systeme gebildet. Für die Bildung dieser verteilten Systeme steht die Gesamtmenge aller kommunikationsfähigen technischen Systeme in der Produktion zur Verfügung. Zur Umsetzung einer Automatisierungsanwendung wird eine Untermenge dieser Gesamtmenge als ein Verbund um eine gemeinsame Datenlogik

angeordnet (siehe Abbildung 4.3). Diese Datenlogik folgt einem allen technischen Systemen bekannten Datenmodell und einer jeweils lokalen Instanziierung des Datenmodells. Dadurch entsteht eine konsistente, zustandsorientierte Informationsgrundlage innerhalb des verteilten Systems für eine Automatisierungsaufgabe. Gebunden an eine Automatisierungsanwendung wird so eine abgeschlossene Logikstruktur geschaffen. Für den kontrollierten Zugang zur Automatisierungslösung wird eine Schnittstellenstruktur aufgebaut, die aus Modellen und technischen Zugriffen auf die Datenlogik besteht. Die Schnittstellenstruktur realisiert einen gezielt aufgebauten ereignis- und funktionsorientierten Zugriff der Außenwelt auf das verteilte System. Über die Schnittstellenstruktur werden externe technische Systeme (Dienste, Anwendungen, Webseiten, Mensch-Maschine-Schnittstellen, ...) angehängt, die kein unmittelbarer Bestandteil des Automatisierungssystems sind.

Die Arbeit auf der lokalen Datenstruktur kann über gegenseitigen Ausschluss konsistent gehalten werden. Die Aktualisierung der lokalen Datenstruktur anhand eingegangener Änderungen erfolgt kontinuierlich oder gebunden an eine lokale Abfolge. Datenfelder in der Struktur besitzen einen schreibenden und mehrere lesende Teilnehmer im verteilten System, sind also nicht fest verwoben. Dadurch ist Eindeutigkeit der Daten gewährleistet. Durch die Datenorientierung als Systemmittelpunkt ist die Freiheit in der technischen Ausführung der angebundenen Anwendungen größer, verglichen mit einer direkten Kommunikation zwischen den Anwendungen oder bei der Bildung einer geschlossenen einzelnen Anwendung für ein Automatisierungsproblem. In der Bereitstellung einer unabhängigen, einfach lokal instanziierten Datenstruktur und der Bindung an statische Datenmodelle für den Determinismus unterscheidet sich der Ansatz von anderen datennahen Integrationen wie DDS (siehe de Jong 1999) oder dem Robot Operating System. Module als Softwareteile im System erweitern das anwendungsspezifische Datenmodell, sodass der Zustand jedes Systems vollständig erkennbar ist. Die starke hierarchische Abfolge, die durch direkte Integration bei Agentensystem üblicherweise entsteht, wird dadurch vermieden und Parallelbetrieb sowie Kooperation ermöglicht. Abbildungen der Datenstruktur zu bestimmten Zeitpunkten können als Auswertungsbasis herangezogen werden.

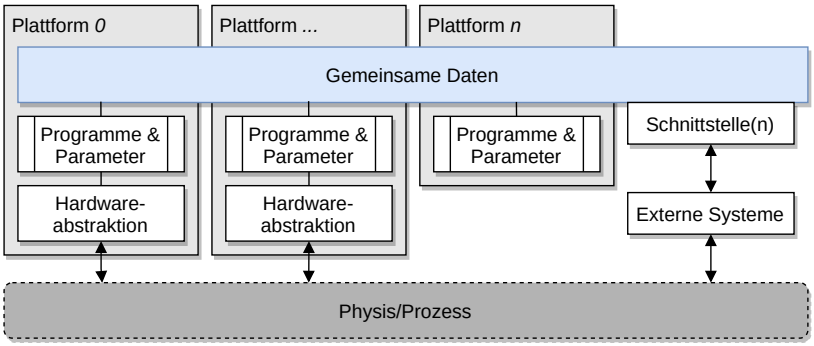


Abbildung 4.1: Verknüpfung auf der Datenebene als Integrationsbasis

4.2.1.3 Determinismus über Zustandsorientierung

Mit der zentralen Datenstruktur im verteilten System kann ein deterministischer endlicher Automat (siehe Abschnitt 2.3.1 sowie Abbildung 2.4) über mehrere technische Systeme aufgespannt werden, skizziert in Abbildung 4.2. Die technischen Systeme stellen Eingangs- und Ausgangsnetzwerke. In der gemeinsamen Datenhaltung des verteilten Systems liegt der Zustandsspeicher des Automaten. Durch diesen Ansatz kann ein deterministischer Automat, der in Steuerungsebene in einem einzelnen Steuerungsprogramm gehalten wird, auf ein verteiltes System ausgebreitet werden. Die Aktualisierung des Zustandsspeicher kann an einen Takt gebunden werden. Die Ausführung von Logikteilen kann über Einträge in der gemeinsamen Datenhaltung in einem Ausführungsmodell angeordnet werden.

Das Datenmodell ist objektorientiert und erlaubt verschachtelte, hierarchische Strukturen, wodurch die Abbildung von Objekten möglich ist. Unterschieden werden die Instanziierung und die logische Struktur in die Datenhaltung und die Datenmodellstrukturen. Die Datenbasierung eines Automatisierungssystems ermöglicht eine zustandsorientierte Implementierung von Automatisierungslösungen und eine laufende Historiebildung. Über die erstellten Datenabbilder können in Verknüpfung mit Erstellungszeiten Zeit- und Logikverhalten des Automatisierungssystems validiert werden.

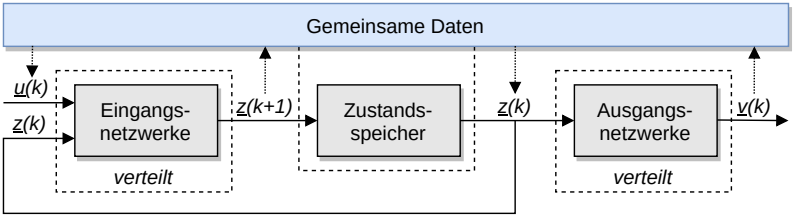


Abbildung 4.2: Ansatz für verteilten endlichen Automat

4.2.1.4 Funktions- und Ereignisschnittstellen

Ein Automatisierungssystem, das einen oder mehrere physische Vorgänge automatisiert steuert, muss mit seiner Umwelt kommunizieren, um in andere Automatisierungs- oder Verwaltungsstrukturen eingebunden werden zu können. Umgebungssysteme, besonders Verwaltungssysteme, sind als nicht programmier- oder erweiterbar anzunehmen, sie sind also an einen Funktionsumfang gebunden. Um eine ausreichende Flexibilität im Gesamtsystem bei der Kommunikation und der darüber gehenden Integration zu gewährleisten, implementiert das Framework Grundlagen für definier- und erweiterbare Kommunikation des Automatisierungssystems nach außen. Diese Kommunikation ist anlass- oder ereignis- und nicht zustandsbezogen, da sie nicht für die Prozesssteuerung vorgesehen ist, und teilt sich in Funktions- und Ereignisschnittstellen. Beide wirken auf eine Datenstruktur des Automatisierungssystems. Funktionsschnittstellen erlauben den kontrollierten, methodenähnlichen Zugriff auf die Datenstruktur von außen. Änderungen über eine Funktionsschnittstelle werden über die Datenstruktur im Automatisierungssystem verteilt. Ereignisschnittstellen kommunizieren nach Vorgabe Informationen aus der Datenstruktur nach außen. Die Schnittstellen werden modellbasiert von Automatisierungsressourcen anhand einer festen Ausprägung oder ihrer Konfiguration aufgebaut und können beliebige Protokolle unterstützen. Die Funktions- und Ereignisschnittstellen arbeiten mit protokoll- und implementierungseigenen Vorverarbeitungen gegen die Dateneinträge einer Datenstruktur.

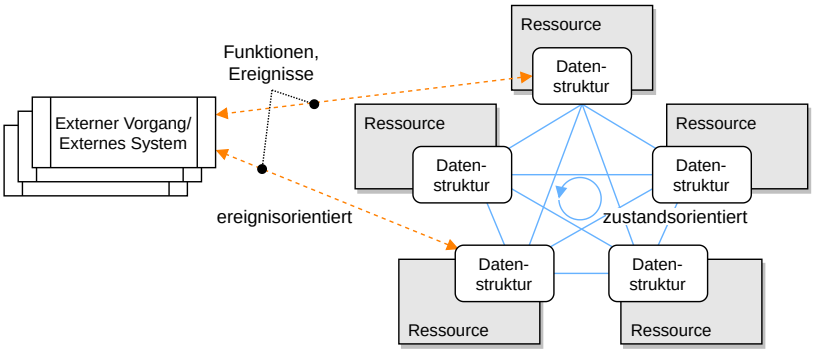


Abbildung 4.3: Ressourcen eines Automatisierungssystems, integriert über gleich aufgebaute, zustandsorientierte Datenstrukturen mit ereignisorientierten Schnittstellen nach außen über ausgewählte Datenstrukturen

4.2.1.5 Aufbau auf Standardtechnologien

Es werden Standardtechnologien in den verschiedenen technischen Bereichen als Basis aufgegriffen, um die technologischen Möglichkeiten der Automatisierung zu erweitern und die bisherige Spezifität der Automatisierung aufzulösen. Durch stärkere Einbringung von Standardtechnologien kann das Framework und die mit ihm umgesetzten Automatisierungssysteme von Entwicklungen außerhalb des Automatisierungsbereichs profitieren. Zudem erleichtert es die gleichberechtigte Integration aller vernetzten Systeme in der Produktion und die Modularisierung und Virtualisierung von Automatisierungssoftware.

Das Framework berücksichtigt Standardtechnologien bei seinen ausführenden technischen Plattformen, bei den Kommunikations- und Integrationstechnologien, den zu erstellenden Softwareteilen und bei den einzubringenden Entwicklungsmöglichkeiten und -werkzeugen. Es wird auf die spätere Verbindung mit Virtualisierungs- und Containermodularisierungstechnologien ausgerichtet. Die Unix-Grundsätze für Softwaresysteme nach Raymond 2003 werden als Grundparadigma herangezogen.

4.2.1.6 Kartierbarkeit auf die Verwaltungsschale

Das für technische Systeme bedeutende Hauptelement im Kontext des RAMI4.0 ist die Verwaltungsschale (VWS). Die VWS ist durch seine allgemeine Auslegung ein umfassendes Sys-

tem und kann in zwei Ansätzen bei technischen Systemen verwertet werden: Erstens kann das Beschreibungsformat und -vorgehen der VWS für die Beschreibung des technischen Systems verwendet werden. Zweitens kann die Struktur der VWS auf komponentenbasierte Softwarestrukturen übertragen werden (Breunig et al. 2020).

Das Framework ist eine praxisorientierte Struktur, die als einfachstmögliches System spezifische Implementierungen bei Unikaten erleichtern soll und gibt dazu eigene technische und logische Strukturen vor. Es entsteht keine Implementierung der VWS. Die VWS ist eine netzwerkorientierte Struktur, die ein technisches System vertritt. Durch die VWS als überall vertretene Grundstruktur sind VWS-Systeme logisch und formal integrierbar. Die Kartierung einer mit dem Framework gebauten Architektur oder Teilen davon auf eine oder mehrere VWS ist eine langfristige Zieleigenschaft, die konzeptuell berücksichtigt wird. Das bedeutet, dass aus den Beschreibungs- und Softwarestrukturen des Frameworks VWS generiert werden können und umgekehrt. Durch die Kartierung der entstehenden Daten- und Logikelemente auf eine VWS-Struktur ist mit einer *I4.0-konformen* Schnittstelle eine Anbindung an bestehende VWS oder die Einbettung in bestehende oder entstehende VWS möglich. Wie in Abbildung 4.4 gezeigt, sind zwei entstehende Artefakte für die Übertragung in VWS vorgesehen: die gebauten einzelnen Automatisierungsressourcen (siehe Abschnitt 4.2.2.2) und die aus Automatisierungsressourcen zusammengesetzten Automatisierungssysteme (Abschnitt 4.2.2.1). Im Falle der Automatisierungsressource ist das verwaltete Asset der VWS die Basissoftwarestruktur der Automatisierungsressource sowie ihre eigene Hardware, beispielsweise ein Aktor oder eine Steuerungsplattform. Im Falle des Automatisierungssystems entspricht das VWS-Asset der gesamten Hard- und Software der Automatisierungsebene sowie der physischen Einrichtung, die gesteuert wird, also die Maschine oder Anlage.

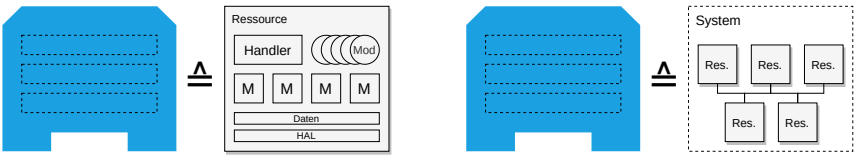


Abbildung 4.4: Mögliche Entsprechungen der Verwaltungsschale: Entsprechung als Automatisierungsressource links, Entsprechung als Automatisierungssystem rechts. Zu Abkürzungen und Architekturen von Automatisierungsressource und Automatisierungssystem siehe Abbildung 4.5

4.2.2 Technische Artefakte

4.2.2.1 Automatisierungssystem

Ein mit dem Framework gebautes Automatisierungssystem ist eine Menge von Automatisierungsressourcen, die durch Handler Anwendungsmodellen folgend mit Software versehen, konfiguriert und verbunden werden. Sie steuern als System gemeinsam einen oder mehrere physische Prozesse mit ihren Schnittstellen zur physischen Welt. In Verbindung mit den gesteuerten physischen Einrichtungen bilden Automatisierungssysteme die CPS, die mit Instanzen des Frameworks gebaut und gesteuert werden.

Die Menge der konfigurierten Automatisierungsressourcen im Automatisierungssystem ist zur Laufzeit um ein gemeinsames, aggregiertes Datenmodell angeordnet, das alle Daten und Zustände der virtuellen und physischen Welten sammelt.

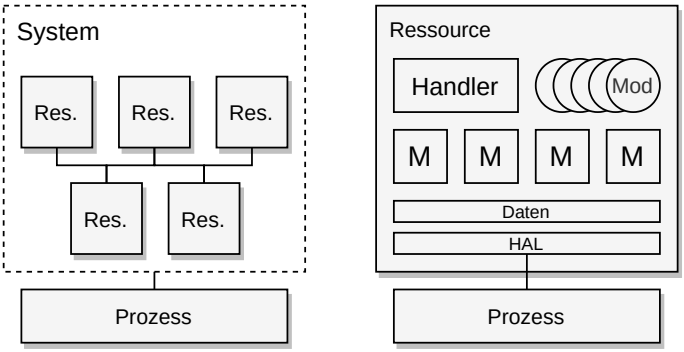


Abbildung 4.5: Das Automatisierungssystem (links) ist eine Integration mehrerer Automatisierungsressourcen (*Res.* in der Darstellung) zur Automatisierung eines physischen Prozesses. Eine Automatisierungsressource (rechts) ist eine technische Plattform bestückt mit einem oder mehreren Softwaremodulen (*M*), einem Handler zur Verwaltung, Modellen (*Mod*) zur Konfiguration, persistenten und Laufzeitdaten sowie einer Hardwareabstraktionsschicht (*HAL*) und Schnittstellen zum physischen Prozess

4.2.2.2 Automatisierungsressource

Automatisierungsressourcen sind die technischen Plattformen, die zur Laufzeit die Software eines Automatisierungssystems ausführen. In Text und Abbildungen der Arbeit werden

Automatisierungsressourcen wo nötig als Ressourcen abgekürzt. Eine Automatisierungsressource ist somit eine Maschine, auf welcher prozessrelevante Software platziert werden kann. Eine virtuelle Maschine oder eine frameworkfähige SPS können Automatisierungsressourcen sein. Eine Automatisierungsressource kann über EA-Schnittstellen zur physischen Welt oder besondere Kommunikationsschnittstellen verfügen. Eine feste Eingrenzung der Systeme, die Automatisierungsressource sein können, erfolgt nicht. Die Befähigung zur Automatisierungsressource im Sinne des Ansatzes ist gegeben, wenn sie von einem Handler des Frameworks verwaltet werden und Software entgegennehmen kann und sie zur Laufzeit am Datenaustausch innerhalb eines Automatisierungssystems beteiligt ist.

Auf der Automatisierungsressource können mehrere Zielebenen vorhanden sein. Gemeint sind virtuelle Teilsysteme (zum Beispiel Dateisysteme oder Programmspeicher) einer Automatisierungsressource, die mit einem Softwareteil versehen werden können. Die Verwaltung dieser Zielebenen und die Verteilung der zugewiesenen Konfigurationsteile auf diese Zielebenen wird vom Handler übernommen. Jede Automatisierungsressource wird von einem Handler integriert, wobei die Handlerinstanz Teil der Automatisierungsressource (eingebettet) oder übergeordnet (gekoppelt) sein kann (siehe auch Abbildungen 4.7, 5.5 sowie Abschnitt A.1.5). Wenn eine Automatisierungsressource nicht nativ am Datenaustausch innerhalb eines Automatisierungssystems teilnehmen kann, kann sie über eine zusätzliche Softwarebrücke verbunden werden. Die entsprechende Instanziierung und Parametrierung wird vom zuständigen Handler verborgen vorgenommen.

4.2.2.3 Datenstruktur

Eine Datenstruktur ist eine technische Struktur auf einer Automatisierungsressource, in der zur Betriebszeit eine Menge eindeutig identifizierbarer Datenfelder als Einträge abgelegt sind. Diese Dateneinträge sind den Modulen auf der Automatisierungsressourcen zugänglich. Jede Automatisierungsressource verfügt über eine oder mehrere Datenstrukturen, über die lokal die Daten der zugewiesenen Automatisierungsanwendungen abgebildet werden. Eine Automatisierungsanwendung ist auf eine Datenstruktur beschränkt. Im Betrieb werden Änderungen in den Datenstrukturen der Automatisierungsressourcen abgeglichen.

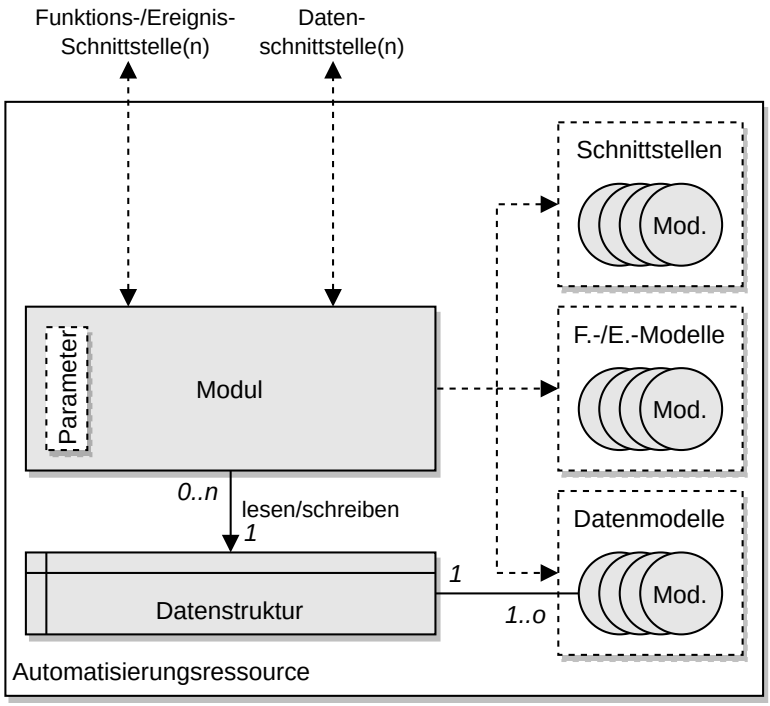


Abbildung 4.6: Module auf einer Automatisierungsressource greifen auf die Datenstruktur zu und betreiben Funktions- (*F.-Modelle*), Ereignis- (*E.-Modelle*) und Datenschnittstellen entsprechend der Modelle (*Mod.*)

Da die Datenstruktur den prozessrelevanten Austausch realisiert, muss sie hinreichend schnell und zuverlässig sein. Die Dateneinträge in den Datenstrukturen werden anhand von Datenmodellen angelegt (Abbildung 4.6). Die Datenmodelle werden durch Anwendungsmodelle und durch Module geliefert. Die Datenmodellvorgaben der Anwendungsmodelle stellen statische Definitionen, die als strukturdeterministischer Kern bei der Automatisierungssystemmodellierung festgelegt werden und zur Betriebszeit unverändert bleiben. Ihre Definitionen in der Datenstruktur sind die Datengrundlage für die Prozesslogik und über alle Automatisierungsressourcen ohne zusätzlichen Informationsaustausch gleich.

4.2.2.4 Module

Module sind aktive Softwareteile als eigenständige Anwendungen oder als Kombination aus einer Logikstruktur und einer ausführenden Umgebung. Alle innerhalb eines Automatisierungssystems platzierten system- oder prozessrelevanten Funktionen sowie die Automatisierungslogik werden über Module realisiert. Zur Laufzeit greifen sie auf die lokale Datenstruktur zu und lesen und schreiben Dateneinträge. Alle Schnittstellen, virtuell und physisch, werden über Module aufgebaut und betrieben. Module auf einer Automatisierungsressource stammen aus einer Konfiguration oder sind fest auf einer Automatisierungsressource vorhanden. Module leisten Beschreibungsteile zu den verschiedenen Modellen (Datenmodell, Funktionsmodell, Ereignismodell, Systembeschreibung, Schnittstellen).

Module bilden ihre programmatischen Schnittstellen sowie Zustände und Vorgänge ihrer Logik in der Datenstruktur ab. Dazu erweitern sie die Datenstrukturen eines Automatisierungssystems mit eigenen Datenmodellen. Für die Implementierung der Automatisierungslogik sind die Module über die Datenstruktur integriert. Für nach außen verfügbare Logik stellen Module Ereignisse und Funktionen zur Verfügung, die sie über Ereignis- und Funktionsmodelle beschreiben und die über die Datenstruktur ausgegeben oder ausgelöst werden. Module sind zur Einrichtung auf einer Automatisierungsressource eindeutig auf Quellen identifizier- und abrufbar. Werden Automatisierungsressourcen als vorgefertigte Komponenten vorgesehen, sind Module als fester Bestandteil einer Automatisierungsressource integrierbar.

4.2.2.5 Handler

Der Handler erfüllt für *1..n* Automatisierungsressourcen drei zentrale Funktionen:

1. Integration in die Administrationsplattform
2. Softwarestrukturaufbau auf Zielebenen anhand der zugewiesenen Konfigurationen
3. allgemeine Soft- und Hardwareadministration

In der Abbildung 4.7 sind die Prinzipzusammenhänge zwischen Handlern und Automatisierungsressourcen dargestellt. Ein Handler kann fester Teil einer Automatisierungsressource

sein (eingebettete oder native Automatisierungsressource). Ist er Teil des Gerätes, beispielsweise bei einer Soft-SPS mit x64- oder x86-Betriebssystem und beschreibbarem Dateisystem, sind die Zielebenen direkt lokal. Die Zielebenen können, wie dargestellt, beispielsweise das Dateisystem, die Diensteverwaltung oder das Betriebssystem sein. Auf die Zielebenen werden durch den Handler in der Anwendungsverteilung Konfigurationen angewandt. Die notwendige Logik ist im Handler verborgen. Sie wird zyklisch ausgeführt (siehe Abbildung 4.8), um eine Integration zwischen Ressourcenfähigkeiten, Administration und Softwarequellen herzustellen.

Anders als bei vielen Agentensystemen besteht keine 1-1-Bindung zwischen einer Automatisierungsressource und Handler und es gibt keine Vorgabe wie der Zugriff auf die Automatisierungsressource zur Umsetzung von Konfigurationen beschaffen sein muss. Der Zugriff auf die Zielebenen kann auch über ein Netzwerkprotokoll erfolgen, wenn Handler und Zielebenen auf verschiedenen Plattformen implementiert sind. Dadurch sind Geräte, auf denen keine (komplexe) eigene Software ausgeführt werden kann oder deren Ausführungsfähigkeiten nicht ausreichend sind, als Automatisierungsressource in ein Automatisierungssystem des Frameworks integrierbar.

Ein Handler kann auch Automatisierungsressourcen verwalten, deren Zielebenen keine Standardtechnologien aufweisen oder deren Architekturen allgemein keinen Standard-IT-Architekturen entsprechen, beispielsweise SPS oder Mikrocontroller (gekoppelt). In diesem Fall übernimmt der Handler zusätzliche Funktionen analog zu einem Programmiergerät. Voraussetzung für eine sinnvolle Nutzung hier ist, dass eine Zielfirmware für ein solches Gerät in einer Konfiguration abgelegt ist. Eine Mischform, bei der ein Handler eine Sonderarchitektur wie eine SPS und eine eigene Standardarchitektur gleichzeitig verwaltet, ist möglich.

Beim Handler werden die notwendigen Informationen zur Ressourcenintegration abgelegt, wenn er kein Teil der Automatisierungsressource ist, um über einen Fernzugriff Konfigurationen umzusetzen. Die Informationen enthalten statische Angaben zur Verbindung und zur Repräsentation gegenüber der Administrationsplattform und werden mit von der Automatisierungsressource abgerufenen und erstellten Informationen erweitert. Im bevorzugten Fall ist der Handler nativer Teil eines Systems mit konfigurierbaren Ebenen (Dateisystem, Diensteverwaltung et cetera, links).

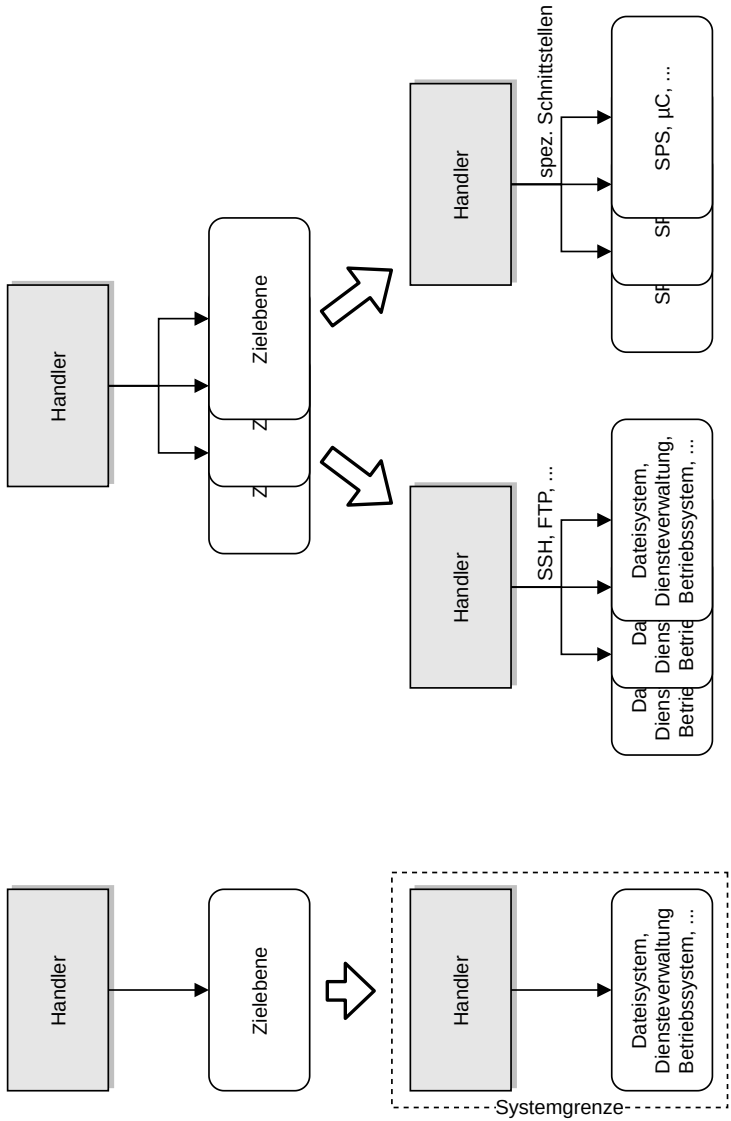


Abbildung 4.7: Anordnungsprinzipien von Handlern (eingebettete Automatisierungsressource links, gekoppelte Automatisierungsressourcen rechts) und Beispielen für ihre Zielebenen. Handler können Standardnetzwerkprotokolle wie SSH oder FTP verwenden, um ihre Zielebenen zu konfigurieren oder spezifische Schnittstellen, beispielsweise wenn Mikrocontroller (µC) oder SPS adressiert werden.

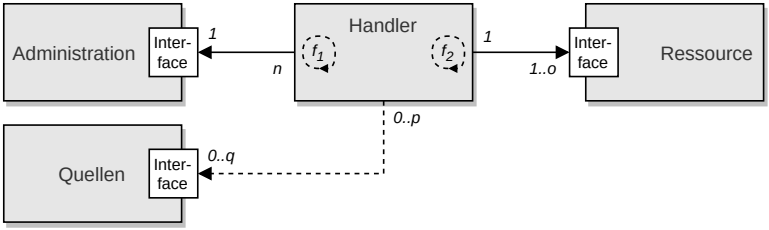


Abbildung 4.8: Integration der Automatisierungsressource gegen Administration und Quellen. Alle Seiten werden zyklisch auf Änderungen geprüft, um Konfigurationen und Beschreibungen vorzunehmen oder zu aktualisieren

4.2.2.6 Administrationsplattform

Die Administrationsplattform ist in Ebenen aufgebaut (siehe Abbildung 4.9). Ihre Basis ist eine Datenhaltung, in der die Instruktionen, Beschreibungen und Modelle aller über die Plattform organisierten Artefakte liegen (Anwendungsmodelle, Ressourcenbeschreibungen, Konfigurationspakete et cetera). Die Datenhaltung kann beispielsweise datei- oder datenbankbasiert sein. Die darüberliegende API ist die Schnittstelle zu den Daten und besitzt einen erweiterbaren Standardbefehlssatz. Die API nutzen beliebige Werkzeuge und die Handler zum Zugang zur Datenhaltung für Lese- und Schreibvorgänge. Sie kann mit einer Zugriffsteuerung erweitert werden, die den Zugriff auf alle oder Teile der Befehle einschränkt oder Authentifizierungen prüft. Die Administrationsplattform besitzt eine interne Verarbeitung, die Daten in festgelegte Formate bringt und aus Modellen Instruktionen für bestimmte Automatisierungsressourcen generiert. Die angebundenen Werkzeuge sind nicht festgelegt und ihre Instanzen nicht auf eine Administrationsplattform beschränkt. Zu ihnen zählen unter anderem Visualisierungs-, Entwicklungs- und Analysetools.

Die Administrationsplattform übernimmt keine Definitions- oder Entwicklungsaufgaben. Die Automatisierungsressourcen sind zur Ausführung ihrer Automatisierungsaufgaben nicht auf die Administrationsplattform angewiesen. Sie verfügt über keine für die Automatisierung laufzeitrelevanten Funktionen und wird unabhängig von Automatisierungsressourcen und Automatisierungssystemen betrieben. Über die Administrationsplattform werden alle mit dem Framework gebauten Systeme zur Laufzeit abgebildet. Ihr vorrangiger Zweck ist die Bereitstellung von Modellen und Beschreibungen für Vorgänge innerhalb einer Frameworkarchitektur, für eigene Werkzeuge und für externe Systeme (nicht abgebildet).

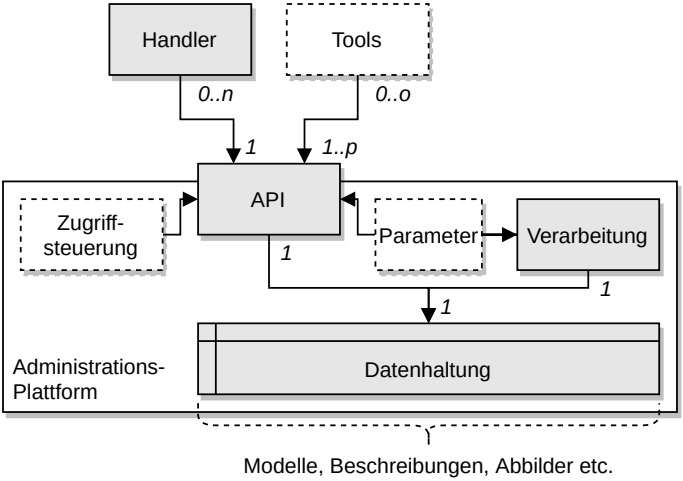


Abbildung 4.9: Kernbestandteile der Administrationsplattform

4.2.2.7 Quelle

Eine Quelle stellt zur Umsetzung von Konfigurationen Software bereit, die von Handlern während der Konfiguration von Automatisierungsressourcen abgerufen werden kann (siehe Abbildung 4.8). Die tatsächliche technologische Ausprägung ist frei. Ihre Beschreibung gibt die Zugriffstechnologie an, die vom Handler unterstützt werden muss. Handler erhalten zur Einrichtung eines Moduls eine Quellenangabe. Quellen können zugriffsbeschränkt sein und vom Handler eine Authentifizierung fordern. Bindungen von Quellen an Konfigurationen oder Automatisierungsressourcen sind nicht vorgesehen. Quellen sind global zugänglich und werden in einem Register in der Administrationsplattform erfasst, das von Handlern abgerufen werden kann (siehe Abbildung 4.10). Für die technische Beschaffenheit einer Quelle wird keine Vorgabe gemacht. Als Quellen sind alle technischen Lösungen zur Bereitstellung von Software einbindbar.

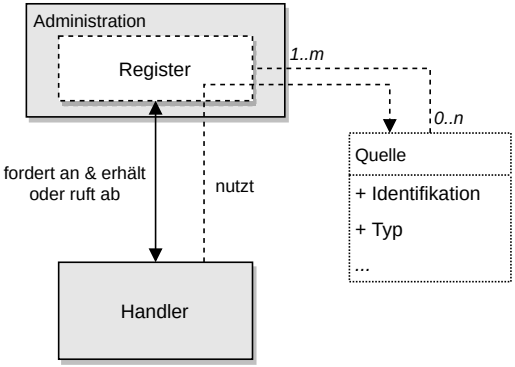


Abbildung 4.10: Zusammenhang Quellenregister in der Administrationsplattform und Handler

4.2.3 Logische Artefakte

4.2.3.1 Anwendungsmodelle

Anwendungsmodelle bilden das Konfigurations- und Integrationsmodell zum Aufbau eines Automatisierungssystems für eine Automatisierungsanwendung. Alle Anwendungsmodelle sind separat voneinander beschrieben und eindeutig identifizierbar. Anwendungsmodelle existieren in zwei Typen: Der erste (*Basis*) beschreibt abstrahiert von den später genutzten Automatisierungsressourcen Modelle für Daten, Funktionen und Ereignisse, und Konfigurationen der Ressourcen für eine Anwendung. Die Konfigurationen sind auf die abstrakten Bezeichnungen der Automatisierungsressourcen kartiert.

Mit dem zweiten Typ (*Instanz*) wird der Bezug zu den Automatisierungsressourcen aufgebaut und die Abstraktion aufgelöst. Ein solches Anwendungsmodell bezieht sich auf eine oder mehrere Basisanwendungsmodelle und löst mit einer Kartierung der Automatisierungsressourcen die abstrakten Referenzen auf. Weiterhin definiert sie weitere Konfigurationen, um möglicherweise notwendige systemspezifische Anpassungen umsetzen zu können. Diese Anwendungsmodelle beschreiben den tatsächlichen Aufbau eines Automatisierungssystems. Anwendungsmodelle werden durch die Administrationsplattform zum Abruf durch Handler bereitgestellt (siehe Abbildung 4.11).

Konfigurationsobjekte in den Anwendungsmodellen weisen Automatisierungsressourcen in einem Automatisierungssystem Konfigurationspakete und einzelne Konfigurationsele-

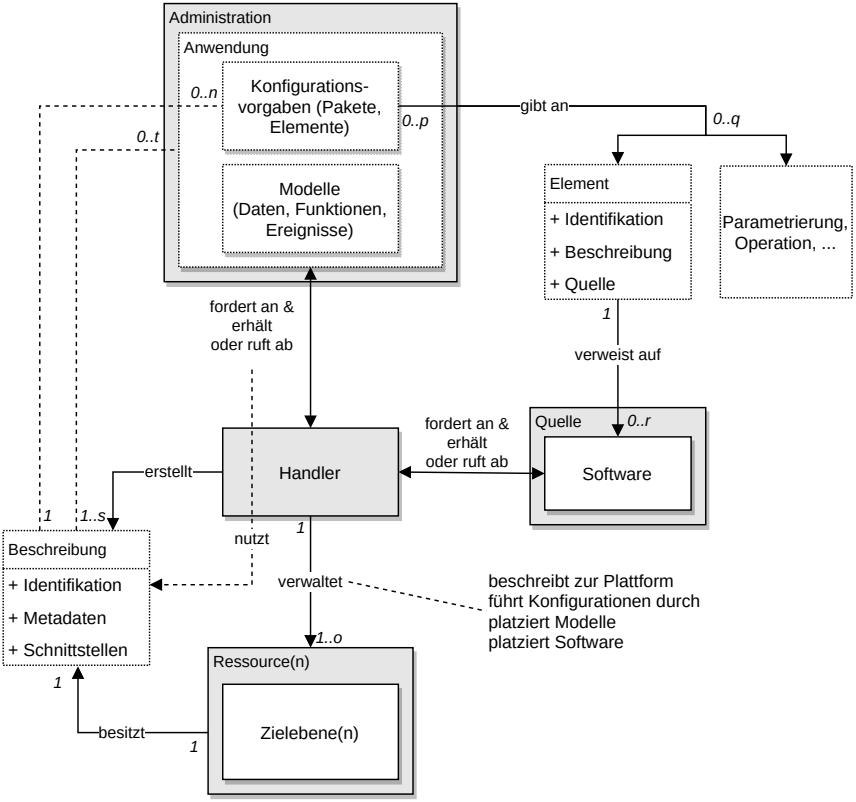


Abbildung 4.11: Handler im Beschreibungs- und Konfigurationszusammenhang

mente zu. Konfigurationspakete erhalten anwendungsspezifische Benennungen, um bei mehrfacher Verwendung im Automatisierungssystem unterschieden werden zu können.

4.2.3.2 Dateneinträge und Datenmodelle

Dateneinträge existieren immer im Bezug zu einem Datenmodell, innerhalb dessen sie über ihre Identifikation referenzierbar sind. Dateneinträge werden zur Laufzeit in der Datenstruktur instanziiert, sie dienen als Variablenstruktur für die Datenverarbeitung im System. Datentyp und Initialwert werden über die Modellierung vorgegeben. Dateneinträge können eine zusätzliche Angabe zum Basiszustand besitzen, auf die der aktuelle Wert zur Laufzeit

gebracht wird, wenn über einen bestimmten Zeitraum keine Änderung stattfindet. So können Datenabbilder anderer Automatisierungsressourcen auf einer Automatisierungsressource stabilisiert und das Gesamtverhalten gegen Verbindungsausfälle oder Logikprobleme abgesichert werden. Der Typ eines Dateneintrags wird mit einer Typangabe eindeutig identifiziert. Als Grundlage dienen die Typen der IEC 62541 (OPC UA, siehe Abschnitt 2.3.4). Die Typen können erweitert werden. Der Dateneintrag enthält ein Flag, das kennzeichnet, ob es sich um einen einfachen oder eine Liste an Werten handelt. Über Dateneinträge werden alle prozessbezogenen und dynamischen Einstellungen des Automatisierungssystems gestellt.

```
1  class Dateneintrag {
2    string Identifikation;
3    object wert;
4    Datentypen datentyp;
5    Bool istListe;
6  }
7
8  enum Datentypen { Int8, Int16, Int32, Int64, ... }
9
10 class Objekt : Dateneintrag {
11   List<string> Elemente;
12 }
```

Objekt 4.1: Entwürfe zu Klassen und Aufzählungstypen zu Dateneinträgen

Module, die Ausgänge stellen oder zur Wertevalidierung von Dateneinträgen verwendet werden, können über ihre Parametrierung oder über eine Erweiterung des jeweiligen Dateneintragmodells Validierungsinformationen erhalten. Diese Informationen geben den oder die zulässigen Zustände eines Dateneintragwertes vor, wobei Abhängigkeiten zu anderen Dateneinträgen vorgegeben werden können. Für die Validierungsinformationen wird keine einzuhaltende Vorgabe gemacht. Die Nutzung ist optional. Die Validierungsinformationen werden als serialisiertes Modell mitgegeben und müssen von den Modulen unterstützt werden. Module können damit unzulässige Dateneintragzustände erkennen oder verhindern, um unzulässige Eingriffe oder Ergebnisse abzufangen.

Objekte unterscheiden sich in ihrer Struktur durch ein zusätzliches Referenzierungsfeld. Die Identifikationen der im Objekt vorhandenen Dateneinträge sind hier gelistet. Die Dateneinträge müssen gesondert deklariert werden.

Ein Datenmodell beschreibt identifizierbar eine Menge eindeutiger Dateneinträge. Ein Datenmodell ist eine eigenständige logische Entität und über seine Identifikation identifizierbar. Datenmodelle können von Anwendungsmodellen, von Handlern respektive Automatisierungsressourcen sowie von Modulen definiert werden. Die Datenmodelle wer-

den bei initialer Ausführung in der Betriebszeit zusammengeführt und können über die Betriebszeit aktualisiert werden, solange gewährleistet ist, dass der sichere Betrieb des Automatisierungssystems erhalten bleibt.

```
1 class Datenmodell {
2   string Identifikation;
3   List<Dateneintrag> Einträge;
4 }
```

Objekt 4.2: Entwurf zu Klassen der Datenmodelle im Konzept

Datenmodelle sind aggregierbar, wobei die Dateneintragidentifikationen angepasst oder erweitert werden, um die Eindeutigkeit der Einträge zu wahren. Die Aggregation findet durch die Handler zur Konfigurationszeit statt.

4.2.3.3 Ereignis- und Funktionsmodelle

Ereignis- und Funktionsmodelle beschreiben Verbreitung von sowie Zugriff auf eine Menge ausgewählter Dateneinträge einer Datenstruktur. Sie sind nicht explizit auf eine Schnittstelle, die durch ein Modul betrieben wird, bezogen. Schnittstellen, die durch Module betrieben werden, kümmern sich automatisiert um die Instanziierung logischer Strukturen basierend auf den Ereignis- und Funktionsmodellen. Ereignisse und Funktionen wirken zwischen Automatisierungssystem und Umwelt. Sie realisieren keine Kommunikation innerhalb des Automatisierungssystems (siehe Abbildung 4.3). Um die Abbildung auf spezielle technologische Paradigmen kümmert sich das jeweilige Schnittstellen-Modul.

Ein Ereignismodell definiert Auslöser und Dateneinträge, deren Zustand veröffentlicht werden soll. Auslöser können gebunden an Dateneinträge oder unabhängig, zum Beispiel zyklisch, sein. Ein Funktionsmodell definiert einen Zugang auf Dateneinträge ähnlich einer Methode. Schnittstellen-Module realisieren die notwendige Logik gänzlich.

```
1 class Funktion {
2   List<string> Ziele;
3 }
4
5 class Ereignis {
6   List<string> Elemente;
7 }
```

Objekt 4.3: Entwürfe zu den Klassen zu Funktion und Ereignis

Funktions- und Ereignismodelle können innerhalb von Anwendungsmodellen und durch Module definiert werden. Ereignis- und Funktionsmodelle von Modulen, die auf einer

Automatisierungsressource allgemein und nicht anwendungsgebunden installiert sind, sind anwendungsübergreifend verfügbar.

4.2.3.4 Konfigurationspakete

Handler führen die technische Konfiguration ihrer Automatisierungsressourcen entsprechend ihrer zugewiesenen Anwendungen durch. Konfigurationspakete sind die hauptsächliche Definitionsstruktur für die technische Konfiguration. Sie beschreiben verallgemeinert Softwarekomponenten und deren Parametrierung, die dann zur Bildung eines Automatisierungssystems auf Automatisierungsressourcen verteilt werden.

```
1 class Konfigurationselement {
2   string Typ;
3   object Vorgang;
4 }
5
6 class Konfigurationspaket {
7   List<Konfigurationselement> Elemente;
8 }
```

Objekt 4.4: Entwürfe zu den Klassen zu Konfigurationselementen und -paketen

Konfigurationspakete beschreiben einen geschlossenen Anweisungsteil zum technischen Aufbau einer Automatisierungsressource. Sie sind nicht fest an eine Automatisierungsanwendung gebunden, sondern eine generisch nutzbare Einheit, beispielsweise zur Installation einer Softwareanwendung. Ihr Inhalt beschreibt eine Schrittreihenfolge, um eine logische Abfolge bei ihrer Abarbeitung einhalten zu können. Die Schritte der Reihenfolge können unterschiedliche erweiterbare Subtypen besitzen (die Installation einer Datei, eine Parametrierung, Erstellung eines Autostart-Eintrags et cetera), die vom Handler unterstützt werden müssen. In den Schritten werden, falls Daten oder Informationen von außen einfließen, Quellen referenziert.

Konfigurationspakete sind eindeutig identifizierbar und werden von Anwendungsmodellen über diese Identifikation referenziert. Konfigurationspakete geben keine Teilmodelle für Daten, Funktionen oder Ereignisse vor. Sie sind zum Aufbau der technischen Beschaffenheit einer Automatisierungsressource bestimmt. In einem Konfigurationspaket enthaltene Vorgänge sind nicht technologiebezogen, sondern abstrakt beschrieben. Der umsetzende Handler berücksichtigt die für die Automatisierungsressource durch Plattformarchitektur oder Betriebssystem gegebenen spezifischen Umsetzungen.

4.2.3.5 Beschreibung

Ein Handler erstellt für jede Automatisierungsressource, die er verwaltet, eine Beschreibung. Sie wird an die Administrationsplattform zyklisch oder bei Bedarf (initial, Änderungen) weitergegeben. Die Beschreibung identifiziert die Automatisierungsressource eindeutig, benennt den erstellenden Handler, liefert einen Mindestsatz an Metainformationen (Lesename, Systemarchitektur et cetera), nennt die platzierten Anwendungen sowie Komponenten und beschreibt die Schnittstellen der Automatisierungsressource. Die Beschreibung ist eine Basisklasse, die beliebig erweitert werden kann, um automatisierungsressourcenspezifische Eigenschaften wiedergeben zu können. Ihre Inhalte können von der Automatisierungsressource vorgegeben und vom Handler belassen, aktualisiert oder erweitert werden. Die serialisierte Beschreibung ist menschlesbar und kann als einzelne Datei abgelegt werden. Die Beschreibung ist der Adaptionspunkt für die Repräsentation als RAMI4.0-Verwaltungsschale entsprechend Abschnitt 4.2.1.6. Die Beschreibungen geben Systemeigenschaften, Systemzustände und Systemkonfigurationen wieder. Sie sind die Grundlage für das Verständnis der einzelnen und verteilten Systeme sowie die Anwendungsmodellierung mit den zur Verfügung stehenden Automatisierungsressourcen.

4.2.3.6 Abbild

Automatisierungsressourcen werden mit Modulen versehen, die Zustände der Datenstruktur zu bestimmten Zeitpunkten oder bei Änderungen erfassen und in Abbildern speichern. Die Module zur Abbilderstellung werden im Automatisierungssystem als Teil der Anwendungen verteilt, falls nicht fest installiert. Die Module bilden intern Abbildserien und speichern diese zeit- oder längenbasiert. Für die Zusammenführung aller Abbildserien wird die Administrationsplattform genutzt. Die Ablage erfolgt durch den Handler (Abbildung 4.13), der die Abbildserie mit der Ressourcenidentifikation versieht. Die Abbilder dienen zur Prüfung und Validierung des Automatisierungsressourcen- und -systemverhaltens.

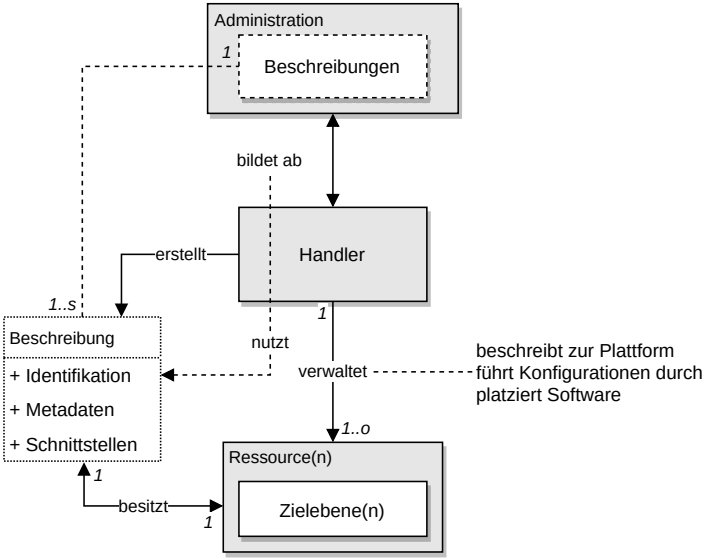


Abbildung 4.12: Beschreibung von Automatisierungsressourcen durch Handler

4.2.3.7 Schnittstelle

Schnittstellen werden mit einer Angabe von Identifikation und Automatisierungsressource gegenüber der Administrationsplattform beschrieben. Die dort abgelegten Beschreibungen können von anderen Automatisierungsressourcen abgefragt werden, um die Kommunikation innerhalb des Automatisierungssystems zu initiieren, ohne initial harte Adressinformationen zu besitzen. Schnittstellenbeschreibungen werden bei Änderungen zur Laufzeit

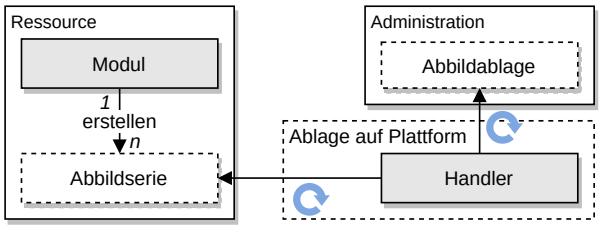


Abbildung 4.13: Ablage von Abbildserien durch Handler

aktualisiert. Die Beschreibungserstellung erfolgt durch Module oder durch den Handler mittels Informationen, die an Module gebunden sind.

Schnittstellen werden über Module implementiert. Sie können über ein festes Modul Standardbestandteil einer Automatisierungsressource sein, beispielsweise, wenn spezifische Kommunikationshardware vorhanden ist, die über das Framework integrierbar ist. Der Handler sammelt die Schnittstellenbeschreibungen, die auf einer Automatisierungsressource anfallen und bündelt sie. Schnittstellenbeschreibungen folgen einem einheitlichen, protokoll- und technologieunabhängigen Basisformat und werden für spezifische Angaben erweitert. In der Schnittstellenbeschreibung werden auch Informationen oder Verweise darauf zu Hard- und Software sowie zur Bedienung abgelegt.

4.3 Anwendung

4.3.1 Aufbau und Ausführung von Modulen

Module sind abgeschlossene Softwarestrukturen, die vorzugsweise als eigene und aktive Softwareanwendungen erstellt werden. Sie besitzen eine lokale exklusive Parametrierung, über die für den Betrieb notwendige statische Informationen, beispielsweise zu den Aktivzuständen, eingebracht werden. Die Module werden vorzugsweise anwendungsinvariant erstellt und anwendungsspezifisch parametriert. Sie können auf dem System fest installiert sein. Ihre interne Logik folgt einem Eingabe-Verarbeitungs-Ausgabe-Muster, das in eine dauernde Ausführung (bei eigenständiger Anwendung zum Beispiel über eine *while*-Schleife) eingebettet ist. Sollte keine dedizierte Ausführungsplanung auf der Automatisierungsressource arbeiten, folgt ein Modul kontinuierlich den Zustandsinformationen in der lokalen Datenstruktur und führt die eigene Logik nur bei Erreichen eines gültigen Aktivzustandes aus. Dem Modul steht eine interne programmatische Kopie der Datenstruktur als Arbeitsbasis für die interne Logik zur Verfügung. Die Kopie wird bei Erreichen eines Aktivzustands aktualisiert und ihr geänderter Stand bei Abschluss der internen Schleife in die Datenstruktur zurückgeführt. Außerhalb ihrer an den lokalen Zustand gebundenen Standardlogik kann das Modul Interruptfunktionen implementieren, die an einzelne Dateneinträge in der Datenstruktur gebunden sind. Die den betroffenen Dateneinträgen entsprechenden

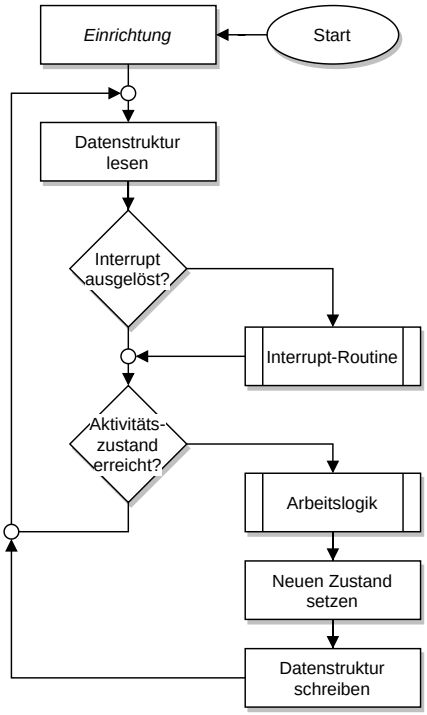


Abbildung 4.14: Konzept zum Programmablauf eines Moduls im Ansatz

Abbildungsteile werden dauernd aktualisiert, um die Interruptfunktionalitäten realisieren zu können.

Ausführungssteuerung

Die Ausführung der Module einer Anwendung auf einer Automatisierungsressource läuft in einer festen lokalen Reihenfolge ab (lokale Zyklusorientierung, siehe auch Anhang A.1.2). Anwendungsbezogen werden in der lokalen Datenstruktur einer Automatisierungsressource Zustandseinträge angelegt, die zur Ausführungssteuerung dienen. Module verfügen über Konfigurationen zu Ein- und Ausgangszustandspaaren. Bei Erreichen eines hinterlegten Eingangszustands im Zustandseintrag wird die Logik aktiviert (siehe 4.14). Nach Abschluss der Logik wird durch das Modul der zum Paar gehörende Ausgangszustand in den Zustandseintrag geschrieben. Aus Ausführungssicht sind Module in fünf Ausführungskategorien

zu unterteilen (siehe Abbildung 4.15), wobei ein Modul in mehreren Kategorien aktiv sein kann: Werteerhalt im Automatisierungssystem, Werteerhalt über Eingänge und externe Systeme, Verarbeitung, Werteausgabe über Ausgänge und externe Systeme, Werteausgabe im Automatisierungssystem. In der Industrieautomatisierung ist davon auszugehen, dass die physischen Ein- und Ausgänge als besonders kritisch gelten, weswegen dafür zuständige Module auf einer Automatisierungsressource unmittelbar vor und nach verarbeitenden Modulen auszuführen sind, um kritische Laufzeiten minimieren zu können.

Bei allen Kategorien gilt, wenn mehrere Module einer Kategorie vorhanden sind, eine Unterabfolge. Nicht-interrupt-bezogene Modulogik ist in Frameworkarchitekturen stets sequentiell auszuführen. Bei interner Ausführungsaktivierung sind Module auf dem System dauernd aktiv und überwachen Zustandseinträge sowie Interrupteinträge oder Interrupteingänge, wenn die Module physische Eingänge oder externe Systeme anbindet. Über Interrupteinträge oder Interrupteingänge werden kritische Reaktionen implementiert.

Der interne Ablauf eines Moduls ist strukturiert in Lese-, Verarbeitungs- und Schreibteil. Lesen und Schreiben bezieht sich auf die Werteübergaben in das und aus dem Datenabbild, das im Modul liegt. Bei verarbeitenden Modulen finden die Werteübergaben in Bezug auf die lokale Datenstruktur statt, bei Ein- und Ausgabemodulen in Bezug auf lokale Datenstruktur, Datenquellen und Datenziele. Datenquellen und -ziele sind Kommunikationspartner im Automatisierungssystem sowie physische EA (Sensorik und Aktorik) und externe Systeme, die an die Automatisierungsressource angeschlossen sind. Module zum Datenerhalt aus dem Automatisierungssystem stehen am Anfang des Ausführungszyklus. Module, die physische Eingänge und externe eingebende Systeme anbinden, folgen. Diese Reihenfolge ist einzuhalten, damit die verarbeitenden Module, über die Automatisierungslogik implementiert wird, möglichst nah gegen den physischen Systemzustand arbeiten und falsche Werte durch kommunizierende Module und lange Reaktionszeiten vermieden werden. Analog erfolgt die Aktivierung der Ausgabemodule für physische Ausgänge und externe ausgebende Systeme direkt nach den verarbeitenden Modulen, um Reaktionszeiten kleinstmöglich zu halten.

Kommunikationsmodule, die aus dem Automatisierungssystem Werte erhalten und in die lokale Datenstruktur schreiben, erhalten eingehende Werte kontinuierlich und puffern diese in einem Zwischenspeicher. Die Werteänderungen werden von den Modulen bei

Erreichen ihres Aktivzustandes aus ihrem jeweiligen Zwischenspeicher in die Datenstruktur geschrieben und der Zwischenspeicher geleert. Kommunikationsmodule, die Werte verteilen, führen dies bei Erreichen ihres Aktivzustandes durch. Sie lesen den aktuellen Zustand der Datenstruktur und verteilen alle oder geänderte Werte.

4.3.2 Aufbau einer Automatisierungsanwendung

Die Automatisierungsanwendung steuert einen oder mehrere physische Prozesse über die physischen Schnittstellen des Automatisierungssystems. Sie besteht aus einem oder mehreren Modulen, die Transitionen zwischen physischen Zuständen anhand von Dateneinträgen in den Datenstrukturen herstellen. Sind es mehrere Module, werden sie über die Automatisierungsressourcen des Systems verteilt. Die gelesenen und gestellten Dateneinträge werden über anwendungsspezifische oder feste Module mit den Ein- und Ausgängen des Automatisierungssystems verbunden.

