Cognitive Interaction Technology
Kognitronik und Sensorik
Prof. Dr.-Ing. U. Rückert
Universität Bielefeld

Institut für die Digitalisierung von Arbeits- und Lebenswelten
Fachhochschule Dortmund

Verteilte Systemarchitektur für mobile Roboter

zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEUR (Dr.-Ing.)

der Technischen Fakultät
der Universität Bielefeld

genehmigte Dissertation

von

Uwe Jahn

Referent:	Prof. Dr.-Ing. Ulrich Rückert
Korreferent:	Prof. Dr.-Ing. Peter Schulz
Korreferent:	PD Dr.-Ing. Sven Wachsmuth

Tag der mündlichen Prüfung: 23.11.2021

Bielefeld / 2021

Kurzfassung

Bei mobilen Robotern gibt es zahlreiche Ansätze für Systemarchitekturen, sowohl als monolithisches, als auch als verteiltes System innerhalb eines Roboters. Ziel der vorliegenden Dissertation ist die Entwicklung einer allgemeingültigen Systemarchitektur für einen typischen mobilen Roboter. Zur Definition eines typischen mobilen Roboters wird eine Taxonomie aus Klassen, Anwendungsgebieten, Fähigkeiten und technischen Realisierungen von mobilen Robotern aufgrund einer umfangreichen Literaturrecherche erarbeitet.

Die Systemarchitektur wird in Form eines konzeptionellen Modells beschrieben. Als grundlegende Struktur wird ein verteiltes System aus der Drei-Schichten Architektur des OCM verwendet. Die drei hierarchisch getrennten Schichten bestehen aus einer *Controller*-Ebene, mindestens einem *reflektorischen Operator* und einem *kognitiven Operator*. Die *Controller* sind direkt mit den Sensoren und Aktuatoren des Roboters verbunden, weshalb die Einhaltung von harter Echtzeit erforderlich ist. Der *kognitive Operator* hingegen wird zur Optimierung des Systems verwendet und muss nicht echtzeitfähig realisiert werden, sodass das konzeptionelle Modell die Ausführung des *kognitiven Operators* in der Cloud vorsieht. Neben der Struktur werden im konzeptionellen Modell Schnittstellen und Funktionen wie z. B. Watchdogs zur Erhöhung der Sicherheit (Safety) erläutert. Außerdem werden mit dem *Motion*, *Health* und *Perception Controller* typische Systemkomponenten für mobile Roboter definiert.

Das konzeptionelle Modell wird zudem experimentell umgesetzt. Hierzu wird die Entwicklung des mobilen Roboters DAEbot gezeigt. Der Prototyp setzt die OCM-basierte Systemarchitektur als verteiltes System aus diversen SBCs um. Durch die Nutzung von vergleichsweise energieeffizienten SBCs in Kombination mit dem cloudbasierten *kognitiven Operator*, ist der DAEbot energieeffizient und kann trotzdem auf ausreichend Rechenkapazität zurückgreifen. Die Umsetzung enthält u. a. die Entwicklung einer Toolbox zur modellbasierten Entwicklung, die Erweiterung des CAN-Busses und die Einbindung der Cloud. Mittels eines im Rahmen dieser Arbeit entwickelten Analyse-Tools namens pulseAT wird die Systemauslastung aller Rechner des verteilten Systems überwacht, an zentraler Stelle gesammelt und analysiert. Hierbei wird u. a. die Antwortzeit der einzelnen Rechner, sowie die Einhaltung der Echtzeit überprüft. Dies erhöht die Zuverlässigkeit des mobilen Roboters.

Die Evaluation des konzeptionellen Modells und die experimentelle Realisierung in Form des DAEbots zeigen, dass sowohl die Realisierung eines verteilten Systems, als auch die OCM Struktur und die Einbindung der Cloud für mobile Roboter sinnvoll sind. Zudem ist ein experimentelles Framework für verteilte Robotiksysteme für zukünftige Projekte entstanden.

Abstract

There are numerous approaches for system architectures in mobile robots, both as monolithic and distributed systems within a robot. This doctoral thesis aims to develop a generally applicable system architecture for a typical mobile robot. A taxonomy of classes, applications, capabilities, and technical implementations of mobile robots is developed based on an extensive literature review to define a typical mobile robot.

The system architecture is described in the form of a conceptual model. A distributed system based on the three-layer architecture of the OCM is used as the basic structure. The three hierarchically separated layers consist of a *Controller* layer, at least one *Reflective Operator*, and one *Cognitive Operator*. The *Controllers* are directly connected to the robot's sensors and actuators, which is why hard real-time compliance is required. The *Cognitive Operator*, on the other hand, is used to optimize the system and does not need to be implemented in real-time, so the conceptual model calls for the *Cognitive Operator* to be executed in the cloud. In addition to the structure, the conceptual model explains interfaces and functions such as watchdogs to increase safety. In addition, typical system components for mobile robots are defined with the *Motion*, *Health* and *Perception Controller*.

The conceptual model is also implemented as an experimental prototype. For this purpose, the development of the mobile robot DAEbot is shown. The prototype implements the OCM-based system architecture as a distributed system of diverse SBCs. By using energy-efficient SBCs in combination with the cloud-based *Cognitive Operator*, the DAEbot is energy-efficient while still being able to access sufficient computing capacity. The implementation includes i. a., the development of a toolbox for model-based development, the extension of the CAN bus, and the cloud's integration. Using an analysis tool called pulseAT, which was developed within the scope of this work, the system utilization of all distributed system computers is monitored, collected at a central location, and analyzed. Also, the response time of the individual computers and the real-time fulfillment are checked. This increases the reliability of the mobile robot.

The evaluation of the conceptual model and experimental implementation in the form of DAEbot shows that both the realization of a distributed system and the OCM structure with the cloud's integration are useful for mobile robots. Furthermore, an experimental framework for distributed robotic systems was compiled for future projects.

Gedruckt auf alterungsbeständigem Papier (gemäß DIN EN ISO 9706).

Inhaltsverzeichnis

1 Einführung

Die vorliegende Dissertation beschäftigt sich mit verteilten Systemarchitekturen für mobile Roboter. Die Einführung beschreibt die Forschungsfragen (Kapitel 1.1) und die Relevanz der Forschungsfragen 1.2. Zudem werden in Kapitel 1.3 die Kernaussage, sowie die Hypothesen (Kapitel 1.4) herausgestellt. Das Kapitel schließt mit der Beschreibung des Aufbaus und der verwendeten Methodik dieser Dissertation (Kapitel 1.5).

1.1 Forschungsfragen

Mobile Roboter umfassen ein breites Spektrum verschiedener Geräteklassen, von Saugrobotern bis hin zu fliegenden Robotern in sogenannten Unmanned Arial Vehicle (UAV)[1]-Schwärmen. Zunächst wird geprüft, ob die große Varianz an unterschiedlichen mobilen Robotern in generische Anforderungen an eine Systemarchitektur zusammengefasst werden können. Weiterhin wird untersucht, ob ein verteiltes Architekturkonzept diese Anforderungen ganzheitlich abbilden kann und ob hieraus eine allgemeingültige Systemarchitektur für mobile Roboter entwickelt werden kann. Des Weiteren wird untersucht, ob dieses Architekturkonzept in der Praxis für die Entwickler*innen mobiler Roboter umsetzbar ist. Das praxisnahe Experiment wird ebenfalls zur Validierung der Vor- und Nachteile einer verteilten Systemarchitektur für mobile Roboter genutzt. Hierbei wird beispielsweise geprüft, ob und wie mobile Roboter als verteiltes System umgesetzt werden können, beispielsweise wie die Kommunikation in einem verteilten System innerhalb eines mobilen Roboters bestenfalls implementiert werden würde und welche Komponenten an welcher Stelle in der Architektur in der Praxis zum Einsatz kommen sollten. Weiterhin stellt sich die Frage, ob und wie die Forschung in Bereichen wie eingebetteten Systemen[2] oder Cyber-Physical Systems (CPS)[3] in die Entwicklung mobiler Roboter integriert werden kann und wie die Optimierung eines mobilen Roboters umgesetzt wird.

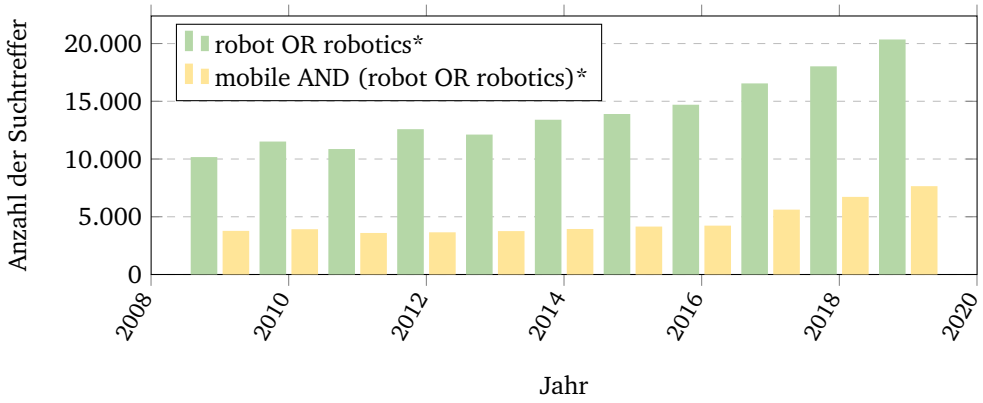
[1] dt. Umbenanntes Luftfahrzeug
[2] engl. Embedded Systems
[3] dt. Cyper-physische Systeme

1.2 Relevanz der Forschungsfragen

Die mobile Robotik ist ein interdisziplinäres Forschungsfeld, aus Informatik (Software), Elektrotechnik (Hardware) und Maschinenbau (Mechanik) und erfreut sich steigender Relevanz in wissenschaftlichen Veröffentlichungen.

1.2.1 Mobile Robotik in der Wissenschaft

Die digitale Bibliothek IEEE Xplore [O1][4,5] erzielt bei der Suche nach *„robot OR robotics“* in den Metadaten ca. 240.000 Suchtreffer. Etwa 77.000 Suchtreffer entfallen hierbei auf die Suche nach mobilen Robotern (*“mobile AND (robot OR robotics)“*).



*Suchmatrix in IEEE Xplore (in den Metadaten im August 2020)

Abbildung 1.1: Relevanz der mobilen Robotik in der Wissenschaft

Abbildung 1.1 macht deutlich, dass die Verbreitung der Forschung im Bereich Robotik im Verlauf der letzten zehn Jahre nahezu konstant ansteigt. Die Relevanz der mobilen Robotik in wissenschaftlichen Veröffentlichungen ist hingegen zwischen 2009 und 2016 fast gleichbleibend und steigt seitdem jährlich stark an. Das zeigt auf der einen Seite, dass das Forschungsgebiet *mobile Robotik* relevanter ist als je zuvor und auf der anderen Seite, dass es zunehmend Lösungen und Forschungsergebnisse in diesem Bereich in Form von Veröffentlichungen gibt.

[4] IEEE Xplore ist eine relevante Quelle für Literaturrecherche im Bereich Informatik und Ingenieurwesen. Die IEEE Bibliothek umfasst alle Publikationen, welche in Verbindung mit dem IEEE veröffentlicht wurden, z. B. aus Konferenzen, Journal, Magazinen, Bücher, Kursen oder Standards.

[5] Online Rerenzen werden mit dem Prefix **O** *(für Online)* kenntlich gemacht ([O *Nr.*]).

1.2.2 Komplexität

Mobile Roboter sind u. a. ein relevanter Forschungsbereich, da die Entwicklung und Wartung von diesen oder auch anderen software-intensiven komplexen technischen Systemen eine Herausforderung für Entwickler*innen ist und somit Herausforderungen und Lösungen zur Beherrschung der Komplexität erforscht werden müssen. Mobile Roboter gehören zu solch komplexen Systemen, da sie typischerweise eine große Anzahl an Sensoren und Aktuatoren in einem Gerät vereinen, zusammen mit einer großen Anzahl an komplexen Softwarealgorithmen, wie beispielsweise diverse Methoden zur autonomen Navigation mobiler Roboter. Ein gutes Beispiel ist hierbei der humanoide Roboter NAO [O2] des französischen Herstellers SoftBank Robotics, welcher besonders in der Lehre und Forschung eingesetzt wird. NAO verfügt in der aktuellen Version V6 aus dem Jahr 2018 über 25 Freiheitsgrade, diverse Sensoren, u. a. Infrarotsensoren, Touchsensoren, zwei Kameras und vier Mikrofone, sowie zahlreiche Motoren zur Bewegung der einzelnen Gliedmaßen und Algorithmen, u. a. zur Spracherkennung [1]. Mobile Roboter sind heute bereits im Alltag vieler Menschen etabliert. Moderne Saugroboter mit einem hohen Grad an Autonomie, wie beispielsweise der S6 MaxV von Roborock [O3], sind komplexe Systeme und verfügen über diverse Sensoren wie Umfeldsensoren, Tiefenkameras und Softwarealgorithmen zur Pfadplanung oder Objekterkennung.

Die Datenverarbeitung und Ausführung üblicher, in der mobilen Robotik verwendeten, Algorithmen hat hohe Anforderungen an die Steuereinheiten dieser Roboter. Leistungsstarke Recheneinheiten haben jedoch einen vergleichsweise hohen Leistungsbedarf, insbesondere, wenn man Notebooks oder Industrie-PCs zur Steuerung dieser Roboter verbaut[6]. Energieeffizienz ist jedoch eine wichtige Anforderung an mobile Roboter, sollen sie doch möglichst lange autark benutzbar sein. Eine Lösung des Problems könnte die Nutzung des massiv wachsenden Marktes der eingebetteten Systeme in mobilen Robotern sein. Diese basieren typischerweise auf der ARM Architektur [2], welche sich durch eine gute Energieeffizienz auszeichnet. Obwohl die Leistung dieser ARM-basierten Kleinstrechner zunimmt, beispielsweise durch Methoden zur Parallelisierung der Software auf mehrere Rechenkerne, ist die Leistung nicht in jedem Fall ausreichend für die hohen Anforderungen an Ressourcen, welche diverse Algorithmen, wie beispielsweise große Partikel-Filter, benötigen. Durch die Kombination mehrerer Kleinstrechner innerhalb eines mobilen Roboters können solche Algorithmen auf verschiedene Recheneinheiten verteilt werden und von Aufgaben, wie der Verarbeitung von Sensordaten, auch örtlich getrennt werden. Die Nutzung von mehreren Recheneinheiten wird beispielsweise beim Autonomous Mini Robot (AMiRo) umgesetzt, welcher in der Basisversion drei Module mit

[6] wird nachfolgend, u. a., in Kapitel 4.2.1 erläutert

je einem Mikrocontroller[7] verwendet und u. a. mit einem ARM-basierten Modul erweitert werden kann [3].

Die Nutzung mehrere Steuereinheiten als verteiltes System innerhalb eines mobilen Roboters erhöht die Komplexität und somit auch die Anforderung an die Entwickler*innen solcher Systeme. Eine Möglichkeit diese Komplexität zu bewältigen, ist die Nutzung von bereits vorhandener Middleware[8], wie dem Robot Operation System (ROS) [4]. ROS ist der De-facto-Standard in der Robotik und beinhaltet eine große Anzahl von Bibliotheken in Form von ROS Paketen, welche frei verfügbar sind. Zudem sind Entwickler*innen dazu aufgerufen, eigene Pakete zur Verfügung zu stellen und weiterzuentwickeln. Neben diesen Paketen bietet ROS eine gesamte Infrastruktur, welche beispielsweise die Kommunikation zwischen den Paketen organisiert. Zwar bietet ROS damit einen leichten Einstieg in die Robotik, das vergleichsweise simple Zusammenfügen einzelner Funktion führt jedoch auch dazu, dass sich die Entwickler*innen weniger mit der Konzeption einer geeigneten Architektur befassen. So können Funktionen, wie beispielsweise die Visualisierung, mit sicherheitskritischen Aufgaben vermischt werden, was die Zuverlässigkeit des Systems beeinträchtigen kann. Beispielsweise aufgrund von neuen Aufgaben können mobile Roboter zudem erweitert werden, z. B. indem ggf. neue oder zusätzliche Recheneinheiten eingesetzt werden. Dies kann dazu führen, dass insbesondere verteilte Systeme historisch wachsen und somit das ursprüngliche Architekturkonzept nicht mehr eingehalten wird oder eingehalten werden kann. Maier weist in [5] auf diese Problematik im Kontext von System of Systems (SoS)[9] hin, indem er erläutert, dass der Fokus mehr auf den Schnittstellen und der Architektur des Systems, als auf den Komponenten liegen sollte.

1.2.3 Architektur

Die Bedeutung einer Architektur für den Entwurf und die Entwicklung eines technischen Systems ist sehr hoch. Auf der Architektur basiert die Struktur der Software und die Auswahl der Hardware. Zudem definiert die Architektur die Interaktionen den einzelnen Komponenten untereinander und mit Einflüssen von außen. Bass, Clements und Kazman erläutern in *Software Architecture in Practice* [6, S. 25 ff.] detailliert, weshalb eine Architektur so wichtig ist. Hierbei wird deutlich, dass die Auswahl und die Umsetzung einer geeigneten Architektur Grundvoraussetzung für ein erfolgreiches Projekt ist. Eine Architektur ermöglicht ein funktionierendes Projekt, kann dieses jedoch auch verhindern. Entscheidungen während des Architekturentwurfs sind zu Beginn eines Projektes zu treffen, beeinflussen diesen jedoch fundamental während der gesamten Nutzung.

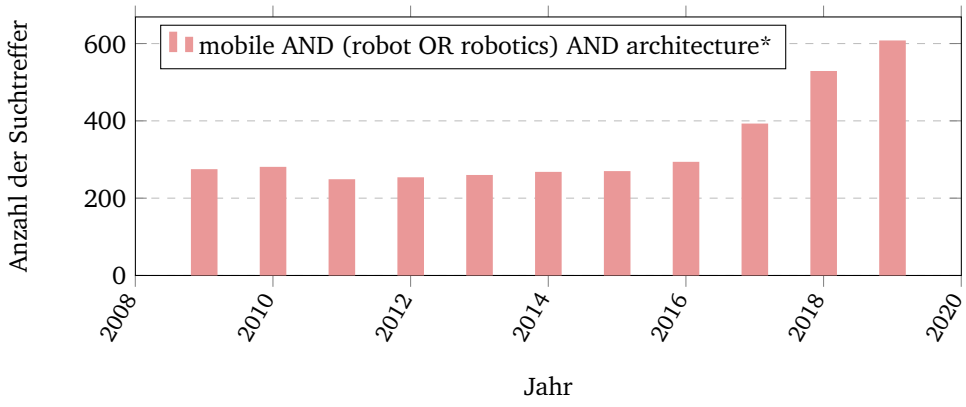
[7] engl. Micro Controller Unit (MCU)

[8] Eine Middleware (dt. Zwischenanwendung) ist eine Softwareschicht, welche Funktionen wie die Datenverwaltung oder Kommunikation für Applikationen bereitstellt und zwischen jenen Applikationen vermittelt.

[9] *SoS* beschreibt ein Verbund aus Systemen, welches zusammen ein neues Gesamtsystem bildet.

Besonders bei komplexen Systemen wie mobilen Robotern, hilft die Architektur dabei das Gesamtsystem in kleinere Teile zu zerlegen und die Komplexität somit zu verringern. Außerdem wird Entwickler*innen von mobilen Robotern mit einer klar definierten Architektur dabei geholfen, ihren Fokus und ihre Kreativität zur Lösung von Problemen und auf Details zu richten. Weiterhin zeigen Bass, Clements und Kazman, dass eine Architektur so umgesetzt werden sollte, dass sie auch auf andere Projekt übertragen werden kann und so ggf. den Kern einer ganzen Projektfamilie bilden kann. Hierbei kann z. B. eine Schichtenarchitektur[10] eingesetzt werden. Bei diesem Ansatz werden einzelne Komponenten, beispielsweise aufgrund ihrer Aufgabe, in verschiedene Schichten verteilt. Diese Komponenten agieren zum Teil getrennt voneinander und können hierbei hierarchisch, d. h. von den umliegenden Schichten abhängig sein.

Die Anforderungen an eine Systemarchitektur von mobilen Robotern ist hierbei aufgrund der vielfältigen Einsatzmöglichkeiten und der diversen Anforderungen, wie Zuverlässigkeit, Sicherheit oder Leistungsfähigkeit, besonders hoch. Durch den Einsatz eines verteilten Systems innerhalb eines Roboters werden die Anforderungen an eine Systemarchitektur für einen mobilen Roboter nochmals erhöht.



*Suchmatrix in IEEE Xplore (in den Metadaten im August 2020)

Abbildung 1.2: Wissenschaftliche Relevanz der Architektur in der mobilen Robotik

Eine allgemeingültige Standardarchitektur oder eine De-Facto Standardarchitektur für mobile Roboter gibt es aktuell nicht, wie beispielsweise das erhöhte Aufkommen von Suchtreffern für *„mobile AND (robot OR robotics) AND architecuture“* in den letzten Jahren zeigt (siehe Abbildung 1.2). Von insgesamt ca. 6.300 Suchtreffern in wissenschaftlichen

[10] wird nachfolgend in Kapitel 2.1.4 erläutert

Veröffentlichungen entfallen allein 10% auf das Jahr 2019. Während der Trend von 2009 bis 2015 insgesamt relativ konstant, in Teilen sogar negativ war, steigt die Anzahl an Suchtreffern seit 2016 jährlich sehr stark an. Dieses übertrifft auch bei weitem den Anstieg der Suchtreffer zu mobilen Robotern allgemein (siehe Abbildung 1.1). Die Forschung an Architekturen für mobile Roboter ist entsprechend aktuell und relevant und kann auch aufgrund der zahlreichen divergierenden Architekturvorschläge für mobile Roboter (einiger dieser Architekturkonzepte werden nachfolgend in Kapitel 2 - „Architekturen und verteilte Systeme“ vorgestellt) als bisher nicht endgültig gelöst betrachtet werden.

Für das Design einer allgemeingültigen Architektur für mobile Roboter muss zunächst **der** mobile Roboter definiert werden. Wie bereits erwähnt, umschließt die mobile Robotik eine große Varianz verschiedener Roboter, von fahrend, schwimmend bis fliegend, von Nano-Robotern bis zu mobilen Lastenrobotern. Die Definition eines archetypischen mobilen Roboters ist zum Zeitpunkt dieses Projektes nicht verfügbar, sodass diese selbst erarbeitet wird. Dieser typische mobile Roboter dient als Grundlage zur Ableitung von typischen Fähigkeiten und Anforderungen an die Systemarchitektur mobiler Roboter.

1.2.4 Entwicklungsprozess

Eine generische Systemarchitektur für mobile Roboter kann genutzt werden, um den Entwicklungsprozess von mobilen Robotern weiter zu vereinfachen, bzw. in weiteren Teilen zu verallgemeinern. Der Entwicklungsprozess für mobile Roboter in verteilter Systemarchitektur ist vereinfacht in Abbildung 1.3 dargestellt und entspricht dem Entwicklungsprozess für die Entwicklung von verteilten Systemen im Automobilsektor, wie u. a. in [7] gezeigt wird.

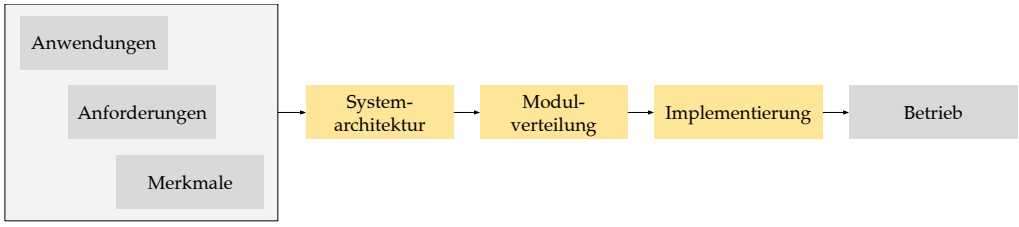


Abbildung 1.3: Vereinfachter Entwicklungsprozess von mobilen Robotern in verteilter Systemarchitektur

Die Kette des Prozesses beginnt mit einer Anforderungsdefinition für den spezifischen Anwendungsfall. Hieraus werden beispielsweise auch die Fähigkeiten und Merkmale des zu entwickelnden Roboters festgelegt. Bestenfalls wird nachfolgend eine Systemarchitektur spezifiziert. Dieser Schritt ist in gelb eingefärbt, da es dazu bereits Einzellösungen gibt.

Handelt es sich um ein verteiltes System, so folgt typischerweise nachfolgend die Auswahl der Komponenten und die Verteilung der Hard- und Software auf diese. Dazu und zur Implementierung gibt es zahlreiche Beispiele, weshalb diese Schritte ebenfalls gelb dargestellt werden. Im hier dargestellten Entwicklungsprozess sind sämtliche Testprozesse aus Darstellungsgründen entfallen, sodass der Entwicklungsprozess mit dem Betrieb des mobilen Roboters endet.

Eine abstrakte Definition der typischen Anwendungen, Anforderungen und Merkmale eines mobilen Roboters erfolgt in Form einer Taxonomie. Die Taxonomie ist ein Verfahren oder Modell zur Einordnung oder Klassifizierung in ein System. Die hier zu erarbeitende Taxonomie für mobile Roboter beschreibt u. a. die Klassen und typische Merkmale von mobilen Robotern. Der mobile Roboter wird damit abstrahiert und verallgemeinert, d. h. es wird nicht eine einzige Anwendung und die dazu spezifizierten Anforderungen und Merkmale betrachtet, sondern die mobile Robotik als abstraktes Modell mit typischen Anforderungen und Merkmalen. Mittels der Taxonomie kann auch die Systemarchitektur verallgemeinert und somit für sämtliche typischen mobilen Roboter eingesetzt werden.

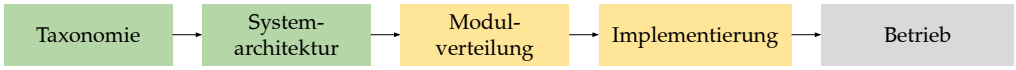


Abbildung 1.4: Vereinfachter Entwicklungsprozess von mobilen Robotern mit allgemeingültiger Systemarchitektur

In Abbildung 1.4 sind diese beiden Entwicklungsschritte grün dargestellt, da die Ergebnisse der vorliegenden Arbeit diese Schritte stark vereinfachen, bzw. eine Vorlage für eine typische Systemarchitektur für mobile Roboter bilden sollen. Die Modulverteilung und Implementierung kann nicht vollständig abstrahiert werden. Durch die Einführung einer allgemeingültigen Systemarchitektur können hier jedoch konkrete Hinweise zur Umsetzung der Verteilung und zur Implementierung, wie der Kommunikation, gegeben werden.

Die Entwicklungskette aus der Definition der Anforderungen, der Spezifikation der Architektur und dem Design des Systems (in Abbildung 1.4 aufteilt in *Modulverteilung* und *Implementierung*) bzw. der Konstruktion beschreibt Rozanski und Woods in [8, S. 88] als *Three Peaks Model*[11] (siehe Abbildung 1.5). Dieses basiert auf dem *Twin Peak Model* von Bashar Nuseibeh [9], welches um die dritte Phase, der Konstruktion des Systems, erweitert wird. Das Three Peak Model zeigt die Abhängigkeit dieser drei Phasen voneinander. So ist beispielsweise die Architektur abhängig von den Anforderungen

[11] dt. Drei-Gipfel-Modell

und vom Design des Systems. Die Schleifen weisen darauf hin, dass Phasen, hier Gipfel, nicht isoliert betrachtet werden können, sondern Auswirkungen aufeinander haben. Die drei Phasen werden nicht strikt seriell hintereinander abgearbeitet, denn Änderungen am Design können beispielsweise Anpassungen an der Architektur nach sich ziehen. Hierbei nimmt der Detailgrad in allen Gipfeln mit der Zeit zu. Das *Three Peaks Model* zeigt u. a., dass es bei der Verallgemeinerung einer Systemarchitektur für mobile Roboter notwendig ist, die Anforderungen möglichst allgemeingültig zu definieren, damit die spezifizierte Architektur allgemeingültig sein kann. Spezifische oder untypische Anforderungen an die Systemarchitektur eines mobilen Roboters sollten in der allgemeingültigen Systemarchitektur einzubinden sein, ohne dass das Grundkonzept der Architektur geändert werden muss. Weiterhin zeigt das *Three Peaks Model* die Notwendigkeit einer exemplarischen Systemimplementierung zur Absicherung der praxistauglich der allgemeingültigen Architektur, da das Design des Roboters selbstverständlich Einfluss auf die Spezifikation der Architektur hat.

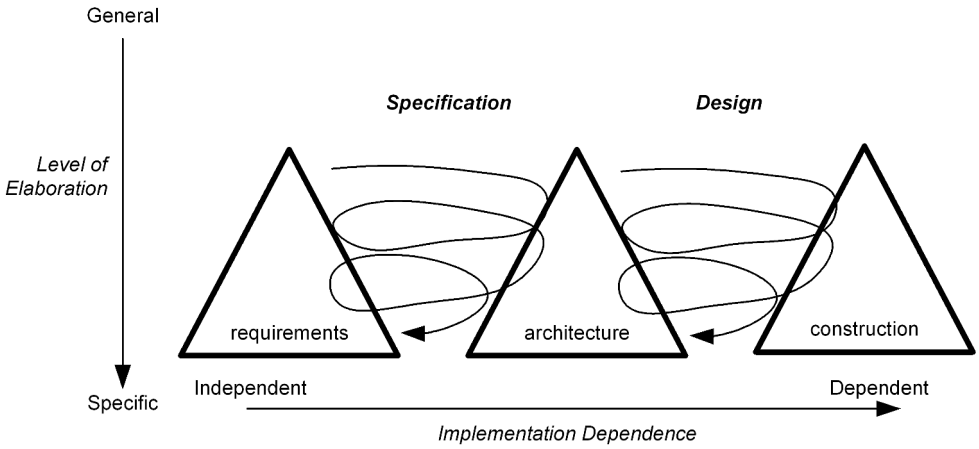


Abbildung 1.5: *Three Peaks Model* zur Definition von Architekturen [8, S. 88]

1.2.5 Analyse und Optimierung

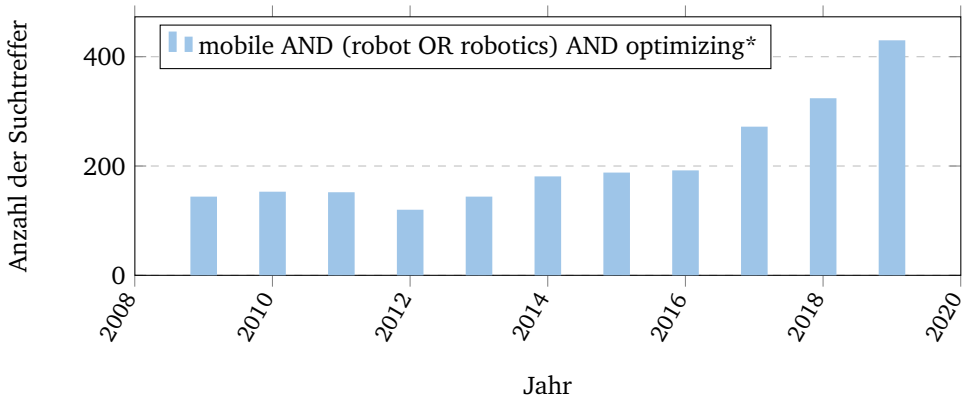
Beim Einsatz von verteilten Systemen sind die Verifikation, Validierung und der Status des Systems, beispielsweise durch *Überwachungs*[12]-Methoden komplex. Als Validierung wird das Überprüfen, ob das entwickelte System den vorgegebenen Anforderungen entspricht, bezeichnet. Ein erfolgreich validiertes System erfüllt seine Nutzungsziele. Bei der

[12] engl. Monitoring

Verifikation wird geprüft, ob das System korrekt umgesetzt wurde, beispielsweise durch das Überprüfen des Zeitverhaltens. Ein verifiziertes System erfüllt seine Spezifikationen. Während bei der Verwendung einer einzelnen Recheneinheit frei verfügbare Tools, wie beispielsweise diverse Task-Manager, auf einen Blick Aufschluss über den internen Systemzustand, z. B. in Form von Central Processing Unit (CPU)[13]-Auslastung oder Random-Access Memory (RAM)[14] Belegung geben, sind diese Überwachungs-Daten im verteilten System auf die einzelnen Recheneinheiten verteilt. Zur Beurteilung des Gesamtzustands des Systems müssen die Informationen der einzelnen Rechner jedoch zentral verfügbar sein und zusammen mit der Überprüfung des Status der Kommunikation analysiert werden. Zudem sind solche internen Systeminformationen nicht für jede Prozessor-Familie verfügbar. So gibt es u. a. für die Arduino Plattform typischerweise keine aktuellen Systeminformationen. Insbesondere in der mobilen Robotik ist die Überwachung jedoch wichtig, da Anforderungen wie die Sicherheit (Safety) und Zuverlässigkeit stark vom Systemzustand abhängen. Wird beispielsweise der Task zum Stoppen der Motoren nicht rechtzeitig vor einem Hindernis abgearbeitet und der Roboter gestoppt, weil beispielsweise die harten Echtzeitbedingungen durch eine zu hohe CPU-Last nicht erfüllt werden, so kann dies zu Unfällen, Defekten und gefährlichen Situationen führen.

Neben dem aktuellen Systemzustand fehlt es in verteilten Systemen an einer dauerhaften Überwachung und einer Analyse des Systemzustands[15]. Diese Überwachung ist notwendig, um die zu Beginn des Entwicklungsprozesses getroffenen Entscheidungen bezüglich der Auswahl der Hardwarekomponenten und der Modulverteilung zu überprüfen und das System ggf. zu optimieren. Mobile Roboter können in unterschiedlichen Umgebungen eingesetzt werden. So werden, insbesondere bei modularen Systemen, Sensoren ausgetauscht, um die neuen Umweltbedingungen abbilden zu können. Dieses kann die Datenverarbeitung stark beeinflussen, sodass ggf. eine Recheneinheit gegen eine leistungsstärkere Version getauscht werden muss. Zur Optimierung, also zur Verbesserung des Systems bedarf es neben dem aktuellen Systemzustand, zusätzlich einer Analyse der bisher aufgetretenen Systemdaten, sowie bestenfalls einer Vorhersage für zukünftig mögliche Zustände. Die Optimierung mobiler Roboter ist auf diversen Ebenen im System und mit unterschiedlichen Methoden möglich. Die Relevanz der Optimierung mobiler Roboter in wissenschaftlichen Veröffentlichungen ist in den letzten Jahren stark gestiegen (siehe Abbildung 1.6), sodass die Verbreitung dieses Forschungsgebiets zwar weiter zunimmt, die absoluten Zahlen jedoch im Vergleich zu den Suchtreffern bezüglich mobiler Robotik und Architekturen nur bei der Hälfte liegen.

[13] dt. Zentrale Recheneinheit
[14] dt. Direktzugriffsspeicher (typischerweise eingesetzt als Arbeitsspeicher)
[15] wird nachfolgend in Kapitel 6.3 erläutert



*Suchmatrix in IEEE Xplore (in den Metadaten im August 2020)

Abbildung 1.6: Wissenschaftliche Relevanz der Optimierung in der mobilen Robotik

Die Analyse des verteilten Systems ist in zwei Phasen besonders hilfreich. Phase I (siehe Abbildung 1.7) erfolgt während der Tests der Implementierungen. Hier ist es notwendig zu analysieren, ob die gewählte Hardware der erwarteten Auslastung entspricht und ob die Anforderungen, wie das Einhalten der Echtzeitfähigkeit auch in der Praxis gegeben ist. In dieser Phase wird die Modulverteilung und u. U. auch die Auswahl der Komponenten mehrfach angepasst. Die Tests bzw. die Analysen sollten in dieser Phase, wie im V-Model [10, S. 83 ff.] üblich, auf verschiedenen Abstraktionsebenen, vom Modultest bis zum Test des Gesamtsystems, durchgeführt werden. Die ersten Abstraktionsebenen können hierbei laut Engblom [11] in Simulationen durchgeführt werden. Das Ergebnis der Analyse während der Implementierung führt zu einem statischen Ergebnis, beispielsweise einer initialen Auswahl der Hardware, welches ggf. im Laufe der Betriebszeit nicht weiter überprüft wird.

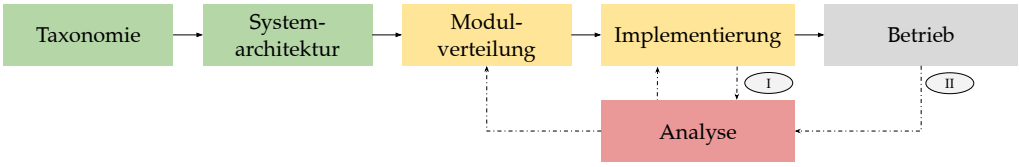


Abbildung 1.7: Einbindung der Analyse in den Entwicklungsprozess von mobilen Robotern

Phase II der Analyse läuft ständig und dauerhaft, wenn der mobile Roboter in Betrieb ist. Hier wird die Analyse u. a. dafür genutzt, um auf Engpässe im Systemzustand hinzuweisen, beispielsweise, wenn der RAM droht vollzulaufen. Überdimensionierte Hardware kann hierbei auch erkannt werden. Anhand der Information kann sowohl die Implementierung

angepasst und, falls nötig ggf., die Modulverteilung (inkl. der Auswahl der Komponenten) überarbeitet werden. Diese Systemüberwachung und Analyse kann somit zur dynamischen und adaptiven Optimierung eines technischen Systems verwendet werden. Weiterhin kann die dauerhafte Analyse, welche auch als *Post-deployment data usage*[16] bezeichnet wird, laut Holmström Olsson und Bosch [12] u. a. zur Verbesserung von nachfolgenden Produkten und Systemen verwendet werden.

1.2.6 Zusammenfassung

Die vorliegende Arbeit umfasst die Definition einer Taxonomie (siehe (1) in Abbildung 1.8) und die Entwicklung einer allgemeingültigen Systemarchitektur (2) für mobile Roboter. Die Taxonomie und die Systemarchitektur werden zunächst theoretisch erarbeitet. Um Hinweise und eine Hilfestellung zur Modulauswahl und Modulverteilung (3a) und zur Implementierung zu geben (3b) wird die Systemarchitektur im Rahmen der vorliegenden Dissertation experimentell umgesetzt und analysiert. Hierzu wird die gesamte Kette an einem Demonstrator erprobt und evaluiert. Die vorliegende Arbeit umfasst zudem die praxisnahe Analyse von verteilten Systemen, wozu ein entsprechendes Tool[17] implementiert wird (4).

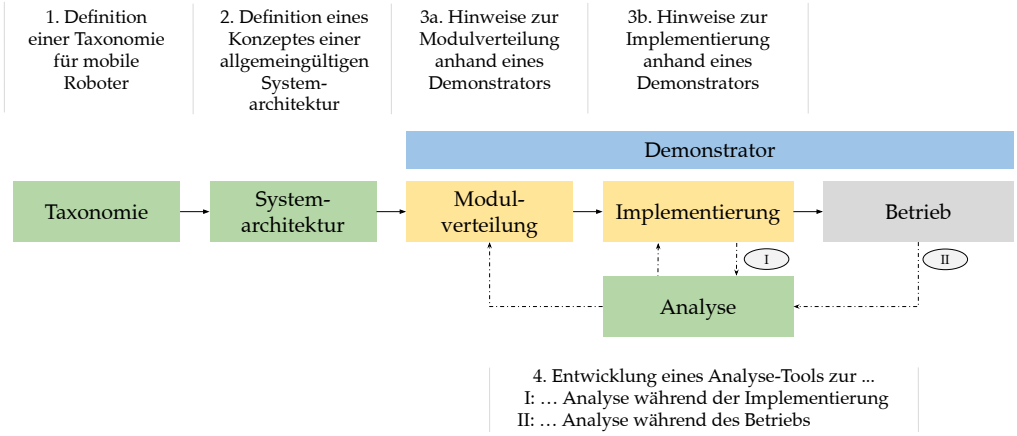


Abbildung 1.8: Zusammenfassung der gesamten Entwicklungskette von mobilen Robotern

[16] dt. Verwendung von Daten nach Übertragung/ Auslieferung

[17] Als Tools (dt. Werkzeuge) werden Hilfswerkzeuge, wie z. B. Programme oder Skripte bezeichnet, welche u. a.bei der Entwicklung von technischen Systemen eingesetzt werden.

1.3 Kernaussage

Die Anforderungen an eine allgemeingültige Architektur für mobile Roboter werden durch eine dauerhaft überwachte und hierarchisch verteilte Systemarchitektur erfüllt.

1.4 Hypothesen

Die Kernaussage stützt sich prinzipiell auf drei Teilhypothesen, welche folgende Teilgebiete betreffen: (1) Die Anforderungen an eine Systemarchitektur für mobile Roboter, (2) die Struktur der Systemarchitektur, (3) die Analyse von verteilten Systemen.

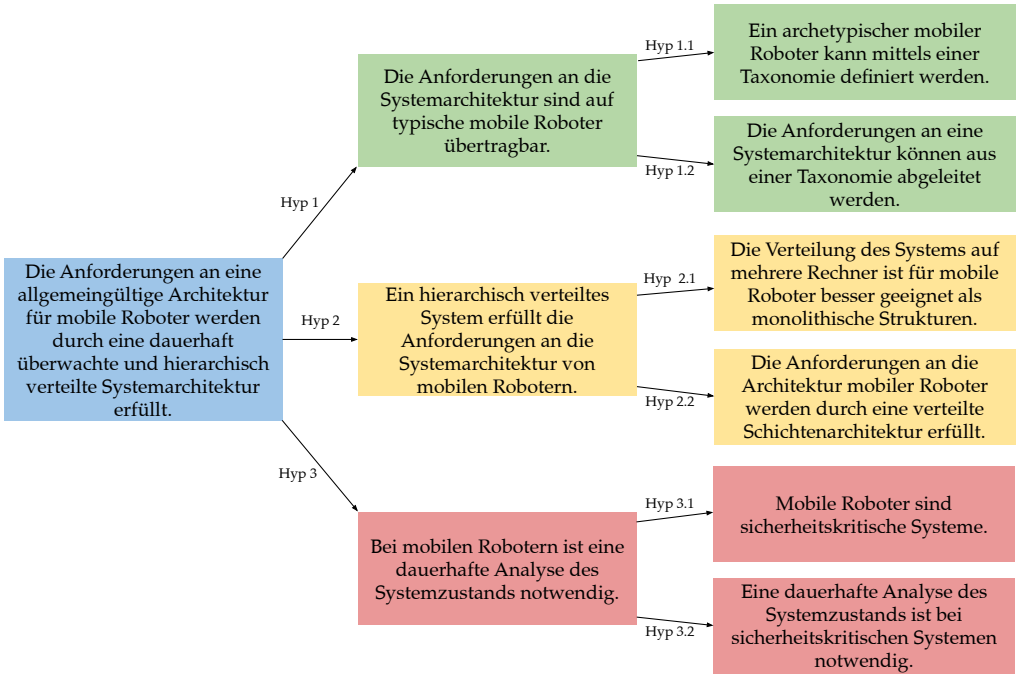


Abbildung 1.9: Hypothesenbaum

In Abbildung 1.9 sind links die Kernaussage und in der Mitte die drei Hypothesen der genannten Teilgebiete dargestellt. Rechts werden die drei Hypothesen wiederum durch Teilhypothesen (Hyp 1.1. bis Hyp 3.2) unterfüttert. Dieser Hypothesenbaum enthält in

diesem Fall die ersten drei Ebenen, kann jedoch um weitere Ebenen erweitert werden. Ziel dieser Dissertation ist es, die aufgestellten Hypothesen nachweislich zu verifizieren oder zu falsifizieren.

1.5 Methodik und Aufbau

Die Methodik und der Aufbau der Dissertation orientieren sich stark an den *Schritten zur problemlösungsorientierten wissenschaftlichen Forschung* (sehe Abbildung 1.10) von Dresch, Lacerda und Antunes aus *Design science research: A method for science and technology advancement* [13].

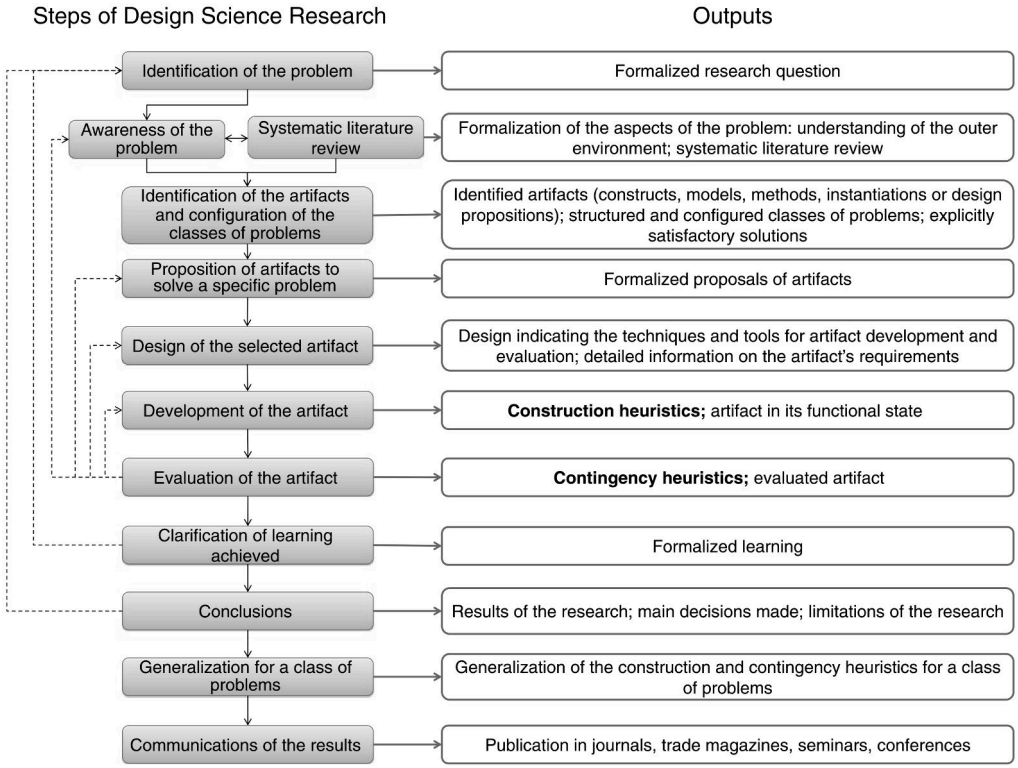


Abbildung 1.10: Schritte zur problemlösungsorientierten wissenschaftlichen Forschung [13, S. 124]

Der erste Schritt, die Identifikation der Forschungsfrage(n), erfolgte bereits in der Einführung (Kapitel 1). Die Problemstellung wurde ebenfalls bereits erläutert, wird jedoch im nachfolgenden Kapitel mit dem Stand der Technik von Architekturen und verteilten Systemen in der mobilen Robotik (Kapitel 2) intensiviert. Hierbei werden zunächst die Grundlagen von Architekturen erläutert. Nachfolgend wird der Stand der Technik von Teilthemen, wie die verteilten Systeme oder das Cloud Computing dargestellt. Nach der Vorstellung des Operator-Controller-Modul (OCM)-Ansatzes folgt der Stand der Technik von Architekturen für mobile Roboter.

In Kapitel 3 „Anforderungen an die Systemarchitektur von mobilen Robotern“ wird zunächst ein Überblick und eine Definition über die mobile Robotik gezeigt. Nachfolgend wird eine Literaturrecherche durchgeführt, um hieraus einen typischen mobilen Roboter zu beschreiben. Zur Recherche werden zunächst drei Arten von Quellen herangezogen. Buchquellen, Übersichtarbeiten und Referenzsysteme der mobilen Robotik, aus welchen auch der Stand der Technik erläutert wird. Anhand dieser Quellen wird eine systematische Literaturrecherche durchgeführt, die dazu dient, einzelne Merkmale von mobilen Robotern zunächst zu identifizieren und nachfolgend zu quantifizieren. Hieraus erfolgt die Definition einer Taxonomie für mobile Roboter. Die Anforderungen an eine allgemeingültige Systemarchitektur für mobile Roboter wird aus der Taxonomie abgeleitet.

Kapitel 4 „Konzeptionelles Modell einer Systemarchitektur für mobile Roboter“ zeigt die entwickelte Systemarchitektur, dessen Ausgangspunkt die zuvor definierten Anforderungen an diese sind. Hierbei beinhaltet das Konzept nicht nur die Systemarchitektur an sich, sondern setzt sich spezifisch mit jeder Anforderung und den dadurch notwendigen Maßnahmen auseinander. Weiterhin werden Hinweise, wie die Definition der Kommunikationsmethoden dargestellt, welche neben der Systemarchitektur verallgemeinert werden können. Hierbei ist das gezeigte Konzept durch diverse Iterationen (siehe Abbildung 1.10) mittels eines Demonstrators und des Analyse-Tools entstanden.

Die Umsetzung des Konzeptes in einem Experiment folgt in Kapitel 5 „Experimentelle Umsetzung der Systemarchitektur für mobile Roboter“. Hierzu wird eigens ein mobiler Roboter entworfen, welcher das Konzept möglichst vollständig umsetzt und zudem das Konzept im physischen Aufbau des Roboters zeigen soll. So sind die Ebenen des hierarchisch verteilten Architekturkonzepts ebenfalls im Aufbau des Demonstrators erkennbar. Der entwickelte Demonstrator, der „Distributed Architecture Evaluation Robot (DAEbot)“, ist als Experimentier- und Evaluationsplattform konzipiert. Einzelne Komponenten können leicht ausgetauscht werden, u. a. um unterschiedliche Architekturansätze, Modulverteilungen oder Hardware, beispielsweise Rechnertypen, zu evaluieren. Hierbei wurde sich für die Entwicklung eines gänzlich neuen Roboters entscheiden, da somit keine Vorgaben in Bezug auf die Auswahl der Komponenten und die Verteilung der Funktionen auf die Rechner des verteilten Systems eingehalten werden

müssen. Es wird die aktuelle Version des DAEbot beschrieben, welche im Rahmen des Promotionsvorhabens, aufgrund der regelmäßigen Evaluationen, angepasst wurde und dessen Anpassungen entsprechend auch Auswirkungen auf die Spezifikation der Systemarchitektur haben (siehe *Three Peak Model* in Abbildung 1.5). Neben dem Systementwurf und der Middleware, basierend auf dem konzeptionellen Modell, wird an einigen relevanten Beispielen die Softwareentwicklung exemplarisch erläutert (wie beispielsweise die Verwendung des Model-driven Software Engineering (MDSE)[18]). Das Kapitel wird mit der kurzen Beschreibung von weiteren Demonstratoren, welche mit dem konzeptionellen Modell umgesetzt wurden, beendet. Hierbei handelt es sich um eine experimentelle alternative Software für den AMiRo und ein UAV, welche für das Drittmittelprojekt Software for Robots (S4R)[19] entwickelt wird.

Da zur Umsetzung aller Anforderungen an die Systemarchitektur auch eine Überwachung des Systems notwendig ist und diese in der benötigten Form nicht vorhanden ist, wurde im Zuge der Dissertation das Tool „pulse Analysis Tool (pulseAT)“ entwickelt. Dieses Tool hat sich für den sicheren Betrieb mobiler Roboter und die Entwicklung dessen als wichtig und notwendig herausgestellt, sodass diesem Tool ein eigenes Kapitel gewidmet wird. Das Kapitel 6 - „Entwicklung eines Tools zur Analyse von verteilten Systemen“ stellt zunächst die Grundlagen der Überwachung, Analyse und Optimierung von technischen Systeme dar. Nachfolgend wird der Stand der Technik zur Analyse von verteilten System erläutert. Das Konzept von pulseAT und schließlich die Umsetzung und der Einsatz dessen mit dem mobilen Roboter DAEbot wird vorgestellt.

Kapitel 7 zeigt die Evaluation des konzeptionellen Modells, des Demonstrators und des Analyse-Tools. Hierzu werden sowohl theoretische Methoden zur Bewertung von Architekturen genutzt, als auch zur Bewertung der Umsetzung des Demonstrators, das Analyse-Tool pulseAT. Mit pulseAT werden beispielsweise die Antwortzeiten und die Systemauslastung während des Einsatzes des DAEbots ausgewertet. Weiterhin wird in Kapitel 7 geprüft, ob die Anforderungen an die Systemarchitektur umgesetzt werden konnten. Weithin wird das konzeptionelle Modell und dessen Umsetzung in den Stand der Technik eingeordnet und bewertet.

Kapitel 8 schließt die Dissertation mit einer Zusammenfassung, einem Fazit und einem Ausblick in mögliche weitere Forschungsthemen, welche an dieses Projekt anknüpfen könnten, ab.

[18] dt. Modellgetriebene Softwareentwicklung

[19] Das Forschungsprojekt S4R wird durch das Bundesministerium für Bildung und Forschung gefördert (Förderkennzeichen 13FH009IX6).

2 Architekturen und verteilte Systeme

Die Entwicklung oder Nutzung von Architekturen bzw. Architekturmustern ist laut Bass, Clements und Kazman [6, S. 25 ff.] bei der Implementierung von komplexeren technischen System unabdingbar. Dieses Kapitel beginnt mit der Definition verschiedener Architektur-Begriffe, sowie Schaltungsvarianten und Computerfamilien und grenzt diese voneinander ab (Kapitel 2.1). Besonderer betrachtet werden die Schichtenarchitektur, verteilte Architekturen (Kapitel 2.2), sowie das Cloud Computing (Kapitel 2.3). Ein Fokus liegt auf dem OCM (Kapitel 2.4), welches die Grundlage für die nachfolgend vorgeschlagenen Systemarchitektur ist. Zuletzt wird in Kapitel 2.5 der Stand der Technik bezüglich Architekturen für mobile Roboter erläutert.

2.1 Grundlagen von Architekturen und Schaltungsvarianten

Die Struktur oder Anordnung und Zuordnung von Komponenten untereinander und innerhalb eines technischen Systems wird als Architektur bezeichnet. Hierbei gibt es unterschiedliche Sichten auf das System und entsprechend zahlreiche Definitionen unterschiedlicher Architektur-Begriffe, wie beispielsweise die Rechnerarchitektur, Softwarearchitektur oder auch die Systemarchitektur.

2.1.1 Systemarchitektur

Der Begriff Systemarchitektur beschreibt eine Gesamtsicht, bzw. eine übergeordnete Sicht auf ein technisches System.

Jaakkola und Thalheim definieren in [14, S. 98] die Systemarchitektur als ein konzeptionelles Modell, welches (1) verschiedene Gesichtspunkte, definiert als Sichten auf das Modell, (2) Facetten oder Verhalten des Systems in Abhängigkeit des Umfangs und der Abstraktionsebene, (3) Einschränkungen für den Einsatz des Systems und die Beschreibung der Qualitätsanforderungen, sowie (4) die Einbettung in andere (Software-) Systeme darstellt.

ANSI/IEEE definiert die Systemarchitektur im Standard 1471-2000 [15, S. 3] als fundamentale Organisation eines Systems dargestellt in Komponenten, ihren Beziehungen zueinander, sowie in der Umgebung und den Prinzipien welche zur Entwicklung beitragen.

Abbildung 2.1 zeigt das Doppeldachmodell für den Entwurf digitaler Hardware/ Software-Systeme aus *Digitale Hardware/Software-Systeme - Spezifikation und Verifikation* von Haubelt und Teich [16, S. 14]. Als Architektur wird in der Darstellung explizit die Hardwarearchitektur bezeichnet. Die Implementierung auf Systemebene hingegen wird als Hardware/ Software-Architektur bezeichnet [16, S. 15]. Der Begriff der Systemarchitektur selbst wird hierbei nicht verwendet, kann jedoch aus Hardware/ Software-Architektur abgeleitet werden.

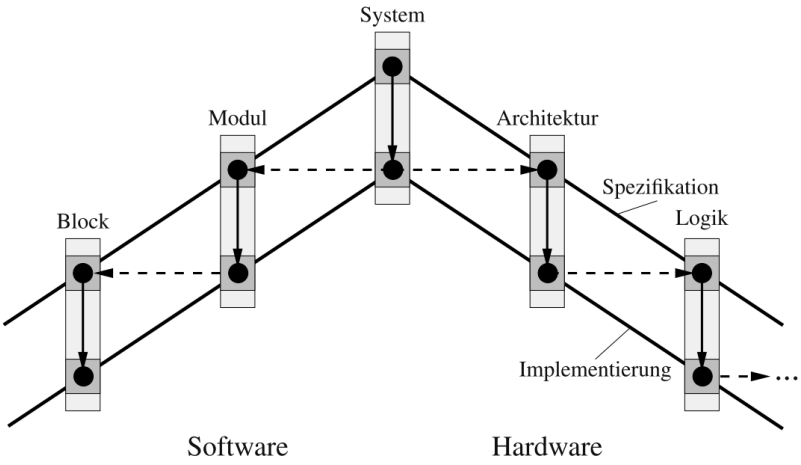


Abbildung 2.1: Doppeldachmodell für den Entwurf digitaler Hardware/ Software-Systeme [16, S. 14]

2.1.2 Rechner- und Prozessor-Architektur

Die Beschreibung der verwendeten physischen Komponenten innerhalb eines Computers wird als Rechnerarchitektur bezeichnet. Abbildung 2.2 zeigt die Rechnerarchitektur eines einfachen Computers mit CPU, Hauptspeicher und zwei General Purpose Input/ Output (GPIO)[1] Geräten.

[1] dt. Universelle Ein- und Ausgabe Schnittstelle

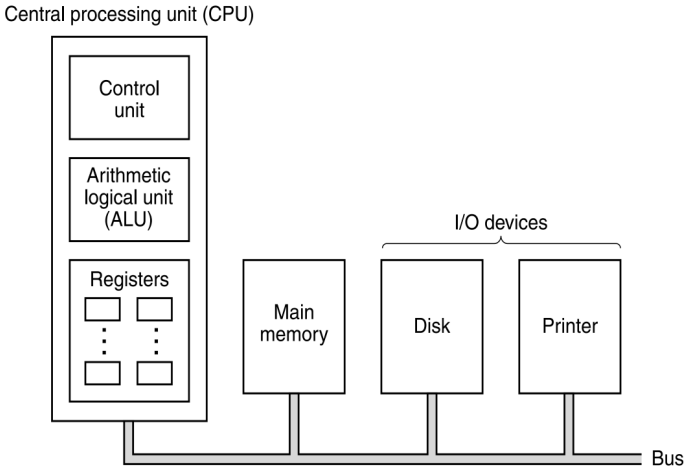


Abbildung 2.2: Beispiel einer einfachen Rechnerarchitektur nach Tanenbaum in *Structured Computer Organization* [17, S. 52]

Das Design der CPU selbst wird als Prozessorarchitektur bezeichnet. Typische Prozessorarchitekturen sind beispielsweise x86- oder ARM-Architekturen. Abbildung 2.3 zeigt die Komponenten eines ARM Cortex™-M4 Prozessors [O4].

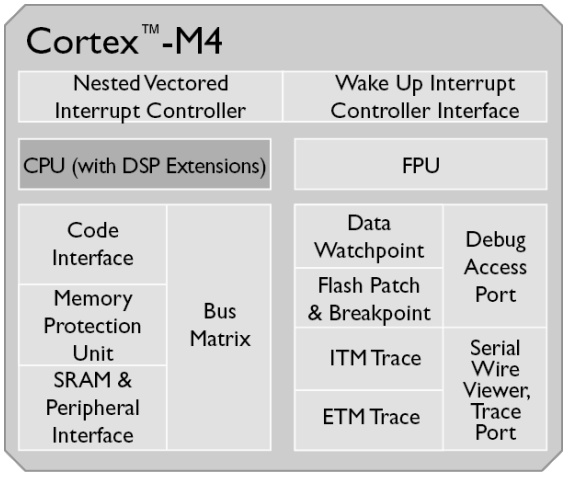


Abbildung 2.3: Komponenten eines ARM Cortex™-M4 [O4]

Die interne Organisation von Prozessoren basiert auf Referenzmodellen, wie beispielsweise der von Neumann- oder der Harvard-Architektur, welche sich u. a. durch die Art der Speicheranbindung unterscheiden. Des Weiteren können sich verschiedene Prozessorfamilien durch unterschiedliche Befehlssatzarchitekturen[2] differenzieren. Hierbei setzen heutzutage fast alle Anbieter von Prozessoren auf die sogenannte (Reduced Instruction Set Computer (RISC))-Architektur. Dabei ist der Befehlssatz so gestaltet, dass Befehle einfacher Komplexität mit hoher Taktfrequenz ausgeführt werden können.

Um energie-effizientere Systeme zu implementieren, werden, insbesondere bei Kleinstrechnern, verschiedene Komponenten einer Rechnerarchitektur lokal in einem einzelnen Chip realisiert. Diese können auch als System on Chip (SoC)[3] bezeichnet werden und sind dann mehr als nur die zentrale Steuereinheit eines Kleinstrechners, z. B. eines Smartphones. Ein SoC, wie beispielsweise der Snapdragon 820 von Qualcomm (siehe Abbildung 2.4) beinhaltet bereits die meisten üblichen Module, wie CPU, Digitaler Signal Prozessor (DSP)-Einheiten und Schnittstellen, wie Ethernet-Module innerhalb des Chips.

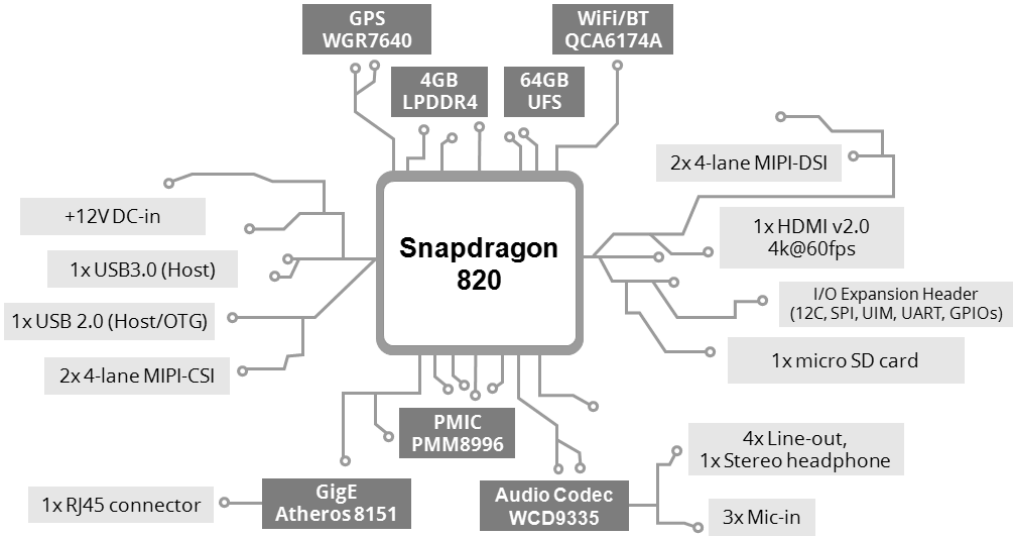


Abbildung 2.4: SoC Snapdragon 820 Blockdiagramm [O5]

[2] engl. Instruction Set Architecture (ISA)
[3] dt. Ein-Chip-System

2.1.3 Schaltungsvarianten

CPUs sind weit verbreitet und nahezu in jedem technischen Gerät vorhanden. Unter anderem durch den weit verbreiteten Befehlssatz sind sie flexibel und somit universal einsetzbar, wenngleich es Anwendungsfälle gibt, bei denen alternative Prozessorarten oder Schaltungsvarianten der klassischen CPU überlegen sind. Während CPUs beispielsweise aus vergleichsweise wenigen Prozessorkernen bestehen, welche jeweils aus einer sehr großen Anzahl von Transistoren realisiert werden, besteht beispielsweise eine Graphics Processing Unit (GPU)[4] aus viel kleineren Kernen, von denen jedoch eine deutlich größere Anzahl zur Verfügung stehen. GPUs sind für massiv parallele Datenverarbeitungsalgorithmen einzusetzen, z. B. zur Verarbeitung von 3D Grafikanwendungen. CPUs hingegen sind bei der sequentiellen Verarbeitung und bei weniger regulärem Kontrollfluss effizienter und flexibler. Heute integrieren viele CPUs bereits eine GPU als sogenannte *Onboard-GPU* [O6].

Neben CPU und GPU gibt es rekonfigurierbare Rechensysteme[5], wie z. B. die Schaltungsvariante des Field Programmable Gate Array (FPGA)[6]. Ein FPGA ist im Vergleich zur CPU oder GPU bei der Auslieferung hardwareseitig nicht starr konfiguriert und durch Befehlssätze über eine Software programmierbar. Bei einem FPGA bezieht sich das Programmieren auf die Konfiguration der Schaltung, also der physischen Bauelemente mittels einer Hardwarebeschreibungssprache. Hierbei kann diese Programmierung und somit die interne Verschaltung der Elemente geändert und andere Funktionen realisiert werden. Somit ist es möglich, Informationen deutlich schneller und effizienter als mit einer CPU zu verarbeiten, wenngleich eine FPGA Konfiguration einem spezifischen Anwendungsfall dienen sollte und nicht für allgemeine Informationsverarbeitungsprozesse geeignet ist.

Eine Alternative zum FPGA kann ein Application-Specific Integrated Circuit (ASIC)[7] sein. Ein ASIC wird speziell für einen Anwendungsfall entwickelt und die Hardware entsprechend hergestellt. Im Gegensatz zum FPGA kann die hardwareseitige Konfiguration nicht mehr geändert werden. Insofern ist die Entwicklung eines ASIC kostenintensiv. Ein ASIC ist jedoch im Vergleich zum FPGA sehr effizient, da viel weniger Bauteile für die gleiche Anwendung benötigt werden. Dieses führt auch zu günstigeren Preisen eines ASIC gegenüber eines FPGA, falls eine große Stückzahl ASICs benötigt wird.

2.1.4 Softwarearchitektur

Die Softwarearchitektur definiert Balzert in *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb* [18, S. 23] wie folgt:

[4] dt. Grafikprozessor
[5] engl. reconfigureable Computing
[6] dt. Programmierbares Logikgatter
[7] dt. Anwendungsspezifische integrierte Schaltung

> „Eine Softwarearchitektur beschreibt die Strukturen eines Softwaresystems durch Architekturbausteine und ihre Beziehungen und Interaktionen untereinander sowie ihre physikalische Verteilung. Die extern sichtbaren Eigenschaften eines Architekturbausteins werden durch Schnittstellen spezifiziert."

Balzert [18, S. 23 ff.] zeigt in Abbildung 2.5 vier grundsätzliche Sichten auf eine Softwarearchitektur. Die statische Sicht stellt die statische Struktur dar. Bei der Laufzeitsicht liegt der Fokus auf dem Ablauf der vorhandenen Prozesse und dessen Zusammenspiel. Die Kontextsicht zeigt die Einbettung des Systems in die Umgebung. Die Verteilungssicht nimmt Bezug auf die Hardwarearchitektur und zeigt, welche Architekturbausteine auf welchen Hardwarekomponenten ablaufen. Hierbei wird, im Gegensatz zur Systemarchitektur, jedoch die Hardware selbst nicht definiert und spezifiziert.

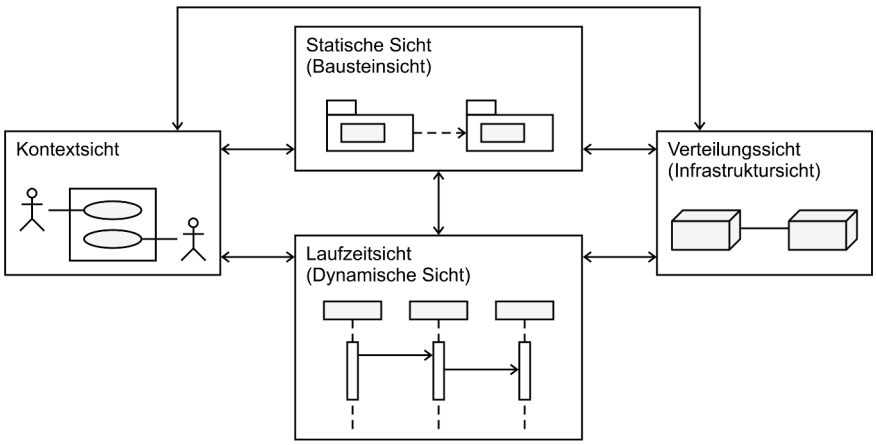


Abbildung 2.5: Sichten auf eine Softwarearchitektur [18, S. 24]

Die Darstellung einer Softwarearchitektur ist stark vom Abstraktionsgrad abhängig. Die Abstraktion erfolgt typischerweise in fünf Bausteinarten (hierarchisch sortiert): Subsysteme, Komponenten, Frameworks[8], Pakete und Klassen.

[8] Ein (Software) Framework (dt. Rahmenstruktur) liefert generische Funktionalitäten, welche Entwickler*innen in eigenen Applikationen wiederverwenden können. Frameworks sind üblicherweise domänenspezifisch und können u. a. Compiler, Bibliotheken und Tools umfassen.

Eine Softwarearchitektur kann mit verschiedenen Modellen bzw. Mustern strukturiert werden. Ein solches Muster ist beispielsweise die Schichtenarchitektur, bei der Subsysteme in verschiedene horizontale Schichten unterteilt werden. Softwarearchitekturen können beispielsweise mit Unified Modeling Language (UML) Diagrammen visualisiert werden. Abbildung 2.6 stellt die statische Sicht einer Schichtenarchitektur dar. Die Darstellung zeigt die Organisation der Subsysteme in drei horizontale Schichten. Die Komponenten der obersten Schicht (hier die Komponenten 3.1, 3.2, 3.3) haben bei den Verbindungen `a` und `b` keinen direkten Zugriff auf die Komponenten der untersten Schicht (hier die Komponenten 1.1, 1.2, 1.3, 1.4). Eine direkte Kopplung zeigt Verbindung `c`. Jede Schicht hat typischerweise eine eigene Aufgabe und fest definierte Schnittstellen zu benachbarten Schichten. [18, S. 46 ff]

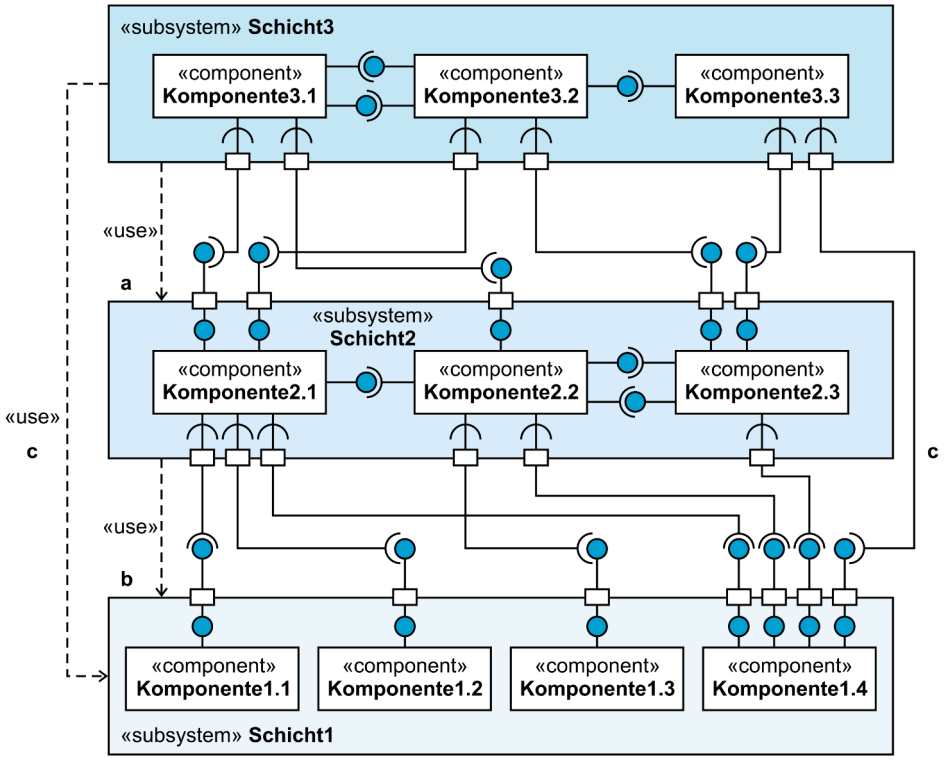


Abbildung 2.6: Beispiel einer Schichten-Architektur, dargestellt als UML-Komponentendiagramm [18, S. 49]

2.2 Grundlagen von verteilten Systemen

Softwarearchitekturen können sowohl auf einem Rechner ausgeführt (monolithisches System), als auch auf verschiedene Rechner verteilt werden (verteiltes System). Hierbei wird die logische Architektur (siehe Abbildung 2.7 am Beispiel einer Schichtenarchitektur) auf die physische Architektur abgebildet. Die technische Infrastruktur gibt eine Gesamtsicht auf das System, welche auch als Systemarchitektur (vgl. Kapitel 2.1.1) bezeichnet wird.

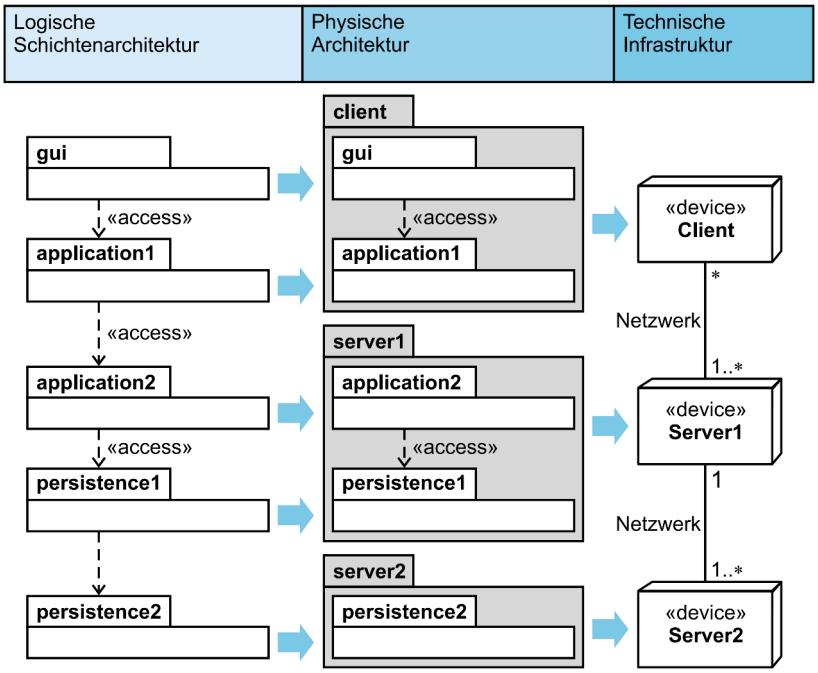


Abbildung 2.7: Verteilung von logischen Schichten auf die Systemarchitektur [18, S. 191]

Die Entwicklung eines verteilten Systems erhöht die Komplexität, da neben den Applikationen eine Middleware notwendig ist oder mindestens ein gemeinsames Kommunikationsmodell benötigt wird, um Daten zwischen den einzelnen Rechnern auszutauschen (siehe Abbildung 2.8). Die Komplexität erhöht sich abermals, wenn es sich bei einem verteilten System um ein heterogenes verteiltes System handelt. Bei heterogenen verteilten Systemen werden unterschiedliche Rechner eingesetzt, die oft mit unterschiedlichen Betriebssystemen (oder gänzlich ohne Betriebssystem) betrieben werden.

Somit muss die zu entwickelnde Middleware entsprechend kompatibel zu den eingesetzten Rechnern sein. Homogene verteilte Systeme hingegen setzen eine Vielzahl Bauart-gleicher Rechner ein, sodass es in der Regel nicht zu Kompatibilitätsproblemen kommt.

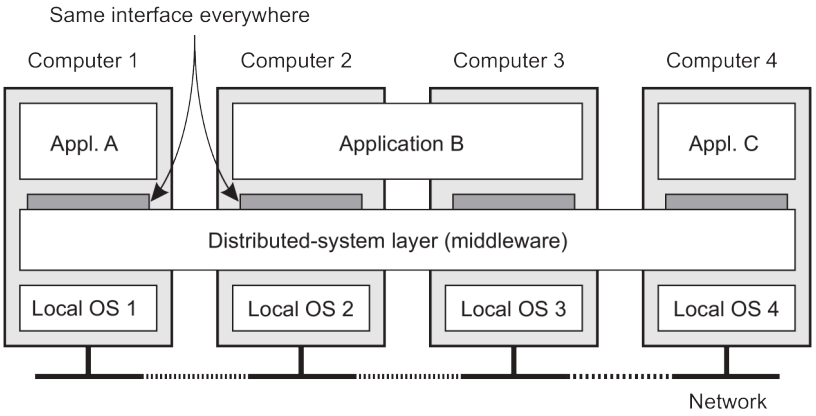


Abbildung 2.8: Verteiltes System mit Middleware [19, S. 5]

Verteilte Systeme können, ähnlich wie Softwarearchitekturmuster, verschiedenen Architekturstilen folgen. In *Distributed Systems - Principles and Paradigms* von Steen und Tanenbaum [19, S. 56 ff.] wird prinzipiell in Systemorientierte Architektur (SOA), Ressourcenorientierte Architektur (ROA), Ereignisgesteuerte Architektur (Event-driven Architecture (EDA)) und Schichtenarchitektur unterschieden.

2.2.1 Serviceorientierte Architektur

Bei der SOA handelt es sich um eine service- oder dienstorientierte Architektur, welche anpassbar und flexibel ist. Ein Kernkonzept ist die Aufteilung der Software in Dienste (teilweise Microservices), welche oft aufgrund von Geschäftsprozessen strukturiert wird. Dabei entsteht meist eine vergleichsweise lose Struktur mit mehreren Ebenen. Die Flexibilität und Anpassungsfähigkeit wird dadurch erreicht, dass kleine Softwarekomponenten entstehen, welche losgelöst und unabhängig voneinander angepasst oder ausgetauscht werden können. Somit wird zusätzlich eine Entkopplung der Softwarekomponenten untereinander erreicht. [19, S. 62 ff.]

2.2.2 Ressourcenorientierte Architektur

Bei der ROA wird ein verteiltes System als Ansammlung von Ressourcen gesehen, welche individuell von einzelnen Softwarekomponenten genutzt werden können. Ressourcen stehen in unterschiedlichen Formen als Repräsentation einer Ressource bereit. Diese

Repräsentationen sind somit strikt von der eigentlichen Ressource getrennt. Das Prinzip der ROA findet Anwendung im Internet bzw. Internetdiensten. Als Interface wird oft Representational State Transfer (REST) eingesetzt, welche beispielsweise das Erstellen *(PUT)*, Löschen *(DELETE)* und Erhalten *(GET)* von Ressourcen bzw. ihren Repräsentationen, sowie das Ändern einer Ressource in einen anderen Zustand *(POST)* unterstützt. [19, S. 64 ff.]

2.2.3 Ereignisgesteuerte Architektur

Bei der EDA wird der Austausch von Daten und Nachrichten durch Ereignisse gesteuert. Hierbei werden verschiedene Softwarekomponenten i. d. R. durch einen gemeinsamen Ereignis-Bus verbunden. Ereignisorientierte Architekturen nutzen heutzutage meist das *Publisher*[9]*-Subscriber*[10] Prinzip [18, S. 54 ff.]. Bei diesem senden Publisher Nachrichten unter Topics[11] in das gesamte System. Subscriber können Topics abonnieren und erhalten dann eine Benachrichtigung[12], wenn neue Daten zur Verfügung stehen. In verteilten Systemen können die einzelnen Publisher und Subscriber auf unterschiedlichen Rechner ausgeführt werden. Die ereignisorientierte Architektur wird oft als Ergänzung zur SOA eingesetzt, indem Dienste durch Ereignisse ausgelöst werden. [19, S. 66 ff.]

2.2.4 Schichtenarchitektur

Das Grundprinzip der Schichtenarchitektur für verteilte Systeme entspricht der gezeigten Schichtenarchitektur (siehe Kapitel 2.1.4). Die Schichtenarchitektur wird bei verteilten Systemen jedoch nicht auf einer Recheneinheit, sondern auf mehrere Hardwarekomponenten verteilt. Bei einer Verteilung der Software in unterschiedliche Schichten und auf verschiedene Hardwarekomponenten kann somit zusätzlich eine hardwareseitige strenge hierarchische Trennung der Softwarekomponenten untereinander erfolgen. Bei einer *strengen* Trennung können einzelne Schichten nicht übersprungen werden. Bei einer *nicht-strengen* Trennung ist es möglich, dass Schichten übersprungen werden, beispielsweise um die Verarbeitung zu beschleunigen. Die Anzahl der Schichten kann je nach Anwendung und System variieren.

Eine typische Anordnung ist die Unterteilung eines Softwaresystems in die drei Schichten: Datenschicht, Verarbeitungsschicht und Präsentationsschicht. Abbildung 2.9 zeigt eine solche Anordnung einer Drei-Schichten-Architektur. Das Beispiel zeigt eine vereinfachte Struktur einer Internet-Suchmaschine. Das Datenlevel besteht aus einer Datenbank und ist im gezeigten Beispiel die tiefste Schicht *(back end)*. Die oberste Schicht, die

[9] dt. Herausgeber/ Veröffentlicher
[10] dt. Abonnenten
[11] dt. Thema
[12] engl. Notification

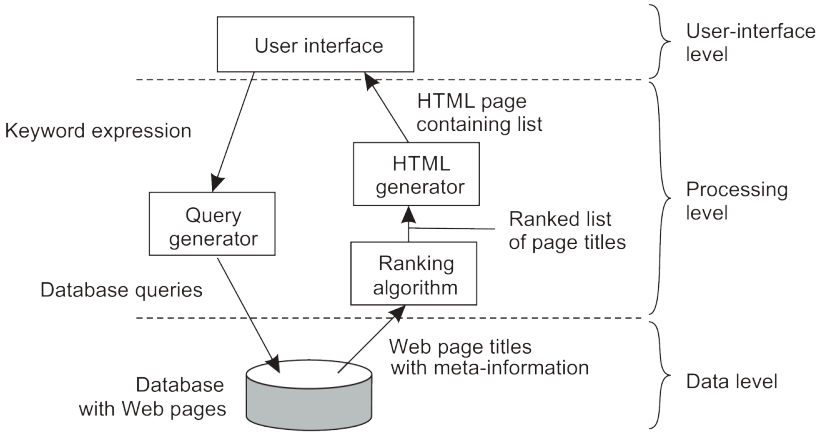


Abbildung 2.9: Beispiel einer Schichtenarchitektur [19, S. 61]

Präsentationsschicht *(front end)* wird zur Interaktion mit Nutzer*innen verwendet. Nur die mittlere Schicht, die Verarbeitungsschicht, hat Zugriff auf die beiden anderen Schichten und kann somit die Nutzereingaben verarbeiten und die entsprechenden Daten aus der Datenschicht anfordern. [19, S. 57 ff.]

2.2.5 Heterogene verteilte Systeme

Neben der gezeigten Architektur oder Struktur von verteilten Systemen, spielt die genutzte Hardware, beispielsweise die Kombination von verschiedenen CPU-Typen oder Schaltungsvarianten, eine Rolle.

Als *Heterogeneous Computing*[13] oder Heterogeneous Multiprocessing (HMP) wird das Verteilen von Software auf heterogene Kerne bezeichnet. Das *Multiprocessing* steht für das koordinierte Verteilen eines Programms auf mehrere Prozessoren. Hierbei kann z. B. die Verteilung auf zwei unterschiedliche CPU-Typen erfolgen.

Möglich ist zudem eine Kombination aus CPU mit GPU, FPGA oder ASIC, welche heute lokal in einem Modul umgesetzt werden kann, wie beispielsweise beim NVIDIA Jetson TX2 [20] als CPU/ GPU Hybrid oder die Zynq-Reihe von Xilinx [21] als CPU/ FPGA Hybrid. Die Kombination aus FPGA und CPU wird auch als System on a Programmable Chip (SoPC) bezeichnet. Hierbei existieren sowohl Lösungen aus synthetisierten CPU-Kernen

[13] dt. Heterogenes verarbeiten

(SoftCores) innerhalb des FPGAs, als auch HardCores Varianten mit nicht veränderbarer CPU und Kombinationen aus beiden Varianten.

Insbesondere bei heterogenen verteilten Systemen kann die Nutzung von Kleinstrechnern sinnvoll sein, da die vergleichsweise begrenzte Rechenleistung dieser Kleinstrechner durch die Kombination mit einem leistungsstarken Rechner, beispielsweise einem Rechner aus einer anderen Schaltungsvariante, ausgeglichen werden kann. Ein Kleinstrechner, wie z. B. ein Single-Board Computer (SBC)[14] ist hierbei eine Computerfamilie. Verschiedene Computerfamilien können aufgrund der Anordnung und Zuordnung von Komponenten einer Rechnerarchitektur zueinander differenziert werden. Bei einem SBC beispielsweise werden alle Komponenten auf einer einzigen Leiterplatte angeordnet. Hierzu gehören z. B. Anschlüsse für Monitore, Netzwerk, Universal Serial Bus (USB) oder auch Flash-Speicher. Besonders beliebt ist u. a. der SBC Raspberry Pi [O7]. SBC, wie der Raspberry Pi, sind ggf. für die Nutzung in mobilen Robotern relevant, da diese energieeffizient sind und viele Anschlussmöglichkeiten für Sensoren und Aktuatoren bieten.

2.3 Grundlagen des Cloud Computing

Cloud Computing[15] hat im letzten Jahrzehnt die Entwicklung von technischen Systemen maßgeblich beeinflusst. Hinter Begriffen wie Industrie 4.0, Internet der Dinge (Internet of Things (IoT)) oder CPS steht die Auslagerung einzelner Softwarekomponenten oder ganzen Systemen in die Cloud. Cloud Computing wird in *Cloud Computing - Web basierte dynamische IT-Services* von Baun, Kunze, Nimis und Tai [22, S. 4] wie folgt definiert:

> „Unter Ausnutzung virtualisierter Rechen- und Speicherressourcen und moderner Web-Technologien stellt Cloud Computing skalierbare, netzwerkzentrierte, abstrahierte IT-Infrastrukturen, Plattformen und Anwendungen als on-demand Dienste zur Verfügung. Die Abrechnung dieser Dienste erfolgt nutzungsabhängig."

Die Einbindung der Cloud in ein software-intensives System führt zu einem verteilten System oder erweitert dieses, da die Software durch die Einbindung der Cloud auf mindestens einen zweiten Rechner verteilt wird. Im Gegensatz zur lokalen Verteilung ist beim Cloud Computing die Zielhardware meistens nicht bekannt, da diese virtualisiert eingebunden wird. Diese Virtualisierung der Hardware macht es möglich, die eigentliche Hardware zu

[14] dt. Einplatinenrechner
[15] dt. Rechnerwolke/ Datenwolke

modifizieren oder zu ersetzen. Somit ist eine Skalierung von Cloud Diensten jederzeit möglich. Die Leistungsfähigkeit bzw. die Hardwareressourcen der Cloud können an die aktuellen Anforderungen angepasst werden.

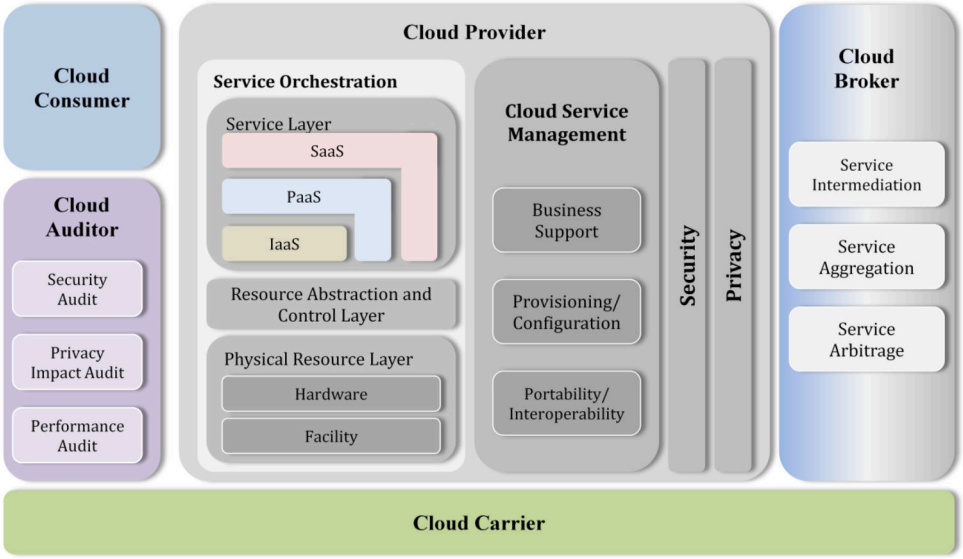


Abbildung 2.10: Konzeptionelles Referenzmodel des Cloud Computing [23, S. 3]

Die Architektur von Cloud Systemen ist service- bzw. dienstorientiert (siehe SOA in Kapitel 2.2.1). Abbildung 2.10 zeigt Komponenten der Architektur, sowie die einzelnen Akteure im Cloud Computing. Der Cloud Consumer[16] ist eine Person oder Organisation, welche eine Geschäftsbeziehung mit dem Cloud Provider[17] eingeht und dessen Dienste nutzt. Der Cloud Provider ist dafür zuständig, die Dienste verfügbar zu machen. Der Cloud Auditor[18] kann eine unabhängige Bewertung der Cloud Dienste in Bezug auf Sicherheit, Datenschutz und Performance durchführen. Der Cloud Broker[19] verwaltet die Nutzung, Performance und die Bereitstellung der Cloud Dienste. Der Cloud Carrier[20] stellt die Verbindung und den Transport der Cloud Dienste vom Cloud Provider zum Cloud Consumer zur Verfügung. [23, S. 3 ff.]

[16] dt. Konsument
[17] dt. Betreiber
[18] dt. Prüfer
[19] dt. Makler
[20] dt. Übertrager/ Beförderer

Anwendungen des Cloud Computing können in die drei Kategorien Software as a Service (SaaS), Platform as a Service (PaaS) und Infrastructure as a Service (IaaS) unterteilt werden (siehe Abbildung 2.11). Diese drei Kategorien stellen einen unterschiedlichen Abstraktionsgrad dar, auf die Nutzer*innen je nach Anforderungen aufsetzen können. Diese Kategorien beschreiben Baun, Kunze, Nimis und Tai [22] wie folgt:

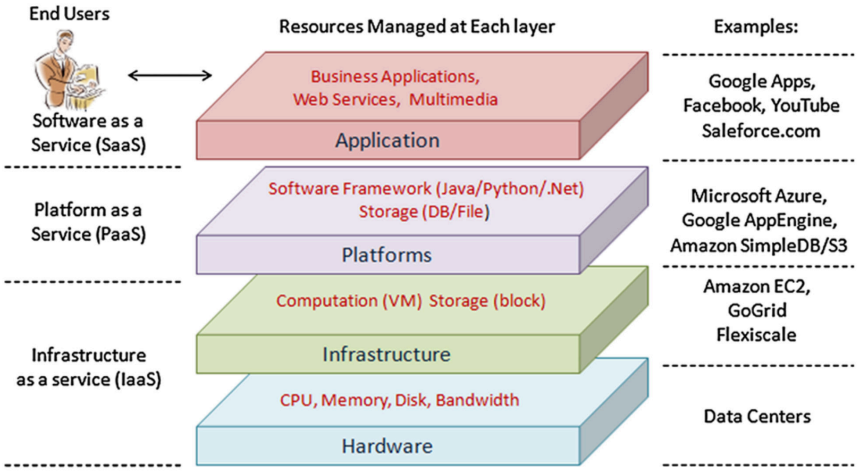


Abbildung 2.11: Drei Kategorien des Cloud Computing [24]

Bei der Infrastrukturschicht (IaaS) setzen Nutzer*innen direkt auf die zur Verfügung gestellten Hardware auf. Nutzer*innen sind selbst für ihre Recheninstanz verantwortlich und können dort beispielsweise selbst eine Virtuelle Maschine (VM) aufsetzen und verwalten. Ein Vorteil gegenüber lokaler Hardware ist die Skalierbarkeit, da die Hardware in der Cloud ständig an die Anforderungen angepasst werden kann. [22, S. 29 ff.]