Die Module der Automatisierungsanwendung können harte spezifische Implementierungen sein oder Ausführungsumgebungen, die dateiorientiert aus Konfigurationen Skripte oder Programme erhalten. Automatisierungsanwendungen werden über die Anwendungsmodelle definiert und sind möglichst generalisiert zu halten. Über Ketten von Anwendungsmodellen können sie spezifiziert werden. In den Anwendungsmodellen werden Datenzusammenhänge über die Datenstruktur definiert, aber keine innere Logik der Automatisierungsanwendung.

Die Erstellung von Anwendungsmodellen ist kein Bestandteil des Frameworks. Es definiert die Grundmodelle und die logischen Zusammenhänge, aber keine Modellierungssysteme zur Modelldefinition, da hierfür belastbares Wissen über ein definiertes neues oder bekanntes Entwicklungsvorgehen benötigt würden. Ausgegangen wird bei der Nutzung von einem üblichen schrittweisen Vorgehen wie in Abschnitt 2.1.4 vorgestellt. Die Ausgangslage für den Aufbau einer Automatisierungsanwendung ist dann die Definition des Automatisierungsproblems und der dafür notwendigen Fähigkeiten sowie Abläufe. Die Definition bildet die Grundlage zur Zusammenstellung bestehender und noch zu erstellender Module sowie physischer Systeme. Mit diesen Informationen sind die ersten Modellierungen im Anwendungsmodell hinsichtlich Konfiguration und Ressourcen festlegbar. Im Anschluss können

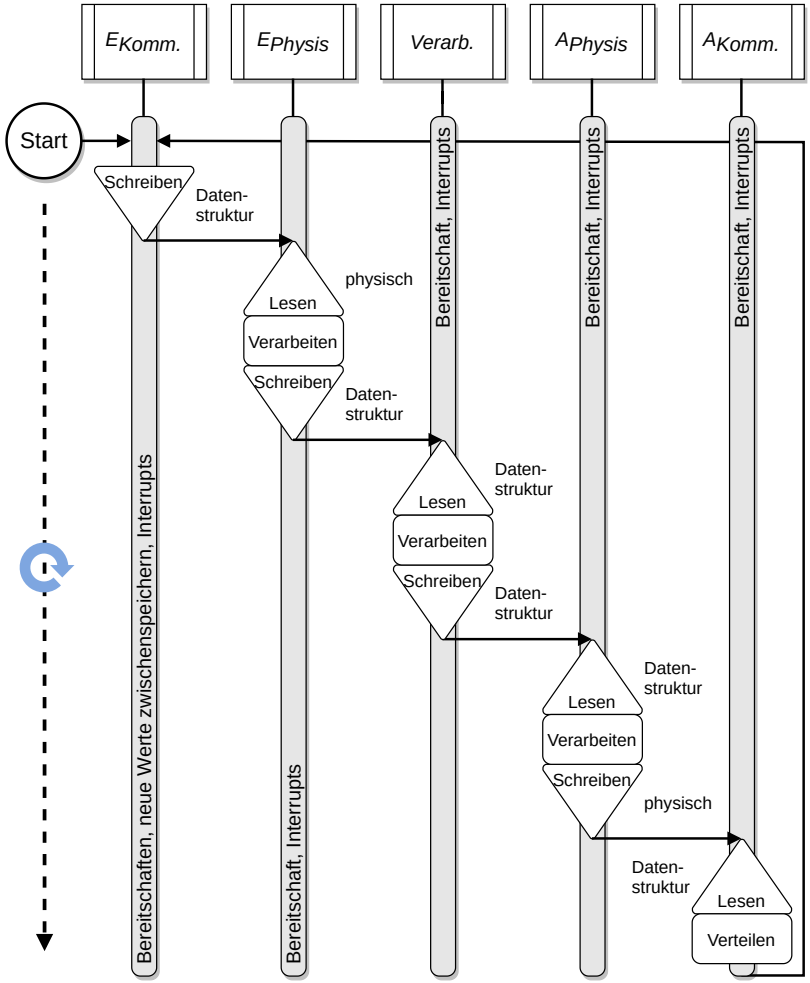


Abbildung 4.15: Ausführungsablauf auf einer eingerichteten Automatisierungsressource

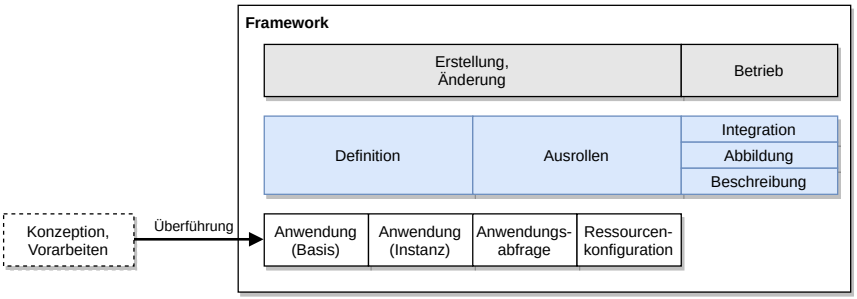


Abbildung 4.16: Der Lebenszyklusansatz im Framework mit den Zyklusabschnitten

Modulkonfiguration, zusätzliche Automatisierungsressourcen, Ressourcenkartierung und lokale Ausführungsabfolgen festgelegt werden. Aus erstellten und erprobten Automatisierungssystemen können Modellteile hinsichtlich Generalisier- und Wiederverwendbarkeit entnommen und in Basismodelle überführt werden.

Die Sammlung und Bestimmung von Anforderungen, Maschinen- und Anlagenspezifikationen, Hardwareimplementierungen und weitere Grundplanungen für eine Anwendung werden vom Framework nicht unmittelbar bedient. Sie werden vom Verständnis des Lebenszyklus einer Automatisierungslösung erfasst, liegen aber außerhalb des Frameworks, siehe Abbildung 4.16. Eigenschaften für diese Vorgänge ergeben sich ausschließlich indirekt aus Festlegungen durch das Framework. Softwareentwicklungsarbeiten, die der Aufstellung einer Automatisierungslösung vorhergehen, werden durch Standardimplementierungen unterstützt. Für die Frameworkverwendung sind zwei Lebenszyklusabschnitte zu betrachten: Erstellung/Änderung und Betrieb. Die Abfolge ist nicht strikt von links nach rechts zu interpretieren. Vom Betrieb kann jederzeit in einen Teil der Erstellung/Änderung zurückgekehrt werden (Rekonfiguration).

Die Erstellung/Änderung unterteilt sich in eine Definitions- und eine Ausrollphase. In der Definitionsphase werden die vorgebenden Modelle für die Ressourcenkonfigurationen erstellt. Konzeptionierungs- und Spezifizierungsarbeiten, die im Vorfeld geleistet wurden, werden in der Definitionsphase in Modelle in Frameworkformate überführt. Die Ausrollphase im Anschluss an die Definitionsphase besteht aus der Handlerabfrage nach umzusetzenden Anwendungen für ihre Automatisierungsressourcen, gefolgt von der Konfiguration der Automatisierungsressourcen entsprechend der Anwendungen. Im Betrieb gewährleistet

das Framework die Integration der Automatisierungsressourcen nach den Anwendungen sowie die dauernde Beschreibung und Abbildung der Automatisierungsressourcen und ihrer Elemente über die Handler.

4.3.3 Validierung von Verhalten und Rechtzeitigkeit

Eine Verhaltensbestimmung basiert auf der Überwachung der Systemreaktion eines technischen Systems. Über das gezielte Setzen von Eingangszuständen und die Beobachtung der Reaktion an den Ausgängen kann das System gegen die Anforderungen validiert werden. Bei einem verteilten System ist dieser Ansatz aufgrund der besonderen Beschaffenheit erschwert, bedingt durch die mögliche große räumliche Verteilung und die unterschiedlichen Systemausprägungen im verteilten System. Die theoretische Ermittlung von Zuverlässigkeit und Rechtzeitigkeit ist nicht relevant: Bedingt durch die Rekonfigurierbarkeit und die Hard- und Softwareunabhängigkeit sind Systemkomplexität und -beschaffenheit nicht festlegbar, weswegen zuverlässige Ermittlungsmodelle nicht pauschal erstell- und validierbar sind. Möglichkeiten zur praktischen Ermittlung ohne physische Erweiterungen und mit der Fähigkeit, verteilte Systeme zu beurteilen, sind deswegen notwendig.

Um die Verhaltensbestimmung im Automatisierungssystem zu ermöglichen, wird ein softwarebasierter Ansatz angewendet: Die Zustände der Datenstrukturen werden über die Abbildfunktionen verfolgt. Module für physische Ein- und Ausgänge werden mit Simulationsversionen ersetzt oder ihre Bindung an die tatsächlichen physischen Ein- und Ausgänge über Flag-Dateneinträge hin zu virtuellen Daten gesteuert. Simulationsversionen der Module, die Eingänge erfassen, dienen als Manipulationsschnittstelle. Simulationsversionen der Module, die Ausgänge stellen, erlauben gleichbleibende interne Logik ohne das tatsächliche Stellen von Ausgängen. So können Systemreaktionen auf Eingangssignale ohne physische Eingriffe oder Auswirkungen untersucht werden.

Die Automatisierungsressourcen müssen zeitsynchronisiert sein, vorzugsweise über PTP oder NTP. Die Validierung muss gebunden an die tatsächliche Anwendung durchgeführt werden. Die erstellten Abbilderfolgen der einzelnen Systeme dienen als Datengrundlage. Sie werden mit Markierungen versehen, an denen die erstellende Automatisierungsressource erkennbar ist, und zusammengeführt. Als Validierungsgrundlage wird die parallele Visuali-

sierung von manuell gewählten, prozessrelevanten Dateneinträgen aus den Abbilderfolgen sowie die datengetriebene Analyse von Zeitabständen und der Verlässlichkeit des Eintretens bestimmter Zielzustände vorgesehen. Aufgrund der anwendungsspezifischen Verhaltensvorgaben und der möglichen aufbauenden Entwicklung von zukünftigen Verfahren wird kein fester Vorschlag als Frameworkbestandteil eingeführt.

Methode

Ein Modul auf einer Automatisierungsressource ändert entsprechend einer Experimentplanung Dateneintragwerte. Die Beschaffenheit der Änderungen ist nicht limitiert, es wird aber ein Standard definiert:

```
1 class Änderung {
2   string Zieldateneintrag;
3   object Wert;
4 }
5
6 class Änderungsplan {
7   Dictionary<uint64, List<Änderung>> ÄnderungenNachZeit;
8 }
```

Objekt 4.5: Entwurf zur Struktur von Änderungsplanen zur Versuchsdurchführung

Zu festgelegten Zeitpunkten in Mikrosekunden nach Start wird eine bestimmte Anzahl Änderungen in angegebenen Zieldateneinträgen durchgeführt. Die Änderungsplanung wird als dateigebundene Standardparametrierung im Konfigurationspaket eines manipulierenden Moduls übergeben und kann über Konfigurationsvorgaben angepasst werden. Erfolgen von mehreren Modulen Änderungen, beispielsweise um von zwei oder mehr Automatisierungsressourcen aus das Automatisierungssystems zu beeinflussen, müssen mehrere Änderungspläne mit abgestimmten Zeitpunkten eingebunden werden.

Auf einer oder mehreren Automatisierungsressourcen werden über Module die Zustände der Datenstruktur über die Abbildbildung protokolliert. Änderungen der Dateneintragwerte werden referenzierbar zum Dateneintrag gespeichert mit Zeitpunkt und neuem Wert. Die Protokollierung wird über einen eigenen Dateneintrag gesteuert. Nach Protokollierungsbeginn wird der wertegebende Vorgang gestartet und beendet.

Auswertung

Angewendet werden Kennzahlen, die mit Methoden der deskriptiven Statistik erstellt werden. Nach Durchführung einer Messwertmenge werden Mittelwert μ (MW in Abbildung)

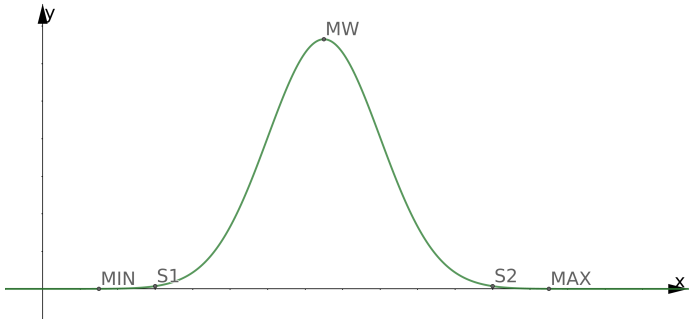


Abbildung 4.17: Graph einer Wahrscheinlichkeitsdichtefunktion mit charakteristischen Punkten

und Standardabweichung σ aus der Menge berechnet sowie Minimal- und Maximalwert für die Messwertmenge bestimmt (MIN, MAX). Die qualitative Visualisierung erfolgt beispielsweise mit der Wahrscheinlichkeitsdichteverteilung (siehe Prinzipabbildung 4.17). Am Maximalwert wird die Anforderungseinhaltung bewertet. An den Kennwerten $\mu \pm 3\sigma$ kann eine qualitative Bewertung für das grundsätzliche Verhalten ermittelt werden (S1, S2).

Bewertung der Rechtzeitigkeit

Grundgegenstand der Rechtzeitigkeitsbewertung eines Automatisierungssystems ist die zeitliche Differenz zwischen Signal und Systemantwort (Reaktionszeit, siehe Abschnitt 2.1.3): Bewertet wird die Zeitdauer zwischen dem Auftreten eines bestimmten Zustands und der Systemreaktion. Eine Übersicht gibt Abbildung 4.18. Bewertungen müssen innerhalb des Automatisierungssystems und mit Bezug auf andere Systeme möglich sein. Aus zwei Protokollen werden die Änderungszeitpunkte zweier Dateneinträge verglichen. Wenn die Kommunikationsdauer untersucht werden soll, werden die zwei Instanzen desselben logischen Dateneintrags auf verschiedenen Automatisierungsressourcen miteinander verglichen (Ermittlung der Zeiten t_1 und t_2). Soll eine Laufzeit, eine Verarbeitungskomponente einschließt, verglichen werden, können verschiedene logische Dateneinträge auf derselben oder verschiedenen Automatisierungsressourcen miteinander verglichen werden (Summierung ausgewählter Zeiten). Zur Einhaltung harter Zeitanforderungen darf der Maximalwert in der ermittelten Serie den Anforderungswert nicht überschreiten.

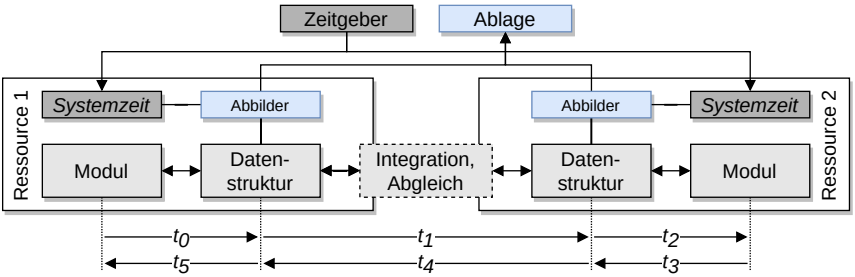


Abbildung 4.18: Zeitenkette in der Integration zwischen verteilten Systemen

4.4 Konzept zur VWS-Kartierung

Für die Kartierung wird das VWS-Metamodell nach Plattform Industrie 4.0 2022 als eine Ausgangsbasis zur Adaption an das VWS-Konzept aufgegriffen. Die relevanten Modellelemente sind in Abbildung 4.19 zu sehen. Die eigentliche VWS (*AssetAdministrationShell*) verfügt über Identifikations- und Beschreibungselemente (*Identifiable*, *AssetInformation*, *SpecificAssetId*) sowie aggregierte Funktions-, Informations- und Datenelemente (*SubmodelElement*), die strukturiert in Teilmodellen (*Submodels*) gebündelt sind. Automatisierungs- und systembezogene Eigenschaften und Fähigkeiten sind hinter den Beschreibungen der Teilmodelle verborgen.

Die interne Struktur der VWS ist für das Framework nicht bedeutend, auch weil es allgemein keine speziellen Anforderungen an die programmatische Umsetzung eines Softwaresystems, das als VWS zu repräsentieren ist, stellt. Hauptziel der VWS ist die Einführung eines Standardformats mit standardisierten Beschreibungs- und Modellstrukturen zur Repräsentation technischer Systeme in der Produktion. Um das Framework an VWS-Umgebungen zu adaptieren, müssen die ausgewählten Frameworkbestandteile als passive VWS oder VWS-Elemente repräsentierbar werden. Für eine initiale Adaption an VWS-Umgebungen werden Automatisierungsressourcen und Automatisierungssysteme als relevante Frameworkelemente ausgewählt. Sie stellen die beiden im Produktionseinsatz aktiven Typen technischer Systeme des Frameworks. Weil davon ausgegangen muss, dass die zum Zeitpunkt der Erstellung der Arbeit verfügbaren Referenzdokumente noch keinen endgültigen oder produktionsfähigen Stand der VWS zeigen, sind die Kartierungsansätze als konzeptio-

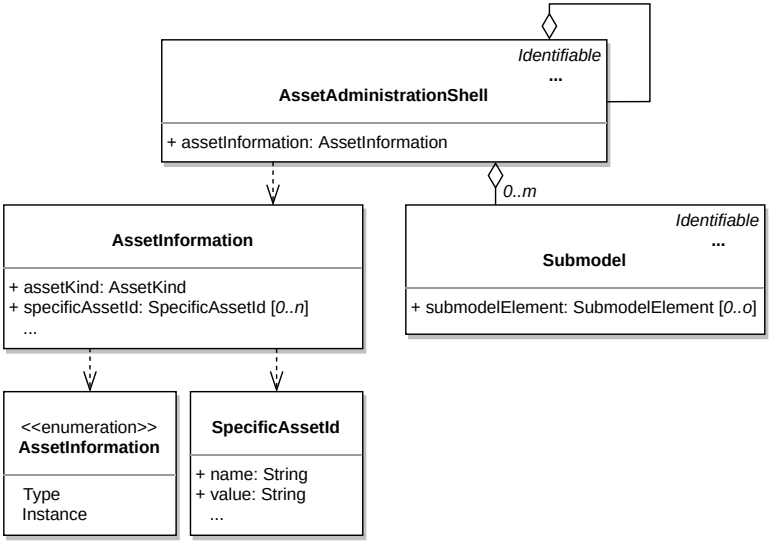


Abbildung 4.19: Teil des Metamodells der Verwaltungsschale nach Plattform Industrie 4.0 2022 mit relevanten inneren Elementen

nell zu sehen. Zudem kann noch keine Kartierung auf niedriger Ebene oder in Modelldetails vorgenommen werden. Verstanden wird die VWS-Kartierung als die Möglichkeit, aus frameworkkonformen Beschreibungen VWS-Strukturen in VWS-konformen Formaten, beispielsweise als AASX-Paket oder serialisiert als JSON oder XML (siehe Plattform Industrie 4.0 2022), zu erstellen, um die Frameworkartefakte in VWS-Umgebungen einbetten zu können. Automatisierungssystem und Automatisierungsressource sind als VWS vorrangig Instanz-VWS, wobei für ihre Grundtypen Typ-VWS denkbar sind, die aus Basisbeschreibungen ohne Anwendungsbezug erstellt werden.

Die Kombination aus Automatisierungsressource und VWS oder Automatisierungssystem und VWS kann als I4.0-Komponente im Sinne des RAMI4.0 (siehe Abschnitt 3.1) gesehen werden. Die Attribute der Automatisierungsressource oder des Automatisierungssystems können in der RAMI4.0-Abstraktion eingebettet werden, indem ihre festen Eigenschaften, Module und Datensysteme auf den verschiedenen Ebenen verortet werden.

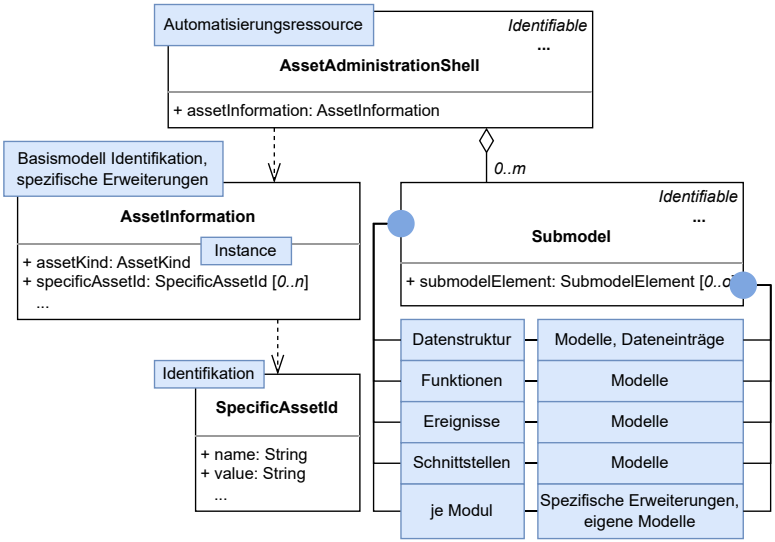


Abbildung 4.20: Kartierungsübersicht einer Automatisierungsressource und seiner Elemente auf die Kernelemente des VWS-Metamodells

Automatisierungsressource und VWS

Die VWS entspricht einer Automatisierungsressource. In der *AssetInformation* wird die Identifikation der Automatisierungsressource eingehängt sowie alle ressourcenspezifischen Erweiterungen des Beschreibungsmodells, die Informationen zur Automatisierungsressource wiedergeben. Als Teilmodelle werden vier immer vorhandene Teilmodelle eingeführt (*Datenstruktur*, *Funktionen*, *Ereignisse*, *Schnittstellen*) sowie ein Teilmodell je Modul. Das Datenstruktur-Teilmodell vertritt die Gesamtmenge der angelegten Dateneinträge auf der Automatisierungsressource sowie die verwendeten Datenmodelle. Die Teilmodelle zu Funktionen, Ereignissen und Schnittstellen beschreiben die vorhandenen Modelle, nach denen Eigenschaften auf der Automatisierungsressource vorhanden sind. Die spezifische Beschreibung jedes Moduls auf dem System wird als eigenes Teilmodell eingebunden. Die modulspezifischen Modelle zu Daten, Funktionen, Ereignissen und Schnittstellen werden als Kindelemente im Teilmodell des Moduls zusätzlich zu den Aggregationen in den immer vorhandenen Teilmodellen aufgeführt.

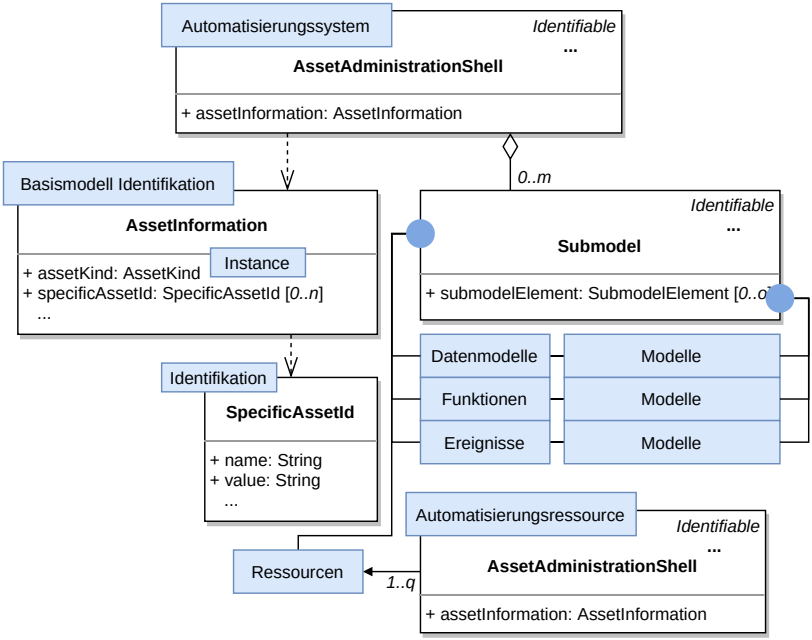


Abbildung 4.21: Kartierungsübersicht eines Automatisierungssystems und seiner Elemente auf die Kernelemente des VWS-Metamodells

Automatisierungssystem und VWS

Die VWS entspricht einem Automatisierungssystem. In der *AssetInformation* wird die Identifikation des Automatisierungssystems aus dem Instanzanwendungsmodell eingebunden. Als Teilmodelle werden vier immer vorhandene Teilmodelle eingeführt: *Datenmodelle*, *Funktionen*, *Ereignisse* und *Ressourcen*. Alle Teilmodelle zu Daten, Funktionen und Ereignissen sind Aggregationen der über die Anwendungsmodelle definierten Bestandteile sowie den Entsprechungen der Automatisierungsressourcen. Über das Ressourcenfeld werden die VWS der am Automatisierungssystem beteiligten Automatisierungsressourcen referenziert.

Schnittstellen und VWS

VWS und VWS-Umgebungen verfügen über modell- und funktionsorientierte Schnittstellen, kartiert auf VWS/REST-API, die von VWS, VWS-Teilmodellen oder VWS-Umgebungssystemen (VWS-, Schnittstellen-Repository) bereitgestellt werden (Plattform Industrie 4.0 2021). Die

Plattform-Schnittstellenfunktionen zur Verwaltung von serialisierten Modellen und Beschreibungen im Framework sind kartierbar auf die Schnittstellen-Repository-Funktionen und Subfunktionen zum Austausch von Modellteilen in der VWS-Architektur. Bei Kartierung eines Automatisierungssystems auf eine VWS können *Invoke*-Operationen der VWS auf Funktionen nach Abschnitt 4.2.1.4 adaptiert werden.

5 Das Integrations- und Steuerungsframework *ISCI*: Umsetzung

Aufgeteilt in Beschreibungen zu den logischen Elementen für Modelle, Objekte und Software, und technisch nahen Beschreibungen der Umsetzungen in Softwareanwendungen folgt nun die Ausarbeitung der Konzeptelemente zu umsetzungsorientierten Strukturelementen. Diese Strukturelemente dienen als Implementierungsbasis, um die im Konzept definierten Bestandteile von ISCI in ein Softwarerahmenwerk zu überführen. Die Beschreibung erfolgt mit implementierungsnahem Pseudocode zur Erläuterung von Klassen, Programmteilen und Applikationen. Die Klassenvariablen und -methoden sind für eine einfache Interpretation der Arbeit als *public*, also öffentlich zugreifbar, zu verstehen.

5.1 ISCI-Klassenstruktur

Aus den Klassen des Frameworks instanziierte Objekte und Modelle greifen für die Logikstrukturen aufeinander zu, sind aber einzeln les- und interpretierbar. Neben der Deskription von logischen Strukturen über dateibasierte Serialisierungen dienen sie zur Parametrierung und Instanziierung in programmatischen Umgebungen. Alle Klassen sind nicht als abgeschlossen zu verstehen, sondern als zueinander in Relation gesetzte Mengen von Standardbestandteilen und Erweiterungen. Über Erweiterungen sind Anpassungen oder zusätzliche Informationen für spezifische Implementierungen, die durch die Anwendung oder Technologien notwendig werden, möglich. Die Basisklassen sind in einer Bibliothek zusammengefasst. Abbildung 5.1 zeigt ihre Bestandteile. Für die Serialisierung wird die Verwendung von JSON vorgesehen, um die Menschlesbarkeit zu ermöglichen. Alle Serialisierungen sind als jeweils eigenständige Datei abzulegen, wenn auf der Administrationsplattform oder einer fähigen Automatisierungsressource gearbeitet wird.

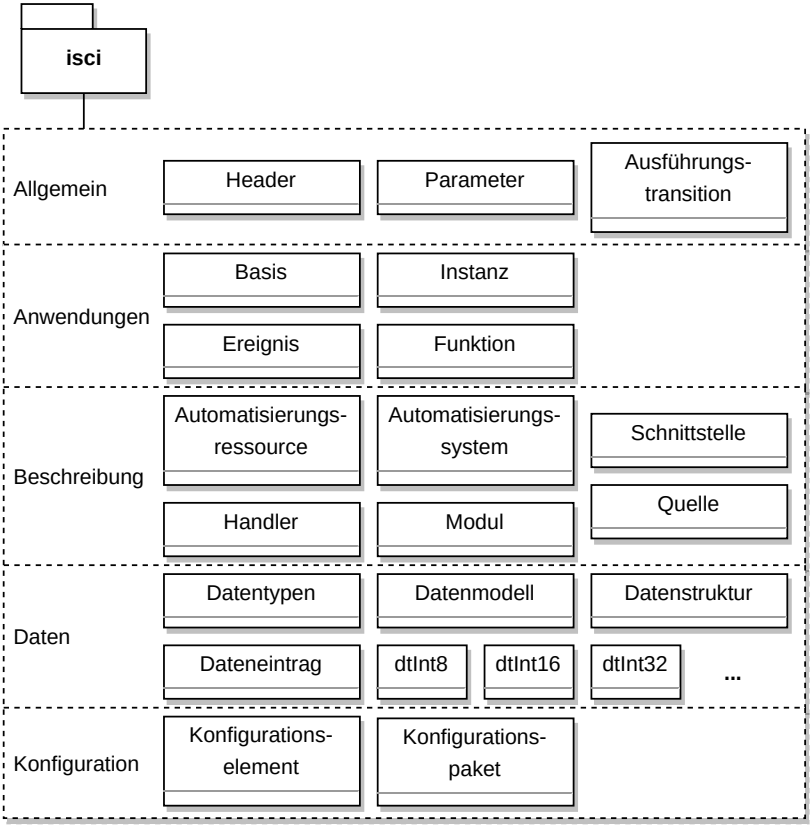


Abbildung 5.1: Überblick der wesentlichen Klassen in der Klassenstruktur der ISCI-Umsetzung

5.1.1 Allgemein

Header

ISCI-Klassen, von denen Instanzen ausgetauscht oder von mehreren technischen Systemen genutzt werden, verfügen über einen Header (Codeteil 5.1) als Klassenbasis, die mittels Vererbung oder direkte Einbettung einzubinden ist, um Identifikation und Metainformationen einheitlich zu repräsentieren sowie den Klassentyp zu kennzeichnen. Die Basisattribute, die von der Headerklasse geliefert werden, dürfen in erbenden Klassen nicht ersetzt werden.

```
1 class Header {
2   string Identifikation;
3   string Name;
4   string Beschreibung;
5   string Typ;
6 }
```

Objekt 5.1: Header-Klasse als Basisklasse für alle weiteren zur Angabe von Metadaten und Typ

Parameter

Module erhalten für ihre Parametrierung Parameterobjekte, die mit Modulen mitgeliefert oder von Modulen erstellt werden. Handler können im abgelegten Objekt Anpassungen über lokale Standardeinstellungen oder Konfigurationselemente vornehmen, um das Modul mit betriebsnotwendigen Informationen zu versehen. In der Umsetzung ist die zugehörige Klasse *Parameter* ausgerichtet auf eingebettete Automatisierungsressourcen mit Dateisystemen. Für andere Ressourcenstrukturen sind die Variablen der Umsetzungsklasse als Ausgangsbasis für notwendige eigene Integrationen heranziehbar. Es ist anzunehmen, dass die Parameterklasse in ihrer Standardform selten verwendet wird und dass eine über Vererbung mit der Parameterklasse erweiterte modulspezifische Klasse der Regelfall ist. Die Parameterklasse enthält auch die Angabe der Zustandstransitionen zur Modulausführung über den lokalen Zustand auf der Automatisierungsressource.

```
1 class Parameter {
2   string Ressource;
3   string Identifikation;
4   string OrdnerAnwendung;
5   string OrdnerDatenstruktur;
6   string Anwendung;
7   string OrdnerDatenmodelle;
8   string OrdnerEreignismodelle;
9   string OrdnerFunktionsmodelle;
10  string OrdnerSchnittstellen;
11  string OrdnerBeschreibungen;
12  Ausführungstransition[] Ausführungstransitionen;
13
14  Parameter(string datei);
15 }
```

Objekt 5.2: Klasse zur Abbildung der Parameter eines Moduls. Die Klasse verfügt über einen Konstruktor zur Instanziierung aus einem serialisierten Objekt in einer Datei

Ausführungstransition

Objekte der Klasse *Ausführungstransition* stellt die Informationsstruktur für die Zustandsüberführung der lokalen Ausführungssteuerung auf einer Automatisierungsressource. Jede

Ausführungstransition beschreibt einen Eingangs- und einen Ausgangszustand. Die durch ein abgeleitetes Objekt beschriebene Zustandsüberführung wird zur Betriebszeit durch ein Modul durchgeführt. Bei identischem Wert von Eingangszustand-Attribut und lokalem Zustandsdateneintrag wird die Prozesslogik im Modul ausgeführt, das nach Abschluss den Wert des Ausgangszustand-Attributs in den lokalen Zustandsdateneintrag schreibt.

```
1 class Ausführungstransition {
2   int Eingangszustand, Ausgangszustand;
3 }
```

Objekt 5.3: Klasse zur Vorgabe einer Zustandsüberführung für die Ausführungssteuerung

5.1.2 Anwendungen

Basis und Instanz

Zur Erstellung von Automatisierungssystemen werden Anwendungsmodelle definiert, die eine Struktur eines Automatisierungssystems beschreiben. Dem Konzept entsprechend werden zwei Klassen für Anwendungsmodellen implementiert: *Basis* sowie *Instanz*. Die Klassen sind nicht zu erweitern. Alle Anwendungsmodelle können andere Anwendungsmodelle referenzieren, um aufeinander aufbauen zu können (*ReferenzierteModelle*). Basisanwendungsmodelle definieren plattformabstrakte, wiederverwendbare Strukturen, die einen Teil eines Automatisierungssystems stellen. Sie beschreiben vorzunehmende Konfigurationen (*Konfigurationspakete*, *Konfigurationselemente*) in Abhängigkeit von Pseudonymen für Automatisierungsressourcen. Die Pseudonyme werden bei der Erstellung über die Instanzanwendungsmodelle aufgelöst. Alle Anwendungsmodelle können Datenmodelle, Funktionen und Ereignisse für das Automatisierungssystem in Listen vorgeben.

```
1 class Basis : Header {
2   List<string> ReferenzierteModelle;
3
4   //Automatisierungsressource --> Lokale Identifikation -->
5   //Identifikation Konfigurationspaket
6   Dictionary<string, Dictionary<string, string>> Konfigurationspakete;
7   //Automatisierungsressource --> Konfigurationselemente
8   Dictionary<string, List<Konfigurationselement>> Konfigurationselemente;
9
10  List<Datenmodell> Datenmodelle;
11  List<Funktion> Funktionen;
12  List<Ereignis> Ereignisse;
13 }
```

Objekt 5.4: Klasse für die Vorgabe einer Anwendungsbasis

Instanzanwendungsmodelle basieren auf Basisanwendungsmodellen und erweitern diese um die Auflösung der Pseudonyme für Automatisierungsressourcen (*Ressourcenkartierung*) sowie um die zu verteilenden Ausführungstransitionen im System, über die Ausführungsreihenfolgen auf den Automatisierungsressourcen für die jeweiligen Module vorgegeben werden.

```
1 class Instanz : Basis {
2   //Identifikation Basis --> Automatisierungsressourcepseudonym
3   //--> Automatisierungsressource
4   Dictionary<string, Dictionary<string, string>> Ressourcenkartierung;
5
6   //Ressource --> Modulidentifikation --> Liste Transitionen
7   Dictionary<string, Dictionary<string, List<Ausführungstransition>>>
8   Ausführungstransitionen;
9 }
```

Objekt 5.5: Klasse für die Instanziierung eines Automatisierungssystems

Ereignis

Ein Objekt der Klasse *Ereignis* beschreibt ein Modell zur Exposition einer strukturierten Menge von Dateneinträgen nach außen bei Auftreten von Auslösebedingungen. Das Objekt besitzt eine Menge an Auslösern (*Auslöser*) sowie eine serialisierte Vorgabe zur Objektstruktur (*Struktur*), in der Dateneinträge eingehängt sein können. Die nutzbare Basisklasse sieht Referenzen auf Dateneinträge im Auslöserfeld vor. Wenn einer der Dateneinträge eine Änderung erfährt, ist das Ereignis auszugeben.

```
1 class Ereignis {
2   string Identifikation;
3   string Name;
4   string Beschreibung;
5   List<string> Auslöser;
6   string Struktur;
7 }
```

Objekt 5.6: Klasse Ereignis, Implementierung

Funktion

Eine Funktion beschreibt einen erlaubten Schreibzugriff von außen auf eine Menge an Zieldateneinträgen in einem Datenmodell. Ein Funktionsobjekt ist nicht selbst-exponierend, sondern eine Vorgabe, nach dem eine Schnittstelle durch ein Softwaremodul betrieben wird. Die Funktion weist eine Liste an Referenzen aus (*Ziele*), die auf verfügbare Dateneinträge im Datenmodell zeigen. Bei Aufruf der instanziierten Funktionsschnittstelle wird ein

Schreibzugriff auf diese Dateneinträge durchgeführt. Die Klasse *Funktion* ist erweiterbar, beispielsweise um Zugriffsbeschränkungen zu beschreiben oder um Mindestvorgaben an die eingehenden Daten für die Dateneinträge zu machen. Es ergibt sich dann eine Bindung an entsprechend fähige Module.

```
1 class Funktion {
2   string Identifikation;
3   string Name;
4   string Beschreibung;
5   List<string> Ziele;
6 }
```

Objekt 5.7: Klasse Funktion, Implementierung

5.1.3 Beschreibung

Automatisierungsressource und Automatisierungssystem

Um Automatisierungsressourcen erfassen und abbilden zu können, wird durch den Handler eine Beschreibung jeder bei ihm eingehängten Automatisierungsressource erstellt. Die Klasse *Beschreibung* ist eine Basisklasse, die verwendbar ist oder erweitert werden kann.

```
1 class Automatisierungsressource : Header {
2   string Handler;
3   Dictionary<string, object> Devicebeschreibung;
4   List<string> Anwendungen;
5   Dictionary<string, Systemteil> Systemteile;
6 }
```

Objekt 5.8: Klasse zur Beschreibung einer Automatisierungsressource

Objekte zur detaillierteren Beschreibung der Systembeschaffenheit (EA-Einrichtungen, Softwarestände et cetera) oder zur Authentifizierung können mittels erbenden Klassen oder über das Feld *Devicebeschreibung* vorgesehen werden. Die Beschreibungsbildung erfolgt auf Basis statischer Daten, die von der Automatisierungsressource vorgegeben werden oder fest für die Automatisierungsressource beim Handler abgelegt sind, und dynamischer Daten, die der Handler zum Bildungszeitpunkt aggregiert oder erstellt. Ob redundante Felder verbunden oder eines über dem oder den anderen priorisiert wird, wird in der Handlerimplementierung festgelegt. Die Klasse beschreibt zudem den erstellenden Handler (*Handler*) und referenziert die ausgerollten Anwendungsmodelle (*Anwendungen*, nur Instanzen). Über Objekte der Klasse *Systemteil* werden die Anteile an einem Automatisierungssystem, die im Rahmen einer Anwendung auf der Automatisierungsressource konfiguriert wurden, gebunden an die Anwendungsreferenz wiedergegeben.

```
1 class Systemteil {
2   Dictionary<string, string> Module;
3   List<Dateneintrag> Dateneinträge;
4   List<Ereignis> Ereignisse;
5   List<Funktion> Funktionen;
6   List<string> Schnittstellen;
7 }
```

Objekt 5.9: Klasse zur Beschreibung des Systemanteils an einem Automatisierungssystem

Ein Systemteil gibt Modultypreferenzen gebunden an die anwendungsspezifischen Modulidentifikationen wieder sowie Dateneinträge, Ereignisse und Funktionen aus den aggregierten Modulbeschreibungen. Ein Systemteil referenziert erstellte Schnittstellen. Die zugehörigen Schnittstellenbeschreibungen sind von der Administrationsplattform abzurufen. Systemteile basieren auf dem Istzustand der anwendungsspezifischen Konfiguration.

```
1 class Automatisierungssystem : Header {
2   string Anwendung;
3   List<string> Automatisierungsressourcen;
4   Dictionary<string, string> Module;
5   Dictionary<string, string> Modulverteilung;
6   List<Dateneintrag> Dateneinträge;
7   List<Ereignis> Ereignisse;
8   List<Funktion> Funktionen;
9   List<string> Schnittstellen;
10 }
```

Objekt 5.10: Klasse zur Beschreibung eines Automatisierungssystems

Über ein Objekt der Klasse *Automatisierungssystem* kann ein erstelltes Automatisierungssystem beschrieben werden. Die Beschreibung wird auf der Administrationsseite, ausgehend von einer erstellten Anwendungsinstanz, aufgebaut und mit Inhalten aus erhaltenen Automatisierungsressourcenbeschreibungen erweitert. Einbezogen werden Systemteilinformationen aus den Automatisierungsressourcenbeschreibungen, die Referenzen auf die Anwendung besitzen. Sie werden unverändert durch Aggregation eingebunden. Einbezogene Automatisierungsressourcen sowie die Verteilung der Module im Automatisierungssystem werden ebenso weitergegeben. Abhängig von der anwendungsspezifischen Modulidentifikation zeigt die *Modulverteilung* die Identifikation der ausführenden Automatisierungsressource.

Abbildserie

Für die Erstellung von zeitgebundenen Abbildern der Dateneinträge zur Verhaltenserkennung nach dem Konzept wird die allgemeine Klasse *Abbildserie* eingeführt, über dessen

Format Abbildserien auf der Administrationsplattform abgelegt werden. Eine Abbildserie ist an Automatisierungsressource und Anwendung gebunden. Sie führt eine Liste an Zeitstempeln (*Zeitstempel*). Die Klasse ist aufgrund ihrer Struktur an eine zyklische Abbildung aller Dateneinträge gebunden. Für jeden Dateneintrag existiert eine Liste in *WerteNachDateneinträgen*. Zu Erfassungszeitpunkten wird die Zeitstempelliste um den Erfassungszeitpunkt und jede Wertliste in WerteNachDateneinträgen um den aktuellen Wert des jeweiligen Dateneintrags erweitert.

```
1 class Abbildserie {
2   string Automatisierungsressource;
3   string Anwendung;
4   List<Datetime> Zeitstempel;
5   //Dateneintragidentifikation --> Liste Werte gebunden an Zeitstempel
6   Dictionary<string, List<object>> WerteNachDateneinträgen;
7 }
```

Objekt 5.11: Klasse zum Speichern und Ablagen von Abbildserien

Modul

Die Klasse *Modul* ist eine verwendbare Basisklasse, die um modulspezifische Felder erweitert werden kann. Sie wird als Modulbeschreibung von einem Modul instanziiert und abgelegt oder bei einer Modulverteilung mitgeliefert und angepasst. Sie fließt in die Beschreibung einer Automatisierungsressource ein. Die Basisklasse beschreibt den Modultyp (*Typidentifikation*) und seine wesentlichen funktionalen Bestandteile (*Dateneinträge*, *Ereignisse*, *Funktionen*) und referenziert durch das Modul erstellte *Schnittstellen*.

```
1 class Modul : Header {
2   string Typidentifikation;
3   List<Dateneintrag> Dateneinträge;
4   List<Ereignis> Ereignisse;
5   List<Funktion> Funktionen;
6   List<string> Schnittstellen;
7 }
```

Objekt 5.12: Klasse zur Beschreibung eines Modulbeitrags zur Automatisierungsressource

Schnittstelle

Die Schnittstellenbeschreibung wird üblicherweise mit einer Automatisierungsressource mitgegeben oder von ihrem Handler erstellt. Ihr können statische oder dynamisch ermittelte Informationen zugrundeliegen. Sie ist über die Header-Identifikation (*Identifikation*) eindeutig identifizierbar. Die Klasse *Schnittstelle* ist als eine generell abstrakte Basisklasse

vorgesehen, von der für die Beschreibung der Automatisierungsressource Kindklassen verwendet werden, die für eine spezifische Schnittstelle geeignet sind. Als Beispiel ist eine Klasse *SchnittstelleUdp* implementiert, mit deren Objekten die Zugriffsadresse einer UDP-Schnittstelle abgebildet werden können.

Schnittstellenbeschreibungen können außerhalb der Automatisierungsressource und des Handlers erstellt und auf der Administrationsplattform abgelegt werden. Dieses Vorgehen ist beispielsweise notwendig, wenn Automatisierungsressource und Handler nicht in der Lage sind, korrekte Adressierungen zu erkennen (beispielsweise aufgrund Routings) oder die Adressen von einem externen System vergeben werden.

```
1  class Schnittstelle : Header {
2    string Automatisierungsressource;
3
4    Schnittstelle();
5  }
6
7  class SchnittstelleUdp : Schnittstelle {
8    string ip;
9    uint16 port;
10
11   SchnittstelleUdp();
12   SchnittstelleUdp(string ip, uint16 port);
13 }
```

Objekt 5.13: Klasse zur Schnittstellenbeschreibung und erweiternde modulspezifische Klasse

Handler

Handler beschreiben sich gegenüber der Administrationsplattform selbst mittels Instanzen der gleichnamigen Klasse und einer Angabe, welche Automatisierungsressourcen sie verwalten. Die Beschreibung ist ähnlich der Beschreibung von Automatisierungssystemen und Modulen eine Verständnisbasis und wird nicht für automatische Vorgänge genutzt. Sie kann entsprechend frei erweitert werden.

```
1 class Handler : Header {
2   List<string> Automatisierungsressourcen;
3 }
```

Objekt 5.14: Klasse zur Handlerbeschreibung

Quellen

Quellen sind Informationen zum Zugriff auf externe technische Systeme, die Software für Automatisierungsressourcen bereitstellen. Sie funktionieren dateibasiert. Quellen sind in der Anwendung zu erweitern, zum Beispiel zur Implementierung einer Softwareversionsverwaltung oder eines bestimmten Protokolls. Die Klasse *Quelle* ist eine abstrakte Klasse, die über protokollspezifische Klassen erweitert wird. Quellenbeschreibungen sind auf der Administrationsplattform abgelegt und werden von Handlern abgerufen, um Konfigurationselemente, die auf Quellen zugreifen, durchführen zu können.

```
1 abstract class Quelle : Header {
2   string typ;
3 }
4
5 class QuelleHttp : Quelle {
6   string URl;
7 }
```

Objekt 5.15: Abstrakte Klasse Quelle mit Beispiel zu spezifischer Implementierung

5.1.4 Daten

Dateneintrag und Datentypen

Typspezifische Dateneinträge (Beispiel *dtInt32*) erben von der Basisklasse *Dateneintrag* und überschreiben generische Lese- und Schreibfunktionen mit typabhängigen, um in der programmatischen Verwendung Typkonvertierungen durch den Programmierer zu vermeiden. Durch die Kapselung der Daten in eigenen typspezifischen Klassen können die innerhalb der programmatischen Struktur verwendeten Variablen direkt mit dem Datenspeicher verbunden werden und es ergibt sich eine Eindeutigkeit im Datenstand. Ausnahme ist der Objekt-Dateneintragtyp *dtObjekt*, der Kinddateneinträge referenziert und keinen eigenen les- oder schreibbaren Wert besitzt.

Die möglichen Datentypen werden an Standarddatentypen aus IEC 62541 (OPC UA) und üblichen Programmiersprachen orientiert. Wie im Konzept gesehen, wird ein Aufzählungstyp *Datentypen* genutzt, um die Deserialisierung für die Modellierung zu ermöglichen. Die verwendbaren Datentypen sind allgemein bekannte und etablierte Standardtypen. Sie sind ähnlich der Grunddatentypen der IEC 62541 (OPC UA), den Datentypen der IEC-61131-Sprachen und den Standardtypen der meisten regulären Mittel- und Hochpro-

grammiersprachen. Aufgrund der Verfügbarkeit in den verbreiteten Sprachen muss keine Implementierung der eigentlichen Datentypen vorgenommen werden.

```
1  class Dateneintrag : Header {
2    string Identifikation;
3
4    object wert;
5    Datentypen datentyp;
6    bool istListe;
7
8    void Start();
9
10   //Datenstrukturfunktionen
11   Mutex mutex;
12   void ImSpeicherAnlegen(string pfad);
13   void MitSpeicherVerknuepfen();
14   void MutexBlockieren();
15   void MutexFreigeben();
16
17   void WertLesen();
18   void WertSchreiben();
19 }
20
21 enum Datentypen {
22   Int8, Int16, Int32, Int64,
23   UInt8, UInt16, UInt32, UInt64,
24   Float, Double,
25   String, Objekt, Boolean, Datetime
26 }
27
28 class dtInt32 : Dateneintrag {
29   override void WertLesen(); //typspezifische Methoden
30   override void WertSchreiben();
31 }
32
33 class dtObjekt : Dateneintrag {
34   List<string> Elemente;
35 }
```

Objekt 5.16: Basisklasse in der Implementierung zu Dateneinträgen, Aufzählungstyp zu Datentypen sowie typspezifische Klassen für 32-Bit-Integer- und Objektdateneinträge

Die Identifikation wird mit an die IEC 62541 (OPC UA) angelehnten Identifikationsvorgehen über serialisierte *Node-Identifier* umgesetzt: Die Datenmodellidentifikation dient als Namespace, die Dateneintragidentifikation als String-Id im Node-Identifier.

```
1 servoDrehzahl.Identifikation = "ns=Servo;s=\"Drehzahl\"";
2 servoPosition.Identifikation = "ns=Servo;s=\"Position\"";
```

Objekt 5.17: Beispiele zu Dateneintragidentifikationen

Datenmodell

Aus der Klasse *Datenmodell* werden die Modelle zur Instruktion von Datenstrukturen gebaut. Sie werden programmatisch aus Modulen erstellt oder bei einem Modul mitverteilt sowie als Bestandteil von Anwendungsmodellen genutzt. Bei der Verteilung wird in der Umsetzung auf dateibasierte Serialisierungen aufgesetzt, sodass neben einem Standardkonstruktor ein dateiorientierter Konstruktor implementiert wird.

Ein instanziiertes Datenmodell besitzt Dateneinträge (*Dateneinträge*) und beschreibt Verknüpfungen zwischen Dateneinträgen (*VerknüpfungenImDatenmodell*). In den Verknüpfungen kann ein Dateneintrag auf mehrere Zieldateneinträge zeigen, an die Werteänderungen weitergegeben werden. Im Gebrauch kann die Datenmodellidentifikation als Namespace nach IEC 62541 (OPC UA) interpretiert werden. Sie wird in die Identifikationen der Dateneinträge eingebunden. Datenmodelle in Anwendungen können ohne Dateneinträge sein und ausschließlich Verknüpfungen enthalten, um die Integration von Dateneinträgen beispielsweise zur Verbindung von Modulen zu instruieren.

```
1 class Datenmodell : Header {
2   Dictionary<string, List<string>> VerknüpfungenImDatenmodell;
3   List<Dateneintrag> Dateneinträge;
4
5   Datenmodell();
6   Datenmodell(string path);
7 }
```

Objekt 5.18: Klasse Datenmodell in der Implementierung

Datenstruktur

Die Klasse *Datenstruktur* implementiert die programmatischen Funktionen für den Datenzugriff eines Moduls auf einer eingebetteten Automatisierungsressource. Sie fügt nach Aufruf in Dateien abgelegte Datenmodelle zu ihrer Struktur hinzu. Wenn sie gestartet wird, legt sie Dateneinträge in dem ihr zugewiesenen Zielpfad an, sollten diese noch nicht existieren. Sie stellt alle Funktionen zum Schreib- und Lesezugriff auf alle oder einzelne Dateneinträge zur Verfügung.

In der programmatischen Logik der Datenstruktur werden einfache Dateneinträge und Objekt-Dateneinträge angelegt (siehe 5.2). Im realen Datenspeicher existieren ausschließlich einfache Dateneinträge. Logische einfache Dateneinträge und ihre Speicherstrukturen sind hart verbunden. Objekt-Dateneinträge verweisen auf ihre Kindelemente und existieren nur im Modell sowie programmatisch in der Datenstruktur.

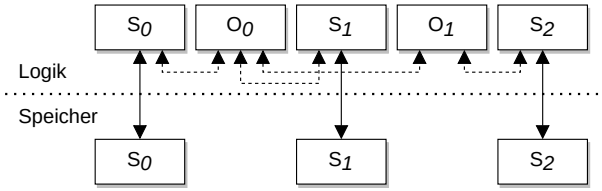


Abbildung 5.2: Verhältnis von Wert- und Objekteinträgen in Logik und Speicher

Wird in die Datenstruktur ein Datenmodell eingehängt, werden die Identifikationen der Dateneinträge um die Datenmodellidentifikation erweitert, sollten sie noch keine Datenmodellangabe besitzen.

```
1  using System;
2  using System.Linq;
3  class Datenstruktur {
4    List<string> datenmodelle;
5    Dictionary<string, Dateneintrag> dateneinträge;
6    string pfad;
7
8    Datenstruktur(string pfad);
9
10   void DatenmodellEinhängen(Datenmodell datenmodell);
11   void DatenmodellEinhängenAusDatei(string pfad);
12   void DatenmodelleEinhängenAusOrdner(string pfad);
13
14   void Start();
15
16   List<string> Lesen();
17   bool Lesen(string Identifikation);
18
19   void Schreiben();
20   void Schreiben(string Identifikation);
21 }
```

Objekt 5.19: Klasse zur Implementierung der Datenstruktur auf eingebetteten Automatisierungsressourcen

5.1.5 Konfiguration

Element und Paket

Konfigurationselemente sind der kleinste Teil einer Konfiguration. Konfigurationselemente beschreiben einzelne Vorgänge, die zu einer Konfiguration durchgeführt werden müssen. Eine Typmarkierung (*typ*) ist notwendig, um der Deserialisierung und dem Handler den korrekten Typ des Elementvorgangs (*vorgang*) zu zeigen. Für die Konzeptumsetzung sind drei Elementvorgänge vorgesehen: Parameter (Ersetzen von Variablen mit Werten in Zielen,

beispielsweise Dateizielen), Dienst (Einrichten einer Anwendung als Systemdienst) und Datei (Abruf einer identifizierbaren Datei aus einer Quelle und Ablage im Zielverzeichnis). Die Logik für die Durchführung ist seitens des Handlers zu berücksichtigen. Die möglichen Vorgangstypen sind nach Notwendigkeiten, Softwarebeschaffenheiten und den Architekturen der Automatisierungsressourcen zu erweitern. Konfigurationselemente benötigen keine Identifikation, da sie unmittelbar in Modelle eingebettet und nicht referenziert werden.

```
1  class Konfigurationselement {
2    string typ;
3    Vorgang vorgang;
4  }
5
6  abstract class Vorgang { }
7
8  class Datei : Vorgang {
9    string Quelle;
10   string Name;
11   string Ordner;
12 }
13
14 class Dienst : Vorgang {
15   string Name;
16   string Ziel;
17   string Operation;
18   string Arbeitspfad;
19 }
20
21 class Parameter : Vorgang {
22   string Ordner;
23   Dictionary<string, string> Variablen;
24 }
```

Objekt 5.20: Klasse Konfigurationselement mit implementierten Vorgängen der Umsetzung

Konfigurationspakete sind Listen durchzuführender Konfigurationselemente. Da Konfigurationspakete referenziert werden und zu beschreiben sind (beispielsweise um die standardisierte, versionsgebundene Verteilung und Einrichtung eines Moduls zu ermöglichen), werden sie mit der Header-Klasse erweitert.

```
1 class Konfigurationspaket : Header {
2   List<Konfigurationselement> Elemente;
3 }
```

Objekt 5.21: Klasse Konfigurationspaket

5.2 ISCI-Softwarestruktur

5.2.1 Automatisierungsressource

Es werden zwei übliche Profile für Automatisierungsressourcen definiert (siehe Abbildung 5.5 sowie Abschnitt A.1.5), mit denen die wesentlichen zu integrierenden technischen Systeme erfasst werden:

- **Eingebettete Automatisierungsressourcen mit offenem Betriebssystem:** die Automatisierungsressource ist eine technische Plattform mit einer Standardhardwarearchitektur und einem Betriebssystem, dessen Funktionen frei genutzt werden können. Der Handler ist auf der Automatisierungsressource fest vorgesehen und betreut nur diese. Alle Vorgänge und Einrichtungen erfolgen dateibasiert. Der Handler nutzt Betriebssystemfunktionen für die Einrichtungen (beispielsweise Dienste). Die Automatisierungsressource unterstützt alle durch ISCI definierten Funktionalitäten und wird zur Laufzeit regelmäßig und schnell rekonfiguriert. Eingebettete Automatisierungsressourcen tragen die Hauptlast spezifischer Implementierungen und Rekonfigurationen in einem Automatisierungssystem.

- **Gekoppelte Automatisierungsressourcen ohne offenes Betriebssystem:** die Automatisierungsressource ist eine technische Plattform mit einer beliebigen, üblicherweise nicht stark verbreiteten Architektur ohne Betriebssystem oder ohne in der Anwendung technisch oder effektiv nutzbaren Betriebssystemfunktionen. Der Handler ist ein abgeschlossenes, zentrales System, das extern der Automatisierungsressource betrieben wird. der Handler betreut mehrere Automatisierungsressourcen. Die Vorgänge und Einrichtungen basieren auf den durch die Automatisierungsressource unterstützten technischen Funktionen. Der Handler instanziiert notwendige weitere Brückensoftware auf der Automatisierungsressource oder extern, um die Automatisierungsressource zu integrieren. Die Automatisierungsressource unterstützt die wesentlichen vom ISCI vorgegebenen Funktionalitäten (Konfigurierbarkeit, Integration in ein Automatisierungssystem) ohne eindeutige Eingrenzung, ist aber nicht notwendigerweise schnell zu aktualisieren oder zu rekonfigurieren.

Eingebettete Automatisierungsressource mit Betriebssystem

Die eingebettete Automatisierungsressource ist ein eigenständiges System mit einem fest installierten Handler, das alle vorgegebenen Funktionalitäten unterstützt. Sie ist ein natives ISCI-System. Der Handler verwaltet die Automatisierungsressource in einer 1:1-Beziehung. Handler und Module sind Teil des lokalen Applikationsraums (siehe Abbildung 5.5, links).

- **Technische Plattform:** Die Automatisierungsressource baut auf einer verbreiteten und von Mittel- und Hochsprachen unterstützten Systemarchitektur auf (x64, x86, ARM etc.). Adressiert werden physische und virtuelle Systeme wie IPC, Soft-SPS oder entsprechende VM sowie Containerumgebungen. Sie verfügt über ein deterministisches, offen nutzbares Betriebssystem, ein beschreibbares persistentes Dateisystem für die Konfiguration der Automatisierungsressource und eine feste und ausreichend fähige Netzwerkintegration mit anderen Automatisierungsressourcen, Quellen und Administrationsplattform.

- **Zielebenen:** Die Automatisierungsressource verfügt über ein les- und schreibbares persistentes Dateisystem für die Anwendungseinrichtung und ein konfigurierbares Dienstesystem für die automatisierte Ausführung und Überwachung der Module. Beide Zielebenen werden vom Handler adressiert. Die Module sind durch den Handler persistent konfigurierbar. Alle Vorgänge finden nach Möglichkeit dateibasiert statt. Es werden Standardpfade für umzusetzende Anwendungen angelegt, die in jeweils eigenen Unterordnern dateibasiert angelegt werden (Abbildung 5.3). In den Konfigurationen werden für Pfade Standardisierungen in Variablenform angelegt, die ressourcenspezifisch durch den Handler gefüllt werden.

- **Technische Erweiterungen:** Die Automatisierungsressource erhält eine hinsichtlich Leistungskriterien ausreichend fähige und von Modulen unabhängig implementierte Datenstruktur, die auf der Basis von Standardfunktionen implementiert ist, um den Abstraktionsgrad möglichst groß zu halten und beliebige Erweiterungen zu unterstützen. Vorgesehen wird eine dateibasierte Abstraktion von Dateneinträgen im Arbeitsspeicher der technischen Plattform (Abbildung 5.4) mit einem eigenen Dateisystem oder mit Integration in das Standarddateisystem der Automatisierungsressource. Die Datenstrukturfunktionalität wird fest integriert.

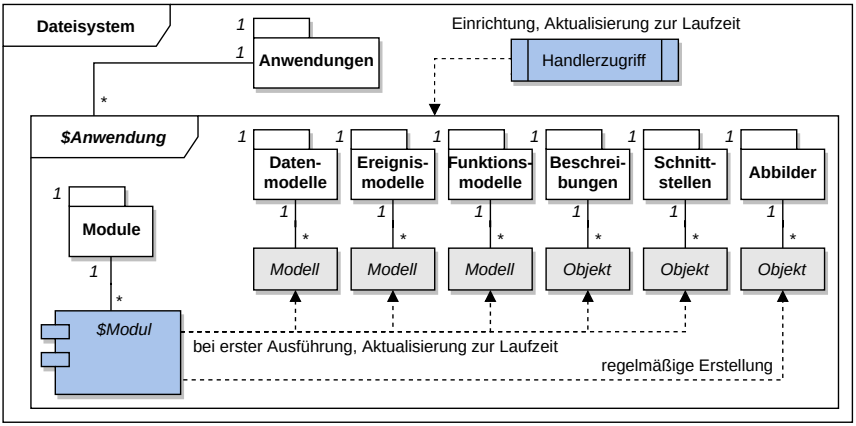


Abbildung 5.3: Ablagestruktur einer Anwendung auf einer Automatisierungsressource

Gekoppelte Automatisierungsressourcen ohne Betriebssystem

Die gekoppelte Automatisierungsressource ist ein System, das auf eine zusätzliche Integration bei Konfiguration oder Betrieb oder beiden Lebenszyklen angewiesen ist, um in einem mit dem ISCI gebauten Automatisierungssystem verwendet zu werden. Sie ist nicht vorrangig auf die Nutzung mit dem ISCI ausgelegt. Ein Handler für gekoppelte Automatisierungsressourcen liegt extern (Abbildung 5.5, rechts) und verwaltet eine oder mehrere Automatisierungsressourcen.

- **Technische Plattform:** Die Automatisierungsressource ist eine technische Plattform basierend auf eingeschränkt oder wenig verbreiteten Architekturen, die vorrangig in Niedrig-, Mittel- oder speziellen eigenen Sprachen (beispielsweise nach IEC 61131-3) programmiert werden. Adressiert werden Systeme wie geschlossene SPS, Mikrocontroller, eingebettete Systeme oder programmierbare Sensoren und Aktoren. Die technische Plattform ist automatisiert programmier- und konfigurierbar durch einen Handler. Sie verfügt über eine ausreichend fähige Netzwerkintegration mit anderen Automatisierungsressourcen und ihrem Handler.

- **Zielebenen:** Die Automatisierungsressource verfügt über eine schreibbare Programmablage für die Anwendungseinrichtung und eine konfigurierbare Ausführungseinrichtung für die automatisierte Ausführung von Modulen. Die Programmablage und

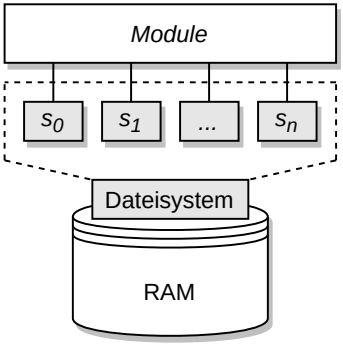


Abbildung 5.4: Abbildung der Dateneinträge in Arbeitsspeicher-Dateisystem

die Konfiguration sind durch den zuständigen Handler persistent modifizierbar. Der zuständige Handler baut, nach Notwendigkeit, außerhalb der Automatisierungsressource die anwendungsspezifische Software für die Automatisierungsressource, sollte keine Gesamtstruktur auf der Ressource schrittweise generierbar sein.

- **Technische Erweiterungen:** Nach Notwendigkeit wird die Funktionalität der Automatisierungsressource mit externer, durch den Handler einzurichtender und kontrollierender Software erweitert, beispielsweise um eine Kommunikationsadaption durchzuführen.

5.2.2 Modulimplementierungen

Das ISCI verwendet einen modularisierten Aufbau der entstehenden Softwarestruktur auf seinen Einsatzsystemen, um eine anwendungsspezifische Konfiguration und Rekonfiguration bei Wiederverwendung von Softwareartefakten zu erlauben. Diese Softwareartefakte müssen als Softwaremodule erstellt und bereitgestellt werden. Naheliegend ist deswegen eine Konzeptfestlegung für die Umsetzung von Modulskeletten in üblichen Programmiersprachen sowie die Erzeugung von Standardmodulen, die abhängig von der technischen Architektur von Automatisierungsressourcen einsetzbar sind. Die Module implementieren die Konzeptbeschreibungen nach Abschnitt 4.3.1. Durch das ISCI wird keine Menge zu implementierender Standardmodule explizit vorgegeben, da diese technologie- und zweckgebunden sind.

Ausführung als eigene Applikation

Aufgegriffen wird der Ablauf nach der Abbildung 4.14. Es wird ein Skelett als Umsetzungsgrundlage definiert, das als Ausgangsbasis für Standard- und Sondermodule dient:

```
1  static void Program() {
2    [...] //Initialisierung
3    var parameter = Parameter("konfiguration.json");
4    var struktur = Datenstruktur(parameter.OrdnerDatenmodelle,
5    parameter.OrdnerDatenstruktur);
6    struktur.Add(new Dateneintrag.dtInt32(0, "LokalerZustand"));
7    //modul.Beschreibung.ToFile(parameter.OrdnerBeschreibungen + "/" +
8    //parameter.Identifikation + "json");
9    [...] //weitere Initialisierung, Beschreibung
10
11   while(true) {
12     struktur.Lesen(parameter.IdentifikationNothalt);
13     if (struktur[parameter.IdentifikationNothalt]) {
14       //Interruptlogik
15     }//für jeden Interrupteintrag
16
17     struktur.Lesen("LokalerZustand");
18     var erfüllteTransitionen = parameter.Ausführungstransitionen.Where
19     (a => a.Eingangszustand == struktur["LokalerZustand"]);
20     if (erfüllteTransitionen.Count() > 0)
21     {
22       struktur.Lesen();
23       //Prozesslogik, Arbeit gegen Struktur
24       struktur["LokalerZustand"] = erfüllteTransitionen.First()
25       .Ausgangszustand;
26       struktur.Schreiben();
27     }
28   }
29 }
```

Objekt 5.22: Modul mit interner Ausführungsaktivierung und Interruptansatz

Das Modul als eigene Applikation initialisiert sich nach dateibasiert vorgegebenen Parametern und richtet eine Datenstruktur unter dem vorgegebenen Pfad ein, unter Nutzung gegebener Datenmodelle. Bei Änderung der Datenmodelle wird die programmatische Abbildung der Datenstruktur aktualisiert oder das Modul neugestartet. Es wird für die Ausführungssteuerung ein lokaler Dateneintrag (*LokalerZustand*) eingerichtet, dessen Wert durch das Modul überwacht wird, um die Prozesslogik gebunden auszuführen. Anschließend werden spezifische Einrichtungen sowie Beschreibungserstellungen vorgenommen. An den Stellen um die Initialisierung, Beschreibung, Interruptlogik und Prozesslogik werden die modulspezifischen Eigenschaften und Fähigkeiten implementiert.

Einbettung in Gesamtprogramm oder Laufzeitumgebung

Innere Daten und Abläufe eines Moduls bestehen prinzipiell aus *Initialisierung*, *Beschreibung*, *Interrupts* und *Prozesslogik*. Durch diese Aufteilung sind Module durch Aggregation ihrer Bestandteile in den vier Kategorien zusammenfassbar, um mehrere Module in einer Softwareentität auszuführen. Des Weiteren erlaubt die kategoriebezogene Aufteilung eine Adaption an und Einbettung in vorgegebene Softwarestrukturen, beispielsweise zur Integration in ein vorgegebenes programmatisch striktes Rahmenwerk für Mikrocontroller- oder IEC-61131-Umgebungen. Die strukturelle Aufteilung und die umgebende programmatische Struktur orientieren sich am Aufbau des Modulskeletts für Module mit einer Applikationsentität nach Quellcode 5.22.

5.2.3 Handler und Handlerintegration

Ein Handler wird im Regelfall als eigenständige, abgeschlossene Softwareanwendung in einer Mittel- oder Hochsprache implementiert und auf einem darunterliegenden Betriebssystem ausgeführt. Ein besonderes Betriebssystem ist nicht vorausgesetzt. Er abstrahiert die ressourcenspezifischen Technologien und Vorgänge gegenüber den technologieinvarianten Modellen. Er verfügt über Client-ISCI-Funktionalität für den Zugriff auf die Administrationsplattform und alle notwendigen technologischen Zugriffsimplementierungen für genutzte Quellen. Beim Handler werden Zugangsinformationen zur Administrationsplattform und den Zielebenen seiner Automatisierungsressource oder -ressourcen abgelegt. Ein Handler für gekoppelte Automatisierungsressourcen implementiert, falls notwendig und wo möglich, Brücken zwischen den spezifischen Strukturen der Automatisierungsressource und dem Automatisierungssystem.

Die Schritte der durch die Handler umgesetzten Integration sind in Abbildung 5.6 markiert. Handler erfassen zyklisch die Beschaffenheit der Ressourcenstruktur (1) hinsichtlich der anfallenden Modelle, Beschreibungen und des Systemaufbaus und aktualisieren die fusionierte Beschreibung der Automatisierungsressource auf der Administrationsplattform (2). Handler rufen die Beschreibungen von Automatisierungsressourcen, die sie selbst nicht verwalten, nach Verwendung in Anwendungen, denen ihre Automatisierungsressourcen zugewiesen sind, von der Administrationsplattform ab (3), und aktualisieren und erweitern

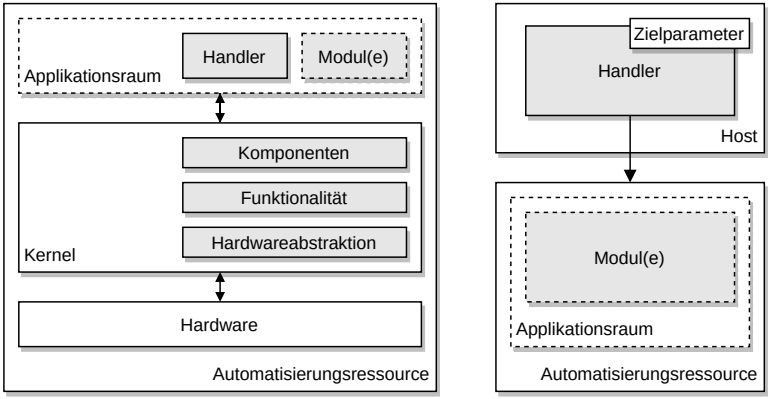


Abbildung 5.5: Handler als Bestandteil einer Automatisierungsressource mit Betriebssystem (links) und als externes System (rechts)

damit die Informationen, die ihre eigenen Automatisierungsressourcen für die Einbettung in einem Automatisierungssystem haben (4). Die Handler rufen notwendige Beschreibungen zyklisch ab und prüfen sie auf Änderungen, um Aktualisierungen berücksichtigen zu können, und rekonfigurieren damit Informationsteile bei ihren Automatisierungsressourcen, falls notwendig. Handler halten zum Abgleich Zwischenspeicher (Caches), um zur Prüfung keine Eingriffe in die laufende Automatisierungsressource vornehmen zu müssen. Die Zwischenspeicherungen erfolgen serialisiert. Die Handler realisieren damit die Informationsbereitstellung für die Automatisierungssystembildung durch die Module der Automatisierungsressourcen (5) und erzeugen die Gesamtmodelle für Daten, Funktionen und Ereignisse. Die in der Abbildung gezeigten Schritte unterliegen keiner festen Reihenfolge: Die Durchführungszyklen der beiden Seiten (Schritte 1 und 2 links; Schritte 3, 4 und 5 rechts) sind unabhängig voneinander. Schritt 2 erfolgt nur nach Änderungen oder der erstmaligen Ausführung von Schritt 1, ebenso wie Schritt 4 und Schritt 5 nur nach durch Schritt 3 erfassten Änderungen durchgeführt werden. Die Konfiguration der Automatisierungsressourcen erfolgt analog den Schritten 3, 4 und 5: Die Handler rufen mit den Identifikationen der Automatisierungsressourcen von der Administrationsplattform ab, in welche Anwendungsinstanzen die Automatisierungsressourcen eingebunden sind und setzen neue oder geänderte Anwendungsinstanzen auf den Automatisierungsressourcen um.

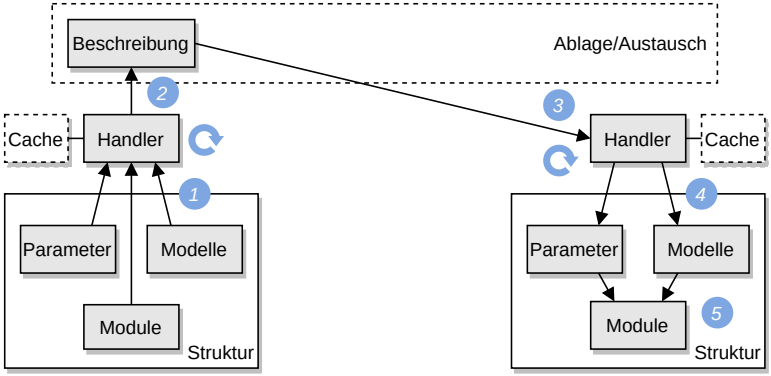


Abbildung 5.6: Ablauf der Modell- und Objektabfragen und des -austausches für die Integration der eingerichteten Softwarestrukturen der Automatisierungsressourcen

Im Zwischenspeicher werden über die umgesetzten Anwendungsinstanzen dokumentiert. Die Prüfung auf Änderungen erfolgt durch Vergleich der Modelle.

Die Handler unterstützen Standardvariablen in den Anwendungsmodellen und Konfigurationsteilen für Pfade, Adressen, Systemnamen und ähnliche, die sie spezifisch für eine Automatisierungsressource ausrollen. Der Handler übernimmt ressourcenspezifische Anpassungen und integriert notwendige Modellinformationen (Datenmodelle, Funktionen und Ereignisse, Datenverknüpfungen, Schnittstellenparameter) automatisiert in Programme, wofür er Module vor der Installation auf der Automatisierungsressource anpasst. Er kann zu gekoppelten Automatisierungsressourcen eigenen Standardquellcode beitragen, beispielsweise um Variablenlisten aufzubauen oder Integrationen über bestimmte Kommunikationsprotokolle im jeweilig notwendigen Format zu definieren und einzubauen.

5.2.4 Administrationsplattform und Quellen

Auf einem von allen Handlern über das Produktionsnetzwerk erreichbaren Server werden die API und die Datenhaltung der Administrationsplattform betrieben. Die API wird nach Bedarf um Authentifizierung von anfragenden Handlern, beispielsweise über Zertifikate, erweitert. Die Administrationsplattform wird vorzugsweise einmal je Produktionsumgebung betrieben.

Bei Administrationsplattform und Quellen ist eine zentrale, dateiorientierte Implementierung auf einem Server mit Verbindung zu den Handlern angebracht. Der Zugriff ist zu jedem Zeitpunkt möglich. Eine HTTP/REST-Implementierung erlaubt ein Kartierung der Zugriffspfade auf eine dateibasierte Ablage. Die HTTP/REST-Verbindung ist durch die Verwendung von TLS absicherbar. Authentifikationsmethoden sind freigestellt. Die Administrationsplattform ist ein sicherheitskritischer Teil und vorzugsweise nur eingeschränkt erreichbar, beispielsweise reduziert auf das verwendete Produktionsnetz.

Die Plattform generiert über die abgelegten Anwendungsmodelle ein Register für jede angegebene Automatisierungsressource, welche Anwendungsinstanzen für sie auszurollen sind. Die Abfrage erfolgt über die Identifikation der Automatisierungsressource. Alle Modelle und Objekte werden, nach ihren Typen sortiert, dateibasiert und serialisiert abgelegt. Für jeden Modelltyp ist ein Register als Liste abfragbar. Der Zugriff auf Modelle und Objekte erfolgt über eine Kombination aus Typ und Identifikation. Über die API sind nach den Methoden in Tabelle 5.1 zugreifbar:

- Anwendungen/
  - Modell: Anwendungsmodelle (Basis und Instanz)
  - Instanziieren: Umzusetzende Instanzen nach Automatisierungsressource
- Beschreibungen/
  - Ressourcen: Beschreibung einer Automatisierungsressource
  - Schnittstellen: Beschreibung einer Schnittstelle
  - Handler: Beschreibung eines Handlers
  - Quellen: Beschreibung einer Quelle
- Konfiguration/
  - Pakete: Beschreibung eines Konfigurationspakets
- Abbilder: Datenstrukturabbilder

Die anfallenden Quellen zur Softwarebereitstellung für die Automatisierungsressourcen können ebenfalls als HTTP/REST-Server oder mit anderen Protokollen ausgeführt werden.

Da Quellen in ihren Funktionen ausschließlich lesende Funktionen bereitstellen müssen, können Quellen frei erreichbar sein, beispielsweise über das Internet. Als HTTP/REST-Server implementiert eine Quelle über eine dateiorientierte GET-Methode Zugriff auf Dateien.

Methode	**Pfad**	**Beispiel**	**Erklärung**
GET	/$Typ	Beschreibungen/Quellen	Abfrage Liste aller Modelle/Objekte eines Typs
GET	/$Typ/$Id	Beschreibungen/Quellen/00-80-77-90-0a-8c	Abfrage eines Modells oder Objekts
PUT	/$Typ/$Id	Beschreibungen/Quellen/00-80-77-90-0a-8c	Modell/Objekt anlegen oder aktualisieren
DELETE	/$Typ/$Id	Beschreibungen/Quellen/00-80-77-90-0a-8c	Modell/Objekt löschen

Tabelle 5.1: HTTP/REST-API der Administrationsplattform zum Modell- und Objekthandling

5.3 Implementierung

Die Softwarestruktur wurde für die Validierung prototypisch implementiert. Als Programmiersprache wurde für alle Implementierungsarbeiten C# mit dem Softwarestandard .NET Core 3.1 gewählt. C# ist eine rein objektorientierte Programmiersprache, die im gewählten Standard plattformübergreifend ausgeführt werden kann. Die für die Just-in-time-Kompilierung von C# notwendige Laufzeitumgebung lässt sich beim gewählten Standard in die gebauten Anwendungen integrieren, wodurch Softwareteile auf wenige Dateien konzentriert und Versions- oder Ausführungsfehler durch falsche oder fehlende Bibliotheken vermieden werden können. Der Programmierstandard erlaubt den Bau von Programmen für verschiedene Architekturen (x86, x64, ARM unter Windows und Linux) ohne programmatische Änderungen. Durch die Wahl einer objektorientierten Sprache können alle ISCI-Datenstrukturen ohne Erstellung eigener Objektorientierungsfunktionen implementiert werden. Als Entwicklungsbasis wird der Aufbau der funktionalen Validierung genutzt (siehe Abschnitt 6.1.1). Die Validierungsimplementierung berücksichtigt eingebettete Automatisierungsressourcen und nutzt HTTP/REST-Server für die Funktionen der Administrationsplattform und einer HTTP/REST-basierten Quelle.

6 Das Integrations- und Steuerungsframework *ISCI*: Anwendung und Evaluation

Es werden zwei Anwendungsfälle definiert, die insgesamt zur Entwicklungsbegleitung und Validierung dienen. Im Sinne der in Abschnitt 1.6 und Abbildung 1.3 vorgestellten Methodik dienen sie weiter zur Realisierung der Konstruktions-Evaluations-Schleife. Mit den Anwendungsfällen und ihren Validierungsanforderungen werden die grundsätzlichen Eigenschaften und Funktionen des ISCI geprüft. Schwerpunkt der *funktionalen Validierung* und des für sie genutzten Versuchsaufbaus sind die Grundlage für die technische Entwicklung und der funktionalen Logik. Die *Szenariovalidierung* dient zur Validierung der Modelle und Strukturlogik.

6.1 Funktionale Validierung

Für die funktionale Validierung wird die Struktur des einfachstmöglichen CPS aufgebaut: zwei rechenfähige technische Plattformen mit stehender Vernetzung, die gemeinsam einen physischen Prozess steuern. Zwei Automatisierungsressourcen sind über ihre Netzwerkschnittstellen verbunden und mit Handlern versehen. Eine der beiden Automatisierungsressourcen verfügt über eine EA-Schnittstelle. Die Systemstruktur ist in Abbildung 6.1 zu sehen. Das Automatisierungssystem simuliert eine kooperative Prozesssteuerung, in die beide technische Plattformen eingebunden sind, über eine Separierung der anfallenden Funktionalitäten: Eine technische Plattform implementiert die EA-Funktionalität und die Prozessintegration, die zweite implementiert die Verarbeitungsfunktion und damit die für den Prozess notwendige Steuerungslogik. Ein PC stellt die Funktionen für die Administration des Systems über HTTP-Server zum Datenabruf und zur Datenablage bereit, sowie zur Auswertung der im Betrieb entstehenden Zustandsdaten des Automatisierungssystems.

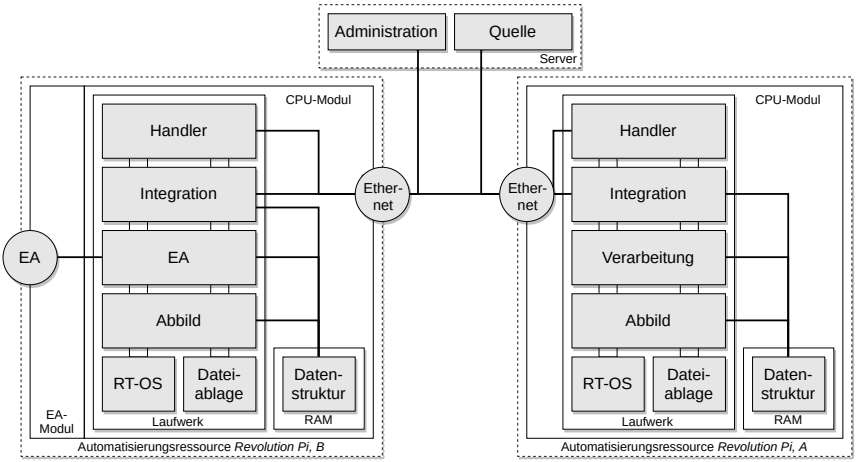


Abbildung 6.1: Strukturen des CPS-Automatisierungssystems für die funktionale Validierung

6.1.1 Implementierung

Die Implementierungsbasis für das CPS der funktionalen Validierung wird mit zwei Embedded-Systemen des Typs Revolution Pi des Herstellers Kunbus aufgebaut. Es handelt sich um Raspberry-Pi-Versionen auf ARMv7-Basis mit Echtzeitbetriebssystemen (Raspian-Linux mit RT-Patch, siehe auch Anhang A.2.3). Die beiden Systeme sind über ihre Ethernetschnittstellen miteinander und dem Server für Administration und Quellen verbunden. Eines der beiden Systeme verfügt über ein EA-Modul für digitale Ein- und Ausgänge. Auf den Embedded-Systemen wurden C#-Implementierungen des Handlers eingebettet, die ihre jeweilige Automatisierungsressource in einer 1:1-Beziehung gegenüber der Administration vertreten. Die Handler sind funktional gleich fähig. Die Vernetzung erfolgt über ein Standardprotokoll (UDP) über Modulimplementierungen.

Es wurde ein Entwicklungs- und Validierungsanwendungsmodell definiert, das beide Automatisierungsressourcen in ein Automatisierungssystem integriert und die Grundmodule für eine funktionale Validierung ausrollt (siehe Anhang D.9). In der Validierungsimplementierung werden alle Module als geschlossene, eigene Anwendungen dauernd ausgeführt, wobei die Ausführungsaktivierung programmatisch in die Module platziert wird. Der jeweilige Handler verteilt dazu die Einträge aus dem Instanzanwendungsmodell auf die

```
  GNU nano 2.9.3                              fstab

# /etc/fstab: static file system information.
#
# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point>   <type>  <options>       <dump>  <pass>
# / was on /dev/nvme0n1p3 during installation
UUID=a9e69d1b-d488-4a4d-8f05-af135ed7db10 /               ext4    errors=remount-ro 0       1
# /boot/efi was on /dev/nvme0n1p1 during installation
UUID=FE9C-6EA4  /boot/efi       vfat    umask=0077      0       1
/swapfile                                 none            swap    sw              0       0
tmpfs       /media/ramdisk      tmpfs   rw,nodev,nosuid,size=100M       0  0
^G Hilfe        ^O Speichern    ^W Wo ist       ^K Ausschneiden ^J Ausrichten   ^C Textmarke
^X Beenden      ^R Datei öffnen ^\ Ersetzen     ^U Ausschn. r   ^T Rechtschr.   ^_ Zu Zeile gehe
```

Abbildung 6.2: Einrichtung eines Arbeitsspeicherdateisystems */media/ramdisk* für die Datenstrukturen einer Automatisierungsressource über */etc/fstab*

Konfigurationen der Module anhand der Modulidentifikationen. Der Scheduler des Betriebssystems wird nicht gesondert angepasst.

Es wird über Betriebssystemfunktionen auf den Automatisierungsressourcen ein nichtpersistentes Standarddateisystem im Arbeitsspeicher erstellt, in dem die Datenstrukturen der Anwendungen dateibasiert angelegt werden. Das Betriebssystem wird entsprechend konfiguriert, wie ihn Abbildung 6.2 gezeigt. Jede Variable erhält als Dateneintrag eine eigene Datei. Die Erstellung der Dateneinträge als Dateien auf Unix-Systemen ist in der Abbildung 6.3 zu sehen. Erkennbar ist in der Abbildung an den Dateinamen die Identifikation der vorgebenden Datenmodelle über den Namespace-Teil des Dateinamens. Der Zugriff auf die Datenstrukturdateien wird über System-Mutex gesteuert. Über den dateibasierten Zugriff können alle Anwendungen, die einen Dateizugriff über das Betriebssystem auslösen können, auf die Informationen des Automatisierungssystems zugreifen. Über diese Datenstruktur wird die Entwicklungs- und Validierungsanwendung implementiert.

Um die Auswertungsansätze für die Zeitverhaltensanalyse des Automatisierungssystems sowie der Automatisierungsressourcen untersuchbar zu machen, werden die auf der Administrationsplattform abgelegten Datenabbilder in eine zeitreihenbasierte Datenbank auf der Administrationsplattform überführt. Auf der genutzten Datenbanksoftware InfluxDB wurden graphische Darstellungen definiert, die gemäß des Arbeitsansatzes die System-

Abbildung 6.3: Textbasiert abgelegte Datenmodelle einer Beispielanwendung (oben) und ihre in der Initialisierung im Arbeitsspeicherdateisystem angelegten Dateneinträge (unten)

Abbildung 6.4: Beispieldarstellung und -auswertung aufgenommener lokaler Zykluszeiten der zwei Automatisierungsressourcen des Aufbaus zur funktionalen Validierung

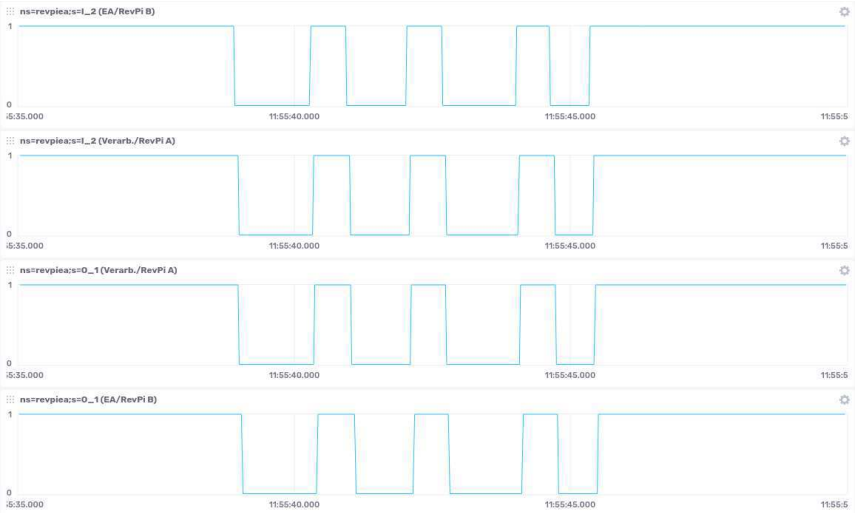


Abbildung 6.5: Lokale Verläufe der Zustände zweier Dateneinträge auf den zwei Automatisierungsressourcen in einem Beispielzeitbereich

zustände sowie Ausführungszeiten abbilden und generierte Kennwerte zu den lokalen Zyklen der Automatisierungsressourcen wiedergeben (Abbildung 6.4). Abbildung 6.5 zeigt Verläufe der Zustände zweier Dateneinträge auf den Automatisierungsressourcen, die einen physischen Ein- und einen physischen Ausgang repräsentieren. Die Verläufe sind in der Abbildung als eine Kette von oben nach unten angeordnet: Verlauf des Eingangs auf der EA-Ressource, Verlauf des Eingangs auf der verarbeitenden Ressource, Verlauf des Ausgang auf der verarbeitenden Ressource, Verlauf des Ausgangs auf der EA-Ressource. Der Zustand des Ausgangs folgt dem Zustand des Eingangs. Die dazu notwendige Logik wird über das Verarbeitungsmodul auf der verarbeitenden Ressource ausgeführt. Über die gemeinsame Visualisierung oder die Auswertung von Änderungszeitpunkten der Dateneinträge in den Datenstrukturen ist das Reaktionsverhalten auf Eingangsänderungen nachvollziehbar (siehe Abbildung 6.6, Eingangszustand und Ausgangszustand (nacheilend) auf EA-Automatisierungsressource). Mit dem ISCI erstellte Automatisierungssysteme sind somit einfach im Verhalten und an einem einzelnen Punkt analysierbar.

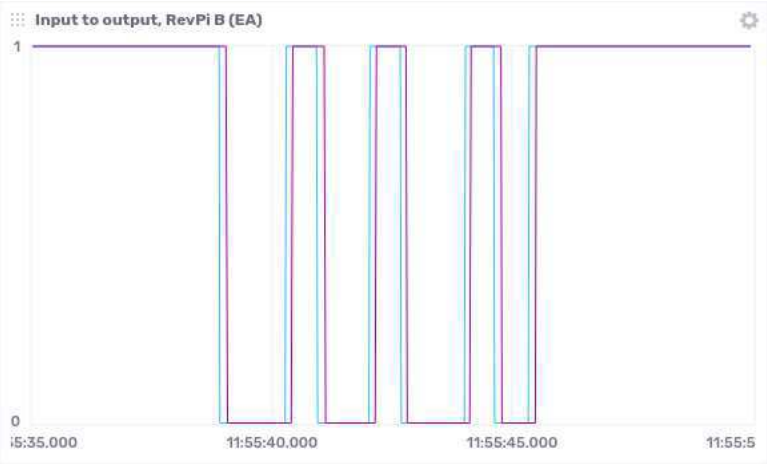


Abbildung 6.6: Zustandsverläufe in einem Beispielzeitbereich von Eingangs- (blau) und Ausgangsdateneintrags (lila) auf der EA-Automatisierungsressource

6.1.2 Modulstruktur

Im Automatisierungssystem ist eine EVA-Kette aufgebaut. Physische Ein- und Ausgänge sind über eine Automatisierungsressource eingebunden, die Verarbeitung findet auf einer anderen Automatisierungsressource statt. Alle Module basieren auf demselben Quellcodeskelett und sind in C# implementiert. Alle Modelle werden dateibasiert lokal abgelegt und durch die Module nach Notwendigkeit instanziiert. Die im Arbeitsspeicherdateisystem entstehenden Datenstrukturen werden über Abbildmodule auf beiden Automatisierungsressourcen für die Analysefunktionen hinsichtlich Zykluszeiten und Wertverläufen erfasst.

Alle entwickelten Module sind gegen die technischen Plattformen des Automatisierungssystem zur funktionalen Validierung entwickelt. Implementierungen zu den Modulen sind im Anhang D aufgeführt. Im Automatisierungssystem verwendete Module auf ISCI-Basis sind (siehe Abbildung 6.1):

- **Integration:** Modul zur Netzwerkintegration der Datenstrukturen der Automatisierungsressourcen über UDP
- **EA:** Hardwarespezifisches Modul zur Einbindung von digitalen Ein- und Ausgängen

- **Verarbeitung:** Validierungsspezifisches Modul zur Integration von Ein- und Ausgängen für die Funktionsuntersuchung
- **Abbild:** Modul zur Abbildung der Datenstrukturen im Betrieb

Mit dem Aufbau wurde ein Standardskelett unter C# entwickelt, das für andere Sprachen, insbesondere objektorientierte, adaptierbar ist. Es baut auf einem EVA-Ansatz mit einer dauernden inneren Arbeitsschleife auf. Das für das ISCI implementierte Skelett ist für die Nutzung auf einer technischen Plattform mit Betriebssystem und selbstkontrollierte Ausführungssteuerung über einen Zustandsdateneintrag ausgelegt. Das Modulstandardskelett in C# wird als implementierungsnaher Prinzipcode im Folgenden über Kommentare erläutert. Die objektorientierte Hochsprache und die durch das ISCI standardisierten Strukturen im Hintergrund erlauben einen großen Abstraktionsgrad gegenüber dem Modulprogrammierer. Das Standardskelett ist in zwei Abschnitte unterteilbar, die aneinandergefügt werden. Der erste Teil ist zur Parametrierung und Einrichtung:

```
1  /* Modulparameter aus Datei einlesen; alle modulspezifischen
   ↪  Einstellungen sowie Standardordner zu Modellen und
   ↪  Datenstrukturbereichen */
2  var konfiguration = new isci.Allgemein.Parameter("konfiguration.json");
3
4  var dm = new isci.Daten.Datenmodell(konfiguration.Identifikation);
5  /* [...] Bereich für die Definition des moduleigenen Datenmodells dm,
   ↪  anschließend Abspeichern im Ablageordner für Datenmodelle unter der
   ↪  Modulidentifikation */
6  System.IO.File.WriteAllText(konfiguration.OrdnerDatenmodelle + "/" +
   ↪  konfiguration.Identifikation + ".json",
   ↪  Newtonsoft.Json.JsonConvert.SerializeObject(dm));
7
8  /* Instanziierung eines modulinternen Datenstrukturobjekts. */
9  var structure = new
   ↪  isci.Daten.Datenstruktur(konfiguration.OrdnerDatenstruktur);
10
11 /* Eingabe der zu verwendenden Datenmodelle über den standardisierten
   ↪  Datenmodellordner der Anwendung in das Datenstrukturobjekt und
   ↪  anschließende Aktivierung. Bei festgestellten Änderungen im
   ↪  Datenmodellordner wird die Datenstruktur aktualisiert. */
12 structure.DatenmodelleEinhängenAusOrdner(
13 konfiguration.OrdnerDatenmodelle);
14 structure.Start();
15
16 /* Modulbeschreibung erstellen als Beitrag für die
   ↪  Ressourcenbeschreibung und anschließendes Ablegen im
   ↪  standardisierten Ordner. Name und Beschreibungstext sind als
   ↪  einzutragende Strings zu verstehen. */
17 var beschreibung = new
   ↪  isci.Beschreibung.Modul(konfiguration.Identifikation, {Modultyp},
```

```
18 dm.Datafields);
19 beschreibung.Name = {Name};
20 beschreibung.Beschreibung = {Beschreibung};
21 beschreibung.Speichern(konfiguration.OrdnerBeschreibungen + "/"
22 + konfiguration.Identifikation + ".json");
23
24 /* Verknüpfung mit Zustandsdateneintrag für Modulausführung. */
25 var Zustand = new isci.Daten.dtInt32(0, "Zustand",
   ↪ konfiguration.OrdnerDatenstruktur + "/Zustand");
26 Zustand.Start();
```

Objekt 6.1: Standardskelett für Modul mit interner Ausführungsaktivierung, Parametrierungs- und Einrichtungsteil

Es werden im Parametrierungs- und Einrichtungsteil zudem alle programmatischen Strukturen für den Modulbetrieb aufgebaut. An verschiedenen Stellen ist die Dateibasierung erkennbar. Durchweg menschlesbare Dateien werden genutzt, um Informationen zur Modulkonfiguration sowie zum Datenumgang und zur Beschreibung zu erhalten. Zudem erfolgt über die Dateibasierung der Rückgriff auf die Datenstruktur. Der zweite Skelettabschnitt ist der Betriebsteil mit der Arbeitsschleife des Moduls:

```
1  while(true) {
2    /* Zustandsdateneintrag lesen */
3    Zustand.WertLesen();
4
5    /* Prüfung, ob Zustandsdateneintrag Ausführungszustand erreicht hat,
     ↪ falls ja, Ausführung der Arbeitslogik des Moduls für den Zustand
     ↪ */
6    var erfüllteTransitionen = konfiguration.Aktivzustände.Where(a =>
     ↪ a.Eingangszustand == (System.Int32)Zustand.value);
7    if (erfüllteTransitionen.Count<Aktivzustand>() > 0) {
8      if (erfüllteTransitionen.ElementAt(0) ==
       ↪ konfiguration.Aktivzustände[0]) {
9        /* Arbeitsbereich, Modul ist ausgelegt für einen
         ↪ Ausführungszustand */
10     }
11
12     /* Lokalen Anwendungsteil in nächsten Zustand führen */
13     Zustand.value =
       ↪ erfüllteTransitionen.First<Aktivzustand>().Ausgangszustand;
14     Zustand.WertSchreiben();
15   }
16 }
```

Objekt 6.2: Standardskelett für Modul mit interner Ausführungsaktivierung, Arbeitsschleife

Ein Modul kann bei mehreren lokalen Zuständen aktiviert werden, die über ihre Reihenfolge innerhalb der Arbeitsschleife relativ unterscheidbar sind. In der Beispielarbeitsschleife ist mit einem aktivierenden Zustand der Regelfall implementiert. Das Modul überführt in der Implementierung den lokalen Zustandsdateneintrag in den nächsten Ausgangszustand.

6.2 Szenariovalidierung

Um die Modellierungsansätze für die Definition eines verteilten Systems hinreichend zu prüfen, wird neben der funktionsorientierten funktionalen Validierung eine Szenariovalidierung definiert. Es wird ein Anwendungsfall festgelegt, für den Modelldefinitionen für die Umsetzung mit dem ISCI getroffen werden. Das Gesamtmodell und die Modellbestandteile werden diskutiert. Als Anwendungsfall wird ein übliches Fertigungssystem der Produktion mit parallel und kooperativ arbeitenden physischen Komponenten gewählt. Notwendige Module für die Szenariovalidierung werden implementiert und sind im Anhang D aufgeführt. Die Funktionserfüllung der Module steht für die Szenariovalidierung nicht im Vordergrund.

6.2.1 Szenario

Für das Szenario wird eine Fräsmaschine als Fertigungssystem gewählt. Die Zielmaschine wird für die Analyse zur Anwendungsdefinition in vier physische prozesserfüllende Komponenten aufgeteilt: drei Linearachsen und eine Frässpindel. Um den physischen Zielprozess zu erfüllen, müssen die vier Komponenten zeitgleich und abgestimmt arbeiten. Der physische Prozess unterliegt kritischen Zeitanforderungen. Die Anweisungen zur Prozessdurchführung werden in den Fertigungsprogrammen für das Fertigungssystem schrittweise beschrieben. Bei der Umsetzung der Automatisierungslogik mit dem ISCI wird die Fertigungssystemstruktur als verteiltes System interpretiert. Das Fertigungssystem des Szenarios wird in Produktionsumgebungen mit nicht-zeitkritischen Systemen aus leitenden und kaufmännischen Umgebungen verbunden, wozu es Schnittstellen benötigt, um Aufgaben entgegenzunehmen und Zustände wiederzugeben.

6.2.2 Herkömmliche Automatisierungsstruktur

Ein heutiges automatisiertes und numerisch gesteuertes Fertigungssystem wie eine CNC-Fräsmaschine verfügt wie in der Abbildung 6.8 gezeigt über drei aktive digitale Systeme: die Mensch-Maschine-Schnittstelle (HMI), das Steuerungssystem für die Bewegungssteuerung (*Numerical Control Kernel*, NCK) und eine SPS für die Steuerung aller nicht unmittelbar bewegungsrelevanten Teile. Alle Systeme verfügen über eigene Hardware und sind hart

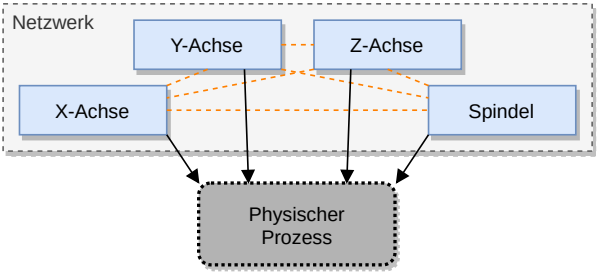


Abbildung 6.7: CPS-Strukturprinzip angewandt auf das Validierungsszenario

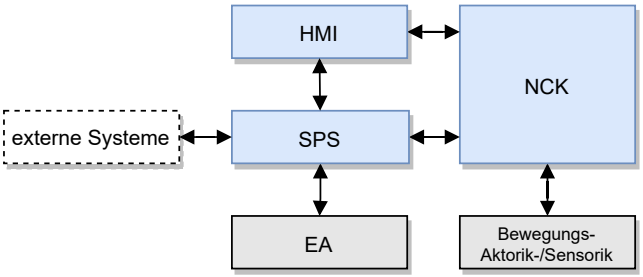


Abbildung 6.8: Automatisierungselemente eines heutigen CNC-Fertigungssystems (Lesi et al. 2023)

miteinander verdrahtet. Aus SPS und NCK werden die physischen Bestandteile der Maschine erfasst und angesteuert. (Lesi et al. 2023) Der NCK betreibt den physischen Fertigungsprozess über die bewegungsbezogene Sensorik und Aktorik entsprechend eines zugewiesenen Programms. Die SPS bedient die übrigen EA der Peripherie und Kommunikationsaufgaben.

6.2.3 Vorstellung der Automatisierungsstruktur mit ISCI

Im Folgenden werden die Bestandteile der Anwendungsmodelle für die Szenariovalidierung vorgestellt. Die Modelle des Automatisierungssystems sind in ihren Gesamtformen im Anhang D.10 angefügt. Erläutert werden die Konfiguration des Automatisierungssystems, die anwendungsspezifischen Datenmodelle, die Funktions-Ereignis-Schnittstelle des Automatisierungssystems, die Ausführungsmodellierung und die Definitionskartierung auf Automatisierungsressourcen. An der Erläuterung der Bestandteile wird das Definitionsvorgehen zur Automatisierungssystemerstellung über den Anwendungsmodellansatz deutlich.

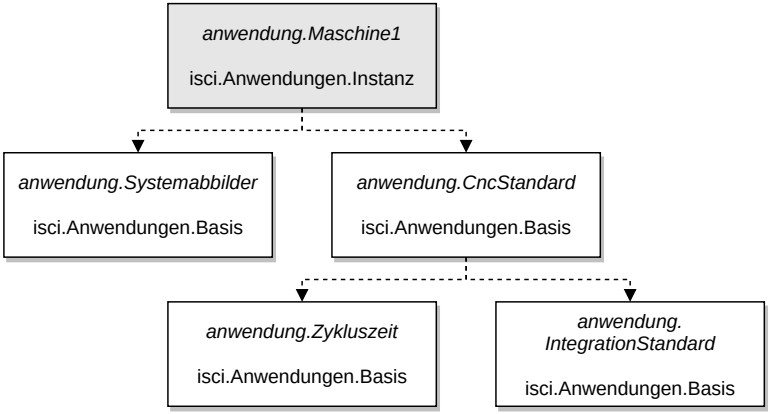


Abbildung 6.9: Baum der umgesetzten Anwendungsmodelle

Nach heutigem Vorgehen in der Softwareentwicklung der Automatisierung werden mehrere Entwicklungsprojekte, meist eines je Gerät oder Gerätehersteller, für einen Maschinen- oder Anlagentyp unterhalten (Eckert 2015). Mit der ISCI-Struktur werden die gerätegebundenen Teile HMI, SPS und NCK der heutigen Automatisierungsstruktur als Softwaremodule in Anwendungsmodelle eingebettet.

Wie durch den Ansatz des ISCI vorgesehen, wird der Aufbau des Automatisierungssystems durch ein Instanzanwendungsmodell *anwendung.Maschine1* endgültig beschrieben. Die Instanz baut über Referenzen auf vier Basisanwendungsmodellen *anwendung.Systemabbilder*, *anwendung.CncStandard*, *anwendung.Zykluszeit* und *anwendung.integrationStandard* auf (siehe Modellbaum in der Abbildung 6.9). Zu sehen ist, dass sich mit der ISCI generische Basisanwendungen definieren lassen, die unabhängig vom Anwendungsfall des Automatisierungssystems verwendbar sind, um Standardfunktionalitäten sowie -technologien im Automatisierungssystem zu verteilen (*anwendung.Systemabbilder*, *anwendung.Zykluszeit*, *anwendung.integrationStandard*). Ebenso ist zu sehen, dass sich anwendungsfallorientierte oder -gebundene Basisanwendungsmodelle definieren lassen (*anwendung.CncStandard*). Mit solchen Basisanwendungsmodellen, die dann den größten Teil der technischen Definitionen für das Automatisierungssystem leisten, sind generalisierbare Teile der Systemdefinition in ein Standardmodell integrierbar, das über ein Instanzanwendungsmodell auf technische Plattformen final adaptiert wird.

Beachtenswert sind Anwendungsbasismodelle, wie unten angeführt zur Verteilung der Module zur Zykluszeitermittlung, die generische Software im Automatisierungssystem verteilen. Allgemeine und generische Systemkonfigurationen, beispielsweise als ein Entwicklerstandard, sind so im verteilten System als Anwendung zu gestalten und über eine Anwendungsreferenz in der spezifischen Anwendung einzuhängen.

```
1  {
2    "ReferenzierteModelle": [],
3    "Konfigurationspakete": {"*": {"zykluszeit": "isci.zykluszeit"}},
4    "Konfigurationselemente": {},
5    "Datenmodelle": [],
6    "Funktionen": [],
7    "Ereignisse": [],
8    "Identifikation": "anwendung.Zykluszeit",
9    "Name": "Zykluszeit",
10   "Beschreibung": "Anwendung zur Zykluszeitermittlung für alle
11   Einzelsysteme",
12   "Typ": "isci.Anwendungen.Basis"
13 }
```

Objekt 6.3: Einfache generalisierte Basisanwendung zur Verteilung von Modulen zur Zykluszeitermittlung auf allen (* in *Konfigurationspakete*) Automatisierungsressourcen

Konfiguration

In der Basisanwendung werden beispielhaft für alle funktionalen Module des späteren Automatisierungssystems jeweils eigene Zielautomatisierungsressourcen (siehe Schlüssel der Konfigurationspakete) angegeben. Die Module (*isci.[...]*) werden mit als Namen wirkenden Schlüsseln versehen, mit denen sie später durch die Handler installiert werden. Zu beachten ist, dass über die Anwendungen hinweg unterschiedliche Schlüssel für die Module verwendet werden, um bei Instanziierung auf der einer gemeinsamen Automatisierungsressource Konflikte zu vermeiden.

Die Konfigurationselemente nehmen im Beispielfall für den Betrieb notwendige Parameteränderungen an den Modulen vor, indem Variablen in Konfigurationsdateien ersetzt werden. In der Beispielanwendung wird dies mit Einstellungen zu Spindelsteigung und Schrittmotorauflösung demonstriert. Die Definition des Konfigurationsvorgangs in einem Konfigurationselement ist den Typ des Konfigurationselements gebunden. Da die vorhandene Breite der Konfigurationselemente mit eigenen Kindklassen erweiterbar ist, sind alle notwendigen Variationen, auch beispielsweise Programmiervorgänge von externen Systemen wie Mikrocontrollern, implementierbar. Voraussetzung ist, dass der umsetzende Handler die Konfigurationselementtypen unterstützt.

```
1  "Konfigurationspakete": {
2    "SteuerungX": {"AchseX": "isci.schrittmotorachse"},
3    "SteuerungY": {"AchseY": "isci.schrittmotorachse"},
4    "SteuerungZ": {"AchseZ": "isci.schrittmotorachse"},
5    "SteuerungSpind": {"Spindel": "isci.modbusspindel"},
6    "SteuerungCnc": {"CNC": "isci.dreiachsansteuerung"},
7    "SchnittstelleMsb": {"MSB": "isci.msb"}
8  },
9  "Konfigurationselemente": {
10   "SteuerungX": [{
11     "Typ": "Parameter",
12     "Vorgang": {
13       "Ordner": "AchseX",
14       "Variablen": {"Steigung": "5", "Aufloesung": "400"}
15     }
16   }],
17   "SteuerungY": [{
18     "Typ": "Parameter",
19     "Vorgang": {
20       "Ordner": "AchseY",
21       "Variablen": {"Steigung": "5", "Aufloesung": "400"}
22     }
23   }],
24   "SteuerungZ": [{
25     "Typ": "Parameter",
26     "Vorgang": {
27       "Ordner": "AchseZ",
28       "Variablen": {"Steigung": "8", "Aufloesung": "400"}
29     }
30   }],
31   "SteuerungSpind": [{
32     "Typ": "Parameter",
33     "Vorgang": {"Ordner": "Spindel", "Variablen": {"Adresse": "1"}}
34   }]
35 }
```

Objekt 6.4: Konfigurationsteil, der die Systemkonfigurationen der Automatisierungsressourcen über auszurollende Module beschreibt

Datenmodell, Datenintegration

Auf die Anwendung wird ein Datenmodell ohne zusätzliche Dateneinträge gelegt. Es realisiert die logische Verknüpfung zweier Module über ihre Dateneinträge, ohne in die Module konfigurierend oder programmatisch eingreifen zu müssen. Die logische Verknüpfung wird zur Laufzeit durch eine Kopie des Wertes vom Schlüsseldateneintrag zu den Werten des oder der in der Liste aufgeführten Dateneinträge durch ein zusätzliches Modul vorgenommen. Voraussetzung für diese Integration von Modulen ist, dass die Modulschnittstellen, die implizit über ihre Dateneinträge gegeben sind, direkt verbunden werden können.

```
1  "Datenmodelle": [{
2    "Dateneinträge": [],
3    "Links": {
4      "ns=AchseX;s=pXist": ["ns=CNC;s=pXist"],
5      "ns=CNC;s=pXsoll": ["ns=AchseX;s=pXsoll"],
6      "ns=AchseX;s=vXist": ["ns=CNC;s=vXist"],
7      "ns=CNC;s=vXsoll": ["ns=AchseX;s=vXsoll"],
8      "ns=AchseY;s=pYist": ["ns=CNC;s=pYist"],
9      "ns=CNC;s=pYsoll": ["ns=AchseY;s=pYsoll"],
10     "ns=AchseY;s=vYist": ["ns=CNC;s=vYist"],
11     "ns=CNC;s=vYsoll": ["ns=AchseY;s=vYsoll"],
12     "ns=AchseZ;s=pZist": ["ns=CNC;s=pZist"],
13     "ns=CNC;s=pZsoll": ["ns=AchseZ;s=pZsoll"],
14     "ns=AchseZ;s=vZist": ["ns=CNC;s=vZist"],
15     "ns=CNC;s=vZsoll": ["ns=AchseZ;s=vZsoll"],
16     "ns=Spindel;s=nIst": ["ns=CNC;s=nIst"],
17     "ns=CNC;s=nSoll": ["ns=Spindel;s=nSoll"]
18   },
19   "Identifikation": "CncStandard",
20   "Name": "",
21   "Beschreibung": "",
22   "Typ": "isci.Daten.Datenmodell"
23 }]
```

Objekt 6.5: Datenmodell ohne eigene Dateneinträge zur Übergabe von Verknüpfungsinformationen zur Verbindung von Modulen über ihre Dateneinträge

An den Referenzierungen der Dateneinträge in den Verknüpfungen ist ihre jeweilige Herkunft (beispielsweise Module *CNC* und *Spindel*) erkennbar. Die Dateneinträge entsprechen jeweils logischen Aus- und Eingängen von Modul zu Modul. Eine Typprüfung wird nicht vorgenommen. Typumwandlungen sind, falls notwendig und durchführbar, von der Verknüpfungsumsetzung (Modul *link*) umzusetzen.

Funktionen und Ereignisse

Es sind drei Funktionen definiert, die auf jeweils einen Dateneintrag des Automatisierungssystems zurückführen. Die Funktionen schaffen eine Anweisungsschnittstelle, die über die Schnittstellenmodule des Automatisierungssystems nach außen zugänglich werden. Im Falle des Szenarios richtet sich die geschaffene Anweisungsschnittstelle an externe Leit- oder Auftragssysteme. Die exponierten Dateneinträge sind nicht prozessintern, sondern starten, halten und instruieren die Prozesslogik.

```
1  "Funktionen": [{
2      "Ziele": ["ns=CNC;s=Start"],
3      "Identifikation": "fStart",
4      "Name": "Start",
5      "Beschreibung": "Maschine Start",
6      "Typ": "isci.Anwendungen.Funktion"
7    }, {
```

```
8      "Ziele": ["ns=CNC;s=Halt"],
9      "Identifikation": "fHalt",
10     "Name": "Halt",
11     "Beschreibung": "Maschine Halt",
12     "Typ": "isci.Anwendungen.Funktion"
13   }, {
14     "Ziele": [
15       "ns=CNC;s=posXsoll", "ns=CNC;s=posYsoll",
16       "ns=CNC;s=posZsoll", "ns=CNC;s=nSpsoll"
17     ],
18     "Identifikation": "fSchritte",
19     "Name": "Schritte Programm",
20     "Beschreibung": "Sollschritte der Bearbeitung",
21     "Typ": "isci.Anwendungen.Funktion"
22   }
23 ]
```

Objekt 6.6: Funktionsmodelle in der Anwendung zur Instanziierung der Funktionsschnittstellen des Automatisierungssystems

Die Funktion *fSchritte* exponiert mehrere Dateneinträge. Über das Modul *SchnittstelleMsb* (siehe Konfiguration) wird ein Client für die ereignisorientierte SoA-Middleware MSB instanziiert. Diese tauscht Nachrichten über eine zentrale Serverinstanz aus. In der Abbildung 6.10 ist der Funktionsteil der instanziierten Middleware-Schnittstelle in der Middleware-GUI dargestellt. Geöffnet sind die Beschreibungen der Funktionen *fStart* und *fSchritte*. Die Beschreibung des Funktionsteils entstammt der Selbstbeschreibung, die der Client aus den Ereignis- und Funktionsmodellen des Automatisierungssystems erstellt. Das Automatisierungssystem des Szenarios ist somit über die Middleware mit nicht-zeitkritischen Informationen integrierbar.

Es sind vier Ereignisse im Modell definiert, über die das Automatisierungssystem Daten nach außen trägt. Zu erkennen ist die Bindung an definierte Dateneinträge als Auslöser für die Verteilung von Daten sowie die Einbettung von Dateneinträgen als Ereigniselemente. Analog zu der Anweisungsschnittstelle, die über die Funktionen definiert wird, wird mit den Ereignissen eine abstrahierende Informationsschnittstelle gestellt, die keine prozessinternen, aber prozess- oder systembeschreibende Informationen nach außen liefert.

```
1  "Ereignisse": [{
2      "Ausloeser": "ns=CNC;s=Gestartet",
3      "Elemente": ["ns=CNC;s=Gestartet"],
4      "Identifikation": "eGestartet",
5      "Name": "Gestartet",
6      "Beschreibung": "Maschine gestartet",
7      "Typ": "isci.Anwendungen.Ereignis"
8    }, {
9      "Ausloeser": "ns=CNC;s=Gehalten",
10     "Elemente": ["ns=CNC;s=Gehalten"],
```

```
11      "Identifikation": "eGehalten",
12      "Name": "Gehalten",
13      "Beschreibung": "Maschine gehalten",
14      "Typ": "isci.Anwendungen.Ereignis"
15    }, {
16      "Ausloeser": "ns=CNC;s=Abschluss",
17      "Elemente": ["ns=CNC;s=Abschluss"],
18      "Identifikation": "eAbschluss",
19      "Name": "Abschluss",
20      "Beschreibung": "Bearbeitung abgeschlossen",
21      "Typ": "isci.Anwendungen.Ereignis"
22    }, {
23      "Ausloeser": "ns=CNC;s=Status",
24      "Elemente": [
25        "ns=CNC;s=AktuellerSchritt",
26        "ns=CNC;s=pXist", "ns=CNC;s=pYist",
27        "ns=CNC;s=pZist", "ns=CNC;s=nIst"
28      ],
29      "Identifikation": "eStatus",
30      "Name": "Status",
31      "Beschreibung": "Status verteilen",
32      "Typ": "isci.Anwendungen.Ereignis"
33    }
34  ]
```

Objekt 6.7: Ereignismodelle in der Anwendung zur Instanziierung der Ereignisschnittstellen´ des Automatisierungssystems

Ressourcenkartierung

Die Menge der adressierten Automatisierungsressourcen ergibt sich aus den Werten der Schlüssel-Werte-Paare der Kartierungen. Im Beispielfall werden vier Automatisierungsressourcen über ihre Identifikationen in das Automatisierungssystem eingehängt: *RevPiXYZ*, *RevPiSpind* sowie *PC*. Die Freinamen (* in den Modellen) der Standardanwendungen *IntegrationStandard*, *Zykluszeit* und *Systemabbilder* werden beibehalten, sodass die enthaltenen Konfigurationspakete und -elemente auf alle aufgezählten Automatisierungsressourcen angewandt werden. Für die Konfigurationspakete und -elemente der Anwendungsinstanz ist keine Ressourcenkartierung notwendig, da die Automatisierungsressourcen hier explizit identifiziert werden. Welche Automatisierungsressourcen Teil des Automatisierungssystems werden, ergibt sich durch die Ressourcenkartierung und die Konfigurationen in der Anwendungsinstanz. Die Modellierung ist darauf angewiesen, dass an mindestens einer Stelle Automatisierungsressourcen eindeutig identifiziert werden.