Die PaaS-Schicht setzt auf eine bereits verfügbare IaaS-Schicht auf. Nutzer*innen haben keinen direkten Zugriff auf die Recheninstanzen, diese wird vom Cloud Provider verwaltet. PaaS wird häufig von Entwickler*innen eingesetzt, um Software in bestimmten Entwicklungsumgebungen (Programming Environment (PE)) zu entwickeln oder Software in unterschiedlichen Umgebungen (Execution Environment (EE)) auszuführen und zu testen. [22, S. 33 ff.]

Die Anwendungsschicht (SaaS) adressiert den Endkunden. Vorhandene Dienste oder Applikation werden benutzt, ohne sich um die darunter liegende Plattform, Infrastruktur oder gar Hardware kümmern zu müssen. [22, S. 35 ff.]

Cloud Systeme bieten im Gegensatz zu lokalen Systemen (ohne Internetanbindung) den Komfort, aber laut Griffor [25] auch eine erhöhte Gefahr für Verletzungen der Sicherheit (*Security*) durch die Erreichbarkeit von außen. Kritische Systeme können zur Umgehung dieses Sicherheitsproblems anstelle einer öffentlichen *(Public)* Cloud, eine private *(private)* Cloud einsetzen. Bei privaten Cloud Systemen ist die Hardware der Cloud Server entweder innerhalb des privaten Netzes oder virtuell vom Internet getrennt (z. B. durch ein virtuelles privates Netz (Virtual Private Network (VPN)). Hybride Lösungen zur Auskopplung von Teilen der Cloud Diensten ist außerdem vorgesehen. [23, S. 10 ff.]

Nichtsdestotrotz müssen Entwickler*innen von Cloud Diensten die Anforderung ihrer Dienste an die Sicherheit besonders beachten. Laut [26] ist die Einhaltung der Sicherheit (Security) die größte Herausforderung bei der Nutzung von Cloud Computing.

2.4 Operator-Controller-Modul

Im Rahmen des Sonderforschungsbereich (SFB) 614 *„Selbstoptimierende Systeme des Maschinenbaus“* [O8] wurde die OCM Architektur entwickelt. Diese basiert auf der bereits erläuterten Schichtenarchitektur. Unter anderem in „Selbstoptimierende Systeme des Maschinenbaus: Definitionen, Anwendungen, Konzepte“ [27] werden das Konzept und die möglichen Anwendungsdomänen des OCM Ansatzes vorgestellt.

Die unterste Schicht, die ***Controller***-Ebene (siehe Abbildung 2.12) bildet die Verbindung zur Hardware im sogenannten motorischen Kreis[21]. Die *Controller* verarbeiten Sensor-Signale und steuern Aktuatoren typischerweise mittels regelungstechnischer Methoden an. Diese motorische Informationsverarbeitung arbeitet quasi-kontinuierlich, d. h. die Verarbeitung der Messwerte erfolgt kontinuierlich und wird unter harten Echtzeitbedingungen weitergegeben.

Die mittlere Schicht des OCM enthält den ***reflektorischen Operator***[22]. Diese ist hierarchisch über der *Controller*-Ebene organisiert und mit dieser im reflektorischen Kreis[23] verbunden. Der *reflektorische Operator* überwacht und steuert die *Controller*, indem Parameter oder ggf. die Struktur der *Controller* angepasst werden. Weiterhin kann der *reflektorische Operator* zwischen verschiedenen Konfigurationen des *Controller* wechseln. Im Gegensatz zum *Controller* arbeitet der *reflektorische Operator* ereignis-gesteuert, indem beispielsweise auf den Eingang neuer Messwerte reagiert wird. Daneben werden zudem quasi-kontinuierliche Funktionen wie Watchdogs[24] implementiert. Die Verbindung zur unteren *Controller*-Ebene

[21] engl. Motor loop
[22] engl. Reflective operator
[23] engl. Reflective loop
[24] Ein Watchdog (dt. Wachhund) wird zur Kontrolle eines digitalen Systems eingesetzt. Watchdogs überwachen kontinuierlich Daten und agieren, sobald eine Auffälligkeit besteht.

ist die Handlungsebene und bedarf der Einhaltung von harter Echtzeit. Der *reflektorische Operator* bildet weiterhin das Interface zum nicht echtzeitfähigen kognitiven Anteil des OCM. Der *reflektorische Operator* verarbeitet die Daten der kognitiven Ebene und bringt sie so in die unterste *Controller*-Ebene ein.

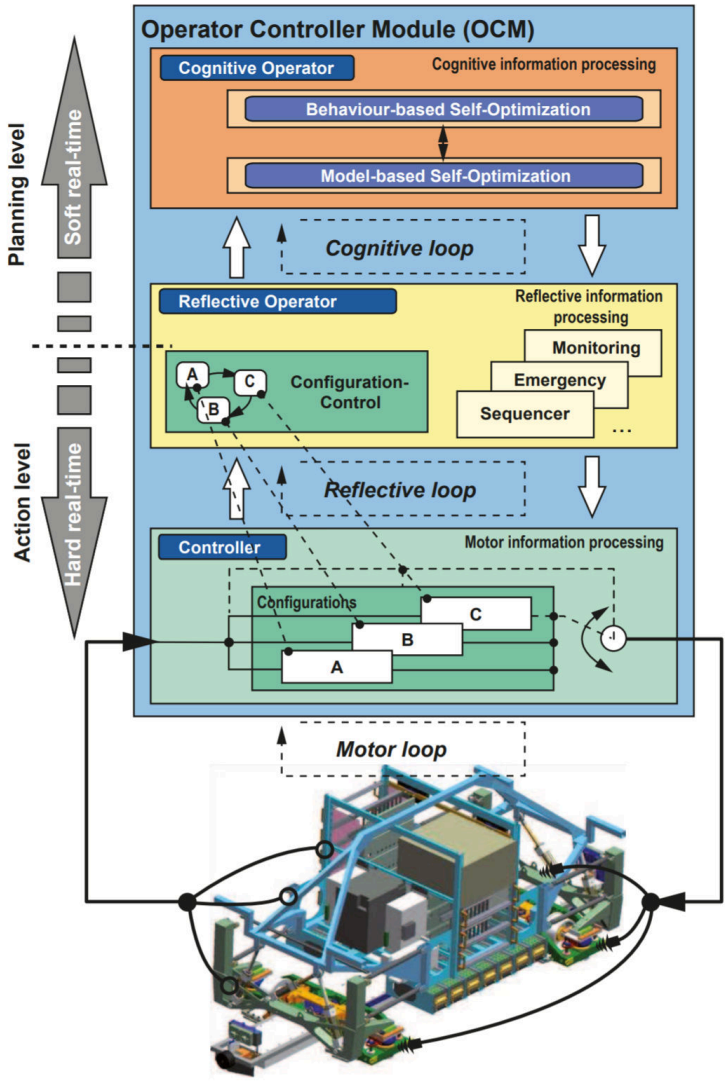


Abbildung 2.12: OCM Architektur [28]

Die oberste Schicht, die Ebene des ***kongitiven Operators***[25] bildet, zusammen mit dem *reflektorischen Operator* den kognitiven Kreis[26] des OCM. Der *kognitive Operator* wird zur Planung und Optimierung des Systems eingesetzt und ist hierarchisch über dem *reflektorischen Operator* angesiedelt und bildet die Planungsebene. Der *kognitive Operator* kann asynchron zur Realzeit arbeiten, unterliegt somit keiner harten Echtzeit. Da etwaige Planungen und Optimierungen jedoch trotzdem zeitnah dem System zur Verfügung stehen sollten, wird hier von weicher Echtzeit gesprochen.

Die Aufteilung eines technischen Systems in Handlungs- und Planungsebene findet man in ähnlicher Form bereits als Ropohls Handlungssystem von 1978 (erste Auflage) in *Allgemeine Technologie: Systemtheorie der Technik* [29, S. 102].

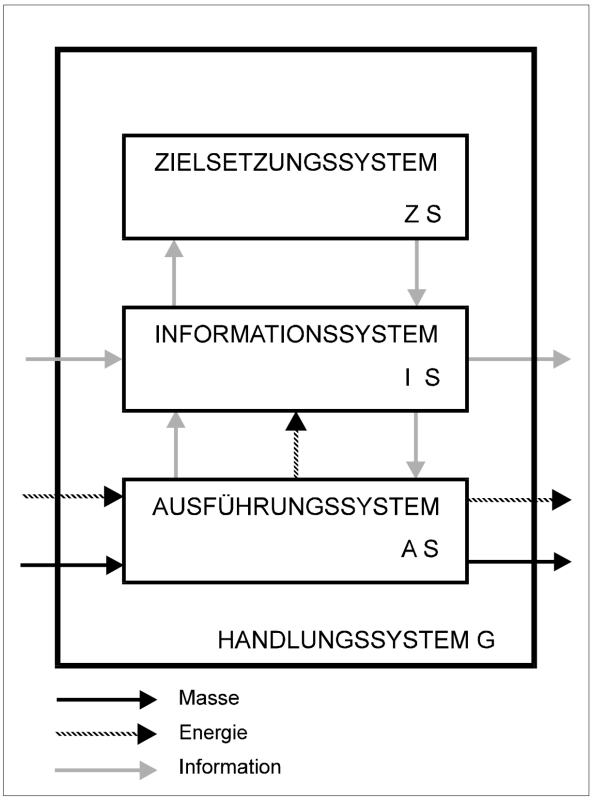


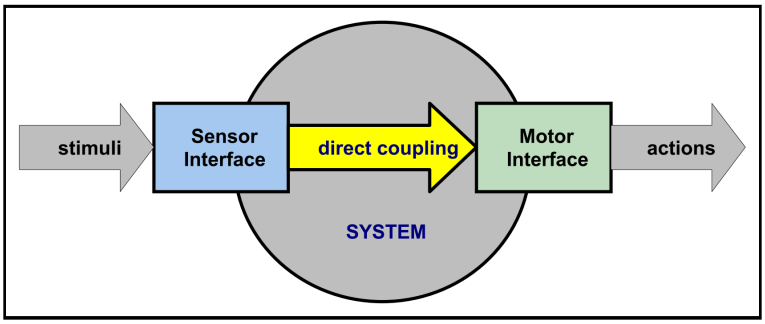
Abbildung 2.13: Struktur eines Handlungssystems nach Ropohl [29, S. 102]

[25] engl. Cognitive operator
[26] engl. Cognitive loop

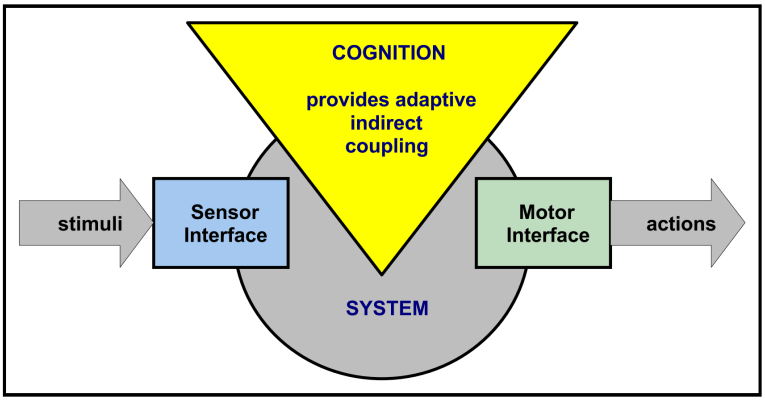
Das Handlungssystem (siehe Abbildung 2.13) strukturiert Ropohl in ein Ausführungs-, Informations- und Zielsetzungssystem. Das Ausführungssystem ist als einziges Teilsystem mit einer Masse, also der physischen Ebene des technischen Systems verbunden und wird von außen mit Energie versorgt, bzw. gibt diese zurück nach außen. Das Informationssystem erhält Energie und Informationen aus der unterliegenden Ebene und verarbeitet diese. Hierbei spricht Ropohl u. a. von Informationsverarbeitung und Speicherung. Das Zielsetzungssystem wird als Teilsystem definiert, welches aufgrund von Informationen des Informationssystems, interne Ziele des Systems vorgibt. Diese dreistufige Interaktion von Teilsystemen entspricht weitestgehend sowohl dem OCM, als auch der kognitiven Verarbeitung von Strube [30, S. 6].

Strube zeigt in „Modelling Motivation and Action Control in Cognitive Systems“ [30, S. 6] zunächst den Unterschied von Systemen ohne kognitive Verarbeitung (siehe Abbildung 2.14a) gegenüber Systemen mit kognitiver Verarbeitung (siehe Abbildung 2.14b). Während Sensordaten und Aktuator-Stellwerte in einem System ohne kognitive Verarbeitung unmittelbar, also direkt miteinander gekoppelt sind, führt die Kognition zu einer indirekten Kopplung. Insbesondere bei mechatronischen Systemen, wie beispielsweise Robotern, ist diese Kognition üblich[27]. Durch die Kognition werden die nächsten Schritte des Systems geplant und ggf. optimiert. Die direkte Verarbeitung von Sensorwerte in Aktuator-Stellwerte mittels Regelkreisen kann jedoch für das Abarbeiten von bereits festgelegten Planungen erfolgen[28]. Eine direkte Kopplung von Sensor- und Aktuatordaten kann jedoch zu einer direkten Fehlerübertragung von fehlerhaften Sensordaten auf das Handeln des Roboters führen. Durch die Kognition werden Informationen eines einzelnen Sensors überprüft und zudem in Zusammenhang mit anderen Sensoren gesetzt. Durch die sogenannte Datenfusion werden Sensordaten validiert und kombiniert [31], sodass das Handeln eines Roboters weniger fehleranfällig ist und komplexere Handlungsoptionen und Umgebungen abgebildet werden können. Die Kognition ist ebenfalls zur Koordination von Multi-Roboter Kooperationen notwendig, da diese für die Verteilung von Aufgaben, beispielsweise innerhalb eines Schwarms unabdingbar ist.

[27] vgl. hybrides deliberatives/ reaktives Paradigma der Robotik im nachfolgenden Kapitel 3.1.1
[28] vgl. reaktives Paradigma der Robotik im nachfolgenden Kapitel 3.1.1



(a)



(b)

Abbildung 2.14: System ohne (a) und mit (b) kognitiver Verarbeitung [30, S. 6]

Strube verdeutlicht zudem die kognitive Regulierung in einem dreistufigen Modell [30, S. 9]. Analog zum OCM und zu Ropohl's Handungssystem wird die unterste Ebene (siehe Abbildung 2.15) zur kontinuierlichen Regulierung bzw. Steuerung des Systems genutzt. Strube bezeichnet dieses auch als Reflexe. Die Reiz-Reaktions-Assoziationen bzw. Konditionierung erfolgt in der mittleren Ebene, welche als assoziative Regulierung bezeichnet wird. Das Zielmanagement bzw. die Planung und Handlungssteuerung erfolgen auf der hierarchisch obersten Ebene.

Abbildung 2.15: Dreischichtenmodell einer Verhaltenskontrolle nach Strube [30, S. 9]

Das OCM wird seit 1998 besonders an Projekten der Universität Paderborn angewendet und schließlich im Rahmen des SFB 614 *„Selbstoptimierende System des Maschinenbaus“* [O8] verankert. Naumann veröffentlichte 2000 in seiner Dissertation *Modellierung und Verarbeitung vernetzter intelligenter mechatronischer Systeme* die bereits vorgestellten Grundprinzipen des OCM [32, S. 27 ff.]. Oberschelp, Hestermeyer und Giese zeigen in [33] die Selbst-Optimierung mittels OCM. Hestermeyer wendet das OCM in seiner Dissertation 2006 für die *Strukturierte Entwicklung der Informationsverarbeitung für die aktive Federung eines Schienenfahrzeugs* [34] am Beispiel des Schienenfahrzeugs RailCab an. In seiner 2008 erschienenen Dissertation *Strukturierter Entwurf selbstoptimierender mechatronischer Systeme* [35] beschäftigt sich Oberschelp mit der Selbstoptimierung des OCM. Münch zeigt 2012 den verteilten Ansatz des OCM in seiner Dissertation *Selbstoptimierung verteilter mechatronischer Systeme auf Basis paretooptimaler Systemkonfigurationen* [36].

Dumitrescu führt die Prinzipien des OCM in seiner 2010 erschienenen Dissertation *Entwicklungssystematik zur Integration kognitiver Funktionen in fortgeschrittene mechatronische Systeme* [37] aus, indem er unter anderem eine Wirkstruktur des OCM erarbeitet (siehe Abbildung 2.16). Die Wirkstruktur verdeutlicht die Aufgaben und Teilfunktionen der drei Ebenen des Systems anhand von Systemelementen und dessen Funktion(en). So wird beispielsweise das Systemelement *Aktuatorik* des *Controllers* dazu benutzt, die Funktion *Aktion auszuführen*, indem *Stellgrößen eingestellt* werden. Eine *Wissensbasis* hingegen wird im *kognitiven Operator* aufgebaut und zur Speicherung von Informationen verwendet. Weithin wird der Informationsfluss dargestellt und zeigt somit die Koordination und Kooperation der Systemelemente mit- und untereinander. Die

Stellgröße des Elements *Aktuatorik* ist beispielsweise abhängig von der aktuellen Konfiguration, welche vom *Ausgangsschalter* des *reflektorischen Operators* umgeschaltet wird.

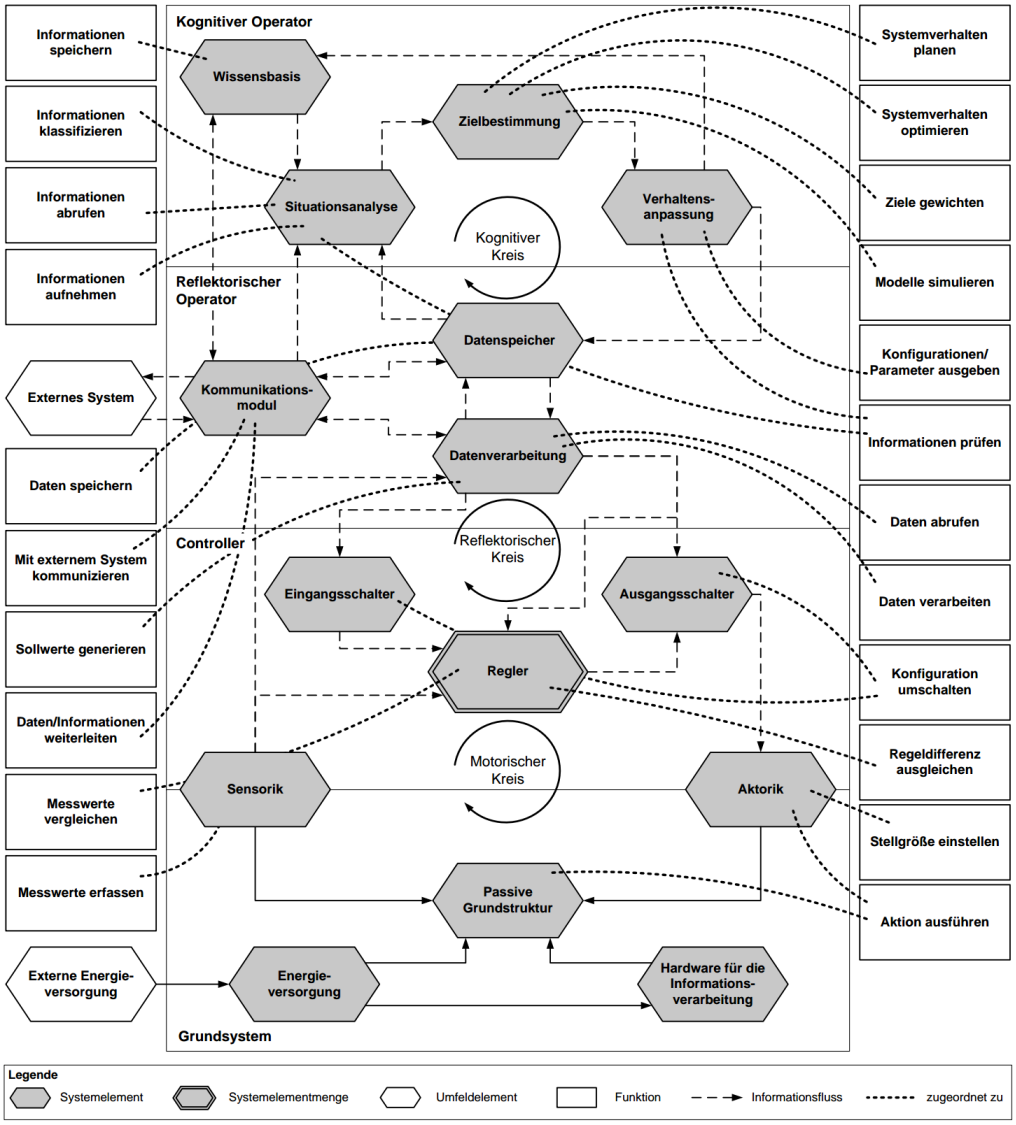


Abbildung 2.16: OCM Wirkstruktur [37, S. 114]

2.5 Stand der Technik von Architekturen für mobile Roboter

Architekturen von Robotern basieren oft auf ROS [4], welches als De-facto-Standard für die Entwicklung von Robotern angesehen wird [38]. ROS unterstützt Entwickler*innen durch die Verfügbarkeit von verschiedensten Software-Paketen (ROS Packages), welche sowohl Treiber für Hardware, als auch Software Algorithmen oder Tools enthalten können. ROS Dienste werden als ROS Nodes[29] bezeichnet und nach dem Publisher-Subscriber Prinzip miteinander gekoppelt. Die Kommunikation ist via Transport Control Protocol (TCP) und User Datagram Protocol (UDP) implementiert, weshalb eine Umsetzung eines eigenen Systems in verteilter Architektur grundsätzlich möglich ist. ROS ist für verschiedene Hardware-Architekturen und Betriebssysteme verfügbar. ROS hilft zwar besonders Einsteigern, enorm schnell Erfolge bei der Entwicklung zu erzielen, nimmt jedoch durch das einfache Zusammenführen und Koppeln von Nodes den Fokus von der eigentlichen Architektur. Oftmals wird ROS auf ungeeigneter Hardware oder Standard-Betriebssystemen wie Ubuntu (Linux ohne Echtzeit-Kernel) ausgeführt, sodass beispielsweise die Einhaltung der harten Echtzeitfähigkeit kaum möglich ist [39]. Laut [39] wird die Echtzeitfähigkeit mit dem Nachfolger ROS 2 [O9] verbessert.

In [40] beschreiben Schöpping, Korthals, Hesse und Rückert eine generische modulare Architektur, welche bei der Entwicklung der mobilen Roboter AMiRo und BeBot entstanden ist. Der modulare Ansatz setzt auf eine flexible Topologie von Teilmodulen, welche durch verschiedene Kommunikationskanäle, wie Controler Area Network (CAN) oder Ethernet gekoppelt werden. Die Veröffentlichung beschreibt weiterhin notwendige Module, Konfigurationen, Protokolle und Tools zur Umsetzung einer Architektur.

In [41] stellen Ahmad und Babar eine systematische Studie zu Software Architekturen für Systeme in der Robotik vor. Hierbei untersuchen die Autoren unter anderem die Art und Anzahl von Veröffentlichungen im Bereich Architekturen für Roboter und analysieren insgesamt 56 Veröffentlichungen (S1 - S56 in Abbildung 2.17). Der Journalbeitrag gruppiert außerdem die analysierten Veröffentlichungen, beispielsweise in verwendete Notationen oder Anwendungsgebiete. In Abbildung 2.17 sind verschiedene Forschungsschwerpunkte und Architekturen für Roboter gruppiert und in zeitlicher Abfolge dargestellt. In den letzten Jahren haben Ahmad und Babar besonders die Bereiche cloudbasierte Robotik und modellbasierte Entwicklung herausgestellt. Der cloudbasierten Robotik wird eine weite Verbreitung und zukunftsorientierte Lösung für Roboter attestiert, indem beispielsweise die Nutzung von leistungsstarken Berechnungen in der Cloud eingesetzt würden. Die modellbasierte Entwicklung in der Robotik wird besonders zur Definition und Einhaltung von Anforderungen eingesetzt, jedoch auch zur Generierung von ausführbarem Quelltext aus den Modellen verwendet. Dieses führe zur Minimierung der Programmier-Komplexität und unterstütze die Weiternutzung von Software mit unterschiedlichen Plattformen.

[29] dt. Knoten

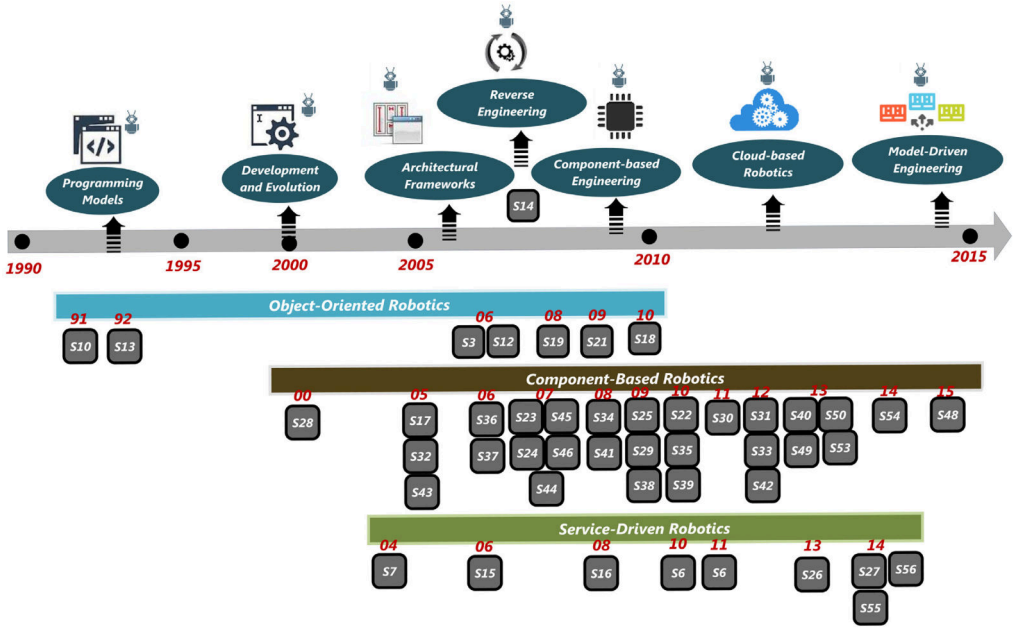


Abbildung 2.17: Überblick über Forschungsverläufe und Typen von Architekturen für Roboter [41, S. 31]

Corke, Sikka, Roberts und Duff präsentieren mit DDX (Dynamic Data eXchange) eine verteilte Architektur für Roboter [42]. Die vorgestellte Architektur unterteilt das System in Clients und Computer. Zum Austausch von Daten zwischen Clients werden in DDX sogenannte *Stores* eingesetzt, welche für einen synchronisierten Zugriff auf gemeinsame Daten sorgen. Zudem kommt ein sogenannter *Catalog* zum Einsatz, welcher ein globales Repositorium von Informationen darstellt und zum Datenaustausch von mehreren Computern als verteiltes System verwendet wird. Die Veröffentlichung enthält keine weiteren Angaben zu verwendeten Robotern oder deren Umsetzung.

In [43] wird eine weitere verteilte Architektur vorgestellt. Hier wird der Roboter in Client und Server unterteilt. Tikanmäki und Röning zeigen in ihrem Ansatz, dass ihr System modular und flexibel durch die Einbindung weiterer Komponenten erweitert werden kann. Hierzu wird lediglich eine Kommunikationsschnittstelle und dessen Protokoll adaptiert. Der gezeigte Ansatz geht auf Anforderungen, wie die Echtzeitfähigkeit nicht ein.

Eine weitere Architektur wird von Kaupp, Brooks, Upcroft und Makarenko in [44] vorgestellt. Der Fokus der vorgestellten Architektur liegt beim gezeigten Orca Framework auf der Interaktion zwischen Mensch und Maschine unter der Verwendung eines verteilten Systems. Das Framework zeigt das Setup aus Basisstation, Bodenfahrzeug, Bodenstation und zwei menschlichen Operatoren. Die Operatoren sind mittels Tablet PCs in das verteilte System eingebunden und können somit mit dem Roboter interagieren. Das gezeigte Framework setzt auf einer hohen Abstraktionsebene auf und zeigt nicht die darunter liegenden Ebenen, wie Hardware-Abstraktionen.

Yang, Mao, Yang und Liu beschreiben in [45] eine High-level Software Architektur einer Schichtenarchitektur. Die beiden Schichten (siehe Abbildung 2.18) bestehen aus einer reagierenden Ebene (*Reaktive Layer*), welche ähnlich zum OCM in direkter Verbindung zur Umgebung steht, Sensordaten aufnimmt und mit der Umwelt interagiert. Die adaptive Ebene (*Adaptive Layer*) beinhaltet sowohl die Modellierung, als auch die Planung der Aktionen. Des Weiteren beinhaltet die adaptive Ebene die Überwachung des Systems. Ausgeführt werden beide Ebenen lokal auf einem Notebook, welches direkt auf dem mobilen Roboter installiert wird. Angaben bezüglich der Entkopplung einzelner Ebenen werden nicht gemacht.

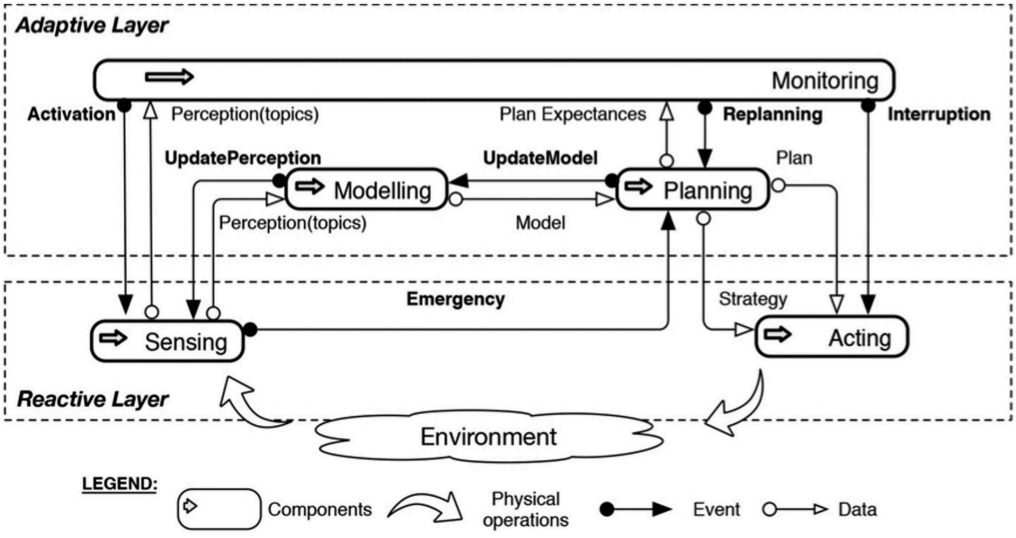


Abbildung 2.18: High-level Framework einer hybriden Software Architektur [45]

Die SERA Architektur [46], vorgestellt von Garcia, Menghi, Pelliccione, Berger und Wohlrab, basiert auf den Architektur-Richtlinien von Kramer und Magee [47]. Die Architektur ist ein verteiltes System aus einer Basistation (siehe *Central Station* in Abbildung 2.19) und dem eigentlichen Roboter. Der Roboter ist analog zum OCM in drei Schichten unterteilt. Mitsamt der Basisstation sind somit vier Schichten vorgesehen. Die unterste Schicht, die *Component Control* Ebene wird für einen Großteil der Funktionen, wie Simultaneous Localization and Mapping (SLAM) Verfahren und auf der Interaktion mit Sensoren und anderer Hardware eingesetzt. Die mittlere Ebene (*Change Management Layer*) beinhaltet einen Informations- und Adaptions-Manager, welche zusammen mit einer ausführenden Einheit (*Plan executor*) die unterste Ebene beeinflussen. Die höchste lokale Ebene, die *Mission Management* Ebene wird unter anderem zur Kommunikation mit der Basisstation und mit anderen Robotern benötigt. Sie beinhaltet einen lokalen Missionsplaner, während die Basisstation zusätzlich für die Planung von globalen Missionen zuständig ist. Der Fokus der SERA Architektur liegt somit stark auf der Kooperation zwischen Roboter(n) und Menschen. Die Aufteilung der Funktionen auf die drei Schichten unterscheidet sich in der Aufteilung vom OCM-Ansatz.

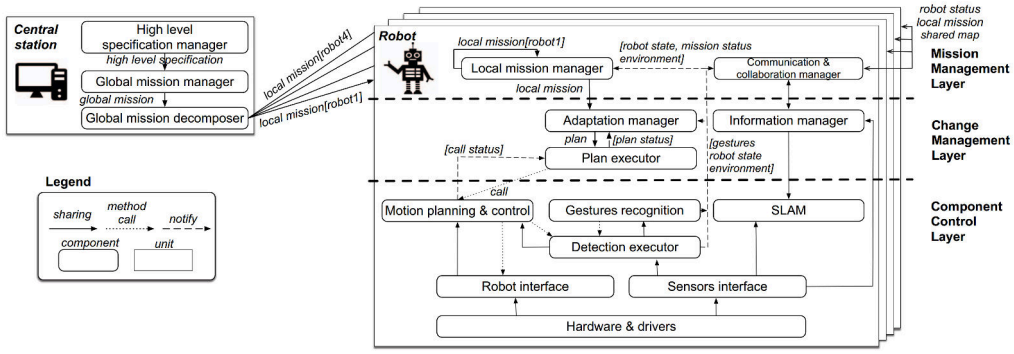


Abbildung 2.19: SERA Architektur [46]

Benjamin, Lyons und Lonsdale stellen in [48] mit ADAPT (Adaptive Dynamics and Adaptive Perception for Thought) eine Architektur für die Robotik mit Fokus auf die Kognition vor. Jene Architektur nutzt Sensor-Aktuator Schemata, um verschiedene Ziele gleichzeitig verfolgen und bearbeiten zu können. Eine Aktion entsteht aus dem Eingang bzw. dem Erhalt neuer Information, aufgrund dessen ein Sensor-Aktuator Schemata aktiviert und ausgeführt wird. Ein solches Schemata kann weitere Schemata auslösen und somit eine Kette von Ereignissen anstoßen. Da mehrere Schemata verschiedene Aktionen der Aktuatoren auslösen könnten, wird ein übergeordneter Operator implementiert, welcher kontinuierlich die einzelnen Schemata steuert, indem er sie hierarchisch sortiert

und somit ihre Ausführung beeinflusst. In der gezeigten Veröffentlichung erläutern die Autoren erste Tests mittels eines mobilen Roboters.

In [49] wird eine kognitive Architektur für humanoide Roboter beschrieben. Burghart, Mikut, Stiefelhagen, Asfour, Holzapfel, Steinhaus und Dillmann wenden hier ebenfalls eine Schichtarchitektur aus drei Ebenen an. Mittig in der Architektur und alle drei Schichten überschreitend, steht die Komponente *Active models* (siehe Abbildung 2.20). Diese ist die zentrale Einheit der Architektur und führt verschiedene Modelle aus der *Global Knowledge Database* aus. Diese Modelle werden auf den unterschiedlichen Ebenen ausgeführt (*Low-Level*), koordiniert (*Mid-Level*) und geplant (*High-Level*). Der *Execution supervisor* erhält vom Task Planer eine Task-Abfolge, welche dieser aufgrund von verfügbaren Ressourcen priorisiert. Die Ausführung der Tasks erfolgt kontinuierlich auf der untersten Ebene der Architektur, sodass zeitkritische Tasks priorisiert ausgeführt werden können.

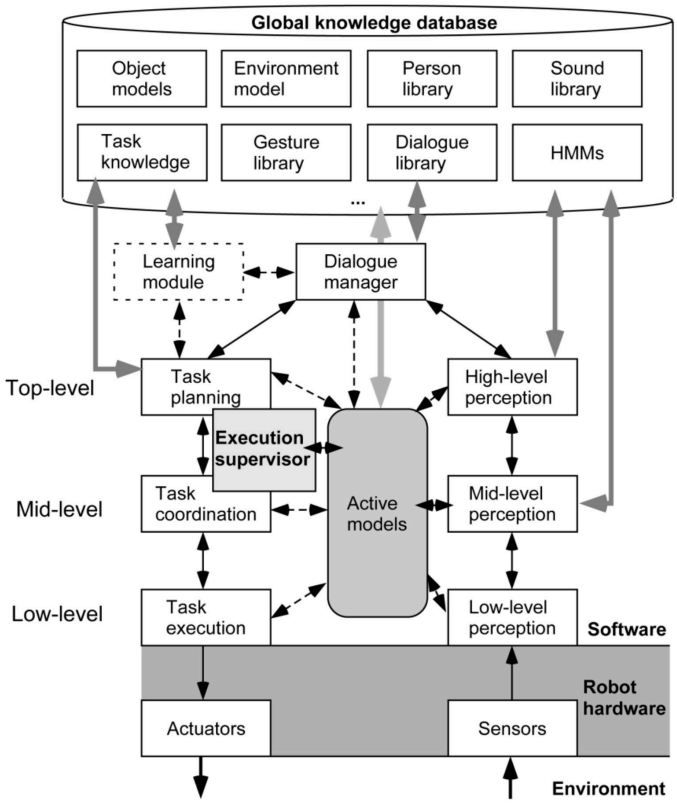


Abbildung 2.20: Kognitive Architektur für einen humanoiden Roboter [49]

Sanchez-Lopez, Fernández, Bavle, Sampedro, Molina, Pestana und Campoy stellen in [50] eine Architektur aus einem Mehrschichtenmodell für Flugroboter vor (siehe Abbildung 2.21). Die Architektur umfasst *N* Roboter Agenten, welche von einem menschlichen Operator gesteuert werden. Die Architektur der jeweiligen Roboter umfasst je fünf Ebenen. Die „soziale“ Ebene dient zur Kommunikation der Roboter mit anderen Robotern und dem Operator. Die reflektorische Ebene überwacht die nachfolgenden Ebenen und reagiert auf Probleme. Die Beratungsebene[30] erarbeitet globale Lösungen zur Bewältigung von komplexen Aufgaben. Ausgeführt werden diese Lösungen durch die ausführende Ebene. Die reaktive Ebene[31] interagiert mit Aktuatoren und Sensoren.

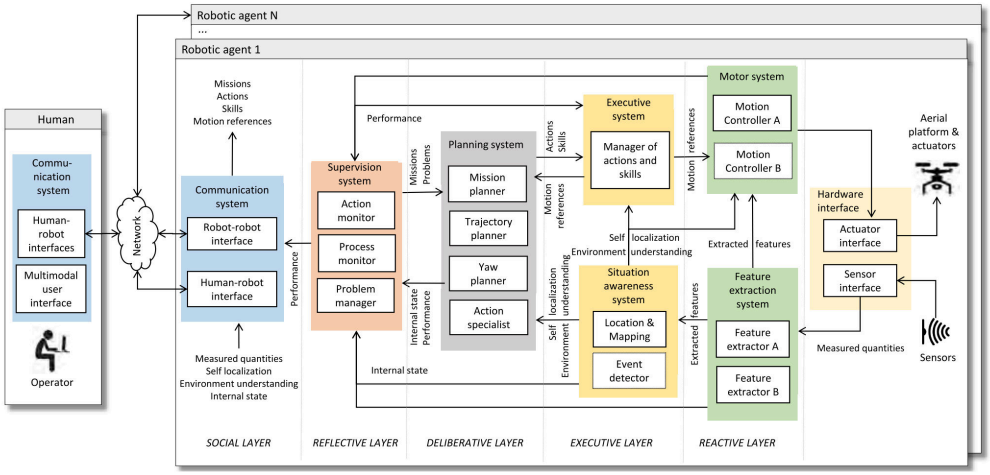


Abbildung 2.21: Mehrschichtenarchitektur für Flugroboter [50]

Feitosa und Nakagawa untersuchen in [51] die sieben nachfolgend beschriebenen Architekturen für mobile Roboter auf der Suche nach geeigneten Referenz-Architekturen.

In [52] beschreibt Albus die 4D/RCS Architektur für unbemannte Fahrzeug-Systeme. Die Architektur besteht aus mehreren Schichten, welche unterschiedliche zeitliche Bedingungen unterliegen. Die unterste Schicht, die Servo-Ebene besteht aus simplen Notes, dessen Planung innerhalb von 50 ms bearbeitet werden. Anweisungen an die Aktuatoren erfolgen alle 5 ms. Die übergeordnete Ebene *Primitive Level* stellt für die Planungen 500 ms bereit, wenngleich der Output alle 50 ms aktualisiert wird. Auf *Subsystem Level* stehen 5 s zur Verfügung. Auf *Section Level* stehen etwaigen Planungen bis zu 10 min zur Verfügung.

[30] engl. Deliberative layer
[31] engl. Reactive layer

Rowe und Wagner beschreiben in [53] die Referenzarchitektur JAUS, ebenfalls für unbemannte Roboter. JAUS wurde federführend vom Verteidigungsministerium der Vereinigten Staaten entwickelt. Die Veröffentlichungen zeigt neben der verwendeten Softwarearchitektur, ein striktes Kommunikations-Interface, welches die Interoperabilität zwischen Roboter Plattformen des Militärs ermöglicht.

In [54] beschreiben Ortiz, Alonso, Pastor, Alvarez und Iborr mit ACROSET eine Architektur, welche für die Teleoperation eines mobilen Roboters eingesetzt wird. Das ACROSET Sub-System besteht aus einer Hardware-Abstraktion *CCAS (Coordinate, Control and Abstraction Subsystem)*, welches die Funktion von der physischen Hardware des Roboters entkoppelt. Das *UIS (User Interfaces Subsystem)* bildet das Interface zwischen Nutzer*innen und dem *CCAS*. Neben diesen beiden Sub-Systemen ist ein sogenanntes *IS (Intelligent Systems)* implementiert, welches Nutzer*innen durch intelligente Algorithmen unterstützen soll. Das *SMCS (Safety Management and Configuration Subsystem)* wird zur Konfiguration und zur Überwachung des *CCAS* verwendet.

Peters, Pauly und Arghir beschreiben in [55] die Architektur der Servicebots. Die Servicebots bestehen aus drei verschiedenen Arten von mobilen Robotern. Die Servicebots sind mittel-intelligente Roboter, wobei sich die mittlere Intelligenz darauf bezieht, dass diese Roboter aufgrund von Sensor Informationen in durchschnittlich komplexen Umgebungen autonom navigieren können. Für komplexere Navigation wird mindestens ein weiterer Roboter benötigt. Servicebots können durch Fixbots und oder Softbots unterstützt werden. Fixbots sind fest in der Umgebung installiert und senden ihre Information nach Vorverarbeitung an die Servicebots. Softbots können sowohl Servicebots als auch Fixbots unterstützen, indem diese (komplexere) Software Tasks ausführen. Die Architektur ist dementsprechend als verteile Architektur anzusehen. Jede Komponente des Systems funktioniert autonom, Vorfälle werden lokal gelöst. Probleme werden im Verbund gelöst. Hierzu verfügt das verteilte System über einen zentralen Service Koordinator.

In [56] erläutern Weyns und Holvoet die SMAS Architektur für Multi-Agenten Systeme. Prinzipiell besteht die Architektur aus zwei Arten von Komponenten. Den *Agenten* und dem *Application Environment*. Die Agenten sind autonom handelnde Einheiten zur Lösung von Problemen im System. Ein Agent kann in verschiedene Umgebungen gebracht werden und dort beobachten und mit der Umwelt interagieren. Typischerweise sind verschiedene Arten von Agenten der gleichen Umgebung ausgesetzt. Die *Application Environment* Komponente koordiniert die Agenten, beispielsweise indem Ressourcen freigegeben werden oder Beschränkungen vorgegeben werden. In der Veröffentlichung werden weiterhin detailliert die einzelnen Aufgaben, sowie die Kooperation der beiden Komponenten dargestellt. Die Autoren sehen die beschriebene Architektur als Hilfe für die Entwicklung von konkreten Software Architekturen.

Álvarez, Iborra, Alonso und De la Puente stellen in [57] eine Referenzarchitektur für Teleoperationen in der Robotik vor. Hierbei wird das System in folgende Komponenten unterteilt: Kommunikation, Fernsteuerungs-Einheit, User-Interface, Grafische Repräsentation, Kollisions-Erkennung, sowie die Controller zur Steuerung der Hardware. Zur Interaktion dieser Komponenten wird zum einem eine typische Server-Client Architektur verwendet, zum anderen der ereignisgesteuerte Architekturansatz. Der Server-Client Ansatz wird zwischen dem Controller und der Kollision-Erkennung und der grafischen Repräsentation verwendet. Der ereignisgesteuerte Ansatz wird zwischen dem User-Interface, dem Kommunikations-Modul und dem Controller verwendet, weil dort die Aktion eigenständig von jedem Modul gestartet werden kann.