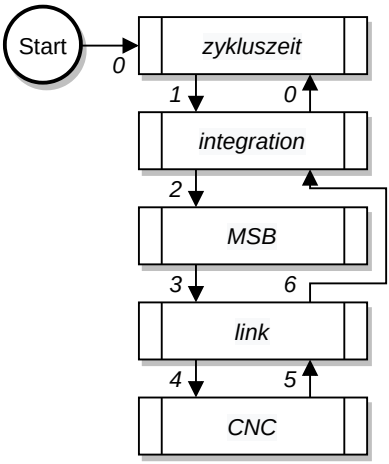
```
1  "Ressourcenkartierung": {
2    "anwendung.IntegrationStandard": {"*": "*"},
3    "anwendung.Zykluszeit": {"*": "*"},
4    "anwendung.Systemabbilder": {"*": "*"},
5    "anwendung.CncStandard": {
6      "SteuerungX": "RevPiXYZ",
7      "SteuerungY": "RevPiXYZ",
8      "SteuerungZ": "RevPiXYZ",
9      "SteuerungSpind": "RevPiSpind",
10     "SteuerungCnc": "PC",
11     "SchnittstelleMsb": "PC"
12   }
13 }
```

Objekt 6.8: Ressourcenkartierung im Instanzanwendungsmodell zur Auflösung der vergebenen Platzhalteridentifikationen zu existierenden Automatisierungsressourcen

Lokale Ausführung

An der Automatisierungsressource *PC* wird der in der Anwendungsinstanz beschriebene lokale Ausführungszyklus gezeigt. Die Module werden über ihre vergebene Identifikation referenziert. Die Verknüpfung mit der technischen Umsetzung der Ausführung wird durch den Handler vorgenommen. Am Entwicklungssystem der funktionalen Validierung erfolgt diese Umsetzung durch Ablage der Ein- und Ausgangszustände in den Parametrierungen der Modulimplementierungen. Der Ausführungszyklus und ein Ablaufplan sind in der Abbildung 6.9 zu erkennen. Es sind fünf Module über das Anwendungsmodell auf der Automatisierungsressource vorgesehen. Zwei Module (*integration*, *link*) sind in mehreren Ausführungszuständen eingeplant. Der aktuelle Zustand wird über lokale Zustandsdateneinträge in der Datenstruktur beschrieben und aktualisiert. Die definierte Abfolge wird zur Laufzeit des Automatisierungssystems ununterbrochen ausgeführt. Das erste Modul (*zykluszeit*) stellt den Einstiegspunkt, zu dem das letzte Modul (*integration*) wieder zurückführt. Die Definition des Ausführungszyklus ist als menschlesbares, textbasiertes Modell vorgesehen. Eine graphische Modellierung, dem ein Textmodell zugrunde liegt, ist möglich, aber kein Bestandteil des ISCI.

```
1  "PC": {
2    "zykluszeit": [{
3        "Eingangszustand": 0,
4        "Ausgangszustand": 1
5    }],
6    "integration": [{
7        "Eingangszustand": 1,
8        "Ausgangszustand": 2
9      }, {
10       "Eingangszustand": 6,
11       "Ausgangszustand": 0
12   }],
13   "MSB": [{
14       "Eingangszustand": 2,
15       "Ausgangszustand": 3
16   }],
17   "CNC": [{
18       "Eingangszustand": 4,
19       "Ausgangszustand": 5
20   }],
21   "link": [{
22       "Eingangszustand": 3,
23       "Ausgangszustand": 4
24     }, {
25       "Eingangszustand": 5,
26       "Ausgangszustand": 6
27   }]
28 }
```



Objekt 6.9: Ausführungsmodell und zugehöriges Ablaufdiagramm für die Module einer Automatisierungsressource

6.3 Ergebnisse

Entwickelt wurde ein implementierungsnahes Framework für die Umsetzung von Automatisierungssystemen als verteilte Systeme (siehe Anhang A.1), das prototypisch in einer Hochsprache implementiert wurde. Das ISCI setzt dabei einen Rekonfigurationsansatz als Lösung für das Adaptierbarkeitsproblem (siehe Abschnitt 2.1.2) um und führt einen datenfokussierten Ansatz im verteilten System ein, über den die Integration des verteilten Systems, die Integration mit der Umgebung und die Umsetzung von Verhaltens- und Echtzeitbewertungen realisiert werden. Das ISCI ist ausgelegt auf die Verwendung von Standardsystemarchitekturen bei seinen technischen Plattformen und die Nutzung von deterministischen Betriebssystemen (siehe Anhang A.2, A.2.3). Entwicklung und Validierung des ISCI erfolgten anhand einer funktionalen und einer Szenariovalidierung. In den

folgenden Abschnitten wird das entwickelte ISCI mit Bezug zu den gestellten Anforderungen und existierenden Frameworks zur Modernisierung (siehe Abschnitt 3.2) besprochen, herausragende Vorteile und Schwächen gezeigt sowie Untersuchungspotentiale vorgestellt.

6.3.1 Einhaltung der generellen Anforderungen

Es wurden sieben generelle Anforderungen festgelegt (siehe Abschnitt 1.3), denen in der Frameworkentwicklung mit Paradigmen und Artefakten begegnet wurde:

1. **Aufarbeitung *Cyberphysischer Systeme* für die Automatisierung:**
   Kooperation und Vernetzung wurden als die Kernmerkmale cyberphysischer Systeme identifiziert und sind Hauptparadigmen für Entwicklungsentscheidungen für das ISCI: Technische Systeme (Automatisierungsressourcen) in den verteilten Systemen (Automatisierungssysteme) arbeiten mit dauernder Vernetzung und Datenaustausch um eine gemeinsam zu steuernde Physis. Jede Automatisierungsressource kennt die Zustände aller Automatisierungskomponenten und hat damit das größtmögliche Abbild des physischen Zustands, was als Basis für die Kooperation dient, da nicht mehr gegenseitige Methodenaufrufe den Prozess leiten, sondern Automatisierungsressourcen eigenständig Logiken bei gleichzeitiger Beachtung der Vorgänge auf anderen Automatisierungsressourcen umsetzen können.

2. **(Re)Konfiguration als native Funktion in die Automatisierung einführen:**
   Es werden Anwendungsmodelle eingeführt, die den Aufbau eines Automatisierungssystems und die Konfiguration seiner Automatisierungsressourcen beschreiben und die über eine Administrationsplattform bereitgestellt werden. Automatisierungsressourcen, die in das Automatisierungssystem eingebettet sind, erhalten Handler als verwaltende Systeme. Handler prüfen kontinuierlich die Beschaffenheiten der Automatisierungsressourcen und zugewiesener Anwendungsmodelle und führen Konfigurationen und Rekonfigurationen der Automatisierungsressourcen selbstständig durch.

3. **Automatisierungsanforderungen (u.a. Zeit, Interoperabilität) abdecken:**
   Das ISCI etabliert ein gemeinsames Datensystem im Automatisierungssystem, wodurch die Zustandsorientierung, die für die Erstellung von Software für kritische

deterministische Systeme notwendig ist, auf allen Automatisierungsressourcen eines Automatisierungssystems ermöglicht wird. Das Datensystem, das auf einem Variablen- und Objektsystem mit Standardtypen orientiert an der IEC 62541 aufbaut, ist kartierbar auf verbreitete Programmiersprachen sowie übliche Kommunikationsprotokolle, wodurch Module für Interoperabilität breit implementiert werden können. Der lokale Ausführungszyklus auf einem Automatisierungssystem sowie die mögliche Einführung von Zyklen über die Datenstrukturen im Automatisierungssystem erlaubt die Erzeugung eines vorhersagbaren und gleichbleibenden Ausführungs- und Reaktionsverhaltens. Über die Beobachtung der Datenstrukturen sind Zeit- und Systemverhalten des Automatisierungssystems sowie der Automatisierungsressourcen nativ und softwarebasiert untersuchbar, ohne das Automatisierungssystem oder die Automatisierungsressourcen über externe Systeme beobachten zu müssen.

4. **Transparenz und Verständlichkeit zur Entwicklungs- und Laufzeit:**
   Alle Objekte und Beschreibungen sowie Modelle werden dateibasiert abgelegt und verteilt. Durch einfache Basisstrukturen und Mindestumfänge bei Metadaten ist der für das Verständnis notwendige Aufwand klein gehalten. Alle serialisierten Daten sind in der Umsetzung in unmittelbar menschenlesbaren JSON-Dateien abgelegt. Anwendungsmodelle zur Erstellung von Automatisierungssystemen sind struktur- und ablauforientiert und abstrahieren Programmquelltexte gegenüber dem Entwickler. Zur Laufzeit existieren vollständige Systemabbilder auf den Automatisierungsressourcen über die Datenstrukturen, wodurch über den Zugriff auf eine Automatisierungsressource ein größtmögliches System- und Prozessverständnis erlangt werden kann.

5. **Konfigurierbare logische Integration verteilter Systeme ermöglichen:**
   Die logische Integration der Automatisierungsressourcen zum Automatisierungssystem wird über die Datenmodelle, die Bestandteil der Anwendungsmodelle sind, abgedeckt. Da die Automatisierungsressourcen lokal identisch aufgebaute Datenstrukturen auf Basis der Datenmodelle erzeugen und kein punktueller Datenaustausch notwendig ist, ist auf allen Automatisierungsressourcen dieselbe Logikbasis gegeben, die bei Konfiguration und Rekonfiguration durch die Handler aktualisiert wird.

6. **Standardparadigmen und freie Software nutzen und fördern:**
   In der Konzeptausarbeitung wurden durchgängig Objektorientierung und Dateibasierung (entsprechend Everything-is-a-file-Ansatz) genutzt. Automatisierungsrelevante Daten können mit dem ISCI in ein Dateisystem einer Automatisierungsressource eingebettet werden, vergleichbar mit Gerätezugriffen auf Kommunikation und Hardware auf Unix-Systemen. Die Umsetzung für Entwicklung und Validierung wurden mit allgemein verwendbaren Technologien und Bibliotheken durchgeführt. Sie kann unmittelbar weiter genutzt oder auf andere Technologien übertragen werden.

7. **Implementierungs- und Anwendungsnähe wahren:**
   Alle Artefakte des ISCI wurden in der Entwicklung von Beginn an umsetzbar ausgelegt. Das Datensystem orientiert sich bei Datentypen und Identifikation an der IEC 62541, ist also standardisiert und implementierbar. Serialisierungen und Ablagen erfolgen mittels JSON. Technische Artefakte wurden in einer objektorientierten Sprache implementiert, deren genutzte Funktionen unmittelbar auf andere objektorientierte Sprachen übertragbar sind. Alle notwendigen Artefakte konnten in der Umsetzung implementiert und vollständig genutzt werden.

Allen festgelegten generellen Anforderungen konnten mit dem ISCI und seinen Umfängen, soweit bewertbar, begegnet werden. Für weitere Bewertungen und mögliche Weiterentwicklungen müssen zusätzliche Validierungsfälle versucht werden.

6.3.2 Vergleich mit anderen Frameworks

Auf der Basis der in den Tabellen 6.2 und 6.3 beschriebenen Ansätze sowie deren Vor- und Nachteile erfolgt eine qualitative Bewertung gegen Einzelkriterien, um eine Gegenüberstellung des ISCI und der ausgewählten existierenden Frameworks vornehmen zu können. Die Einzelkriterien sind in sieben Kategorien unterteilt:

1. Modellierung verteilter Systeme
   a) Modelle zur Definition anwendungsbezogener verteilter Systeme
   b) Native Werkzeuge zur Modellierung

2. Integration verteilter Systeme
    a) Anfallender Aufwand bei Verbindungsdefinition
    b) Umfang, Qualität zur Verfügung stehender Protokolle, Protokolleignung(en)
    c) Rahmen zur Erweiterung mit anderen Protokollen
3. (Re)Konfiguration, Softwareverteilung
    a) Formate, Modelle für Softwarekonfigurationen
    b) Umfang, Qualität Verteilfunktionen für Softwarekonfigurationen
    c) Umfang, Qualität Automatisierung der Softwarekonfigurationen
4. Determinismusfunktionen
    a) Zustandsorientierung am physischen Zustand
    b) Gesicherte, vorhersagbare Ausführung
    c) Native Werkzeuge zur Ermittlung des Systemverhaltens
5. Integration mit externen Systemen
    a) Native Möglichkeiten zur Integration mit externen Systemen
    b) Aufwand zur Adaption an externe Systeme
    c) Umfang, Qualität Interoperabilität im verteilten System
6. Abbildung, Administration
    a) Abbildung der Zielsysteme zu ihrer Laufzeit
    b) Abbildung der verteilten Systeme zur Laufzeit
7. Standardisierung
    a) Umfang, Qualität Standardisierung zum Framework

ISCI gegen / Kriterium	IEC 61499	FDI	VWS	AUTOSAR	DOME
1a. Modelle verteilte Systeme	0	2	2	1	0
1b. Werkzeuge Modellierung	-1	1	2	0	0
2a. Aufwand Verbindungsdefinition	-1	2	2	0	1
2b. Protokolle, Protokolleignung	0	2	2	1	0
2c. Mögliche Erweiterung	0	0	0	1	0
3a. Formate, Modelle Konfiguration	1	-1	1	-1	1
3b. Verteilfunktionen	1	0	2	-1	0
3c. Automatisierung	1	1	1	-1	1
4a. Zustandsorientierung	1	2	2	0	0
4b. Gesicherte Ausführung	0	2	2	-1	0
4c. Werkzeuge Systemverhalten	1	2	2	0	1
5a. Integration mit externen Systemen	1	1	0	2	2
5b. Aufwand zur Adaption	0	1	0	1	1
5c. Interoperabilität	0	1	1	1	0
6a. Abbildung Zielsysteme	2	0	0	2	2
6b. Abbildung verteilte Systeme	2	2	0	2	2
7a. Standardisierung	-2	-2	-2	-2	0
Durchschnitt nach Kategorie					
1. Modellierung verteilter Systeme	-1	2	2	1	0
2. Integration verteilter Systeme	0	2	1	1	0
3. (Re)Konfiguration, Softwareverteilung	1	0	1	-1	1
4. Determinismusfunktionen	1	2	2	0	0
5. Integration mit externen Systemen	0	1	0	1	1
6. Abbildung, Administration	2	1	0	2	2
7. Standardisierung	-2	-2	-2	-2	0

Tabelle 6.1: Die Bewertung des ISCI hinsichtlich der Bewertungskriterien der Liste 6.3.2 gegenüber den Referenzen. Die Bewertung erfolgt qualitativ mit Wertungen von -2 (ISCI deutlich schlechter) bis 2 (ISCI deutlich besser) auf Basis der Tabellen 6.2 und 6.3. Im unteren Teil werden die durchschnittlichen Wertungen jeder Kategorie gerundet wiedergegeben.

Vorteile gegenüber bisherigen Ansätzen

Die Bewertung anhand der Kriterien ist in Tabelle 6.1 hinsichtlich einzelner Kriterien sowie der gruppierten Kategorien dargestellt. Aus der Einschätzung der Tabelle sowie aus den Aufstellungen nach den Tabellen 6.2 und 6.3 werden mehrere ISCI-Ansätze gesehen, die Vorteile gegenüber den identifizierten Normen und Frameworks ermöglichen:

- Die **Datenorientierung als Basis für das verteilte System** ermöglicht große Modularisierbarkeit und Interoperabilität bei der Integration von Software und mehreren Automatisierungssystemen zu einem Automatisierungssystem durch die Verbindung dieser Komponenten auf dem gemeinsamen technischen Ansatz von Variablen und Objekten. Die datenorientierte Integration erlaubt zudem eine
  - **Zustandsorientierung** auf Automatisierungsressource und Automatisierungssystem, die für die Steuerungsprogrammierung in kritischen physischen Systemen notwendig ist (siehe auch Abschnitt A.1.2) sowie die
  - **Verhaltensbeobachtung** von Automatisierungsressourcen und Automatisierungssystemen über Datenabbilder, was die praktische Validierung von erstellten Automatisierungssystemen bezogen auf ihr Zeit- und Systemverhalten sowie einzelne Kennwerte wie Jitter (siehe Abschnitt 2.1.3) ermöglicht.
- Die native Definitionsmöglichkeit von **Schnittstellen für eine externe Integration** über Funktionen und Ereignisse führt einen Ansatz ein, mit dem ein kontrollierter Zugriff auf Daten und Modelle des Automatisierungssystems ermöglicht wird, ohne dass ein Eingriff in programmatische Strukturen notwendig ist. So kann ein erstelltes verteiltes System über seine Konfiguration an nicht prozesskritische externe Systeme angebunden werden, was neue Abstraktionsansätze (Abschnitt 3.1) unterstützt. Ein Beispiel dafür ist die Integration mit einer SoA-Middleware in der Szenariovalidierung (siehe Abbildung 6.12), die bei anderen Frameworks nicht implementierbar ist.
- Der **Handleransatz** zur Konfiguration von Automatisierungsressourcen führt eine zusätzliche Ebene zwischen technischer Ressource und Entwicklungsebene ein, die als eigenständige Softwarebrücke fungiert. Durch Handler als Softwarebrücke sind Rekonfiguration als Adaptierbarkeitsansatz (siehe Abschnitt 2.1.2) und die In-

tegration im Automatisierungssystem automatisierbar, indem Handler eigenständig Ressourcenkonfigurationen nach Modellen und Beschreibungsteilen vornehmen.

Schwächen gegenüber vorherigen Ansätzen

Den Vorteilen der ISCI-Ansätze stehen einige Schwächen gegenüber:

- Verteilte Systeme um einen physischen Prozess wie die ISCI-Automatisierungssysteme sind bislang in der Praxis und damit bei Entwicklern und Anwendern nicht bekannt. Die neuen Komplexitäten, die mit der Umsetzung von Automatisierungsstrukturen als verteilte Systeme entstehen, können eine Herausforderung darstellen. Diese zeigt sich voraussichtlich am stärksten in den Modell- und Datenstrukturen. Die Definition eines Automatisierungssystems über ein oder mehrere Anwendungsmodelle kann zu einer großen **Entwicklungskomplexität** führen. Die Aggregation aller Datenmodelle in konzentrierten Datenstrukturen kann die Übersichtlichkeit für Entwickler erschweren.

- Das ISCI beschreibt Artefakte und anwendbare Technologien, ist aber im Gegensatz zu IEC-61131-Industriesystemen oder neuen Ansätzen wie der IEC 61499 noch ohne **Entwicklungswerkzeuge, -methoden und -vorgehen**. ISCI wirkt in verschiedenen Lebenszyklusphasen: mit ihm werden Automatisierungsressourcen konfiguriert, allgemeine und spezifische Anwendungen definiert, Automatisierungssysteme instanziiert und validiert, sowie Softwaremodule erstellt. Alle beschriebenen Wirkpunkte benötigen für eine effektive Nutzung eine Entwicklungs- und Betreuungsumgebung orientiert an bekannten Normen, Technologien und wissenschaftlichen Arbeiten im Bereich verteilter Automatisierungssysteme (beispielsweise nach Eckert 2015).

- Für die Validierung im Automatisierungssystem, also über ein einzelne Automatisierungsressource hinweg, setzt ISCI ein gemeinsames Zeitverständnis aller Automatisierungsressourcen voraus, um konsistente Abbilder generieren zu können. **Synchronisationsfunktionen** wie in Ethercat- oder Profinet-Feldbussen sind nur mit besonderer Hard- und Software verfügbar und müssen an jeder Automatisierungsressource installiert sein. Das allgemein unterstützte NTP ist für Genauigkeiten kleiner als eine Millisekunde nur bedingt fähig. Das fähigere PTP muss auf Automatisierungsressourcen gesondert implementiert und konfiguriert werden.

Erzielbare Verbesserungen in der Anwendung

Die Automatisierungstechnik ist durch industrielle Entwicklungen geprägt, die inkrementell und stark anwendungsorientiert erfolgen. Technologien werden in der wirtschaftlich getriebenen Anwendung über die drei Kategorien Zeit, Kosten und Qualität bewertet. Praktisch relevant sind fast ausschließlich Industrieadaptionen nach der IEC 61131 (Rothhöft 2022). In allen Kategorien werden Verbesserungen insbesondere gegenüber der IEC 61131 erzielt:

- **Zeit:** Die native Ausrichtung auf verteilte Systeme und einen modellgetriebenen Aufbau dieser Systeme aus wiederverwendbaren und generalisierbaren Softwaremodulen reduziert Implementierungs- und Einrichtungsaufwände, insbesondere bei komplexen Maschinen und Anlagen mit umfangreicher Automatisierung. Mit ISCI kann ein ganzheitliches Engineering einer verteilten Automatisierung in einem konzentrierten Modell erfolgen, das dann in einem Schritt auf die Automatisierungsressource wiederholbar konfiguriert wird. Es ist keine systemspezifische Programmierung und Konfiguration mehr notwendig. Zudem werden Eingriffe in programmatische Strukturen minimiert. Die Funktion modellbasiert Schnittstellenmodule auszuliefern und sie konfigurationsbasiert einzurichten, verschnellert die Integration mit der Umgebung.

- **Kosten:** Durch Orientierung an Standardplattformen und reine Softwarebasierung kann mit ISCI im besten Fall eine Reduktion notwendiger Hardware auf Sensorik, Aktorik und Netzwerkelemente erfolgen. Der Entfall der kosten- und installationsintensiven Steuerungshardware erzeugt eine erhebliche Kostenverringerung. Durch die Nutzung von Standardtechnologien sind Sprachen und Werkzeuge frei wählbar und es müssen keine Kosten für Herstellersoftware aufgewendet werden.

- **Qualität:** ISCI konzentriert Entwicklungs- und Laufzeitelemente verteilter Systeme in wenigen, jeweils eindeutigen Modellen, wogegen bisherige Technologien für jede technische Plattform (beispielsweise ein Steuerungssystem) einzelne und isolierte Entwicklungs- und Laufzeitelemente vorsehen. So verfügt das verteilte System über eine konsistente Struktur, wodurch Entwicklungs- und Laufzeitqualitäten gegenüber den bisherigen isolierten Systemen verbessert werden. Durch die konzentrierten Laufzeitelemente (Datenmodelle und Datenstrukturen) werden verteilte Systeme im Laufzeitverhalten nativ transparent und gemeinsam validierbar.

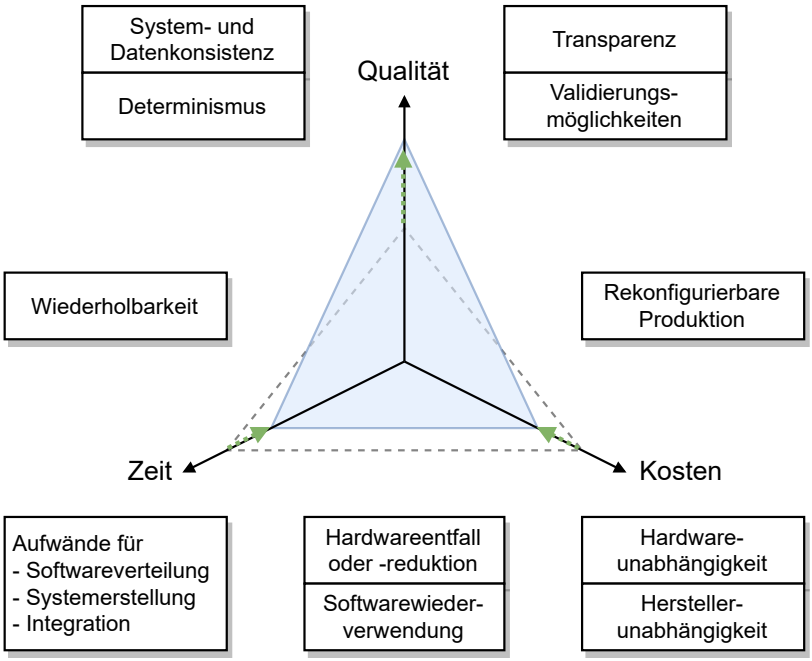


Abbildung 6.11: Qualitative Verbesserungen durch ISCI in der Betrachtung von Zeit- und Kosten-aufwänden und erzielbarer Qualität

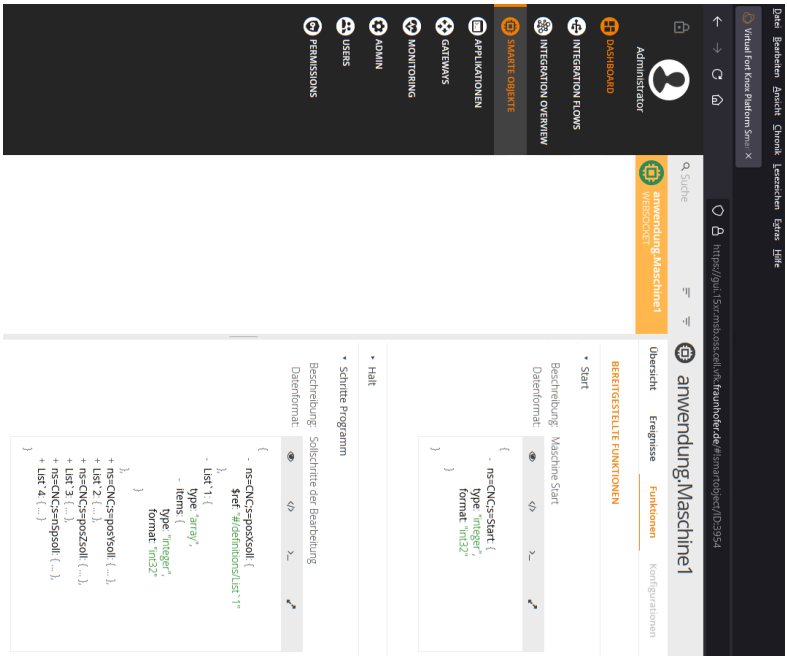


Abbildung 6.10: Durch das *MSB*-Modul auf Basis der Funktionsmodelle im Szenarioanwendungsmodell implementierte Funktionen in der Oberfläche einer *SoA*-Middleware-Instanz

6.3.3 Untersuchungspotentiale

Ausgehend von den Validierungserkenntnissen sowie den im Vergleich mit anderen Frameworks ermittelten Nachteilen (siehe Tabellen 6.2, 6.3 und 6.1) werden Untersuchungspotentiale gesehen, die für eine Weiterentwicklung aufgegriffen werden können.

Untersuchung mit weiteren Ansätzen

Das ISCI ist aufgrund seiner Ausrichtung auf verbreitete Standardarchitekturen geeignet für eine eingehendere Untersuchung und mögliche Erweiterungen mit Ansätzen aus verschiedenen Gebieten. Es werden vier Themen als besonders untersuchungswert gesehen:

- **Tiefere Einbettung in Betriebssysteme und Systemarchitekturen:** Der dateibasierte Zugriff des Datensystems in der Umsetzung für eingebettete Automatisierungsressourcen bietet einen Ausgangspunkt zur nativen Integration des Datensystems in ein Betriebssystem, sodass Automatisierungsdaten und der Zugriff darauf fester Funktionsbestandteil eines Betriebssystems sein könnten. In der Umsetzung wird ein arbeitsspeicherbasiertes Dateisystem als lokale Datenstruktur genutzt. Die Erweiterung einer Systemarchitektur zum direkten Austausch von Automatisierungsdaten über einen gemeinsamen Speicherbereich (beispielsweise orientiert an Cheng & Zhang 2011), Kernelfunktionen oder Gerätetreiber unter Beibehaltung des dateibasierten Zugriffs können Leistungsverbesserungen und native automatisierungsorientierte Funktionen ermöglichen.

- **Verbindung der Modellstrukturen mit generativer künstlicher Intelligenz (KI):** Verteilte Systeme werden in ISCI über textbasierte Beschreibungsmodelle beschrieben. Diese Modelle können aufgrund ihrer Textbasierung unmittelbar Gegenstand generativer sprachbasierter KI-Verfahren werden. Mit einer ausreichenden Basis an Anwendungs- und Datenmodellen und in Verknüpfung mit weiteren externen Eingangsdaten wie Schaltplänen oder Stücklisten könnten automatisiert Modelle für die Erstellung von Automatisierungssystemen generiert werden.

- **Verbindung mit Containermodularisierung:** Das Konfigurationsvorgehen des ISCI lässt sich mittels Containermodularisierung umsetzen, indem für Module Containerstrukturen vorbereitet werden, die in der Umsetzung einer Konfiguration instanziiert

werden. Die Dateibasierung der Umsetzung ermöglicht eine direkte Verbindung mit Containertechnologien. Das ISCI würde von der stärkeren Unabhängigkeit von der Systemarchitektur sowie den Möglichkeiten zur Softwareverteilung bei Containern ergänzt. Container sind automatisierungsfähig (siehe Abschnitt 3.3.3).

- **Untersuchung mit Ethernet TSN:** Ethernet TSN ist eine mögliche neue deterministische Kommunikationsbasis für ethernetfähige Standardsysteme. Da das ISCI auf Verwendung mit diesen Systemen ausgerichtet ist, ist eine praktische Untersuchung zur Verbindung mit Ethernet TSN angebracht. Annehmbar ist, dass mit Ethernet TSN auch proprietäre Feldbusprotokolle abgelöst werden können.

Ausarbeitung der VWS-Integration

Die Integration des ISCI mit der VWS wurde konzeptionell betrachtet, da keine Referenzimplementierungen der VWS als aktive Softwarestruktur vorhanden waren und keine belastbare Beurteilung vorgenommen werden konnte. Für die Nutzung als abbildendes Modell oder als Struktur zur Sammlung von Metadaten und Funktionsbeschreibungen ist die VWS als kartierbar anzunehmen, wobei sich eine starke Abhängigkeit von der Modulumsetzung und dem möglichen Abbildungsdetailgrad ergibt. Wenn weitere Erkenntnisse zur VWS-Verwendung und den notwendigen Informationen und Funktionen bekannt sind, kann die Kartierung von ISCI-Elementen auf VWS ausgearbeitet werden. Die Umsetzung der Softwarestruktur auf Automatisierungsressourcen als VWS ist nicht interessant, da verschiedene ISCI-Elemente, wie die datenbasierte Integration von Modulen sowie die Ausführungssteuerung, keine direkte Entsprechung in der VWS-Struktur finden und ein Nutzen der notwendigen Architekturänderungen als nicht gegeben gesehen wird.

6.3.4 Betrachtung der Forschungsfragen

Im Abschnitt 1.6 werden Fragestellungen vorgestellt, mit denen, in Komponenten unterteilt, die Untersuchung durchgeführt wurde. Konzept und Umsetzung für die Arbeit wurden schrittweise und im Wechsel zwischen Konstruktion und Evaluation entwickelt (siehe Forschungskern in Abbildung 1.3 zur Wissenschaftsmethodik). Die Fragestellungen gaben

Anhaltspunkte, die in der Schleife von Konstruktion und Evaluation betrachtet wurden. Die Antworten wurden schrittweise entwickelt und bilden die Untersuchungsergebnisse:

- **Definition:** Wie müssen cyberphysische und virtuelle Komponenten im Automatisierungsnetz grundsätzlich beschaffen sein? Wie können Fähigkeiten wie Self-X-Funktionen (Selbstorganisation, Selbstkonfiguration etc.) verhaltensdeterminierbar und regelgerecht integriert werden?
  Untersuchungsergebnis: Notwendig ist die Vorgabe einer Softwarestruktur für Automatisierungssysteme mit Überwachungs- und Abbildungsfähigkeit. Die Automatisierungssysteme benötigen eine interne gemeinsame Verständnisbasis (insbesondere datenbezogen). Die Softwarestrukturen müssen im Sinne eines digitalen Zwillings in einem plattformunabhängigen Format spiegelbar sein, um gegen Modelle und allgemein geprüft werden zu können. Kombinationen vorgegebener und definierbarer Strukturen auf Automatisierungskomponenten müssen implementierbar sein, über abgeschlossene und erweiterbare Komponenten mit programmier- und konfigurierbaren Teilen sowie festen Rahmendefinitionen.

- **Struktur:** Wie sind Modellierung, Organisation und Verwaltung eines Automatisierungsnetzes zu gestalten und mit kollektiven Self-X-Funktionen (z.B. Fähigkeitsbeschreibung) zu vereinen?
  Untersuchungsergebnis: Als Basis ist eine gemeinsame Modellierung über eine singuläre Plattform möglich. Die Modellierungsfähigkeiten müssen anwendungs- und gerätebezogen sein. Ein stehendes Informationsmodell je Gerät ist eine fähige Repräsentationsgrundlage, die mit gegebenen Modellen für ein verteiltes System erweitert werden können muss. Über anwendungsbezogene Modelle können einzubindende Softwareapplikationen vorgegeben werden, auch als Standard. Eine Aggregationsmöglichkeit von Modellen aus Komponenten, Softwareteilen und Anwendungen zu Gesamtmodellen ist hinreichend fähig, konzentrierte Beschreibungsmodelle für verteilte Systeme zu erzeugen, diese können aber Komplexitätshindernisse darstellen. Bestimmte Modelle sollten in Standardformate konvertierbar sein für eine Interoperabilität.

- **Kommunikation:** Können Vermittelungs- und Beschreibungsfähigkeiten im Automatisierungsnetz semantikoffen und generisch integriert werden?
  Untersuchungsergebnis: Die Anforderung ist teilweise bewältigbar. Es muss eine selbstdefinierte Beschreibung zugelassen werden, die einen Mindestumfang in einem bekannten Format aufweist. Erweiterung mit eigenen Beschreibungen und ein dateibasierter Ansatz für eine möglichst offene und leicht konvertierbare Struktur können fremde Semantiksysteme berücksichtigen. Das relevanteste System ist die Verwaltungsschale, deren Grundstruktur als Konvertierungsziel berücksichtigt werden kann.

- **Hierarchie:** Wie ist ein verhaltensdeterminierbarer Aufbau und Ablauf im Automatisierungsnetz beschaffen? Von wem und wie werden Aufgaben und Abläufe abstrahiert, abgestimmt und verteilt?
  Untersuchungsergebnis: Im Automatisierungsnetz müssen aufgaben- und zustandsorientierte Ressourcen mit überwachenden Applikationen und datengetriebener Evaluation vorhanden sein. Es sollte auf der verteilten, konfigurierten Software mit bekanntem Verhalten aufgebaut werden. Eingehende Aufgaben können über ein im Automatisierungsnetz gemeinsames Datensystem eingehen. Für den Determinismus müssen Aufgaben in der Anwendung über die Implementierungslogik getrennt werden. Programmsequenzen auf einem gemeinsamen Datenabbild in einem verteilten System ermöglichen eine verteilte, deterministische Logik. Ein gemeinsames Zeitverständnis über Synchronisation oder gleiche Zeitgeber muss vorhanden sein.

- **Koordination:** Wie werden parallel stattfindende Vorgänge und Abläufe auf verschiedenen Systemen, insbesondere bei zeit- und materialkritischen Aufgaben, koordiniert?
  Untersuchungsergebnis: Selbstständige Koordination bei parallel stattfindenden Vorgängen und Abläufen ist auf einem einzelnen System möglich, wenn hinreichende Informationen über das Gesamtsystem bereitstehen. Ein möglicher Weg dazu ist die Bereitstellung vollständiger Systemabbilder über aggregierte und kontinuierliche Datensysteme, die über das verteilte System konsistent gehalten werden. Für kritische Prozesse müssen Zeit- und Systemverhalten abstimm- und hart definierbar sein. Die Implementierung kann zustandsorientiert erfolgen, statt verbunden mit der Daten-

struktur eines Einzelsystems aber verbunden mit einer konsistenten Datenstruktur eines verteilten Systems. Die notwendige Verhaltensprüfung kann über Datenerfassung erfolgen. Ein gemeinsames Zeitverständnis über Synchronisation oder gleiche Zeitgeber muss vorhanden sein.

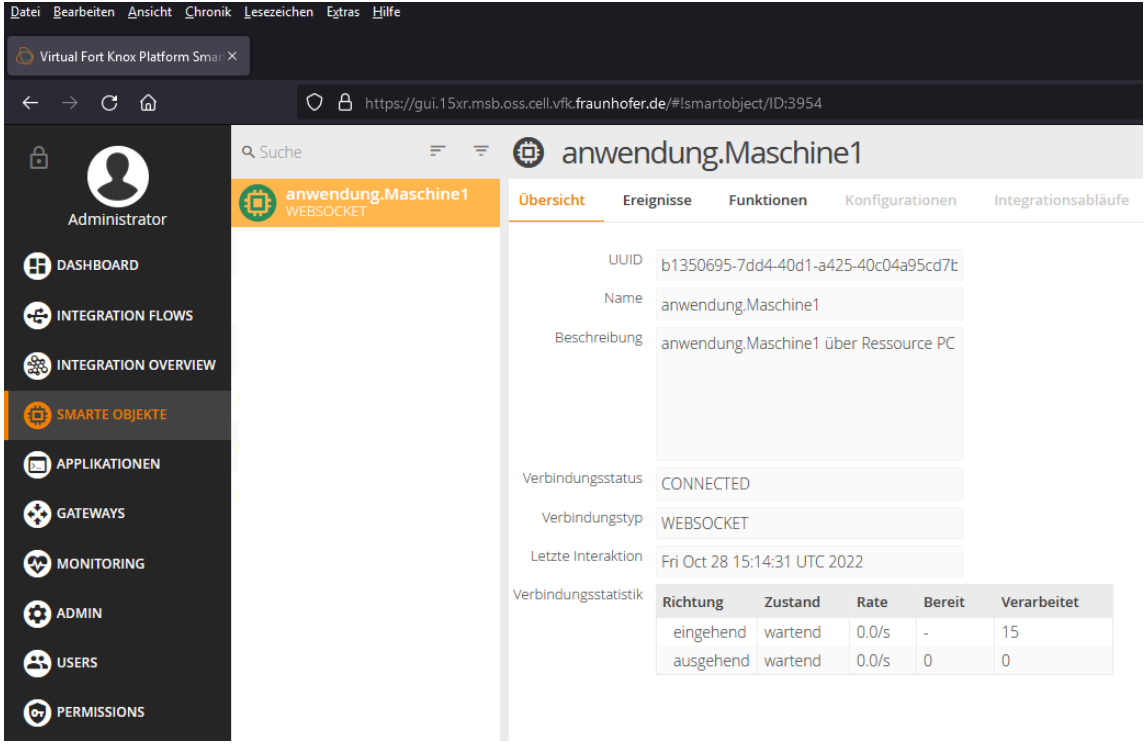


Abbildung 6.12: Über das *MSB*-Modul ist das Automatisierungssystem der Szenarioanwendung in eine Instanz der SoA-Middleware MSB eingebunden. Gezeigt wird die im Middlewareformat erstellte Beschreibung des Automatisierungssystem in der Middleware-GUI. Das Modul generiert aus Modellen und Daten des Automatisierungssystems die Beschreibung ohne auf andere Module einzuwirken.

Framework	Umsetzungsansatz	Vorteile gegenüber ISCI	Nachteile gegenüber ISCI
IEC 61499	• Auswahl 61499-fähiger Zielsysteme in der Produktion • Anwendungsmodellerstellung • Zielsystemberücksichtigung im Ressourcenmodell • Implementierung Steuerungsprogramm und Funktionalitäten mit Funktionsblöcken oder verborgen in neuen, spezifisch programmierten Funktionsblöcken • Verteilungsdefinition in der Anwendung, manuelle Softwareverteilung	• Standardisierung • Graphische Entwicklungsumgebungen mit eingebetteter Programmierung • Festgelegter Verteilprozess bei Implementierungen	• Automatisierungsrelevante Integration ist selbst zu implementieren • Keine Funktionalität zu Systembewertung, Verhaltensbestimmung • Determinismus generell schwierig (keine Zustandsorientierung, graphische Programmierung) • Keine Anwendungskaskadierung, muss über Spezialfunktionsblöcke stattfinden • Interoperabilität eingeschränkt, muss programmatisch eingebettet werden • Keine Abbildungs-, Administrationsfunktionen
FDI	• Auswahl FDI-fähiger Zielsysteme in der Produktion • Erstellung spezifischer Implementierung, Einbettung in Packages • Einbettung von spezifischen (Kommunikations)Parametern • Externe Erstellung eines Verteilungsplans als Anwendungsersatz • Manuelles Ausliefern der Packages nach Verteilungsplan	• Standardisierung • Umfassende Administrationsunterstützung und Werkzeuge für Administration • Integration mit Normen für einfache Komponentensysteme • Voll definiertes Package-System mit Logik-, Daten- und GUI-Elementen	• Kein Anwendungsbezug oder Verteilungsmodell, muss extern geplant werden • Automatisierungsrelevante Integration ist selbst zu implementieren • Keine Bereitstellung notwendiger Laufzeitfunktionalität • Keine Funktionalität zu Systembewertung, Verhaltensbestimmung

Tabelle 6.2: Bisherige Frameworks zur Modernisierung der Automatisierung bezogen auf die Szenariovalidierung und im Vergleich mit ISCI (1/2)

Framework	Umsetzungsansatz	Vorteile gegenüber ISCI	Nachteile gegenüber ISCI
VWS	• Umsetzung eingeschränkt, da keine Laufzeit- und Konfigurationsfunktionen oder aktive Softwarestrukturen definiert sind • Bau begleitender VWS mit Repräsentationsfokus möglich, mit Anbindung frei implementierter Funktionen an VWS-API	• Bedingte Standardisierung • Umfassende Repräsentation gegenüber Administrationssystemen • Standard-API zu Informationen und Methoden	• Kein Anwendungsbezug oder Verteilungsmodell, muss extern geplant und programmiert werden • Automatisierungsrelevante Integration ist selbst zu implementieren • Keine Bereitstellung notwendiger Laufzeitfunktionalität • Keine Funktionalität zu Systembewertung, Verhaltensbestimmung
AUTOSAR	• Modellierung einer verteilten Gesamtanwendung mit festgelegten, fähigen Zielsystemen • Festlegung, Implementierung von notwendigen Modulen, Bibliotheken, Funktionen • Kommunikationsmodellierung in der Gesamtanwendung • Anwendungsbau, Verteilung	• Standardisierung • Umfassende Software für Laufzeitfunktionen • Vorgegebene deterministische Kommunikation • Strikte Verhaltensermittlung vor Auslieferung möglich	• Umfang und Komplexität sehr groß • Keine umfassende Zustandsorientierung in Anwendungen möglich • Integration ist selbst zu modellieren • Interoperabilität eingeschränkt, programmatisch vorzusehen • Keine Abbildungs-, Administrationsfunktionen
DOME	• Erstellung eines Anwendungsobjekts, Einbettung fähiger Zielsysteme • Implementierung zusätzlicher Funktionen als DOME-Klassen • Modellierung des Steuerungsprogramms über funktionale, Kontext- und Integrationsobjekte • Anwendungsbau, Verteilung	• Untersuchungen zu Leistungsmetriken vorhanden • Verhaltensermittlung, logischer Determinismus möglich	• Integration ist selbst zu modellieren • Schwache Zustandsorientierung • Interoperabilität eingeschränkt, programmatisch vorzusehen • Keine Abbildungs-, Administrationsfunktionen

Tabelle 6.3: Bisherige Frameworks zur Modernisierung der Automatisierung bezogen auf die Szenariovalidierung und im Vergleich mit ISCI (2/2)

7 Zusammenfassung

Digitale Systeme in der Produktion sind geprägt von der bekannten hierarchischen Denkweise der Automatisierungspyramide und ihrer Unterteilung der Produktion in Ebenen. Systeme verfügen über stark unterschiedliche Technologien, starr umgesetzte Kommunikation und Integration, häufige Kommunikationsbrüche, dedizierte und schwer oder nicht änderbare Anwendungsimplementierungen sowie viele verschiedene Entwicklungs-, Modellierungs- und Administrationswerkzeuge. Die Systeme tauschen Daten aus, kooperieren aber nicht. Ihnen fehlt eine fähige, gemeinsame Grundstruktur, um die bisherigen isolierten und starren Architekturen in kooperierende, adaptierbare Systeme zu überführen, die die heutigen Potentiale softwaredefinierter Systeme nutzen. Dies gilt in besonderem Maße für die physisnahe Automatisierungstechnik, weil durch neue Vernetzungsanforderungen, günstige Hardware und offene Betriebssysteme die Zahl datenliefernder und -verarbeitender Komponenten um einen physischen Prozess besonders zunimmt. Die Notwendigkeit für eine neue Grundstruktur für die dynamische und einfache Bildung verteilter, logisch integrierter Systeme auf Basis von Standardtechnologien ist hier herausragend.

Die Arbeit begegnet dieser Notwendigkeit mit der Entwicklung des grundlagenorientierten *Integrations- und Steuerungsframework für die Cyberphysische Industrieautomatisierung (ISCI)*, das adaptierbare Strukturen zur Architekturdefinition von Automatisierungssystemen bereitstellt und ein darauf aufsetzendes Softwareframework implementiert. Mit dem ISCI wird eine softwaregetriebene und softwaredefinierte Automatisierung mit verteilten, kooperierenden Systemen nach CPS-Paradigmen umgesetzt. Es stellt Strukturen und Modelle für ein verteiltes System, für die daran beteiligten Einzelsysteme und für Softwarekomponenten bereit und ist während Entwicklung, Konfiguration und Betrieb aktiv. Es löst keine Steuerungsaufgaben, sondern stellt ein Grundgerüst für verteilte, rekonfigurierbare Automatisierungsanwendungen mit fähigen Schnittstellen bereit.

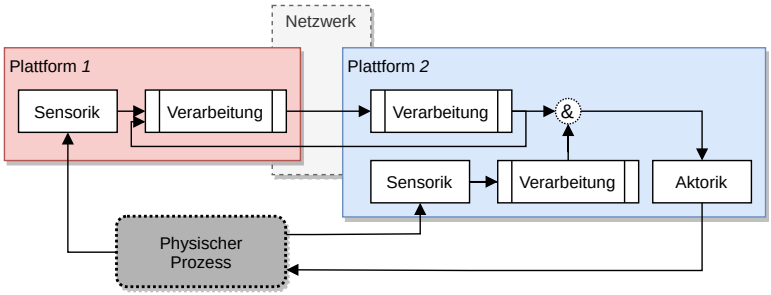


Abbildung 7.1: Beispiel für ein cyberphysisches System (nach Lee & Seshia 2012)

Ansatz *Cyberphysische Systeme*

Mit *Cyberphysische Systeme* (CPS) werden fest verbundene Zusammenstellungen rechenfähiger Systeme bezeichnet, die gemeinsam einen physischen Prozess steuern. Das einfachste CPS besteht, wie in Abbildung 7.1 zu sehen aus zwei verbundenen technischen Plattformen mit Sensorik und Aktorik sowie jeweils eigener Rechenfähigkeit. Die wesentlichen CPS-Eigenschaften sind die feste Vernetzung der Plattformen und die Kooperation in der Prozesssteuerung. Diese beiden Eigenschaften sind die wesentlichen CPS-Paradigmen, die als Grundlagen für verschiedenste Entwicklungen in der Wissenschaft genutzt werden. Die beiden Eigenschaften werden als Paradigmen in die Frameworkentwicklung einbezogen.

Das *Integrations- und Steuerungsframework für die Cyberphysische Industrieautomatisierung ISCI*

Auf Basis der heutigen allgemeinen Anforderungen, den speziellen Anforderungen der Automatisierung und einer Analyse bestehender Modernisierungsansätze für die Automatisierungsarchitekturen werden für das ISCI sechs spezielle Lösungsansätze definiert, mit denen den Aufgaben begegnet wird und die neben den CPS-Paradigmen die Frameworkentwicklung bestimmen:

- Dynamischer, modellbasierter Aufbau verteilter Systeme
- Gemeinsame Datenmodelle und datenbasierte Integration
- Determinismus über Zustandsorientierung

- Funktions- und Ereignisschnittstellen
- Aufbau auf Standardtechnologien
- Kartierbarkeit auf die Verwaltungsschale

Abhängig von der technischen Relevanz, der Verfügbarkeit von Implementierungen oder technischen Beschreibungen und der Möglichkeit, verteilte Systeme umzusetzen, werden fünf Frameworks als Referenzen für Weiterentwicklungsansätze evaluiert: die IEC 61499, Field Device Integration, die RAMI4.0-Verwaltungsschale, DOME und AUTOSAR. Sie dienen auch als Bewertungsreferenzen für das Framework.

Es wird eine Konzeptstruktur, bestehend aus logischen und technischen Artefakten skizziert, die als Strukturgrundlage Lösungsbestandteile für die einzelnen Ansätze sind. Alle Artefakte werden grundsätzlich und ohne unmittelbaren Technologiebezug beschrieben, um sie möglichst umsetzungsabstrakt und technologieoffen zu halten. Besonders relevante logische Artefakte sind *Anwendungsmodelle*, die den Aufbau, die Integration, die Daten- und Kommunikationsstruktur eines verteilten Systems beschreiben; *Daten-, Ereignis- und Funktionsmodelle*, über die funktionale zustands- und ereignisorientierte Strukturen im verteilten System entstehen; *Konfigurationen*, mit denen die Auslieferung von Software und Parametern auf technische Systeme organisiert wird; und *Beschreibungen*, die für verschiedene technische Artefakte herangezogen werden, um Repräsentations- und Integrationsprobleme zu lösen.

Die technischen Artefakte sind die aus Anwendungsmodellen erstellten verteilten Systeme als *Automatisierungssysteme*; die konfigurations-, integrations- und abbildungsfähigen einzelnen Systeme als *Automatisierungsressourcen*; die Softwareartefakte, die als Anwendungen oder Programme alle Sensor-, Aktor-, Analyse-, Verarbeitungs-, Steuerungs- und Kommunikationsfunktionen implementieren als *Module*; sowie die *Handler*, die sich um Konfiguration, Integration und Abbildung von Automatisierungsressourcen kümmern. Sie werden ergänzt durch eine *Administrationsplattform* und *Quellen*, die Modelle, Beschreibungen und Daten bereitstellen und entgegennehmen. Auf den Automatisierungsressourcen setzen *Datenstrukturen* die datengetriebene Integration des verteilten Systems um. Automatisierungsressourcen, die den Gesamtumfang des Frameworks umsetzen und damit

eingebettet werden können, werden als *eingebettete* Automatisierungsressourcen bezeichnet.

Automatisierungssysteme werden im Framework um ein gemeinsames Datenverständnis (Abbildung 4.3) gebaut: Automatisierungsressourcen im Automatisierungssystem verfügen über eine identische Grundmenge eindeutig identifizierbarer Variablen, sogenannter Dateneinträge, die aus statischen Datenmodellen gebildet wird. Diese Datenmodelle werden durch Anwendungsmodelle des Automatisierungssystems bereitgestellt. Dieses gemeinsame Datenverständnis bildet die Integrationsgrundlage: Alle Sensoren, Aktoren, die Kommunikation und verarbeitende Software arbeiten in einem Automatisierungssystem auf Basis des gemeinsamen Datenverständnisses. Über Ereignis- und Funktionsmodelle werden gezielt Dateneinträge exponiert, um die Integration mit externen Systemen umzusetzen.

Das gemeinsame Datenverständnis der statischen Datenmodelle wird im Betrieb durch die Datenmodellerweiterungen der Module ergänzt. Module auf einer Automatisierungsressource, fest installiert oder durch Konfiguration ausgeliefert, können zusätzliche Datenmodelle einbringen, die durch zyklische oder bedarfsweise Prüfung im Automatisierungssystem verteilt werden. Dadurch entsteht ein gemeinsames, umfassendes und teildynamisches Verständnis mit statischem Kern im verteilten System, wodurch eine ganzheitliche Kooperation der einzelnen Systeme statt punktueller Integration möglich wird. Variablen und die durch sie dargestellten Daten sind der kleinste gemeinsame Nenner fast aller Programmiersprachen, Laufzeitumgebungen und Kommunikationsprotokolle. Dies macht eine zentral modellierte und ständig abgeglichene Datenstruktur, deren Aufbau allen technischen Systemen bekannt und zugänglich ist, zu einem geeigneten Abstraktionsansatz, da er zum einen alle Informationen des Automatisierungssystems bereitstellt und zum anderen auf allen verbreiteten Architekturen umsetzbar ist. Der Ansatz ermöglicht dem Framework zudem die Kopplung von Automatisierungsressourcen, die technisch nicht den vollen Frameworkumfang abbilden können, über variablenfähige Kommunikationsprotokolle als *gekoppelte* Automatisierungsressourcen.

Die Datenorientierung des ISCI erlaubt eine datenbasierte Untersuchung von Verhalten und Eigenschaften. Die Untersuchungen können auch bezogen auf eine einzelne Automatisierungsressource erfolgen. Das Framework implementiert solche Untersuchungsansätze, um die praktische Evaluation eines Automatisierungssystems zu ermöglichen: Auf Automa-

tisierungsressourcen werden die Zustände der lokalen Datenstruktur über Datenabbilder erfasst und global dokumentiert. Es entsteht ein kontinuierlich wachsender Datenzwilling des Automatisierungssystems, über den Verhalten und Fähigkeiten ermittelt werden können, ohne zusätzliche Modelle oder Prüfumgebungen anwenden zu müssen. Das ISCI implementiert zudem eine Möglichkeit, die Modulausführung auf der Automatisierungsressource deterministisch zu gestalten.

Das Konzept und die Umsetzung des ISCI berücksichtigen etablierte Standards bei Technologien und technologiebezogenen Paradigmen. Es wird eine durchgehende Dateiorientierung im Sinne des *Everything-is-a-file*-Ansatzes verfolgt: Modelle, Softwaremodule, Beschreibungen, Konfigurationen und Datenstruktur sind dateibasiert. In der Datenstruktur sind Dateneinträge eines konfigurierten Automatisierungssystems auf den eingebetteten Automatisierungsressourcen über ein lokales Dateisystem verfügbar, gleich dem Zugriff auf übliche Hardware auf Unixsystemen. Das Framework macht keine Vorgabe hinsichtlich Kommunikationsprotokollen. Es können mehrere Kommunikationsprotokolle in einem Automatisierungssystem arbeiten. Die Kommunikationsimplementierung wird über Module vorgenommen, die auf Basis der Datenstruktur arbeiten. Die Datenstruktur ist somit die Schnittstelle zum Automatisierungssystem. Vorgaben an Kommunikationsmodule existieren nur über die durch das Framework vorgegebenen allgemeinen Datenformate nach IEC 62541 (OPC UA), die lokal zu nutzen sind, und über die Datenmodelle der Anwendung. Dadurch entsteht große Technologieunabhängigkeit bei der Kommunikationsumsetzung. Alle Modelle, Objekte und Beschreibungen sind so klein und einfach wie möglich ausgelegt. Sie werden JSON-basiert und menschenlesbar abgelegt und sind, ohne Klassen zu kennen, verständlich. ISCI-Automatisierungssysteme können durch die Modellkaskadierung und -aggregation eine große Modellkomplexität entwickeln, für die in der Arbeit kein Lösungsansatz entwickelt wurde.

Das entwickelte Framework wurde funktional und in einem Szenario validiert und seine konzeptuelle Entwicklung durch dauernde Implementierungen in einer objektorientierten Hochsprache begleitet. Die wesentlichen logischen und technischen Artefakte, Strukturen der eingebetteten Automatisierungsressourcen, Handler, Abbildungs-, EA-, Analyse-,

Kommunikations- und Steuerungsmodule und praktische Anwendungsmodelle werden implementiert und erfolgreich auf x64- und ARM-Umgebungen getestet. Das ISCI wird mit bestehenden Entwicklungsansätzen verglichen und eine qualitative Bewertung vorgenommen, bei dem sich das Framework bei Modellierungselementen, der Integration zum verteilten System, den Determinismus- und Abbildungsfunktionen hervorhebt. Dem Framework fehlt allerdings eine Standardisierung sowie eine Unterstützung mit Entwicklungselementen. Das ISCI beschreibt Artefakte und Technologien, verfügt aber im Gegensatz zu IEC-61131-Industriesystemen oder anderen Ansätzen über keine Entwicklungswerkzeuge, -methoden und -vorgehen. Um in der Praxis angewandt werden zu können, muss das Framework um diese Elemente erweitert werden. Zudem muss eine einfache und universale Methode für die Zeitsynchronisation, auf die ein ISCI-Automatisierungssystem angewiesen ist, entwickelt werden.

Ausblick

Aufgrund des standardbasierten technischen Aufbaus und der Konzentration auf Standardsysteme mit verbreiteten Architekturen und Betriebssystemen erscheint eine Untersuchung zur tieferen Betriebssystemintegration von ISCI-Funktionen für Prozesssteuerung und Bildung von verteilten, deterministischen Systemen naheliegend und sollte entsprechend in weiteren Arbeiten betrachtet werden.

Die textbasierten Modellstrukturen im ISCI könnten einen Ausgangspunkt für die Anwendung von generativen KI-Verfahren zur automatisierten Modellerzeugung für Automatisierungssysteme bereitstellen.