In [58] wird die AIS Architektur erläutert. Hayes-Roth, Pfleger, Lalanda, Morignot und Balabanovic zeigen eine Schichtenarchitektur aus einer physischen und einer kognitiven Ebene. Die physische Ebene beobachtet die Umgebung und agiert in dieser. Die kognitive Ebene ist für abstraktere bzw. komplexere Aktionen, wie Planung oder Problemlösung zuständig. Ein Fokus der gezeigten Architektur ist die Adaption der Roboter an verschiedene Aufgaben, bzw. unterschiedliche Umgebungen. Hierzu ist die Software in einer Komponenten-Bibliothek implementiert, sodass einzelne Komponenten mit einem *Application Configuration Tool* kombiniert werden können.

Ein Blick in die Automobilindustrie lohnt im Fall der mobilen Robotik ebenfalls. Zwar sind heute noch erhebliche Unterschiede zwischen beiden Bereichen vorhanden, diese könnten sich in Zukunft jedoch auflösen. Ein autonom fahrendes Auto ist aus technischer Sicht nicht großartig von mobilen Roboter zu unterscheiden, wenngleich es sich hierbei um bemannte mobile Robotik handeln wird, welche besonders im Bereich Sicherheit noch höhere Anforderungen mit sich bringt, als heute in der mobilen Robotik üblich ist. Als Standard-Architektur wird im Automobilsektor vermehrt auf AUTOSAR [O10] gesetzt. In der Schichtenarchitektur von AUTOSAR befindet sich zwischen der Hardware, also den Mikrocontrollern und der Applikations-Ebene eine *Runtime Environment (RE)* Zwischenschicht. Diese Zwischenschicht setzt auf standardisierte Schnittstellen zu den Treibern und weiteren Diensten, wie beispielsweise Netzwerkprotokolle und abstrahiert diese für die Applikationsschicht. Somit ist diese Applikationsschicht weitestgehend von der Hardware entkoppelt und somit einfacher zu implementieren und auf verschiedenen Plattformen einsetzbar. Ziel der AUTOSAR Architektur ist somit die Vereinheitlichung der Software und letztendlich die Senkung des immensen Aufwandes der Softwareentwicklung für Automobile.

3 Anforderungen an die Systemarchitektur von mobilen Robotern

Die Robotik ist seit Jahrzehnten ein weit verbreitetes Forschungsfeld. Es sind verschiedene Teilthemen, wie beispielsweise die mobile Robotik entstanden. Oftmals wird dabei von **der** mobilen Robotik gesprochen. Dieses Kapitel beschäftigt sich mit der Frage, wie die mobile Robotik definiert wird, beispielsweise welche Anwendungsgebiete und typischen Fähigkeiten mobile Roboter haben und welche Anforderungen und Herausforderungen hieraus an eine Systemarchitektur folgen. Diese Systemarchitektur soll generisch auf den typischen mobilen Roboter abbildbar sein. In Kapitel 3.1 werden zunächst die Grundlagen der Robotik kurz erläutert. Die Ergebnisse einer systematischen Literaturrecherche (Kapitel 3.2) werden in Form einer Taxonomie (Kapitel 3.3) dargestellt. Diese Taxonomie beschreibt einen archetypischen mobilen Roboter, ein abstraktes Modell eines mobilen Roboters, welcher die Grundlage für die Entwicklung einer geeigneten Architektur bildet. Hierzu werden aus der Taxonomie Anforderungen und Herausforderungen an die Systemarchitektur abgeleitet (Kapitel 3.4). Die Literaturrecherche beschreibt zudem den Stand der Technik von mobilen Robotern, indem Merkmale, wie beispielsweise die Fähigkeiten von mobilen Robotern, kurz dargestellt werden.

Der Autor hat die Ergebnisse der nachfolgend gezeigten Taxonomie und deren Erarbeitung (Kapitel 3.2 und Kapitel 3.3) bereits in [A14][1] veröffentlicht.

3.1 Grundlagen der Robotik

Die Grundlagen der Robotik sind umfassend und komplex. Nachfolgend werden wichtige Grundlagen kurz dargestellt, welche für das Verständnis dieser Dissertation notwendig sind. Für umfassendere Erläuterungen zu den Grundlagen der Robotik empfiehlt der Autor u. a. die Bücher: *Artificial Intelligence: A Modern Approach* [59] von Russell und Norvig, *Probabilistic Robotics* [60] von Thrun, Burgard und Fox und *Springer Handbook of Robotics* [61].

[1] Quellennachweise der eigenen Veröffentlichungen werden mit dem Prefix **A** *(für Autor)* kenntlich gemacht ([A*Nr.*]).

3.1.1 Definition und Paradigmen der Robotik

Der Begriff der Robotik oder die Bezeichnung Roboter findet in verschiedenen Kontexten, beispielsweise Science-Fiction Literatur oder technischen Geräten Anwendung und wird durchaus in unterschiedlichen Epochen unterschiedlich definiert. Corke schreibt in [62, S. 1], dass sowohl die ersten automatisierten Systeme, als auch moderne Roboter „eine physische Aufgabe ausführen und programmierbar sind“. Als Beispiel einer historisch programmierbaren Maschine nennt Corke eine mechanische Ente aus 1739 *(Vaucanson's Ente)*. Aktuelle Roboter unterteilt Corke aufgrund ihrer Funktion in die vier Bereiche *Industrie*[2], *Service*, *Feld*[3] und *Humanoide* Roboter.

Industrieroboter werden in Fabriken eingesetzt und gehören zu den technologischen Trägern der ersten Generation der modernen Roboter. Serviceroboter unterstützen den Menschen beispielsweise beim Säubern in Form von Saug- und Wischrobotern. Die Gruppe der Feldroboter werden typischerweise im Außenbereich verwendet, beispielsweise zur Überwachung eine Areals oder in der Landwirtschaft. Humanoide Roboter werden heute in der Regel auch zur persönlichen Unterstützung des Menschen eingesetzt.

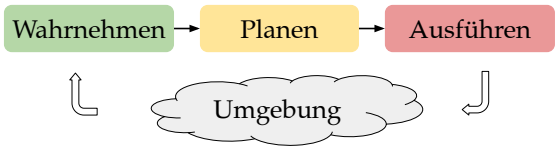
Eine weitere Definition liefert das IEEE [O11]:

> „Ein Roboter ist eine autonome Maschine, welche imstande ist, die Umgebung zu erfassen, Berechnungen durchzuführen um Entscheidungen zu treffen und Handlung in der realen Welt auszuführen.“

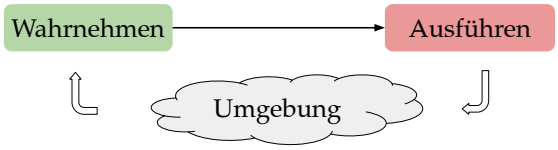
Letztendlich bedient der Begriff der Robotik ein großes Feld von simpel bis komplex, in unterschiedlichsten Formen und Anwendungsgebieten. Das Grundprinzip, aus *Wahrnehmen*, *Planen* und *Ausführen* wird in allen Bereichen der Robotik angewendet.

Durch die Überwachung der Umgebung bzw. Umwelt werden Informationen für den jeweiligen Anwendungszweck generiert. Nachfolgend werden diese Informationen verarbeitet. Aufgrund der ausgewerteten Sensordaten werden Entscheidungen getroffen, ob und wie das System auf die Sensordaten agiert. Das Ausführen dieser Entscheidung wiederum nimmt Einfluss auf die Umgebung und ist somit ein Beispiel für ein CPS. Dieser Ablauf kann unendlich fortgesetzt werden und ist je nach Art des Roboters und der Situation, in dem sich dieser befindet unterschiedlich komplex. Die Art der Nutzung und Verarbeitung dieser Sensorinformationen wird in *Introduction to AI Robotics* [63] in drei Paradigmen der Robotik (siehe Abbildung 3.1) unterschieden und detailliert erläutert. In [64] werden die drei Paradigmen der Robotik zusammengefasst.

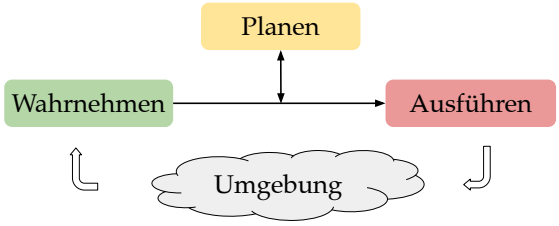
[2] engl. Manufacturing
[3] engl. Field



(a) Hierarchisches/ deliberatives Paradigma



(b) Reaktives Paradigma



(c) Hybrides deliberatives/ reaktives Paradigma

Abbildung 3.1: Paradigma der Robotik (basiert auf [63, S. 7])

Murphy [63] beschreibt das hierarchische oder deliberative oder beratende Paradigma als das älteste Paradigma, welches in den Jahren 1967 bis 1990 relevant war. Bei diesem Paradigma wird nach jedem Schritt des Wahrnehmens geplant, welche Aktion als nächstes ausgeführt werden soll. Das hat den Nachteil, dass das Paradigma stark von der Planung abhängt. Hierbei werden in jedem Schritt alle verfügbaren Daten zusammengeführt und die Aktion aufgrund aller verfügbaren Daten im System geplant. Da diese Planung des Gesamtsystems in jedem Schritt ausgeführt wird, ist die Reaktionsgeschwindigkeit des Roboters limitiert. [63, S. 6 ff., S. 41 ff.]

Murphy [63] beschreibt das reaktive Paradigma als Reaktion auf das hierarchisch/ deliberative Paradigma. Das reaktive Paradigma wurde insbesondere zwischen 1988 und 1992 benutzt. Hierbei wird das Gesamtsystem in viele einzelne Funktionen unterteilt. Bei diesen Funktionen bedarf es nicht unbedingt in jedem Schritt einer Planung. Sensorinformationen, wie das Erkennen eines Hindernisses können so direkt und ohne weitere Planung in die Aktion „weiche aus“ umgesetzt werden. Dieses wird auch als reaktives Verhalten bezeichnet. Infolgedessen führt das reaktive Paradigma zu sehr viel schnellerem Reagieren des Roboters. Es stellte sich jedoch heraus, dass das Weglassen der

Planung nur in Einzelfällen möglich ist, die Planung jedoch für viele Anwendungsfälle benötigt wird. [63, S. 6 ff., S. 105 ff.]

Murphy [63] beschreibt das hybride deliberative/ reaktive Paradigma als aktuellen Stand der Technik, welches ab den 1990er Jahren relevant wurde. Hierbei wird zunächst in einer Missionsplanung berechnet, wie eine Aufgabe erfüllt werden kann. Als Ergebnis dieser Planung werden Teilaufgaben erstellt, die als reaktives Verhalten, also mit dem reaktiven Paradigma realisiert werden können. Das hybride deliberative/ reaktive Paradigma bietet somit schnell agierendes reaktives Verhalten und kombiniert dieses mit einer übergeordneten Planungsphase. Hierbei kann die Planung beispielsweise alle 5 Sekunden erfolgen, während das reaktive Verhalten oft alle 1/60 Sekunden durchgeführt wird. [63, S. 6 ff., S. 257 ff.]

Die Robotik kann nicht strikt einem Fachgebiet zugeteilt werden, sondern ist vielmehr ein interdisziplinäres Zusammenspiel verschiedener Fachrichtungen. Der Maschinenbau kommt bei der Entwicklung der Mechanik zum Einsatz, die Elektrotechnik bei der Implementierung von digitaler Hardware, wie beispielsweise eigener Sensor- und Aktuator-Komponenten. Die Softwareentwicklung der Planungskomponente fällt in das Gebiet der Informatik.

3.1.2 Mobile Robotik

Mobile Roboter sind in der Lage, sich in ihrer Umgebung zu bewegen und sind dabei nicht auf einen Ort beschränkt. Mobile Roboter nutzen oft ähnliche oder gleiche Prinzipien und Funktionen wie andere Arten von Robotern, unterscheiden sich im Detail jedoch von diesen. So ist beispielsweise die Navigation der Roboter eine notwendige Funktion bei mobilen Robotern, nicht jedoch bei stationären Robotern. Weiterhin ergeben sich durch den Anwendungszweck der mobilen Roboter spezifische Anforderungen an diese, wie beispielsweise die Notwendigkeit einer mobilen Energieversorgung, welche im Verlauf dieses Kapitels erläutert werden.

Die Definition der Klassen von mobilen Robotern erfolgt meist über die Art der Bewegung, die Umgebung, in der der mobile Roboter eingesetzt wird oder über besondere Merkmale. In *Springer Handbook of Robotics* [61] werden fünf Klassen von mobilen Robotern definiert, die diese Charakteristika umfassen.

Mobile Roboter können grob in drei verschiedenen Fortbewegungsarten kategorisiert werden. Eine klassische Art der mobilen Roboter, auf dessen Fokus diese Arbeit liegt, sind die fahrbaren Roboter, sogenannte Wheeled Mobile Robots (WMR). Siciliano und Khatib beschreiben in [61, S. 575 ff.] die Gruppe der WMR als weit verbreitete mobile Roboter,

welche viele Vorteile, wie die simple Struktur, Energie-Effizienz, hohe Geschwindigkeit und geringe Produktionskosten aufweisen.

Die zweite Gruppe, die Unterwasserroboter oder unbemannte Unterwasser-Fahrzeuge (Unmanned Underwater Vehicle (UUV)) werden in [61, S. 595 ff.] beschrieben. UUVs werden u. a. zur Untersuchung von Umweltfragen und zur Bekämpfung der Verschmutzung der Meere eingesetzt. Unterwasserroboter sind tendenziell als Remotly Operated Vehicle (ROV) umgesetzt und werden somit ferngesteuert betrieben. Der Datentransfer erfolgt oft mittels eines Kabels am Roboter. Neben ROV gibt es autonom agierende Roboter, sogenannte Autonomous Mobile Robot (AMR). Diese navigieren autonom und erfüllen ihre Aufgabe unter Wasser autark. Die Gruppe der AMR sind vermehrt auch in anderen Bereichen der Robotik zu finden, beispielsweise der Gruppe der WMR. [61, S. 595 ff.]

Eine untergeordnete Rolle hat aktuell die Klasse der unbemannten Wasserfahrzeuge (Unmanned Marine Vehicle (UMV)) inne. Hierbei handelt es sich um schwimmende Roboter, welche beispielsweise zur Inspektion von Brücken verwendet werden [65]. In [66] wird zudem u. a. der Einsatz von UMVs zur Bewältigung von Katastrophen, wie Unfällen mit Ölverschmutzungen genannt. Nicht betrachtet wird die Gruppe der UMVs in *Springer Handbook of Robotics* [61].

Gänzlich ohne Kabel müssen fliegende Roboter auskommen. Hier hat die Autonomie einen noch größeren Anteil, als bei Unterwasserrobotern, weil das Fernsteuern von fliegenden Robotern aus großer Entfernung nicht einfach ist. Hier werden zumindest autonome Funktionen, wie die automatische Kollisionsvermeidung benötigt. Die fliegenden Roboter gehören der Gruppe der unbemannten Flugzeuge an (UAV). Zusammen mit der Infrastruktur, System- und Mensch-Maschinen-Interfaces werden diese auch als Unmanned Aircraft System (UAS) bezeichnet. In [61, S. 623 ff.] wird erläutert, dass besonders die Gruppe der fliegenden Roboter komplexe Systeme sind, da diese, beispielsweise im Vergleich zu WMR, sechs Freiheitsgrade haben und die Verarbeitung der Navigation selbst eine komplexe Aufgabe eines solchen mobilen Roboters ist. Außerdem sind fliegende Roboter streng limitiert, was die Flugdauer, Nutzlastkapazität und Größe betrifft. Ein großer Vorteil ist das Erreichen von großen Entfernungen beispielsweise mit Flächenfliegern. Multicopter sind günstige Alternativen für tiefere Höhen und eignen sich beispielsweise für die Überwachung kritischer Infrastrukturen [67] oder den Transport von Ladung, z. B. von medizinischen Artikeln [68]. Hierbei sind Multicopter agiler als Flächenflieger und können, u. a. aufgrund ihrer Rotoren an einem Punkt in der Luft „stehen bleiben“.

Als weitere Klasse der mobilen Roboter können bionische Roboter bezeichnet werden [61, S. 543 ff.]. Bionische Systeme, bzw. die Bionik beschreibt die Übertragung von Vorbildern aus der Natur auf technische Systeme. Bei mobilen Roboter bezieht sich die Bionik oftmals

auf die Fortbewegungsart. So gibt es mobile Roboter in Wurm- oder Schlangenform, welche die Fortbewegungsart dieser Tiere kopieren. Als bionische Roboter können auch humanoide Roboter bezeichnet werden, wie beispielsweise der bereits vorgestellte Roboter NAO, welcher dem menschlichen Körper nachempfunden ist. Humanoide Roboter werden besonders in der Kooperation mit Menschen eingesetzt, weil laut Goetz, Kiesler und Powers [69] durch die vertraute Form, die Akzeptanz des Roboters durch den menschlichen Partner gefördert werden kann.

Die Klasse der Micro- und Nano-Roboter betrifft die kompakte Bauweise der Roboter [61, S. 671 ff.]. Typischerweise ist die Größe von Roboter der Micro/ Nano Klasse im Bereich von Nanometern bis Millimetern. Mobile Micro/ Nano Roboter werden u. a. für den Einsatz in der Medizin erforscht, bei dem diese Roboter in Lebewesen eingesetzt werden [70]. Da diese Roboter allein wegen der Größe nicht mehrere Rechner integrieren und somit nicht als verteiltes System realisiert werden, wird diese Klasse der mobilen Roboter im Folgenden nicht weiter betrachtet.

3.1.3 Künstliche Intelligenz

In der Robotik und insbesondere im Bereich der mobilen Roboter bedarf es typischerweise eines hohen Maßes an Autonomie, d. h. der Fähigkeit eigenständig agieren und somit eigene Entscheidungen treffen zu können. So müssen UAV beispielsweise autonom Kollisionen vermeiden, wenn sie außer Sichtweite geflogen werden. WMR werden ebenfalls oft in Szenarien eingesetzt, bei denen diese autonom agieren, wie beispielsweise im Katastrophenschutz. Ein hohes Maß an Autonomie wird u. a. durch Künstliche Intelligenz (KI) erreicht.

Russell und Norvig beschreiben in [59, S. 1] die künstliche Intelligenz als das Bestreben intelligente Instanzen zu erschaffen. Über tausende von Jahren wurde erforscht, wie die menschliche Intelligenz funktioniert. Seit etwa dem Ende des Zweiten Weltkrieges liegt der Fokus darauf, wie diese Intelligenz künstlich nachgebildet werden kann. Laut Russell und Norvig ist dieser Forschungsbereich das weit verbreitetste Forschungsfeld mit unzähligen nicht gelösten Fragestellungen. In [59, S. 2 ff.] wird die KI in vier Bereiche aufgeteilt:

a) Menschliches Denken
b) Menschliches Handeln
c) Rationales Denken
d) Rationales Handeln

Um zu entscheiden, ob eine künstliche Intelligenz menschliches Handeln imitieren kann, wurde bereits vor 70 Jahren der Turing-Test [71] veröffentlicht. Hierbei dürfen Proband*innen nicht erkennen, ob sie mit einem Menschen oder einer KI kommunizieren. Bis heute ist dieser Test nicht vollständig bestanden worden.

Rationales Handeln und Denken ist heute jedoch Stand der Technik in technischen Systemen und besonders im Feld der Robotik intensiv erforscht. Heute wird die KI auch in starke oder schwache KI klassifiziert. Die heute kommerziell erfolgreichen Saugroboter gehören hierbei der Klasse der schwachen KI an. Leistungsfähigere KI im Bereich der Robotik ist beispielsweise bei den selbstfahrenden Autos zu finden. So hat das selbstfahrende Auto „Stanley“, um Projektleiter Sebastian Thrun, im Jahr 2005 als erstes Auto in einem Rennen mit selbstfahrenden Autos das Ziel einer ca. 210 km langen Strecke erfolgreich eigenständig erreicht [72].

3.2 Literaturrecherche

Nachfolgend wird eine systematische Literaturrecherche durchgeführt [73, 74], welche zur Definition eines typischen mobilen Roboters angewendet wird. Zur Durchführung einer systematischen Literaturrecherche müssen zunächst Suchbegriffe definiert werden. Diese Suchbegriffe werden bei der Durchsicht von drei Referenzarten (Buchquellen, Übersichtsarbeiten[4] und Referenzsysteme) und einer Umfrage ermittelt (siehe Abbildung 3.2). Die drei Referenzarten werden auf wiederkehrende Merkmale der mobilen Robotik untersucht. Parallel werden in einer Umfrage mit vier Teilnehmern aus dem Forschungsbereich der Robotik am Institut für die Digitalisierung von Arbeits- und Lebenswelten (IDiAL) der Fachhochschule Dortmund weitere typische Merkmale von mobilen Robotern definiert. Die ermittelten Suchbegriffe werden in die Kategorien *Fähigkeiten* und *technische Realisierungen* unterteilt und u. a. in den Tabellen 3.3 und 3.4 aufgelistet.

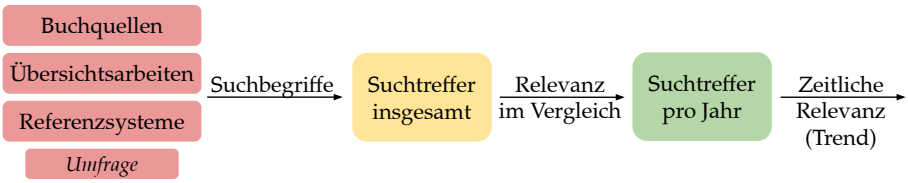


Abbildung 3.2: Ablauf der Literaturrecherche

[4] engl. Survey paper

Im weiteren Verlauf dieses Kapitels werden die Anzahl an Suchtreffern für die definierten Suchbegriffe mittels der bereits vorgestellten Bibliothek IEEE Xplore ermittelt und in den Abbildungen 3.5 und 3.10 gegenübergestellt. Dieses gibt Aufschluss über die Relevanz der *Fähigkeiten* und *technischen Realisierungen* im Vergleich zueinander.

Nachfolgend wird die Recherche vertieft, indem die Anzahl der Suchtreffer für jeden einzelnen Suchbegriff im Verlauf der letzten zehn Jahre betrachtet wird. Diese zeigt den jeweiligen Trend jeder *Fähigkeit* oder *technischen Realisierung*.

Zunächst werden jedoch, in den nächsten drei Abschnitten, die drei erwähnten unterschiedlichen Referenzarten kurz vorgestellt und nachfolgend in Form von Tabellen verglichen und zusammengefasst. Zuerst werden übliche Buchquellen untersucht (3.2.1). Nachfolgend wird eine Auswahl an Veröffentlichungen dargestellt, welche einen Überblick über den Stand der Technik in der Robotik darstellen (3.2.2). Weitere Suchbegriffe, bzw. das letzte Validieren dieser Suchbegriffe erfolgt anhand einiger Referenzsysteme aus der mobilen Robotik (3.2.3). Die Relevanz jeder gewählten Veröffentlichung wird in Tabelle 3.1 aufgelistet. Hierzu wird sowohl die Suchmaschine für wissenschaftliche Veröffentlichungen *Google Scholar*[5] [O12], als auch die digitale Bibliothek IEEE Xplore genutzt.

Die Auswahl der diversen Veröffentlichungen erfolgt nicht einzig anhand der Relevanz. Es wurde bewusst ein breites Spektrum aus etwas älteren bis aktuellen, sowie oft bis wenig zitierte Veröffentlichungen ausgewählt. So soll sichergestellt werden, dass bei der Definition der Suchbegriffe, nicht nur die üblichen Forschungsthemen gefunden werden, sondern z. B. neue Themen betrachtet werden. Die Quantifizierung und Analyse erfolgt anschließend anhand der Anwendung der Suchbegriffe mit der ausgewählten Suchmaschine und nicht an der Auswahl der Suchbegriffe.

[5] Google Scholar führt eine Suche in allen, in Google Scholar verfügbaren, wissenschaftlichen Veröffentlichungen durch. Die Suche ist hierbei im Gegensatz zu IEEE Xplore unabhängig vom Herausgeber.

Typ	Referenz: Titel/ Bezeichnung [Quelle]	Jahr	Anzahl Zitierungen/ Suchergebnisse*: Google Scholar	IEEE Xplore
Bücher aus 3.2.1	Artificial Intelligence [59]	2009	35011	N/A
	Probabilistic Robotics [60]	2005	10226	N/A
	Springer Handbook of Robotics [61]	2016	3974	N/A
	Intro. to Auton. Mobile Robots [75]	2011	3665	N/A
	Robotics, Vision and Control [62]	2016	1545	N/A
	Embedded Robotics [76]	2008	568	N/A
Übersichtsarbeiten aus 3.2.2	Modular Self-Reconfigurable Robot Systems [77]	2007	813	N/A
	The Grand Challenges of Science Robotics [78]	2018	303	N/A
	Multi-Robot Taxonomy and Survey [79]	2016	126	N/A
	Robotic Challenges for Mars Exploration [80]	2000	86	N/A
	Robot Teleoperation Taxonomy [81]	2015	33	N/A
	Review of Mobile Robots [82]	2019	20	N/A
	ANR Design and Requirements [83]	2018	3	N/A
	Review on Challenges of Mobile Robots [84]	2020	1	N/A
Referenzsysteme aus 3.2.3	Spot [O13]	2019	945	70
	Khepera IV [85]	2015	187	46
	TurtleBot3 [O14]	2017	180	35
	AMiRo [86]	2018	105	15
	WolfBot [87]	2014	80	10
	e-puck2 [88]	2018	75	8
	OmniMan [89]	2020	32	1
	ArEduBot [90]	2011	24	1
	Savvy [91]	2017	6	2
	Arduino Robot [92]	2018	5	1
	Omnidirectional Mobile Robot [93]	2016	4	0

*Für die Referenzsysteme wird nicht die Anzahl an Zitierungen, sondern die Ergebnisse einer Suchmatrix angegeben. Die entsprechende Suchmatrix ist in Anhang A, Tabelle A1 aufgeführt.

Tabelle 3.1: Relevanz der ausgewählten Quellen

3.2.1 Buchquellen

Artificial Intelligence: A Modern Approach [59] von Russell und Norvig ist ein Standardwerk im Bereich KI und Robotik. Das Buch wurde seit der Veröffentlichung der ersten Version in 2009 bereits über 35.000 Mal zitiert (siehe Tabelle 3.1) und wird vielerorts als Lehrbuch eingesetzt. Der Fokus des Buches liegt auf der KI und deren Anwendung, beispielsweise in der Robotik. Es werden detailliert Hintergründe, u. a. zur Problemlösung, Planung, Vorhersage, Lernen und Wahrnehmung mit oder durch KI erläutert. Wenngleich das Kapitel zur Robotik vergleichsweise kurz gefasst ist, wird beispielsweise ein Überblick über Hardware, Softwarearchitekturen oder Anwendungsgebiete aufgezeigt.

Probabilistic Robotics [60] von Thrun, Burgard und Fox ist ein weiteres unverzichtbares Buch über die Robotik und die notwendigen mathematischen Hintergründe zur Arbeit mit dieser. Im Buch werden zunächst die mathematischen Hintergründe, wie beispielsweise gaußsche Filter beschrieben. Dieser und viele weitere Algorithmen und Filter werden in den Kapiteln Lokalisierung, Kartografie und Planung angewendet. *Probabilistic Robotics* zeigt einen sehr hohen Detailgrad und viele praktische Beispiele. Die Popularität des Buches zeigt sich auch in der hohen Relevanz in der Wissenschaft mit über 10.000 Zitierungen.

Springer Handbook of Robotics [61] ist, mit ca. 2.200 Seiten, auch in der Tiefe ein sehr guter Überblick über die Robotik. Auch in der vorliegenden zweiten Version aus 2016 beschreiben eine große Anzahl von Autor*innen den aktuellen Stand der Technik. Im Buch werden die Grundlagen der Robotik, das Design von Robotern, Sensorik und Wahrnehmung, Manipulation und Schnittstellen, Bewegung in der Umwelt, Roboter im Arbeitsumfeld und zuletzt das Kapitel „Roboter und Menschen“ gezeigt. Siciliano und Khatib beanspruchen für ihren Titel eine sehr hohe Aktualität. Das Buch (beide Versionen) wurde ca. 4.000 mal zitiert.

Introduction to Autonomous Mobile Robots [75] ist im Gegensatz zu den bisher vorgestellten ein Buch, welches sich speziell mit mobilen Robotern befasst. So beginnen Siegwart, Nourbakhsh und Scaramuzza mit einem Überblick über die verschiedenen Klassen der mobilen Roboter und erläutern dann im Detail die Kinematik mobiler Roboter. Die Wahrnehmung, Lokalisierung, Planung und Navigation unterscheidet sich im Grunde nicht von der allgemeinen Robotik und wird in den nachfolgenden Kapiteln beschrieben. Besonders hilfreich sind die vielen Beispiele aus der mobilen Robotik, wie beispielsweise die Vorstellung von typischen Sensoren.

Robotics, Vision and Control: Fundamental Algorithms in MATLAB [62] vereinfacht die Arbeit mit Robotern sehr stark, insofern man die Absicht hat Mathworks MATLAB [O15] einzusetzen. Corke stellt eine große Anzahl von Algorithmen und Funktion dar, welche

direkt in MATLAB verwendet werden können. Hierzu ist zu den praktischen Beispielen der MATLAB Quelltext abgedruckt. Ein großer Teil des Buches gilt der Bildverarbeitung[6], dessen Anwendung in MATLAB besonders gut unterstützt wird. Außerdem wird die Modellierung von reaktiven Systemen, wie mobilen Robotern in MATLAB Simulink [O16] anhand von praxisnahen Beispielen erläutert.

Embedded Robotics [76] von Bräunl zeigt die Kombination aus eingebetteten Systemen und mobilen Robotern. Das Buch gibt zunächst einen Einblick in eingebettete Systeme und erläutert anschließend das Design von mobilen Robotern mit diesen. Im letzten Abschnitt werden Anwendungen der mobilen Robotik dargestellt. *Embedded Robotics* ist hierbei immer auf die hardwarenahe Entwicklung fokussiert und nicht, wie die meisten der anderen vorgestellten Bücher, auf die Algorithmen und technischen und mathematischen Hintergründe. Hierbei geht Bräunl detailliert bis in die Bit- oder Logikebene herunter.

3.2.2 Übersichtsarbeiten

In [77] werden Herausforderungen und Chancen für Roboter dargestellt. Der Beitrag aus 2007 wurde bereits etwa 800 mal zitiert und bezieht sich nicht explizit auf die mobile Robotik, zeigt jedoch einige Trends in der Robotik, welche auch auf die mobile Robotik abbildbar sind. Der Fokus der Veröffentlichung liegt auf modularen und selbst rekonfigurierbaren Systemen. Diese sind laut Yim, Shen, Salemi, Rus, Moll, Lipson, Klavins und Chirikjian besonders flexibel einsetzbar und somit für zukünftige Aufgaben gewappnet. Weiterhin ist es eine Herausforderung, Roboter robust zu konstruieren und zu implementieren. Hinzu kommt, dass sich Roboter selbst reparieren können müssen, wenn sie beispielsweise für längere Zeit auf unerreichbarem Gelände, wie z. B. auf dem Mars, agieren sollen. Algorithmen, welche die optimale Konfiguration des Systems berechnen, werden genauso gesucht, wie eine effiziente und skalierbare Kommunikation zwischen verschiedenen Robotern in einem größeren Verbund.

In [78] werden Trends für Roboter in Lehre und Forschung erläutert. Mit (1) neuen Materialien und Fertigungsmethoden, (2) bio-hybride und bio-inspirierte Roboter, (3) Energie, (4) Schwarm-Robotik, (5) Navigation und Erkundung, (6) KI für Roboter, (7) Gehirn-Computer-Interfaces, (8) soziale Interaktion, (9) medizinische Roboter und (10) Ethik und Sicherheit werden 10 Trends definiert. Die Veröffentlichung aus 2018 zeigt hierbei ein weites Spektrum an übergeordneten Trends, welche auf die mobile Robotik abbildbar sind. Trotz der relativ kurzen Verfügbarkeit von ungefähr zwei Jahren, wurde die Veröffentlichung bereits mehr als 300 mal referenziert.

[6] engl. Computer Vision (CV)

In [79] wird eine Übersicht über die Zielerkennung und Zielverfolgung in einem Multi-Roboter System in Form einer Taxonomie dargestellt. Zwar handelt es sich hierbei nicht um eine Übersichtsarbeit über mobile Robotik im Allgemeinen, einzelne Aspekte können jedoch vereinheitlicht und übernommen werden. So beschreiben Robin und Lacroix beispielsweise, dass es notwendig ist, dezentrale Algorithmen einzusetzen, um die Zielerkennung und Zielverfolgung mit mobilen Robotern durchzuführen. Außerdem wird erläutert, dass die Analyse und Überwachung der Zielsysteme notwendig ist. Diese Systemüberwachung sollte direkt auf den Zielsystemen und nicht nur in Simulationen erfolgen.

Die Anforderungen an die Hardware bei der Entwicklung eines mobilen Roboters zur Mars Erforschung werden in [80] beschrieben. Huntsberger, Rodriguez und Schenker zeigen hierbei ausschließlich die Herausforderungen an die Hardware von mobilen Roboter, nicht jedoch die Anforderungen an die Software. Besonderer Fokus liegt auf der Entwicklung einer adäquaten Energieversorgung für den mobilen Roboter. Außerdem wird beschrieben, dass es notwendig ist, dass der Roboter in Teilen in der Lage sein muss, eigene Defekte an der Hardware festzustellen und zu reparieren.

In [81] wird eine Taxonomie für Richtlinien zur Entwicklung von Teleoperationen mit Robotern gezeigt. Weiterhin liegt ein Fokus der Veröffentlichung auf der Integration von Endnutzer*innen in das System, indem Richtlinien zur Bedienbarkeit dargestellt werden. Adamides, Christou, Katsanos, Xenos und Hadzilacos definieren acht Kategorien in ihrer vorgestellten Taxonomie. (1) Architektur und Skalierbarkeit, (2) Fehler-Prävention und Fehlerkorrektur, (3) visuelles Design des Human Machine Interface (HMI)[7], (4) Darstellungsform der Information, (5) Anzeige des Status des Roboters, (6) Effektivität und Effizienz der Interaktion, (7) Wahrnehmung der Roboter-Umgebung und (8) kognitive Faktoren.

Rubio, Valero und Llopis-Albert zeigen in 2019 einen Überblick über Konzepte, Methoden, Frameworks und Anwendungen von mobilen Robotern [82]. Neben der bereits erwähnten Einteilung in Land-, Luft- und Wasserroboter wird auch der Anwendungsbereich der Navigation detailliert erläutert. Hier unterscheiden die Autoren beispielsweise zwischen der Lokalisierung und der Pfadplanung. Außerdem werden typische Sensoren vorgestellt und in zwei Kategorien, (1) interne Sensorik, welche interne Daten wie die Geschwindigkeit der Motoren oder die Akkumulatorkapazität betrachten und (2) passive oder aktive Sensoren, welche die Umgebung erfassen, unterteilt.

[7] dt. Mensch-Maschinen Schnittstelle

In [83] werden Konzepte und Anforderungen an Autonomous Networked Robots (ANR)[8] beschrieben. Außerdem werden verschiedene Architekturformen für Multi-Roboter Systeme miteinander verglichen. So sind laut Chukwuemeka und Habib zentrale Architekturen in Bezug auf Zuverlässigkeit, Robustheit und Skalierbarkeit hybriden Architekturmodellen aus hierarchischen und dezentralen Ansätzen unterlegen. Zentrale Architekturen hingegen schließen im Bereich Effizienz besser ab. Die Nutzung von mobilen Robotern in ANR sind laut der Autoren besonders in Überwachungs- und Erkundungsszenarien geeignet, indem die einzelnen Roboter in Zonen eingeteilt werden, welche sie auch aus Gründen der Kommunikation untereinander einhalten müssen.

Alatise und Hancke zeigen in [84] aktuelle Herausforderungen an autonome und mobile Roboter, sowie an Sensordatenfusions-Methoden. Die Navigation ist laut den Autor*innen in allen Anwendungsgebieten mobiler Roboter notwendig. Weitere Herausforderungen für mobile Roboter sind die Pfadplanung, Hindernisvermeidung und die Lokalisierung. Die Sensordatenfusion ist laut Alatise und Hancke notwendig, um zuverlässige Informationen der Sensorik zu bekommen. Fehlerhafte Datensätze oder Sensoren können in der Sensordatenfunktion erkannt und herausgerechnet werden. Erläutert wird weiterhin, welche Methodik für welche Gruppe von Sensoren Stand der Technik sind.

3.2.3 Referenzsysteme

Dieser Abschnitt erweitert den Stand der Technik mit der Betrachtung von Referenzsystemen aus der mobilen Robotik. Um eine Vergleichbarkeit zur experimentellen Umsetzung eines mobilen Roboters[9] herzustellen, wird hierbei nur die Gruppe der WMR betrachtet. Als Referenzsysteme werden sowohl weit verbreitete Systeme, wie beispielsweise der Khepera [85], als auch dem Autor bekannte oder zur Verfügung stehende Systeme wie der AMiRo [86] ausgewählt. Damit auch bei den Referenzsystemen ein weites Spektrum von Systemen betrachtet wird, werden zudem zufällig ausgewählte und wenig verbreitete und zitierte Referenzsysteme untersucht. Die Relevanz in der wissenschaftlichen Diskussion kann anhand von der Anzahl an Zitierung gemessen werden (siehe Tabelle 3.1). Die nachfolgenden Referenzsysteme werden detaillierter untersucht und in die Ergebnisse der systematischen Literaturrecherche eingeordnet.

Als Stand der Technik sticht aktuell besonders das US-Robotik Unternehmen Boston Dynamics mit verschiedenen Robotern, wie Atlas oder auch Spot [O13], heraus. Boston Dynamics verstehen es, ihre Entwicklungen eindrucksvoll in Szene zu setzen, besonders in kurzen Videos. Technische Details oder Veröffentlichungen zu den Produkten von Boston Dynamics sind jedoch kaum vorhanden.

[8] dt. Autonome vernetze Roboter
[9] wird nachfolgend in Kapitel 5 gezeigt

Abbildung 3.3: Der mobile Roboter Spot [O13]

Der mobile Roboter Spot wird seit kurzem kommerziell angeboten, sodass hier einige Details veröffentlicht wurden [O17]. Spot (siehe Abbildung 3.3) ist ein mobiler Roboter, dessen Bewegungsabläufe stark von natürlichen Vorbildern inspiriert wurden. So hat Spot vier Beine und kann hiermit äußerst robust laufen und klettern. Die eingesetzte Hardware lässt sich individualisieren, so gibt es beispielsweise einen Manipulator in Form eines Greifarms oder einer drehbaren Kamera. Der Roboter kann mit einem Gamepad-ähnlichen Controller gesteuert werden, wobei beispielsweise der Bewegungsablauf intern berechnet wird. Entwickler*innen stehen eine High-Level Application Programming Interface (API)[10] und ein Software Development Kit (SDK)[11] zur Verfügung, um den Roboter für den eigenen Anwendungszweck zu programmieren oder auch mit eigener Hardware zu erweitern. Boston Dynamics gibt als Anwendungsgebiet diverse Bereiche, von Paket-Lieferungen bis zur Überwachung von sicherheitskritischen Infrastrukturen an. Letztendlich wird Spot kommerziell bisher nur an Industriepartner vertrieben, um den Einsatz in verschiedenen Anwendungsgebieten zu evaluieren.

Ein weiterer weit verbreiteter Roboter ist der Khepera IV Miniroboter, dessen erste Version bereits 1991 veröffentlicht wurde. Die aktuelle Version wird von Soares, Navarro und Martinoli in [85] detailliert evaluiert. Der mobile Roboter wird durch einen ARM Cortex-A8 Prozessor betrieben und verfügt über zahlreiche Sensoren. Hier zählen beispielsweise Infrarot- und Ultraschallsensoren zur Kollisionsvermeidung und eine RGB Kamera zur Objekterkennung. Daneben sind im Khepera IV zwei Mikrofone verbaut,

[10] Eine API (dt. Anwendungsprogrammierschnittstelle) ist eine Softwareschnittstelle, welche Programmen die notwendigen Methoden und Funktionen zur Anbindung an ein Softwaresystem bereitstellt.

[11] Ein SDK beinhaltet Tools und Software-Bibliotheken, welche zur Entwicklung von Software benötigt werden (z. B. Dienstprogramme und Konfigurationen die zum Kompilieren von Applikationen notwendig sind).

welche das Orten von Geräuschen ermöglichen. Weiterhin kann die Khepera Plattform durch Erweiterungsboards modular angepasst werden. Der Roboter wird hauptsächlich in der Lehre und Forschung eingesetzt. Der Einsteig in die Nutzung des Miniroboters wird durch die Verfügbarkeit einer API erleichtert.

Besonders beliebt als Demonstrator für ROS ist der mobile Roboter TurtleBot, dessen dritte Generation, der TurtleBot3 [O14] aktuell vertrieben wird. Der Open-Source Roboter wurde ursprünglich von Willow Garage entwickelt, welche auch ROS entwickelt haben. Der Roboter ist modular erweiterbar und verfügt in der Basis über einen 360° Light Detection and Ranging (LiDAR) Scanner, eine ARM basierte Steuereinheit, welche u. a, eine Inertial Measurement Unit (IMU) und zahlreiche Schnittstellen verarbeitet. Eine Kamera ist als kommerzielle Erweiterung verfügbar. Als Rechner kommt der SBC Raspberry Pi (3) zum Einsatz. Hervorzuheben ist die Integration von ROS und ROS 2, welche besonders Anfängern den Einstieg in die Robotik mit dem TurtleBot3 erleichtert. So gibt es zahlreiche Anwendungsbeispiele, von SLAM bis zu Hardware-Erweiterungen, um einen Manipulator, um Gegenstände zu bewegen.

Am Center for Cognitive Interaction Technology (CITEC) der Universität Bielefeld wird der mobile Mini Roboter AMiRo entwickelt. In zahlreichen Publikationen, wie [86] wird der AMiRo im Detail vorgestellt, die Entwicklung des AMiRo selbst ist in der Dissertation von Herbrechtsmeier [94] dokumentiert. Der AMiRo ist als verteiltes und modulares System

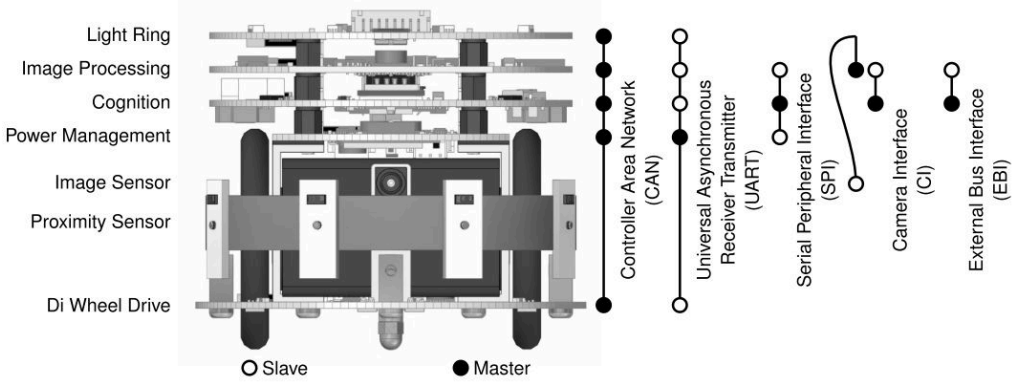


Abbildung 3.4: Aufbau und Schnittstellen des AMiRos [94, S.59]

konzipiert und verfügt über drei Hauptmodule (siehe Abbildung 3.4). Das *Di Wheel Drive*-, das *Power Management*- und *Light Ring*-Modul bilden die Basis des AMiRo und verfügen jeweils über eine eigene MCU, die unter Echtzeitbedingungen Sensor- und Aktuatordaten verarbeiten. Das *Cognition*-Modul erweitert den Roboter um einen ARM-Prozessor, welcher

ungleich leistungsstärker als die drei Basiskomponenten ist. Hier werden Applikationen und Schnittstellen zu Middlewares wie ROS oder Robotics Service Bus (RSB) [95] zur Verfügung gestellt. Das *Image Processing*-Modul wird zur Beschleunigung der Verarbeitung von CV Algorithmen durch die Nutzung eines FPGA benutzt. Für die Kommunikation zwischen den Modulen des verteilten Systems stehen verschiedene Schnittstellen, wie beispielsweise ein CAN-Bus zur Verfügung. Der AMiRo wird in der Lehre und Forschung verwendet und kann beispielsweise in Multi-Roboter Szenarien eingesetzt werden, bei denen die Farbe des AMiRo *Light Rings* der Zuordnung der einzelnen Roboter dient.

Der WolfBot [87] ist ein mobiler Roboter, welcher von Betthauser, Benavides, Schornick, O'Hara, Patel, Cole und Lobaton als Open-Source Design veröffentlicht wurde. Der WolfBot hat einen omnidirektionalen Antrieb mittels drei Mecanum Rädern und verfügt zudem über zahlreiche Sensoren. Hierzu sind u. a. Infrarotsensoren, eine Kamera und ein Mikrofon vorhanden. Als Recheneinheit wird ein BeagleBone [O18] SBC, verwendet, welcher über einen ARM Cortex-A8 Prozessor verfügt. Der Roboter kann eigenständig via SLAM Algorithmen navigieren. Hervorgehoben wird der geringe Leistungsbedarf von 3,48 W bis 6,27 W. Neben der Hardware wurde auch die Software des WolfBots als Open-Source veröffentlicht. Der Roboter wurde ebenfalls für den Einsatz in Lehre und Forschung entwickelt.