In der Arbeit wird die Integration des ISCI mit der Verwaltungsschale für das Referenzarchitekturmodells Industrie 4.0 konzeptionell betrachtet. Das RAMI4.0 ist ein bereits standardisierter Ansatz zur neuen Abstraktion industrieller Systeme. Die VWS als Bestandteil des RAMI4.0 ist eine abbildende und verwaltende Softwarestruktur um Produktionsassets. Aufgrund der zum Zeitpunkt der Arbeit starken allgemeinen Unterstützung des RAMI4.0 und der VWS sowie der bereits erfolgten Standardisierung, ist die VWS als Repräsentations- und Abbildungsarchitektur bedeutend. Mit zunehmender Verbreitung und Verfügbarkeit fachlich belastbarer Implementierungen sollte die Kartierung zur programmatischen Integration von Framework und VWS detailliert ausgebaut werden.

Literatur

AUTOSAR 2021
AUTOSAR, 2021.
Adaptive Platform R21-11: Adaptive Platform Release Overview.
AP-R21-11.
URL: https://www.autosar.org/fileadmin/standards/R21-11/AP/AUTOSAR_TR_AdaptivePlatformReleaseOverview.pdf
Zugriff am: 20.02.2022

Aggarwal et al. 2021
Aggarwal, Ishan; Anirudh, A.; Buddala, Raviteja, 2021.
Literature Review: ERP Implementation in Various Industries.
In: *2021 Innovations in Power and Advanced Computing Technologies (i-PACT)*.
Kuala Lumpur, Malaysia, 27.11.2021–29.11.2021, S. 1–6.
DOI: 10.1109/i-PACT52855.2021.9696962

Albers & Lohmeyer 2012
Albers, Albert; Lohmeyer, Quentin, 2012.
Advanced Systems Engineering - Towards a Model-based and Human-centered Methodology.
In: *Proceedings of TMCE 2012*.
Karlsruhe, Deutschland, 07.05.2012–11.05.2012.
ISBN: 978-90-5155-082-5

Alkhabbas et al. 2020
Alkhabbas, Fahed; Spalazzese, Romina; Cerioli, Maura; Leotta, Maurizio; Reggio, Gianna, 2020.
On the Deployment of IoT Systems: An Industrial Survey.
In: *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*.
Salvador, Brasilien, 16.03.2020–20.03.2020, S. 17–24.
DOI: 10.1109/ICSA-C50368.2020.00012

Amemiya & Nakao 2020
Amemiya, Koichiro; Nakao, Akihiro, 2020.
Layer-Integrated Edge Distributed Data Store for Real-time and Stateful Services.
In: *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*.
Budapest, Ungarn, 20.04.2020–24.04.2020, S. 1–9.
DOI: 10.1109/NOMS47738.2020.9110436

Arcangeli et al. 2015
Arcangeli, Jean-Paul; Boujbel, Raja; Leriche, Sébastien, 2015.
Automatic deployment of distributed software systems: Definitions and state of the art.
Journal of Systems and Software **103**, S. 198–218.
DOI: 10.1016/j.jss.2015.01.040

Arica & Powell 2017 Arica, Emrah; Powell, Daryl J., 2017.
Status and future of manufacturing execution systems.
In: *2017 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*.
Singapur, 10.12.2017–13.12.2017, S. 2000–2004

Ashiwal & Zoitl 2021 Ashiwal, Virendra; Zoitl, Alois, 2021.
Messaging Interaction Patterns for a Service Bus Concept of PLC-Software.
In: *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*.
Västerås, Schweden, S. 1–8.
DOI: 10.1109/ETFA45728.2021.9613638

AUTOSAR 2017a AUTOSAR, 2017.
Specification of RTE Software.
AUTOSAR Classic Platform Release 4.3.1.
URL: https://www.autosar.org/fileadmin/standards/R4.3.1/CP/AUTOSAR_SWS_RTE.pdf
Zugriff am: 20.02.2022

AUTOSAR 2017b AUTOSAR, 2017.
Virtual Functional Bus.
AUTOSAR Classic Platform Release 4.3.1.
URL: https://www.autosar.org/fileadmin/standards/R4.3.1/CP/AUTOSAR_EXP_VFB.pdf
Zugriff am: 20.02.2022

Azangoo et al. 2020 Azangoo, Mohammad; Taherkordi, Amir; Olaf Blech, Jan, 2020.
Digital Twins for Manufacturing Using UML and Behavioral Specifications.
In: *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*.
Wien, Österreich, 08.09.2020–11.09.2020, S. 1035–1038.
DOI: 10.1109/ETFA46521.2020.9212165

Bahtat et al. 2023 Bahtat, Chaymae; El Barkany, Abdellah; Jabri, Abdelouahhab, 2023.
Reconfigurable Manufacturing Systems: From Automation Through Industry 4.0.
International Journal of Industrial Engineering & Production Research **34**, S. 1–22.
DOI: 10.22068/ijiepr.34.2.2

Banerjee & Großmann 2017 Banerjee, Suprateek; Großmann, Daniel, 2017.
Aggregation of information models - An OPC UA based approach to a holistic model of models.
In: *2017 4th International Conference on Industrial Engineering and Applications (ICIEA)*.
Nagoya, Japan, 21.04.2017–23.04.2017, S. 296–299.
ISBN: 978-1-5090-6774-9

Bangemann et al. 2014 Bangemann, Thomas; Karnouskos, Stamatis; Camp, Roberto; Carlsson, Oscar; Riedl, Matthias; McLeod, Stuart; Harrison, Robert; Colombo, Armando W.; Stluka, Petr, 2014. State of the Art in Industrial Automation. In: Colombo, Armando W.; Bangemann, Thomas; Karnouskos, Stamatis; Delsing, Jerker; Stluka, Petr; Harrison, Robert; Jammes, Francois; Lastra, Jose L. (Hrsg.): *Industrial Cloud-Based Cyber-Physical Systems*. Cham: Springer International Publishing, S. 23–47. ISBN: 978-3-319-05623-4

Bangemann et al. 2016 Bangemann, Thomas; Riedl, Matthias; Thron, Mario; Diedrich, Christian, 2016. Integration of Classical Components Into Industrial Cyber–Physical Systems. *Proceedings of the IEEE* **104** (5), S. 947–959. DOI: 10.1109/JPROC.2015.2510981

Benra & Halang 2009 Benra, Juliane T.; Halang, Wolfgang A., 2009. *Software-Entwicklung für Echtzeitsysteme*. Berlin, Heidelberg: Springer. ISBN: 978-3-642-01595-3

Berardinelli et al. 2016 Berardinelli, Luca; Biffl, Stefan; Lüder, Arndt; Mätzler, Emanuel; Mayerhofer, Tanja; Wimmer, Manuel; Sint, Sabine, 2016. Cross-disciplinary engineering with AutomationML and SysML. *at - Automatisierungstechnik* **64**. DOI: 10.1515/auto-2015-0076

Binder et al. 2023 Binder, Christoph; Calà, Ambra; Vollmar, Jan; Neureiter, Christian; Lüder, Arndt, 2023. Model-driven Engineering of flexible Production Systems with the RAMI Toolbox. In: *2023 IEEE 21st International Conference on Industrial Informatics (INDIN)*. Lemgo, Deutschland, 18.07.2023–20.07.2023, S. 1–8. DOI: 10.1109/INDIN51400.2023.10217882

Bloch et al. 2016 Bloch, Henry; Fay, Alexander; Hoernicke, Mario, 2016. Analysis of service-oriented architecture approaches suitable for modular process automation. In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. Berlin, Deutschland, 06.09.2016–09.09.2016, S. 1–8. DOI: 10.1109/ETFA.2016.7733651

Boelmann et al. 2016 Boelmann, Christopher; Schwittmann, Lorenz; Waltereit, Marian; Wander, Matthäus; Weis, Torben, 2016. Application-Level Determinism in Distributed Systems. In: *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*. Wuhan, China, 13.12.2016–16.12.2016, S. 989–998. DOI: 10.1109/ICPADS.2016.0132

Bohn et al. 2006 Bohn, Hendrik; Bobek, Andreas; Golatowski, Frank, 2006. SIRENA - Service Infrastructure for Real-time Embedded Networked Devices: A service oriented framework for different domains. In: *International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies*. Morne, Mauritius, 23.04.2006–29.04.2006, S. 43–43. ISBN: 978-0-7695-2552-5

Brand et al. 2021 Brand, Max; Mayer, Albrecht; Slomka, Frank, 2021. A Matter of Overhead - Response Time Analysis of Hard Real-Time Systems in Theory and Practice. In: *MBMV 2021 - 24. Workshop MBMV*. online, 18.03.2021–19.03.2021, S. 1–7. ISBN: 978-3-8007-5500-4

Brecher et al. 2017 Brecher, Christian; Ecker, Christian; Herfs, Werner; Obdenbusch, Markus; Jeschke, Sabina; Hoffmann, Max; Meisen, Tobias, 2017. The Need of Dynamic and Adaptive Data Models for Cyber-Physical Production Systems. In: Song, Houbing; Rawat, Danda B.; Jeschke, Sabina; Brecher, Christian (Hrsg.): *Cyber-Physical Systems*. Boston: Academic Press, S. 321–338. ISBN: 978-0-12-803801-7

Brecher et al. 2019 Brecher, Christian; Buchsbaum, Melanie; Storms, Simon, 2019. Control from the Cloud: Edge Computing, Services and Digital Shadow for Automation Technologies. In: *2019 International Conference on Robotics and Automation (ICRA)*. Montreal, Kanada, 20.05.2019–24.05.2019, S. 9327–9333. DOI: 10.1109/ICRA.2019.8793488

Breunig & Götz 2018 Breunig, David A.; Götz, Benjamin, 2018. Simultaneously acting network layers in an IEC 61499 modeling system at the example of Eclipse-4DIAC, the cloud-oriented MSB and XML-RPC. *Procedia CIRP* **72**, S. 928–933. DOI: 10.1016/j.procir.2018.03.143

Breunig & Schneider 2019 Breunig, David A.; Schneider, Matthias, 2019.
Multi-protocol Data Aggregation and Acquisition for Distributed Control Systems.
Procedia CIRP **81**, S. 310–315.
DOI: 10.1016/j.procir.2019.03.054

Breunig et al. 2020 Breunig, David A.; Stock, Daniel; Bauernhansl, Thomas, 2020.
Requirements and Concept for a Modular and State-Oriented Control Device Architecture.
Procedia Manufacturing **42**, S. 281–287.
International Conference on Industry 4.0 and Smart Manufacturing (ISM 2019).
DOI: 10.1016/j.promfg.2020.02.096

Bukeikhanov et al. 2020 Bukeikhanov, N.; Nikishechkin, A.; Gvozdkova, S., 2020.
Using Petri Nets for Simulation, Analysis, and Synthesis of Automated Systems.
In: *2020 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon)*.
Wladiwostok, Russland, 06.10.2020–09.10.2020, S. 1–5.
DOI: 10.1109/FarEastCon50210.2020.9271457

Cavalieri & Salafia 2020 Cavalieri, Salvatore; Salafia, Marco Giuseppe, 2020.
Asset Administration Shell for PLC Representation Based on IEC 61131–3.
IEEE Access **8**, S. 142606–142621.
DOI: 10.1109/ACCESS.2020.3013890

Cheng & Zhang 2011 Cheng, Xiao-hui; Zhang, Liang, 2011.
A research of inter-process communication based on shared memory and address-mapping.
In: *Proceedings of 2011 International Conference on Computer Science and Network Technology*.
Harbin, China, 24.12.2011–26.12.2021, S. 111–114.
DOI: 10.1109/ICCSNT.2011.6181920

Chobanov & Hardalov 2022 Chobanov, Vesselin; Hardalov, Ivailo, 2022.
The cost of man and machine labor in 21-st century.
In: *2022 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*.
Istanbul, Türkei, 23.05.2022–24.05.2022, S. 1–5.
DOI: 10.1109/HORA55278.2022.9799915

Chryssolouris 2006 Chryssolouris, George, 2006.
Manufacturing Systems: Theory and Practice. New York, NY: Springer Science+Business Media Inc. Mechanical Engineering Series.
ISBN: 9780387256832

Colombo et al. 2017 Colombo, Armando W.; Karnouskos, Stamatis; Kaynak, Okyay; Shi, Yang; Yin, Shen, 2017.
Industrial Cyberphysical Systems: A Backbone of the Fourth Industrial Revolution.
IEEE Industrial Electronics Magazine **11** (1), S. 6–16.
DOI: 10.1109/MIE.2017.2648857

Corbet et al. 2005 Corbet, Jonathan; Rubini, Alessandro; Kroah-Hartman, Greg, 2005.
Linux Device Drivers. 3rd Edition. Sebastopol, Kanada: O'Reilly Media, Inc.
ISBN: 978-0-5960-0590-0

Cucinotta et al. 2009 Cucinotta, Tommaso; Mancina, Antonio; Anastasi, Gaetano F.; Lipari, Giuseppe; Mangeruca, Leonardo; Checcozzo, Roberto; Rusina, Fulvio, 2009.
A Real-Time Service-Oriented Architecture for Industrial Automation.
IEEE Transactions on Industrial Informatics **5** (3), S. 267–277.
DOI: 10.1109/TII.2009.2027013

Dafflon et al. 2021 Dafflon, Baudouin; Moalla, Nejib; Ouzrout, Yacine, 2021.
The challenges, approaches, and used techniques of CPS for manufacturing in Industry 4.0: a literature review.
The International Journal of Advanced Manufacturing Technology **113** (7), S. 2395–2412.
DOI: 10.1007/s00170-020-06572-4

Dai et al. 2015 Dai, Wenbin; Vyatkin, Valeriy; Christensen, James H.; Dubinin, Victor N., 2015.
Bridging Service-Oriented Architecture and IEC 61499 for Flexibility and Interoperability.
IEEE Transactions on Industrial Informatics **11** (3).
DOI: 10.1109/TII.2015.2423495

Dai et al. 2014 Dai, Wenbin W.; Vyatkin, Valeriy; Christensen, James H., 2014.
The application of service-oriented architectures in distributed automation systems.
In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*.
Hongkong, China, 31.05.2014–05.06.2014, S. 252–257.
ISBN: 978-1-4799-3685-4

De Jong 1999 De Jong, Edwin, 1999.
Distributed shared data space architecture for real-time systems.
In: *2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)*.
Saint-Malo, Frankreich, 02.05.1999–05.05.1999, S. 159–162.
DOI: 10.1109/ISORC.1999.776370

Delamer & Lastra 2007 Delamer, Ivan M.; Lastra, Jose L. Martinez, 2007.
Loosely-coupled Automation Systems using Device-level SOA.
In: *2007 5th IEEE International Conference on Industrial Informatics*.
Wien, Österreich, 23.06.2007–27.06.2007, S. 743–748.
ISBN: 978-1-4244-0850-4

DIN 61131-1 DIN 61131-1:2004-03.
Speicherprogrammierbare Steuerungen - Teil 1: Allgemeine Informationen (IEC 61131-1:2003)

DIN 61131-10 DIN 61131-10:2020-11.
Speicherprogrammierbare Steuerungen - Teil 10: XML-basiertes Austauschformat für Programme nach IEC 61131-3 (IEC 61131-10:2019)

DIN 61131-3 DIN 61131-3:2014-06.
Speicherprogrammierbare Steuerungen - Teil 3: Programmiersprachen (IEC 61131-3:2013)

DIN 61158-1 DIN 61158-1:2020-04.
Industrielle Kommunikationsnetze - Feldbusse - Teil 1: Überblick und Leitfaden zu den Normen der Reihen IEC 61158 und IEC 61784 (IEC 61158-1:2019)

DIN 61158-3 DIN 61158-3:2022-02.
Industrielle Kommunikationsnetze - Profile - Teil 3: Funktional sichere Übertragung bei Feldbussen - Allgemeine Regeln und Festlegungen für Profile (IEC 61784-3:2021)

DIN 61784-2 DIN 61784-2:2020-07.
Industrielle Kommunikationsnetze - Profile - Teil 2: Zusätzliche Feldbusprofile für Echtzeitnetzwerke basierend auf ISO/IEC 88023 (IEC 61784-2:2019)

DIN 62264-1 DIN 62264-1:2014-07.
Integration von Unternehmensführungs- und Leitsystemen – Teil 1: Modelle und Terminologie (IEC 62264-1:2013)

DIN 62264-5 DIN 62264-5:2017-05.
Integration von Unternehmensführungs und Leitsystemen – Teil 5: Transaktionen zwischen Unternehmungsführungs und Produktionsleitsystemen (IEC 622645:2016)

DIN 62714-1 DIN 62714-1:2019-02.
Datenaustauschformat für Planungsdaten industrieller Automatisierungssysteme - Automation markup language - Teil 1: Architektur und allgemeine Festlegungen (IEC 62714-1:2018)

DIN 62948 DIN 62948:2018-08.
Industrielle Kommunikationsnetze - Funkkommunikationsnetze und Kommunikationsprofile - WIAFA

DIN 66253
DIN 66253:2018-03.
Informationsverarbeitung - Programmiersprache PEARL - SafePEARL

DIN EN 61499-1
DIN EN 61499-1:2014-09.
Funktionsbausteine für industrielle Leitsysteme - Teil 1: Architektur (IEC 61499-1:2012)

DIN 61804-2
DIN EN 61804-2:2018.
Funktionsbausteine für die Prozessautomation und elektronische Gerätebeschreibungssprache – Teil 2: Festlegung des Funktionsbausteinkonzepts (IEC 61804-2: 2018)

DIN EN 61804-4
DIN EN 61804-4:2016-08.
Funktionsbausteine für die Prozessautomation und elektronische Gerätebeschreibungssprache – Teil 4: Interpretation von Gerätebeschreibungen (IEC 61804-4:2015)

DIN EN 62453-1
DIN EN 62453-1:2017-11.
Field Device Tool (FDT)Schnittstellenspezifikation – Teil 1: Überblick und Leitfaden (IEC 62453-1: 2016)

DIN EN 62769-1
DIN EN 62769-1:2016-03.
Feldgeräteintegration (FDI) – Teil 1: Überblick (IEC 62769-1:2015)

DIN EN 62769-4
DIN EN 62769-4:2015-06.
Feldgeräteintegration (FDI) – Teil 4: FDI-Packages (IEC 62769-4:2015)

DIN EN 62769-5
DIN EN 62769-5:2022-09.
Feldgeräteintegration (FDI) – Teil 5: FDI-Informationsmodell (IEC 65E/592/CDV:2018)

DIN SPEC 91345
DIN SPEC 91345:2016-04.
Referenzarchitekturmodell Industrie 4.0 (RAMI4.0)

Drath et al. 2021
Drath, Rainer; John, Michael; Lüder, Arndt; Pauly, Johanna; Prinz, Josef; Rentschler, Markus; Riedl, Matthias; Schleipen, Miriam; Schmidt, Nicole; Stecken, Jannis; Strahilov, Anton; Wally, Bernhard, 2021.
AutomationML - Standardized data exchange in the engineering process of production systems. AutomationML e. V. c/o IAF.
URL: https://www.automationml.org/wp-content/uploads/2021/06/AutomationML-Brochure.pdf
Zugriff am: 20.02.2022

Dupont et al. 2017
Dupont, Corentin; Giaffreda, Raffaele; Capra, Luca, 2017.
Edge computing in IoT context: Horizontal and vertical Linux container migration.
In: *2017 Global Internet of Things Summit (GIoTS)*.
Genf, Schweiz, 06.06.2017–09.06.2017, S. 1–4.
ISBN: 978-1-5090-5873-0

Ebert & Hochstein 2023 Ebert, Christof; Hochstein, Lorin, 2023.
DevOps in Practice.
IEEE Software **40** (1), S. 29–36.
DOI: 10.1109/MS.2022.3213285

Eckert 2015 Eckert, Karin, 2015.
Funktionaler Anwendungsentwurf verteilter Automatisierungssysteme: Anwendung von Entwurfsmustern in der Automatisierungstechnik.
Hamburg, Helmut-Schmidt-Universität Hamburg, Doktorarbeit.
DOI: 10.24405/526

Ejersbo et al. 2023 Ejersbo, Henrik; Lausdahl, Kenneth; Frasheri, Mirgita; Esterle, Lukas, 2023.
Dynamic Runtime Integration of New Models in Digital Twins.
In: *2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*.
Melbourne, Australien, 15.05.2023–16.05.2023, S. 44–55.
DOI: 10.1109/SEAMS59076.2023.00016

Ellwein et al. 2023 Ellwein, Carsten; Neumann, Rebekka; Verl, Alexander, 2023.
Software-defined Manufacturing: Data Representation.
Procedia CIRP **118**, S. 360–365.
DOI: 10.1016/j.procir.2023.06.062

ElMaraghy et al. 2021 ElMaraghy, Hoda; Monostori, Laszlo; Schuh, Guenther; ElMaraghy, Waguih, 2021.
Evolution and future of manufacturing systems.
CIRP Annals **70** (2), S. 635–658.
DOI: 10.1016/j.cirp.2021.05.008

ElMaraghy & Wiendahl 2009 ElMaraghy, Hoda A.; Wiendahl, Hans-Peter, 2009.
Changeability - An Introduction.
In: ElMaraghy, Hoda A. (Hrsg.): *Changeable and Reconfigurable Manufacturing Systems*.
London: Springer, S. 3–24.
ISBN: 978-1-84882-067-8

Eymüller et al. 2020 Eymüller, Christian; Hanke, Julian; Hoffmann, Alwin; Kugelmann, Markus; Reif, Wolfgang, 2020.
Real-time capable OPC-UA Programs over TSN for distributed industrial control.
In: *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*.
Wien, Österreich, 08.09.2020–11.09.2020, S. 278–285.
DOI: 10.1109/ETFA46521.2020.9212171

Fahmideh et al. 2021 Fahmideh, Mahdi; Abbasi, Aakash Ahmad; Behnaz, Ali; Grundy, John; Susilo, Willy, 2021.
Software Engineering for Internet of Things.
IEEE Transactions on Software Engineering, S. 1–1.
DOI: 10.1109/TSE.2021.3070692

Feichtinger et al. 2022 Feichtinger, Kevin; Meixner, Kristof; Rinker, Felix; Koren, István; Eichelberger, Holger; Heinemann, Tonja; Holtmann, Jörg; Konersmann, Marco; Michael, Judith; Neumann, Eva-Maria; Pfeiffer, Jérôme; Rabiser, Rick; Riebisch, Matthias; Schmid, Klaus, 2022.
Industry Voices on Software Engineering Challenges in Cyber-Physical Production Systems Engineering.
In: *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*.
Stuttgart, Deutschland, 06.09.2022–09.09.2022, S. 1–8.
DOI: 10.1109/ETFA52439.2022.9921568

Feldhusen & Grote 2013 Feldhusen, Jörg; Grote, Karl-Heinrich, 2013.
Pahl/Beitz Konstruktionslehre: Methoden und Anwendung erfolgreicher Produktentwicklung. Springer.
ISBN: 9783642295690

Felter et al. 2015 Felter, Wes; Ferreira, Alexandre; Rajamony, Ram; Rubio, Juan, 2015.
An updated performance comparison of virtual machines and Linux containers.
In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.
Philadelphia, Pennsylvania, USA, 29.03.2015–31.03.2015, S. 171–172.
DOI: 10.1109/ISPASS.2015.7095802

Fernández et al. 2021 Fernández, Igor Azkarate; Cortabarría, Juan Carlos Mugarza; Echeverría, Luka Eciolaza, 2021.
Petri net implementation in programmable logic controllers: methodology for development and validation.
In: *2021 IEEE 19th World Symposium on Applied Machine Intelligence and Informatics (SAMI)*.
online, 21.01.2021–23.01.2021, S. 000015–000020.
DOI: 10.1109/SAMI50585.2021.9378673

Fischer et al. 2020 Fischer, Juliane; Vogel-Heuser, Birgit; Haben, Fabian; Schaefer, Ina, 2020.
Reengineering Workflow for Planned Reuse of IEC 61131-3 Legacy Software.
In: *2020 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*.
Singapur, 14.12.2020–17.12.2020, S. 1126–1130.
DOI: 10.1109/IEEM45057.2020.9309857

Fürst & Bechter 2016 Fürst, Simon; Bechter, Markus, 2016.
AUTOSAR for Connected and Autonomous Vehicles: The AUTOSAR Adaptive Platform.
In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*.
Toulouse, Frankreich, 28.06.2016–01.07.2016, S. 215–217.
ISBN: 978-1-5090-3688-2

Gampig et al. 2021 Gampig, Jonas; Terzimehic□, Tarik; Dorofeev, Kirill, 2021.
PLC Integration into Industry 4.0 Middleware: Function Block Library for the Interaction with REST and OPC UA Asset Administration Shells.
In: *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*.
online, 07.09.2021–10.09.2021, S. 1–8.
DOI: 10.1109/ETFA45728.2021.9613267

Gangoiti et al. 2021 Gangoiti, Unai; López, Alejandro; Armentia, Aintzane; Estévez, Elisabet; Marcos, Marga, 2021.
Model-Driven Design and Development of Flexible Automated Production Control Configurations for Industry 4.0.
Applied Sciences **11** (5).
DOI: 10.3390/app11052319

Gemlau et al. 2021 Gemlau, Kai-Björn; Köhler, Leonie; Ernst, Rolf, 2021.
Efficient Run-Time Environments for System-Level LET Programming.
In: *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*.
online, 01.02.2021–05.02.2021, S. 749–754.
DOI: 10.23919/DATE51398.2021.9474257

Givehchi et al. 2017 Givehchi, Omid; Landsdorf, Klaus; Simoens, Pieter; Colombo, Armando Walter, 2017.
Interoperability for Industrial Cyber-Physical Systems: An Approach for Legacy Systems.
IEEE Transactions on Industrial Informatics **13** (6), S. 3370–3378.
DOI: 10.1109/TII.2017.2740434

Goldschmidt et al. 2015 Goldschmidt, Thomas; Murugaiah, Mahesh Kumar; Sonntag, Christian; Schlich, Bastian; Biallas, Sebastian; Weber, Peter, 2015.
Cloud-Based Control: A Multi-tenant, Horizontally Scalable Soft-PLC.
In: *2015 IEEE 8th International Conference on Cloud Computing*.
New York City, New York, USA, 27.06.2015–02.07.2015, S. 909–916.
DOI: 10.1109/CLOUD.2015.124

Goldschmidt et al. 2018 Goldschmidt, Thomas; Hauck-Stattelmann, Stefan; Malakuti, Somayeh; Grüner, Sten, 2018.
Container-based Architecture for Flexible Industrial Control Applications.
Journal of Systems Architecture **84**.
DOI: 10.1016/j.sysarc.2018.03.002

Gomes & Baunach 2018 Gomes, Renata Martins; Baunach, Marcel, 2018.
A Model-Based Concept for RTOS Portability.
In: *2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA)*.
Akaba, Jordanien, 28.10.2018–01.11.2018, S. 1–6.
DOI: 10.1109/AICCSA.2018.8612862

Göppert et al. 2021 Göppert, Amon; Grahn, Lea; Rachner, Jonas; Grunert, Dennis; Hort, Simon; Schmitt, Robert H., 2021.
Pipeline for ontology-based modeling and automated deployment of digital twins for planning and control of manufacturing systems.
Journal of Intelligent Manufacturing.
DOI: 10.1007/s10845-021-01860-6

Gore et al. 2020 Gore, Rahul Nandkumar; Lisova, Elena; Åkerberg, Johan; Björkman, Mats, 2020.
Clock Synchronization in Future Industrial Networks: Applications, Challenges, and Directions.
In: *2020 AEIT International Annual Conference (AEIT)*.
Catania, Italien, 23.09.2020–25.09.2019, S. 1–6.
DOI: 10.23919/AEIT50178.2020.9241154

Gräßler et al. 2016 Gräßler, Iris; Pöhler, Alexander; Pottebaum, Jens, 2016.
Creation of a Learning Factory for Cyber Physical Production Systems.
Procedia CIRP **54**, S. 107–112.
DOI: 10.1016/j.procir.2016.05.063

Griffor et al. 2017 Griffor, Edward R.; Greer, Chris; Wollman, David A.; Burns, Martin J., 2017.
Framework for cyber-physical systems: volume 2, working group reports. Gaithersburg, MD, USA: National Institute of Standards und Technology. NIST SP 1500-202.
DOI: 10.6028/NIST.SP.1500-202

Großmann et al. 2014 Großmann, Daniel; Bregulla, Markus; Banerjee, Suprateek; Schulz, Dirk; Braun, Roland, 2014.
OPC UA server aggregation: The foundation for an internet of portals.
In: *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*.
Barcelona, Spanien, 16.09.2014–19.09.2014, S. 1–6.
ISBN: 978-1-4799-4845-1

Grüner et al. 2018 Grüner, Sten; Malakuti, Somayeh; Schmitt, Johannes; Terzimehic, Tarik; Wenger, Monika; Elfaham, Haitham, 2018. Alternatives for Flexible Deployment Architectures in Industrial Automation Systems.
In: *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*.
Turin, Italien, 04.09.2018–07.09.2018, S. 35–42.
DOI: 10.1109/ETFA.2018.8502526

Gundall et al. 2020a Gundall, Michael; Reti, Daniel; Schotten, Hans D., 2020. Application of Virtualization Technologies in Novel Industrial Automation: Catalyst or Show-Stopper?
In: *2020 IEEE 18th International Conference on Industrial Informatics (INDIN)*.
online, 20.07.2020–23.07.2020, S. 283–290.
DOI: 10.1109/INDIN45582.2020.9442123

Gundall et al. 2020b Gundall, Michael; Glas, Calvin; Schotten, Hans D., 2020. Introduction of an Architecture for Flexible Future Process Control Systems as Enabler for Industry 4.0.
In: *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*.
Wien, Österreich, 08.09.2020–11.09.2020, S. 1047–1050.
DOI: 10.1109/ETFA46521.2020.9212031

Gundall et al. 2021 Gundall, Michael; Glas, Calvin; Schotten, Hans D., 2021. Feasibility Study on Virtual Process Controllers as Basis for Future Industrial Automation Systems.
In: *2021 22nd IEEE International Conference on Industrial Technology (ICIT)*.
online, 10.03.2021–12.03.2021, S. 1080–1087.
DOI: 10.1109/ICIT46573.2021.9453651

Hamzah et al. 2023 Hamzah, Muzaffar; Islam, Md. Monirul; Hassan, Shahriar; Akhtar, Md. Nasim; Ferdous, Most. Jannatul; Jasser, Muhammed Basheer; Mohamed, Ali Wagdy, 2023. Distributed Control of Cyber Physical System on Various Domains: A Critical Review.
Systems **11** (4).
DOI: 10.3390/systems11040208

Hästbacka et al. 2022 Hästbacka, David; Halme, Jari; Barna, Laurentiu; Hoikka, Henrikki; Pettinen, Henri; Larrañaga, Martin; Björkbom, Mikael; Mesiä, Heikki; Jaatinen, Antti; Elo, Marko, 2022.
Dynamic Edge and Cloud Service Integration for Industrial IoT and Production Monitoring Applications of Industrial Cyber-Physical Systems.
IEEE Transactions on Industrial Informatics **18** (1), S. 498–508.
DOI: 10.1109/TII.2021.3071509

Heidel et al. 2017 Heidel, Roland; Hoffmeister, Michael; Hankel, Martin; Döbrich, Udo, 2017.
Industrie4.0 Basiswissen RAMI4.0 - Referenzarchitekturmodell mit Industrie4.0-Komponente. 1. Auflage.
Für Normung e. V., DIN Deutsches Institut (Hrsg.): Berlin: Beuth Verlag GmbH, VDE Verlag GmbH

Helo et al. 2014 Helo, Petri; Suorsa, Mikko; Hao, Yuqiuge; Anussornnitisarn, Pornthep, 2014.
Toward a cloud-based manufacturing execution system for distributed manufacturing.
Computers in Industry **65** (4), S. 646–656.
DOI: 10.1016/j.compind.2014.01.015

Herterich et al. 2015 Herterich, Matthias M.; Uebernickel, Falk; Brenner, Walter, 2015.
The Impact of Cyber-physical Systems on Industrial Services in Manufacturing.
Procedia CIRP **30**, S. 323–328.
DOI: 10.1016/j.procir.2015.02.110

Hoffmann et al. 2021 Hoffmann, Martin; Malakuti, Somayeh; Grüner, Sten; Finster, Soeren; Gebhardt, Jörg; Tan, Ruomu; Schindler, Thorsten; Gamer, Thomas, 2021.
Developing Industrial CPS: A Multi-Disciplinary Challenge.
Sensors **21**, S. 1991.
DOI: 10.3390/s21061991

Hromkovic 2011 Hromkovic, Juraj, 2011.
Theoretische Informatik: formale Sprachen, Berechenbarkeit, Komplexitätstheorie, Algorithmik, Kommunikation und Kryptographie. 4., aktualisierte Aufl. Wiesbaden: Vieweg-+-Teubner. Leitfäden der Informatik.
OCLC: 688609618.
ISBN: 978-3-8348-0650-5

IEC PAS 63088 IEC PAS 63088:2017-03.
IEC PAS 63088:2017 Smart manufacturing – Reference architecture model industry 4.0 (RAMI4.0)

IEEE Std 61158 IEEE Std 61158-2017:05.
IEEE Standard for Industrial Hard Real-Time Communication

Ioana & Korodi 2019 Ioana, Alexandru; Korodi, Adrian, 2019.
VSOMEIP - OPC UA Gateway Solution for the Automotive Industry.
In: *2019 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC)*.
Valbonne, Frankreich, 17.06.2019–19.06.2019, S. 1–6.
DOI: 10.1109/ICE.2019.8792619

Ishihara et al. 2019 Ishihara, Yasunori; Kato, Hiroyuki; Nakano, Keisuke; Onizuka, Makoto; Sasaki, Yuya, 2019.
Toward BX-Based Architecture for Controlling and Sharing Distributed Data.
In: *2019 IEEE International Conference on Big Data and Smart Computing (BigComp)*.
Kyoto, Japan, 27.02.2019–02.03.2019, S. 1–5.
DOI: 10.1109/BIGCOMP.2019.8679145

ISO/IEC/IEEE 42010 ISO/IEC/IEEE 42010:2011-12.
System-und Software-Engineering - Architekturbeschreibung

Jacobson et al. 2019 Jacobson, Ivar; Lawson, Harold; Ng, Pan-Wei; McMahon, Paul E.; Goedicke, Michael, 2019.
The Essentials of Modern Software Engineering: Free the Practices from the Method Prisons! Association for Computing Machinery und Morgan & Claypool.
ISBN: 9781947487277

Jammes & Smit 2005 Jammes, François; Smit, Harm, 2005.
Service-Oriented Paradigms in Industrial Automation.
IEEE Transactions on Industrial Informatics **1** (1), S. 62–70.
DOI: 10.1109/TII.2005.844419

Jones et al. 2020 Jones, David; Snider, Chris; Nassehi, Aydin; Yon, Jason; Hicks, Ben, 2020.
Characterising the Digital Twin: A systematic literature review.
CIRP Journal of Manufacturing Science and Technology **29**, S. 36–52.
DOI: 10.1016/j.cirpj.2020.02.002

Juhlin et al. 2021 Juhlin, Prerna; Schlake, Jan-Christoph; Janka, Dennis; Hawlitschek, Adrian, 2021.
Metamodeling of Cyber-Physical Production Systems using AutomationML for Collaborative Innovation.
In: *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*.
Västerås, Schweden, S. 1–4.
DOI: 10.1109/ETFA45728.2021.9613560

Karnouskos et al. 2019 Karnouskos, Stamatis; Ribeiro, Luis; Leitão, Paulo; Lüder, Arndt; Vogel-Heuser, Birgit, 2019.
Key Directions for Industrial Agent Based Cyber-Physical Production Systems.
In: *2019 IEEE International Conference on Industrial Cyber Physical Systems (ICPS)*.
Taipeh, Taiwan, 06.05.2015–09.05.2015, S. 17–22.
DOI: 10.1109/ICPHYS.2019.8780360

Kegel 2010
Kegel, Günther, 2010.
Neue Informationsarchitektur in der industriellen Automatisierung.
In: *Karlsruher Leittechnisches Kolloquium 2010: Tagungsband*.
Karlsruhe, Deutschland, 09.06.2010–10.06.2010.
ISBN: 978-3-8396-0130-3

Koren et al. 2023
Koren, István; Rinker, Felix; Meixner, Kristof; Matevska, Jasminka; Walter, Jörg, 2023.
Challenges and Opportunities of DevOps in Cyber-Physical Production Systems Engineering.
In: *2023 IEEE 6th International Conference on Industrial Cyber-Physical Systems (ICPS)*.
Wuhan, China, 08.05.2023–11.05.2023, S. 1–6.
DOI: 10.1109/ICPS58381.2023.10128073

Körner et al. 2019
Körner, Marc-Fabian; Bauer, Dennis; Keller, Robert; Rösch, Martin; Schlereth, Andreas; Simon, Peter; Bauernhansl, Thomas; Fridgen, Gilbert; Reinhart, Gunther, 2019.
Extending the Automation Pyramid for Industrial Demand Response.
Procedia CIRP **81**, S. 998–1003.
DOI: 10.1016/j.procir.2019.03.241

Križanic et al. 2020
Križanic, Snježana; Sestanj-Peric, Tanja; Kutnjak, Ana, 2020.
ERP Solutions in Cloud Technologies as a Driver for Digital Transformation of Businesses.
In: *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*.
Opatija, Kroatien, 28.09.2020–02.10.2020, S. 1274–1279.
DOI: 10.23919/MIPRO48935.2020.9245170

Kruchten 1995
Kruchten, Philippe, 1995.
Architectural Blueprints—The "4+1" View Model of Software Architecture.
IEEE Software 12 (6) FIXEN, S. 15

Kweon et al. 1999
Kweon, Seok-Kyu; Shin, Kang G.; Zheng, Zin, 1999.
Statistical Real-Time Communication over Ethernet for Manufacturing Automation Systems.
In: *IEEE Real-Time Technology and Applications Symposium*.
Vancouver, Kanada, 02.06.1999–04.06.1999, S. 192–202.
DOI: 10.1109/RTTAS.1999.777672

Lam et al. 2022
Lam, An Ngoc; Haugen, Oystein; Delsing, Jerker, 2022.
Dynamical Orchestration and Configuration Services in Industrial IoT Systems: An Autonomic Approach.
IEEE Open Journal of the Industrial Electronics Society **3**.
DOI: 10.1109/OJIES.2022.3149093

Lamnabhi-Lagarrigue & Samad 2023
Lamnabhi-Lagarrigue, Francoise; Samad, Tariq, 2023.
Social, Organizational, and Individual Impacts of Automation.
In: Nof, Shimon Y. (Hrsg.): *Springer Handbook of Automation*.
Cham: Springer International Publishing, S. 61–75.
ISBN: 978-3-030-96729-1

Lee 2006
Lee, Edward A., 2006.
Position Paper for NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap. Cyber-Physical Systems - Are Computing Foundations Adequate?,
URL: http://chess.eecs.berkeley.edu/pubs/329.html

Lee 2008
Lee, Edward A., 2008.
Cyber Physical Systems: Design Challenges.
In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*.
Orlando, Florida, USA, 05.05.2008–07.05.2008, S. 363–369.
DOI: 10.1109/ISORC.2008.25

Lee & Seshia 2012
Lee, Edward Ashford; Seshia, Sanjit Arunkumar, 2012.
Introduction to embedded systems: a cyber physical systems approach. 1. ed., print. 1.08. Lulu: LeeSeshia.org.
OCLC: 915495730.
ISBN: 978-0-557-70857-4

Lehner et al. 2021
Lehner, Daniel; Sint, Sabine; Vierhauser, Michael; Narzt, Wolfgang; Wimmer, Manuel, 2021.
AML4DT: A Model-Driven Framework for Developing and Maintaining Digital Twins with AutomationML.
In: *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*.
Västerås, Schweden, S. 1–8.
DOI: 10.1109/ETFA45728.2021.9613376

Leitao et al. 2017
Leitao, Paulo; Karnouskos, Stamatis; Ribeiro, Luis; Moutis, Panayiotis; Barbosa, Jose; Strasser, Thomas. I., 2017.
Common practices for integrating industrial agents and low level automation functions.
In: *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*.
Beijing, China, 29.10.2017–01.11.2017, S. 6665–6670.
ISBN: 978-1-5386-1127-2

Leitão et al. 2021
Leitão, Paulo; Strasser, Thomas I.; Karnouskos, Stamatis; Ribeiro, Luis; Barbosa, José; Huang, Victor, 2021.
Recommendation of Best Practices for Industrial Agent Systems based on the IEEE 2660.1 Standard.
In: *2021 22nd IEEE International Conference on Industrial Technology (ICIT)*.
online, 10.03.2021–12.03.2021, S. 1157–1162.
DOI: 10.1109/ICIT46573.2021.9453511

Lesi et al. 2023 Lesi, Vuk; Jakovljevic, Zivana; Pajic, Miroslav, 2023.
IoT-Enabled Motion Control: Architectural Design Challenges and Solutions.
IEEE Transactions on Industrial Informatics **19** (3), S. 2284–2294.
DOI: 10.1109/TII.2022.3202175

Li et al. 2020 Li, Chen; Mantravadi, Soujanya; Møller, Charles, 2020.
AAU Open Source MES Architecture for Smart Factories – Exploiting ISA 95.
In: *2020 IEEE 18th International Conference on Industrial Informatics (INDIN)*.
online, 20.07.2020–23.07.2020, S. 369–373.
DOI: 10.1109/INDIN45582.2020.9442130

Liu et al. 2020 Liu, Jui-Hung; Lu, Yu-Sheng; Yu, Yan-Ting, 2020.
Common Management Technology of Automation Equipment in Industry 4.0.
In: *2020 IEEE Eurasia Conference on IOT, Communication and Engineering (ECICE)*.
Yunlin, Taiwan, 23.10.2020–25.10.2020, S. 355–358.
DOI: 10.1109/ECICE50847.2020.9302021

López et al. 2020 López, Alejandro; Estévez, Elisabet; Casquero, Oskar; Marcos, Marga, 2020.
Using industrial standards for modeling flexible manufacturing systems.
In: *2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS)*.
online, 10.06.2020–12.06.2020, S. 41–46.
DOI: 10.1109/ICPS48405.2020.9274785

Luo et al. 2021 Luo, Zhiyong; Huang, Diandian; Wang, Yao; He, Yuchen, 2021.
Deterministic Communication Based on TSN and OPC UA Embedded Server.
In: *2021 40th Chinese Control Conference (CCC)*.
Shanghai, China, 26.07.2021–28.07.2021, S. 6518–6523.
DOI: 10.23919/CCC52363.2021.9550163

Lyu & Brennan 2021 Lyu, Guolin; Brennan, Robert William, 2021.
Towards IEC 61499-Based Distributed Intelligent Automation: A Literature Review.
IEEE Transactions on Industrial Informatics **17** (4).
DOI: 10.1109/TII.2020.3016990

Lyu et al. 2019 Lyu, Minhu; Biennier, Frederique; Ghodous, Parisa, 2019.
Control as a Service Architecture to Support Cloud-Based and Event-Driven Control Application Development.
In: *2019 IEEE International Congress on Internet of Things (ICIOT)*.
Mailand, Italien, 08.07.2019–13.07.2019, S. 41–49.
DOI: 10.1109/ICIOT.2019.00019

Martínez-Fernandez et al. 2015
Martínez-Fernandez, Silverio; Ayala, Claudia P.; Franch, Xavier; Nakagawa, Elisa Y., 2015.
A survey on the benefits and drawbacks of AUTOSAR.
In: *2015 First International Workshop on Automotive Software Architecture (WASA)*.
Montreal, Kanada, 04.05.2015–08.05.2015, S. 19–26.
DOI: 10.1145/2752489.2752493

Masior et al. 2020
Masior, Jonathan; Schneider, Benjamin; Tasci, Timur; Riedel, Oliver, 2020.
Modellvernetzung im Advanced Systems Engineering.
Zeitschrift für wirtschaftlichen Fabrikbetrieb **115** (7-8), S. 545–549.
DOI: 10.3139/104.112305

Menard et al. 2020
Menard, Christian; Goens, Andrés; Lohstroh, Marten; Castrillon, Jeronimo, 2020.
Achieving Determinism in Adaptive AUTOSAR.
In: *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*.
Grenoble, Frankreich, 09.03.2020–13.03.2020, S. 822–827.
DOI: 10.23919/DATE48585.2020.9116430

Meyer 2002
Meyer, Dirk, 2002.
Objektverwaltungskonzept für operative Prozessleittechnik.
Düsseldorf, RWTH Aachen, Doktorarbeit.
ISBN: 978-3-18-394008-0

Mohamed et al. 2021
Mohamed, Mustafa Abshir; Kardas, Geylani; Challenger, Moharram, 2021.
Model-Driven Engineering Tools and Languages for Cyber-Physical Systems–A Systematic Literature Review.
IEEE Access **9**, S. 48605–48630.
DOI: 10.1109/ACCESS.2021.3068358

Monostori et al. 2016
Monostori, László; Kádár, Botond; Bauernhansl, Thomas; Kondoh, Shinsuke; Kumara, Soundar; Reinhart, Gunther; Sauer, Olaf; Schuh, Günther; Sihn, Wilfried; Ueda, Kanji, 2016.
Cyber-physical systems in manufacturing.
CIRP Annals **65** (2), S. 621–641.
DOI: 10.1016/j.cirp.2016.06.005

Mowad et al. 2022
Mowad, Abrar Mohammad; Fawareh, Hamed; Hassan, Mohammad A., 2022.
Effect of Using Continuous Integration (CI) and Continuous Delivery (CD) Deployment in DevOps to reduce the Gap between Developer and Operation.
In: *2022 International Arab Conference on Information Technology (ACIT)*.
Abu Dhabi, Vereinigte Arabische Emirate, 22.11.2022–24.11.2022, S. 1–8.
DOI: 10.1109/ACIT57182.2022.9994139

Nakano et al. 2021
Nakano, Wakako; Shinohara, Yukino; Ishiura, Nagisa, 2021.
Full Hardware Implementation of FreeRTOS-Based Real-Time Systems.
In: *TENCON 2021 - 2021 IEEE Region 10 Conference (TENCON)*.
Auckland, Neuseeland, 07.12.2021–10.12.2021, S. 435–440.
DOI: 10.1109/TENCON54134.2021.9707328

Napoleone et al. 2023
Napoleone, Alessia; Andersen, Ann-Louise;
Brunoe, Thomas Ditlev; Nielsen, Kjeld, 2023.
Towards human-centric reconfigurable manufacturing systems: Literature review of reconfigurability enablers for reduced reconfiguration effort and classification frameworks.
Journal of Manufacturing Systems **67**, S. 23–34.
DOI: 10.1016/j.jmsy.2022.12.014

National Institute of Standards & Technology 2002
National Institute of Standards & Technology, Intelligent Systems Division, 2002.
Introduction to Linux for Real-Time Control - Introductory Guidelines and Reference for Control Engineers and Managers Prepared. Gaithersburg, MD, USA.
Version 2.0.0, frederick.proctor@nist.gov

Neumann et al. 2022
Neumann, Eva-Maria; Vogel-Heuser, Birgit;
Gnadlinger, Michael; Fischer, Juliane; Reimoser, Laura;
Diehm, Sebastian; Englert, Tobias; Schwarz, Michael, 2022.
Metric-based Identification of Target Conflicts in the Development of Industrial Automation Software Libraries.
In: *2022 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*.
Kuala Lumpur, Malaysia, 07.12.2022–10.12.2022,
S. 1493–1499.
DOI: 10.1109/IEEM55944.2022.9989691

Nikolov et al. 2021
Nikolov, Neven; Nakov, Ognyan; Gotseva, Daniela, 2021.
Operating Systems for IoT Devices.
In: *2021 56th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*.
Sosopol, Bulgarien, 16.06.2021–18.06.2021, S. 41–44.
DOI: 10.1109/ICEST52640.2021.9483469

Nof 2009
Nof, Shimon Y., 2009.
Springer Handbook of Automation. Nof, Shimon Y. (Hrsg.):
Berlin, Heidelberg: Springer.
ISBN: 978-3-540-78830-0

Nof 2023
Nof, Shimon Y., 2023.
Automation: What It Means to Us Around the World, Definitions, Its Impact, and Outlook.
In: Nof, Shimon Y. (Hrsg.): *Springer Handbook of Automation*.
Cham: Springer International Publishing, S. 3–38.
ISBN: 978-3-030-96729-1

OPC Foundation 2015
OPC Foundation, 2015.
OPC Unified Architecture Part 2 - Security Model.
Release 1.03.
URL: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-2-security-model/

OPC Foundation 2017a
OPC Foundation, 2017.
OPC Unified Architecture Part 1 - Overview and Concepts.
Release 1.04.
URL: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-1-overview-and-concepts/

OPC Foundation 2017b
OPC Foundation, 2017.
OPC Unified Architecture Part 3 - Address Space Model.
Release 1.04.
URL: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-3-address-space-model/

OPC Foundation 2017c
OPC Foundation, 2017.
OPC Unified Architecture Part 5 - Information Model.
Release 1.04.
URL: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-5-information-model/

Paez et al. 2020
Paez, Nicolas; Fontdevila, Diego; Oliveros, Alejandro, 2020.
On the Influence of Agile in the Usage of Software Development Practices.
In: *2020 IEEE Congreso Bienal de Argentina (ARGENCON)*.
online, 01.12.2020–04.12.2020, S. 1–7.
DOI: 10.1109/ARGENCON49523.2020.9505407

Pallasch et al. 2018a
Pallasch, Christoph; Wein, Stephan; Hoffmann, Nicolai; Obdenbusch, Markus; Buchner, Tilman; Waltl, Josef; Brecher, Christian, 2018.
Edge Powered Industrial Control: Concept for Combining Cloud and Automation Technologies.
In: *2018 IEEE International Conference on Edge Computing (EDGE)*.
San Francisco, Kalifornien, USA, 02.07.2018–07.07.2018, S. 130–134.
ISBN: 978-1-5386-7238-9

Pallasch et al. 2018b Pallasch, Christoph; Wein, Stephan; Hoffmann, Nicolai; Obdenbusch, Markus; Buchner, Tilman; Waltl, Josef; Brecher, Christian, 2018.
Edge Powered Industrial Control: Concept for Combining Cloud and Automation Technologies.
In: *2018 IEEE International Conference on Edge Computing (EDGE)*.
San Francisco, Kalifornien, USA, 02.07.2018–07.07.2018, S. 130–134.
DOI: 10.1109/EDGE.2018.00026

Plattform Industrie 4.0 2016 Plattform Industrie 4.0, 2016.
Struktur der Verwaltungsschale - Fortentwicklung des Referenzmodells für die Industrie-4.0-Komponente.
Bundesministerium für Wirtschaft und Energie (BMWi) (Hrsg.): Berlin

Plattform Industrie 4.0 2019 Plattform Industrie 4.0, 2019.
Verwaltungsschale in der Praxis - Diskussionspapier.
Bundesministerium für Wirtschaft und Energie (BMWi) (Hrsg.): Berlin

Plattform Industrie 4.0 2021 Plattform Industrie 4.0, 2021.
Details of the Asset Administration Shell Part 2 - Interoperability at Runtime - Exchanging Information via Application Programming Interfaces (Version 1.0RC02). Berlin: Federal Ministry for Economic Affairs und Energy (BMWi)

Plattform Industrie 4.0 2022 Plattform Industrie 4.0, 2022.
Details of the Asset Administration Shell Part 1 - The exchange of information between partners in the value chain of Industrie 4.0 (Version 3.0RC02). Berlin: Federal Ministry for Economic Affairs und Climate Action (BMWK)

Poștovei et al. 2020 Poștovei, Danuț Adrian; Bulac, Constantin; Triștiu, Ion; Camachi, Balduino, 2020.
The evolution and challenges of modern Distributed Control Systems.
In: *2020 IEEE 14th International Symposium on Applied Computational Intelligence and Informatics (SACI)*.
Timisoara, Rumänien, 21.05.2020–23.05.2020, S. 000089–000094.
DOI: 10.1109/SACI49304.2020.9118829

Prasetyo & Soliman 2021 Prasetyo, Yogi Tri; Soliman, Karim Osama Saad, 2021.
Usability Evaluation of ERP Systems: A Comparison between SAP S/4 Hana & Oracle Cloud.
In: *2021 IEEE 8th International Conference on Industrial Engineering and Applications (ICIEA)*.
Barcelona, Spanien, 08.01.2021–11.01.2021.
DOI: 10.1109/ICIEA52957.2021.9436697

Rao & Shorey 2017 Rao, Shilpa; Shorey, Rajeev, 2017.
Efficient device-to-device association and data aggregation in industrial IoT systems.
In: *2017 9th International Conference on Communication Systems and Networks (COMSNETS)*.
Bengaluru, Indien, 04.01.2017–08.01.2017, S. 314–321.
ISBN: 978-1-5090-4250-0

Raymond 2003 Raymond, Eric Steven, 2003.
The Art of Unix Programming. Revision 1.0.
URL: http://www.catb.org/~esr/writings/taoup/html/
Zugriff am: 20.02.2022

Riedl et al. 2004 Riedl, Matthias; Diedrich, Christian; Naumann, Frank; Simon, René, 2004.
An object based approach for distributed automation.
In: *2004 IEEE Africon. 7th Africon Conference in Africa*.
Gaborone, Botswana, 15.09.2004–17.09.2004, S. 1253–1260.
ISBN: 978-0-7803-8605-1

Riedl 2005 Riedl, Matthias, 2005.
Distributed Object Model Environment - Ein objektorientiertes Softwaremodell für verteilte Automatisierungssysteme. Aachen: Shaker Verlag GmbH.
ISBN: 3-8322-4644-4

Riedl et al. 2011 Riedl, Matthias; Zipper, Holger; Bangemann, Thomas; Meier, Marco, 2011.
Distributed automation system supports process monitoring and control.
Proceedings - ISIE 2011: 2011 IEEE International Symposium on Industrial Electronics.
DOI: 10.1109/ISIE.2011.5984319

Riedl et al. 2013 Riedl, Matthias; Zipper, Holger; Meier, Marco; Diedrich, Christian, 2013.
Automation meets CPS.
IFAC Proceedings Volumes **46** (7), S. 216–221.
11th IFAC Workshop on Intelligent Manufacturing Systems.
DOI: 10.3182/20130522-3-BR-4036.00071

Riedl et al. 2014 Riedl, Matthias; Zipper, Holger; Meier, Marco; Diedrich, Christian, 2014.
Cyber-physical systems alter automation architectures.
Annual Reviews in Control **38** (1), S. 123–133.
DOI: 10.1016/j.arcontrol.2014.03.012

Riehle 2000 Riehle, Dirk, 2000.
Framework Design: A Role Modeling Approach.
Zürich, Swiss Federal Institute of Technology at Zurich (ETH Zurich), Doktorarbeit

Rigazzi et al. 2015 Rigazzi, Giovanni; Pratas, Nuno K.; Popovski, Petar; Fantacci, Romano, 2015.
Aggregation and trunking of M2M traffic via D2D connections.
In: *2015 IEEE International Conference on Communications (ICC)*.
London, Vereinigtes Königreich, 08.06.2015–12.06.2014, S. 2973–2978.
ISBN: 978-1-4673-6432-4

Rikalovic et al. 2022 Rikalovic, Aleksandar; Suzic, Nikola; Bajic, Bojana; Piuri, Vincenzo, 2022.
Industry 4.0 Implementation Challenges and Opportunities: A Technological Perspective.
IEEE Systems Journal **16** (2), S. 2797–2810.
DOI: 10.1109/JSYST.2021.3101673

Rödel 2020 Rödel, Alain, 2020.
Entwicklung eines Frameworks zur Produktionsorganisation auf Basis des Konzepts der Verwaltungsschale.
Universität Stuttgart, Masterarbeit

Rothhöft 2018 Rothhöft, Michaela, 2018.
Marktstudie SPS-Systeme 2018. Hemer: Rothhöft, Michaela.
URL: https://www.marktstudien.org/

Rothhöft 2022 Rothhöft, Michaela, 2022.
Marktstudie SPS-Systeme 2022. Hemer: Rothhöft, Michaela.
URL: https://www.marktstudien.org/

Rupprecht et al. 2021 Rupprecht, Bernhard; Trunzer, Emanuel; König, Simone; Vogel-Heuser, Birgit, 2021.
Concepts for Retrofitting Industrial Programmable Logic Controllers for Industrie 4.0 Scenarios.
In: *2021 22nd IEEE International Conference on Industrial Technology (ICIT)*.
Valencia, Spanien, 10.03.2021–12.03.2021, S. 1034–1041.
DOI: 10.1109/ICIT46573.2021.9453558

Sakurada et al. 2021 Sakurada, Lucas; Leitao, Paulo; De la Prieta, Fernando, 2021.
Towards the Digitization using Asset Administration Shells.
In: *IECON 2021 – 47th Annual Conference of the IEEE Industrial Electronics Society*.
Toronto, Kanada, 13.10.2021–16.10.2021, S. 1–6.
DOI: 10.1109/IECON48115.2021.9589370

Schill & Springer 2012 Schill, Alexander; Springer, Thomas, 2012.
Verteilte Systeme. Berlin, Heidelberg: Springer. eXamen.press.
ISBN: 978-3-642-25795-7

Schleipen et al. 2018 Schleipen, Miriam; Aro, Jouni; Bajorat, Jan; Berger, Christoph, 2018.
Praxishandbuch OPC UA: Grundlagen, Implementierung, Nachrüstung, Praxisbeispiele. 1. Auflage. Würzburg: Vogel Business Media.
ISBN: 9783834334138

Schmidt et al. 2016 Schmidt, Jan-Philipp; Zeller, Andreas; Weyrich, Michael, 2016.
Ansatz zur modellgetriebenen Entwicklung flexibler Automatisierungssysteme durch Serviceorientierung.
In: *EKA 2016: 14. Fachtagung Entwurf komplexer Automatisierungssysteme*.
Magdeburg, Deutschland, 24.05.2016–25.05.2016, S. 8

Schmidt et al. 2005 Schmidt, Marc-Thomas; Hutchison, Beth; Lambros, Peter; Phippen, Rob, 2005.
The Enterprise Service Bus: Making service-oriented architecture real.
IBM Systems Journal **44** (4), S. 781–797.
DOI: 10.1147/sj.444.0781

Schmied et al. 2020 Schmied, Sebastian; Mathias, Selvine G.; Großmann, Daniel; Jumar, Ulrich, 2020.
Integration of Existing Cyber-Physical Manufacturing Systems into a Common Information Model.
IFAC-PapersOnLine **53** (2), S. 10905–10910.
21st IFAC World Congress.
DOI: 10.1016/j.ifacol.2020.12.2827

Schnicke et al. 2022 Schnicke, Frank; Haque, Ashfaqul; Kuhn, Thomas; Espen, Daniel; Antonino, Pablo Oliveira, 2022.
Architecture Blueprints to Enable Scalable Vertical Integration of Assets with Digital Twins.
In: *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*.
Stuttgart, Deutschland, 06.09.2022–09.09.2022, S. 1–8.
DOI: 10.1109/ETFA52439.2022.9921728

Sehr et al. 2021 Sehr, Martin A.; Lohstroh, Marten; Weber, Matthew; Ugalde, Ines; Witte, Martin; Neidig, Joerg; Hoeme, Stephan; Niknami, Mehrdad; Lee, Edward A., 2021.
Programmable Logic Controllers in the Context of Industry 4.0.
IEEE Transactions on Industrial Informatics **17** (5), S. 3523–3533.
DOI: 10.1109/TII.2020.3007764

Seitz 2015
Seitz, Matthias, 2015.
Speicherprogrammierbare Steuerungen für die Fabrik- und Prozessautomatisierung - Strukturierte und objektorientierte SPS-Programmierung, Motion Control, Sicherheit, vertikale Integration. 4. Auflage. München: Carl Hanser Verlag GmbH & Co. KG.
ISBN: 978-3-446-44273-3

Seitz 2021
Seitz, Matthias, 2021.
Speicherprogrammierbare Steuerungen in der Industrie 4.0 - Objektorientierter System- und Programmentwurf, Motion Control, Sicherheit, Industrial IoT. 5. Auflage. München: Carl Hanser Verlag GmbH & Co. KG.
ISBN: 978-3-446-46579-4

Serino & Cheng 2020
Serino, Anthony; Cheng, Liang, 2020.
Real-Time Operating Systems for Cyber-Physical Systems: Current Status and Future Research.
In: *2020 International Conferences on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*.
Rhodos, Griechenland, 02.11.2020–06.11.2020, S. 419–425.
DOI: 10.1109/iThings-GreenCom-CPSCom-SmartData-Cybermatics50389.2020.00080

Shan et al. 2010
Shan, Yugang; Wang, Hong; Li, Jun; Hu, Weiguo, 2010.
Research on Fieldbus Device Integration Based on EDD and FDT.
In: *2010 Second International Conference on Intelligent Human-Machine Systems and Cybernetics*.
Nanjing, China, 26.08.2010–28.08.2010, S. 133–136.
DOI: 10.1109/IHMSC.2010.132

Sieffert & Dorofeev 2021
Sieffert, David; Dorofeev, Kirill, 2021.
Generation of Inter-PLC Communication in Distributed Control Systems Using IEC 61499.
In: *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*.
Västerås, Schweden, S. 1–4.
DOI: 10.1109/ETFA45728.2021.9613617

Sollfrank et al. 2021
Sollfrank, Michael; Loch, Frieder; Denteneer, Steef; Vogel-Heuser, Birgit, 2021.
Evaluating Docker for Lightweight Virtualization of Distributed and Time-Sensitive Applications in Industrial Automation.
IEEE Transactions on Industrial Informatics **17** (5).
DOI: 10.1109/TII.2020.3022843

Stepien et al. 2022 Stepien, Leonard; Hallerbach, Sven; Köster, Frank, 2022.
Accelerating Simulation-Enabled Engineering.
In: *2022 IEEE International Symposium on Systems Engineering (ISSE)*.
Wien, Österreich, 24.10.2022–26.10.2022, S. 1–8.
DOI: 10.1109/ISSE54508.2022.10005397

Stock et al. 2014 Stock, Daniel; Stöhr, Matthias; Rauschecker, Ursula; Bauernhansl, Thomas, 2014.
Cloud-based Platform to Facilitate Access to Manufacturing IT.
Procedia CIRP **25**, S. 320–328.
DOI: 10.1016/j.procir.2014.10.045

Subramanian et al. 2019 Subramanian, Geetha; Patil, Bhushan T; Kokate, Meera, 2019.
Review of Modern Technologies in Manufacturing Sector.
In: *2019 International Conference on Advances in Computing, Communication and Control (ICAC3)*.
Mumbai, Indien, 20.12.2019–21.12.2019, S. 1–6.
DOI: 10.1109/ICAC347590.2019.9036835

Tabim et al. 2021 Tabim, Verônica Maurer; Ayala, Néstor Fabián; Frank, Alejandro G., 2021.
Implementing Vertical Integration in the Industry 4.0 Journey: Which Factors Influence the Process of Information Systems Adoption?
Information Systems Frontiers.
DOI: 10.1007/s10796-021-10220-x

Tantik & Anderl 2017 Tantik, Erdal; Anderl, Reiner, 2017.
Integrated Data Model and Structure for the Asset Administration Shell in Industrie 4.0.
Procedia CIRP **60**, S. 86–91.
DOI: 10.1016/j.procir.2017.01.048

Tantik & Anderl 2018 Tantik, Erdal; Anderl, Reiner, 2018.
Concept of the asset administration shell as a software-defined system.
In: *2018 Fifth International Conference on Software Defined Systems (SDS)*.
Barcelona, Spanien, S. 52–58.
DOI: 10.1109/SDS.2018.8370422

Tasci et al. 2018 Tasci, Timur; Melcher, Jan; Verl, Alexander, 2018.
A Container-based Architecture for Real-Time Control Applications.
In: *2018 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC)*.
Stuttgart, Deutschland, 17.06.2018–20.06.2018, S. 1–9.
DOI: 10.1109/ICE.2018.8436369

Thramboulidis 2013 Thramboulidis, Kleanthis, 2013.
IEC 61499 vs. 61131: A Comparison Based on Misperceptions.
Journal of Software Engineering and Applications **06** (08), S. 405–415.
DOI: 10.4236/jsea.2013.68050

Vathoopan et al. 2019 Vathoopan, Milan; Cabral, Jose; Wenger, Monika; Knoll, Alois; Zoitl, Alois, 2019.
Planning and Engineering Component-Based Automation Systems in AutomationML.
In: *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*.
Zaragoza, Spanien, 10.09.2019–13.09.2019, S. 118–125.
DOI: 10.1109/ETFA.2019.8869437

VDI 3687 VDI 3687 - 1999-11.
Auswahl von Feldbussystemen durch Bewertung ihrer Leistungseigenschaften für industrielle Anwendungsbereiche

VDI 3697-1 VDI 3697 Blatt 1 - 2018-06).
Empfehlung zur technischen Umsetzung des Datenaustauschs zwischen den Engineering-Systemen für PLT und PCS - Datenaustausch von PLT-Stellen gemäß NE 150 mit AutomationML

VDI 3697-2 VDI 3697 Blatt 2 - 2019-09.
Empfehlung zur technischen Umsetzung des Datenaustauschs zwischen Engineering-Systemen - Datenaustausch zwischen CAE-Systemen der Verfahrensauslegung und der PLT-Hardware-Planung gemäß NE 159 mit AutomationML

VDI 5600-1 VDI 5600 Blatt 1 - 2016-10.
Fertigungsmanagementsysteme (Manufacturing Execution Systems – MES)

VDI 5600-3 VDI 5600 Blatt 3 - 2016-10.
Fertigungsmanagementsysteme (Manufacturing Execution Systems – MES)

VDI 5600-7 VDI 5600 Blatt 7 - 2021-04.
Fertigungsmanagementsysteme (Manufacturing Execution Systems – MES): MES und Industrie 4.0

VDI/VDE-Fachausschuss 7.20 2013 VDI/VDE-Fachausschuss 7.20, 2013.
Cyber-Physical Systems: Chancen und Nutzen aus Sicht der Automation. VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik (GMA).
URL: https://www.vdi.de/ueber-uns/presse/publikationen/details/cyber-physical-systems-chancen-und-nutzen-aus-sicht-der-automation
Zugriff am: 20.02.2022

Vogel-Heuser et al. 2014
Vogel-Heuser, Birgit; Diedrich, Christian; Fay, Alexander; Jeschke, Sabina; Kowalewski, Stefan; Wollschlaeger, Martin; Göhner, Peter, 2014.
Challenges for Software Engineering in Automation.
Journal of Software Engineering and Applications **7**, S. 440–451.
DOI: 10.4236/jsea.2014.75041

Vogel–Heuser et al. 2022
Vogel–Heuser, Birgit; Neumann, Eva–Maria; Fischer, Juliane; Marcos, Marga; Estévez, Elisabet; Barbieri, Giacomo; Sonnleithner, Lisa; Rabiser, Rick, 2022.
Automation Software Architecture in CPPS - Definition, Challenges and Research Potentials.
In: *2022 IEEE 5th International Conference on Industrial Cyber-Physical Systems (ICPS)*.
Coventry, Vereinigtes Königreich, 24.05.2022–26.05.2022, S. 01–08.
DOI: 10.1109/ICPS51978.2022.9816893

Vyatkin 2013
Vyatkin, Valeriy, 2013.
Software Engineering in Industrial Automation: State-of-the-Art Review.
IEEE Transactions on Industrial Informatics **9** (3), S. 1234–1249.
DOI: 10.1109/TII.2013.2258165

Wagner et al. 2014
Wagner, Marco; Zobel, Dieter; Meroth, Ansgar, 2014.
Towards runtime adaptation in AUTOSAR: Adding Service-orientation to automotive software architecture.
In: *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*.
Barcelona, Spanien, 16.09.2014–19.09.2014, S. 1–7.
ISBN: 978-1-4799-4845-1

Wei 2021
Wei, Yaxin, 2021.
Research on Real-time Improvement Technology of Linux Based on Multi-core ARM.
In: *2021 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*.
Dalian, China, 28.06.2021–30.06.2021, S. 1061–1066.
DOI: 10.1109/ICAICA52286.2021.9498165

Weyns et al. 2021
Weyns, Danny; Bures, Tomas; Calinescu, Radu; Craggs, Barnaby; Fitzgerald, John; Garlan, David; Nuseibeh, Bashar; Pasquale, Liliana; Rashid, Awais; Ruchkin, Ivan; Schmerl, Bradley, 2021.
Six Software Engineering Principles for Smarter Cyber-Physical Systems.
In: *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*.
Washington, D.C., USA, 27.09.2021–01.10.2021, S. 198–203.
DOI: 10.1109/ACSOS-C52956.2021.00058

Wiesmayr et al. 2021 Wiesmayr, Bianca; Zoitl, Alois; Rabiser, Rick, 2021.
Assessing the Usefulness of a Visual Programming IDE for Large-Scale Automation Software.
In: *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*.
Fukuoka, Japan, 10.10.2021–15.10.2021, S. 297–307.
DOI: 10.1109/MODELS50736.2021.00037

Wollschlaeger et al. 2017 Wollschlaeger, Martin; Sauter, Thilo; Jasperneite, Juergen, 2017.
The Future of Industrial Communication: Automation Networks in the Era of the Internet of Things and Industry 4.0.
IEEE Industrial Electronics Magazine **11** (1), S. 17–27.
DOI: 10.1109/MIE.2017.2649104

Xu et al. 2021 Xu, Lei; Chen, Lin; Gao, Zhimin; Moya, Hiram; Shi, Weidong, 2021.
Reshaping the Landscape of the Future: Software-Defined Manufacturing.
Computer **54** (7), S. 27–36.
DOI: 10.1109/MC.2021.3074851

Ye & Hong 2019 Ye, Xun; Hong, Seung Ho, 2019.
Toward Industry 4.0 Components: Insights Into and Implementation of Asset Administration Shells.
IEEE Industrial Electronics Magazine **13** (1), S. 13–25.
DOI: 10.1109/MIE.2019.2893397

Ye et al. 2021 Ye, Xun; Yu, Mengmeng; Song, Won Seok; Hong, Seung Ho, 2021.
An Asset Administration Shell Method for Data Exchange Between Manufacturing Software Applications.
IEEE Access **9**, S. 144171–144178.
DOI: 10.1109/ACCESS.2021.3122175

Ye et al. 2022 Ye, Xun; Song, Won Seok; Hong, Seung Ho; Kim, Yu Chul; Yoo, Nam Hyun, 2022.
Toward Data Interoperability of Enterprise and Control Applications via the Industry 4.0 Asset Administration Shell.
IEEE Access **10**, S. 35795–35803.
DOI: 10.1109/ACCESS.2022.3163738

Yoong et al. 2015 Yoong, Li Hsien; Roop, Partha S.; Bhatti, Zeeshan E.; Kuo, Matthew M. Y., 2015.
Model-Driven Design Using IEC 61499. Cham: Springer International Publishing.
ISBN: 978-3-319-10520-8

Yuan et al. 2021 Yuan, Huimiao; Hao, Huijuan; Zhang, Meng, 2021.
Overview of OPC UA TSN.
In: *2021 IEEE 5th Information Technology,Networking,Electronic and Automation Control Conference (ITNEC)*.
Xi'an, China, 15.10.2021–17.10.2021, S. 715–718.
DOI: 10.1109/ITNEC52019.2021.9586911

Zhang et al. 2023 Zhang, Kunwu; Shi, Yang; Karnouskos, Stamatis; Sauter, Thilo; Fang, Huazhen; Colombo, Armando Walter, 2023.
Advancements in Industrial Cyber-Physical Systems: An Overview and Perspectives.
IEEE Transactions on Industrial Informatics **19** (1), S. 716–729.
DOI: 10.1109/TII.2022.3199481

Zhong et al. 2017 Zhong, Ray Y.; Xu, Xun; Klotz, Eberhard; Newman, Stephen T., 2017.
Intelligent Manufacturing in the Context of Industry 4.0: A Review.
Engineering **3** (5), S. 616–630.
DOI: 10.1016/J.ENG.2017.05.015

Zoitl & Lewis 2014 Zoitl, Alois; Lewis, Robert W., 2014.
Modelling control systems using IEC 61499. 2nd ed. Stevenage: Institute of Engineering and Technology. IET control engineering series.
ISBN: 9781849197601

Zoitl & Strasser 2016 Zoitl, Alois; Strasser, Thomas, 2016.
Distributed control applications: guidelines, design patterns, and application examples with the IEC 61499. Zoitl, Alois; Strasser, Thomas (Hrsg.): Boca Raton, FL: CRC Press Taylor & Francis Group.
ISBN: 978-1-4822-5905-6

A Technische Grundlagen des Frameworks

A.1 Verteilte Systeme

Verteilte Systeme zeichnen sich durch die Zerlegung und Verteilung einer Gesamtaufgabe auf mehrere getrennte Plattformen aus. Die Bewältigung der gestellten Aufgabe kann durch kooperationsorientiertes Arbeiten unter Einhaltung festgelegter Abläufe gewährleistet werden. Als ein häufiges Merkmal gilt der systemübergreifende Rückgriff auf zentrale Daten oder Betriebsmittel. Dieser Rückgriff wird durch dedizierte Systeme ohne aufgaben- oder prozessgebundene Funktionalität ermöglicht, sodass in freiestmöglicher Konfiguration zugegriffen werden kann. (Schill & Springer 2012) In der logischen Betrachtung steht das durch die Komponenten gebildete Gesamtsystem als ein singuläres, geschlossenes System. Es sind vier Motive für die Bildung verteilter Systeme zu nennen (Schill & Springer 2012):

- **Räumliche Trennung der Systemkomponenten**: Einzelsysteme eines umzusetzenden Prozesses können an verschiedenen Standorten betrieben werden.
- **Parallelisierung und Lastenausgleich**: Daten oder Rechenaufgaben können aufgebrochen, verteilt und parallel bearbeitet werden, um verfügbare Fähigkeiten auszulasten und beispielsweise Terminanforderungen besser einhalten zu können.
- **Verfügbarkeit**: Um eine ausreichende Systemzuverlässigkeit zu gewährleisten, können Systemkomponenten schneller getauscht oder in mehrfach instanziiert werden. Die Auswirkungen eines Komponentenausfalls sind begrenzbar.
- **Skalierbarkeit**: Die Systemkapazität kann durch den flexiblen Betrieb der Systemkomponenten an den tatsächlichen Bedarf angepasst werden.

Verteilte Systeme unterscheiden sich von herkömmlichen Systemen besonders durch ihre teilweise oder komplette physische Verteilung ihrer Komponenten aus. Ihre Funktionalität

ist auf intern standardisierten Logikstrukturen und Schnittstellen aufgebaut, um die Integration zwischen den Systemen herstellen zu können. (Schill & Springer 2012) Die Integration einzelner Systeme in ein verteiltes System stellt feste und je nach Umsetzungsform optionale Anforderungen in der Systemqualität. Grundvoraussetzungen eines verteilten Systems sind die Fähigkeiten der Komponenten, miteinander kommunizieren zu können, und die Betreibbarkeit als dezentrale Lösung, um die Verteilung überhaupt zu ermöglichen. Eine Herausforderung im verteilten System ist die Konsistenzwahrung beim Umgang mit Daten: Durchschreiten Daten eine Komponentenkette bei der Bearbeitung eines Geschäftsprozesses durch ein verteiltes System, muss gesichert werden, dass die Transaktionen zwischen den Komponenten konsistent erfolgen, die Daten nicht unzulässig verändert werden und eine Nachvollziehbarkeit aller Veränderungen integriert ist. Hinzu kommt eine ausreichende Sicherheit gegenüber Manipulationen, da durch die Auflösung des Gesamtsystems in mehrere Komponenten mit Kommunikationskanälen die theoretische Angriffsfläche vergrößert wird. Um maschinelle Vorgänge und Entscheidungen im Betrieb nachvollziehen zu können und notwendige menschliche Entscheidungen oder Eingriffe zu unterstützen, müssen Gesamtsystem und Komponenten ausreichend transparent sein. Wird ein verteiltes System rekonfigurierbar gestaltet, muss die Integration trotz konfigurationsbedingter Änderungen gewährleistet sein.

A.1.1 Paradigmen im verteilten System

Die Strukturbeschaffenheit eines verteilten Systems ist abstrahierbar. Kernabstraktionselemente für die Charakterisierung eines verteilten Systems sind die Integration im verteilten System, der Aufbau der Softwareanwendungen sowie der logische Aufbau der Kommunikation zwischen den Softwareanwendungen. (Schill & Springer 2012)

Integrationsparadigmen

Für die Integration von verteilten Systemen existieren zwei grundsätzliche Integrationsparadigmen, auf denen weitere, spezifischere Unterparadigmen aufbauen können: (Schill & Springer 2012):

- Client-Server-Paradigma
  - klassische Struktur
  - zentraler leitender Server stellt Daten, Funktionen, Anweisungen etc. bereit
  - Clientsysteme bearbeiten Aufgaben unter Verbindung mit Server
  - Aufgaben gehen über Clients oder Server ein
  - Arbeit erfolgt synchron
  - einfache Aufrufstrukturen, weil an einem Punkt (Server) dirigiert wird
- Peer-to-peer-Paradigma:
  - Aufbau eines Netzwerks aus eigenständigen Systemen
  - Systeme im Netzwerk weitestgehend gleichberechtigt
  - direkte Kommunikation zwischen Systemen
  - Server- und Clientfunktionalitäten, -daten auf Systemen verteilt
  - Arbeit erfolgt asynchron
  - kompliziertere Aufrufstrukturen, weil parallel und dezentral gearbeitet wird

Misch- oder Parallelformen sowie Verkettung, beispielsweise in einer schichtenweisen Abfolge von Client-Server-Interaktionsmustern, sind möglich. (Schill & Springer 2012) Das umgesetzte Integrationsparadigma bildet die Basisschicht im Aufbau eines verteilten Systems. Da Integrations- und Kommunikationsparadigmen sowie die darauf gelegten Anwendungsausführung in festen Beziehungen zueinander stehen von voneinander abhängig sind, werden die jeweiligen Bestandteile oft gemeinsam implementiert, beispielsweise bei ESB-Implementierungen (Schmidt et al. 2005).

Anwendungsparadigmen

- Geschlossene Anwendungen: Abgeschlossene Programme mit fester Funktionalität und Schnittstellen
- Objektorientierung: definiertes Objekt durch ausführende Systeme durchgereicht, Bearbeitungsfunktionalität sitzt im System

- Komponentenorientierung: abgeschlossene Komponente wird auf System gelegt und ausgeführt unter Nutzung bestehender Ressourcen
- Diensteorientierung: zustandslose Dienste, funktionalitätsorientierte Softwareteile

Das Anwendungsparadigma eines verteilten Systems ist mit den Paradigmen von Integration und Kommunikation verbunden, weil je nach Paradigma, besonders bei einer Diensteorientierung, Vorgaben für Integration und Kommunikation folgen.

Kommunikationsparadigmen

Inhaltlich beruht Kommunikation im verteilten System auf:

- Prozedur- bzw. Methodenaufruf: Ein System löst eine Routine bei einem anderen System mit oder ohne Datenübergabe direkt aus.
- Daten- bzw. Objektübergabe: Ein System verteilt Daten mit oder ohne Anfrage direkt oder allgemein.

Bei einer direkten Integration von verteilten Funktionen muss eine Formatbekanntheit vorliegen: beim Prozedur-/Methodenaufruf müssen ausführendes System, Prozedur-/Methodenidentifikation und bereitzustellende Parameter bekannt sein. Beim Erhalt von Daten- oder Objekten muss ein empfangendes System die Daten verarbeiten können. Die Formatbekanntheit kann durch Implementierung, durch explizite (Modell) oder durch implizite Beschreibung (Metadaten, Variableninhalt etc.) geschaffen werden.

A.1.2 Ausführungsmodelle

In ereignisgetriebenen Softwarearchitekturen (*Event-driven architectures, EDA*) werden Vorgänge und Funktionen in einem Programm durch von innen oder außen eingehende Ereignisse ausgelöst. Es wird kein *Polling* (also ein kontinuierliches Abfragen) vorgenommen, sondern ausschließlich nach Eingang eines bestimmten Signals gearbeitet. Es ist vergleichbar mit der *Interupt*-Funktionalität auf Prozessor- oder Mikrocontrollerebene. Das ereignisgetriebene Vorgehen ist der Quasistandard bei größeren oder serviceorientierten

Softwaresystemen, um die bei einem Polling anfallende Rechenlast zu vermeiden. Ein Ereignis muss kein abstraktes Signal sein, sondern kann auch als Nachricht mit konkretem Inhalt gestaltet sein.

Zyklusbasierte Architekturen prüfen Eingangszustände oder -signale nach einem gleichbleibendem Muster in einer festgelegten logischen und zeitlichen Folge (Polling). Hierzu gehören auch speicherprogrammierbare Steuerungen nach der IEC 61131. Nach Ende einer feststehenden Abfolge von Operationen wird ohne Unterbrechung von vorne begonnen. Der wesentliche Vorteil dieses Ansatzes liegt in der Determinierbarkeit des Systemverhaltens, insbesondere hinsichtlich der Antwortzeit. Für Systeme, die harte Echtzeitanforderungen erfüllen müssen, ist zyklusbasiertes Vorgehen der Standard. Üblicherweise sind zyklusbasierte Systeme einzelne Geräte, es existieren allerdings auch verteilte Systeme, deren Programme zyklusbasiert ausgeführt werden (beispielsweise rtSOA).

A.1.3 System- und Datenaggregation

Da sich die Hersteller, Einführungszeitpunkte und Lebenszeiten von technischen Systemen zur Automatisierung von industriellen und Verwaltungsprozessen unterscheiden, ist die praktische Anwendung mit dem Problem verschiedener technologischer Stände und Möglichkeiten konfrontiert. In dieser Situation zeigen sich dadurch mehrere Probleme:
Um Systeme aus verschiedenen Ebenen der Automatisierungspyramide zu verbinden, müssen meist mehrere Kommunikationsprotokolle integriert werden. Da nicht jedes teilnehmende System zwangsläufig alle benötigten Kommunikationsprotokolle unterstützt, entsteht zusätzlicher Anpassungsbedarf. Da einzelne Systeme oft Informationen von mehreren anderen Systemen benötigen, müssen mehrere Kommunikationsverbindungen aufgespannt und unterhalten werden. Dies belastet die Netzwerkinfrastruktur zusätzlich, vergrößert die Komplexität des Gesamtsystems und damit den Aufwand, der bei Erweiterung und Wartung notwendig wird. Zudem wird die Wahrscheinlichkeit von Eindeutigkeitsproblemen vergrößert, bei denen nicht klar ist, welche Instanz einer Datei die Grundinstanz ist oder welches System die tatsächlich gültige Instanz einer Messgröße liefert. Die entstehende Redundanz durch zusätzliche Verbindungen und mehrfache Datenhaltung erschwert die

automatisierte Integration, belastet das zugrundeliegende Netzwerk stärker und vergrößert den Wartungs- und Energiebedarf. Werden Kommunikationsplattformen zusätzlich zu einem globalen Netzwerk (üblicherweise das Internet) hin exponiert, müssen für jede Plattform einzeln Sicherheitsrichtlinien und -technologien implementiert werden, was zu weiterem Installations- und Wartungsaufwand führt.

Die Datenaggregation entstammt den Mobilfunk- und M2M-Bereichen, in denen die Datenaggregation als Ansatz für die Reduzierung von Datenmengen verfolgt wird (Rigazzi et al. 2015; Rao & Shorey 2017). Datenmodelle im Sinne von Variablenstruktur existieren hierbei nicht. Aggregationsziel ist der Datenverkehr.

Ansätze zur Aggregation von OPC-UA-Servern und den zugehörigen Informationsmodellen auf einem gemeinsamen Server sind bekannt und in Benutzung (Breunig & Schneider 2019; Großmann et al. 2014; Banerjee & Großmann 2017; Schmied et al. 2020). Ziel ist die Integration von OPC-UA-fähigen Geräten an einem Punkt. Datenkunden und -lieferanten können so mehrere Zielsysteme über einen Punkt erreichen. Die Ansätze basieren auf Gesamtinformationsmodellen, die aus den Informationsmodellen der einzelnen Server in ein gemeinsames Informationsmodell zusammengeführt werden. (Großmann et al. 2014; Banerjee & Großmann 2017) OPC UA ist ohne Änderungen oder Erweiterungen aggregationsfähig, da Typsicherheit gewährleistet ist und durch einfache Anpassung der Node-Identifikationen der Ausgangsmodelle eindeutige Verweise gehalten werden können. Die dynamische Aggregation kann während der Entwicklungszeit stattfinden (Brecher et al. 2017).

Breunig & Schneider 2019 stellten eine an einem singulären Punkt ausgeführte, modellgetriebene Aggregation vor. Eine zentrale Applikation verbindet nach einem definierten zentralen Modell zu Kommunikationspartnern, ruft Daten von diesen ab und stellt selbst Daten zur Verfügung. Für die notwendigen Kommunikationsverbindungen werden Kommunikationsmodule, für Transformationen der Daten Operationsmodule instanziiert. Der Gesamtaufbau wird in einem menschenlesbaren Modell beschrieben. Die insgesamt bereitgestellten Daten können entsprechend selbst generiert sein, abgerufene Daten direkt wiedergeben oder auf abgerufenen Daten basieren, beispielsweise als Teil eines Bündels oder einer Transformation. Kommunikations- und Operationsmodule tragen keinen Teil zum Gesamtmodell bei. Es ist starr definiert. Die Module sind nicht synchronisiert und

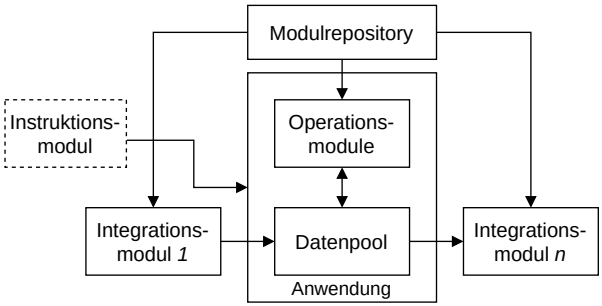


Abbildung A.1: Die Struktur der *mdaa*-Architektur nach Breunig & Schneider 2019

verfügen über keine Ausführungsplanung. Amemiya & Nakao 2020 stellten einen Ansatz zur Bereitstellung einer verteilten Datenbank vor. Eine zentrale, internetbasierte Datenbank bildet den Datenmaster, der auf Clients teilweise oder ganz repliziert und ergänzt wird. Auf Serverebene und beim Datenmaster werden Informationen über die Verteilung der Daten gehalten, damit lokale zustandsorientierte Softwareanwendungen gegen internet-unabhängige Daten arbeiten können. Die Datenstrukturierung und Datenplatzierungen sind nicht statisch und werden von einem clientunabhängigen, zentralen Managementteil organisiert. Aussagen zu Zugriffspriorisierungen oder -beschränkungen sowie Datentypen werden nicht gemacht. Die Arbeit präferiert Datenverteilung mit niedriger Latenz gegenüber Konsistenzanforderungen. Ishihara et al. 2019 zeigten eine verteilte Datenarchitektur, in der in Datenbanken organisierte Datenstrukturen zwischen Clients ausgetauscht werden. Abhängig von Datenänderungen und -abfragen werden Aktualisierungsgraphen geformt, die Rückschlüsse auf das Aktualisierungsverhalten zulassen, sodass Arbeitszyklen auf Aktualisierungszyklen abgestimmt werden können.

Ein Ansatz für ein implizit aggregierendes System wird in Kegel 2010 gezeigt, dargestellt in Abbildung A.2 und referenziert in der Norm VDI 5600-3, welches technische Systeme um ein zentrales Informationsmodell anordnet. Das Informationsmodell dieses Konzepts ist aggregierend, weil es die Informationsschnittstellen aller Anwendungen und Plattformen sammeln muss, um diese ansprechen zu können. Die technischen Systeme arbeiten um ein gemeinsames zentrales Informationsmodell, ohne in eine Hierarchie eingebettet zu sein. Das Modell wird nicht in der detailliert definiert.

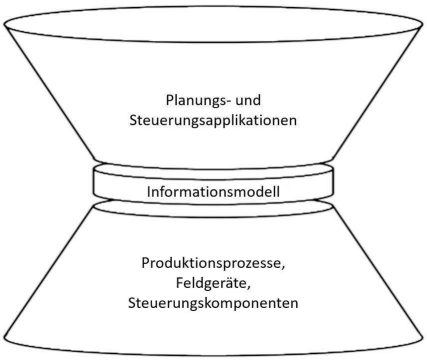


Abbildung A.2: Neue Strukturlogik in der Automatisierung nach Kegel 2010

A.1.4 Echtzeitfähigkeit und Determinismus

Bei Steuerungs- und Überwachungsaufgaben im Feld müssen Automatisierungssysteme Echtzeitanforderungen einhalten. Bei Systemen mit modularisierter Software und Softwareausführung ist die Einhaltung solcher Anforderungen durch die modulweise Verteilung der Gesamtaufgabe, die Modulabhängigkeiten untereinander, die Ausführungs-/Planungskomplexität und die zusätzlich anfallende Kommunikation zwischen den Modulen schwerer. Echtzeitanforderungen müssen auf vier eigenen Ebenen getroffen werden:

1. **Einhaltung auf Modulebene**: Jedes Modul muss seine Teilaufgabe mit einem deterministischen Ablauf erfüllen.

2. **Einhaltung durch die Ausführungsplanung**: Das System, das die Ausführung der Module koordiniert, muss dies in einem eindeutigen und vorhersagbaren Rahmen tun. Relevant ist dies vor allem bei Systemen, bei denen die Module als eigene Anwendungen existieren und ein komplexes Betriebssystem die Taskplanung durchführt.

3. **Einhaltung auf der Kommunikationsebene**: Zwischen den Modulen muss eine Kommunikation stattfinden: durch eine gemeinsame Laufzeitumgebung, eine Interprozesskommunikation oder eine Netzwerkverbindung. Der anfallende Daten-

austausch über die Kommunikationsverbindung(en) muss wiederholbar unter den nötigen Qualitäts- und Zeitanforderungen stattfinden.

4. **Akkumulierte Einhaltung auf Prozessebene**: Die Gesamtanforderung ist indirekt. Ihre Erfüllung resultiert aus der Qualität der vorhergehenden Anforderungen. Die gegenüber der monolithischen Steuerungsausführung anfallende prinzipielle Mehrzeit durch die notwendige Modulkommunikation und -ausführungsplanung muss von vornherein bedacht werden, wenn ein Echtzeitsystem modularisiert werden soll.

Verteilte Softwarearchitekturen können mindestens für weiche Echtzeit befähigt werden, indem die Softwarekomponenten um Beschreibungen hinsichtlich der geforderten zeitlichen Randbedingungen erweitert werden und eine Ausführungsplanung über die gesamte Architektur aufgebaut wird, die entsprechende Ressourcen einplant und die Ausführung überwacht (Cucinotta et al. 2009).

A.1.5 Integration mit physischen Anwendungen

Die Verbindung von verteilten Systemen und physischen Anwendungen erfordert zusätzliche Ansätze. Die am verteilten System beteiligten Einzelsysteme können im Regelfall aufgrund physischer Schnittstellen und Zeitanforderungen nicht beliebig mit Software versehen werden, sondern müssen spezifisch angesteuert oder eingebunden werden. Bei der Integration verteilter Systeme mit physischen Anwendungen können nicht alle Paradigmen verteilter Systeme vollständig eingehalten werden. Eine entstehende Sonderform sind Agentensysteme, in denen jeder Agent (ein aktionsfähiges, eingebundenes Gerät) flexibel und physisorientiert Aufgaben erfüllen kann. Die Ausprägung solcher Agenten in Automatisierungsumgebungen kann in drei Stufen unterschieden werden (Leitao et al. 2017):

- **Gekoppelt** (*Coupled approach*): Entfernt vom physischen System betriebener Agent mit Fernzugriff zum Lesen und Schreiben auf dieses.
  Beispiel: IEC-61131-SPS mit IPC oder VM für Agentensoftware

- **Schwach eingebettet** (*Weak embedded approach*): Auf dem physischen System betriebener Agent mit Zugriff auf Automatisierungsteil über zusätzliche Schicht.
  Beispiel: Hybrid-SPS mit IEC-61131-Teil und IPC-Teil für Agentensoftware

- **Stark eingebettet** (*Strong embedded approach*): Auf dem physischen System betriebener Agent, der fest mit der Automatisierungssoftware integriert ist.
  Beispiel: Soft-SPS (Automatisierung und Agentensoftware auf demselben Kernel)

Die Agenten werden mit ausgelagerten Daten und Funktionen aus den rein virtuellen Systemen des verteilten Systems unterstützt.

A.1.6 Entwicklung und Auslieferung

Entwicklungen verteilter Systeme sind keine standardisierbaren Abläufe, sondern unterliegen, wie die allgemeine Softwareentwicklung, ständiger Veränderung durch neue Werkzeuge, Ziele und Technologien sowie durch neue organisatorische Ansätze (Paez et al. 2020). Entwicklung ist nicht gleichzusetzen mit Programmierung. Für eine längerfristige Betrachtung muss abstrahiert gearbeitet werden. Als Verständnisbezug wird das verteilte System als technische Lösung gesehen, die als Ergebnis am Entwicklungsende steht. (Jacobson et al. 2019) Softwareentwicklung für ständig vernetzte Systeme mit Physisschnittstellen unterliegen keinen anderen Entwicklungsvorgängen als bisherige digitale Systeme mit Physisschnittstellen (Fahmideh et al. 2021).

Die Entwicklung eines verteilten Systems wird für das Verständnis in zwei Bereiche unterteilt: 1. die Entwicklung der funktionalen Software, die im verteilten System betrieben wird, 2. die Instanziierung und Integration der funktionalen Software auf den Teilsystemen des verteilten Systems. Verteilte Systeme können auf festgelegten Architekturvorgaben oder Implementierungen beruhen (Standardzusammenhänge, vorgegebene Protokolle, Bibliotheken et cetera), wodurch in den Bereichen Vorgaben für den Entwicklungsvorgang entstehen. Je ausgeprägter diese Vorgaben sind, desto eingeschränkter ist die Freiheit bei der Wahl an Entwicklungswerkzeugen, Paradigmen und Technologien.

Funktionale Software

Das Ergebnis der Entwicklung der funktionalen Software umfasst Softwareteile wie

- Dienste für Serviceorientierte Architekturen
- Anwendungen/Programme beispielsweise für Laufzeitumgebungen, Steuerungen oder prozedurale Systeme
- Funktionsblöcke für Umgebungen wie IEC 61499 oder IEC 61131-3
- Komponenten für Frameworks wie AUTOSAR

oder weitere sowie (standardisierte) Zusammenstellungen daraus. Diese Teile oder ihre Zusammenstellungen erfüllen Funktionalitäten auf einem Teilsystem des verteilten Systems und werden wiederverwertbar erstellt. Diese Funktionalitäten können allgemein oder auf einen bestimmten Prozess bezogen sein. Ein beispielhaftes technisches Artefakt für eine Funktionalität ist in Abbildung A.3 dargestellt. Neben dem technischen Artefakt, das als vorrangiges Ergebnis entsteht, muss der Bereich ein beschreibendes Artefakt erstellen, oder dem technischen Artefakt die Fähigkeit verleihen, eine Selbstbeschreibung zu erstellen. Die Beschreibung ist notwendig, um das technische Artefakt in einer Instanziierung und Integration einbetten zu können. Wenn die Modellierung der funktionalen Software nicht durch eine Standardarchitektur vorgegeben wird (wie beispielsweise bei AUTOSAR oder der IEC 61499), sind freie Modellierungssysteme wie UML (Klassendiagramme, Komponentendiagramme) oder Petri-Netze anwendbar (Fernández et al. 2021; Bukeikhanov et al. 2020; Schill & Springer 2012; Vyatkin 2013). Für die modellgetriebene Entwicklung von Automatisierungssystemen existieren spezifische Ansätze für UML (Vyatkin 2013; Berardinelli et al. 2016). Das Verhalten des funktionalen Softwareteils muss bekannt oder ableitbar sein.

In Verbindung mit festgelegten technischen Plattformen können mit der programmatisch weitestgehend abgeschlossenen, funktionalen Software Automatisierungskomponenten erstellt werden. Für solche Automatisierungskomponenten bestehen zusätzliche, umfassende Beschreibungssysteme wie AutomationML (Lehner et al. 2021; Vathoopan et al. 2019; Berardinelli et al. 2016) und die Abbildung als Verwaltungsschale (siehe Abschnitt 3.2.3).

Instanziierung und Integration

Um das verteilte System aus bereitstehender funktionaler Software zu erstellen, muss eine definierte Instanziierung und Integration vorgenommen werden. Instanziierung bedeutet

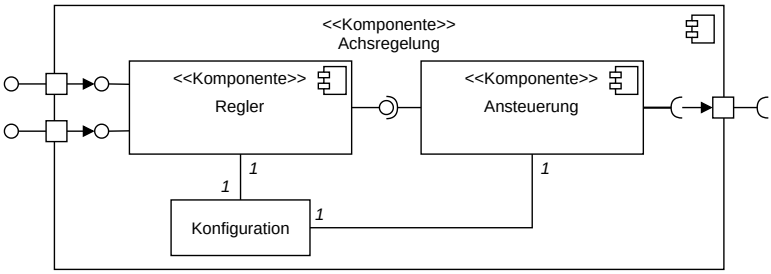


Abbildung A.3: Beispiel eines funktionalen Softwareartefakts im verteilten System

hierbei, dass die funktionale Software auf den Einzelsystemen des verteilten Systems betriebsbereit vorhanden ist, durch eine explizite Einrichtung oder durch eine Inbetriebnahme vorhandener Software. Als Grundlage dienen erstellte funktionale Softwareteile und deren zugehörige Beschreibungen. Integration bedeutet die Herstellung von technischen Verbindungen zwischen Systemen und das Festlegen von logischen Zusammenhängen zwischen Daten und Funktionalitäten im verteilten System. Auf Basis der Integration führen die Teilsysteme des verteilten Systems die Kommunikation zur Aufgabenerfüllung durch. Am Ende steht ein betriebsfähiges verteiltes System, bestehend aus einer Menge an Elementen funktionaler Software, die auf verbundenen technischen Plattformen platziert sind und zur Erfüllung eines festgelegten Prozesses integriert und parametriert sind.

Aus der Entwicklung der Instanziierung und Integration müssen zur Umsetzung des verteilten Systems drei Informationsbereiche hervorgehen. Diese sind als generalisiert zu betrachten, können sich in der Anwendung überschneiden oder bedingt durch technische Festlegungen nicht vorhanden sein.

- **Systemstruktur**: Teilsysteme, funktionale Software, Verteilung der Software
  - Serviceorientierte Architekturen: Beteiligte Dienste
  - IEC 61499: Ressourcenmodell, Anwendungsmodell, Verteilungsmodell
- **Kommunikation**: Verbindungstopologie, Verbindungen zwischen den Einzelsystemen, Verbindungen zwischen Softwareteilen
  - Serviceorientierte Architekturen: Verzeichnismodell

  - IEC 61499: Ressourcenmodell, Anwendungsmodell, Verteilungsmodell

- **Systemlogik**: Parametrierungen, logische Verknüpfungen, logische Abläufe
  - Serviceorientierte Architekturen: Prozessmodelle
  - IEC 61499: Anwendungsmodell

Für Administration und Einbettung des verteilten Systems muss eine Beschreibung entstehen, die erstellt wird oder durch das verteilte System erfolgt. Für Automatisierungssysteme sind neben Standardsystemen wie UML AutomationML oder ähnliche sowie die Verwaltungsschale aufgreifbar. (Azangoo et al. 2020; Juhlin et al. 2021; Lehner et al. 2021; Vathoopan et al. 2019; Berardinelli et al. 2016; Vyatkin 2013). Aufgrund der Bandbreite an abzudeckenden Informationen sind Kombinationen aus mehreren Informationsformaten üblich (beispielsweise mehrere Modelle in IEC 61499 oder mehrere UML-Diagrammtypen zu Komponenten, Abläufen und Sequenzen) (Vyatkin 2013).

Eine Herausforderung ist die Synchronisation der Teilsysteme des verteilten Systems. Ist ein gemeinsamer Zeitbezug notwendig, beispielsweise zur korrelierbaren Datenablage oder wenn keine prozesszustandsbasierte Abstimmung der Teilsysteme möglich ist, muss eine Zeitsynchronisation über die Netzwerkintegration erfolgen. Wesentliche Protokolle für Ethernetverbindungen sind hierfür NTP (*Network Time Protocol*) und PTP (*Precision Time Protocol*). NTP ist internetfähig. PTP benötigt für größtmögliche Genauigkeit Hardwareunterstützung. Mit NTP sind realistische Synchronitäten im einstelligen Millisekundenbereich, mit PTP im einstelligen Mikrosekundenbereich erreichbar. (Gore et al. 2020) Die Wahl der Kommunikationstechnologien im verteilten System ist mit der Wahl der Integrations- und Kommunikationsparadigmen verwoben. Im IoT-Bereich ist das brokerbasierte Pub-/Sub-Protokoll MQTT am stärksten verbreitet (Alkhabbas et al. 2020).

Bei rekonfigurierbaren Softwarearchitekturen muss die Software vorgehalten und bei Bedarf modellgerecht instanziiert und verteilt werden (*Deployment*). Abhängig vom Aufbau der modularisierten Softwarearchitektur sind verschieden große Umfänge und Typen an Software zu verteilen sowie vor Ort in die Struktur des Zielsystems einzubetten.

Das Zielsystem kann vorgegeben oder dynamisch beim Deploymentfall zu bestimmen sein. Die dynamische Bestimmung setzt voraus, dass die zu verteilende Software nicht auf ein bestimmtes Zielsystem angewiesen ist. Ist die zu verteilende Software an feste Randbe-

dingungen gebunden, muss eines oder mehrere Zielsysteme vorgegeben sein. Dieser Fall ist gegeben, wenn die Software auf besondere Hardware wie physische Ein- und Ausgänge angewiesen ist. Ein Deploymentvorgang erfolgt nach einem von zwei wesentlichen Mustern (Arcangeli et al. 2015):

- **Push-Deployment**: Die administrierende Ebene platziert Software auf einem Zielsystem. Die Deployment-Logik liegt auf der administrierenden Ebene. Auf der Systemebene sind Schnittstellen und teilweise Anwendungen einbezogen. Die administrierende Ebene benötigt hinreichende Informationen zu den Fähigkeiten des Zielsystems.
- **Pull-Deployment**: Die Logik liegt auf dem Zielsystem und platziert Deployment-Software aus einer Quelle auf dem eigenen System. Das Zielsystem benötigt Informationen zu Umfängen und Anforderungen der Software.

Die zu verteilende Software kann, wie prinziphaft in Abbildung A.4 dargestellt, in ihrer Ausprägung variieren und ist von der Beschaffenheit des Zielsystems abhängig. Gegenstand eines Deployments kann zum Beispiel eine abgeschlossene Anwendung sein, eine Firmware auf einem einbetteten System oder eine gekapselte Anwendung in einer Container- oder VM-Umgebung. In allen Fällen muss auf der gleichen Ebene des Deploymentgegenstands oder der darunterliegenden Ebene eine Fähigkeit gegeben sein, Software zu platzieren (Pull-Deployment) oder entgegenzunehmen (Push-Deployment). Abhängig von den Fähigkeiten des Zielsystems können mehrere Deployments parallel vorliegen. Die Integrität der Systemstruktur muss garantiert sein, um technische Beeinträchtigungen eines Deployments durch ein anderes auszuschließen, beispielsweise durch Kapselung der platzierten Softwareteile. Zudem müssen für die Konsistenzwahrung alle Deploymentvorgänge gezielt beschrieben, geplant und nachvollziehbar ausgeführt werden. (Arcangeli et al. 2015)

A.2 Betriebssysteme

Betriebssysteme bilden das Softwarefundament heutiger digitaler Rechner. Sie dienen als Schnittstelle für und Ausführungsgrundlage von Programmen. Indem Betriebssysteme Programmen standardisierte, abstrahierende Umgebungen für den Zugriff auf Daten und Funktionen bereitstellen, können Programme weitgehend hardwareunabhängig ausgeführt

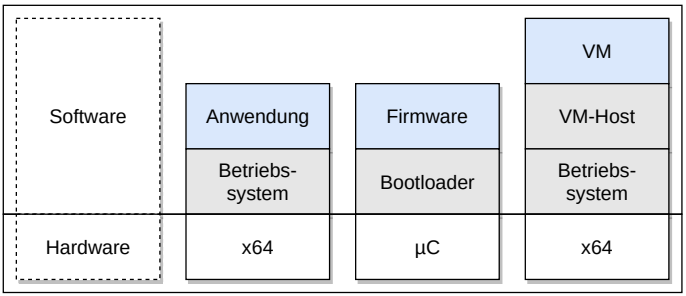


Abbildung A.4: Beispiele für Zielebenen, die für das Softwaredeployment interessant sind

werden. (National Institute of Standards & Technology 2002) Durch die von Betriebssystemen bereitgestellte Hardwareabstraktion werden Programme und Dateien portabel. Außer in anwendungsdedizierten Systemen ist die Verwendung eines Betriebssystems Standard, insbesondere im Bereich verbreiteter Systemarchitekturen (x86, x64, ARM et cetera). Die Verfügbarkeit von Betriebssystemen ist abhängig von den Systemarchitekturen. Sie unterscheiden sich in ihren technischen Ansätzen und ihrem Aufbau. Es existieren spezialisierte Betriebssysteme, insbesondere im Embedded- sowie Mikrocontrollerbereich. Teilweise wird dort das Betriebssystem beim Bau einer Anwendung über Bibliotheken in die Anwendung eingehängt.

A.2.1 Aufgaben und Bestandteile

Betriebssysteme implementieren Funktionen und Abläufe zu Grundaufgaben, die eigenständig oder auf Programmaufruf als Systemgrundlage durchgeführt werden. Dazu zählen:

- **Speicherverwaltung und -schutz:** Betriebssysteme allozieren und deallozieren voneinander isolierte Programmspeicher für Programmausführungen. In Abhängigkeit von der Programmanforderung bezüglich Größe und Geschwindigkeit wird Speicher im prozessoreigenen Zwischenspeicher (klein, sehr schnell), im Arbeitsspeicher (mittel bis groß, schnell) oder im Hauptspeicher (groß, langsam) reserviert.

- **Prozessverwaltung, Zeit- und Ausführungsplanung:** Betriebssysteme sind in der Lage mehrere Programme gleichzeitig zu verwalten. Dazu werden die einzelnen

Programmen in eigenen Prozessen verwaltet und die anstehenden Befehle dieser Prozesse, basierend auf Anforderung, der Rechenkernverfügbarkeit und der Priorität, abgearbeitet oder aufgeschoben (Ausführungsplanung, *Scheduling*). (National Institute of Standards & Technology 2002) Ein vorhersagbares Verhalten der Ausführungsplanung ist eine Anforderung, die ein echtzeitfähiges Betriebssystem erfüllen muss.

- **Hardwareabstraktion:** Betriebssysteme implementieren über hardwarespezifische Treiber Standardschnittstellen zu den physischen Teilen eines Rechners, damit Software standardisierte Befehle und Umgebungen zur Verfügung haben.

- **Dateiverwaltung:** Betriebssysteme strukturieren und steuern den Zugriff auf persistente Datenspeicher über eine Dateiverwaltung. Das Betriebssystem reserviert und blockiert Dateien bei Lese- und Schreibzugriff einzelner Programme. Auf Unixsystemen ist die Dateiverwaltung auch der Zugriffsweg auf bestimmte Betriebssystemfunktionen oder Hardwareteile (*Everything-as-a-file*). (Raymond 2003)

- **Kommunikation:** Betriebssysteme liefern Funktionen zur Kommunikation von Prozessen untereinander (Interprozesskommunikation) und zur Kommunikation über Systemschnittstellen. Es existieren drei relevante Funktionen für die Interprozesskommunikation: gemeinsamer Speicher (*Shared memory*), Verkettung von Prozessen über Ein- und Ausgaben (*Pipes*) sowie Bitsignale (*Signals*). Für andere, nicht direkt integrierte Prozesskommunikation oder externe Kommunikation kann der Betriebssystemkernel Schnittstellen wie TCP verwalten.

Für die Umsetzung der genannten Aufgaben muss ein Betriebssystem Softwarekomponenten implementieren. Zentrales Element eines Betriebssystems ist der Kernel, der als ausführendes Element absolute Kernfunktionen implementiert. Abhängig von der weiteren Betriebssystemarchitektur implementiert er alle weiteren notwendigen Funktionen selbst (monolithischer Kernel, beispielsweise Linux) oder wird durch funktionsfokussierte Elemente erweitert (Mikrokernel und Services, beispielsweise Windows NT).

Bedingung für die robuste Nutzung eines monolithischen Kernels wie bei Linuxsystemen ist das präemptive Multitasking, das die Unterbrechung von Prozessen zur Bearbeitung

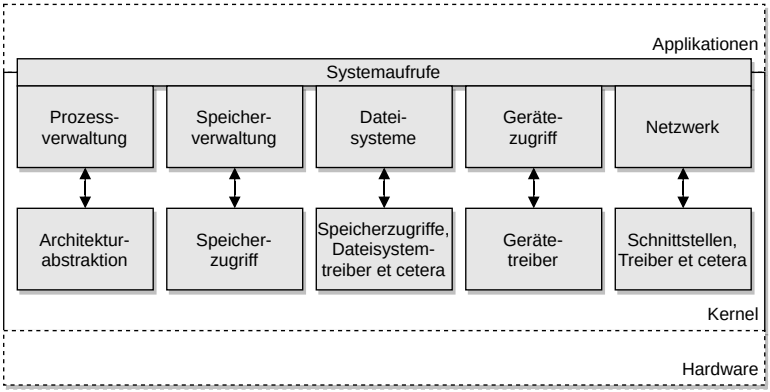


Abbildung A.5: Der monolithische Linux-Kernel zwischen Hardware und Applikationen (nach Corbet et al. 2005

anderer Prozesse zulässt. Dadurch kann auch eine logische Deadlock-Situation innerhalb einer Kernelfunktion keine Deadlock-Situation beim Gesamtsystem erzeugen.

In eigenständigen Anwendungen, verwaltet vom Kernel, Kernelmodulen oder im Kernel fest integriert, sind Softwarekomponenten für die Hardwareabstraktion und die darüberliegende Funktionalität (siehe Abbildung A.5). Jeweils in der Hardwareabstraktion und in der Bereitstellung der Funktionalität realisieren diese Komponenten Logik und Schnittstellen für die im vorherigen Abschnitt beschriebenen Aufgaben.

A.2.2 Taskparadigma und Bestandteile

Ein rechenfähiges und programmierbares System soll mehrere Aufgaben gleichzeitig bearbeiten. Wenn die jeweilige Aufgabenlogik nicht in einem starren Muster, sondern bedarfsgerecht bearbeitet werden soll, muss nach aktueller Anforderung zwischen den Aufgaben gewechselt werden können. Allgemeine Betriebssysteme agieren rechenzeitbasiert und teilen die verfügbare Rechenzeit in Zeitschlitze auf (Präemption), die abhängig von Prioritäten auf laufende Prozesse verteilt werden. Prozesse werden am Ende ihres Zeitschlitzes unterbrochen, gespeichert und nach Beendigung des Zeitschlitzes eines anderen Prozesses wieder aufgenommen (*Time-slicing*). Interruptfunktionen können die zeitorientierte Bearbeitung für kritische Prozesse unterbrechen.

Ereignisbasierte Betriebssysteme weisen Prozessen Rechenfähigkeit ausschließlich auf der Basis des Eintretens von Ereignissen wie Eingangsänderungen zu, wobei ein laufender Prozess unterbrochen wird, wenn ein eintretendes Ereignis einen höherpriorisierten Prozess auslöst. Der laufende Prozess wird dann gesichert, unterbrochen und nach Beendigung des höher priorisierten Prozesses weiterbearbeitet. Ereignisbasierte Betriebssysteme sind meistens dedizierte deterministische Systeme, allerdings gegenüber allgemeinen Betriebssystemen in verfügbarer Software und Änderbarkeit eingeschränkt. Einige ereignisbasierte Betriebssysteme werden, beispielsweise als Bibliothek, zur Bauzeit in eine Firmware eingebaut (zum Beispiel FreeRTOS) und sind kein eigenständiges ausführbares System. Applikation und Betriebssystem werden dann im Kompilierprozess als eine gemeinsame Softwarekomponente gebaut.

Im Betrieb werden Anwender- und Kernelraum unterschieden. Diese Räume unterteilen Speicher und Zugriffsrechte auf Prozess- und Systemteile. Je nach Betriebssystem können auch Betriebssystemkomponenten Teil des Anwenderraums sein. Softwareteile der verschiedenen Räume sind voneinander abgeschottet, um Integritäten zu wahren.

A.2.3 Echtzeitfähigkeit

Für die Umsetzung deterministischer Anwendungen mit digitalen technischen Systemen ist ein gleichbleibendes und vorhersagbares Zeit- und Zuverlässigkeitsverhalten notwendig. Wenn die technischen Systeme über Betriebssysteme verfügen, die im Betrieb dauernd wirken, werden zusätzliche Einflussfaktoren für Zeit- und Zuverlässigkeitsverhalten eingebracht, da Betriebssysteme weitere Softwareschichten zwischen Applikation und Hardware einführen. Durch diese Schichten entstehen zusätzliche logische Wege und Verarbeitungsstellen, die Auswirkungen auf die Einhaltung von Zeit- und Zuverlässigkeitsanforderungen haben. Diese Auswirkungen sind betriebssystemabhängig und applikations- und hardwarespezifisch: Applikationen greifen während ihrer Ausführung auf verschiedene Betriebssystemfunktionen zu, wodurch sich eine Abhängigkeit von der Betriebssystemimplementierung ergibt. Die Hardwareabstraktion des Betriebssystems greift auf Funktionen der darunterliegenden Hardware zu, weswegen die Hardwareabstraktion abhängig von der Hardware variieren kann. Die einzuhaltenden Anforderungen sind die gleichen wie bei Systemen ohne

Betriebssystem (siehe Abschnitt 2.1.3), zu denen insbesondere die Einhaltung von maximalen Vorgaben bei Antwortzeiten und deren Schwankungen gehören (siehe Abbildung 2.1). Betriebssysteme können für deterministische Aufgaben über Änderungen an Betriebssystemkomponenten angepasst werden. Dazu müssen unter anderem Prozessverwaltung, Zeit- und Ausführungsplanung, Prozesspriorisierung sowie Kommunikationsimplementierungen angepasst werden, abhängig auch davon, ob es sich um ein einkompiliertes, festes Betriebssystem handelt (wie beispielsweise bei AUTOSAR-CP) oder um ein kernelbasiertes, veränderbares Betriebssystem (wie beispielsweise übliche Linuxsysteme). Einkompilierte Echtzeitbetriebssysteme können eine festgelegte Ausführungsplanung besitzen, die zur Laufzeit nicht oder nur bedingt geändert wird. In kernelbasierten Echtzeitbetriebssystemen ist die Ausführungsplanung Kernelaufgabe (Serino & Cheng 2020). Beispiele für universelle einkompilierte Echtzeitbetriebssysteme sind VxWorks und FreeRTOS (Nikolov et al. 2021). Diese Betriebssysteme sind für den Betrieb auf eingebetteten Systemen ausgelegt, mit großer Hardwarenähe und niedrigem Ressourcenbedarf (Nikolov et al. 2021) sowie mit größtmöglicher Leistung auf einer technischen Plattform (Nakano et al. 2021). Dazu gehören auch SPS und IoT-Systeme. Sie sind keine verbreiteten Standardbetriebssysteme und benötigen spezialisiertes Wissen. Die allgemeine technische Unterstützung ist schwächer als bei Standardbetriebssystemen und nur einzelne Technologien von Standardsystemen, beispielsweise Dateisysteme und Kommunikationsprotokollen, werden direkt unterstützt (Serino & Cheng 2020). Die interoperable, plattformunabhängige Softwareentwicklung erfordert bei diesen Betriebssystemen besondere Entwicklungs- und Modellierungsansätze (beispielsweise Gomes & Baunach 2018) und ist nicht etabliert. Bei kernelbasierten Betriebssystemen ist aufgrund der Systemuniversalität die Komplexität der notwendigen Anpassungen größer. Da die Applikationsausführung im Betrieb variieren kann und Einflüsse durch Parallelprozesse, Betriebssystem und weitere Vorgänge nicht vorab bekannt und dadurch nicht in hinreichender Qualität modellierbar sind, ist zudem eine theoretische Verhaltensermittlung nicht ausreichend (Brand et al. 2021). Brand et al. 2021 stellten einen Ansatz zur Zeitverhaltensanalyse vor, der Nebeneffekte indirekt einbezieht und als Messgrundlage die Anzahl der vergangenen internen Systemzeit-Perioden (*System tick*, kleinste Einheit der Systemzeitgeber) nutzt. Zur Belastung wird ein festgelegter Satz an Prozessen durchgeführt, deren Ereignisse zeitlich erfasst werden. Die Festlegung einer bekannten

Aufgabenlast und die gleichzeitige Beobachtung des Systemverhaltens ist auch nach der DIN 66273 anwendbar, um die Leistungsfähigkeit datenverarbeitender Systeme bewerten zu können.