Der Roboter e-puck [88] bzw. e-puck2 [O19] ist ein kommerzieller Miniroboter, welcher durch zahlreiche Module erweitert werden kann. Die Basis des e-puck2 nutzt eine STM32F4 MCU, eine IMU, eine einfache Kamera sowie Sensoren zur Hinderniserkennung. Erweitert werden kann das System unter anderem durch eine weitere und leistungsfähigere Recheneinheit, weitere Kameras oder selbst entwickelte Module. Der mobile Roboter hat zahlreiche interne Schnittstellen zur Kommunikation. Da der Roboter kommerziell verfügbar ist, gibt es in der Literatur eine Vielzahl von Anwendungsbeispielen (siehe Tabelle 3.1), die sich im Bereich der Lehre und Forschung bewegen.

OmniMan heißt die mobile Roboter Plattform, welche von Röhrig und Heß in [89] gezeigt wird. Der Roboter verfügt über einen omnidirektionalen Antrieb aus vier Mecanum Rädern und einem mechanischen Arm mitsamt Greifer, welcher beispielsweise Sortierkästen heben kann. OmniMan wird durch einen PC gesteuert und mit einem Laserscanner zur Kartierung und Kollisionsvermeidung werden notwendige Sicherheitskriterien eingehalten. Weiterhin wird die echtzeitfähige Synchronisation zwischen der Bewegung des Roboters und dessen Arms mit dem PC gezeigt, sowie detailliert die kinematische Berechnung der Bewegung des Arms und die Navigation erläutert. Der mobile Roboter wurde für den Einsatz in Forschungslaboren zur Unterstützung des Menschen entwickelt.

Gartseev, Lee und Krovi stellen in [90] den Roboter ArEduBot vor. Als Basis werden die Plattform iRobot Create [O20] und ein Hardware Entwicklungsset der bekannten Saugroboter der Firma iRobot verwendet. Zur dedizierten Hardware des iRobot Create wird ein Arduino Board verbaut. Die Entwickler*innen des ArEduBot setzen den Fokus ihrer Veröffentlichung auch auf die Entwicklung einer Toolbox für MATLAB, mit der die Software für den mobilen Roboter modellbasiert entwickelt wird. Der dabei generierte Quelltext ist echtzeitfähig, wenngleich die erläuterte Anwendung nur aus dem Fahren des Roboters und dem Ausweichen von Hindernissen besteht und somit vergleichsweise simpel ist. Positions- und Sensordaten des Roboters können nach der Ausführung der Anwendungen via MATLAB ausgewertet und visualisiert werden. Der ArEduBot wird in der Lehre und Forschung eingesetzt.

Der namenlose Roboter auf Basis der *Savvy* Plattform [91] von Wu, Lv, Zhao, Li und Wang nutzt einen omnidirektionalen Antrieb mittels Mecanum-Rädern. Außerdem ist dieser in einer modularen Bauweise realisiert, weshalb *Savvy* durch Erweiterung für zukünftige Anwendungen vorbereitet ist. Die Architektur ist hierarchisch in eine echtzeitfähige und lokale Ebene und eine Hochleistungs-Verarbeitungsebene unterteilt. Die lokale Ebene besteht aus einem STM32 basierten MCU und einem Mini-PC. Die übergeordnete Ebene nutzt ROS und stellt beispielsweise Lokalisierung, SLAM, CV oder auch ein HMI zur Verfügung. Über die Hochleistungs-Verarbeitungsebene können mehrere Roboter in einem Multi-Roboter Szenario koordiniert werden. Der mobile Roboter kann zur Kartierung von unbekannten Umgebungen oder zur Verfolgung von Fußgängern mittels einer 3D-Tiefenkamera eingesetzt werden.

Meghana, Nikhil, Murali, Sanjana, Vidhya und Mohammed stellen in [92] einen namenlosen mobilen *Arduino Roboter* vor, welcher zur Überwachung von Außenanlagen eingesetzt wird. Hierzu verfügt der Roboter über eine Kamera, welche um 360° gedreht werden kann. Der Roboter soll mithilfe seiner Infrarotkamera besonders im Dunkeln Gebäude oder Areale vor Einbrüchen schützen, bzw. diese melden. Ein Radio-Frequency Identification (RFID) Lesegerät kann autorisierte Personen erkennen. Ein Arduino Uno Board übernimmt die Steuerung des Roboters und der Kamera. Das Kameramodul überträgt ihre Daten mittels eines GSM Mobilfunkmoduls, wobei die Anzahl von Objekterkennungen aufgrund der geringen Bandbreite im GSM limitiert wird.

Yaseen Ismael und Hedley von der Universität Newcastle stellen die Entwicklung und Evaluation eines namenlosen *omnidirektionalen Roboters* in [93] vor. Der Roboter wird von einem Arduino Board [O21] gesteuert, welches die Algorithmen wie die Kinematik für den omnidirektionalen Antrieb berechnet, die Daten der Ultraschallsensoren auswertet, die Pfadplanung berechnet, sowie Daten des Roboters zum Analysieren an einen PC sendet. Technische Details zur softwareseitigen Umsetzung liefert die Veröffentlichung nicht. Der Roboter wird in der Forschung eingesetzt.

3.2.4 Analyse der Anwendungsgebiete

Zur Analyse der Anwendungsgebiete bedarf es keiner umfangreichen Literaturrecherche. Nachfolgend werden in Tabelle 3.2 drei Quellen und deren Nennungen von Anwendungsgebieten verglichen. Hierbei wird vordergründig die Definition des IEEE [O22] betrachtet und mittels zweier bereits vorgestellter Quellen validiert. Es zeigt sich, dass sich zwar die Namensgebung und der Detailgrad der einzelnen Kategorien der Anwendungsgebiete unterscheiden, allgemein birgt dieses jedoch wenig Diskussionspotenzial. Letztendlich wird in [59, S. 1006 ff.] beispielsweise nochmal zwischen *Transport* und *selbstfahrenden Autos* unterschieden, während diese beiden Begriffe in [84] in *Transport* zusammengefasst sind. Ebenso könnte die *Telepräsenz* zum Bereich *persönliche Dienstleistung* gezählt werden. Der Bereich *Medizin* fehlt hingegen in [84], *Bildung/ Lehre* wird in [59] nicht aufgeführt. Der Anwendungsbereich *Militär* wird ausschließlich in [O22] aufgelistet.

Referenz		**Anwendungsgebiet**											
Titel/ Bezeichnung [Quelle]	Jahr	Industrie/ Landwirtschaft	Transport/ Logistik	Selbstfahrende/ Autonome Fahrzeuge	Medizin	Gefährliche Umgebung/ Katastrophenhilfe	Erkundung	Persönliche Dienstleistung	Unterhaltung	Bewegungshilfe für Menschen	Bildung/ Forschung	Militär	Telepräsenz
IEEE - Types of Robots [O22]	2020	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Artificial Intelligence [59]	2009	✓	✓	✓	✓	✓	✓	✓	✓	✓			
Chal. of Mobile Robots [84]	2020		✓			✓	✓	✓			✓		

✓: Anwendungsgebiet aufgelistet

Tabelle 3.2: Anwendungsgebiete mobiler Roboter in der Literatur

3.2.5 Analyse der Fähigkeiten

Nachfolgend werden die *Fähigkeiten* von mobilen Robotern in den drei Schritten analysiert (siehe Abbildung 3.2). Als *Fähigkeit* eines mobilen Roboters wird hierbei die Eignung zum Erfüllen jener Fähigkeit bezeichnet. Ein mobiler Roboter hat beispielsweise die *Fähigkeit*, zu navigieren oder ist dazu fähig, sich selbst heilen zu können.

Zunächst werden Suchbegriffe, welche aus den drei Referenzarten und der Umfrage gewonnen wurden, verglichen und ausgewertet. Hierzu werden *Fähigkeiten* (siehe Tabelle 3.3) notiert, welche in den jeweiligen Veröffentlichungen beschrieben oder erläutert werden. Die Tabelle zeigt, welche Begriffe in den ausgewählten Veröffentlichungen häufig und weniger häufig benutzt werden.

Tabelle 3.3 zeigt, dass es einen differenzierten Blick auf die einzelnen Begriffe geben muss, da die subjektiv ausgewählten Referenzen keine belastbaren Aussagen über die Relevanz der *Fähigkeiten* zulassen. Unstrittig dabei sind die Begriffe *Navigation* und *Autonomie*, da diese durchweg in nahezu allen Quellen adressiert werden. Die *Selbstheilung* hingegen wird zwar in den Übersichtarbeiten vorgestellt und in einem Buch erläutert, die betrachteten Referenzsysteme setzen diese jedoch nicht um.

Der nächste Schritt in der Analyse der *Fähigkeiten* mobiler Roboter erfolgt deshalb durch die systematische Anwendung der bereits definierten Suchbegriffe mit der digitalen Bibliothek IEEE Xplore. Die Syntax und Suchmatrix der Suchanfrage in IEEE Xplore kann in Anhang A, Tabelle A2 nachgeschlagen werden.

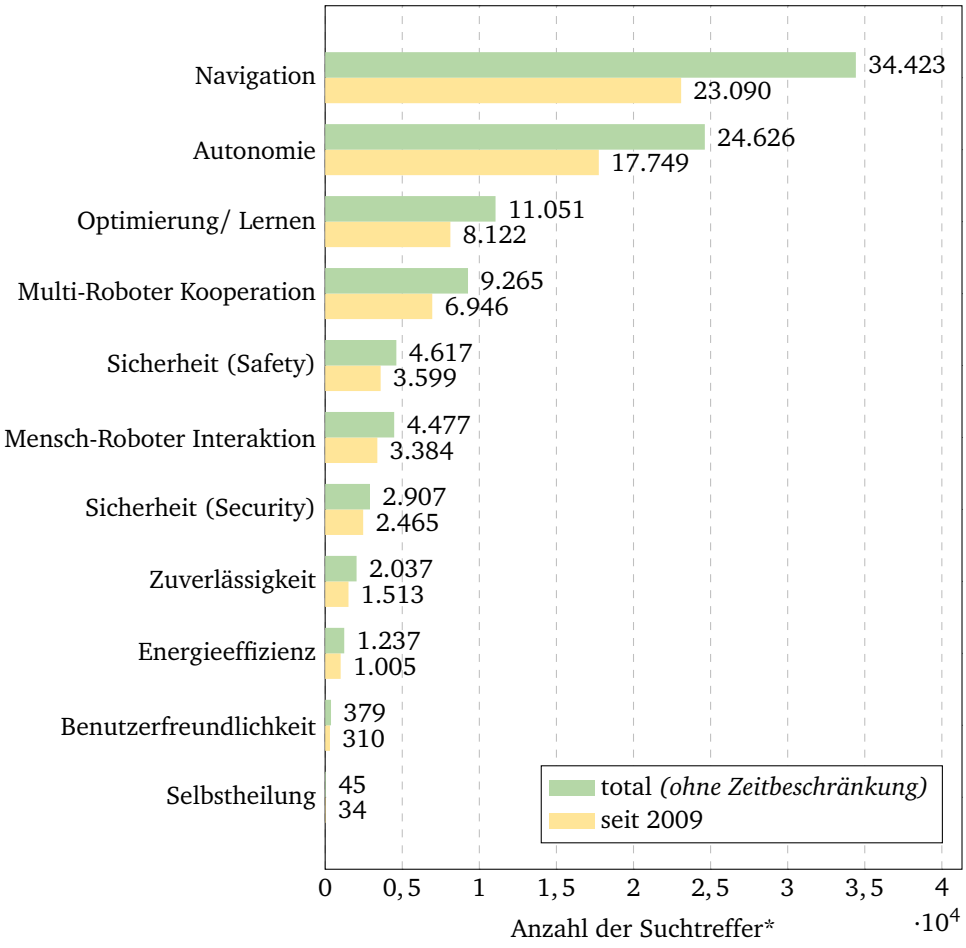
Einen Überblick über die systematische Literaturrecherche der *Fähigkeiten* wird in Abbildung 3.5 dargestellt. Hier sind die Anzahl der Suchtreffer ohne Zeitangabe, sowie im Zeitraum seit 2009 abgebildet. Analog zu Tabelle 3.3 kann festgehalten werden, dass die klassischen *Fähigkeiten* mobiler Roboter, wie die *Navigation* besonders viele Suchtreffer liefern und auch in den letzten zehn Jahren immer noch sehr weit verbreitete Forschungsthemen sind. Die *Benutzerfreundlichkeit* hingegen hat kaum Relevanz in wissenschaftlichen Veröffentlichungen, besonders im Zeitraum vor 2009.

Um den Trend der Suchbegriffe genauer zu analysieren, wird nachfolgend und im letzten Schritt der Analyse, jeder Suchbegriff einzeln betrachtet, erläutert und analysiert. Die Suche umfasst jeweils die Jahre von 2009 bis 2019, sodass ein guter Eindruck über die Entwicklung und Relevanz der einzelnen Themen gezeigt werden kann. Tabelle 3.3 und Abbildung 3.5 werden bei der Einordnung einzelner *Fähigkeiten* und dem Vergleich miteinander herangezogen.

Referenz		Fähigkeit										
Typ	Titel/ Bezeichnung [Quelle]	Navigation	Autonomie	Optimierung/ Lernen	Multi-Roboter Kooperation	Sicherheit (Safety)	Mensch-Roboter Interaktion	Sicherheit (Security)	Zuverlässigkeit	Energieeffizienz	Benutzerfreundlichkeit	Selbstheilung
Bücher aus 3.2.1	Artificial Intelligence [59]	✓	✓	✓	✓		✓					
	Probabilistic Robotics [60]	✓	✓	✓								
	Springer Handbook of Robotics [61]	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
	Intro. to Auton. Mobile Robots [75]	✓	✓		✓		✓			✓		
	Robotics, Vision and Control [62]	✓	✓		✓		✓		✓		✓	
	Embedded Robotics [76]	✓	✓	✓	✓		✓			✓		
Übersichtsarbeiten aus 3.2.2	Modular Reconfigurable Robots [77]			✓					✓	✓		✓
	Challenges of Science Robotics [78]	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
	Multi-Robot Taxonomy [79]	✓	✓	✓	✓	✓		✓	✓	✓		
	Challenges for Mars Expl. [80]	✓	✓		✓		✓		✓	✓		✓
	Robot Teleop, Taxonomy [81]	✓	✓	✓	✓		✓		✓		✓	
	Review of Mobile Robots [82]	✓	✓	✓	✓	✓	✓			✓		
	ANR Requirements [83]	✓	✓	✓	✓		✓		✓	✓		
	Challenges of Mobile Robots [84]	✓	✓	✓						✓		
Referenzsysteme aus 3.2.3	Spot [O13]	✓	🕒	🕒	🕒	✓	✓				✓	
	Khepera IV [85]	✓	✓	✓	✓					✓		
	TurtleBot3 [O14]	✓	✓		✓		✓					
	AMiRo [86]	✓	✓		✓						✓	
	WolfBot [87]	✓	✓		✓					✓		
	e-puck2 [88]	✓	✓		✓		✓				✓	
	OmniMan [89]	✓					✓			✓	✓	
	ArEduBot [90]	✓	✓								✓	
	Savvy [91]	✓	✓		✓		✓					
	Arduino Robot [92]	✓	✓				✓					
	Omnidirectional Mobile Robot [93]	✓	✓		🕒		✓			✓		

✓: Fähigkeit adressiert/ beschrieben
🕒: Als zukünftige Fähigkeit adressiert/ beschrieben

Tabelle 3.3: *Fähigkeiten* mobiler Roboter in der Literatur



*in IEEE Xplore am 07.08.2020

Abbildung 3.5: Übersicht der Anzahl an Suchtreffern zu *Fähigkeiten*

Navigation

Die Navigation ist der Suchbegriff mit den meisten Suchtreffern (ca. 34.400, siehe Abbildung 3.5) im Zusammenhang mit der mobilen Robotik. Der Begriff Navigation, also die Fähigkeit eines mobilen Roboters, in unbekannter Umgebung selbstständig zu manövrieren, umfasst

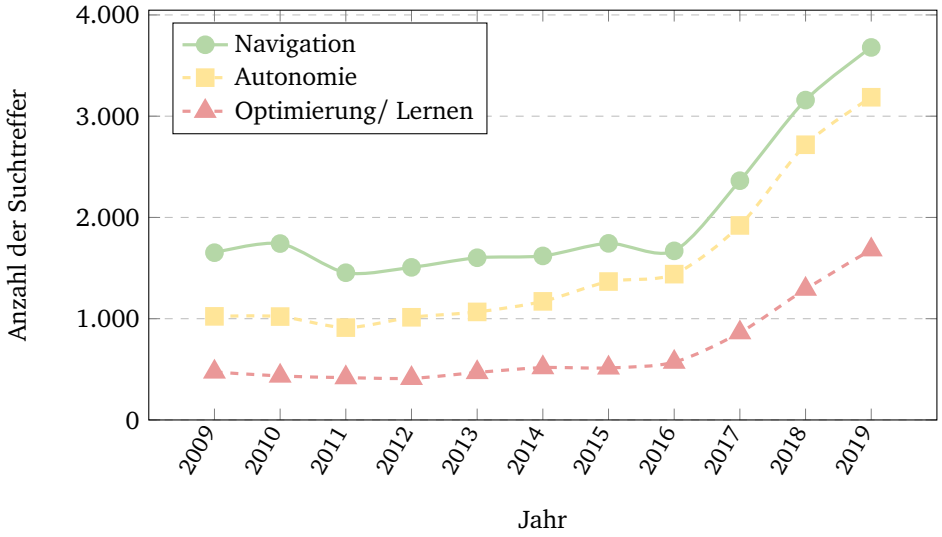


Abbildung 3.6: Verlauf Suchtreffer für *Navigation*, *Autonomie* und *Optimierung/ Lernen*

hierbei auch die Kartografie[12], die Kollisionsvermeidung und Pfadplanung. Abbildung 3.6 zeigt eine hohe Anzahl an Suchtreffern über die Jahre 2009 bis 2016 und einen starken Anstieg seit 2017. Dieser Anstieg ist nicht auf die Popularität der mobilen Robotik in wissenschaftlichen Veröffentlichen zurückzuführen, da hier der Anstieg nahezu linear erfolgt (siehe Abbildung 1.1). Vielmehr zeigt der Anstieg die hohe Bedeutung der Navigation in der mobilen Robotik und die Fortschritte in diesem Bereich, insbesondere seit 2016. Einen Überblick über den Stand der Technik und über die Verfahren und Algorithmen der Navigation und der Kartografie zeigen Paden, Čáp, Yong, Yershov und Frazzoli am Beispiel von selbstfahrenden Autos [96]. Es werden sowohl der Ablauf der autonomen Navigation, von der Routenplanung bis zur Regelung des Antriebsstrangs, als auch die aktuellen Limitierungen in der Genauigkeit erläutert.

Autonomie

Als Autonomie wird in der mobilen Robotik die Fähigkeit bezeichnet, selbstständig zu agieren, d. h. eigene Entscheidungen zu treffen, beispielsweise um eine Aufgabe auszuführen. Analog zur Navigation ist die Autonomie mobiler Roboter in der Wissenschaft bereits über einen langen Zeitraum auf der Agenda, seit 2016 ist jedoch auch hier die Anzahl an Veröffentlichungen der AMR nochmals angestiegen (siehe Abbildung 3.6). Der

[12] engl. Mapping

Fortschritt in der Forschung der Autonomie kann gut mit der Entwicklung der Mars Rover dargestellt werden [97]. Sojourner landete 1997 und war der erste mobile Roboter der autonom auf einem fremden Planeten agierte. Mit jedem Nachfolger (Spirit und Opportunity 2004, Curiosity 2012 und Perseverance 2021) wurde der Grad der Autonomie erhöht, aktuell wird der Rover Perseverance durch den autonomen „Mars Helicopter“ erweitert [98]. Dieser Grad der Autonomie kann gemessen werden, beispielsweise findet hier oft der SAE J3016 Standard aus der Automobilindustrie Anwendung [O23]. SAE J3016 beschreibt sechs Kategorien von SAE Level 0 (keine Autonomie) bis SAE Level 5 (vollständiges autonomes Fahren). Auch in der Robotik gibt es eigenständige Klassifizierungen der Autonomie, welche beispielsweise in [99] oder [100] verglichen werden. Hierbei schlagen auch Beer, Fisk und Rogers [99] eine 10-stufige Einteilung der Autonomie für die Robotik vor.

Optimierung/ Lernen

Als Selbstoptimierung bezeichnet Münch [36] die Fähigkeit eines technischen Systems, selbstständig die eigenen Ziele anzupassen. Die Ziele werden auf die aktuelle Betriebssituation abgestimmt und die Systemkonfiguration bezüglich dieser Ziele optimiert [36]. Die Optimierung als solche ist hierbei die Anpassung an die entsprechende Betriebssituation oder Umgebung. Das Erlernen von neuen Fähigkeiten geht über die Selbstoptimierung hinaus, wird jedoch hier der Optimierung zugeordnet. In [101] wird gezeigt, wie ein Roboter die Handbewegung eines Menschen imitiert und erlernt. Das Forschungsthema rund um die Optimierung ist vor 2009 kaum, aber seitdem oft in wissenschaftlichen Veröffentlichungen vorgekommen (siehe Abbildung 3.5). Die Entwicklung in den letzten 10 Jahren (siehe Abbildung 3.6) ist dagegen relativ konstant hoch, der nahezu lineare Anstieg ab 2016 kann auf die erhöhte Anzahl wissenschaftlicher Veröffentlichungen in der mobilen Robotik zurückgeführt werden. Wenngleich das Lernen und die Optimierung hier zusammengefasst werden, ist das Optimieren bestehender Funktionen und Algorithmen einfacher, als das Erlernen gänzlich neuer Fähigkeiten. Letzteres gilt als eine Notwendigkeit, um eine echte, vollständige und dauerhafte Autonomie von technischen Systemen zu erreichen. In [102] wird u. a. gezeigt, wie mobile Roboter die Navigation in einer bekannten Umgebung verbessern, indem die Route erlernt wird. Interessant ist weiterhin, dass das Optimieren und das Lernen zwar in den Übersichtsarbeiten und Büchern adressiert, in den auswählten Referenzsystem jedoch kaum umgesetzt wird (siehe Tabelle 3.3).

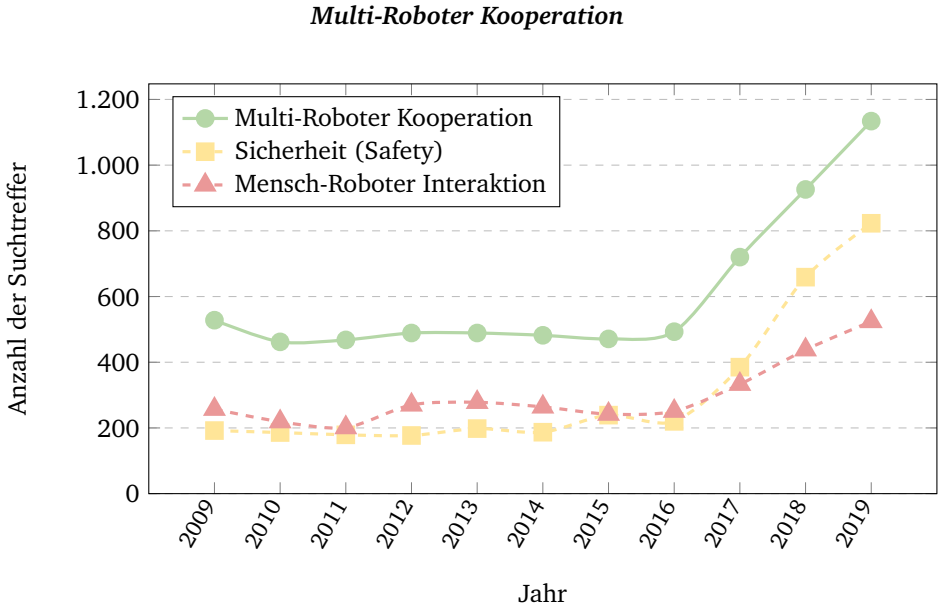


Abbildung 3.7: Verlauf Suchtreffer *Multi-Roboter Interaktion*, *Sicherheit (Safety)* und *Mensch-Roboter Interaktion*

Die Multi-Roboter Kooperation umfasst sowohl die Kooperation zweier Roboter, als auch die Interaktion ganzer Schwärme und die sogenannten ANR. Insbesondere die Schwarm-Robotik kann als Trend der letzten Jahre bezeichnet werden, da sich die Suchtreffer hier seit 2016 mehr als verdreifacht haben (siehe Abbildung 3.7). In [103] werden z. B. heterogene Roboterschwärme gezeigt. Eine große Herausforderung bei der Entwicklung von Multi-Roboter Systemen ist die Implementierung der Fähigkeiten, der Verteilung der Aufgabe(n) untereinander, sowie das Machine to Machine (M2M)-Interface, bzw. die Kommunikation zwischen den einzelnen Robotern.

Sicherheit (Safety)

Mit dem deutschen Begriff *Sicherheit* werden zwei englische Begriffe (*Safety* und *Security*) zusammengefasst, welche getrennt betrachtet werden müssen. *Safety* beschreibt die Sicherheit aus dem technischen System selbst heraus. Roboter sind als sicherheitskritische Systeme einzustufen, da sie eine potenzielle Gefahr für die Nutzer, die Umwelt und sich selbst darstellen können, wie Guiochet, Machin und Waeselynck in [104] erläutern. Ein sicherer mobiler Roboter besitzt die Fähigkeit, niemand anderen, anderes oder sich selbst zu schaden. Hierzu werden Software-seitig Algorithmen implementiert und oder Hardware-

seitig besonderes Material genutzt, welches zum Schutz des Systems und der Umgebung dient. Für die Einhaltung der Sicherheit bedarf es u. a. umfangreicher Tests [105]. Vasic und Billard zeigen Risikofaktoren bei der Interaktion von Robotern und Menschen [106]. Die hier betrachtete Sicherheit (Safety) kann als Trend bezeichnet werden, da hier die Suchtreffer bis 2016 nahezu konstant sind und sich seitdem vervierfacht haben (siehe Abbildung 3.7).

Mensch-Roboter Interaktion

Der Verlauf der Suchtreffer für *Mensch-Roboter Interaktion* liegt in etwa bei 200 Suchtreffern und steigt ab 2016 analog zur Popularität der mobilen Robotik in wissenschaftlichen Veröffentlichungen an. Im Vergleich zur *Multi-Roboter Interaktion* liegen die Suchtreffer in etwa bei der Hälfte, nichtsdestotrotz wird die Interaktion zwischen Robotern und Menschen in fast allen ausgewählten Referenzen adressiert (siehe Tabelle 3.3). Die Fähigkeiten und Herausforderungen bei der Interaktion zwischen Menschen und Robotern liegen hierbei oft in der Kommunikation (HMI). Hier kann beispielsweise auch die Richtung der Kommunikation eine Rolle spielen, da die Mensch-Roboter Interaktion nicht zwingend bidirektional erfolgt. So werden in der Literatur auch die Begriffe Machine to Human (M2H) oder Human to Machine (H2M) verwendet. In [107] werden die Notwendigkeit der Mensch-Roboter Interaktion und die Problemstellung bei der Umsetzung dieser beschrieben. Riek und Member erläutern in [108] Methoden zur Koordination von Bewegungen von Menschen und mobilen Robotern.

Sicherheit (Security)

Im Gegensatz zu *Safety* bezieht sich die Sicherheit in Bezug auf *Security* nicht auf die Sicherheit aus dem System heraus, sondern auf die Sicherheit vor Angriffen auf das System von außen. Die Relevanz in wissenschaftlichen Veröffentlichungen hat dabei bis 2017 stets zugenommen (siehe Abbildung 3.8), wenngleich die Suchtreffer im Vergleich zu Navigation oder Autonomie nur einen kleinen Teil ausmachen (siehe Abbildung 3.5). Auch die gewählten Referenzsysteme adressieren dieses Thema nicht, in den Übersichtsarbeiten und den Büchern ist das Thema ebenfalls wenig präsent (siehe Tabelle 3.3). Zudem sinkt die Anzahl an Suchtreffern in den Jahren 2018 und 2019 wieder. Nichtsdestotrotz wird beispielsweise in [109] erläutert, wie wichtig die Fähigkeit ist, sich gegen Angriffe von außen schützen zu können, insbesondere bei mobilen Robotern. Einerseits sammeln mobile Roboter in der Regel Daten, welche im Sinne des Datenschutzes nicht durch Dritte abgegriffen werden sollten, andererseits sind insbesondere mobile Roboter potenziell gefährlich, indem sie physische Schäden anrichten könnten. So kann beispielsweise ein fremd gesteuertes UAV einen enormen Schaden anrichten, wenn dieses bewusst in Menschenansammlungen gesteuert wird.

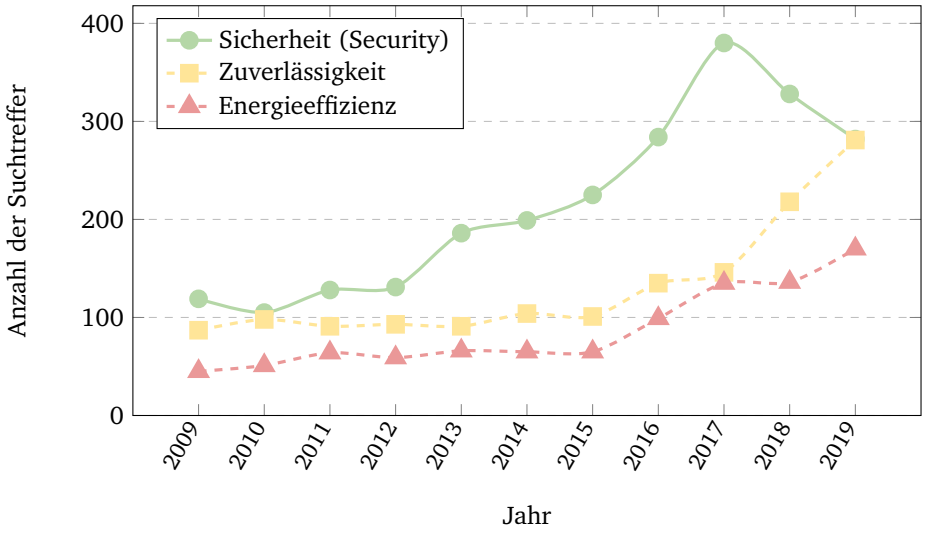


Abbildung 3.8: Verlauf Suchtreffer für *Sicherheit (Security)*, *Zuverlässigkeit* und *Energieeffizienz*

Zuverlässigkeit

Die Zuverlässigkeit kann als Trend in der mobilen Robotik bezeichnet werden. Die Anzahl der Suchtreffer steigen ab 2015, außerdem gab es im Jahr 2018 und 2019 große Sprünge in Richtung 300 Suchtreffer pro Jahr (siehe Abbildung 3.8). Analog zur Sicherheit (Security) ist auch die Zuverlässigkeit bei den ausgewählten Referenzsystemen bedeutungslos (siehe Tabelle 3.3), wenngleich beispielsweise Carlson und Murphy bereits 2003 eine Analyse der Zuverlässigkeit veröffentlicht haben, welche die Notwendigkeit von Verbesserungen in diesem Bereich deutlich macht [110]. Die Fähigkeit mobiler Roboter zuverlässig zu sein, bringt viele positive und wichtige Aspekte mit sich. So trägt eine hohe Zuverlässigkeit zur Akzeptanz mobiler Roboter bei und erhöht die Sicherheit (Safety), welche beispielsweise durch Fehlfunktionen beeinträchtigt werden kann.

Energieeffizienz

Die Fähigkeit, Energie effektiv zu nutzen oder Low Energy (LE)[13]-Komponenten einzusetzen, ist in vielen Anwendungsfällen mobiler Roboter unabdingbar. Die Anzahl der Suchtreffer hierzu sind dabei auf einem relativ konstanten geringen Niveau (siehe Abbildung 3.8), der Anstieg ab 2015 erfolgt analog zum Anstieg der Suchtreffer für mobile Roboter. In

[13] dt. Niedrigenergie

den ausgewählten Referenzen spielt die Energieeffizienz eine vergleichsweise große Rolle, besonders in den Übersichtsarbeiten ist dieses Thema überrepräsentiert (siehe Tabelle 3.3). Das Erhöhen der Energieeffizienz kann in verschiedenen Bereichen erreicht werden. Eine Möglichkeit, die verfügbare Energie effizient zu nutzen, kann durch eine effiziente Nutzung des Antriebs erfolgen, da dieser meist einen großen Anteil am Energieverbrauch mobiler Roboter hat. Künemund, Hess und Röhrig zeigen in [111] Methoden und Algorithmen zur effizienten Pfadplanung.

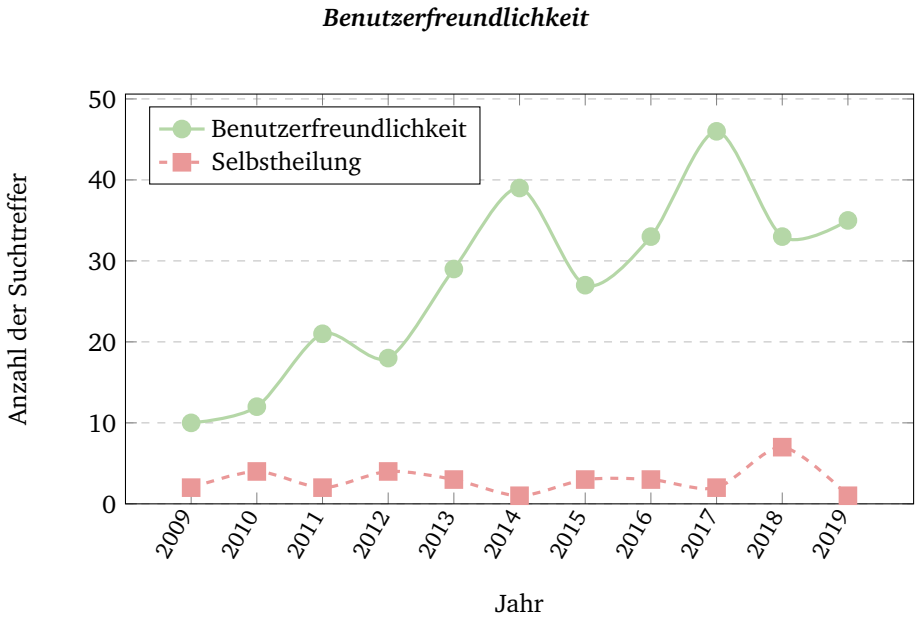


Abbildung 3.9: Verlauf Suchtreffer für *Benutzerfreundlichkeit* und *Selbstheilung*

Die Benutzerfreundlichkeit bei der Nutzung und Entwicklung mobiler Roboter ist in wissenschaftlichen Veröffentlichungen nahezu irrelevant (siehe Abbildung 3.9). Zwar gab es vereinzelt Veröffentlichungen hierzu, die absolute Zahl der Veröffentlichungen ist jedoch mit maximal 50 Suchtreffern pro Jahr gering. In der Praxis spielt die Benutzerfreundlichkeit gerade bei so komplexen Systemen wie Robotern eine große Rolle. Die Notwendigkeit, die Daten aufzuarbeiten und in vereinfachter Form darzustellen und die Interaktion mit Menschen möglichst einfach zu gestalten, spielt besonders in der Interaktion von mobilen Robotern mit älteren und kranken Menschen eine große Rolle [112].

Selbstheilung

Die Selbstheilung oder selbstständige Reparatur mobiler Roboter ist eine Fähigkeit, welche benötigt wird, um einen vollständig autonomen Roboter zu entwickeln. So muss ein mobiler Roboter selbstständig defekte Hardware austauschen können oder diesen Defekt anders ausgleichen können, um wirklich dauerhaft autonom agieren zu können, wie es beispielsweise bei der Nutzung von mobilen Robotern auf dem Mars notwendig ist [80]. Die Relevanz in wissenschaftlichen Veröffentlichungen hierbei ist verschwindend gering (siehe Abbildung 3.9). Dies lässt darauf schließen, dass es bisher nur sehr wenige Lösungen für dieses Forschungsfeld gibt. Dieses Forschungsgebiet scheint innerhalb der mobilen Robotik eine offene Herausforderung zu sein, denn in *Springer Handbook of Robotics* [61] wird dieses Thema durchaus adressiert. Einige der wenigen Veröffentlichungen der *Selbstheilung / Selbst-Reparatur* zeigen den Einsatz von Material, welches nach Beschädigungen wiederhergestellt werden kann [113].

3.2.6 Analyse der technischen Realisierungen

Nachfolgend erfolgt eine Analyse der *technischen Realisierungen*. Als *technische Realisierung* eines mobilen Roboters werden hierbei die Umsetzung, Implementierung, Realisierung oder verwendete Technologie bezeichnet. Die Abgrenzung zu den *Fähigkeiten* eines mobilen Roboters ist dabei nicht immer einfach. Prinzipiell soll die *technische Realisierung* der *Fähigkeit* dienen, bzw. diese umsetzen. Die *Systemüberwachung* wird beispielsweise benötigt, um die Zuverlässigkeit des mobilen Roboters zu gewährleisten, wenngleich die *Systemüberwachung* selbst auch eine *Fähigkeit* sein könnte. Gleiches trifft beispielsweise auch auf die *Echtzeitfähigkeit* zu, welche offensichtlich als *Fähigkeit* bezeichnet werden kann. Hier wird die *Echtzeitfähigkeit* jedoch als *technische Realisierung* zur Erfüllung der Sicherheit (Safety) zugeordnet.

Die verwendete Technologie ändert sich bei technischen Geräten häufig, zumal die mobile Robotik ein sehr weit verbreitetes Forschungsfeld ist und deswegen zügig neue Forschungsergebnisse und Erkenntnisse gewonnen werden. Des Weiteren haben *technische Realisierungen* einen großen Anteil an Anforderungen an eine generische Systemarchitektur. Die Analyse wird analog zur Analyse der *Fähigkeiten* (siehe Kapitel 3.2.5) durchgeführt. Zunächst werden die *technischen Realisierungen* aus den drei Referenzarten verglichen. Tabelle 3.4 stellt dar, welche *technischen Realisierungen* in welcher Referenz erläutert oder beschrieben wurden.

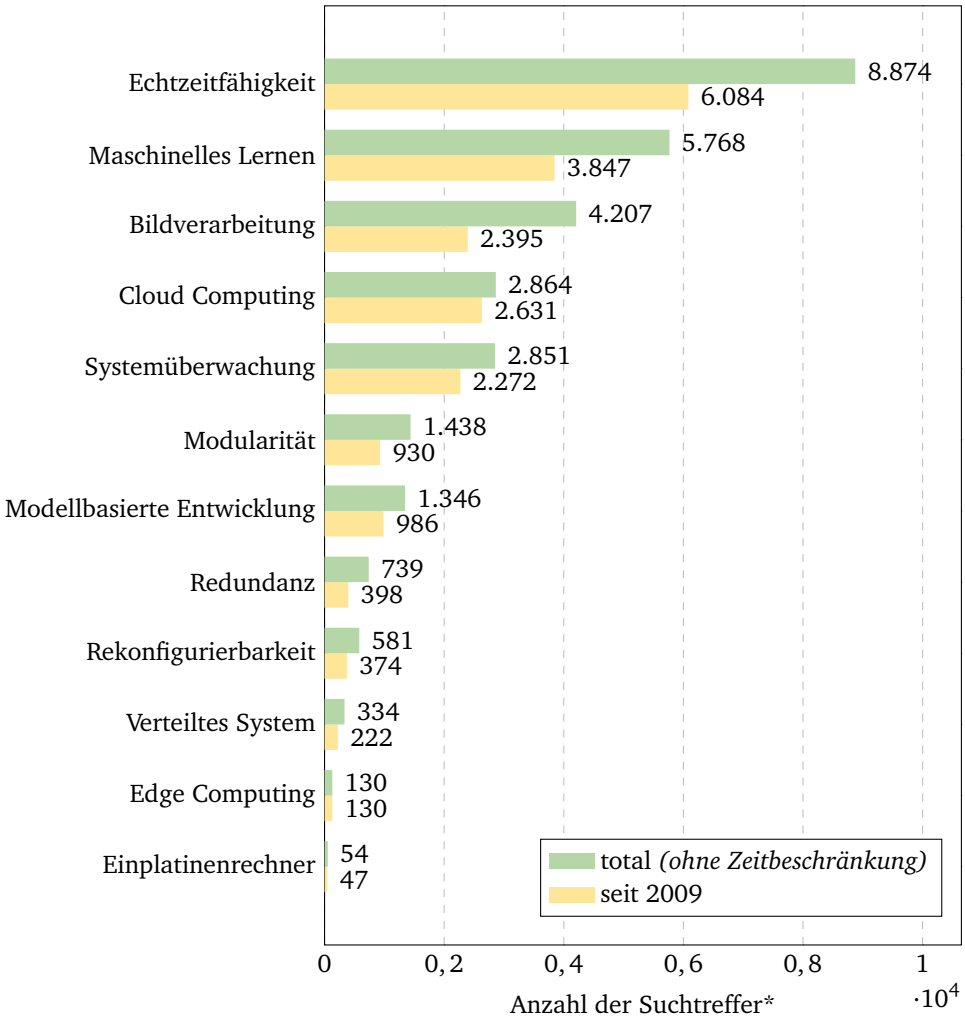
Referenz		Technische Realisierung											
Typ	Titel/ Bezeichnung [Quelle]	Echtzeitfähigkeit	Maschinelles Lernen	Bildverarbeitung	Cloud Computing	Systemüberwachung	Modularität	Modellbasierte Entwicklung	Redundanz	Rekonfigurierbarkeit	Verteiltes System	Edge Computing	Einplatinenrechner
Bücher aus 3.2.1	Artificial Intelligence [59]	✓	✓	✓	✓	✓	✓			✓			
	Probabilistic Robotics [60]	✓	✓	✓									
	Springer Handbook of Robotics [61]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		
	Intro. to Auton. Mobile Robots [75]	✓		✓			✓						
	Robotics, Vision and Control [62]	✓		✓			✓	✓	✓				
	Embedded Robotics [76]	✓		✓			✓				✓	✓	✓
Übersichtsarbeiten aus 3.2.2	Modular Reconfigurable Robots [77]						✓			✓			
	Challenges of Science Robotics [78]	✓	✓	✓	✓	✓	✓		✓	✓	✓		
	Multi-Robot Taxonomy [79]			✓		✓	✓	✓			✓		
	Challenges for Mars Expl. [80]												
	Robot Teleop, Taxonomy [81]				✓	✓							
	Review of Mobile Robots [82]	✓	✓	✓			✓		✓	✓			
	ANR Requirements [83]	✓			✓	✓	✓			✓	✓		
	Challenges of Mobile Robots [84]	✓	✓	✓				✓	✓				
Referenzsysteme aus 3.2.3	Spot [O13]	✓		✓			✓						
	Khepera IV [85]			✓			✓						
	TurtleBot3 [O14]		✓	✓			✓				✓		✓
	AMiRo [86]	✓	✓	✓			✓				✓		
	WolfBot [87]	✓		✓									✓
	e-puck2 [88]	✓		✓		✓	✓				✓		
	OmniMan [89]	✓									✓		
	ArEduBot [90]	✓						✓					✓
	Savvy [91]	✓		✓			✓				✓		✓
	Arduino Robot [92]	✓		✓									
	Omnidirectional Mobile Robot [93]												✓

✓: Technische Realisierung adressiert/ beschrieben

Tabelle 3.4: *Technische Realisierungen* mobiler Roboter in der Literatur

Bei der Betrachtung von Tabelle 3.4 wird deutlich, dass auch hier eine differenziertere Analyse der einzelnen Themen vonnöten ist. Während die *Echtzeitfähigkeit* allgegenwärtig ist, werden Forschungsgebiete wie die *Systemüberwachung* oder *Rekonfigurierbarkeit* nur in wenigen Fällen genannt. Nachfolgend wird jeder Suchbegriff in einer systematischen Literaturrecherche abgesichert. Zunächst als Übersicht (siehe Abbildung 3.10) und später einzeln in der Analyse der zeitlichen Entwicklung. Die Syntax und die Suchmatrix der Suchanfrage in IEEE Xplore kann in Anhang A, Tabelle A3 nachgeschlagen werden.

Der Überblick über die systematische Literaturrecherche der *technischen Realisierungen* in Abbildung 3.10 zeigt die Anzahl der Suchtreffer ohne Zeitangabe, sowie die Anzahl der Suchtreffer seit 2009. Die heute noch allgegenwärtigen Forschungsgebiete *Echtzeitfähigkeit* und *Bildverarbeitung* waren bereits vor 2009 stark im Fokus. Das *Cloud Computing* hingegen kam vor 2019 praktisch nicht vor und ist heute trotzdem unter den am meisten erzielten Suchtreffern.