Linux als Echtzeitbetriebssystem

Zur Echtzeitbefähigung Linux' ist ein echtzeitfähiger Kernel nötig. Bereitgestellt wird dieser durch eine Modifikation des Standard-Kernels (Kernelpatch), den Ersatz durch einen besonderen Echtzeitkernel oder durch die Ergänzung des Standard-Kernels mit einem weiteren echtzeitfähigen Kernel, der parallel betrieben wird (National Institute of Standards & Technology 2002; Wei 2021). Standard-Linuxsysteme sind nicht deterministisch. Linux ist für die Weiterentwicklung zum deterministischen System interessant, weil es ausreichende Grundfähigkeiten bereitstellt, erweiterbar und der Quellcode allgemein verfügbar ist und eine große Verbreitung und Unterstützung vorhanden sind. (Nikolov et al. 2021; Serino & Cheng 2020)

Die Modifikation des Standard-Kernels (*PREEMPT_RT-Patch*) ist technisch am einfachsten umzusetzen. Bestandteile sind ab der Kernelversion 2.6 in Linux integriert (Nikolov et al. 2021). Die Modifikation führt präemptibles Multitasking ein, eine Unterform des standardmäßigen präemptiven Multitaskings. Die vorgehaltenen Zeitschlitze, in denen Befehle abgearbeitet werden, erfassen hier auch Betriebssystemfunktionen. Betriebssysteminterne Rechenaufgaben werden dadurch im Gegensatz zum präemptiven Multitasking nicht gegenüber Applikationsrechenaufgaben priorisiert. Rechenaufgaben aus aufgesetzten Applikationen können dadurch nicht durch Betriebssystemaufgaben blockiert werden. (Wei 2021) Wesentliche Anpassungen eines Linux-Systems für eine Echtzeitfähigkeit betreffen (Benra & Halang 2009; Wei 2021):

- **Ausführungsplanung:** Präemptible Ausführung von Rechenaufgaben
- **Priorisierung:** Besonderer Umgang mit höher- und niedriger priorisierten Tasks
- **Zeitgeber:** Gesamtes System auf hochgenaue Zeitgeber bezogen
- **Umgang mit Unterbrechungen:** Interrupts eingeschränkt oder unterbunden

Verglichen mit dedizierten und insbesondere einkompilierten Echtzeitsystemen können Standardbetriebssysteme wie Linux auf derselben technischen Plattform nicht dieselben

Reaktionsgeschwindigkeiten erzielen, ermöglichen jedoch eine breite Wiederverwertung existierender Software, zudem sind Standardbetriebssysteme meist bereits mehrkernfähig im Gegensatz zu hardwarenahen Frameworks wie beispielsweise AUTOSAR. (Nikolov et al. 2021; Serino & Cheng 2020)

B Bewertung der technischen Frameworks

Es wurden fünf relevante technische Frameworks betrachtet, die als Grundlage für verteilte Systeme in der modernen Industrieautomatisierung dienen können. Diese Frameworks sind die normierte IEC 61499, Field Device Integration, die Verwaltungsschale des RAMI4.0, das DOME-System sowie AUTOSAR aus dem Automotive-Bereich. Entscheidend für die Auswahl war die technische Relevanz für die Automatisierung, die Verfügbarkeit von Implementierungen oder ausreichenden technischen Beschreibungen und die Möglichkeit mit ihnen verteile Systeme umzusetzen.

B.1 IEC 61499

Die IEC 61499 ist ein modellbasiertes Programmiersystem für Automatisierungsanwendungen auf einzelnen oder verteilten Systemen. Die Entwicklung eines Automatisierungssystems mit der IEC 61499 generiert ein gemeinsames Programmmodell, das im Anschluss entsprechend eines zeitgleich definierten Verteilungsmodells auf die Geräte des Automatisierungssystems verteilt wird. Verwaltung und Wartung der Geräte bleiben unberücksichtigt. Die Programmierung erfolgt graphisch und ereignisbasiert über die Verknüpfung von Funktionsblöcken (FB), die Daten, Funktionen und Automaten beinhalten können. Die IEC 61499 verbirgt einen Großteil der Programmierlogik hinter FB. Deren interne Daten, Abläufe und Zustände sind nicht transparent. Eine explizite Zustandsdefinition für Teile oder das gesamte physische System ist nicht möglich. Durch die graphische Definition besteht die Gefahr unübersichtlicher und komplexer Programme. Tatsächlicher Ablauf und Zeitverhalten einer IEC-61499-Anwendung sind schwer bestimmbar. (Thramboulidis 2013)

Die IEC 61499 sieht keine Variablen in einem Steuerungsprogramm vor. Daten der Geräte sind lokal und nur innerhalb von FB vorhanden. Eine Datenbereitstellung benötigt somit eine Implementierung in einem FB. Die vollständige Abbildung eines verteilten Systems

erfordert somit erheblichen Aufwand. Der Einsatz setzt voraus, dass eine implementierungskompatible Laufzeitumgebung auf allen Geräten vorhanden ist. Zwar kann aus Programm und Laufzeitumgebung heraus mit anderen Systemen kommuniziert werden, es müssen aber anpassbare Kommunikations-FB vorhanden sein oder implementiert werden. Da auf dem Programmiergerät ausschließlich modelliert, aber nicht kompiliert wird, müssen die FB den Laufzeitumgebungen bekannt sein.

B.2 Field Device Integration

Die Field Device Integration (FDI) ist ein nicht prozessbezogenes Framework zur Integration von Feld- und Steuerungsgeräten sowie unterstützenden Serversystemen. FDI wirkt, bezogen auf Struktur und Topologie einer Gesamtmenge an Automatisierungskomponenten und -systemen, in einer Produktionsumgebung und erlaubt die Geräteadministration und Softwareverteilung für alle FDI-fähigen Geräte sowie die Erweiterung der Automatisierungsgeräte mit einem zentralen FDI-Server, der die einzelnen Geräte administriert und mit zentralen Services erweitert. FDI führt ein Paketsystem für die Softwareverteilung ein, wobei diese nicht an Anwendungen oder Programme gebunden sind. Die Verteilung kann einzeln und gerätebezogen erfolgen. FDI erzeugt keine verteilten Systeme auf der logischen Ebene, sondern nur auf der technischen und ermöglicht vor allem die Gerätebetreuung, deren Struktur aufgreifbar ist. Für die Implementierung von Programmen oder Softwareteilen macht FDI außer der Anforderungseinhaltung der Softwareverteilungsfunktionen keine Festlegungen und definiert keine Entwicklungs- oder Gestaltungsvorgaben.

Zwar kann über die über FDI hergestellte Integration über OPC UA ein prozessbezogener Datenaustausch realisiert werden, allerdings zielt FDI nicht auf die Gestaltung von Automatisierungslogik und ihre Instanziierung ab. Eine FDI-Instanz kann eine zentrale, OPC-UA-basierte und persistente Datenhaltung über den FDI-Server bereitstellen, über die Parameter im System sowie Kommunikation realisiert werden können, die aber nicht auf eine deterministische Prozesskommunikation ausgelegt sind.

B.3 Verwaltungsschale

Die VWS ist eine objektorientierte Softwarestruktur mit verschiedenen Ausprägungen (passiv, passiv mit API, aktiv), die als Standardformat für die Repräsentation und wahlweise die interne Softwarestruktur digitaler Komponenten herangezogen werden kann. Die VWS ist im Normierungsprozess und es fehlt zum Erstellungszeitpunkt dieser Arbeit eine offizielle Implementierung der Verwaltungsschale. Die in der ersten Normierung beschriebene Struktur der VWS kann aber für eigene Implementierungen und Repräsentationsformate herangezogen werden, vor allem für aktive Verwaltungsschalen auf Geräteebene. Die wesentlichen relevanten Punkte aktiver VWS sind:

- Die VWS ist eine Softwarehülle mit Selbstbeschreibungsfunktion und der Fähigkeit zur Anbindung an Plattformen oder übliche, I4.0-verstandene Kommunikationsschichten.
- Softwaremodule können als Teilmodelle in die VWS eingehängt werden.
- Die VWS kümmert sich bei Notwendigkeit um die Instanziierung und Verwaltung der Teilmodelle und stellt ein allgemeines Datenmodell bereit, auf dem statische Daten liegen und auf das Daten abgelegt werden können.
- Die VWS wird anhand einer Konfiguration erstellt und installiert.

Das RAMI4.0 und die VWS machen keine Festlegungen hinsichtlich eines Administrations- oder Konfigurationssystems. Es existiert kein Standardsystem zur Erstellung, Konfiguration oder Administration einer VWS selbst, noch zur Erstellung, Konfiguration oder Administration einer Menge von VWS oder ihrer möglichen Zusammenhänge. Eine Systemdefinition für ein verteiltes System kann durch die Kaskadierung oder die Zusammenfassung von VWS in einer übergeordneten VWS stattfinden. Dieser Vorgang und die Beschreibung von Zusammenhängen zwischen den Unter-VWS in der Gesamt-VWS sind nicht festgelegt. Die Integration von VWS muss manuell erfolgen und ist nicht geführt. Die Definition logischer Zusammenhänge oder Programmabläufe mit VWS oder VWS-Teilmodellen ist ebenfalls nicht geführt oder gesteuert.

Die VWS ist ein Abbildungs- und einfaches Integrationssystem für Automatisierungssysteme im Sinne eines digitalen Zwillings, das den Fokus auf abgeschlossene Komponenten legt

und nicht auf Systeme, die spezifisch für eine Anwendung erstellt werden. Sie bietet keine technischen Grundlagen oder Umgebungen für die Implementierung von Automatisierungslogik. Sie begrenzt die Formatvielfalt von Systemen und Komponenten. Ihre Klassenstruktur ist ausdefiniert, wobei die genaue Einhaltung bei Implementierungen nicht notwendig ist, wenn die Implementierung zur reinen Beschreibungserstellung verwendet werden soll.

B.4 Distributed Object Model Environment

DOME beschreibt ein modell- und objektorientierte System zur Modellierung und Instanziierung von Automatisierungsgeräten und verteilten Systemen in der Automatisierung. DOME führt sechs Klassen ein, deren Objekte in sechs Schichten aufeinander aufbauen und hierarchisch eine Automatisierungsanwendung modellieren. Im in der Hierarchie an der Spitze stehenden Anwendungsmodell sind einzelne Geräte als sogenannte Nodes eingehängt, auf denen Prozessobjekte ausgeführt werden. In den Prozessobjekten sind über Kontext- und funktionale Objekte Automatisierungsabläufe implementiert. Verbindungen zwischen den Objekten werden über Port- und Verknüpfungsobjekte modelliert.

Die Ausführung der definierten und einem Gerät zugewiesenen Anwendungsteile ist auf Laufzeitumgebungen ausgelegt. Es sind keine standardisierten Formate zum internen Datenaustausch definiert. Ähnlich wie die IEC 61499 implementiert DOME ein System zur modellbasierten Entwicklung einer Automatisierungsanwendung, deren Bestandteile auf mehrere eingebundene Geräte verteilt werden kann. Alle Geräte müssen DOME vollständig implementieren, um in der modellierten Automatisierungsanwendung eingebettet werden zu können. Schnittstellen zu externen Systemen sind nicht implementiert. Die Implementierungen von Schnittstellen als Objekte wird nicht erläutert. Da die Ausführung innerhalb des Systems nicht daten-, sondern aufruforientiert in Sequenzen funktioniert, ist die Verbindung mit den datenorientierten Kommunikationsarchitekturen der Automatisierung erschwert. Repräsentationen gegenüber Plattformen oder anderen externen Systemen in Form von Systembeschreibungen sind nicht berücksichtigt. DOME ist in der Anwendung auf die Nutzung einer technischen Plattform mit einem deterministischen multiprozessfähigen Betriebssystem ausgelegt. Das System verfügt über keine eingebetteten Funktionen zur

Verhaltensbestimmung oder Ausführungssteuerung. Letztere kann implizit über Ausführungssequenzen der Objekte sowie über von Objektgruppen geteilten Steuerungsobjekten programmatisch implementiert werden. Für die Einhaltung von Zeitanforderungen wird auf die Nutzung der Priorisierungsfunktionen des Betriebssystems verwiesen.

B.5 AUTOSAR

AUTOSAR beschreibt zwei technische Frameworks zur Umsetzung von Softwarearchitekturen auf Steuergeräten und Steuergerätverbünden im Automotive-Bereich. Die ältere Classic Platform (CP) sieht ähnlich der IEC-61499-Norm eine modellbasierte Definition verteilter Steuerungssysteme vor. Es wird bei der Entwicklung eines Steuerungssystems ein Systemmodell entworfen, das die Verteilungs- und Integrationsstruktur eines oder mehrerer Steuerungsprogramme beschreibt. Anwendungen werden aus Softwarekomponenten zusammengestellt und es wird festgelegt, wie diese Komponenten auf die Geräte des Systems verteilt werden. Es wird ein Kommunikationsmodell mithilfe einer virtuellen Middleware beschrieben, die als Modellierungssystem dient und beim Programmbau in feste Verbindungen aufgelöst wird. Die Konfigurierbarkeit bei AUTOSAR-CP existiert ausschließlich zur Entwicklungszeit. AUTOSAR-CP wurde mehrfach für die Industrieautomatisierung adaptiert. Die beteiligten Systeme werden zur Entwicklungszeit festgelegt. Die neuere Adaptive Platform (AP) führt eine SoA-ähnliche Struktur ein, bei der eine flexible Kommunikationsstruktur und instanziierbare Softwarekomponenten eingeführt werden. Die Modellierungsstruktur wird aus AUTOSAR-CP übernommen.

AUTOSAR führt kein Betriebssystem und keine Datenhaltung ein. Die Softwarekomponenten sind voneinander abgetrennt und implementieren allgemeine Funktionsgruppen wie EA-Treiber oder anwendungs- oder systemgebundene Funktionsbündel. Bedingt durch die Komplexität und technische Ausrichtung der Frameworks sind AUTOSAR-CP und -AP auf die einmalige oder seltene Konfiguration von leistungsschwachen Geräten ausgelegt, deren Zusammensetzung sich zur Lebenszeit des Gesamtsystems nicht ändert und die für große Produktstückzahlen mit wenigen programmatischen Änderungen ausgelegt sind.

C Kernelemente der ISCI-Bibliothek

```
//isci/Daten/Dateneintrag.cs
namespace isci.Daten
{
  public class Dateneintrag
  {
    public string Identifikation;
    [Newtonsoft.Json.JsonIgnore]
    private System.Threading.Mutex mutex;
    public object value;
    [Newtonsoft.Json.JsonIgnore]
    public bool aenderung;
    [Newtonsoft.Json.JsonIgnore]
    public string path;
    [Newtonsoft.Json.JsonIgnore]
    public bool write_flag;
    public Datentypen type;
    public bool istListe;
    public UInt16 listeDimensionen;
    public string parentEintrag;

    public Dateneintrag()
    {

    }

    public Dateneintrag(string Identifikation)
    {
      this.Identifikation = Identifikation;
    }

    public void Start()
    {
      ImSpeicherAnlegen();
      MitSpeicherVerknuepfen();
      Lesen();
    }

    public Dateneintrag(Newtonsoft.Json.Linq.JObject jObject)
    {
      Identifikation = jObject.SelectToken("Identifikation").ToString();
      type = jObject.SelectToken("type").ToObject<Datentypen>();

      try {
        var value = jObject.SelectToken("Wert");
        AusJToken(value);
      } catch {

      }
    }

    public static Dateneintrag DatafieldTyped(Newtonsoft.Json.Linq.JObject
      ↪ jObject)
    {
      var df_ = new Dateneintrag(jObject);
```

```
54
55        switch(df_.type)
56        {
57          case Datentypen.UInt8: return jObject.ToObject<dtUInt8>();
58          case Datentypen.UInt16: return jObject.ToObject<dtUInt16>();
59          case Datentypen.UInt32: return jObject.ToObject<dtUInt32>();
60          case Datentypen.Int8: return jObject.ToObject<dtInt8>();
61          case Datentypen.Int16: return jObject.ToObject<dtInt16>();
62          case Datentypen.Int32: return jObject.ToObject<dtInt32>();
63          case Datentypen.Float: return jObject.ToObject<dtFloat>();
64          case Datentypen.Double: return jObject.ToObject<dtDouble>();
65          case Datentypen.Bool: return jObject.ToObject<dtBool>();
66        }
67
68        return null;
69      }
70
71      public void ImSpeicherAnlegen()
72      {
73        try
74        {
75          var dir = System.IO.Path.GetDirectoryName(path);
76          if (!System.IO.Directory.Exists(dir))
77          System.IO.Directory.CreateDirectory(dir);
78        } catch {
79
80        }
81
82        try
83        {
84          mutex = new System.Threading.Mutex(false, (path +
            ↪  "_mutex").Replace('/', '.'));
85        } catch {
86
87        }
88
89        try
90        {
91          if (!System.IO.File.Exists(path))
92          {
93            System.IO.File.Create(path).Close();
94            Schreiben();
95          }
96        }
97        catch
98        {
99          Console.WriteLine("ImSpeicherAnlagen fehlgeschlagen: " +
            ↪  this.Identifikation);
100       }
101     }
102
103     public void MitSpeicherVerknuepfen()
104     {
105       try
106       {
107         mutex = System.Threading.Mutex.OpenExisting((path +
            ↪  "_mutex").Replace('/', '.'));
108       }
109       catch
110       {
111         Console.WriteLine("MitSpeicherVerknuepfen fehlgeschlagen: " +
            ↪  this.Identifikation);
```

```
112         }
113     }
114
115     public void MutexBlockierenSynchron()
116     {
117       try
118       {
119         mutex.WaitOne();
120       }
121       catch
122       {
123         Console.WriteLine("MutexBlockierenSynchron fehlgeschlagen: " +
          ↪  this.Identifikation);
124       }
125     }
126
127     public void MutexFreigeben()
128     {
129       mutex.ReleaseMutex();
130     }
131
132     public void Lesen()
133     {
134       MutexBlockierenSynchron();
135
136       var stream = new System.IO.FileStream(path, System.IO.FileMode.Open);
137       var reader = new System.IO.BinaryReader(stream);
138
139       try {
140         LesenSpezifisch(reader);
141       } catch {
142         Console.WriteLine("Lesen fehlgeschlagen: " + this.Identifikation);
143       }
144
145       reader.Close();
146       stream.Close();
147
148       MutexFreigeben();
149     }
150
151     public virtual void LesenSpezifisch(System.IO.BinaryReader reader)
152     {
153
154     }
155
156     public virtual void SchreibenSpezifisch(System.IO.BinaryWriter writer)
157     {
158
159     }
160
161     public void Schreiben()
162     {
163       MutexBlockierenSynchron();
164
165       var stream = new System.IO.FileStream(path, System.IO.FileMode.Truncate);
166       var writer = new System.IO.BinaryWriter(stream);
167
168       SchreibenSpezifisch(writer);
169
170       writer.Close();
171       stream.Close();
```

```
172
173       MutexFreigeben();
174     }
175
176     public virtual string Serialisieren()
177     {
178       return string.Empty;
179     }
180
181     public virtual void AusString(System.String s) {}
182
183     public virtual void AusJTokenSpezifisch(Newtonsoft.Json.Linq.JToken token)
         ↪ {}
184
185     public void AusJToken(Newtonsoft.Json.Linq.JToken token)
186     {
187       AusJTokenSpezifisch(token);
188     }
189
190     public Newtonsoft.Json.Linq.JToken NachJToken()
191     {
192       return NachJTokenSpezifisch();
193     }
194     public virtual Newtonsoft.Json.Linq.JToken NachJTokenSpezifisch() { return
         ↪ null; }
195
196     public string getName()
197     {
198       if (!this.Identifikation.StartsWith("ns=3;s=")) return Identifikation;
199       if (!this.Identifikation.Contains('.')) return
           ↪ this.Identifikation.Substring(7);
200
201       return this.Identifikation.Substring(this.Identifikation.LastIndexOf('.')
           ↪ + 1);
202     }
203
204     public string getFullname()
205     {
206       if (!this.Identifikation.StartsWith("ns=3;s=")) return Identifikation;
207       else return this.Identifikation.Substring(7);
208     }
209
210     public int getNamespace()
211     {
212       var splits = new string[2];
213       splits[0] = "ns=";
214       splits[1] = ";";
215       var teile = this.Identifikation.Split(splits,
           ↪ StringSplitOptions.RemoveEmptyEntries);
216       return int.Parse(teile[0]);
217     }
218
219     public string getDatenmodell()
220     {
221       if (!this.Identifikation.StartsWith("ns=3;s=")) return "";
222       if (!this.Identifikation.Contains('.')) return
           ↪ this.Identifikation.Substring(7);
223
224       var teile = this.Identifikation.Split(new char[]{'.'},
           ↪ StringSplitOptions.RemoveEmptyEntries);
225       if (teile.Length > 1) return teile[0];
```

```
226       else return "";
227     }
228
229     public string getTop()
230     {
231       if (this.parentEintrag != null) return this.parentEintrag;
232
233       if (!this.Identifikation.Contains('.')) return this.getDatenmodell();
234
235       var top = this.Identifikation.Substring(0,
          ↪  this.Identifikation.LastIndexOf('.'));
236
237       return top;
238     }
239   }
240 }
241
242 //isci/Daten/Datenmodell.cs
243 namespace isci.Daten
244 {
245   public class Datenmodell : Header
246   {
247     public string Stand;
248     public ListeDateneintraege Dateneinträge;
249     public VerweislisteDateneintraege Links;
250
251     public Datenmodell(string Identifikation, ListeDateneintraege Dateneinträge
        ↪  = null) : base(Identifikation)
252     {
253       if (this.Dateneinträge == null)
254       {
255         this.Dateneinträge = (Dateneinträge == null ? new ListeDateneintraege()
            ↪  : Dateneinträge);
256       }
257
258       if (this.Links == null) this.Links = new VerweislisteDateneintraege();
259     }
260
261     public static Datenmodell AusDatei(string path)
262     {
263       var file = System.IO.File.ReadAllText(path);
264       var datamodel_ = Newtonsoft.Json.Linq.JObject.Parse(file);
265
266       var datamodel = AusJObject(datamodel_);
267
268       return datamodel;
269     }
270
271     public override string ToString()
272     {
273       var settings = new Newtonsoft.Json.JsonSerializerSettings();
274       settings.Converters.Add(new
          ↪  Newtonsoft.Json.Converters.StringEnumConverter());
275       settings.Formatting = Newtonsoft.Json.Formatting.Indented;
276       var ret = Newtonsoft.Json.JsonConvert.SerializeObject(this, settings);
277       return ret;
278     }
279
280     public void ToFile(string path)
281     {
282       var file = this.ToString();
```

```
283       System.IO.File.WriteAllText(path, file);
284     }
285
286     public static Datenmodell AusJObject(Newtonsoft.Json.Linq.JObject
         ↪  datamodel_)
287     {
288       var dm = new
           ↪  Datenmodell(datamodel_.SelectToken("Identifikation").ToString());
289
290       var entries = datamodel_.SelectToken("Dateneinträge");
291
292       foreach (Newtonsoft.Json.Linq.JObject entry in entries)
293       {
294         var de = Dateneintrag.DatafieldTyped(entry);
295         dm.Dateneinträge.Add(de);
296       }
297
298       var link_entries = datamodel_.SelectToken("Links");
299
300       try {
301         dm.Links = link_entries.ToObject<VerweislisteDateneintraege>();
302       } catch {
303
304       }
305
306       var stand = datamodel_.SelectToken("Stand");
307
308       try {
309         dm.Stand = stand.ToObject<string>();
310       } catch {
311
312       }
313
314       return dm;
315     }
316
317     public void AddEvaluationSet(int size, string identifikation)
318     {
319       for(int i = 0; i < size; ++i)
320       {
321         Dateneinträge.Add(new dtInt32(0, identifikation + i.ToString()));
322       }
323     }
324
325     public void AddEvaluationLead(int size)
326     {
327       AddEvaluationSet(size, "evalLead_");
328     }
329   }
330 }
331
332 //isci/Daten/Datenstruktur.cs
333 namespace isci.Daten
334 {
335   public class Datenstruktur
336   {
337     public System.Collections.Generic.List<string> datenmodelle;
338     public System.Collections.Generic.Dictionary<string, Dateneintrag>
         ↪  dateneinträge;
339     private VerweislisteDateneintraege verweise;
340     public VerweislisteDateneintraegeAktiv verweiseAktiv;
```

```
341     public string pfad;
342
343     public Datenstruktur(string pfad)
344     {
345       datenmodelle = new System.Collections.Generic.List<string>();
346       dateneinträge = new System.Collections.Generic.Dictionary<string,
          ↪  Dateneintrag>();
347       verweise = new VerweislisteDateneintraege();
348       verweiseAktiv = new VerweislisteDateneintraegeAktiv();
349       this.pfad = pfad;
350     }
351
352     public void DatenmodellEinhängen(Datenmodell datenmodell)
353     {
354       try
355       {
356         foreach (var eintrag in datenmodell.Dateneinträge)
357         {
358           string ident = "ns=3;s=" + datenmodell.Identifikation + "." +
              ↪  eintrag.Identifikation;
359           eintrag.Identifikation = ident;
360           eintrag.path = pfad + "/" + eintrag.Identifikation;
361           dateneinträge.Add(ident, eintrag);
362         }
363         foreach (var eintrag in datenmodell.Links)
364         {
365           var Links_ = new ListeDateneintraege();
366           var key = eintrag.Key;
367           var value = new System.Collections.Generic.List<string>();
368           if (!key.StartsWith("ns="))
369           {
370             key = "ns=3;s=" + datenmodell.Identifikation + "." + key;
371           }
372           foreach (var untereintrag in eintrag.Value)
373           {
374             if (untereintrag.StartsWith("ns=")) value.Add(untereintrag);
375             else value.Add("ns=3;s=" + datenmodell.Identifikation + "." +
                ↪  untereintrag);
376           }
377           verweise.Add(key, value);
378         }
379         datenmodelle.Add(datenmodell.Identifikation);
380       } catch (Exception)
381       {
382
383       }
384     }
385
386     public void DatenmodellEinhängenAusDatei(string pfad)
387     {
388       var dm_ = Datenmodell.AusDatei(pfad);
389       DatenmodellEinhängen(dm_);
390     }
391
392     public void DatenmodelleEinhängenAusOrdner(string pfade, string excludeown
        ↪  = "")
393     {
394       if (pfade == null)
395       {
396         Console.WriteLine("DatenmodelleEinhängenAusOrdner: pfade ist NULL");
397         return;
```

```
398         }
399
400         var dms_ = System.IO.Directory.GetFiles(pfade);
401         foreach (var dm_ in dms_)
402         {
403           if (excludeown != "")
404           {
405             if (dm_.Contains(excludeown)) continue;
406           }
407           DatenmodellEinhängenAusDatei(dm_);
408         }
409       }
410
411       public void DatenmodelleEinhängenAusOrdnern(string[] pfade, string
          ↪ excludeown = "")
412       {
413         foreach (var pfad in pfade)
414         {
415           DatenmodelleEinhängenAusOrdner(pfad, excludeown);
416         }
417       }
418
419       public void VerweiseErzeugen()
420       {
421         foreach (var eintrag in verweise)
422         {
423           foreach (var subentry in eintrag.Value)
424           {
425             try {
426               verweiseAktiv.Add(dateneinträge[eintrag.Key],
                  ↪ dateneinträge[subentry]);
427             } catch {
428
429             }
430           }
431         }
432       }
433
434       public void Start()
435       {
436         foreach (var eintrag in dateneinträge)
437         {
438           eintrag.Value.Start();
439         }
440       }
441
442       public System.Collections.Generic.List<string> Lesen()
443       {
444         var result = new System.Collections.Generic.List<string>();
445
446         foreach (var eintrag in dateneinträge)
447         {
448           eintrag.Value.Lesen();
449           if (eintrag.Value.aenderung)
450           {
451             result.Add(eintrag.Key);
452           }
453         }
454
455         return result;
456       }
```

```
457
458     public bool AenderungVorhanden()
459     {
460       foreach (var eintrag in dateneinträge)
461       if (eintrag.Value.aenderung) return true;
462
463       return false;
464     }
465
466     public void AenderungenZuruecksetzen()
467     {
468       foreach (var eintrag in dateneinträge)
469       if (eintrag.Value.aenderung) eintrag.Value.aenderung = false;
470     }
471
472     public void
        ↪  AenderungenZuruecksetzen(System.Collections.Generic.List<string> liste)
473     {
474       foreach (var eintrag in liste)
475       if (dateneinträge[eintrag].aenderung)dateneinträge[eintrag].aenderung =
          ↪  false;
476     }
477
478     public bool Lesen(string Identifikation)
479     {
480       if (this.dateneinträge.ContainsKey(Identifikation))
481       {
482         var eintrag = this.dateneinträge[Identifikation];
483         eintrag.Lesen();
484         return eintrag.aenderung;
485       }
486       return false;
487     }
488
489     public void Schreiben()
490     {
491       foreach (var eintrag in dateneinträge)
492       {
493         eintrag.Value.Schreiben();
494       }
495     }
496
497     public void Schreiben(string Identifikation)
498     {
499       if (this.dateneinträge.ContainsKey(Identifikation))
500       {
501         var eintrag = this.dateneinträge[Identifikation];
502         eintrag.Schreiben();
503       }
504     }
505   }
506 }
507
508 //isci/Daten/Typisiert/dtBool.cs
509 namespace isci.Daten
510 {
511   public class dtBool : Dateneintrag
512   {
513     public dtBool(System.Boolean value, String Identifikation, String path =
        ↪  "") : base(Identifikation)
514     {
```

```
515       this.type = Datentypen.Bool;
516       this.value = value;
517       if (path != "") this.path = path;
518     }
519
520     public override void LesenSpezifisch(System.IO.BinaryReader reader)
521     {
522       var tmp = reader.ReadBoolean();
523       if (tmp != (System.Boolean)value)
524       {
525         value = tmp;
526         aenderung = true;
527       }
528     }
529
530     public override void SchreibenSpezifisch(System.IO.BinaryWriter writer)
531     {
532       writer.Write((System.Boolean)value);
533     }
534
535     public override string Serialisieren()
536     {
537       var s = value.ToString();
538       return s;
539     }
540
541     public override void AusString(System.String s)
542     {
543       value = System.Boolean.Parse(s);
544     }
545
546     public override void AusJTokenSpezifisch(Newtonsoft.Json.Linq.JToken token)
547     {
548       value = token.ToObject<System.Boolean>();
549     }
550
551     public static bool operator ==(dtBool left, dtBool right)
552     {
553       return (System.Boolean)left.value == (System.Boolean)right.value;
554     }
555
556     public static bool operator !=(dtBool left, dtBool right)
557     {
558       return (System.Boolean)left.value != (System.Boolean)right.value;
559     }
560
561     public static bool operator ==(dtBool left, System.Boolean right)
562     {
563       return (System.Boolean)left.value == right;
564     }
565
566     public static bool operator !=(dtBool left, System.Boolean right)
567     {
568       return (System.Boolean)left.value != right;
569     }
570   }
571 }
572
573 //isci/Anwendungen/Basis.cs
574 namespace isci.Anwendungen
575 {
```

```
576   public class Basis : Header {
577     public List<string> ReferenzierteModelle;
578
579     //Automatisierungsressource --> Name Konfiguration -->
580     //Identifikation Konfiguration
581     public Dictionary<string, Dictionary<string, string>> Konfigurationspakete;
582     //Automatisierungsressource --> Konfigurationselemente
583     public Dictionary<string, List<Konfigurationselement>>
        ↪  Konfigurationselemente;
584
585     //Modelle
586     public List<Datenmodell> Datenmodelle;
587     public List<Funktion> Funktionen;
588     public List<Ereignis> Ereignisse;
589   }
590 }
591
592 //isci/Anwendungen/Ereignis.cs
593 namespace isci.Anwendungen
594 {
595   public class Ereignis : Header {
596     public string Ausloeser;
597     public List<string> Elemente;
598   }
599 }
600
601 //isci/Beschreibung/Automatisierungsressource.cs
602 namespace isci.Beschreibung
603 {
604   public class Automatisierungsressource : Header
605   {
606     public string Handler;
607     public Dictionary<string, object> Devicebeschreibung;
608     public List<string> Anwendungen;
609     public Dictionary<string, Systemteil> Systemteile;
610   }
611 }
612
613 //isci/Beschreibung/Automatisierungssystem.cs
614 namespace isci.Beschreibung
615 {
616   public class Automatisierungssystem : Header
617   {
618     public List<string> Automatisierungsressourcen;
619     public string Anwendung;
620     public Dictionary<string, string> Module;
621     public Dictionary<string, string> Modulverteilung;
622     public ListeDateneintraege Dateneinträge;
623     public List<Ereignis> Ereignisse;
624     public List<Funktion> Funktionen;
625
626     public Automatisierungssystem() : base()
627     {
628
629     }
630
631     public Automatisierungssystem(string pfad) : base()
632     {
633       var list = System.IO.Directory.GetFiles(pfad, "*",
          ↪  System.IO.SearchOption.TopDirectoryOnly).ToList<string>();
634
```

```
635       foreach (var file in list)
636       {
637         var file_ = System.IO.File.ReadAllText(file);
638         var teil =
          ↪  Newtonsoft.Json.JsonConvert.DeserializeObject<Systemteil>(file_);
639         foreach (var m in teil.Module)
640         {
641           Module.Add(m.Key, m.Value);
642         }
643         Dateneinträge.AddRange(teil.Dateneinträge);
644         Ereignisse.AddRange(teil.Ereignisse);
645         Funktionen.AddRange(teil.Funktionen);
646       }
647     }
648   }
649 }
650
651 //isci/Konfiguration/Paket.cs
652 namespace isci.Konfiguration
653 {
654   public class Konfigurationspaket : Header {
655     public List<Konfigurationselement> Elemente;
656   }
657
658   public class Konfigurationsteil {
659     //Automatisierungsressource --> Name Konfiguration -->
660     //Identifikation Konfiguration
661     public Dictionary<string, Dictionary<string, string>> Konfigurationspakete;
662     //Automatisierungsressource --> Konfigurationselemente
663     public Dictionary<string, List<Konfigurationselement>>
      ↪  Konfigurationselemente;
664   }
665 }
```

Objekt C.1: Kernelemente der ISCI-Bibliothek

Verfügbar unter https://github.com/408b7f8b/isci.

D Implementierte Modulelemente auf ISCI-Basis

D.1 Modulskelett für die Modulerstellung

```
1  using System;
2  using System.Net.Sockets;
3  using System.Net;
4  using System.Linq;
5  using System.Collections.Generic;
6  using isci.Allgemein;
7  using isci.Daten;
8  using isci.Beschreibung;
9
10 namespace isci.modulbasis
11 {
12   public class Konfiguration : Parameter
13   {
14     public Konfiguration(string datei) : base(datei) {
15
16     }
17   }
18
19   class Program
20   {
21     static void Main(string[] args)
22     {
23       var konfiguration = new Konfiguration("konfiguration.json");
24
25       var structure = new Datenstruktur(konfiguration.OrdnerDatenstruktur);
26
27       var dm = new Datenmodell(konfiguration.Identifikation);
28       var example = new dtInt32(0, "example");
29       dm.Dateneinträge.Add(example);
30
31       var beschreibung = new Modul(konfiguration.Identifikation,
         ↪  "isci.modulbasis", new ListeDateneintraege(){example});
32       beschreibung.Name = "Modulbasis Ressource " +
         ↪  konfiguration.Identifikation;
33       beschreibung.Beschreibung = "Modulbasis";
34       beschreibung.Speichern(konfiguration.OrdnerBeschreibungen + "/" +
         ↪  konfiguration.Identifikation + ".json");
35
36       dm.Speichern(konfiguration.OrdnerDatenmodelle + "/" +
         ↪  konfiguration.Identifikation + ".json");
37
38       structure.DatenmodellEinhängen(dm);
39
         ↪  structure.DatenmodelleEinhängenAusOrdner(konfiguration.OrdnerDatenmodelle);
40       structure.Start();
41
```

```
42        var Zustand = new dtZustand(konfiguration.OrdnerDatenstruktur);
43        Zustand.Start();
44
45        while(true)
46        {
47          Zustand.Lesen();
48
49          var erfüllteTransitionen =
            ↪  konfiguration.Ausführungstransitionen.Where(a => a.Eingangszustand
            ↪  == (System.Int32)Zustand.value);
50          if (erfüllteTransitionen.Count<Ausführungstransition>() <= 0) continue;
51
52          if ((System.Int32)Zustand.value == 0)
53          {
54            example.value = 0;
55          } else {
56            example.value = 1;
57          }
58          structure.Schreiben();
59
60          Zustand.value =
            ↪  erfüllteTransitionen.First<Ausführungstransition>().Ausgangszustand;
61          Zustand.Schreiben();
62        }
63      }
64    }
65  }
```

Objekt D.1: Modulskelett für die Erstellung neuer Module

Verfügbar unter https://github.com/408b7f8b/isci.modulbasis.

D.2 Modul Datenverknüpfung

```
1  using System;
2  using System.Linq;
3  using isci.Allgemein;
4  using isci.Beschreibung;
5  using isci.Daten;
6
7  namespace isci.link
8  {
9    class Program
10   {
11     static void Main(string[] args)
12     {
13       var konfiguration = new Parameter("konfiguration.json");
14
15       var beschreibung = new Modul(konfiguration.Identifikation, "isci.link");
16       beschreibung.Name = "Link Ressource " + konfiguration.Identifikation;
17       beschreibung.Beschreibung = "Modul zur Verknüpfung von Dateneinträgen";
18       beschreibung.Speichern(konfiguration.OrdnerBeschreibungen + "/" +
         ↪  konfiguration.Identifikation + ".json");
19
20       var struktur = new Datenstruktur(konfiguration.OrdnerDatenstruktur);
21
22       var dm = new Datenmodell(konfiguration.Identifikation);
23
```

```
24        System.IO.File.WriteAllText(konfiguration.OrdnerDatenmodelle + "/" +
           ↪  konfiguration.Identifikation + ".json",
           ↪  Newtonsoft.Json.JsonConvert.SerializeObject(dm));
25
26        struktur.Datenmodell Einhängen(dm);
27        struktur.DatenmodelleEinhängenAusOrdner(konfiguration.OrdnerDatenmodelle,
           ↪  konfiguration.Identifikation);
28
29        struktur.VerweiseErzeugen();
30
31        struktur.Start();
32
33        var Zustand = new dtInt32(0, "Zustand", konfiguration.OrdnerDatenstruktur
           ↪  + "/Zustand");
34        Zustand.Start();
35
36        while(true)
37        {
38          Zustand.Lesen();
39          var erfüllteTransitionen = konfiguration.Aktivzustände.Where(a =>
             ↪  a.Eingangszustand == (System.Int32)Zustand.value);
40          if (erfüllteTransitionen.Count<Aktivzustand>() > 0)
41          {
42            struktur.Lesen();
43
44            foreach (var eintrag in struktur.verweiseAktiv)
45            {
46              if (eintrag.Key.aenderung)
47              {
48                foreach (var untereintrag in eintrag.Value)
49                {
50                  Console.WriteLine(System.DateTime.Now.ToString("O") + ": " +
                     ↪  eintrag.Key.Identifikation + " --> " +
                     ↪  untereintrag.Identifikation);
51                  untereintrag.value = eintrag.Key.value;
52                  eintrag.Key.aenderung = false;
53                  untereintrag.Schreiben();
54                }
55              }
56            }
57
58            Zustand.value =
               ↪  erfüllteTransitionen.First<Aktivzustand>().Ausgangszustand;
59            Zustand.Schreiben();
60          }
61        }
62      }
63    }
64  }
```

Objekt D.2: Modul zur Dateneintragverknüpfung

Verfügbar unter https://github.com/408b7f8b/isci.link.

D.3 Modul Abbilderstellung in Datenbank

```
1 using System;
2 using System.Linq;
3 using System.Diagnostics;
4 using System.Collections.Generic;
5 using InfluxDB.Client;
6 using InfluxDB.Client.Api.Domain;
7 using isci;
8 using isci.Allgemein;
9 using isci.Beschreibung;
10 using isci.Daten;
11
12 namespace isci.abbild
13 {
14   class Program
15   {
16     public class Konfiguration : Parameter
17     {
18       public string token;
19       public string adresse;
20       public string orgId;
21       public string pfad;
22       public int abbildlaenge;
23       public int pause;
24
25       public Konfiguration(string datei) : base(datei) { }
26     }
27
28     static async System.Threading.Tasks.Task Main(string[] args)
29     {
30       var konfiguration = new Konfiguration("konfiguration.json");
31
32       var beschreibung = new Modul(konfiguration.Identifikation,
         ↪  "isci.abbild");
33       beschreibung.Name = "Abbild Ressource " + konfiguration.Identifikation;
34       beschreibung.Beschreibung = "Modul zur Abbilderstellung gegen externe
         ↪  Datenbank";
35       beschreibung.Speichern(konfiguration.OrdnerBeschreibungen + "/" +
         ↪  konfiguration.Identifikation + ".json");
36
37       var structure = new Datenstruktur(konfiguration.OrdnerDatenstruktur);
38
         ↪  structure.DatenmodelleEinhängenAusOrdner(konfiguration.OrdnerDatenmodelle);
39       structure.Start();
40
41       var influxDBClient = InfluxDBClientFactory.Create(konfiguration.adresse,
         ↪  konfiguration.token);
42       influxDBClient.SetLogLevel(InfluxDB.Client.Core.LogLevel.None);
43
44       var writeOptions = WriteOptions
45       .CreateNew()
46       .BatchSize(50000)
47       .FlushInterval(10000)
48       .Build();
49
50       var writeApi = influxDBClient.GetWriteApi(writeOptions);
51       var orgname = (await influxDBClient.GetOrganizationsApi()
52       .FindOrganizationByIdAsync(konfiguration.orgId)).Name;
53
54       var bucketApi = influxDBClient.GetBucketsApi();
```

```
55        var bucketApiOrg = (await bucketApi.FindBucketsByOrgNameAsync(orgname));
56
57        var buckets = new List<string>();
58        foreach (var bucket in bucketApiOrg)
59        {
60          buckets.Add(bucket.Name);
61        }
62
63        if (!buckets.Contains(konfiguration.Anwendung))
64        {
65          bucketApi.CreateBucketAsync(new Bucket(name:konfiguration.Anwendung,
            ↪  orgID:konfiguration.orgId,
66          retentionRules:new List<BucketRetentionRules>(){
67            new BucketRetentionRules(BucketRetentionRules.TypeEnum.Expire, 0,
              ↪  3600)
68          })).Wait();
69        }
70
71        var abbild = new List<string>();
72
73        while(true)
74        {
75          structure.Lesen();
76          var dt = DateTimeOffset.UtcNow.ToUnixTimeMilliseconds().ToString();
77
78          foreach (var feld in structure.dateneinträge)
79          {
80            var l = $"{feld.Key},ressource={konfiguration.Ressource}
              ↪  value={feld.Value.Serialisieren()} {dt}";
81            abbild.Add(l);
82          }
83
84          if (abbild.Count > konfiguration.abbildlaenge)
85          {
86            writeApi.WriteRecords(abbild, bucket:konfiguration.Anwendung,
              ↪  org:konfiguration.orgId, precision:WritePrecision.Ms);
87            abbild.Clear();
88          }
89
90          System.Threading.Thread.Sleep(konfiguration.pause);
91        }
92      }
93    }
94  }
```

Objekt D.3: Modul zur laufenden Abbildung der Datenstruktur in eine InfluxDB-Datenbank

Verfügbar unter https://github.com/408b7f8b/isci.abbild.

D.4 Modul zur Zykluszeitermittlung

```
1  using System;
2  using System.Net.Sockets;
3  using System.Net;
4  using System.Linq;
5  using System.Collections.Generic;
6  using isci.Allgemein;
7  using isci.Daten;
8  using isci.Beschreibung;
9
10 namespace isci.zykluszeit
11 {
12   class Program
13   {
14     static void Main(string[] args)
15     {
16       var konfiguration = new Parameter("konfiguration.json");
17
18       var structure = new Datenstruktur(konfiguration.OrdnerDatenstruktur);
19
20       var dm = new Datenmodell(konfiguration.Identifikation);
21       var zykluszeit = new dtInt32(0, "zykluszeit");
22       dm.Dateneinträge.Add(zykluszeit);
23
24       var beschreibung = new Modul(konfiguration.Identifikation,
         ↪  "isci.zykluszeit", new ListeDateneintraege(){zykluszeit});
25       beschreibung.Name = "Zykluszeit Ressource " +
         ↪  konfiguration.Identifikation;
26       beschreibung.Beschreibung = "Modul zur Zykluszeitermittlung";
27       beschreibung.Speichern(konfiguration.OrdnerBeschreibungen + "/" +
         ↪  konfiguration.Identifikation + ".json");
28
29       dm.Speichern(konfiguration.OrdnerDatenmodelle + "/" +
         ↪  konfiguration.Identifikation + ".json");
30
31       structure.DatenmodellEinhängen(dm);
32       structure.Start();
33
34       var Zustand = new dtInt32(0, "Zustand", konfiguration.OrdnerDatenstruktur
         ↪  + "/Zustand");
35       Zustand.Start();
36
37       long curr_ticks = 0;
38
39       bool dbg = System.Environment.UserInteractive;
40
41       while(true)
42       {
43         System.Threading.Thread.Sleep(10);
44         Zustand.Lesen();
45
46         if (dbg)
47         {
48           System.Console.Clear();
49           System.Console.WriteLine("Zustand: " + (System.Int32)Zustand.value);
50           System.Console.WriteLine("Zykluszeit: " +
             ↪  (System.Int32)zykluszeit.value);
51         }
52
```

```
53          var erfüllteTransitionen =
             ↪  konfiguration.Ausführungstransitionen.Where(a => a.Eingangszustand
             ↪  == (System.Int32)Zustand.value);
54          if (erfüllteTransitionen.Count<Ausführungstransition>() > 0)
55          {
56            var curr_ticks_new = System.DateTime.Now.Ticks;
57            var ticks_span = curr_ticks_new - curr_ticks;
58            curr_ticks = curr_ticks_new;
59            zykluszeit.value = (System.Int32)(ticks_span /
               ↪  System.TimeSpan.TicksPerMillisecond);
60            zykluszeit.Schreiben();
61
62            Zustand.value =
               ↪  erfüllteTransitionen.First<Ausführungstransition>().Ausgangszustand;
63            Zustand.Schreiben();
64          }
65        }
66      }
67    }
68  }
```

Objekt D.4: Modul zur Ermittlung der Zykluszeit in einem Dateneintrag

Verfügbar unter https://github.com/408b7f8b/isci.zykluszeit.

D.5 Modul Datenbereitstellung via OPC UA

```
1  using System;
2  using System.Net.Sockets;
3  using System.Net;
4  using System.Linq;
5  using System.Collections.Generic;
6  using isci.Allgemein;
7  using isci.Daten;
8  using isci.Beschreibung;
9
10 namespace isci.opcuaserver
11 {
12   using System;
13   using System.Net;
14   using System.Linq;
15   using System.Collections.Generic;
16   using Opc.Ua;
17   using isci.Daten;
18
19   namespace isci.opcuaserver
20   {
21     public class OpcUaServer : Opc.Ua.Server.StandardServer
22     {
23       [Newtonsoft.Json.JsonIgnore]
24       private Datenstruktur Struktur;
25       public string name = "opcUaServer";
26       private ApplicationConfiguration application;
27
28       [Newtonsoft.Json.JsonIgnore]
29       private List<string> hinzugefügt = new List<string>();
30       [Newtonsoft.Json.JsonIgnore]
```

```
31      public Dictionary<string, object> puffer = new Dictionary<string,
         ↪  object>();
32      public bool puffer_mutex = false;
33
34      public void WriteLine(string content)
35      {
36        Console.WriteLine($"OPC-UA-Server {name}: {content}");
37      }
38
39      public static ApplicationConfiguration
         ↪  standardApplicationConfiguration(string name)
40      {
41        var application = new ApplicationConfiguration()
42        {
43          ApplicationName = name,
44          ApplicationUri = "urn:localhost:UA:" + name,
45          ApplicationType = ApplicationType.Server,
46          SecurityConfiguration = new SecurityConfiguration()
47          {
48            ApplicationCertificate = new CertificateIdentifier
49            {
50              StoreType = @"Directory",
51              StorePath = @"CertificateStores/MachineDefault",
52              SubjectName = Utils.Format(@"CN={0}, DC={1}", name,
                 ↪  System.Net.Dns.GetHostName())
53            },
54            TrustedPeerCertificates = new CertificateTrustList
55            {
56              StoreType = @"Directory",
57              StorePath = @"CertificateStores/UA Applications"
58            },
59            NonceLength = 32,
60            AutoAcceptUntrustedCertificates = true,
61            AddAppCertToTrustedStore = false
62          },
63          TransportConfigurations = new TransportConfigurationCollection(),
64          TransportQuotas = new TransportQuotas()
65          {
66            OperationTimeout = 600000,
67            MaxStringLength = 1048576,
68            MaxByteStringLength = 1048576,
69            MaxArrayLength = 65535,
70            MaxMessageSize = 4194304,
71            MaxBufferSize = 65535,
72            ChannelLifetime = 300000,
73            SecurityTokenLifetime = 3600000
74          },
75          ServerConfiguration = new ServerConfiguration
76          {
77            BaseAddresses = new StringCollection()
78            {
79              "opc.tcp://localhost:62541/" + name
80            },
81            SecurityPolicies = new ServerSecurityPolicyCollection
82            {
83              new ServerSecurityPolicy
84              {
85                SecurityMode = MessageSecurityMode.None,
86                SecurityPolicyUri =
                   ↪  "http://opcfoundation.org/UA/SecurityPolicy#None"
87              }
```

```
88            },
89            UserTokenPolicies = new UserTokenPolicyCollection
90            {
91              new UserTokenPolicy { TokenType = UserTokenType.Anonymous }
92            },
93            AlternateBaseAddresses = new StringCollection
94            {
95              "ocp.tcp://localhost:4840/" + name
96            },
97            DiagnosticsEnabled = true,
98            MaxSessionCount = 100,
99            MinSessionTimeout = 5000,
100           MaxSessionTimeout = 10000,
101           MaxBrowseContinuationPoints = 10,
102           MaxQueryContinuationPoints = 10,
103           MaxHistoryContinuationPoints = 100,
104           MaxRequestAge = 600000,
105           MinPublishingInterval = 100,
106           MaxPublishingInterval = 3600000,
107           PublishingResolution = 50,
108           MaxSubscriptionLifetime = 3600000,
109           MaxMessageQueueSize = 10,
110           MaxNotificationQueueSize = 100,
111           MaxNotificationsPerPublish = 1000,
112           MinMetadataSamplingInterval = 1000,
113           RegistrationEndpoint = new EndpointDescription()
114           {
115             EndpointUrl = "opc.tcp://localhost:4840",
116             Server = new ApplicationDescription()
117             {
118               ApplicationUri = "opc.tcp://localhost:4840",
119               ApplicationType = ApplicationType.DiscoveryServer,
120               DiscoveryUrls = new StringCollection(){
121                 "opc.tcp://localhost:4840"
122               }
123             },
124             SecurityMode = MessageSecurityMode.None,
125             SecurityPolicyUri = "",
126             UserIdentityTokens = new UserTokenPolicyCollection()
127             {
128
129             }
130           }
131         }
132       };
133
134       return application;
135     }
136
137     public OpcUaServer(ApplicationConfiguration application = null, string
        ↪  name = "opcUaServer")
138     {
139       this.name = name;
140       this.application = (application != null ? application :
          ↪  standardApplicationConfiguration(name));
141
142       var instanz = new
          ↪  Opc.Ua.Configuration.ApplicationInstance(this.application);
143
144       var tsk = instanz.CheckApplicationInstanceCertificate(false, 1024);
145       try {
```

```
146          tsk.Wait();
147        } catch {
148          var tsk2 = instanz.DeleteApplicationInstanceCertificate();
149          tsk2.Wait();
150          tsk = instanz.CheckApplicationInstanceCertificate(false, 1024);
151          tsk.Wait();
152        }
153
154        var resval = this.application.Validate(ApplicationType.Server);
155
156        resval.Wait();
157
158        var d = this.application.GetServerDomainNames();
159      }
160
161      public void Start()
162      {
163        WriteLine("Start");
164        this.Start(application);
165      }
166
167      public void Stopp()
168      {
169        WriteLine("Stopp");
170        this.Stop();
171      }
172
173      public void StrukturEintragen(Datenstruktur struktur)
174      {
175        WriteLine("StrukturEintragen");
176        Struktur = struktur;
177      }
178
179      public void nodeHinzufügen(AddNodesItemCollection coll, string key)
180      {
181        if (hinzugefügt.Contains(key)) return;
182
183        var node = new AddNodesItem();
184        node.NodeAttributes = new ExtensionObject(
185        new Opc.Ua.VariableAttributes()
186        {
187          AccessLevel = AccessLevels.CurrentReadOrWrite,
188          UserAccessLevel = AccessLevels.CurrentReadOrWrite,
189          SpecifiedAttributes = (uint) NodeAttributesMask.All,
190          WriteMask = (uint)AttributeWriteMask.None,
191          UserWriteMask = (uint)AttributeWriteMask.None,
192          MinimumSamplingInterval = 0,
193          Historizing = false
194        }
195        );
196
197        string top = "";
198
199        if (Struktur.dateneinträge.ContainsKey(key))
200        {
201          var item = Struktur.dateneinträge[key];
202
203          node.NodeClass = NodeClass.Variable;
204
205          node.BrowseName = item.getName();
206          node.RequestedNewNodeId = new ExpandedNodeId(item.getFullname(), 3);
```

```
207
208           top = item.getTop();
209         } else {
210           node.NodeClass = NodeClass.Object;
211           node.TypeDefinition = DataTypeIds.ObjectNode;
212
213           node.BrowseName = (key.Contains('.')) ?
               ↪  key.Substring(key.LastIndexOf('.')+1) :
               ↪  key.Substring(key.LastIndexOf('=')+1);
214           node.RequestedNewNodeId = new
               ↪  ExpandedNodeId(key.Substring(key.LastIndexOf('=')+1), 3);
215
216           top = "ns=0;i=85";
217         }
218
219         if (top != "")
220         {
221           if (!hinzugefügt.Contains(top) && top != "ns=0;i=85")
222           {
223             nodeHinzufügen(coll, top);
224             hinzugefügt.Add(top);
225           }
226           node.ParentNodeId = new ExpandedNodeId(top);
227         }
228
229         WriteLine($"Trage {node.BrowseName} als
             ↪  {node.RequestedNewNodeId.ToString()} ein ");
230         coll.Add(node);
231       }
232
233       public void DatenstrukturAufbauen()
234       {
235         WriteLine("DatenstrukturAufbauen");
236         var r = new AddNodesItemCollection();
237
238         Dictionary<string, ExpandedNodeId> Ordner_Nodes = new
             ↪  Dictionary<string, ExpandedNodeId>();
239
240         var Ausstehend = Struktur.dateneinträge.Keys.ToList();
241         Ausstehend.Sort();
242         var entfernen = new List<string>();
243
244         foreach (var existiert in hinzugefügt)
245         {
246           if (Ausstehend.Contains(existiert))
247           {
248             Ausstehend.Remove(existiert);
249           } else {
250             entfernen.Add(existiert);
251           }
252         }
253
254         if (entfernen.Count > 0)
255         {
256           var req_del = new RequestHeader();
257           var coll_del = new DeleteNodesItemCollection();
258           var res_del = new StatusCodeCollection();
259           var diag_del = new DiagnosticInfoCollection();
260
261           foreach (var loeschen in entfernen)
262           {
```

```
263             var item = new DeleteNodesItem();
264             item.NodeId = new NodeId(loeschen);
265             coll_del.Add(item);
266
267             hinzugefügt.Remove(loeschen);
268           }
269
270           this.DeleteNodes(req_del, coll_del, out res_del, out diag_del);
271         }
272
273         while (Ausstehend.Count > 0)
274         {
275           for (int i = Ausstehend.Count-1; i >= 0; --i)
276           {
277             var item = Struktur.dateneinträge[Ausstehend[i]];
278             this.nodeHinzufügen(r, Ausstehend[i]);
279
280             hinzugefügt.Add(Ausstehend[i]);
281             Ausstehend.RemoveAt(i);
282           }
283         }
284
285         var res = new AddNodesResultCollection();
286         var diag = new DiagnosticInfoCollection();
287         var req = new RequestHeader();
288         this.AddNodes(req, r, out res, out diag);
289       }
290
291       public List<Datenmodell> DatenmodelleLiefern()
292       {
293         return new List<Datenmodell>();
294       }
295
296       public void DatenwertVerbreiten(string feld)
297       {
298         var datenfeld = Struktur.dateneinträge[feld];
299         WriteInternal(datenfeld.Identifikation, datenfeld.value);
300       }
301
302       public void WriteInternal(string ident, object value)
303       {
304         var nodeId = NodeId.Parse(ident);
305         var node = ServerInternal.CoreNodeManager.GetLocalNode(nodeId);
306         if (node == null) return;
307         node.Write(Attributes.Value, new DataValue(new Variant(value)));
308       }
309
310       public override ResponseHeader Write(RequestHeader requestHeader,
          ↪ WriteValueCollection nodesToWrite, out StatusCodeCollection results,
          ↪ out DiagnosticInfoCollection diagnosticInfos)
311       {
312         results = new StatusCodeCollection();
313         diagnosticInfos = null;
314
315         ValidateRequest(requestHeader);
316
317         foreach (var item in nodesToWrite)
318         {
319           try
320           {
321             var node =
                ↪ ServerInternal.CoreNodeManager.GetLocalNode(item.NodeId);
```

```
322             node.Write(Attributes.Value, item.Value);
323
324             var id = item.NodeId.ToString();
325
326             while(puffer_mutex) { }
327
328             puffer_mutex = true;
329
330             if (!puffer.ContainsKey(id))
331             {
332               puffer.Add(id, new object());
333             }
334
335             var dateneintrag = Struktur.dateneinträge[id];
336             switch (dateneintrag.type)
337             {
338               case Datentypen.UInt8: puffer[id] =
                   ↪  item.Value.GetValue(typeof(System.Byte)); break;
339               case Datentypen.UInt16: puffer[id] =
                   ↪  item.Value.GetValue(typeof(System.UInt16)); break;
340               case Datentypen.UInt32: puffer[id] =
                   ↪  item.Value.GetValue(typeof(System.UInt32)); break;
341               case Datentypen.Int8: puffer[id] =
                   ↪  item.Value.GetValue(typeof(System.SByte)); break;
342               case Datentypen.Int16: puffer[id] =
                   ↪  item.Value.GetValue(typeof(System.Int16)); break;
343               case Datentypen.Int32: puffer[id]=
                   ↪  item.Value.GetValue(typeof(System.Int32)); break;
344               case Datentypen.Bool: puffer[id] =
                   ↪  item.Value.GetValue(typeof(System.Boolean)); break;
345               case Datentypen.Float: puffer[id] =
                   ↪  item.Value.GetValue(typeof(float)); break;
346               case Datentypen.Double: puffer[id] =
                   ↪  item.Value.GetValue(typeof(System.Double)); break;
347               case Datentypen.String: puffer[id] =
                   ↪  item.Value.GetValue(typeof(System.String)); break;
348             }
349
350             puffer_mutex = false;
351
352             results.Add(StatusCodes.Good);
353           } catch {
354             results.Add(StatusCodes.Bad);
355           }
356         }
357
358         return CreateResponse(requestHeader, StatusCodes.Good);
359       }
360
361       public override ResponseHeader AddNodes(
362       RequestHeader requestHeader,
363       AddNodesItemCollection nodesToAdd,
364       out AddNodesResultCollection results,
365       out DiagnosticInfoCollection diagnosticInfos)
366       {
367         results = null;
368         diagnosticInfos = null;
369
370         ValidateRequest(requestHeader);
371
372         foreach (var item in nodesToAdd)
```

```
373         {
374           WriteLine($"Füge Node {item.RequestedNewNodeId.ToString()} ein ");
375           //NodeClass.Object
376           if (item.NodeClass == NodeClass.Variable)
377           {
378             var ident = item.RequestedNewNodeId.ToString();
379
380             var struktur_item = Struktur.dateneinträge[ident];
381
382             var node = new VariableNode
383             {
384               NodeId = new NodeId(item.RequestedNewNodeId.Identifier,
                 ↪  item.RequestedNewNodeId.NamespaceIndex),
385               DisplayName = new LocalizedText(item.BrowseName.Name),
386               References = new ReferenceNodeCollection(){ },
387               BrowseName = item.BrowseName,
388               Description = new LocalizedText(""),
389               NodeClass = NodeClass.Variable,
390               AccessLevel =
                 ↪  ((Opc.Ua.VariableAttributes)item.NodeAttributes.Body).AccessLevel,
391               UserAccessLevel =
                 ↪  ((Opc.Ua.VariableAttributes)item.NodeAttributes.Body).UserAccessLevel,
392               Value = new Variant(Struktur.dateneinträge[ident].value)
393             };
394
395             switch (struktur_item.type)
396             {
397               case Datentypen.UInt8: node.DataType = DataTypeIds.Byte; break;
398               case Datentypen.UInt16: node.DataType = DataTypeIds.UInt16;
                 ↪  break;
399               case Datentypen.UInt32: node.DataType = DataTypeIds.UInt32;
                 ↪  break;
400               case Datentypen.Int8: node.DataType = DataTypeIds.SByte; break;
401               case Datentypen.Int16: node.DataType = DataTypeIds.Int16; break;
402               case Datentypen.Int32: node.DataType = DataTypeIds.Int32; break;
403               case Datentypen.Bool: node.DataType = DataTypeIds.Boolean; break;
404               case Datentypen.Float: node.DataType = DataTypeIds.Float; break;
405               case Datentypen.Double: node.DataType = DataTypeIds.Double;
                 ↪  break;
406               case Datentypen.String: node.DataType = DataTypeIds.String;
                 ↪  break;
407             }
408
409             ServerInternal.CoreNodeManager.AttachNode(node);
410             ServerInternal.CoreNodeManager.AddReference(new
                 ↪  NodeId(item.ParentNodeId.Identifier,
                 ↪  item.ParentNodeId.NamespaceIndex), ReferenceTypeIds.Organizes,
                 ↪  false, node.NodeId, false);
411           }
412           else if (item.NodeClass == NodeClass.VariableType)
413           {
414             var node = new VariableTypeNode
415             { }
416             ServerInternal.CoreNodeManager.AttachNode(node);
417           }
418           else if (item.NodeClass == NodeClass.Object)
419           {
420             var ident = item.RequestedNewNodeId.Identifier.ToString();
421             var node = new ObjectNode
422             {
423               NodeId = new NodeId(item.RequestedNewNodeId.Identifier,
                 ↪  item.RequestedNewNodeId.NamespaceIndex),
```

```
424                 DisplayName = new LocalizedText(item.BrowseName.Name),
425                 References = new ReferenceNodeCollection(){ },
426                 BrowseName = item.BrowseName,
427                 Description = new LocalizedText(""),
428                 NodeClass = NodeClass.Object
429               };
430
431               ServerInternal.CoreNodeManager.AttachNode(node);
                   ↪  item.ParentNodeId.NamespaceIndex), ReferenceTypeIds.Organizes,
                   ↪  false, node.NodeId, false);
432               ServerInternal.CoreNodeManager.AddReference(new
                   ↪  NodeId(item.ParentNodeId.Identifier,
                   ↪  item.ParentNodeId.NamespaceIndex), ReferenceTypeIds.Organizes,
                   ↪  false, node.NodeId, false);
433             }
434             else { }
435           }
436
437           return CreateResponse(requestHeader, StatusCodes.Good);
438         }
439       }
440     }
441
442     public class Konfiguration : Parameter
443     {
444       public Konfiguration(string datei) : base(datei) {
445
446       }
447     }
448
449     class Program
450     {
451       static void Main(string[] args)
452       {
453         var konfiguration = new Konfiguration("konfiguration.json");
454
455         var structure = new Datenstruktur(konfiguration.OrdnerDatenstruktur);
456
457         var beschreibung = new Modul(konfiguration.Identifikation,
             ↪  "isci.opcauserver", new ListeDateneintraege(){});
458         beschreibung.Name = "OPC-UA-Server Ressource " +
             ↪  konfiguration.Identifikation;
459         beschreibung.Beschreibung = "OPC-UA-Server";
460         beschreibung.Speichern(konfiguration.OrdnerBeschreibungen + "/" +
             ↪  konfiguration.Identifikation + ".json");
461
462
             ↪  structure.DatenmodelleEinhängenAusOrdner(konfiguration.OrdnerDatenmodelle);
463         structure.Start();
464
465         var Zustand = new dtZustand(konfiguration.OrdnerDatenstruktur);
466         Zustand.Start();
467
468         var server = new OpcUaServer();
469         server.StrukturEintragen(structure);
470         server.Start();
471         server.DatenstrukturAufbauen();
472
473         if (konfiguration.Ausführungstransitionen != null)
474         {
475           while(true)
```

```
476         {
477           System.Threading.Thread.Sleep(1000);
478
479           Zustand.Lesen();
480
481           var erfüllteTransitionen =
               ↪  konfiguration.Ausführungstransitionen.Where(a =>
               ↪  a.Eingangszustand == (System.Int32)Zustand.value);
482           if (erfüllteTransitionen.Count<Ausführungstransition>() <= 0)
               ↪  continue;
483
484           if (erfüllteTransitionen.ElementAt(0) ==
               ↪  konfiguration.Ausführungstransitionen[0])
485           {
486             while (server.puffer_mutex)
487             {
488
489             }
490
491             server.puffer_mutex = true;
492
493             foreach (var neuer_wert in server.puffer)
494             {
495               structure.dateneinträge[neuer_wert.Key].value = neuer_wert.Value;
496             }
497
498             server.puffer.Clear();
499
500             server.puffer_mutex = false;
501
502             structure.Schreiben();
503           } else if (erfüllteTransitionen.ElementAt(0) ==
               ↪  konfiguration.Ausführungstransitionen[1])
504           {
505             var aenderungen = structure.Lesen();
506
507             foreach (var geandert in aenderungen)
508             {
509               server.DatenwertVerbreiten(geandert);
510             }
511
512             structure.AenderungenZuruecksetzen(aenderungen);
513           }
514
515           Zustand.value =
               ↪  erfüllteTransitionen.First<Ausführungstransition>().Ausgangszustand;
516           Zustand.Schreiben();
517         }
518       } else {
519         while(true)
520         {
521           System.Threading.Thread.Sleep(50);
522
523           while (server.puffer_mutex) { }
524
525           server.puffer_mutex = true;
526
527           foreach (var neuer_wert in server.puffer)
528           {
529             structure.dateneinträge[neuer_wert.Key].value = neuer_wert.Value;
530           }
```

```
531
532             server.puffer.Clear();
533
534             server.puffer_mutex = false;
535
536             structure.Schreiben();
537
538             System.Threading.Thread.Sleep(50);
539
540             var aenderungen = structure.Lesen();
541
542             foreach (var geandert in aenderungen)
543             {
544               server.DatenwertVerbreiten(geandert);
545             }
546
547             structure.AenderungenZuruecksetzen(aenderungen);
548           }
549         }
550       }
551     }
552   }
```