*in IEEE Xplore am 07.08.2020

Abbildung 3.10: Übersicht der Anzahl an Suchtreffern zu *technische Realisierungen*

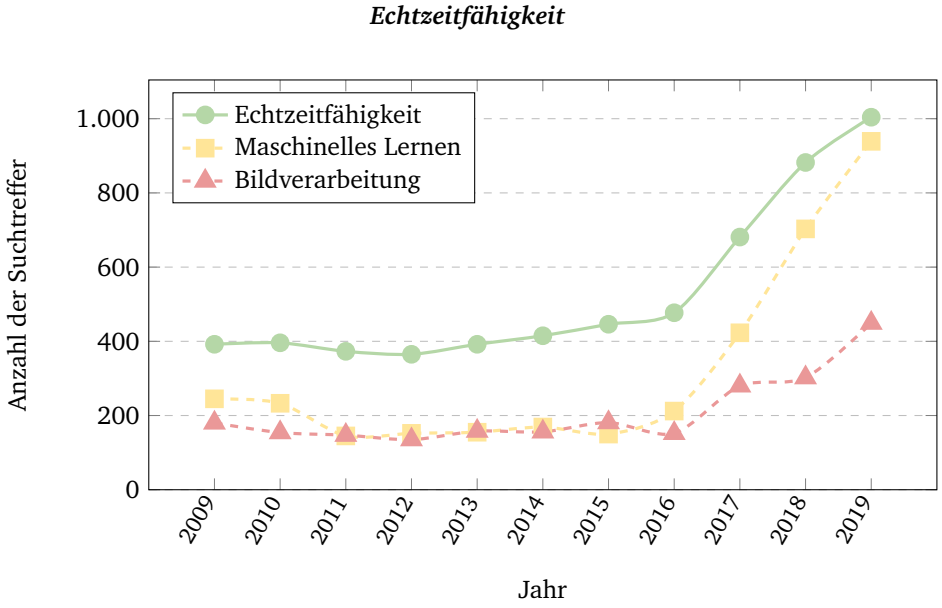


Abbildung 3.11: Verlauf Suchtreffer für *Echtzeitfähigkeit*, *Maschinelles Lernen* und *Bildverarbeitung*

Wie bereits erwähnt ist die Einhaltung der Echtzeitfähigkeit eine Anforderung, welche in der mobilen Robotik unbedingt erforderlich ist. Durch das Einhalten einer definierten Zeitbedingung, können die Sicherheit (Safety) und auch die Zuverlässigkeit des Systems erhöht werden [114]. So kann das unmittelbare Ausführen einer Aktion, beispielsweise das Stoppen der Motoren innerhalb einer vorher definierten Zeit nach Erkennung eines Hindernisses, Unfälle, Defekte oder gefährliche Situationen vermeiden. Die Betrachtung der Entwicklung der Suchtreffer seit 2009 (siehe Abbildung 3.11) zeigt ein Wachstum seit 2006 und einen starken Anstieg seit 2016. Dieser Trend geht einher mit vielen Suchbegriffen in Bezug auf die mobile Robotik, da auch diese selbst an Popularität zugelegt hat (siehe Abbildung 1.1).

Maschinelles Lernen

Die Nutzung oder der Einsatz des maschinellen Lernens wurde in den ausgewählten Quellen vergleichsweise wenig adressiert (siehe Tabelle 3.4). In den Suchtreffern hingegen ist das Thema sehr präsent, mit einem leicht negativem Trend bis 2015, ab 2015 jedoch sehr stark ansteigend (siehe Abbildung 3.11). Maschinelles Lernen kann dazu verwendet werden, große Mengen von Daten auszuwerten, beispielsweise um Muster in diesen zu erkennen

und hieraus zu lernen. Zur Kategorie und Suchmatrix des *maschinellen Lernens* gehören auch *neuronale Netze*, welche auch als Künstliche Neuronale Netze (KNN) und *Deep Learning*[14] bezeichnet werden. Neuronale Netze sind eine von vielen Möglichkeiten, das maschinelle Lernen anzuwenden. Hierbei werden Daten einem Netzwerk an Informationen zugeführt, welche durch diverse Algorithmen, beispielsweise für die Objekterkennung, Ergebnisse in Form von Mustern liefert [115]. Deep Learning ist eine Methode, welche dieses Netzwerk selbstständig erweitert und somit sein Wissen bzw. die Informationsbasis erweitert. Zu jedem Dateneingang werden positive oder negative Gewichtungen berechnet. Je mehr Daten verarbeitet werden, desto besser und genauer wird das Ergebnis, es bildet sich nach und nach ein tieferes und breites Netz aus Informationen. Zunächst müssen neuronale Netze jedoch trainiert werden, indem korrekte Pfade im Netzwerk generiert werden. In der Robotik wird sich diese Technologie beispielsweise bei der Navigation zunutze gemacht, insbesondere jedoch bei der Objekterkennung.

Bildverarbeitung

Die Objekterkennung fällt auch in die Bildverarbeitung, für welche jedoch mehrere Technologien und Methoden verwendet werden können. Entsprechend ist der Verlauf der Suchergebnisse nahezu deckungsgleich zu den Suchergebnissen zu neuronalen Netzen (siehe Abbildung 3.11). Die Bildverarbeitung wird beispielsweise als technische Realisierung dazu eingesetzt, Objekte zu erkennen, deren Positionen für die Navigation benötigt werden. Als Quelle zur Bilderverarbeitung dient Bildmaterial. In der mobilen Robotik werden i. d. R. Videostreams verwendet, welche die zeitliche Abfolge einzelner Bildaufnahmen darstellen. Typisch ist hierbei die Prüfung auf markante Merkmale und eindeutige Bildpunkte, wofür OpenCV [116] eingesetzt werden kann. Die Quelle des Bildmaterials kann dabei durch diverse Sensoren erzeugt werden, von klassischen 2D Kameras bis hin zu 360 Grad LiDAR Kameras. Einen Überblick über aktuelle Hardware, Methoden und Algorithmen erläutert beispielsweise Arnold, Al-Jarrah, Dianati, Fallah, Oxtoby und Mouzakitis in [117].

Cloud Computing

Am Cloud Computing in Kombination mit der mobilen Robotik wird, bis auf wenige Ausnahmen, erst seit 2009 geforscht (siehe Abbildung 3.10). Der Trend seit 2009 ist nahezu in jedem Jahr stark ansteigend (siehe Abbildung 3.12). Die Suchtreffer beinhalten hierbei nicht nur den Trend des Cloud Computing, sondern zudem noch den Ansatz der Integration von Servern in die technische Umsetzung mobiler Roboter. Cloud Computing oder auch die Nutzung von Servern kann dazu genutzt werden, einem technischen System zusätzliche Rechenleistung und Speicherplatz zur Verfügung zu stellen.

[14] dt. Mehrschichtiges Lernen

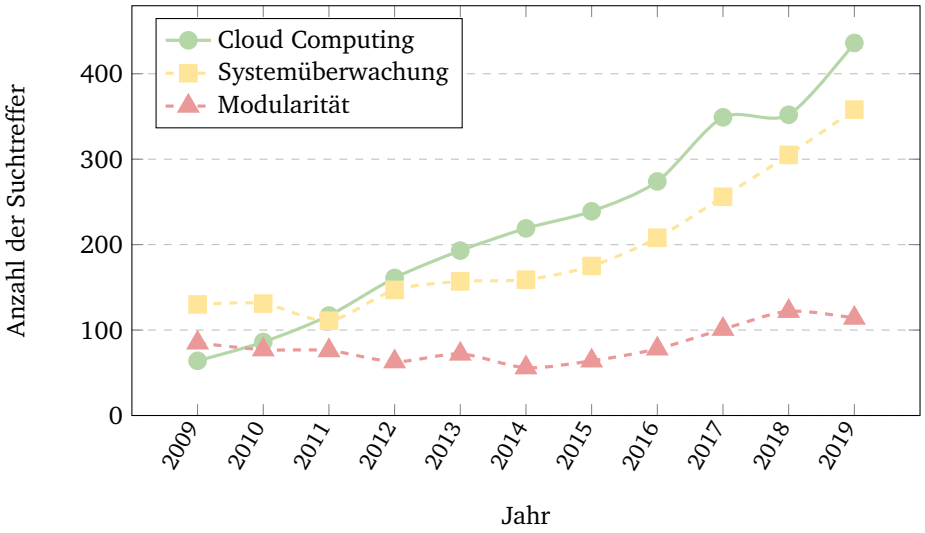


Abbildung 3.12: Verlauf Suchtreffer für *Cloud Computing*, *Systemüberwachung* und *Modularität*

Cloud Computing kann eingesetzt werden, um Algorithmen und Methoden, wie neuronale Netze auszulagern, deren Leistungsanforderungen lokal auf einem mobilen Roboter nicht zur Verfügung gestellt werden können. Hinzu kommt, dass insbesondere Cloud Dienste adaptiv angepasst werden können, falls beispielsweise die Leistungsanforderungen spontan zunehmen. In „Robots with their heads in the clouds“ [118] gibt es einen kurzen Überblick über die Nutzung der Technologie *Cloud Computing* in der Robotik. In [119] wird detaillierter erläutert, wie der aktuelle Stand der Technik ist und was die offenen Fragestellungen im Forschungsgebiet *Cloud Robotics* sind.

Systemüberwachung

Die Überwachung und Überprüfung des Systems bzw. des Systemzustands ist ein breites Forschungsfeld, welches hier als *Systemüberwachung* zusammengefasst wird. Hierzu zählt die *Validierung* des Systems, d. h. das Überprüfen, ob das System dem definierten Zweck dient, sowie die *Verifikation*, d. h. das Überprüfen, ob das System korrekt umgesetzt wird, welches beispielsweise durch regelmäßige Messungen erfolgt. Hierbei kann beispielsweise verifiziert werden, ob zeitliche Abläufe im System eingehalten werden (*Time Verification*). Die Einhaltung der Echtzeit und die Verifikation weiterer Faktoren können dazu dienen, zu validieren, ob der mobile Roboter die Anforderungen der Sicherheit (Safety) erfüllt. Neben der Validierung und Verifikation umfasst die Systemüberwachung auch den Suchbegriff

der *Diagnose* des Systems, sowie die *Selbstwahrnehmung*[15]. Zudem umfasst die Suchmatrix die Forschungsfelder *Systemüberwachung*[16] und *Fehlererkennung*[17], welche ebenfalls die Überwachung und Prüfung des Systems zum Ziel haben. Die Anzahl der Suchtreffer zum Forschungsfeld der Systemüberwachung in der mobilen Robotik ist nahezu linear steigend, wobei dieser Anstieg ab dem Jahr 2015 nochmals stärker ausfällt (siehe Abbildung 3.12). In [120] wird ein Beispiel der Überwachung des Zustands eines mobilen Roboters gezeigt, welches Nutzer*innen und Entwickler*innen mit einer Graphical User Interface (GUI)[18] den aktuellen internen Zustand darstellt. Ein Modell zur Selbstwahrnehmung von Robotern wird in [121] erläutert, welches interne Fehler erkennt und somit zu einer höheren Sicherheit (Safety) und Zuverlässigkeit führt. Die frühzeitige Erkennung von Fehlern, insbesondere im Antrieb eines WMR wird u. a. in [122] gezeigt.

Modularität

Mobile Roboter, welche modular aufgebaut sind, sind weit verbreitet, so auch in den ausgewählten Referenzsystemen (siehe Tabelle 3.4). In den Jahren 2009 bis 2015 sind die Veröffentlichungen zur Modularität in der mobilen Robotik konstant, ab 2016 erfolgt ein leichter Anstieg (siehe Abbildung 3.12), analog zur ansteigenden Popularität mobiler Roboter in wissenschaftlichen Veröffentlichungen. Modularität bedeutet, dass das technische System in Form eines Baukastens aufgebaut ist. Das Gesamtsystem ist in einzelne, kleinere Komponenten unterteilt. Die Unterteilung erfolgt meist nach Dienst oder Funktion und betrifft in jedem Fall die Software, teilweise auch die Hardware. Durch die Modularität ist eine Anpassung und Erweiterung des Systems sehr viel einfacher mögliche, als bei einem Gesamtsystem, welches keine streng definierten Schnittstellen zwischen Hard- und Softwarekomponenten vorsieht. In der mobilen Robotik ist ein modularer Aufbau besonders gefragt, da sich der Einsatzort bzw. die Umgebung oft ändert und ggf. neue Sensoren oder Aktuatoren benötigt werden. Die Umsetzung der Modularität und Erweiterbarkeit des mobilen Roboters AMiRo stellen Herbrechtsmeier, Korthals, Schopping und Rückert u. a. in [3] vor.

[15] engl. Self-awareness
[16] engl. System monitoring
[17] engl. Fault detection
[18] dt. Grafische Benutzeroberfläche

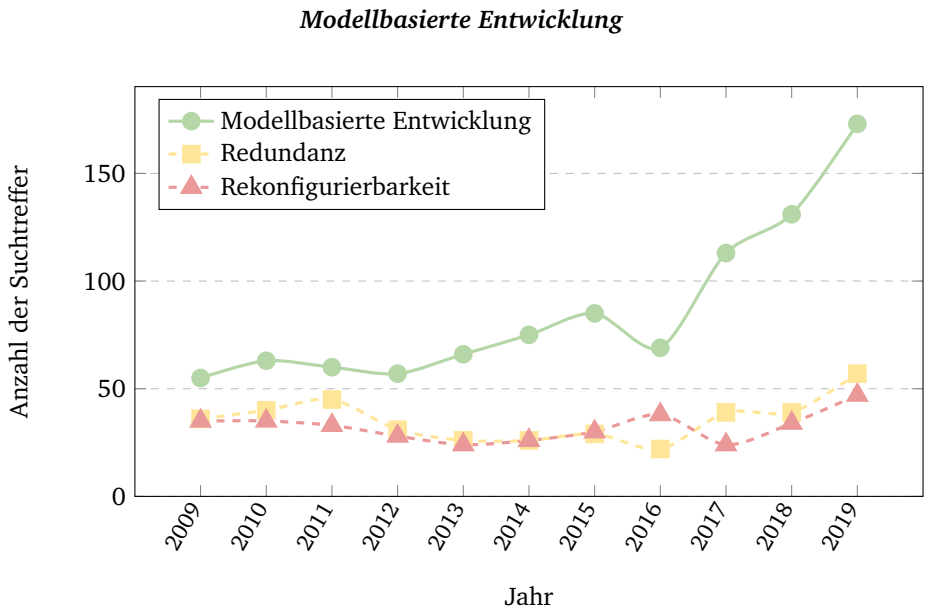


Abbildung 3.13: Verlauf Suchtreffer für *modellbasierte Entwicklung*, *Redundanz* und *Rekonfigurierbarkeit*

Die Entwicklung und Wartung von komplexen technischen Systemen, wie mobiler Roboter, ist aufwendig und eine Herausforderung an die Entwickler. Die Software ist hierbei meist sehr umfangreich. In den *Umsetzungsempfehlungen für das Zukunftsprojekt Industrie 4.0* [123, S. 46 ff.] wird erläutert, dass zur Bewältigung dieser Anforderungen, Methoden des Model-based Software Engineering (MBSE)[19] angewendet werden sollten. Durch den Einsatz dieser Methoden soll die Entwicklung und Dokumentation vereinfacht werden. Dieses führt beispielsweise zu einer höheren Benutzerfreundlichkeit im Sinne des Entwicklers. In „Applying model-based principles on a distributed robotic system application“ hat der Autor bereits eine Umsetzung von Methoden zur MBSE und MDSE gezeigt, vornehmlich mit dem Einsatz von MATLAB Simulink zur Entwicklung eines mobilen Roboters [A3]. Die MBSE ist hierbei der Oberbegriff für die Entwicklung von Software anhand von Modellen. MBSE kann jedoch auch bedeuten, dass der Quelltext händisch auf Grundlage eines Modells erstellt wird. Beim MDSE hingegen wird der Quelltext anhand der Modelle generiert. Die MBSE hat in wissenschaftlichen Veröffentlichungen in den letzten Jahren einen großen Zuwachs erlangt (siehe Abbildung 3.13) und ist mit etwa 1.400 Suchtreffern insgesamt ein vergleichsweise relevantes Schlagwort bei der systematischen Literatursuche (siehe Abbildung 3.10).

[19] dt. Modellbasierte Softwareentwicklung

Redundanz

Eine weitere technische Realisierung, um die Sicherheit (Safety) und Zuverlässigkeit eines mobilen Roboters zu erhöhen, ist der Einsatz von Redundanzen. Redundanzen beschreiben die Verfügbarkeit von Informationen an mehr als einer Stelle in der Software oder die Verfügbarkeit von mehr als einem Bauteil oder einer Komponente dessen Weglassen keine direkten Konsequenzen hat. Bei einem Ausfall, z. B. bei einem Defekt, von einer Komponente oder Funktion, wird diese vom redundanten Pendant übernommen und somit eine Selbstheilung durchgeführt. Redundanzen können weiterhin verwendet werden, um fehlerhafte Sensoren zu erkennen. In [124] wird gezeigt, wie auf redundante Quellen zurückgegriffen wird, um fehlerhafte Datensätze von Motordecodern zu identifizieren. Redundanzen werden in den ausgewählten Referenzsystemen nicht adressiert (siehe Tabelle 3.4), werden bei den Übersichtsarbeiten und den Büchern jedoch genannt. Die Anzahl der Suchtreffer liegt konstant bei unter 50 Suchtreffern pro Jahr (siehe Abbildung 3.13).

Rekonfigurierbarkeit

Die Anzahl der Suchtreffer zur Rekonfigurierbarkeit mobiler Roboter verhält sich analog zu den Suchtreffern der Redundanz (siehe Abbildung 3.13). Die Rekonfigurierbarkeit beschreibt auf der einen Seite die hardwareseitige Rekonfigurierbarkeit, um die Konfiguration an die Umweltbedingungen anzupassen. Laut [77] ist die Entwicklung dieser Systeme eine Herausforderung für Roboter im Allgemeinen und beschreibt mobile Roboter, welche ihre Form anpassen können. Die Anpassung der Hardware kann auch dazu genutzt werden, defekte Teile durch bereits vorhandene Hardware zu ersetzen, welches zur Selbstheilung des Roboters führt [125]. Die Software mobiler Roboter kann jedoch auch rekonfiguriert werden, indem beispielsweise die Abfolge von Funktionen geändert wird oder zeitliche Bedingungen in der Software angepasst werden. Diese softwareseitige Rekonfiguration kann beispielsweise auch zur automatisierten Verteilung von Aufgaben in Multi-Roboter Interaktionen verwendet werden [126].

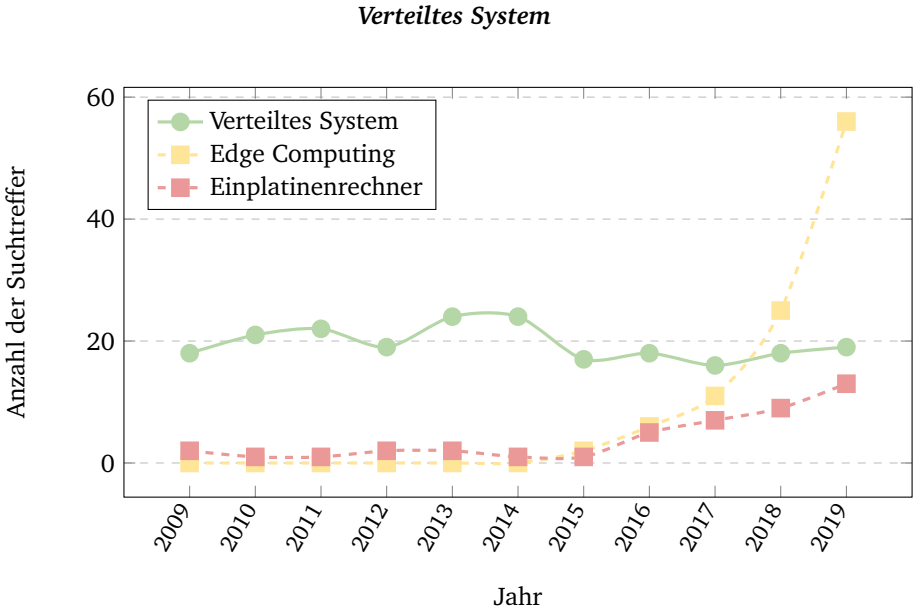


Abbildung 3.14: Verlauf Suchtreffer für *Verteiltes System*, *Edge Computing* und *Einplatinenrechner*

Wie bereits ausführlich in Kapitel 2 „Architekturen und verteilte Systeme“ erläutert, sind verteilte Systeme in der Robotik keine neue Entwicklung. Die Suchergebnisse von 2009 bis 2019 zeigen dazu einen konstanten bis leicht negativen Trend mit je unterdurchschnittlicher Anzahl an Suchtreffern mit ungefähr 20 Suchtreffern pro Jahr (siehe Abbildung 3.14). Bei verteilten Systemen werden entweder Aufgaben und Funktionen auf mehrere Roboter verteilt oder innerhalb eines Roboters auf diverse Hardwarekomponenten. Der Aufbau eines mobilen Roboters als ein verteiltes System geht oftmals mit der Modularität einher. So ist beispielsweise der AMiRo ein Beispiel für einen mobilen Roboter in verteilter Systemarchitektur, welcher in der Basisvariante aus drei Modulen mit je einem MCU besteht [3].

Edge Computing

Edge[20] und Fog[21] Computing *(im Folgenden nur als Edge Computing bezeichnet)* ist eine Technologie, welche erst im Jahr 2015 die ersten Suchtreffer in Kombination mit der mobilen Robotik erzielte (siehe Abbildung 3.14). In 2016 und 2017 gibt es weitere vereinzelte Suchtreffer, in 2018 und 2019 ist ein klarer Trend zu verzeichnen. Edge Computing ist generell ein Trend, welcher aus dem Bereich IoT kommt. Hierbei wird die Verarbeitung von Anwendungen oder Diensten von zentralen Knoten, beispielsweise der Cloud, hin zu den Rändern im Netzwerk verlagert. Dieses reduziert die Datenmenge, welche beispielsweise zur Analyse von Daten an einen Server verschickt werden muss und die Abhängigkeit von der Cloud, indem diese Analyse vor Ort stattfindet und nur die bereits analysierten Informationen an den Server gesendet werden. In [127] wird das Edge Computing u. a. dazu verwendet, den SLAM Algorithmus zwischen den lokalen Robotern und der Cloud in eine sogenannten Edge-Fog Ebene auszulagern. Dieses reduziert die Latenzzeit im Vergleich zur Verarbeitung der Algorithmen in der Cloud. Durch die kürzere Latenz kann der Roboter schneller agieren und so z. B. noch rechtzeitig Hindernissen ausweichen. Somit steigt die Echtzeitfähigkeit, Zuverlässigkeit und Sicherheit (Safety) der Anwendung.

Einplatinenrechner

Die Verwendung von SBC als Recheneinheit(en) für mobile Roboter war vor 2009 nahezu nicht relevant (siehe Abbildung 3.10) und seit 2009 weithin mit einer geringen Anzahl an Suchtreffern vertreten. Die Nutzung von SBC, welche in der Regel eine einfache Handhabung für Entwickler*innen auszeichnet, kann die Benutzerfreundlichkeit aus Sicht der Entwickler*innen steigern. Außerdem sind SBCs im Vergleich zu konventionellen Lösungen wie Notebooks oder Industrie-PCs auf mobilen Robotern vergleichsweise energieeffizient. Nebenbei sind SBC oft preisgünstig und bieten sich deshalb gut für die Entwicklung von Prototypen oder preisgünstigen mobilen Robotern an, wie Krauss in [128] zeigt.

[20] dt. Rand/ Kante
[21] dt. Nebel

3.3 Ableitung einer Taxonomie

Die Definition einer Taxonomie stammt ursprünglich aus der Biologie und beschreibt ein Verfahren, mit dem Objekte klassifiziert werden. In der Wissenschaft werden Taxonomien verwendet, um Einzelfälle in allgemeinere bzw. generische Zusammenhänge zu bringen und somit generalisiert behandelt werden zu können. Bei der Erstellung einer Taxonomie entstehen Strukturen, in die ein Objekt eingeordnet wird oder dessen Elemente strukturell erfasst werden. Laut Krcmar „basiert eine Taxonomie zumeist auf der Analyse von quantitativen empirischem Daten. Basierend auf den Daten werden die Taxonomien durch statistische Cluster erstellt und können dadurch unproblematisch durch andere repliziert werden“ [129, S. 135].

In der vorliegenden Arbeit wird die Erstellung einer Taxonomie verwendet, um einen archetypischen Roboter zu definieren. Der Begriff „archetypisch“ meint hier eine allgemein gültige Definition einer Art oder eines Gerätes, hier die archetypische Definition eines mobilen Roboters. Durch eine Taxonomie wird ein System abstrahiert und verallgemeinert. Ein mobiler Roboter muss somit nicht anhand seiner spezifischen Anwendung definiert werden. Die Spezifikation und die hier gesuchten Anforderungen an eine Systemarchitektur können für ein abstraktes Modell eines mobilen Roboters mit typischen Merkmalen erfolgen, sodass die Systemarchitektur allgemeingültig für jenen archetypischen Roboter ist. Die Taxonomie beantwortet die Frage: *Was sind typische Merkmale eines mobilen Roboters, bzw. was ist ein mobiler Roboter?*

Die für die vorliegende Arbeit benötigte Taxonomie steht aktuell nicht zur Verfügung. Die Suchergebnisse, welche bei der Literaturrecherche nach Taxonomien für Roboter oder mobile Roboter erzielt werden, beschreiben Teilaspekte oder Taxonomien für besondere Anwendungsfälle. So beschreibt [81] eine Taxonomie für die Benutzerfreundlichkeit für das Design von Robotern in Mensch-Roboter Interaktionen. Eine Taxonomie für die Mensch-Roboter Interaktion wird auch in [130] gezeigt. Diverse Publikationen beschreiben Taxonomien für Multi-Roboter Kooperationen [79, 131, 132]. Eine Taxonomie für die Ethik in der Robotik wird in [133] erläutert. Einen Überblick, aus der prinzipiell eine Taxonomie abgeleitet werden könnte, über Entwicklungsumgebungen für autonome Roboter zeigt [134]. Typische Middelwares für Roboter werden in [135] erläutert.

Die Taxonomie wird aus der systematischen Literaturrecherche (Kapitel 3.2) abgeleitet, in der Eigenschaften von mobilen Robotern analysiert wurden, indem geprüft wurde, ob diese Eigenschaften typisch und allgemein gültig für mobile Roboter sind. Die Taxonomie für mobile Roboter wird nachfolgend in verschiedene Kategorien unterteilt. Die vorgeschlagene Taxonomie wird in Abbildung 3.5 dargestellt. In Kapitel 5 wird die Taxonomie mittels einer experimentellen Umsetzung eines mobilen Roboters validiert.

3.3.1 Klassen

Ein mobiler Roboter kann in eine von sechs Klassen eingeordnet werden. Die Definition von fünf dieser Klassen stammt aus dem *Springer Handbook of Robotics* [61]. Zudem wurde in Kapitel 3.1.2 die Klasse der UMVs betrachtet. Diese sechs Klassen werden direkt in die Taxonomie übernommen.

3.3.2 Anwendungsgebiete

Die Analyse der Anwendungsgebiete in Kapitel 3.2.4 hat gezeigt, dass die Anwendungsgebiete bzw. Tätigkeitsfelder mobiler Roboter nicht eindeutig definiert sind. Allerdings liegen die Unterschiede weitestgehend im Detailgrad der Auflistung. Für die Taxonomie eines mobilen Roboters werden die bereits vorgestellten Anwendungsgebiete aus [O22] des IEEE in die Taxonomie übernommen, da diese die umfangreichste Auflistung aus [59, 84, O22] liefert.

3.3.3 Fähigkeiten

Die Fähigkeiten mobiler Roboter wurden in Kapitel 3.2.5 erläutert. Aus der Analyse dieser können Fähigkeiten abgeleitet werden, welche heute jeder Roboter typischerweise unterstützen sollte. Hierbei werden aus den gezeigten Fähigkeiten nicht alle übernommen, sondern nur jene, dessen Relevanz in den ausgewählten Quellen und oder durch häufige Suchtreffer gerechtfertigt wird. So sind die Fähigkeiten *Benutzerfreundlichkeit* und *Selbstheilung* heute keine typischen Fähigkeiten mobiler Roboter. Alle anderen Fähigkeiten sind in der Taxonomie in Tabelle 3.5 zu finden.

3.3.4 Technische Realisierungen

Die technischen Realisierungen mobiler Roboter wurden in Kapitel 3.2.6 untersucht. Hieraus kann abgeleitet werden, welche Technologien, bzw. welche technischen Realisierungen heute beim Entwurf und der Entwicklung eines mobilen Roboters betrachtet und unterstützt werden sollten. Analog zu den Fähigkeiten mobiler Roboter, werden auch hier nur jene Realisierungen in die Taxonomie übernommen, deren Relevanz aus den ausgewählten Quellen oder der Anzahl an Suchtreffern gerechtfertigt ist. Hierbei wurde festgestellt, dass die *Redundanz*, die *Rekonfigurierbarkeit*, die Umsetzung als *verteiltes System*, das *Edge Computing* und der Einsatz von *Einplatinenrechnern* heute nicht in einem typischen mobilen Roboter zu finden sind.

#	**Klasse**
K1	Fahrbar (WMR)
K2	Unterwasser (UUV)
K3	Schwimmend (UMV)
K4	Fliegend (UAV)
K5	Bionisch
K6	Micro/ Nano

#	**Anwendungsgebiet**
T1	Industrie/ Landwirtschaft
T2	Transport/ Logistik
T3	Selbstfahrende/ Autonome Fahrzeuge
T4	Medizin
T5	Gefährliche Umgebung/ Katastrophenhilfe
T6	Erkundung
T7	Persönliche Dienstleistung
T8	Unterhaltung
T9	Bewegungshilfe für Menschen
T10	Bildung/ Forschung
T11	Militär
T12	Telepräsenz

#	**Fähigkeit**
F1	Navigation
F2	Autonomie
F3	Optimierung/ Lernen
F4	Multi-Roboter Kooperation
F5	Sicherheit (Safety)
F6	Mensch-Roboter Interaktion
F7	Sicherheit (Security)
F8	Zuverlässigkeit
F9	Energieeffizienz

#	**Technische Realisierung**
R1	Echtzeitfähigkeit
R2	Maschinelles Lernen
R3	Bildverarbeitung
R4	Cloud Computing
R5	Systemüberwachung
R6	Modularität
R7	Modellbasierte Entwicklung

Tabelle 3.5: Taxonomie für mobile Roboter

3.4 Ableitung der Anforderungen und Herausforderungen an die Systemarchitektur

Das gesamte Kapitel 3 dient zur Definition von Anforderungen und Herausforderungen an die Systemarchitektur. Diese Anforderungen und Herausforderungen setzen sich aus allgemeinen Anforderungen an Systemarchitekturen und den systemspezifischen Anforderungen und Herausforderungen an mobile Roboter zusammen.

Im *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb* [18, S. 109] und *Software requirements* [136, S. 7] werden Anforderungen in Functional Requirement (FR)[22] und Nonfunctional Requirement (NFR)[23] unterschieden. FR beschreiben, analog zu den *Fähigkeiten* mobiler Roboter aus Kapitel 3.2.5, Fähigkeiten, welche der Funktion eines Systems dienen und deshalb stark von dem Einsatz des zu entwerfenden Gerätes abhängig sind. So sind Anforderungen aus der mobilen Robotik wie das Unterstützen der Navigation und der Autonomie eindeutig eine FR. NFR werden auch als technische Anforderungen bezeichnet und beschreiben, analog zu den *technischen Realisierungen* aus Kapitel 3.2.6, technische Anforderungen, die erfüllt werden müssen, damit die Funktionen des Systems hergestellt werden können. Die Zuordnung hierzu ist laut Balzert [18] nicht einfach und oftmals nicht eindeutig möglich. So kann bei der mobilen Robotik beispielsweise die Echtzeitfähigkeit sowohl als FR als auch als NFR bezeichnet werden.

3.4.1 Allgemeine Anforderungen

Als Quelle für allgemein gültige Anforderungen an Systemarchitekturen dient das *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb* [18, S. 109 ff.] von Balzert. Das Buch widmet sich in einem Abschnitt ausführlich den NFR und stellt jede NFR im Detail vor. Die in [18] dargestellten Anforderungen an Architekturen entsprechen weitestgehend den in *Software Architecture in Practice* [6] dargestellten Anforderungen, sodass die NFR von Balzert [18] übernommen und nachfolgend kurz eingeführt werden können.

Mit Abbildung 3.15 zeigt Balzert [18] eine Reihe von allgemeinen Anforderungen an Systeme, welche zu fünf Systemtypen zugeordnet werden können. So betrifft die *Zuverlässigkeit* beispielsweise sowohl *Echtzeitsysteme*, als auch *sicherheitskritische Systeme*, *Informationssysteme* und *Prozesssteuerungssysteme*. Die Anforderungen an die gewählte Anwendungsdomäne sind in der Abbildung 3.15 aufgelistet. Bei der mobilen Robotik handelt es sich sowohl um ein *Echtzeitsystem*, ein *sicherheitskritisches System*, ein *Informationssystem*

[22] dt. funktionale Anforderungen
[23] dt. nichtfunktionale Anforderungen

und ein *Prozesssteuerungssystem*. Kommt das Cloud Computing im System mit einem oder mehreren mobilen Robotern zum Einsatz, so sind auch die Anforderungen aus dem Systemtyp *Websysteme* relevant. Die Abbildung soll einen Überblick über die große Anzahl und Varianz an die Anforderungen geben. Nachfolgend werden Anforderungen von Balzert in sieben Kategorien unterteilt und erläutert, welche nahezu vollständig als allgemeine Anforderungen, welche in Tabelle 3.6 dargestellt werden, übernommen werden.

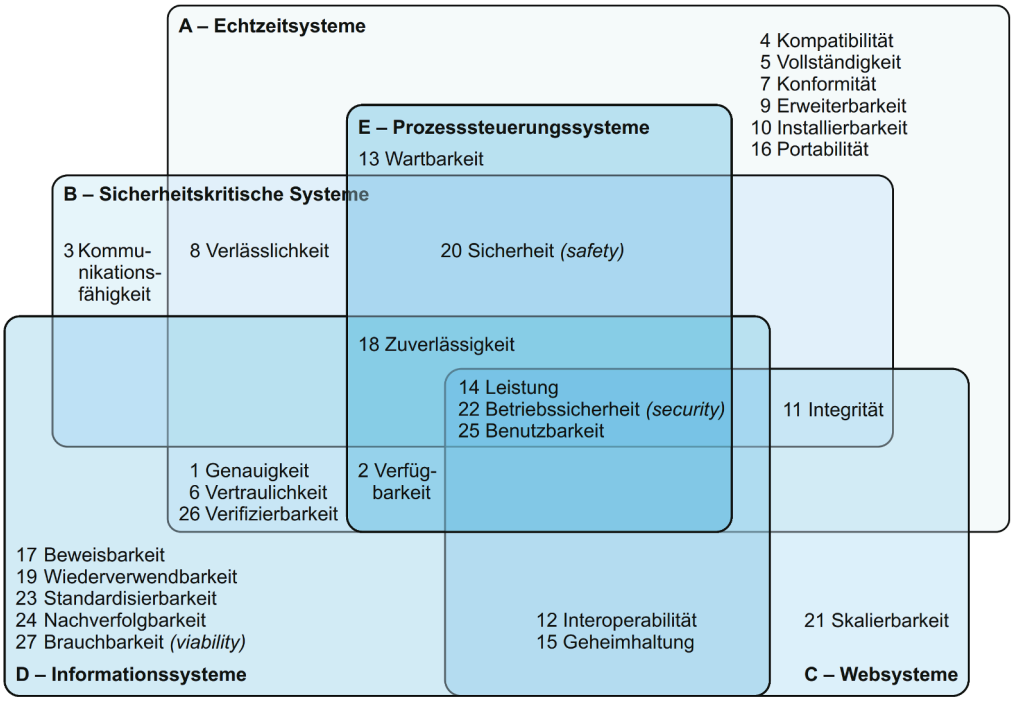


Abbildung 3.15: Zuordnung von nichtfunktionalen Anforderungen zu Systemtypen [18, S. 112]

Wartbarkeit

Die Wartbarkeit wird im Standard *ISO/IEC 9126-1:2001. Software engineering - Product quality* [137] wie folgt definiert.

> „Fähigkeit des Softwareprodukts änderungsfähig zu sein. Änderungen können Korrekturen, Verbesserungen oder Anpassungen der Software an Änderungen der Umgebung, der Anforderungen und der funktionalen Spezifikationen einschließen."

Zu der Wartbarkeit gehören die Teilmerkmale Analysierbarkeit, Änderbarkeit, Stabilität und Testbarkeit. Um diese Anforderungen zu erfüllen, sollte man in der Entwurfsphase Komponenten und Funktionen identifizieren, welche sich später ändern könnten. Weiterhin sollte die Software abstrahiert werden und geeignete generische Schnittstellen definiert werden, welche auch nach einer Änderung noch Bestand haben können. [18, S. 116 ff.]

Weiterentwickelbarkeit

Die Weiterentwickelbarkeit wird in [18, S. 119] wie folgt definiert.

> „Die nichtfunktionale Anforderung Weiterentwickelbarkeit (Evolvability) – besser Nachhaltigkeit genannt – bezeichnet die Fähigkeit eines Softwareprodukts sich langfristig an geänderte Anforderungen und Techniken, die einen Einfluss auf die Architektur und/oder die funktionalen Erweiterungen haben, anzupassen, ohne die architektonische Integrität zu verletzen."

Die Weiterentwickelbarkeit beinhaltet die Teilmerkmale Analysierbarkeit, Änderbarkeit, Testbarkeit, Portabilität, Erweiterbarkeit und Integrität der Architektur. Um die Weiterentwickelbarkeit zu unterstützen, sollten Komponentenmodelle eingesetzt werden und eine Rückverfolgbarkeit der Entwurfsentscheidungen und Abhängigkeiten gegeben sein. [18, S. 119 ff.]

Betriebssicherheit und Funktionssicherheit

Die Betriebssicherheit wird im Standard *ISO/IEC 9126-1:2001. Software engineering - Product quality* [137] folgendermaßen definiert.

> „Fähigkeit des Softwareprodukts, Informationen und Daten so zu schützen, dass nicht autorisierte Personen oder Systeme sie nicht lesen oder verändern können und autorisierten Personen oder Systemen der Zugriff nicht verweigert wird."

Zur Definition der Funktionssicherheit nutzt Balzert die Definition von Birolini in *Zuverlässigkeit von Geräten und Systemen* [138]:

> „Die Funktionssicherheit (Safety) eines Systems ist ein Maß für die Fähigkeit einer Betrachtungseinheit, weder Menschen, Sachen noch die Umwelt zu gefährden."

Zur Betriebssicherheit und Funktionssicherheit zählt Balzert die Vertraulichkeit, Integrität, Zugriffssteuerung und Authentifizierung. Zur Erhöhung der beschriebenen Sicherheit sollten einzelne Strukturen und Komponenten entkoppelt werden, damit einzelne Komponenten nicht direkt manipuliert werden können. [18, S. 121 ff.]

Zuverlässigkeit

Die Definition der Zuverlässigkeit aus dem Standard *ISO/IEC 9126-1:2001. Software engineering - Product quality* [137] lautet:

> „Fähigkeit des Softwareprodukts, ein spezifiziertes Leistungsniveau zu bewahren, wenn es unter festgelegten Bedingungen benutzt wird."

Die Zuverlässigkeit wird auch durch die Teilmerkmale Vollständigkeit, Genauigkeit, Konsistenz, Verfügbarkeit, Integrität, Korrektheit, Reife, Fehlertoleranz, Wiederherstellbarkeit und Einhaltung gesetzlicher, unternehmensinterner und vertraglicher Regelungen definiert. Die Zuverlässigkeit eines Systems kann erhöht werden, indem Subsysteme identifiziert und besonders getestet und auf Qualität geprüft werden, dessen Zuverlässigkeit maßgeblich für das System sind, Nutzereingaben auf Korrektheit überprüft werden, Fehlerquellen im Quelltext durch Ausnahmeregeln abgefangen und Dienste und Programme entwickelt werden, welche bei einem Systemabsturz die Wiederherstellung der Betriebsbereitschaft und Daten ermöglichen. [18, S. 124 ff.].

Leistung und Effizienz

Die Effizienz wird im Standard *ISO/IEC 9126-1:2001. Software engineering - Product quality* [137] so definiert:

> „Fähigkeit des Softwareprodukts, ein angemessenes Leistungsniveau bezogen auf die eingesetzten Ressourcen unter festgelegten Bedingungen bereitzustellen."

Für die Leistung zieht Balzert eine Definition von Mairiza, Zowghi und Nurmuliani [139, S. 315] heran:

> „Anforderungen, die die Fähigkeit eines Softwareprodukts spezifizieren, eine angemessene Leistung relativ zu den benötigten Ressourcen zu erbringen, wobei die volle Funktionalität unter festgelegten Bedingungen bereitgestellt wird."

Zu den Anforderungen an Leistung und Effizienz zählen zudem die Teilmerkmale Zeitverhalten, Verbrauchsverhalten, Antwortzeit, Speicherplatz, Kapazität, Wartezeit, Durchsatz, Rechenzeit, Ausführungsgeschwindigkeit, Übergangsverzögerung, Auslastung, Verbrauchsverhalten, Speicher-Inanspruchnahme, Genauigkeit, Modi, Verzögerung, Fehlerraten, Datenverlust und nebenläufige Transaktionsverarbeitung. Zur Verbesserung der Leistung und Effizienz sollten Algorithmen so ausgewählt werden, dass die geforderte Effizienz erreicht wird. Zudem sollen Subsysteme identifiziert und optimiert werden, deren Leistungsfähigkeit das Gesamtsystem beeinflusst. [18, S. 128 ff.]

Benutzbarkeit

Die Benutzbarkeit wird in „An Investigation into the Notion of Non-Functional Requirements" [139, S. 315] wie folgt definiert.

> „Anforderungen, die die Benutzer-Interaktionen mit dem System und den benötigten Aufwand, um das System zu erlernen, zu benutzen, die Eingaben in das System vorzubereiten und die Ausgabe des Systems zu interpretieren, festlegen."

Hierzu zählen auch die Teilmerkmale Bedienungskomfort, Ausführungsgeschwindigkeit, Ergonomie, Benutzerfreundlichkeit, Einprägsamkeit, Effizienz, Benutzerproduktivität, Bedienungskomfort, Ergonomie, Brauchbarkeit, Erwartungskonformität und Benutzerreaktionszeit. Zur Bewältigung dieser Anforderungen empfiehlt Balzert, alle Benutzerinteraktionen in Subsysteme abzukoppeln, um zu vermeiden, dass Nutzer*innen auf falsche Systemstrukturen zugreifen. [18, S. 130 ff.]

Portabilität

Die Portabilität wird im Standard *ISO/IEC 9126-1:2001. Software engineering - Product quality* [137] folgendermaßen definiert:

> „Fähigkeit des Softwareprodukts, von einer Umgebung in eine andere übertragen zu werden. Umgebung kann organisatorische Umgebung, Hardware- oder Software-Umgebung einschließen."

Balzert fügt hierzu die Teilmerkmale Anpassbarkeit, Installierbarkeit, Koexistenz, Austauschbarkeit und Konformität der Portabilität zu. Auch hier wird empfohlen, Teile, die möglicherweise portiert werden sollen, zu identifizieren und in eigene Subsysteme zu kapseln. [18, S. 132 ff.]

3.4.2 Anforderungen aus der mobilen Robotik

Die Anforderungen aus der mobilen Robotik werden direkt aus der Taxonomie (siehe Tabelle 3.5) in die Gesamtanforderungen (siehe Tabelle 3.6) übernommen. Hierbei werden die Kategorien *Fähigkeiten mobiler Roboter* und *technische Realisierungen mobiler Roboter* übernommen, sodass die Anforderungen an die Systemarchitektur hieraus abgeleitet werden können. Die Systemarchitektur muss beispielsweise die Fähigkeit *Navigation* unterstützen, indem beim Entwurf der Architektur bedacht wird, an welcher Stelle im System, diese Fähigkeit ausgeführt werden kann und welche Ressourcen dazu notwendig sind. Eine technische Realisierung wie das *maschinelle Lernen* muss von der Systemarchitektur insofern unterstützt werden, damit auch hier genug Ressourcen zur Verfügung stehen und die Anbindung dieser Technologie an das Gesamtsystem in der Entwurfsphase betrachtet wird.

3.4.3 Gesamtanforderungen

Die Gesamtanforderungen setzen sich aus den erläuterten allgemeinen NFR aus Kapitel 3.4.1 und den FR und NFR aus Kapitel 3.4.2 zusammen. Hierbei entsprechen einige allgemeine Anforderungen den *Fähigkeiten mobiler Roboter*. Diese werden zusammengefasst und unter der Kategorie „allgemeine Anforderungen" geführt (siehe

Tabelle 3.6). Balzerts [18] „Betriebssicherheit und Funktionssicherheit' beinhaltet die Sicherheit als *Fähigkeit mobiler Roboter* in Bezug auf Security (Betriebssicherheit) und Safety (Funktionssicherheit). Die Zuverlässigkeit ist ebenfalls sowohl eine *Fähigkeit mobiler Roboter*, als auch eine allgemeine Anforderung nach Balzert. Wenngleich die Energieeffizienz bei mobilen Robotern im Vergleich zu vielen anderen Systemen einen größeren Einfluss hat, wird sie durch Balzerts „Leistung und Effizienz" abgedeckt.

Tabelle 3.6 zeigt alle Anforderungen an die Systemarchitektur, welche durch die systematische Literaturrecherche, die Taxonomie für mobile Roboter und die Anforderungen aus *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb* [18], erarbeitet wurden.