Objekt D.5: Modul für die Bereitstellung der Datenstruktur über OPC UA

Verfügbar unter https://github.com/408b7f8b/isci.opcuaserver.

D.6 Modul Interne Kommunikation via UDP

```
1  using System;
2  using System.Net.Sockets;
3  using System.Net;
4  using System.Linq;
5  using System.Collections.Generic;
6  using isci.Allgemein;
7  using isci.Daten;
8  using isci.Beschreibung;
9
10 namespace isci.integration
11 {
12   public class Konfiguration : Parameter
13   {
14     public string[] Targets;
15     public string Adapter;
16     public int Port;
17
18     public Konfiguration(string datei) : base(datei) {
19
20     }
21   }
22
23   class Program
24   {
25     static Socket udpSock;
26     static byte[] buffer = new byte[1024];
27     static Dictionary<string, string> aenderungen = new Dictionary<string,
       ↪  string>();
28
```

```
29      static void udpStart(IPAddress ip, int port){
30        udpSock = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
           ↪  ProtocolType.Udp);
31        udpSock.Bind(new IPEndPoint(ip, port));
32        udpSock.BeginReceive(buffer, 0, buffer.Length,
           ↪  socketFlags:SocketFlags.None, udpCallback, udpSock);
33      }
34
35      static void udpCallback(IAsyncResult asyncResult){
36        try
37        {
38          Socket recvSock = (Socket)asyncResult.AsyncState;
39          EndPoint client = new IPEndPoint(IPAddress.Any, 0);
40          int length = recvSock.EndReceiveFrom(asyncResult, ref client);
41          var conv_string = System.Text.UTF8Encoding.UTF8.GetString(buffer, 0,
           ↪  length);
42          conv_string = conv_string.TrimEnd();
43
44          var string_parts = conv_string.Split('#');
45          while(change_lock) {}
46          try
47          {
48            if (aenderungen.ContainsKey(string_parts[0]))
             ↪  aenderungen[string_parts[0]] = string_parts[1];
49            else aenderungen.Add(string_parts[0], string_parts[1]);
50          } catch {
51
52          }
53
54          udpSock.BeginReceive(buffer, 0, buffer.Length,
           ↪  socketFlags:SocketFlags.None, udpCallback, udpSock);
55        } catch {
56
57        }
58      }
59
60      static List<IPEndPoint> targets = new List<IPEndPoint>();
61      static bool change_lock;
62      static Datenstruktur structure;
63
64      static IPAddress holeLokaleAdresse(string adapterName)
65      {
66        var adapters =
           ↪  System.Net.NetworkInformation.NetworkInterface.GetAllNetworkInterfaces();
67        var ips = Dns.GetHostAddresses(Dns.GetHostName());
68
69        foreach (var adapter in adapters)
70        {
71          if (adapter.Name != adapterName) continue;
72          var properties = adapter.GetIPProperties();
73          foreach (var ip in properties.UnicastAddresses)
74          if (ip.Address.AddressFamily == AddressFamily.InterNetwork)
75          return ip.Address;
76        }
77
78        return new IPAddress(0);
79      }
80
81      static void Main(string[] args)
82      {
83        var konfiguration = new Konfiguration("konfiguration.json");
```

```
84
85      structure = new Datenstruktur(konfiguration.OrdnerDatenstruktur);
86
87      var dm = new Datenmodell(konfiguration.Identifikation);
88      var cycle = new dtInt32(0, "cycle");
89      dm.Dateneinträge.Add(cycle);
90
91      var beschreibung = new Modul(konfiguration.Identifikation,
         ↪  "isci.integration", new ListeDateneintraege(){cycle});
92      beschreibung.Name = "Integration Ressource " +
         ↪  konfiguration.Identifikation;
93      beschreibung.Beschreibung = "Modul zur Integration";
94      beschreibung.Speichern(konfiguration.OrdnerBeschreibungen + "/" +
         ↪  konfiguration.Identifikation + ".json");
95
96      dm.Speichern(konfiguration.OrdnerDatenmodelle + "/" +
         ↪  konfiguration.Identifikation + ".json");
97
98      var ip = holeLokaleAdresse(konfiguration.Adapter);
99      while (ip.Equals(new IPAddress(0)))
100     {
101       System.Threading.Thread.Sleep(5000);
102       ip = holeLokaleAdresse(konfiguration.Adapter);
103     }
104
105     var schnittstelle = new SchnittstelleUdp(konfiguration.Ressource + ". " +
         ↪  konfiguration.Anwendung + "." + konfiguration.Identifikation,
         ↪  ip.ToString(), konfiguration.Ressource);
106     schnittstelle.Speichern(konfiguration.OrdnerSchnittstellen + "/" +
         ↪  schnittstelle.Identifikation + ".json");
107
108     structure.DatenmodellEinhängen(dm);
109
         ↪  structure.DatenmodelleEinhängenAusOrdner(konfiguration.OrdnerDatenmodelle);
110     structure.Start();
111
112     var Zustand = new dtInt32(0, "Zustand", konfiguration.OrdnerDatenstruktur
         ↪  + "/Zustand");
113     Zustand.Start();
114
115     for (int i = 0; i < konfiguration.Targets.Length; ++i)
116     {
117       try
118       {
119         var target_ =
             ↪  System.IO.File.ReadAllText(konfiguration.OrdnerSchnittstellen +
             ↪  "/" + konfiguration.Targets[i]);
120         var schnittstelleTarget =
             ↪  Newtonsoft.Json.JsonConvert.DeserializeObject<SchnittstelleUdp>(target_);
121         targets.Add(IPEndPoint.Parse(schnittstelleTarget.adresse));
122       } catch {
123
124       }
125     }
126
127     udpStart(ip, konfiguration.Port);
128     long curr_ticks = 0;
129
130     while(true)
131     {
132       Zustand.Lesen();
```

```
133
134         var erfüllteTransitionen =
             ↪ konfiguration.Ausführungstransitionen.Where(a => a.Eingangszustand
             ↪ == (System.Int32)Zustand.value);
135         if (erfüllteTransitionen.Count<Ausführungstransition>() > 0)
136         {
137           if ((System.Int32)Zustand.value == 0)
138           {
139             var curr_ticks_new = System.DateTime.Now.Ticks;
140             var ticks_span = curr_ticks_new - curr_ticks;
141             curr_ticks = curr_ticks_new;
142             cycle.value = (System.Int32)(ticks_span /
                 ↪ System.TimeSpan.TicksPerMillisecond);
143             cycle.Schreiben();
144
145             change_lock = true;
146             foreach (var aenderung in aenderungen)
147             {
148               try {
149
                    ↪ structure.dateneinträge[aenderung.Key].AusString(aenderung.Value);
150                 structure.dateneinträge[aenderung.Key].Schreiben();
151               } catch {
152
153               }
154             }
155             aenderungen.Clear();
156             change_lock = false;
157           } else {
158             var updated = structure.Lesen();
159
160             foreach (var entry in updated)
161             {
162               if (structure.dateneinträge[entry].Identifikation ==
                   ↪ "ns=integration;cycle") continue;
163               var snd = System.Text.UTF8Encoding.UTF8.GetBytes(entry + "#" +
                   ↪ structure.dateneinträge[entry].Serialisieren());
164               foreach (var target in targets) {
165                 udpSock.BeginSendTo(snd, 0, snd.Length, 0, target, null, null);
166               }
167
168               structure.dateneinträge[entry].aenderung = false;
169             }
170           }
171
172           Zustand.value =
               ↪ erfüllteTransitionen.First<Ausführungstransition>().Ausgangszustand;
173           Zustand.Schreiben();
174         }
175       }
176     }
177   }
178 }
```

Objekt D.6: Modul zur UDP-basierten Kommunikation im Automatisierungssystem

Verfügbar unter https://github.com/408b7f8b/isci.integration.

D.7 Modul für Kunbus Revolution Pi EA

```
1  using System;
2  using System.Linq;
3  using isci;
4  using isci.Allgemein;
5  using isci.Daten;
6  using isci.Beschreibung;
7  using isci.Anwendungen;
8  using System.Collections.Generic;
9
10 namespace revpiea
11 {
12   class Program
13   {
14     public static class RevPiZugriff
15     {
16       public static PiControl control = new PiControl();
17       static PiConfiguration config = new PiConfiguration();
18
19       static Systemkonfiguration Konfiguration;
20
21       public static void SystemkonfigurationLesen(string pfad =
         ↪  "/etc/revpi/config.rsc")
22       {
23         var inh = System.IO.File.ReadAllText(pfad).Replace("\"out\":",
           ↪  "\"Out\":");
24         var r =
           ↪  Newtonsoft.Json.JsonConvert.DeserializeObject<Systemkonfiguration>(inh);
25         Konfiguration = r;
26       }
27
28       public static Dictionary<string, ioObjekt> Eingänge;
29
30       public static Dictionary<string, ioObjekt> Ausgänge;
31
32       public static void EinUndAusgängeAufstellen()
33       {
34         Eingänge = new Dictionary<string, ioObjekt>();
35         Ausgänge = new Dictionary<string, ioObjekt>();
36
37         foreach(var device in Konfiguration.Devices)
38         {
39           foreach (var _inp in device.inp)
40           {
41             var o = new ioObjekt(_inp.Value, device.offset);
42             if (!o.sichtbar) continue;
43             //Eingänge.Add(_inp.Key, o);
44             Eingänge.Add(o.Name, o);
45           }
46           foreach (var _out in device.Out)
47           {
48             var o = new ioObjekt(_out.Value, device.offset);
49             if (!o.sichtbar) continue;
50             //Ausgänge.Add(_out.Key, o);
51             Ausgänge.Add(o.Name, o);
52           }
53         }
54       }
55     }
56
```

```
57    public class Systemkonfiguration
58    {
59      public List<Device> Devices;
60    }
61
62    public class Device
63    {
64      public int offset;
65      public Dictionary<string, List<object>> inp;
66      public Dictionary<string, List<object>> Out;
67    }
68
69    public class ioObjekt
70    {
71      public string Name;
72      public int AdresseByte;
73      public int AdresseBit;
74      public bool sichtbar;
75      private object wert;
76      public enum Typ {
77        BOOL = 1, BYTE = 8, WORD = 16, INT = 32
78      };
79      public Typ typ;
80
81      private byte[] wert_;
82
83      public ioObjekt(List<object> objektliste, int offset = 0)
84      {
85        Name = (string)objektliste[0];
86        wert = objektliste[1];
87        typ = (Typ) System.Int32.Parse((string)objektliste[2]);
88        AdresseByte = offset + System.Int32.Parse((string)objektliste[3]);
89        sichtbar = (bool)objektliste[4];
90        var AdresseBit_ = System.Int32.Parse((string)objektliste[5]);
91        var AdresseBit_g = (int) (AdresseBit_ / 8.0);
92        var AdresseBit_m = AdresseBit_ % 8;
93        AdresseBit = AdresseBit_m;
94        AdresseByte += AdresseBit_g;
95        int l = (typ == Typ.INT ? 4 : (typ == Typ.WORD ? 2 : 1));
96        wert_ = new byte[l];
97      }
98
99      public dtInt32 EintragErstellen(string praefix)
100     {
101       dtInt32 dateneintrag;
102
103       switch(typ)
104       {
105         case Typ.BOOL: dateneintrag = new dtInt32(wert_[0] != 0 ? 1 : 0,
            ↪  Name); break;
106         case Typ.BYTE: dateneintrag = new dtInt32(wert_[0], Name); break;
107         case Typ.INT: dateneintrag = new dtInt32((wert_[3] << 24) + (wert_[2]
            ↪  << 16) + (wert_[1] << 8) + wert_[0], Name); break;
108         case Typ.WORD: dateneintrag = new dtInt32((short)((wert_[1] << 8) +
            ↪  wert_[0]), Name); break;
109         default: return null;
110       }
111
112       return dateneintrag;
113     }
114
```

```
115     public bool Zustandlesen()
116     {
117       int l = (typ == Typ.INT ? 4 : (typ == Typ.WORD ? 2 : 1));
118
119       var wert__ = new byte[l];
120       wert__ = RevPiZugriff.control.Read(AdresseByte, l);
121
122       for (int i = 0; i < l; ++i)
123       {
124         if (wert__[i] != wert_[i])
125         {
126           goto Raus;
127         }
128       }
129
130       return false;
131
132       Raus:
133       wert_ = wert__;
134       return true;
135     }
136
137     public bool Zustandlesen(out object o)
138     {
139       bool r = Zustandlesen();
140
141       if (!r) {
142         o = null;
143         return false;
144       }
145
146       switch(typ)
147       {
148         case Typ.BOOL: bool b = ((int)wert_[0] & (1 << AdresseBit)) != 0; o =
            ↪ b; break;
149         case Typ.BYTE: o = wert_[0]; break;
150         case Typ.INT: int i = (wert_[3] << 24) + (wert_[2] << 16) + (wert_[1]
            ↪ << 8) + wert_[0]; o = i; break;
151         case Typ.WORD: int j = (wert_[1] << 8) + wert_[0]; o = j; break;
152         default: o = null; break;
153       }
154
155       if (o != null) return true;
156
157       return false;
158     }
159
160     public void Zustandschreiben(object o)
161     {
162       switch(typ)
163       {
164         case Typ.BOOL: wert_[0] = BitConverter.GetBytes((wert_[0] & ~(1 <<
            ↪ AdresseBit)) | ( ((bool)o ? 1 : 0) << AdresseBit))[0]; break;
165         case Typ.BYTE: wert_[0] = (byte)o; break;
166         case Typ.INT: wert_ = BitConverter.GetBytes((int)o); break;
167         case Typ.WORD: wert_ = BitConverter.GetBytes((short)o); break;
168         default: o = null; break;
169       }
170
171       RevPiZugriff.control.Write(AdresseByte, wert_);
172     }
```

```
173     }
174
175     static void Main(string[] args)
176     {
177       var konfiguration = new Parameter("konfiguration.json");
178
179       var structure = new Datenstruktur(konfiguration.OrdnerDatenstruktur);
180
181       RevPiZugriff.SystemkonfigurationLesen();
182       RevPiZugriff.EinUndAusgängeAufstellen();
183       RevPiZugriff.control.Open();
184
185       var dm = new Datenmodell(konfiguration.Identifikation);
186
187       var Ausgaenge = new Dictionary<dtInt32, ioObjekt>();
188       var Eingaenge = new Dictionary<ioObjekt, dtInt32>();
189
190       foreach (var eintrag_ in RevPiZugriff.Eingänge)
191       {
192         var eintrag = eintrag_.Value.EintragErstellen("");
193         dm.Dateneinträge.Add(eintrag);
194         Eingaenge.Add(eintrag_.Value, eintrag);
195       }
196
197       foreach (var eintrag_ in RevPiZugriff.Ausgänge)
198       {
199         var eintrag = eintrag_.Value.EintragErstellen("");
200         dm.Dateneinträge.Add(eintrag);
201         Ausgaenge.Add(eintrag, eintrag_.Value);
202       }
203
204       System.IO.File.WriteAllText(konfiguration.OrdnerDatenmodelle + "/" +
          ↪  konfiguration.Identifikation + ".json",
          ↪  Newtonsoft.Json.JsonConvert.SerializeObject(dm));
205
206       structure.DatenmodellEinhängen(dm);
207       structure.Start();
208
209       var beschreibung = new Modul(konfiguration.Identifikation,
          ↪  "isci.revpiea", dm.Dateneinträge);
210       beschreibung.Name = "RevPiEA Ressource " + konfiguration.Identifikation;
211       beschreibung.Beschreibung = "Modul zur EA-Integration von RevPi";
212       beschreibung.Speichern(konfiguration.OrdnerBeschreibungen + "/" +
          ↪  konfiguration.Identifikation + ".json");
213
214       var Zustand = new dtZustand(konfiguration.OrdnerDatenstruktur);
215       Zustand.Start();
216
217       while(true)
218       {
219         Zustand.Lesen();
220         var erfüllteTransitionen =
            ↪  konfiguration.Ausführungstransitionen.Where(a => a.Eingangszustand
            ↪  == (System.Int32)Zustand.value);
221         if (erfüllteTransitionen.Count<Ausführungstransition>() > 0)
222         {
223           if (erfüllteTransitionen.ElementAt(0).Eingangszustand ==
              ↪  (System.Int32)Zustand.value)
224           {
225             foreach (var Eingang in Eingaenge)
226             {
```

```
227                 object o = null;
228                 if (Eingang.Key.Zustandlesen(out o))
229                 {
230                   switch (Eingang.Key.typ)
231                   {
232                     case ioObjekt.Typ.BOOL: Eingang.Value.value = ((bool)o ? 1 :
                         ↪  0); break;
233                     case ioObjekt.Typ.BYTE: Eingang.Value.value = (int)((byte)o);
                         ↪  break;
234                     case ioObjekt.Typ.WORD: Eingang.Value.value = o; break;
235                     case ioObjekt.Typ.INT: Eingang.Value.value = o; break;
236                     default: continue;
237                   }
238
239                   Eingang.Value.Schreiben();
240                 }
241               }
242               structure.Schreiben();
243               Zustand.value = erfüllteTransitionen.ElementAt(0).Ausgangszustand;
244             } else if (erfüllteTransitionen.ElementAt(1).Eingangszustand ==
                 ↪  (System.Int32)Zustand.value)
245             {
246               foreach (var Ausgang in Ausgaenge)
247               {
248                 Ausgang.Key.Lesen();
249                 if (Ausgang.Key.aenderung)
250                 {
251                   Console.WriteLine("Aenderung " + Ausgang.Key.Identifikation);
252                   switch (Ausgang.Value.typ)
253                   {
254                     case ioObjekt.Typ.BOOL:
                         ↪  Ausgang.Value.Zustandschreiben((System.Int32)Ausgang.Key.value
                         ↪  == 1 ? true : false); break;
255                     case ioObjekt.Typ.BYTE:
                         ↪  Ausgang.Value.Zustandschreiben((byte)Ausgang.Key.value);
                         ↪  break;
256                     case ioObjekt.Typ.WORD:
                         ↪  Ausgang.Value.Zustandschreiben((short)Ausgang.Key.value);
                         ↪  break;
257                     case ioObjekt.Typ.INT:
                         ↪  Ausgang.Value.Zustandschreiben((System.Int32)Ausgang.Key.value);
                         ↪  break;
258                     default: continue;
259                   }
260
261                   Ausgang.Key.aenderung = false;
262                 }
263               }
264               Zustand.value = erfüllteTransitionen.ElementAt(1).Ausgangszustand;
265             }
266
267             Zustand.Schreiben();
268           }
269         }
270       }
271     }
272   }
```

Objekt D.7: Einbindung Kunbus Revolution Pi EA

Verfügbar unter https://github.com/408b7f8b/isci.revpiea.

D.8 Modul für Manufacturing Service Bus

```
1  using System;
2  using System.Net.Sockets;
3  using System.Net;
4  using System.Linq;
5  using System.Collections.Generic;
6  using isci.Allgemein;
7  using isci.Daten;
8  using isci.Beschreibung;
9
10 namespace isci.msb
11 {
12   public class Konfiguration : Parameter
13   {
14     public string uuid, name, description, token, target_interface;
15     public Konfiguration(string datei) : base(datei) {
16
17     }
18   }
19
20   class Program
21   {
22     static Dictionary<string, object> puffer = new Dictionary<string,
       ↪ object>();
23
24     public static void msbCallback_Connected(object sender, System.EventArgs e)
25     {
26       msbClient.RegisterAsync(msbApplication);
27     }
28     public static void msbCallback_Disconnected(object sender, System.EventArgs
       ↪ e)
29     {
30       msbActive = false;
31     }
32     public static void msbCallback_Registered(object sender, System.EventArgs
       ↪ e)
33     {
34       msbActive = true;
35     }
36     public static void msbCallback_FunctionCallMethod(
37     [Fraunhofer.IPA.MSB.Client.API.Attributes.MsbFunctionParameter(Name =
       ↪ "val")]System.Collections.Generic.Dictionary<string, object> val,
       ↪ Fraunhofer.IPA.MSB.Client.API.Model.FunctionCallInfo info)
38     {
39       foreach (var obj in val)
40       {
41         if (puffer.ContainsKey(obj.Key))
42         {
43           puffer[obj.Key] = obj.Value;
44         } else {
45           puffer.Add(obj.Key, obj.Value);
46         }
47       }
48     }
49
50     public static void ApplikationBauen()
51     {
52       var Events = new
         ↪ System.Collections.Generic.List<isci.Anwendungen.Ereignis>();
```

```
53      var Funktionen = new
         ↪  System.Collections.Generic.List<isci.Anwendungen.Funktion>();
54
55      var dateien =
         ↪  System.IO.Directory.GetFiles(konfiguration.OrdnerEreignismodelle,
         ↪  "*.json");
56
57      foreach (var datei_ in dateien)
58      {
59        var datei =
           ↪  Newtonsoft.Json.JsonConvert.DeserializeObject<System.Collections.Generic.List
60        <isci.Anwendungen.Ereignis>>(System.IO.File.ReadAllText(datei_));
61        if (datei != null)
62        {
63          Events.AddRange(datei);
64        }
65      }
66
67      dateien =
         ↪  System.IO.Directory.GetFiles(konfiguration.OrdnerFunktionsmodelle,
         ↪  "*.json");
68
69      foreach (var datei_ in dateien)
70      {
71        var datei =
           ↪  Newtonsoft.Json.JsonConvert.DeserializeObject<System.Collections.Generic.List
72        <isci.Anwendungen.Funktion>>(System.IO.File.ReadAllText(datei_));
73        if (datei != null)
74        {
75          Funktionen.AddRange(datei);
76        }
77      }
78
79      foreach (var f in Funktionen)
80      {
81        var df = new Fraunhofer.IPA.MSB.Client.Websocket.Model.DataFormat();
82        var mInfo =
           ↪  typeof(Program).GetMethod("msbCallback_FunctionCallMethod");
83        var f_ = new
           ↪  Fraunhofer.IPA.MSB.Client.API.Model.Function(f.Identifikation,
           ↪  f.Name, f.Beschreibung, null, mInfo, null);
84        msbApplication.AddFunction(f_);
85      }
86
87      foreach (var e in Events)
88      {
89        var df = new Fraunhofer.IPA.MSB.Client.Websocket.Model.DataFormat();
90        var e_ = new
           ↪  Fraunhofer.IPA.MSB.Client.API.Model.Event(e.Identifikation, e.Name,
           ↪  e.Beschreibung, df);
91        msbApplication.AddEvent(e_);
92
93        AuslöserEreignis = new
           ↪  System.Collections.Generic.List<KeyValuePair<isci.Daten.Dateneintrag,
           ↪  isci.Anwendungen.Ereignis>>();
94        EreignisEvent = new
           ↪  System.Collections.Generic.Dictionary<isci.Anwendungen.Ereignis,
           ↪  Fraunhofer.IPA.MSB.Client.API.Model.Event>();
95
96        AuslöserEreignis.Add(new KeyValuePair<Dateneintrag,
           ↪  Anwendungen.Ereignis>(structure.dateneinträge[e.Ausloeser], e));
```

```
97          EreignisEvent.Add(e, e_);
98        }
99      }
100
101     static
         ↪  System.Collections.Generic.List<KeyValuePair<isci.Daten.Dateneintrag,
         ↪  isci.Anwendungen.Ereignis>> AuslöserEreignis;
102     static System.Collections.Generic.Dictionary<isci.Anwendungen.Ereignis,
         ↪  Fraunhofer.IPA.MSB.Client.API.Model.Event> EreignisEvent;
103     static Fraunhofer.IPA.MSB.Client.Websocket.MsbClient msbClient;
104     static Fraunhofer.IPA.MSB.Client.API.Model.Application msbApplication;
105     static bool msbActive = false;
106     static Datenstruktur structure;
107     static Konfiguration konfiguration;
108     static void Main(string[] args)
109     {
110       konfiguration = new Konfiguration("konfiguration.json");
111
112       msbClient = new MsbClientFP(konfiguration.target_interface);
113       msbClient.Connected += msbCallback_Connected;
114       msbClient.Disconnected += msbCallback_Disconnected;
115       msbClient.Registered += msbCallback_Registered;
116       msbClient.AutoReconnect = true;
117       msbClient.AutoReconnectIntervalInMilliseconds = 10000;
118
119       structure = new Datenstruktur(konfiguration.OrdnerDatenstruktur);
120
           ↪  structure.DatenmodelleEinhängenAusOrdner(konfiguration.OrdnerDatenmodelle);
121
122       msbApplication = new
           ↪  Fraunhofer.IPA.MSB.Client.API.Model.Application(konfiguration.uuid,
           ↪  konfiguration.name, konfiguration.description, konfiguration.token);
123       ApplikationBauen();
124
125       var beschreibung = new Modul(konfiguration.Identifikation, "isci.msb");
126       beschreibung.Name = "Modul MSB " + konfiguration.Identifikation;
127       beschreibung.Beschreibung = "Modul MSB";
128       beschreibung.Speichern(konfiguration.OrdnerBeschreibungen + "/" +
           ↪  konfiguration.Identifikation + ".json");
129
130       msbClient.ConnectAsync();
131
132       structure.Start();
133
134       var Zustand = new dtZustand(konfiguration.OrdnerDatenstruktur);
135       Zustand.Start();
136
137       while(true)
138       {
139         Zustand.Lesen();
140
141         var erfüllteTransitionen =
             ↪  konfiguration.Ausführungstransitionen.Where(a => a.Eingangszustand
             ↪  == (System.Int32)Zustand.value);
142         if (erfüllteTransitionen.Count<Ausführungstransition>() <= 0) continue;
143
144         if (erfüllteTransitionen.ElementAt(0) ==
             ↪  konfiguration.Ausführungstransitionen.ElementAt(0))
145         {
146           foreach (var gespeichert in puffer)
147           {
```

```
148             var eintrag = structure.dateneinträge[gespeichert.Key];
149             var val = gespeichert.Value;
150             switch (eintrag.type)
151             {
152               case Datentypen.Bool:
                   ↪  structure.dateneinträge[gespeichert.Key].value = (bool)val;
                   ↪  break;
153               case Datentypen.Int8:
                   ↪  structure.dateneinträge[gespeichert.Key].value = (SByte)val;
                   ↪  break;
154               case Datentypen.Int16:
                   ↪  structure.dateneinträge[gespeichert.Key].value = (Int16)val;
                   ↪  break;
155               case Datentypen.Int32:
                   ↪  structure.dateneinträge[gespeichert.Key].value = (Int32)val;
                   ↪  break;
156               case Datentypen.UInt8:
                   ↪  structure.dateneinträge[gespeichert.Key].value = (byte)val;
                   ↪  break;
157               case Datentypen.UInt16:
                   ↪  structure.dateneinträge[gespeichert.Key].value = (UInt16)val;
                   ↪  break;
158               case Datentypen.UInt32:
                   ↪  structure.dateneinträge[gespeichert.Key].value = (UInt32)val;
                   ↪  break;
159               case Datentypen.String:
                   ↪  structure.dateneinträge[gespeichert.Key].value = (String)val;
                   ↪  break;
160               case Datentypen.Float:
                   ↪  structure.dateneinträge[gespeichert.Key].value = (float)val;
                   ↪  break;
161               case Datentypen.Double:
                   ↪  structure.dateneinträge[gespeichert.Key].value = (Double)val;
                   ↪  break;
162               default:continue;
163             }
164             eintrag.Schreiben();
165           }
166           puffer.Clear();
167
168         } else if (erfüllteTransitionen.ElementAt(0) ==
             ↪  konfiguration.Ausführungstransitionen.ElementAt(1))
169         {
170           structure.Lesen();
171
172           foreach (var ausloeser in AuslöserEreignis)
173           {
174             if (ausloeser.Key.aenderung)
175             {
176               var eventData = new
                   ↪  Fraunhofer.IPA.MSB.Client.API.Model.EventData(EreignisEvent[ausloeser.V
177               var data = new Dictionary<string, object>();
178               foreach (var element in ausloeser.Value.Elemente)
179               {
180                 data.Add(element, structure.dateneinträge[element].value);
181               }
182               eventData.Value = data;
183               msbClient.PublishAsync(msbApplication, eventData);
184             }
185           }
186
```

```
187             structure.AenderungenZuruecksetzen();
188           }
189
190           Zustand.value =
              ↪  erfüllteTransitionen.First<Ausführungstransition>().Ausgangszustand;
191           Zustand.Schreiben();
192         }
193       }
194     }
195 }
```

Objekt D.8: Modul zur Anbindung an einen Manufacturing Service Bus

Verfügbar unter https://github.com/408b7f8b/isci.msb.

D.9 Anwendung für die funktionale Validierung

```
1 {
2   "Ressourcenkartierung": {
3     "anwendung.IntegrationStandard": {
4       "*": "*"
5     },
6     "anwendung.Zykluszeit": {
7       "*": "*"
8     },
9     "anwendung.Systemabbilder": {
10      "*": "*"
11    }
12  },
13  "Ausführungstransitionen": {
14    "RevPiA": {
15      "zykluszeit": [
16        {
17          "Eingangszustand": 0,
18          "Ausgangszustand": 1
19        }
20      ],
21      "integration": [
22        {
23          "Eingangszustand": 1,
24          "Ausgangszustand": 2
25        },
26        {
27          "Eingangszustand": 3,
28          "Ausgangszustand": 0
29        }
30      ],
31      "verarbeitung": [
32        {
33          "Eingangszustand": 2,
34          "Ausgangszustand": 3
35        }
36      ]
37    },
38    "RevPiB": {
39      "zykluszeit": [
40        {
41          "Eingangszustand": 0,
42          "Ausgangszustand": 1
43        }
44      ],
45      "integration": [
46        {
47          "Eingangszustand": 1,
48          "Ausgangszustand": 2
49        },
50        {
51          "Eingangszustand": 3,
52          "Ausgangszustand": 0
53        }
54      ],
55      "revpiea": [
56        {
57          "Eingangszustand": 2,
58          "Ausgangszustand": 3
59        }
```

```
60        ]
61      }
62    },
63    "ReferenzierteModelle": [
64      "anwendung.Zykluszeit",
65      "anwendung.IntegrationStandard",
66      "anwendung.Systemabbilder"
67    ],
68    "Konfigurationspakete": {
69      "RevPiA": {
70        "verarbeitung": "isci.validverarbeitung"
71      },
72      "RevPiB": {
73        "revpiea": "isci.revpiea"
74      }
75    },
76    "Konfigurationselemente": {},
77    "Datenmodelle": [],
78    "Funktionen": [],
79    "Ereignisse": [],
80    "Identifikation": "anwendung.Validierung",
81    "Name": "Validierungsanwendung",
82    "Beschreibung": "Validierungsanwendung",
83    "Typ": "isci.Anwendungen.Instanz"
84  }
```

Objekt D.9: Objekt der Anwendungsinstanz für die funktionale Validierung

D.10 Anwendungen für das Evaluationsszenario

```
1  {
2    "ReferenzierteModelle": [],
3    "Konfigurationspakete": {
4      "*": {
5        "integration": "isci.integration",
6        "link": "isci.link"
7      }
8    },
9    "Konfigurationselemente": {},
10   "Datenmodelle": [],
11   "Funktionen": [],
12   "Ereignisse": [],
13   "Identifikation": "anwendung.IntegrationStandard",
14   "Name": "IntegrationStandard",
15   "Beschreibung": "Anwendung zur Standardintegration für alle Einzelsysteme",
16   "Typ": "isci.Anwendungen.Basis"
17 }
18
19 {
20   "ReferenzierteModelle": [],
21   "Konfigurationspakete": {
22     "*": {
23       "abbild": "isci.abbild"
24     }
25   },
26   "Konfigurationselemente": {},
27   "Datenmodelle": [],
28   "Funktionen": [],
```

```
29   "Ereignisse": [],
30   "Identifikation": "anwendung.Systemabbilder",
31   "Name": "Systemabbilder",
32   "Beschreibung": "Anwendung mit Systemabbildung für alle Einzelsysteme",
33   "Typ": "isci.Anwendungen.Basis"
34 }
35
36 {
37   "ReferenzierteModelle": [],
38   "Konfigurationspakete": {
39     "*": {
40       "zykluszeit": "isci.zykluszeit"
41     }
42   },
43   "Konfigurationselemente": {},
44   "Datenmodelle": [],
45   "Funktionen": [],
46   "Ereignisse": [],
47   "Identifikation": "anwendung.Zykluszeit",
48   "Name": "Zykluszeit",
49   "Beschreibung": "Anwendung zur Zykluszeitermittlung für alle Einzelsysteme",
50   "Typ": "isci.Anwendungen.Basis"
51 }
```

Objekt D.10: Objektbeispiele für generische Standardanwendungen

```
1 {
2   "ReferenzierteModelle": [
3     "anwendung.Zykluszeit",
4     "anwendung.IntegrationStandard"
5   ],
6   "Konfigurationspakete": {
7     "SteuerungX": {
8       "AchseX": "isci.schrittmotorachse"
9     },
10    "SteuerungY": {
11      "AchseY": "isci.schrittmotorachse"
12    },
13    "SteuerungZ": {
14      "AchseZ": "isci.schrittmotorachse"
15    },
16    "SteuerungSpind": {
17      "Spindel": "isci.modbusspindel"
18    },
19    "SteuerungCnc": {
20      "CNC": "isci.dreiachsansteuerung",
21      "HMI": "isci.dashboard"
22    },
23    "SchnittstelleMsb": {
24      "MSB": "isci.msb"
25    }
26  },
27  "Konfigurationselemente": {
28    "SteuerungX": [
29      {
30        "Typ": "Parameter",
31        "Vorgang": {
32          "Ordner": "AchseX",
```

```
33          "Variablen": {
34            "Steigung": "5",
35            "Aufloesung": "400"
36          }
37        }
38      }
39    ],
40    "SteuerungY": [
41      {
42        "Typ": "Parameter",
43        "Vorgang": {
44          "Ordner": "AchseY",
45          "Variablen": {
46            "Steigung": "5",
47            "Aufloesung": "400"
48          }
49        }
50      }
51    ],
52    "SteuerungZ": [
53      {
54        "Typ": "Parameter",
55        "Vorgang": {
56          "Ordner": "AchseZ",
57          "Variablen": {
58            "Steigung": "8",
59            "Aufloesung": "400"
60          }
61        }
62      }
63    ],
64    "SteuerungSpind": [
65      {
66        "Typ": "Parameter",
67        "Vorgang": {
68          "Ordner": "Spindel",
69          "Variablen": {
70            "Adresse": "1"
71          }
72        }
73      }
74    ]
75  },
76  "Datenmodelle": [
77    {
78      "Dateneinträge": [],
79      "Links": {
80        "ns=AchseX;s=pXist": [
81          "ns=CNC;s=pXist"
82        ],
83        "ns=CNC;s=pXsoll": [
84          "ns=AchseX;s=pXsoll"
85        ],
86        "ns=AchseX;s=vXist": [
87          "ns=CNC;s=vXist"
88        ],
89        "ns=CNC;s=vXsoll": [
```

```
90          "ns=AchseX;s=vXsoll"
91        ],
92        "ns=AchseY;s=pYist": [
93          "ns=CNC;s=pYist"
94        ],
95        "ns=CNC;s=pYsoll": [
96          "ns=AchseY;s=pYsoll"
97        ],
98        "ns=AchseY;s=vYist": [
99          "ns=CNC;s=vYist"
100       ],
101       "ns=CNC;s=vYsoll": [
102         "ns=AchseY;s=vYsoll"
103       ],
104       "ns=AchseZ;s=pZist": [
105         "ns=CNC;s=pZist"
106       ],
107       "ns=CNC;s=pZsoll": [
108         "ns=AchseZ;s=pZsoll"
109       ],
110       "ns=AchseZ;s=vZist": [
111         "ns=CNC;s=vZist"
112       ],
113       "ns=CNC;s=vZsoll": [
114         "ns=AchseZ;s=vZsoll"
115       ],
116       "ns=Spindel;s=nIst": [
117         "ns=CNC;s=nSoll"
118       ],
119       "ns=CNC;s=nSoll": [
120         "ns=Spindel;s=nIst"
121       ]
122     },
123     "Identifikation": "CNC",
124     "Name": "",
125     "Beschreibung": "",
126     "Typ": "isci.Daten.Datenmodell"
127   }
128 ],
129 "Funktionen": [
130   {
131     "Ziele": [
132       "ns=CNC;s=Start"
133     ],
134     "Identifikation": "fStart",
135     "Name": "Start",
136     "Beschreibung": "Maschine Start",
137     "Typ": "isci.Anwendungen.Funktion"
138   },
139   {
140     "Ziele": [
141       "ns=CNC;s=Halt"
142     ],
143     "Identifikation": "fHalt",
144     "Name": "Halt",
145     "Beschreibung": "Maschine Halt",
146     "Typ": "isci.Anwendungen.Funktion"
```

```
147     },
148     {
149       "Ziele": [
150         "ns=CNC;s=Schritte"
151       ],
152       "Identifikation": "fSchritte",
153       "Name": "Schritte",
154       "Beschreibung": "Maschine Schritte",
155       "Typ": "isci.Anwendungen.Funktion"
156     }
157   ],
158   "Ereignisse": [
159     {
160       "Ausloeser": "ns=CNC;s=Gestartet",
161       "Elemente": [
162         "ns=CNC;s=Gestartet"
163       ],
164       "Identifikation": "eGestartet",
165       "Name": "Gestartet",
166       "Beschreibung": "Maschine gestartet",
167       "Typ": "isci.Anwendungen.Ereignis"
168     },
169     {
170       "Ausloeser": "ns=CNC;s=Gehalten",
171       "Elemente": [
172         "ns=CNC;s=Gehalten"
173       ],
174       "Identifikation": "eGehalten",
175       "Name": "Gehalten",
176       "Beschreibung": "Maschine gehalten",
177       "Typ": "isci.Anwendungen.Ereignis"
178     },
179     {
180       "Ausloeser": "ns=CNC;s=Abschluss",
181       "Elemente": [
182         "ns=CNC;s=Abschluss"
183       ],
184       "Identifikation": "eAbschluss",
185       "Name": "Abschluss",
186       "Beschreibung": "Bearbeitung abgeschlossen",
187       "Typ": "isci.Anwendungen.Ereignis"
188     },
189     {
190       "Ausloeser": "ns=CNC;s=Status",
191       "Elemente": [
192         "ns=CNC;s=AktuellerSchritt",
193         "ns=CNC;s=pXist",
194         "ns=CNC;s=pYist",
195         "ns=CNC;s=pZist",
196         "ns=CNC;s=nIst"
197       ],
198       "Identifikation": "eStatus",
199       "Name": "Status",
200       "Beschreibung": "Status verteilen",
201       "Typ": "isci.Anwendungen.Ereignis"
202     }
203   ],
```

```
204   "Identifikation": "anwendung.CncStandard",
205   "Name": "Basismaschine CNC",
206   "Beschreibung": "Standardanwendung",
207   "Typ": "isci.Anwendungen.Basis"
208 }
```

Objekt D.11: Standardanwendung für die Szenariovalidierung

```
1 {
2   "Ressourcenkartierung": {
3     "anwendung.IntegrationStandard": {
4       "*": "*"
5     },
6     "anwendung.Zykluszeit": {
7       "*": "*"
8     },
9     "anwendung.Systemabbilder": {
10      "*": "*"
11    },
12    "anwendung.CncStandard": {
13      "SteuerungX": "RevPiXYZ",
14      "SteuerungY": "RevPiXYZ",
15      "SteuerungZ": "RevPiXYZ",
16      "SteuerungSpind": "RevPiSpind",
17      "SteuerungCnc": "PC",
18      "SchnittstelleMsb": "PC"
19    }
20  },
21  "Ausführungstransitionen": {
22    "RevPiXYZ": {
23      "zykluszeit": [
24        {
25          "Eingangszustand": 0,
26          "Ausgangszustand": 1
27        }
28      ],
29      "integration": [
30        {
31          "Eingangszustand": 1,
32          "Ausgangszustand": 2
33        },
34        {
35          "Eingangszustand": 7,
36          "Ausgangszustand": 0
37        }
38      ],
39      "AchseX": [
40        {
41          "Eingangszustand": 3,
42          "Ausgangszustand": 4
43        }
44      ],
45      "AchseY": [
46        {
47          "Eingangszustand": 4,
48          "Ausgangszustand": 5
```

```
49          }
50        ],
51        "AchseZ": [
52          {
53            "Eingangszustand": 5,
54            "Ausgangszustand": 6
55          }
56        ],
57        "link": [
58          {
59            "Eingangszustand": 2,
60            "Ausgangszustand": 3
61          },
62          {
63            "Eingangszustand": 6,
64            "Ausgangszustand": 7
65          }
66        ]
67      },
68      "RevPiSpind": {
69        "zykluszeit": [
70          {
71            "Eingangszustand": 0,
72            "Ausgangszustand": 1
73          }
74        ],
75        "integration": [
76          {
77            "Eingangszustand": 1,
78            "Ausgangszustand": 2
79          },
80          {
81            "Eingangszustand": 5,
82            "Ausgangszustand": 0
83          }
84        ],
85        "Spindel": [
86          {
87            "Eingangszustand": 3,
88            "Ausgangszustand": 4
89          }
90        ],
91        "link": [
92          {
93            "Eingangszustand": 2,
94            "Ausgangszustand": 3
95          },
96          {
97            "Eingangszustand": 4,
98            "Ausgangszustand": 5
99          }
100       ]
101     },
102     "PC": {
103       "zykluszeit": [
104         {
105           "Eingangszustand": 0,
```

```
106           "Ausgangszustand": 1
107         }
108       ],
109       "integration": [
110         {
111           "Eingangszustand": 1,
112           "Ausgangszustand": 2
113         },
114         {
115           "Eingangszustand": 6,
116           "Ausgangszustand": 0
117         }
118       ],
119       "MSB": [
120         {
121           "Eingangszustand": 2,
122           "Ausgangszustand": 3
123         }
124       ],
125       "CNC": [
126         {
127           "Eingangszustand": 4,
128           "Ausgangszustand": 5
129         }
130       ],
131       "link": [
132         {
133           "Eingangszustand": 3,
134           "Ausgangszustand": 4
135         },
136         {
137           "Eingangszustand": 5,
138           "Ausgangszustand": 6
139         }
140       ]
141     }
142   },
143   "ReferenzierteModelle": [
144     "anwendung.Systemabbilder",
145     "anwendung.CncStandard"
146   ],
147   "Konfigurationspakete": {},
148   "Konfigurationselemente": {
149     "RevPiXYZ": [
150       {
151         "Typ": "Parameter",
152         "Vorgang": {
153           "Ordner": "AchseX",
154           "Variablen": {
155             "Id": "101"
156           }
157         }
158       },
159       {
160         "Typ": "Parameter",
161         "Vorgang": {
162           "Ordner": "AchseY",
```

```
163           "Variablen": {
164             "Id": "102"
165           }
166         }
167       },
168       {
169         "Typ": "Parameter",
170         "Vorgang": {
171           "Ordner": "AchseZ",
172           "Variablen": {
173             "Id": "103"
174           }
175         }
176       }
177     ]
178   },
179   "Datenmodelle": [],
180   "Funktionen": [],
181   "Ereignisse": [],
182   "Identifikation": "anwendung.Maschine1",
183   "Name": "CNC-Maschine 1",
184   "Beschreibung": "Instanz",
185   "Typ": "isci.Anwendungen.Instanz"
186 }
```

Objekt D.12: Objekt der Anwendungsinstanz für die Szenariovalidierung

E Definitionen

E.1 RAMI4.0

Bekanntheitsklasse	Beschreibung
nicht bekannt	ohne Eintrag im Informationssystem
anonym bekannt	mit Eintrag im Informationssystem ohne Verknüpfung zu einer physischen Entität
individuell bekannt	mit Eintrag im Informationssystem, Eintrag mit eindeutiger Verknüpfung zur physischen Entität
entitätsverwaltet	mit Eintrag im Informationssystem über gesamten Lebenszyklus, mit eindeutiger Verknüpfung zur physischen Entität

Tabelle E.1: Bekanntheitsklassen eines RAMI4.0-Assets (DIN SPEC 91345; IEC PAS 63088)

Kommunikationsfähigkeit	Beschreibung
nicht kommunikationsfähig	Asset kann nicht kommunizieren
passiv kommunikationsfähig	Asset hält Daten und kann ausgelesen werden
aktiv kommunikationsfähig	Asset hat eine aktive Netzwerkschnittstelle mit Zugriffsmöglichkeiten
I4.0-kommunikationsfähig	Asset hat eine I4.0-konforme Netzwerkschnittstelle

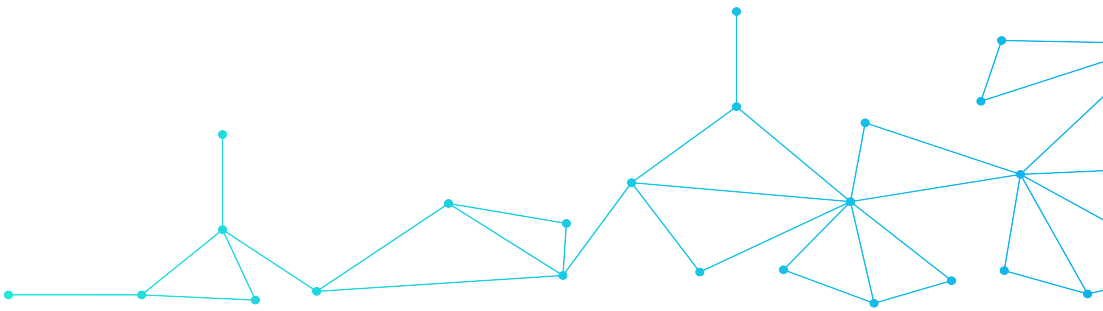
Tabelle E.2: Kommunikationsfähigkeiten eines RAMI4.0-Assets (DIN SPEC 91345; IEC PAS 63088)

Merkmalklasse	Träger	standardisiert	verpflichtend
Basismerkmal	Verwaltungsschale	x	x
Pflichtmerkmal	Teilmodell	x	x
optionales Merkmal	Teilmodell	x	
freies Merkmal	Teilmodell		

Tabelle E.3: Mögliche Klassen eines Merkmals im RAMI4.0 (DIN SPEC 91345; IEC PAS 63088)

Name	Sprache	Technologien	Lizenz	verf.
BaSyx 4.0	Java, C++, C#	REST	EPL 2.0	x
openAAS	-	OPC UA	ASL 2.0	
NOVAAS	-	REST, beliebige	-	
SAP AIN	Typescript	Docker, REST, GRPC	ASL 2.0	x
SemAnz40	-	OPC UA, AutomationML	-	
IBM Cognitive Twin	-	-	-	
AASX Package Explorer	C#	OPC UA, REST	EPL 2.0	

Tabelle E.4: Bekannte Implementierungen der RAMI4.0-Verwaltungsschale (aus Rödel 2020)

E.2 SoA

- **Wiederverwendbarkeit:** Ein Service ist wiederverwendbar und nicht auf eine bestimmte Aufgabe festgelegt, oder zeitlich oder physisch gebunden.
- **Formale Beschreibung:** Ein Service besitzt eine standardisierte Beschreibung im Format der jeweiligen serviceorientierten Architektur, der seine Schnittstellen mittels Metadaten verständlich macht.
- **Lose Kopplung:** Ein Service ist in einer serviceorientierten Architektur nicht fest gebunden, sondern nur lose und basierend auf einer Konfiguration.
- **Komplexitätsabstraktion:** Beschreibung und Bedienung abstrahieren den eigentlichen internen Ablauf eines Services und damit seine Komplexität vom Nutzer.
- **Kombinierbarkeit:** Services können innerhalb einer serviceorientierten Architektur zusammengestellt werden, um in festgelegten Sequenzen größere Aufgaben zu lösen.
- **Autarkie:** Ein Service ist frei instanziierbar. Er ist auf keinen bestimmten Dateneingang oder einen anderen Service angewiesen.
- **Zustandslosigkeit:** Services innerhalb einer serviceorientierten Architektur besitzen keinen Zustand. Sie sind funktionsorientiert aufgebaut.
- **Entdeckbarkeit:** Ein Service ist für alle anderen Services einer serviceorientierten Architektur mittels standardisierten Funktionen findbar und zugänglich.