Kategorie	**#**	**Anforderung**
Allgemeine Anforderungen	A1	Wartbarkeit
	A2	Weiterentwickelbarkeit
	A3	Sicherheit (Safety)
	A4	Sicherheit (Security)
	A5	Zuverlässigkeit
	A6	Leistung und Effizienz
	A7	Benutzbarkeit
	A8	Portabilität
Fähigkeiten mobiler Roboter	A9	Navigation
	A10	Autonomie
	A11	Optimierung/ Lernen
	A12	Multi-Roboter Kooperation
	A13	Mensch-Roboter Interaktion
Technische Realisierungen mobiler Roboter	A14	Echtzeitfähigkeit
	A15	Maschinelles Lernen
	A16	Bildverarbeitung
	A17	Cloud Computing
	A18	Systemüberwachung
	A19	Modularität
	A20	Modellbasierte Entwicklung

Tabelle 3.6: Anforderungen an die Systemarchitektur mobiler Roboter

Durch die Erstellung der Taxonomie für mobile Roboter (siehe Kapitel 3.3) wurden einige der untersuchten Merkmale mobiler Roboter nicht als typische Fähigkeit oder typische technische Realisierung mobiler Roboter angesehen und somit nicht in die Taxonomie aufgenommen. Trotzdem können diese Merkmale für moderne mobile Roboter relevant sein, wie die Erwähnung in den ausgewählten Quellen (siehe Kapitel 3.2) und die Analyse der einzelnen Merkmale (siehe Kapitel 3.2.5 und 3.2.6) zeigt. Da die Merkmale bisher jedoch vergleichsweise wenige Suchtreffer in wissenschaftlichen Veröffentlichungen erreichen, werden sie als Herausforderungen an die mobile Robotik weiter verfolgt (siehe Tabelle 3.7). Nicht aufgeführt ist hierbei die Herausforderungen der „Benutzerfreundlichkeit“ von mobilen Robotern, da die ebenso eine Anforderung nach Balzert [18] ist (vgl. Benutzbarkeit (H7) in Tabelle 3.6). Selbstverständlich liegt der Fokus beim Entwurf der Systemarchitektur auf den bereits definierten Anforderungen. Die Herausforderungen werden jedoch zur Betrachtung hinzugezogen, insofern sie nicht im Konflikt mit einer der Anforderungen stehen.

Kategorie	**#**	**Herausforderung**
Fähigkeiten mobiler Roboter	H1	Selbstheilung
Technische Realisierungen mobiler Roboter	H2	Redundanz
	H3	Rekonfigurierbarkeit
	H4	Verteiltes System
	H5	Edge Computing
	H6	Einplatinenrechner

Tabelle 3.7: Herausforderungen an die Systemarchitektur mobiler Roboter

4 Konzeptionelles Modell einer Systemarchitektur für mobile Roboter

Dieses Kapitel beschreibt das konzeptionelle Modell einer allgemeingültigen Systemarchitektur für mobile Roboter. Ziel dieses Modells ist die Erfüllung der Anforderungen an eine Systemarchitektur für einen archetypischen mobilen Roboter (siehe Kapitel 3).

Zunächst wird das auf dem OCM basierende Grundkonzept der Systemarchitektur beschrieben (Kapitel 4.1). Anschließend werden die Schlüsselmerkmale der Architektur erläutert und die Auswahl und Zusammensetzung der Struktur des konzeptionellen Modells begründet (Kapitel 4.2). Nachfolgend wird ein Vorschlag für Systemkomponenten und deren Verteilung in der verteilten Struktur gezeigt (Kapitel 4.3). Abschließend folgt eine Erläuterung der Schnittstellen im verteilten System auf verschiedenen Ebenen in Kapitel 4.4.

Das vorgestellte konzeptionelle Modell wurde in mehreren iterativen Schritten erstellt. Anpassungen und Erweiterungen wurden jeweils an einem experimentellen Roboter praxisnah evaluiert und die Rückschlüsse daraus in das Modell eingearbeitet. Die experimentelle Umsetzung wird nachfolgend in Kapitel 5 *„Experimentelle Umsetzung der Systemarchitektur für mobile Roboter“* dargestellt.

Der Autor hat das konzeptionelle Modell bereits in Teilen als „Design of an Operator-Controller Based Distributed Robotic System“ [A8] vorgestellt. Eine erweiterte Version des Artikels entstand im Rahmen eines Journal Beitrags mit „Concepts of a Modular System Architecture for Distributed Robotic Systems“ [A9]. Diese beiden Veröffentlichungen, sowie die Veröffentlichung als Poster [A10] beschreiben das konzeptionelle Modell jeweils am Beispiels eines WMR. „A Recommendation for a Systems Engineering Process and System Architecture for UAS“ [A13] zeigt die Nutzung der Systemarchitektur für ein UAS.

Eine Systemarchitektur beschreibt, wie bereits in Kapitel 2.1.1 erläutert, eine Gesamtsicht auf ein technisches System. Die Beschreibung der Architektur besteht in der Regel aus unterschiedlichen Perspektiven, wie in „IEEE Recommended Practice for Architectural Description for Software-Intensive Systems“ [15, S. 5] vorgeschlagen wird (siehe Abbildung

4.1). Das konzeptionelle Modell wird zunächst in vereinfachter Form dargestellt. Die Darstellung wird in mehreren Schritten erweitert, sodass der Leser den Konzepten und der Struktur der Architektur gut folgen kann. Besonders im nachfolgenden Kapitel 5 werden weitere Perspektiven gezeigt, beispielsweise die technische Realisierung der Kommunikation oder die Verteilung der Softwaremodule innerhalb der Systemarchitektur.

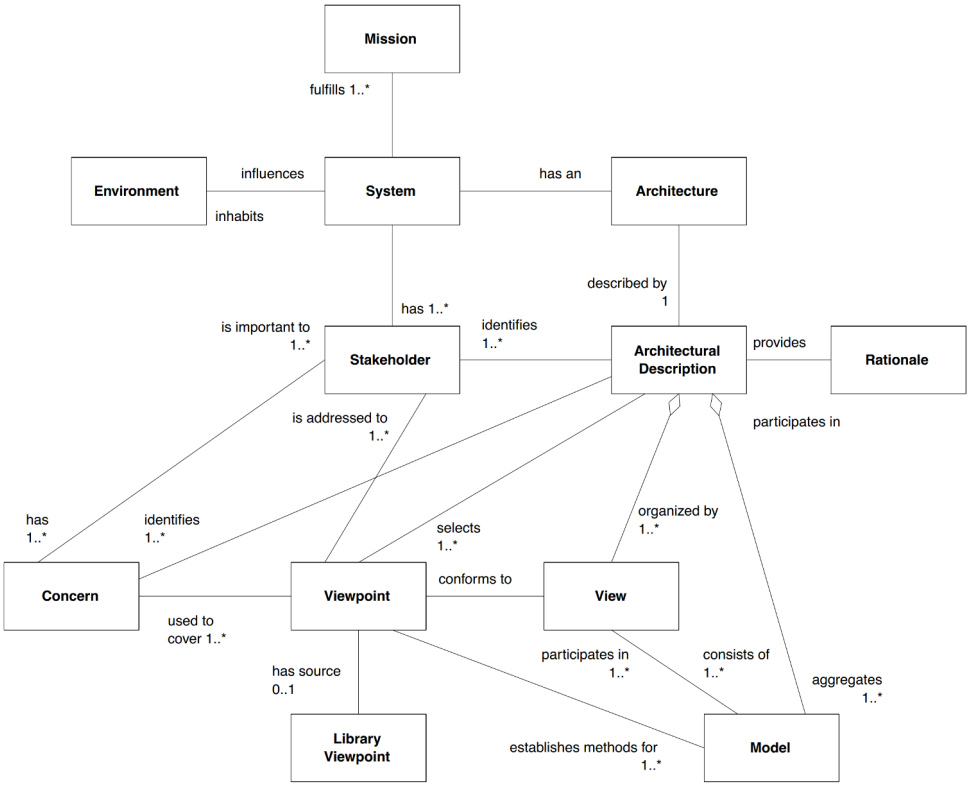


Abbildung 4.1: Abhängigkeiten einer Architektur [15, S. 5]

4.1 Grundkonzept

Das konzeptionelle Modell der Systemarchitektur für mobile Roboter basiert auf dem OCM, welches in Abschnitt 2.4 des Kapitels „Architekturen und verteilte Systeme“ vorgestellt wurde. Die grundlegende Struktur des OCM scheint auch für mobile Roboter geeignet, sodass dieser Ansatz für die Entwicklung einer Systemarchitektur für mobile Roboter

übernommen und umfassend evaluiert wird. Zur Vollständigkeit wird dieses nachfolgend detailliert erläutert, sowie die Erweiterungen am *SFB-614-OCM*[1] kenntlich gemacht. Hierbei sei zu erwähnen, dass es zwar einige Veröffentlichungen zum Konzept des OCM gibt, welche in Kapitel 2.4 vorgestellt wurden, die praktische Umsetzung jedoch nur vereinzelt und insbesondere im Bereich des sogenannten RailCab Schienenfahrzeugs [34] gezeigt wird. Die Anforderungen an die Architektur des RailCab Systems unterscheiden sich maßgeblich von den Anforderungen an eine Architektur für mobile Roboter. Die nachfolgende Architektur ist eine Adaption des *SFB-614-OCM* für mobile Roboter.

Das zugrunde liegende Konzept der Architektur ist die namensgebende Unterteilung des OCM Ansatzes in ***Controller*** und Operatoren in einer Schichtenarchitektur mit strenger Trennung. Die Basis der Systemarchitektur für mobile Roboter bilden die *Controller* und mindestens ein ***reflektorischer Operator*** (siehe Abbildung 4.2). Das konzeptionelle Modell sieht die Umsetzung der Schichtenarchitektur als verteiltes System vor, also der Verteilung des Systems auf mehrere Rechner.

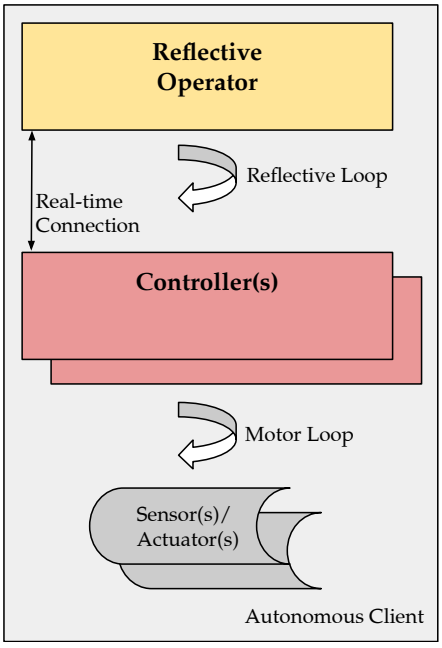


Abbildung 4.2: Lokaler Bereich der Systemarchitektur

[1] Als das *SFB-614-OCM* wird im Folgenden die in Paderborn entwickelte Version [O8] bezeichnet (entspricht Abbildung 2.12 in Kapitel 2.4).

Die unterste Schicht der Architektur, die *Controller*, stehen in direkter Verbindung mit den angebundenen Sensoren und Aktuatoren und bilden zusammen mit diesen den **motorischen Kreis**. Die *Controller* lesen Sensordaten aus und betreiben Aktuatoren. Dieses bedarf **harter Echtzeit**, d. h. der Austausch und die Bearbeitung der Daten erfolgt innerhalb einer fest vorgegebenen Zeit, welche möglichst kurzzuhalten ist. Typischerweise werden mehrere *Controller* eingesetzt, welche unterschiedliche Teilfunktionen des mobilen Roboters adressieren. So ist es beispielsweise üblich, dass es einen *Controller* für das interne Energiemanagement gibt und meistens davon getrennt, einen *Controller* zum Betreiben der Motoren. Letzterer sollte hingegen die Sensoren zur Hinderniserkennung, typischerweise Ultraschall oder Infrarotsensoren, beinhalten. Diese beispielhaft genannten *Controller* zeigen zudem unterschiedliche Anforderungen an die Echtzeitfähigkeit. Wenngleich auch die Sensordaten zum Batteriezustand möglichst aktuell dem Informationssystem zur Verfügung stehen sollten, insbesondere wenn beispielsweise der aktuelle Energieverbrauch gemessen wird, ist die Umsetzung von Motorstellwerten zeitkritischer. Im Falle eines drohenden Zusammenstoßes muss der mobile Roboter so schnell wie möglich, d. h. mit der kleinstmöglichen Verzögerung ausweichen oder zum Stillstand kommen. Hierzu arbeiten die *Controller* quasi-kontinuierlich, die Verarbeitung der Mess- und Aktuatorwerte erfolgt ständig und in äquidistanten Zyklen. Verfügt der *Controller* zum Betreiben der Motoren auch über Sensoren zur Hinderniserkennung, so werden diese Daten lokal vor dem Anfahren neuer Stellwerte intern validiert[2]. So wird sichergestellt, dass der mobile Roboter keinesfalls Hindernisse berührt, ungeachtet vom Gesamtzustand des verteilten Systems.

Die Implementierung der *Controller* erfolgt i. d. R. mit Methoden der Regelungstechnik, d. h. Stellwerte werden mittels Regelkreisen ermittelt. Zudem liefern die *Controller* üblicherweise Messwerte nicht als Rohdaten in das System bzw. die nächsthöhere Instanz der Architektur, sondern führen eine Vorverarbeitung durch. Diese Vorverarbeitung konsolidiert die Daten, indem beispielsweise invalide Daten erkannt und geprüft werden. Weiterhin werden nicht zwangsläufig alle dem *Controller* zur Verfügung stehenden Daten versendet, damit die Datenmenge auf dem Kommunikationskanal reduziert werden kann. So werden z. B. aus einem Kamerabild bereits die Merkmale extrahiert, sodass nicht das gesamte Bild, sondern einzelne Pixelkoordinaten übertragen werden. Des Weiteren werden unterschiedliche Modi oder Konfigurationen bei der Implementierung der *Controller* eingesetzt. Befindet sich der Roboter im Stillstand, so kann die Erkennung von Hindernissen langsamer erfolgen oder die Komponente gänzlich ausgeschaltet werden, um Energie zu sparen oder die Informationsdichte im Gesamtsystem und besonders auf den Kommunikationskanälen zu reduzieren.

[2] wird nachfolgend in Kapitel 4.2.4 erläutert

Der hierarchisch über den *Controllern* angesiedelte *reflektorische Operator* steuert die Modi und Konfigurationsparameter der *Controller*. Zusammen mit den *Controllern* bilden sie den **reflektorischen Kreis** und sind echtzeitfähig miteinander verbunden. Der *reflektorische Operator* arbeitet hauptsächlich ereignis-gesteuert, indem auf neue Informationen reagiert wird. Hierbei wird die Verarbeitung dieser Ereignisse typischerweise in Zustandsautomaten[3] abgebildet. Die Steuerung der *Controller* erfolgt durch das Übersenden neuer Stellwerte und durch die (Re-) Konfiguration dieser, beispielsweise durch die Auswahl anderer Modi und die Änderung von Konfigurationsparametern. Hierzu werden beispielsweise Umweltinformationen, Informationen zum Zustand des Systems oder Informationen zu Hindernissen aller *Controller* fusioniert und ausgewertet. Dieses führt dann, nach dem reaktivem Paradigma der Robotik[4], zum Ausführen einer Aktion. Dieses erwirkt der *reflektorische Operator* beispielsweise durch das Übersenden neuer Führungsgrößen für die Motoren, welche durch den entsprechenden Regelkreis im *Controller* umgesetzt werden.

Die Ausführung des reaktiven Paradigmas der Robotik im OCM ist hierbei auf zwei Ebenen möglich. Die Koordination aus mehreren *Controllern* erfolgt mittels des *reflektorischen Operators*. Hierbei liefert beispielsweise *Controller #1* (siehe Abbildung 4.3a) Sensorinformationen, die zu einer Reaktion auf *Controller #2* führen. Typischerweise erfolgt eine Aktion aus der Fusion von mehreren Informationen mittels des *reflektorischen Operators*. Im OCM ist es jedoch zudem möglich, direkt auf Ebene der *Controller* das reaktive Paradigma aus „Wahrnehmen und Ausführen“ durchzuführen, z. B. wenn sowohl die benötigten Sensorinformationen, als auch der benötige Aktuator am gleichen *Controller* angebunden sind (siehe Abbildung 4.3b). Dieses wird beispielsweise bei Sicherheitsfunktionen (Safety) eingesetzt[5].

Der *reflektorische Operator*, welcher auch durch eine Kombination mehrerer Recheneinheiten gebildet sein kann, arbeitet hierbei in Echtzeit, wobei die Anforderungen an die Echtzeit nicht so hoch sind, wie bei den *Controllern*, da diese neuen Stellwerte vor dem Ausführen intern überprüfen. Der *reflektorische Operator* verarbeitet typischerweise die Informationen aller *Controller*, sodass dieser auf leistungsfähigerer Hardware implementiert wird.

Die Kombination aus *Controllern* und dem *reflektorischen Operator* bilden den **autonomen Klienten**[6] (siehe Abbildung 4.2) des Gesamtsystems, d. h. das Grundsystem aus *reflektorischem Operator* und *Controller* enthält alle notwendigen Komponenten, damit der mobile Roboter nach dem reaktiven Paradigma der Robotik eingesetzt werden kann. Der

[3] engl. Finite State Machine (FSM)
[4] vgl. Abbildung 3.1b in Kapitel 3.1.1
[5] wird nachfolgend in Kapitel 4.2.4 erläutert
[6] engl. Autonomous client

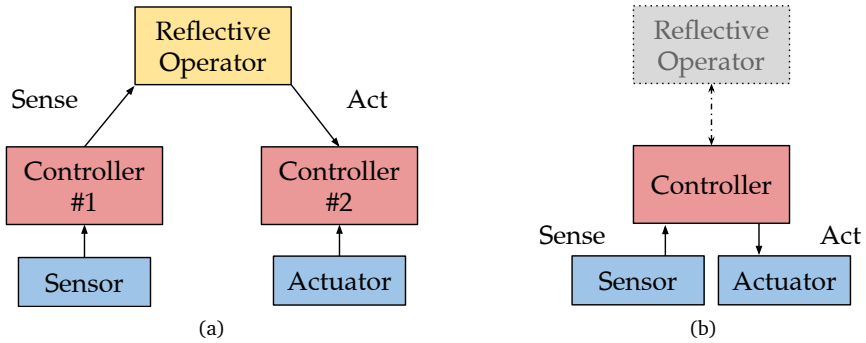


Abbildung 4.3: Reaktives Paradigma der Robotik im OCM

reflektorische Operator ist in der Lage, die aktuelle Situation des mobilen Roboters zu erkennen und darauf zu reagieren. Hierzu bedarf es zudem der Informationen des *Controller*-basierten motorischen Kreises.

Der *reflektorische Operator* bildet zudem die Schnittstelle zur obersten Schicht des OCM, dem sogenannten ***kognitivem Operator***. Der *kognitive Operator*, bzw. die Unterteilung in *reflektorischen Operator* und *kognitiven Operator*, war in der OCM Variante von Naumann [32] zunächst nicht vorgesehen, ist jedoch auch im *SFB-614-OCM* enthalten.

Im Gegensatz zum *SFB-614-OCM* kann der *kognitive Operator* des hier vorgeschlagenen Modells einer Systemarchitektur für mobile Roboter außerhalb des physischen Systems verortet sein und erweitert den autonomen Klienten mit einem cloudbasierten Ansatz (siehe Abbildung 4.4). Somit wird das verteilte System in einen **lokalen** und einen **Cloud Bereich** unterteilt. Die Aufgaben des *kognitiven Operators* sind die Missionsplanung, sowie die Optimierung des Systems. Der *kognitive Operator* erweitert den reaktiven autonomen Klienten somit zu einem hybriden deliberativen/ reaktiven System[7], bei welchem parallel zum reaktiven Paradigma, also dem „Wahrnehmen und Ausführen“, die Planung durchgeführt wird (siehe Abbildung 4.5). Hierbei wird die Planung asynchron zum „Wahrnehmen und Ausführen“ realisiert. Während das reaktive Paradigma in kurzen Zeitabständen und mit harter Echtzeit implementiert wird, kann die Planung, also der deliberative Anteil des hybriden Paradigmas in längeren Abständen und mit **weichen Echtzeitanforderungen** erfolgen.

[7] vgl. hybrides deliberatives/ reaktives Paradigma in Abbildung 3.1c in Kapitel 3.1.1

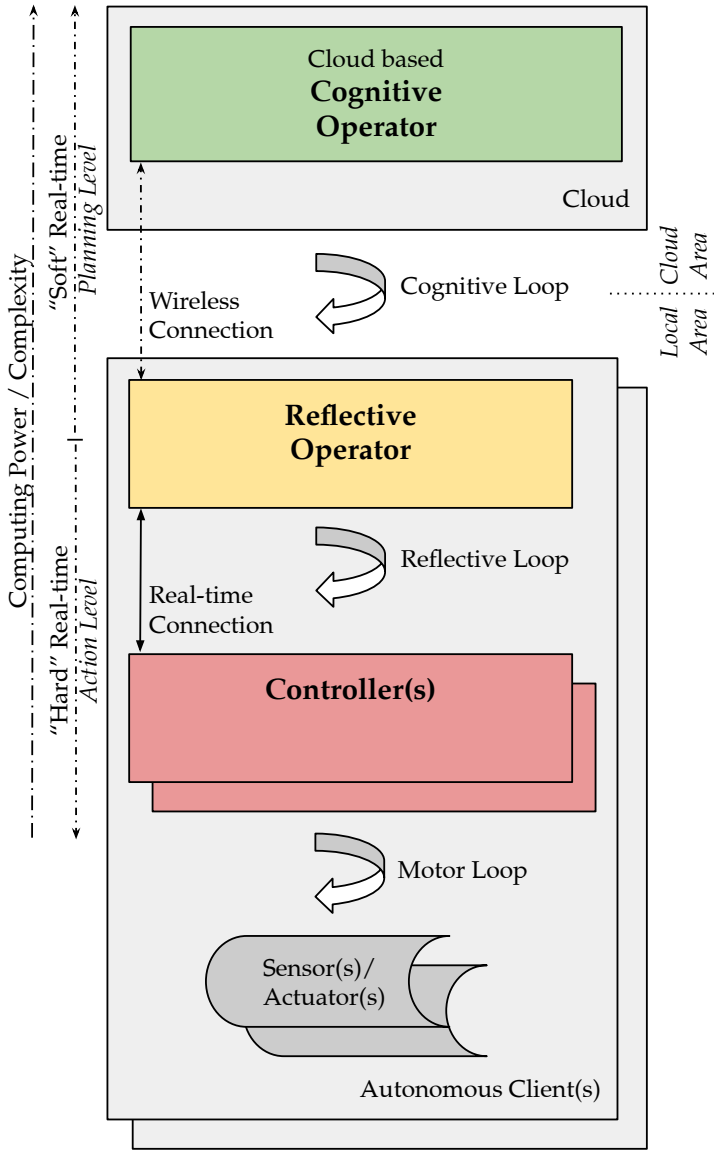


Abbildung 4.4: Architekturkonzept für mobile Roboter

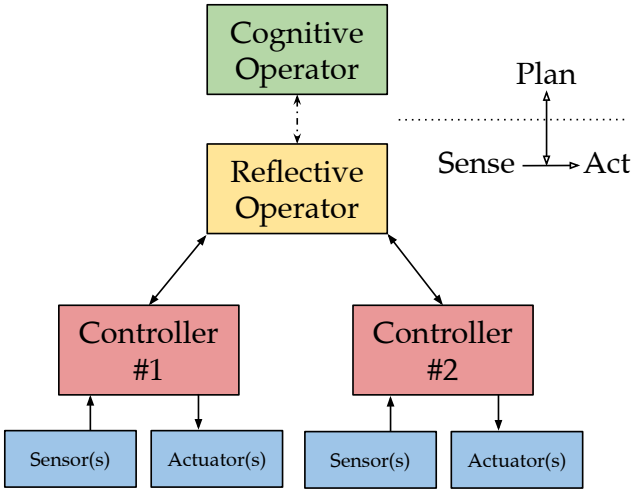


Abbildung 4.5: Hybrides deliberatives/ reaktives Paradigma der Robotik im OCM

Die Methoden und Algorithmen zur Optimierung des Systems benötigen i. d. R. leistungsstarke und weniger energieeffiziente Hardware, welche im Widerspruch zur Nutzung von energieeffizienter Hardware in mobilen Robotern steht. Durch die Topologie der Architektur ist die Einbindung einer Cloud möglich, da es für die Optimierung und die Planung des Systems keine harte, sondern weiche Echtzeit bedarf. Die Optimierungen erfolgen wie die Planungen asynchron zum lokalen Klienten und müssen nicht unmittelbar ausgeführt werden.

Unter Umständen ist die Umsetzung des *kognitiven Operators* als lokale Komponente auf dem mobilen Roboter möglich, wenngleich dies typischerweise nicht vorgesehen ist. In Ausnahmefällen, wenn beispielsweise bereits bei der Entwicklung des Roboters klar ist, dass dieser niemals Zugriff auf die Cloud haben wird, kann der *kognitive Operator* mit leistungsstarker Hardware lokal umgesetzt werden. Hierzu wird eine leistungsstarke Energiequelle oder eine autarke Energieversorgung benötigt. Als Beispiel sei hier der Einsatz in unerschlossenen Gegenden oder gar auf anderen Planeten zu nennen, wie beispielsweise im Falle des Mars Rovers [97], welcher durch Solarkollektoren energetisch versorgt wird.

Der *kognitive Operator* bildet zusammen mit dem *reflektorischen Operator* den **kognitiven Kreis**. Dieser Kreis ist zum Betrieb des mobilen Roboters nicht unmittelbar erforderlich. Das Konzept sieht vor, dass der *kognitive Operator* immer dann eingesetzt wird, wenn dieser verfügbar ist. Ist der *kognitive Operator* nicht erreichbar, so werden die Informationen für den *kognitiven Operator* lokal zwischengespeichert und bei der nächsten

Verfügbarkeit der Cloud-Dienste hochgeladen. Die Integration der Cloud ermöglicht zudem die Analyse des mobilen Roboters, wenn dieser nicht verfügbar ist. So können die Sensor- und Aktuatordaten des Roboters nach dem Einsatz betrachtet und analysiert werden, während der Roboter ausgeschaltet ist, gewartet oder repariert wird. Zu diesem Zeitpunkt können ebenfalls Simulationen für kommende Aufgaben durchgeführt werden, ohne dass hierzu der Roboter selbst verwendet werden muss.

Der *kognitive Operator* ist außerdem als kognitive Ebene zur Planung und Koordination von mehreren mobilen Robotern vorgesehen. Das konzeptionelle Modell sieht vor, dass es je System einen *kognitiven Operator* gibt, welcher **gemeinsam** genutzt wird. Der gemeinsam genutzte und cloudbasierte *kognitive Operator* übernimmt hierbei die Planung und Verteilung von Aufgaben auf die einzelnen Roboter in einem Multi-Roboter System.

Der *kognitive Operator*, sowie die Schnittstellen des *reflektorischen Operators* zu diesem, bilden die **Planungsebene**[8]. Die *Controller* bestimmen, zusammen mit den Schnittstellen des *reflektorischen Operators* zu diesem, unmittelbar die Aktionen des Systems und werden der **Aktionsebene**[9] zugeordnet.

Die **Rechenleistung**[10] sowie die **Komplexität**[11] nehmen im hierarchischen Ansatz des verteilten Systems nach oben hin zu. Wie bereits erwähnt, führen die *Controller* ausschließlich Teilaufgaben des Systems durch und nutzen dazu typischerweise Methoden der Regelungstechnik. Der *reflektorische Operator* hingegen koordiniert die *Controller* und verarbeitet dazu (parallel) die Informationen aller *Controller*. Dies führt zu einer ungleich komplexeren Aufgabe, deren Algorithmen und Methoden typischerweise mehr Rechenleistung benötigen. Der *kognitive Operator* verwendet Methoden zur Optimierung des Systems und führt dazu beispielsweise Simulationen oder auch Langzeitanalysen durch. Hier kommen oft Methoden des maschinellen Lernens zum Einsatz, welche hochkomplex und rechenintensiv sind, wie Sze, Chen, Emer, Suleiman und Zhang in [140] zeigen.

[8] engl. Planning level
[9] engl. Action level
[10] engl. Computing power
[11] engl. Complexity

4.2 Schlüsselmerkmale

Die Darstellung der Schlüsselmerkmale des konzeptionellen Modells zeigt ausgewählte Teilaspekte der Architektur im Detail und begründet damit die Entscheidung zur Auswahl des OCM-Ansatzes als Architektur für mobile Roboter, bezugnehmend auf die Anforderungen an die Systemarchitektur von mobilen Robotern (vgl. Kapitel 3.4.3). Des Weiteren werden Anpassungen des Architekturkonzeptes im Vergleich zum *SFB-614-OCM* begründet.

Die *SFB-614-OCM* Architektur beschreibt die Aufteilung auf die drei Schichten *Controller*, *reflektorischem Operator* und *kognitivem Operator*. Das konzeptionelle Modell sieht eine Umsetzung dieser Architektur als verteiltes System vor, beschreibt also nicht nur eine Softwarearchitektur, sondern eine Systemarchitektur. Hierzu wird auch die Auswahl der Hardware erläutert, wenngleich sich diese Auswahl auf die verfügbare Hardware zum Zeitpunkt des Verfassens der Dissertation (Ende 2020) bezieht.

4.2.1 Energieeffizienz und Leistungsfähigkeit

Die Verteilung der Gesamtfunktionalität auf mehrere Rechner und die Einbindung der Cloud machen es möglich, sowohl komplexere Algorithmen zu verarbeiten, als auch die Energieeffizienz des Systems zu gewährleisten. Die Energieeffizienz hängt auch von der Auswahl der Rechner ab. Die Realisierung des Systems aus energieeffizienten Rechnern, beispielsweise aus SoCs, welche insbesondere durch die weite Verbreitung von eingebetteten Systemen populär sind, erhöht hierbei die energie-autarke Betriebszeit der Roboter bei gleichbleibender Energieversorgung im Vergleich zum klassischen Computer, beispielsweise einem PC oder Notebook, als Recheneinheit. Ein SoC ist hierbei energieeffizienter, u. a. durch die direkte Anbindung der Peripherie innerhalb des Chips und die daraus resultierende geringere physikalische Verlustleistung durch kürzere Verbindungsstrecken im Vergleich zu aktuellen x64-Prozessorarchitekturen. Dieses wird auch dadurch erreicht, dass ein SoC notwendige Peripherie, wie RAM, GPU und typische Schnittstellen, integriert. Die Leistung der CPU des SoC, welche i. d. R. auf der ARM-Architektur basiert, ist hierbei in den letzten Jahren zwar gestiegen, nichtsdestotrotz sind x64-Prozessoren im Vergleich typischerweise weiterhin höher getaktet (siehe CPU-Takt in Tabelle 4.1). Durch den Einsatz der ausgewählten verteilten Architektur ist es jedoch möglich, mehrere Kleinstrechner zu verwenden, damit insgesamt ausreichend Leistung zur Verfügung steht. Neben SoC ist es, insbesondere bei der Entwicklung von Prototypen, möglich, SBCs einzusetzen. Diese verfügen neben dem SoC typischerweise auch über Anschlüsse und Schnittstellen zur Anbindung von Sensoren und Aktuatoren.

In Tabelle 4.1 werden Kleinst- und Kleinrechner miteinander verglichen, welche besonders bei der Entwicklung von Prototypen genutzt werden. Hierbei werden vier unterschiedliche Ansätze verglichen, von einem langsam getakteten Arduino Mega [O24], bis hin zu einem leistungsstarken x64-Prozessor basierten PC in kleinem Formfaktor, dem NUC von Intel [O25].

	Arduino Mega REV3	**SMT32F3 Discovery**	**Raspberry Pi 3 Model B+**	**Intel NUC NUC8i5BEK**
Prozessorarchitektur	ATmega2560	ARMv7 Cortex-M4	ARMv8 Cortex-A53	Intel Core i5-8259U
Anzahl Prozessorkerne	1	1	4	4
CPU-Takt	16 MHz	72 MHz	1,4 GHz	2,3 - 3,8 GHz
RAM	8 KB	48 KB	1 GB	bis 32 GB
Schnittstellen (Auszug)	54 I/O Pins (digital), 16 I/O Pins (analog)	USB Port, 16 I/O Pins	4 USB 2.0 Ports, 26 I/O Pins	6 USB Ports
Kommunikation (Auszug)	Diverse (durch Aufsteck-Module)	Diverse (durch Aufsteck-Module)	Ethernet, WLAN, Bluetooth	Ethernet, WLAN, Bluetooth
Betriebsystem	*Bare machine*[1]	ChibiOS, FreeRTOS	Diverse, meist Linux basiert	Windows 10
Echtzeitfähigkeit	✓	✓	✓ (mit Echtzeit-OS)	× (ggf. mit alternativem Betriebssystem)
Leistungsbedarf	0,4 W [O26]	1 W [2]	1,5 - 5 W	5 - 70 W [O27]
Preis	~35€	~20€	~45€ (inkl. SD-Karte)	>200€ (abhängig von der Konfiguration)
Eignung als *Controller*	✓	✓	✓ (mit Echtzeit-OS)	×
Eignung als *reflektorischer Operator*	×	×	✓	Eingeschränkt

[1] Befehlssatz wird direkt auf der Logik der Hardware aufgeführt, ohne dass ein Betriebssystem eingesetzt wird
[2] Eigene Messung

Tabelle 4.1: Eignung von Rechnern für die Systemarchitektur

Die NUC Familie von Intel, hier in der Version NUC8i5BEK [O25], repräsentiert einen klassischen Computer mit einer x64-Prozessorarchitektur von Intel, wenngleich der NUC in einem kleinen Gehäuse verbaut ist und im Vergleich zu Desktop-Rechnern und Notebooks zu den energieeffizienteren Varianten der klassischen Computer zählt. Dieser Rechner ist potenziell alleinstehend leistungsstark genug, um alle Aufgaben des *reflektorischen Operators* und der *Controller* zu übernehmen, auch die Aufgaben des *kognitiven Operators* könnten parallel ausgeführt werden. So könnte der mobile Roboter als monolithisches System umgesetzt werden, welches, wie nachfolgend erläutert wird, die Anforderungen an die mobile Robotik jedoch nicht erfüllt. Im Hinblick auf die Energieeffizienz und ggf. den Preis ist auch die Eignung eines klassischen Rechners als *reflektorischer Operator* nur eingeschränkt empfehlenswert. So kann es bei sehr großen mobilen Robotern mit leistungsstarker Energieversorgung möglich sein, dass der hohe Energieverbrauch keinen großen Einfluss auf den Gesamtverbrauch des Systems hat. Durch das Fehlen eines Betriebssystems, welches harte Echtzeitbedingungen erfüllen kann, ist auch der Einsatz als *Controller* nicht empfehlenswert. Der Vergleich der Kleinstrechner mit dem Intel NUC in Tabelle 4.1 zeigt zudem, dass durch den Ansatz, ein verteiltes System innerhalb eines mobilen Roboters aus mehreren der dargestellten Kleinstrechnern zu realisieren, der Gesamtverbrauch des Systems geringer ist, als bei der Wahl eines einzelnen klassischen Rechners.

Die Arduino Familie zeichnet sich besonders durch eine einfache Programmierung aus. Außerdem bietet insbesondere der Arduino Mega 2560 [O24] eine große Anzahl an GPIO Pins zum Anschluss von Sensoren und Aktuatoren. Durch diverse Aufsteckmodule (*Shields*) ist die Arduino Familie erweiterbar, beispielsweise durch Module für übliche Kommunikationstechnologien, wie Wireless Local Area Network (WLAN) oder Bluetooth. Ein Arduino wird durch einen vergleichsweise schwachen Rechner betrieben, wodurch der Arduino Mega von einem geringen Leistungsbedarf von unter 1 W profitiert. Als *reflektorischer Operator* ist ein Arduino nicht geeignet, zur parallelen Abarbeitung der Systemdaten fehlt beispielsweise auch mindestens ein weiterer Prozessor-Kern. Zudem ist die Leistung für einen *reflektorischen Operator* nicht ausreichend. Interessant ist ein Kleinstrechner aus der Arduino Familie jedoch als *Controller*, beispielsweise zur Anbindung von einfachen Sensoren und Aktuatoren.

Die STM32 Familie, hier ein STM32F3 Discovery [O28] Entwicklungsboard, ist ebenfalls vergleichsweise leistungsschwach, wenngleich hier bereits ein ARM-Prozessor eingesetzt wird. Dieser verfügt ebenfalls über nur einen Prozessorkern und ist somit nicht für den Einsatz als *reflektorischer Operator* geeignet. Die Möglichkeit, externe Sensoren anzuschließen, sowie das intern verfügbare Gyroskop ist jedoch prädestiniert für den Einsatz als *Controller*. Hierfür spricht auch der geringe Energieverbrauch und die

Unterstützung von Echtzeitbetriebssystemen wie ChibiOS [O29] und FreeRTOS [O30] für viele Entwicklungsboards der STM32 Familie.

Die Raspberry Pi Produktfamilie, hier in der Version Raspberry Pi 3 Model B+, ist mit über 10 Millionen verkauften Einheiten (Stand 2016, [O31]) der erfolgreichste SBC. Der Kleinstrechner findet heute in diversen Anwendungsgebieten Einsatz, von der Hausautomatisierung wie in [141] beschrieben, bis hin zur mobilen Robotik, wie u. a. [142] zeigt. Der Energieumsatz ist entsprechend der höheren Rechenleistung im Vergleich zum Arduino höher, jedoch nur einen Bruchteil so groß wie der Energieverbrauch eines NUC. Der Raspberry Pi bietet zudem zahlreiche Schnittstellen und eine große Auswahl an Betriebssystemen, unter denen auch echtzeitfähige Betriebssysteme sind. Der Raspberry Pi ist im verteilten OCM für *Controller* mit höheren Leistungsanforderungen, sowie auch für die Realisierung eines *reflektorischen Operators* geeignet. Hier ist die parallele Verarbeitung der Systeminformationen erforderlich, welche der Raspberry Pi durch seine Mehrkern-Architektur unterstützt.

Durch die Verschiebung des *kognitiven Operators* in die Cloud, kann das Roboter-System sowohl anspruchsvolle Berechnungen durchführen, als auch lokal energieeffiziente Rechner einsetzten. Hierbei ist der lokale Bereich leistungsstark genug, um den Roboter autonom zu steuern. Der *kognitive Operator* in der Cloud kann bedarfsgerecht skaliert werden. Dieses kann von der Simulation des gesamten lokalen verteilten Systems mit einer alternativen *Controller*-Konfiguration reichen oder zur Koordination von mehreren Robotern genutzt werden. Dieses wäre in einem monolithischen System nur beim Einsatz von sehr leistungsstarker Hardware umsetzbar, deren Energieverbrauch vor allem bei kleinen mobilen Robotern mit stark limitierter Akkumulatorkapazität nicht vereinbar ist.

In „Robots with their heads in the clouds“ [118] wird die These unterstützt, dass mobile Roboter mit Cloud-Einbindung ein logischer nächster Schritt bei der Entwicklung von mobilen Robotern sind. „Cloud Computing kann Roboter kleiner, günstiger und intelligenter machen“ ist die Kernaussage des Artikels. Da Silva, Vilao, Costa und Bianchi [143] zeigen, wie die Kognition von Robotern mit neuronalen Netzen, sowie Deep Learning realisiert werden kann. Im konzeptionellen Modell kann deren Realisierung auf dem cloudbasierten *kognitiven Operator* ausgeführt werden. Weitere Deep Learning Methoden und Algorithmen für Roboter, deren Realisierung die vorgestellte Systemarchitektur unterstützt, vergleichen Shabbir und Anwer [144]. Die Notwendigkeit bzw. die Vorteile einer Cloud werden außerdem in „Cloud robotics: Current status and open issues“ [119] dargestellt. Hierbei zeigen die Autor*innen, dass das Cloud Computing in der Robotik die Entwicklung von Multi-Roboter Systemen fördert. Weiterhin wird beschrieben, wie das Verschieben von komplexen Berechnungen in die Cloud, den Einsatz von Hochleistungsrechnern auf den Robotern unnötig macht.

4.2.2 Modularität

Schöpping, Korthals, Hesse und Rückert beschreiben in [40] die Notwendigkeit und Wichtigkeit der Modularität insbesondere in der modernen Robotik und Roboter ähnlichen Maschinen. Roboter werden in diversen Umgebungen eingesetzt und müssen entsprechend in der Lage sein, sich wechselnden Gegebenheiten anzupassen. Hierbei kann es von Vorteil sein, wenn der mobile Roboter durch einen Sensor erweitert wird, beispielsweise durch eine Wärmebildkamera zum Einsatz in einem Feuerwehr-Szenario.

Das vorgestellte Konzept fördert diese Modularität. Ein solches verteiltes System besteht aus mehreren Modulen, welche durch Kommunikationskanäle miteinander verbunden sind. Die Module selbst sind hardwareseitig voneinander getrennt und können sich gegenseitig nicht in Bezug auf die vorhandenen Hardwareressourcen beeinflussen, da jedem Modul ein eigener Rechner zur Verfügung steht. Parallele Systeme oder monolithische Systeme mit modularer Programmierung teilen sich i. d. R. Hardware wie den Arbeitsspeicher, auch wenn Softwaremodule streng auf verschiedene Prozessorkerne verteilt werden können. Bei der Veränderung eines Moduls oder dem Einfügen eines neuen Moduls, müssen die vorhanden Hardwareressourcen ggf. neu verteilt werden.

Im vorgeschlagenem verteilten System kann beispielsweise ein Sensor, inklusive eines eigens dafür entwickelten *Controllers* in das System eingebunden werden. Durch den modularen Aufbau als verteiltes System, ist die Struktur des OCM für solche Fälle vorbereitet. Um neue Hardware einzubinden, sind keine grundlegenden Veränderungen am System notwendig. Die Realisierung einer Schichtenarchitektur als verteiltes System führt hierbei zu einer klaren hardwareseitigen Trennung. Neue Komponenten können anderen Funktionen des Systems keine Hardwareressourcen entziehen, sodass die Echtzeitfähigkeit der anderen *Controller* in jedem Fall erhalten bleibt. Die Modularität fördert somit die Zukunftssicherheit des mobilen Roboters, wodurch dessen Anpassung für veränderte Aufgaben ohne großen Aufwand umgesetzt werden kann.

Die Notwendigkeit von modularen Robotern wird u. a. in „Evolutionary modular robotics: Survey and analysis“ [145] erläutert. Die Autoren heben hierbei die Vielseitigkeit und Robustheit der mobilen Roboter mit modularem Ansatz hervor. Die gleichen Hauptmerkmale werden in „Modular self-reconfigurable robot systems [Grand challenges of robotics]“ [77] beschrieben. Die Vielseitigkeit wird u. a. mit der Re-Konfiguration der Module und somit des Roboters begründet, welche dazu führt, dass mobile Roboter in unterschiedlichen Umgebungen und Aufgaben eingesetzt werden können. Die höhere Robustheit im Vergleich zu nicht-modularen mobilen Robotern erläutern Alattas, Patel und Sobh [145] mit der Möglichkeit, redundante Module einzusetzen, sowie der möglichen Selbstreparatur, bei der die Aufgaben eines defekten Moduls von einem anderen Modul übernommen werden.

4.2.3 Entkopplung

Ein Vorteil der OCM Schichtenarchitektur als verteiltes System innerhalb eines mobilen Roboters ist die klare Unterteilung in Architekturbereiche mit harter Echtzeit und Bereiche mit weniger harten Anforderungen an die Echtzeitfähigkeit, sowie in zeit-diskrete und ereignis-gesteuerte Methoden. Auf dem Aktionslevel ist hierbei harte Echtzeit notwendig. Die davon getrennte Planungsebene hingegen kann mit weichen Echtzeitanforderungen implementiert werden. Auf *Controller*-Ebene werden Sensoren zeitlich äquivalent abgefragt (quasi-kontinuierlich), der *reflektorische Operator* hingegen arbeitet asynchron dazu und reagiert auf Ereignisse, welche intern generiert werden oder durch das Erhalten neuer Informationen von außen erfolgen. Die Schichtenarchitektur und deren Umsetzung als verteiltes System macht das asynchrone Ausführen und die Entkopplung der Hardware der Komponenten möglich. So können beispielsweise Echtzeitbetriebssysteme mit Betriebssystemen ohne Echtzeitfähigkeit kombiniert werden. Die Interaktion der Rechner untereinander erfolgt ausschließlich über Kommunikationskanäle. Die klare Schichtenstruktur fördert außerdem das einfache und klare Einordnen von Modulen in die Architektur. Beispielsweise bei Komfortfunktionen, wie dem Anzeigen von Nachrichten mittels einer GUI, bedarf es keiner harten Echtzeit. Ein solches Modul wird entweder dem *reflektorischen Operator* zugeordnet, falls die Daten lokal angezeigt werden sollen oder mittels des *kognitiven Operators* realisiert werden, falls die GUI in der Cloud verfügbar sein soll. Der OCM-Ansatz verhindert somit das Mischen von Funktionen oder Algorithmen von harten und weichen Echtzeitanforderungen, was ansonsten das Erfüllen der Gesamtanforderungen des mobilen Roboters gefährden kann. Das Vermischen von Funktionen mit unterschiedlichen Anforderungen ist u. a. aus der Entwicklung von *Systems of Systems* bekannt [5]. Wenn keine klare Struktur und Hierarchie im Architekturkonzept definiert wird, ist die Integration in das vorhandene System nicht einfach und kann zu Fehlern durch Störungen untereinander führen. Besonders bei wachsenden Systemen, also beim nachträglichen Hinzufügen von neuer Hard- und Software, ist eine eindeutige Struktur und Hierarchie von Vorteil.

Bei monolithischen Systemen sind Softwaremodule zudem von der gemeinsamen Hardware abhängig, z. b. vom Prozessortakt. Gelingt die Integration von neuer Software in die Architektur, weil diese beispielsweise in Softwareschichten voneinander getrennt ist, so ist diese trotzdem, aufgrund der gemeinsamen Hardware, nicht als vollständig entkoppelt zu betrachten. Ein fehlerbehaftetes und blockierendes Softwaremodul mit weicher Echtzeit kann beispielsweise die Funktionen mit harter Echtzeit beeinflussen und einen mobilen Roboter in einen unsicheren Zustand führen.

4.2.4 Sicherheit und Zuverlässigkeit

Das OCM als verteilte Systemarchitektur kommt besonders der Sicherheit und Zuverlässigkeit von mobilen Roboter zu Gute. Das Sicherstellen von Sicherheit und Zuverlässigkeit ist insbesondere bei mobilen Robotern notwendig, da diese eine potenzielle Gefahr für die Umgebung darstellen. So kann beispielsweise ein Fehler in der Rotorsteuerung von UAVs zum potenziell gefährlichen Absturz eines UAV auf Menschen führen.

Safety

Die Sicherheit (Safety) und Zuverlässigkeit von technischen Systemen ist systemkritisch, d. h. sind mobile Roboter nicht sicher oder zuverlässig, so können sie sich selbst schädigen oder anderweitig Schäden verursachen. Hierbei führt ein zuverlässiges System zur Erhöhung der Sicherheit (Safety). Das Testen und die Verifikation der einzelnen Funktionen sowie der Integration des Gesamtsystems ist während der Entwicklung unabdingbar und trifft auf jede Systemarchitektur gleichermaßen zu und wird hier nicht näher betrachtet. Nachfolgend werden Maßnahmen beschrieben, welche die Sicherheit der verteilten OCM Architektur u. a. im Vergleich zu monolithischen Systemen erhöhen und somit die Anforderungen an die Sicherheit (Safety) der Systemarchitektur für mobile Roboter erfüllen.

Ein Nachteil von monolithischen Systemen ist der Single Point of Failure (SPoF)[12]. Ein monolithisches System ist von einem einzigen Rechner abhängig. Im Fehlerfall des einzigen Rechners, fällt das gesamte System aus. Hierbei reicht es z. B. aus, wenn eine Komfortfunktion wie eine GUI einen Fehler enthält, um die gesamte CPU zu blockieren. Dieses bringt den gesamten Roboter in einen kritischen Zustand, weil beispielsweise Hindernisse nicht mehr erkannt und die Motoren nicht gestoppt werden können. Im verteilten System sind die Funktionen des Systems auf mehrere, nahezu unabhängig agierende Rechner verteilt. Zudem ist das OCM streng in drei Ebenen unterteilt, bei denen *Controller*-Funktionen und Funktionen des *reflektorischen Operators* voneinander getrennt sind. Im oben beschriebenen Fall, können selbst bei einem Fehler der GUI, die Kamera und Abstandssensoren weiterhin Hindernisse erkennen. Die klare Unterscheidung des OCM von Komponenten mit harten Echtzeitanforderungen und Komponenten mit weicher Echtzeit fördert zudem, dass sicherheitskritische Funktionen nicht von nicht-sicherheitskritischen Funktionen unterbrochen werden können.

Ein verteiltes System ist nichtsdestotrotz nicht frei von SPoF. Hier sei beispielsweise die Energieversorgung zu nennen. Falls das gesamte verteilte System von einem Akkumulator betrieben wird, so führt ein Ausfall dessen zwangsweise auch zum Stoppen des Systems.

[12] dt. Einzelner Ausfallpunkt

Hierbei würde dieses jedoch auch die Motoren betreffen, was bei WMR weniger gefährlich ist, als bei UAVs. Dieser SPoF kann verhindert werden, indem es entweder eine redundante, also mehrfach vorhandene Stromversorgung gibt oder die einzelnen Komponenten von verschiedenen Akkumulatoren versorgt werden. Ein weiterer SPoF bei verteilten Systemen ist typischerweise die Kommunikation. Hier muss sichergestellt werden, dass der Ausfall der Kommunikation selbstständig entdeckt wird und das System entsprechend handelt, beispielsweise durch den Wechsel in einen Fail-Safe Modus[13].

Das konzeptionelle Modell sieht den Einsatz von Watchdogs vor, welche das System überprüfen. Hierbei wird zwischen Kommunikations-Watchdogs und internen Watchdogs unterschieden. Jede Komponente verfügt über mindestens einen Watchdog, welcher die Verbindung zu einer Partner-Komponente im verteilten System überprüft (Abbildung 4.6). Wird die Trennung der Kommunikation erkannt, so wechseln die Komponenten in den Fail-Safe Mode, indem u. a. unverzüglich die Motoren gestoppt werden. Ggf. kann eine Komponente, zu welcher beispielsweise der *reflektorische Operator* keine Verbindung aufrechterhalten kann, neu gestartet werden. Dieses wird über den sogenannten *Health Controller*[14] realisiert, welcher mittels Relais die Stromversorgung der einzelnen Komponenten steuert. Des Weiteren kann so auch der Fehlerfall des Controllers zum Betreiben der Motoren behoben werden. Wird der Verbindungsverlust zum *Motion Controller* vom *reflektorischen Operator* erkannt, so weist dieser den *Health Controller* an, die Motortreiber auszuschalten. Dadurch werden die Motoren gestoppt. Um das System in den Normalzustand bringen zu können, wird anschließend der *Motion Controller* aus- und wieder eingeschaltet (Selbstheilung).

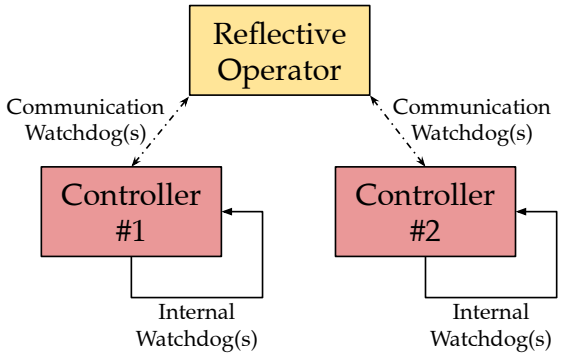


Abbildung 4.6: Komponenten Watchdogs

[13] Modus welcher im Falle eines Fehlers zum geringstmöglichen Schaden führt.
[14] wird nachfolgend in Kapitel 4.3.2 erläutert

Die internen Watchdogs der einzelnen Komponenten (siehe Abbildung 4.6) überprüfen die lokal anliegenden Informationen. Stellt *Controller #1* beispielsweise fest, dass die angebundene Energieversorgung überlastet oder nahezu aufgebraucht ist, kann diese Information selbstständig und ohne dass der *reflektorische Operator* diese Information erfragt hat, an diesen weitergegeben werden. Sicherheitskritische Überprüfungen finden ebenfalls mit internen Watchdogs direkt auf jeder Komponente statt. So kann beispielsweise *Controller #2* die angebundenen Motoren sofort stoppen, wenn die ebenfalls an diesen *Controller* angebundenen Abstandssensoren ein Hindernis erkennen. Dies erfolgt zunächst ohne Anweisung des *reflektorischen Operators*, da hier die Sicherheit im Vordergrund steht und die Aktion des Stoppens in möglichst kurzer Zeit, mit harter Echtzeit, erfolgen soll. Das Überprüfen mit Watchdogs, sowie die Entkopplung der einzelnen Schichten und Rechner im OCM hilft zudem dabei, die Fehlerreaktionsketten so kurz wie möglich zu halten. Fehler werden lokal erkannt, bzw. die einzelnen Rechner werden lokal validiert, sodass keine Fehler im System weitergereicht werden und somit keine unvorhersehbaren Nebeneffekte auftreten sollten.

In „Safety-critical advanced robots: A survey“ [104] wird ebenfalls eine Schichtenarchitektur als Beispiel für eine Architektur gezeigt, an der verschiedene Sicherheitsmethoden erläutert werden. Hierunter fällt beispielsweise auch die Methode *Fehlertoleranz*[15], welche die Zuverlässigkeit erhöht und somit auch zur Sicherheit des Roboters beiträgt.

Zuverlässigkeit

Die Topologie des OCM sieht vor, dass die Regler-basierten *Controller*, Stellwerte vom *reflektorischen Operator* bekommen, welche den Regelkreis parametrisieren. Diese Stellwerte werden im Regler entsprechend verarbeitet, also nicht unmittelbar an den Aktuator weitergegeben. Dies kann dazu genutzt werden, dass der Regler dem Aktuator nur Stellwerte im zulässigen Arbeitsbereich des Aktuators vorgibt. Invalide Stellwerte, wie beispielsweise der direkte Wechsel von Motoren von maximaler Geschwindigkeit im Vorwärtslauf zur maximalen Geschwindigkeit im Rückwärtslauf, ist somit nicht möglich. Die Hardware der Motoren arbeitet somit im vorgesehenen Bereich, wird nicht beschädigt und arbeitet zuverlässig.

Die Fehlertoleranz, also die Realisierung von Maßnahmen um mögliche Fehler auszugleichen bzw. zu tolerieren, trägt zur Zuverlässigkeit eines Systems bei [104]. Bei einem verteilten System ist die Fehlertoleranz an mehreren Stellen umsetzbar. Hier gibt es beispielsweise die Möglichkeit, dass die Aufgabe einer defekten Komponente von einer anderen Komponente ausgeführt wird, damit das System weiter seiner Aufgabe nachgehen kann. Dies setzt voraus, dass schon bei der Programmierung der entsprechenden Programmteile, die Übergabe an eine zweite Komponente berücksichtigt wird. Weiterhin

[15] engl. Fault Tolerance

ist dies nicht möglich, falls beispielsweise die Kamera ausfällt, die typischerweise nur mit einem *Controller* verbunden ist. Hier kann es jedoch u. U. möglich sein, dass die Aufgaben der Kamera, z. B. das Erkennen von Hindernissen durch andere Sensoren, wie z. B. Abstandssensoren kompensiert werden kann. Im konzeptionellen Modell ist es weiterhin vorgesehen, dass Komponenten neu gestartet werden können. So kann die Aufgabe kurzzeitig übernommen und nach dem Neustart des *Controllers* zurückgegeben werden. Das verteilte System hat somit eine Selbstheilung vollzogen.

Ein weiterer Ansatz zur Fehlertoleranz neben der Selbstheilung ist die Implementierung eines (hardware) redundanten Systems, wie in [145] erläutert wird. Hierbei werden i. d. R. Teile der Hardware des Systems, also einzelne Rechner, Sensoren oder Aktuatoren mehrfach ausgeführt und im Fehlerfall aktiviert. Die experimentelle Umsetzung des konzeptionellen Modells (siehe Kapitel 5) hat jedoch gezeigt, dass dieses für den typischen mobilen Roboter nicht notwendig ist. Sollte das System besonders robust umgesetzt werden, weil es beispielsweise über einen langen Zeitraum autonom agieren soll, können redundante Komponenten leicht in die verteilte Systemarchitektur eingebunden werden, indem diese an die vorhandene Kommunikationsschnittstelle angebunden werden.

Die Überwachung und Analyse des Systems, also das Überprüfen des Zustands zum Zeitpunkt des Betriebes, erhöht die Zuverlässigkeit und Sicherheit des mobilen Roboters ebenfalls. Diese Überwachung und Analyse eines verteilten Systems ist aufgrund der Verteilung der Software auf mehrere Rechner nicht einfach, da zu der Überwachung jedes einzelnen Rechners u. a. die Überwachung der Kommunikation im gesamten verteilten System hinzukommt. Die Analyse aller Prüfungen geben dann Aufschluss über den Zustand des Gesamtsystems. Die Analyse des Systemzustands eines verteilten Systems wird in Kapitel 6 erläutert, welches zudem die Entwicklung eines Tools zur Analyse von verteilten Systemen zeigt.

Security

Grundsätzlich sind technische Systeme vor Angriffen von außen zu schützen. Diese Anforderungen an die Sicherheit (Security) berücksichtigt das konzeptionelle Modell durch die strenge Trennung der Schichten im verteilen OCM. Der *kognitiver Operator* hat keinen direkten Zugriff auf die *Controller*, da der *reflektorische Operator* diese beiden Schichten voneinander trennt. Der Roboter kann somit nicht direkt über den *kognitiven Operator* ferngesteuert werden. Der *reflektorische Operator* kann zwar direkt Nachrichten an die *Controller* schicken, diese werden jedoch, wie erläutert, vor der Ausführung geprüft. Die Realisierung als verteiltes System erschwert zudem das Beeinflussen des mobilen Roboters durch Fremdeingriffe, da zur vollständigen Kontrolle zu allen Rechnern des verteilten Systems Zugang hergestellt werden müsste und nicht nur zu einem einzelnen Rechner wie beim monolithischen System. Die *Controller* verfügen hierbei nicht zwangsweise über

drahtlose Kommunikationsschnittstellen, sodass eine physische Verbindung zu den Rechnern hergestellt werden müsste.

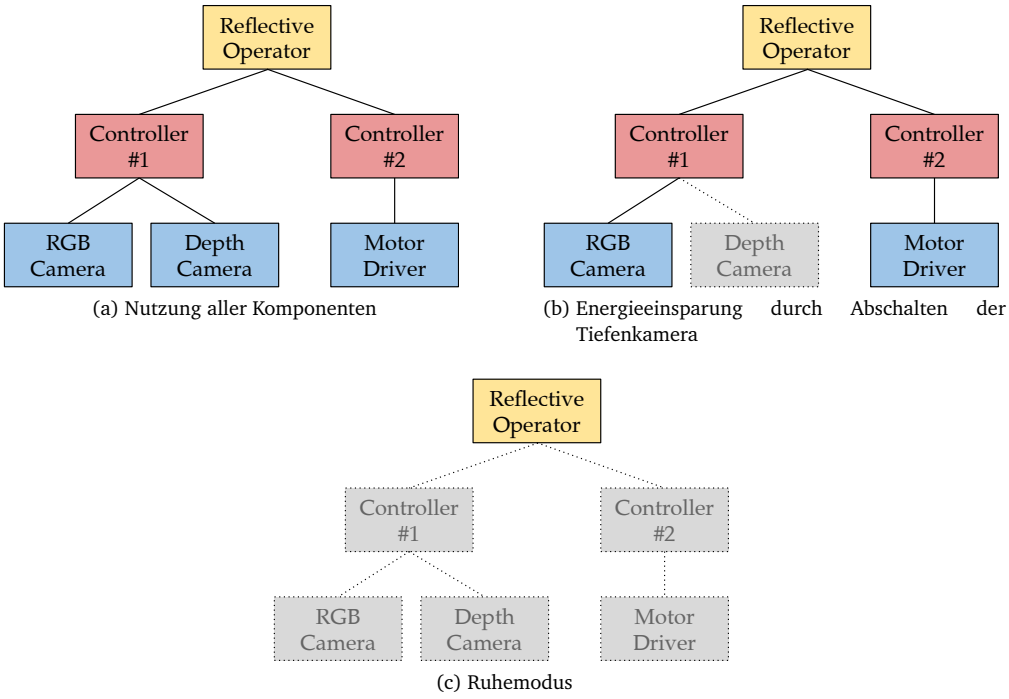
Nichtsdestotrotz ist jeglicher ungewollter Zugriff auf den mobilen Roboter zu vermeiden, wie Steen und Tanenbaum in *Distributed Systems - Principles and Paradigms* [19, S. 501 ff.] erläutern. Hierbei bietet der cloudbasierte *kognitive Operator* den potenziell größten Angriffspunkt, da dieser per Definition mindestens für den Roboter selbst und ggf. für Nutzer*innen und Entwickler*innen des Systems geöffnet werden muss. Die Sicherheit kann erhöht werden, indem eine private Cloud und keine öffentliche Cloud[16] eingesetzt wird. Zudem sollte die Kommunikation durch Methoden und Protokolle wie Transport Layer Security (TLS)[17]/ Secure Socket Layer (SSL) geschützt werden. TLS oder dessen Vorgänger SSL, ist eine Verschlüsselung aus mehreren Protokollen für die Datenübertragung im Internet. Hierzu wird u. a. die Authentizität der Kommunikationspartner sichergestellt. Des Weiteren können Methoden wie das Prüfen der Kommunikation auf Anomalien[18] die Sicherheit erhöhen [146]. Zudem sollten alle Kommunikationstechnologien deaktiviert werden, die nicht benötigt werden, wie beispielsweise drahtlose Verbindungen bei den *Controllern*, falls diese per Kabel im verteilten System angebunden sind.

4.2.5 Bedarfsgesteuerte Komponentennutzung

Die Energieeffizienz eines verteilten Systems kann weiter gesteigert werden, indem, neben der Aktivierung von dynamischen Taktfrequenzen der einzelnen Rechner, einzelne Komponenten des verteilten Systems bedarfsgerecht ein- und ausgeschaltet werden. Abbildung 4.7 zeigt ein Beispiel für mehrere Modi eines mobilen Roboters.

Das Modell besteht aus einem *reflektorischen Operator*, *Controller #1* zur Anbindung einer Red Green Blue (RGB)- und Tiefenkamera, sowie *Controller #2* zur Anbindung der Motortreiber. In 4.7a sind alle Komponenten des verteilten Systems angeschaltet, beispielsweise weil der Roboter sich gerade bewegt und die Tiefenkamera zum Erkennen von Hindernissen benötigt wird. Ändert sich die Umgebung, indem beispielsweise ausreichend Licht vorhanden ist und die Tiefeninformationen von Objekten auch für weitere Funktionen nicht benötigt werden, so reicht ggf. die RGB Kamera für das Erkennen von Hindernissen aus. In dem Fall kann die Tiefenkamera durch die bedarfsgesteuerte Komponentennutzung ausgeschaltet werden und somit Leistung gespart werden, welche im Falle einer Microsoft Kinect bei über 10 W liegt [O32] (siehe Abbildung 4.7b). Außerdem nimmt der Rechenaufwand in *Controller #1* ab, wodurch beispielsweise andere Funktionen des *Controllers* schneller abgearbeitet werden könnten. Im Stillstand können sowohl der Betrieb der Kameras zur Erkennung von Hindernissen, als auch das

[16] wird nachfolgend in 4.4 erläutert
[17] dt. Transportschichtsicherheit
[18] engl. Anomaly Detection



(a) Nutzung aller Komponenten

(b) Energieeinsparung durch Abschalten der Tiefenkamera

(c) Ruhemodus

Abbildung 4.7: Bedarfsgesteuerte Komponentennutzung

Betreiben des dazugehörigen Rechners (*Controller #1*) unnötig sein. Dazu können zudem die Motortreiber und *Controller #2* ausgeschaltet werden. Abbildung 4.7c zeigt diesen Ruhemodus, welcher im Vergleich zum in Abbildung 4.7a dargestellten Modus das gesamte verteilte System auf ein Minimum reduziert und somit einen Großteil der Energie einspart. Im Bedarfsfall kann der *reflektorische Operator* die Komponenten wieder einschalten. Damit die einzuschaltenden *Controller* schnell einsatzbereit sind, muss das Hochfahren der Software (ggf. inkl. Betriebssystem) schnell (z. B. bei WMR in maximal wenigen Sekunden) erfolgen. Dies ist insbesondere bei Echtzeitbetriebssystemen wie FreeRTOS oder ChibiOS, sowie Betriebssystemlosen Systemen, wie z. b. Kleinstrechnern der Arduino Familie, möglich.

4.2.6 Multi-Roboter Kooperation und Mensch-Roboter Interaktion

Die Kooperation eines mobilen Roboters mit anderen Robotern in einem Multi-Roboter Szenario, sowie die Mensch-Roboter Interaktion sind heute Stand der Technik und sollten von der Systemarchitektur unterstützt werden. Hierbei sind, laut Chukwuemeka und Habib

[83], zentrale Architekturen in Bezug auf Zuverlässigkeit, Robustheit und Skalierbarkeit den hybriden Architekturmodellen aus hierarchischen und dezentralen Ansätzen für Multi-Roboter Systeme unterlegen. Der vorgestellte OCM-basierte Ansatz ist, wie bereits erwähnt, für die Kooperation und Interaktion des mobilen Roboters mit Partnern vorbereitet. Hierzu ist vorrangig die Umsetzung des *kognitiven Operators* in der Cloud zu nennen (siehe Abbildung 4.4). In einer Multi-Roboter Kooperation dient der gemeinsam genutzte Cloud-basierte *kognitive Operator* als Austauschplattform für die einzelnen Teilnehmer. Hier können Synergien genutzt werden, indem sowohl Daten ausgetauscht, (wie beispielsweise das Teilen von Sensordaten mit anderen Robotern) als auch Aufgaben verteilt werden. Dieser Ansatz ist auch in „Cloud robotics: Current status and open issues“ [119] beschrieben, worin gezeigt wird, dass das Cloud Computing besonders gut für die Kooperation von mehreren Robotern über eine geteilte Cloud Plattform geeignet ist und die Multi-Roboter Kooperation durch das Cloud Computing potenziell stark verbessert wird. Weiterhin ist die Kopplung auf mittlerer Ebene, also die Kopplung von *reflektorischen Operatoren* möglich. Ein möglicher Anwendungsfall ist die Koordination zweier Roboterarme auf je einem Roboter. Diese Koordination muss ggf. in harter Echtzeit realisiert werden, beispielsweise wenn die beiden Roboterarme gemeinsam einen Gegenstand anheben. Ein weiteres Szenario ist die Steuerung eines mobilen Roboters mittels einer GUI, bei der z. B. das Auswählen einer Notabschaltung sofort umgesetzt werden muss. Durch die strenge Hierarchie und Abtrennung der einzelnen Ebenen kann sichergestellt werden, dass die Interaktion auf kognitiver oder reflektorischer Ebene keine Auswirkung auf die *Controller* der Roboter hat und diese weiterhin harte Echtzeitbedingungen erfüllen können. Gleiches gilt für die Interaktion eines Menschen mit einem mobilen Roboter. Benutzer*innen oder Entwickler*innen können auf kognitiver Ebene über die Cloud mittels eines HMI mit dem Roboter agieren oder über die vorhandene Sensorik auf der *Controller*-Ebene mit diesem interagieren.

4.2.7 Softwareentwicklung

Die Softwareentwicklung von komplexen technischen Systemen ist eine Herausforderung für die Entwickler*innen solcher Systeme. Das dargestellte Modell bedarf eines höheren Implementierungsaufwands als ein monolithisches System, da nicht ein Rechner, sondern mehrere Rechner und die Kommunikation zwischen den einzelnen Rechnern implementiert werden müssen. Der Komplexität wird durch die klar definierte Architektur und dem modularen Aufbau des Systems begegnet. Die Gesamtfunktionalität wird auf einzelne, nahezu unabhängige Komponenten verteilt und kann entsprechend unabhängig voneinander, mit Ausnahme der Kommunikationsschnittelle(n), implementiert und getestet werden. Durch die Realisierung als verteiltes System können die Rechner einzeln in Hardware-in-the-Loop (HiL) Tests[19] getestet werden. Die Softwareentwicklung

[19] Die zu testende Hardware wird in eine simulierte Testumgebung eingebunden, in der die Ein- und Ausgänge der Hardware mit Testszenarien beschaltet werden und somit das Verhalten der zu testenden Hardware geprüft werden kann.

divergiert auf den drei Ebenen des OCM, auf allen drei Ebenen können diverse Bibliotheken und Frameworks eingesetzt werden.

Auf *Controller* Ebene wird harte Echtzeit gefordert, entsprechend liegt der Fokus bei der Entwicklung der Software für die *Controller* auf der effizienten und zuverlässigen Ausführung der Software mit definiertem Zeitverhalten, welche u. a. durch hardwarenahe Softwareentwicklung erreicht werden kann. Für hardwarenahe Programmierung können u. a. die Programmiersprachen *C* oder *Rust* [O33] eingesetzt werden. Laut Balasubramanian, Baranowski, Burtsev, Panda, Rakamarić und Ryzhyk [147] ist *C* bisher die typische Wahl bei hardwarenaher Programmierung, *Rust* ist hierfür jedoch mindestens genauso geeignet, da mit *Rust* ebenfalls sehr sichere und effiziente Software implementiert werden kann. Hardwarenahe Programmiersprachen bieten Entwickler*innen vergleichsweise weniger Unterstützung als beispielsweise *Java*, *C#* oder *C++*. Die Entwickler*innen müssen sich z. b. selbst um die Speicherallokation kümmern. Da die Komplexität jedoch auf dieser Ebene überschaubar ist (die Anbindung der Sensoren und Aktuatoren sowie die Implementierung von Algorithmen der Regelungstechnik stehen im Vordergrund), ist die Programmierung der *Controller* mit *C* oder *Rust* zu bevorzugen. Insbesondere für die Entwicklung und Simulation der Regler bietet sich die MDSE mit MATLAB Simulink an, aus dessen Modellen automatisch *C*-Quelltext generiert werden kann.

Die Softwareentwicklung des *reflektorischen Operators* hat weniger harte Echtzeitanforderungen. Im Vergleich zu den *Controllern* ist der *reflektorische Operator* jedoch weitaus komplexer, wodurch der Umfang der Softwareentwicklung höher ist, als auf der *Controller* Ebene. Um dieser Komplexität zu begegnen, wird in den *Umsetzungsempfehlungen für das Zukunftsprojekt Industrie 4.0* [123, S. 46 ff.] empfohlen, dass MBSE angewendet werden sollten. Zur MDSE kann beispielsweise die MATLAB Simulink Erweiterung Stateflow [O34] eingesetzt werden, welche laut [148, S. 477] besonders gut zur Implementierung von ereignis-orientierter Software geeignet ist. Hierbei werden Zustandsdiagramme grafisch modelliert und daraus Quelltext generiert. Dieser Quelltext kann hierbei, wie bei MATLAB Simulink, für verschiedene Zielplattformen generiert werden, sodass bei einem Wechsel des Rechners, die Modelle weiter genutzt werden können. Neben der MDSE ist der Einsatz von Hochsprachen, wie *C#* oder *C++* möglich, da die Effizienz der Software weniger wichtig ist, als bei der Softwareentwicklung der *Controller*.

Die Entwicklung der Cloud Anwendungen für den *kognitiven Operator* unterliegt keinen harten Echtzeitanforderungen und praktisch keinen Limitierungen durch Hardwareressourcen. In „The Making of Cloud Applications: An Empirical Study on Software Development for the Cloud“ [149] wird in einer systematischen Studie durch Umfragen erläutert, wie professionelle Softwareentwickler*innen Cloud Applikationen

implementieren. Ein Resultat der Studie ist, dass bei der Entwicklung von Cloud Anwendungen viele Tools, wie Docker [O35] und diverse Frameworks, wie .NET [O36] eingesetzt werden. Weiterhin wird auch in „A Systematic Review of Cloud Modeling Languages“ [150] gezeigt, dass die MBSE in Form von diversen Modellierungssprachen bei der Entwicklung von Cloud Anwendungen eingesetzt wird.

4.2.8 Zusammenfassung

In Tabelle 4.2 sind die Vor- und Nachteile einer verteilten Architektur gegenüber einem monolithischen System für mobile Roboter aufgrund der erläuterten Schlüsselmerkmale zusammengefasst. Im Vergleich zu einem einzelnen klassischen PC, kann ein verteiltes System aus mehreren Kleinstrechnern energieeffizienter umgesetzt werden, indem u. a. energieeffiziente Rechner ausgewählt und nicht benötigte Rechner im Sinne einer bedarfsgesteuerten Komponentennutzung ausgeschaltet werden. Die Leistungsfähigkeit ist durch die Integration der Cloud ebenfalls höher als bei einem einzelnen lokalen Rechner. In Bezug auf die Modularität, Zuverlässigkeit und Sicherheit (Safety) ist ein verteiltes System, u. a. aufgrund der Entkopplung der einzelnen Rechner, einem monolithischen System vorzuziehen. Ein cloudbasierter *kognitiver Operator* kann weniger sicher (Security) sein, als ein einzelner Rechner, falls dessen Schnittstellen nach außen abgesichert sind. Der Implementierungsaufwand eines verteilten Systems ist vergleichsweise hoch, da nicht nur ein Rechner, sondern mehrere Rechner und eine Kommunikation zwischen diesen realisiert werden muss.

	Energieeffizienz	Leistungsfähigkeit	Modularität	Sicherheit (Safety)	Sicherheit (Security)	Implementierungsaufwand
Verteiltes System	✓	✓	✓	✓		
Monolithisches System					✓	✓

✓: Vorteil(e) für

Tabelle 4.2: Vor- und Nachteile eines verteilten Systems für mobile Roboter

In Tabelle 4.3 sind analog die Vor- und Nachteile der Umsetzung des *kognitiven Operators* in der Cloud dargestellt. Die Optimierung und das Lernen in der Cloud macht es möglich, Methoden einzusetzen, welche viel Rechenleistung benötigen. Zudem können eine Optimierung, Analyse und Simulation des Systems aufgrund der Daten in der Cloud erfolgen, ohne dass der Roboter dazu benötigt wird. Die Multi-Roboter Kooperation und die Mensch-Roboter Interaktion werden durch die Umsetzung eines cloudbasierten *kognitiven Operators* erleichtert, da die Infrastruktur zum Austausch von Informationen zwischen Robotern oder Mensch und Roboter zur Verfügung steht. Die Öffnung des Systems in die Cloud birgt die Gefahr, die Sicherheit (Security) des Systems zu verringern, sodass hier die beschriebenen Gegenmaßnahmen getroffen werden sollten. Die Energieeffizienz des Cloud-basierten Ansatzes kann größer als bei der lokalen Umsetzung des *kognitiven Operators* sein, da bei der lokalen Umsetzung ein leistungsstärkerer Rechner benötigt wird, welcher potenziell mehr Energie benötigt. Weitere Nachteile der Umsetzung des *kognitiven Operators* in der Cloud sind der erhöhte Implementierungsaufwand und die Komplexität des mobilen Roboters. Durch das Cloud Computing wird eine weitere Abstraktionsebene in das System integriert, welche weitere Kommunikationsstrategien und z. B. Cloud-Frameworks benötigt.

	Optimierung/ Lernen	Multi-Roboter Kooperation	Mensch-Roboter Interaktion	Sicherheit (Security)	Energieeffizienz	Implementierungsaufwand
Cloudbasierter *kognitiver Operator*	✓	✓	✓		✓	
Lokaler *kognitiver Operator*				✓		✓

✓: Vorteil(e) für

Tabelle 4.3: Vor- und Nachteile des cloudbasierten *kognitiven Operators*

4.3 Systemkomponenten

Ein typischer mobiler Roboter verfügt über die in Abbildung 4.8 dargestellten Komponenten, wobei der *reflektorische Operator+* eine optionale Komponente darstellt und der cloudbasierte *kognitive Operator* nicht zwingend zum Betrieb des mobilen Roboters erforderlich ist.

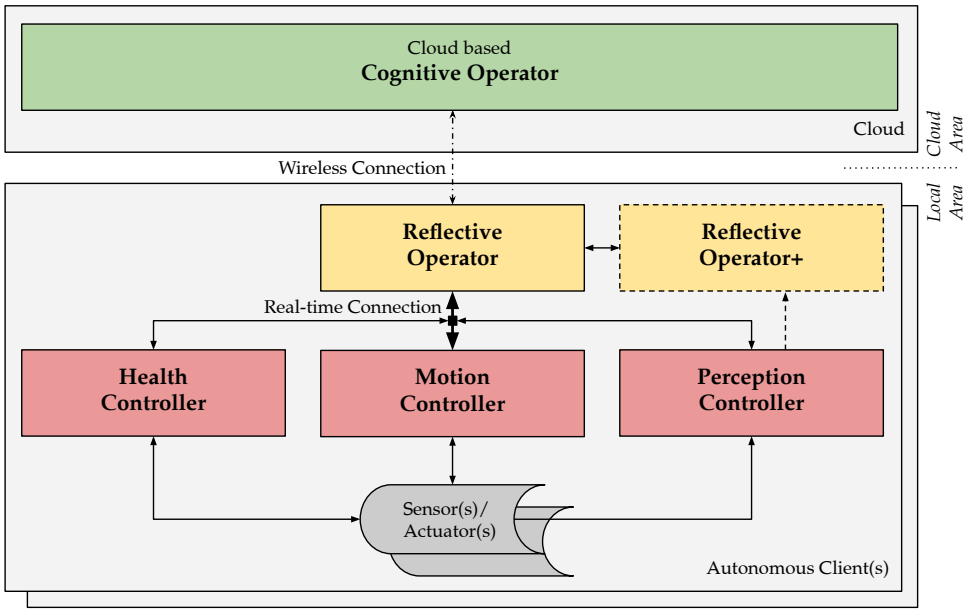


Abbildung 4.8: Systemkomponenten

4.3.1 Motion Controller

Auf *Controller*-Ebene sei zunächst der *Controller* zur Verarbeitung von Sensoren und Aktuatoren zur Bewegung des mobilen Roboters zu nennen, welcher im Folgenden als *Motion Controller* bezeichnet wird. Diese Sensoren und Aktuatoren sind typischerweise als Basis eines Roboters vorhanden, d. h. diese werden zum Betrieb des Roboters immer benötigt und sind auch bei kommerziell verkauften Robotern immer existent. Typischerweise werden Sensoren zur Hinderniserkennung benötigt, beispielsweise Ultraschall- oder Infrarotsensoren und Aktuatoren zum Betreiben der Motorik, meist Motortreiber. Die Systemkomponente *Motion Controller* ist softwaretechnisch nicht komplex und benötigt wenig Rechenleistung. Nichtsdestotrotz kommt dem *Motion Controller* eine besondere

Bedeutung zu, da dieser mit der Kontrolle der Motoren in Bezug auf die Sicherheit (Safety) besonders kritisch betrachtet werden muss. Oft verfügt der *Motion Controller* zudem über weitere Sensoren, welche mit der Bewegung des Roboters zu tun haben, beispielsweise einem Gyroskop. Zur Bewältigung der genannten Aufgaben ist kein besonders leistungsstarker Rechner notwendig. Hier kann beispielsweise ein STM32-basierter Kleinstrechner zur Anwendung kommen, welcher die notwendigen Echtzeitbedingungen durch den Einsatz von FreeRTOS oder ChibiOS erfüllen kann (siehe Tabelle 4.1).

4.3.2 Health Controller

Der sogenannte *Health Controller*[20] erweitert das verteilte System um einige Funktionen, beispielsweise die Sicherheit (Safety) des mobilen Roboters betreffend. Die Hauptaufgabe des *Health Controllers* ist die Erfassung des Zustands des mobilen Roboters und der Umgebung in der sich dieser befindet. Hierzu verfügt der *Health Controller* über zahlreiche Sensoren zur Ermittlung des internen Systemzustands. Dazu gehören Sensoren zur Messung des Zustands der Energieversorgung, sowie der aktuelle Leistungsbedarf des Roboters. Weiterhin kann es notwendig sein, Temperatursensoren zu verwenden, um beispielsweise die Temperatur der Motortreiber zu überwachen. Sensoren zur Ermittlung des Zustands der Umgebung sind, je nach Anwendung z. b. Magnetometer, Temperatursensoren, Feuchtigkeitsmesser oder Helligkeitssensoren. Als vorteilhaft hat sich während des Experimentes (siehe Kapitel 5) herausgestellt, dass der *Health Controller* über Aktuatoren, beispielsweise Relais, zum Ein- und Ausschalten einzelner Komponenten des verteilten Systems verfügt. Indem unnötige Komponenten ausgeschaltet werden, können diese Relais dabei helfen, Energie zu sparen oder Schaden am Akkumulator zu verhindern. Die Verarbeitung der Daten des *Health Controllers* ist simpel und benötigt wenig Rechenleistung, oft jedoch viele GPIO-Schnittstellen zur Anbindung der Sensoren und Aktuatoren. Hier bietet sich der Einsatz eines Kleinstrechners aus der Arduino Familie an, welcher viele Schnittstellen bietet und zudem wenig Energie benötigt (siehe Tabelle 4.1).

4.3.3 Perception Controller

Ein weiterer und typischer *Controller* für mobile Roboter ist die nachfolgend als *Perception Controller*[21] bezeichnete Komponente. Dieser bindet Sensoren zur Wahrnehmung und Überwachung der Umgebung ein, welche nicht unmittelbar an den *Motion Controller* angeschlossen werden. Typisch hierfür ist die Einbindung von Kamerasensoren. Hier werden typischerweise RGB[22]-Mono-Kameras oder Tiefenkameras mit 3D-Informationen eingesetzt. Letztere sind stark abhängig von Budget und Anwendungsfall. Preisgünstig und trotzdem einer RGB-Mono-Kamera technisch überlegen, sind Tiefenkameras, welche das

[20] dt. Gesundheitszustand/ Befinden
[21] dt. Wahrnehmungs-Controller
[22] dt. Rot Grün Blau (Farbraum)

Triangulationsverfahren [151] einsetzen. Hierbei werden Tiefenkameras, wie Microsofts Kinect [152] eingesetzt, welche mit zwei Linsen (Infrarot-Sender und -Empfänger) ausgestattet sind, aus deren Versatz die Entfernung eines Bildpunktes berechnet werden kann. Solche Kameras senden dazu ein Infrarotmuster aus und können somit auch bei Dunkelheit eingesetzt werden. Weitere übliche Methoden sind Time of Flight (ToF)[23] Sensoren und Laserkameras, wie LiDAR-Sensoren. Die softwaretechnische Anbindung bzw. Verarbeitung der Sensoren kann hierbei einen Sonderfall im Vergleich zu den anderen *Controllern* darstellen. Wenngleich die Auswertung und Vorverarbeitung von RGB Signalen vergleichsweise simpel ist, beispielsweise durch die Nutzung des Frameworks OpenCV, bedarf es zur Vorverarbeitung von 3D-Daten hohe Rechenleistung, insbesondere wenn diese in harter Echtzeit erfolgen soll. Somit ist die Hardware des *Perception Controllers* typischerweise leistungsstark. Der *Perception Controller* könnte beispielsweise mit einem Raspberry Pi unter Einsatz eines Echtzeitbetriebssystems realisiert werden (siehe Tabelle 4.1).

4.3.4 Weitere Controller

Die Einbindung weiterer *Controller* in das verteilte System ist durch den modularen Ansatz einfach möglich. Die Auswahl weiterer Sensoren und Aktuatoren ist abhängig vom Anwendungsfall und dem Aufbau des mobilen Roboters. Sensoren, die keine besonderen Anforderungen an die Hardware Ressourcen haben, können den bereits erläuterten *Controllern* zugeordnet werden. Komplexere Komponenten, wie beispielsweise ein mechanischer Arm, sollten in einem dafür vorgesehenen neuen *Controller* realisiert werden.

4.3.5 Reflektorischer Operator und reflektorischer Operator+

Ein *reflektorischer Operator* ist zwingend erforderlich, um die Daten der *Controller* zu verarbeiten und diese zu koordinieren. Der *reflektorische Operator* verfügt hierbei über keine weiteren (externen) Sensoren und ist nicht mit Aktuatoren des Systems verbunden. Bei der Realisierung des *reflektorischen Operators* ist darauf zu achten, dass dieser auf viele Ereignisse reagiert, welche durch diverse *Controller* ausgelöst werden. Diese werden parallel bearbeitet. Diese Parallelisierung sollte die Zielhardware unterstützen. Dies ist beispielsweise bei einem Raspberry Pi SBC der Fall, welcher über vier Rechenkerne zur parallelen Ausführung von Software verfügt (siehe Tabelle 4.1).

Zudem ist es möglich, die Aufgaben des *reflektorischen Operators* auf mehr als eine Recheneinheit zu verteilen. Hierbei kann es sinnvoll sein, unterschiedliche Typen von Recheneinheiten als *reflektorische Operatoren* einzusetzen und somit eine heterogene Verteilung (siehe HMP in Kapitel 2.2.5) zu erhalten, indem die Stärken der einzelnen

[23] dt. Laufzeitverfahren

Recheneinheiten im Sinne von Koprozessoren oder Hardwarebeschleunigern genutzt werden. So kann die Kombination aus einer ARM basierten CPU und einem FPGA sinnvoll sein, da ein FPGA deutlich energieeffizienter und schneller Daten verarbeiten kann, wenn dies parallel erfolgt. Hierbei sei beispielsweise eine umfassende Bildverarbeitung parallel zum *Perception Controller* zu nennen, dessen Verarbeitung durch das massive Parallelisieren beschleunigt werden kann, wie Honegger, Oleynikova und Pollefeys in [153] oder Rethinagiri, Palomar, Moreno, Unsal und Cristal in [154] beschreiben. Zudem kann der *reflektorische Operator+* zur Ausführung von Methoden des maschinellen Lernens eingesetzt werden, bei denen typischerweise parallele Berechnungen durchgeführt werden [155].

Diese Systemkomponenten, welche den *reflektorischen Operator* unterstützen und die Datenverarbeitung des *reflektorischen Operators* somit beschleunigen, werden im Folgenden als *reflektorischer Operator+* bezeichnet. Der *reflektorische Operator+* ist eine optionale Komponente, welche nicht in jedem Anwendungsfall und typischerweise nicht dauerhaft benötigt wird. Diese Komponente ist entweder eine vollständig unabhängige Hardware-Komponente oder Teil eines hybriden, eingebetteten Systems, welches über ein HMP-Modul aus CPU/ FPGA oder CPU/ GPU verfügt, wie beispielsweise NVIDIAs Jetson TX2. Hier würde der *reflektorische Operator* auf der ARM-basierten CPU und der *reflektorische Operator+* auf der GPU des Jetson TX2 realisiert werden. Die Onboard-GPU einer CPU wird hierbei nicht als *reflektorischer Operator+* angesehen, da diese zwar zur Beschleunigung von parallelen Aufgaben beitragen kann, jedoch keine eigenständige Einheit darstellt.

Welche der folgenden Kombinationen aus CPU/ GPU, CPU/ FPGA oder CPU/ ASIC für den *reflektorischen Operator*/ *reflektorischen Operator+* eingesetzt wird, hängt von der individuellen Auswahl der Hardware und dem individuellen Anwendungsfall des mobilen Roboters ab. Thomas, Howes und Luk erläutern in [156] die Nutzung von CPU, GPU und FPGA am Beispiel einer massiv parallelen Generierung von Zahlen. Vansteenkiste stellt CPU, FPGA, ASIC und GPU in [157, S. 4] in einer Tabelle gegenüber (siehe Tabelle 4.4). Mobile Roboter profitieren besonders beim Energieverbrauch, welcher bei FPGA und ASIC vergleichsweise gering ist. Die Nutzung eines ASIC ist nur sinnvoll, wenn der zu entwickelnde mobile Roboter für die Massenproduktion geplant ist, damit die hohen Kosten für die Entwicklung, sowie die lange Markteinführungszeit sich bei hoher Stückzahl und längerer Produktionszeit rentieren. Prototypen und kleine Stückzahlen werden bestenfalls mit einem FPGA als *reflektorischer Operator+* implementiert, falls dieser für die Verarbeitung der Daten des *reflektorischen Operators* notwendig ist.

	CPU	**FPGA**	**ASIC**	**GPU**
Beispiel	*ARM Cortex-A9*	*Virtex Ultrascale XCVU440*	*Bitfury 16nm*	*NVIDIA Titan X*
Flexibilität während Entwicklung	Mittel	Hoch	Sehr hoch	Gering
Flexibilität nach Entwicklung[1]	Hoch	Hoch	Gering	Hoch
Parallelisierung[2]	Gering	Hoch	Hoch	Mittel
Leistungsfähigkeit	Gering	Mittel	Hoch	Mittel
Energieverbrauch	Hoch	Mittel	Gering	Hoch
Entwicklungskosten	Gering	Mittel	Hoch	Gering
Produktionskosten[3]	*Keine*	*Keine*	Hoch	*Keine*
Kosten pro Einheit	Mittel	Hoch	Gering	Hoch
Markteinführungszeit	Gering	Mittel	Hoch	Mittel
Eignung als *reflektorischer Operator+*	*Referenz*	✓	✓ (bei Massenproduktion)	Eingeschränkt

[1] U. a. um Fehler zu korrigieren und neue Funktionen zu implementieren
[2] Für geeignete parallele Anwendungen
[3] Produktionskosten für den ersten Chip

Tabelle 4.4: Eignung von CPU, GPU, FPGA und ASIC als *reflektorischer Operator+* (basiert auf [157, S. 4])

4.3.6 Kognitiver Operator

Wie bereits erläutert, wird das verteilte System um einen *kognitiven Operator* erweitert. Dieser wird virtuell eingebunden, die Aufgaben des *kognitiven Operators* werden in der Cloud verarbeitet. Dies spart eine weitere Recheneinheit auf dem mobilen Roboter und hat den Vorteil, dass in der Cloud eine hohe Rechenleistung zur Verfügung steht, ohne dass lokal, außer zur Kommunikation mit der Cloud, weiterer Energieverbrauch entsteht. In Tabelle 4.5 werden die drei Cloud Kategorien, SaaS, PaaS und IaaS verglichen, um zu prüfen, auf welcher Schicht der *kognitive Operator* aufsetzen sollte. [158] zeigt mit der Tabelle, dass prinzipiell sowohl PaaS, als auch IaaS für die Umsetzung des *kognitiven Operators* geeignet sind, da beide die Kontrolle über die Anwendung und die Daten liefern. IaaS bietet die größte Kontrolle für den Nutzer, die Implementierung auf dieser Ebene ist jedoch mit viel Aufwand verbunden, da beispielsweise selbst das Betriebssystem verwaltet wird. SaaS ist zwar am einfachsten zu handhaben, ist jedoch auch am wenigsten flexibel und liefert nicht ausreichend Kontrolle über die zu implementierende Anwendung. PaaS bietet den besten Kompromiss aus Aufwand, Kontrolle und Flexibilität und ist deshalb besonders gut als Plattform für den *kognitiven Operator* geeignet.

	SaaS	**PaaS**	**IaaS**
Servicetyp	Anwendung mit spezifischer Funktion	Anwendung inkl. Umgebung	Basis Berechnungen, Speicherung, Netzwerk Ressource
Benutzer*in hat Kontrolle über	-	Anwendung, Daten	Anwendung, Daten, Laufzeit, Middleware, Betriebssystem,
Provider hat Kontrolle über	Anwendung, Daten, Laufzeit, Middleware, Betriebssystem, Virtualisierung, Server, Speicher, Netzwerk	Laufzeit, Middleware, Betriebssystem, Virtualisierung, Server, Speicher, Netzwerk	Virtualisierung, Server, Speicher, Netzwerk
Flexibilität	Niedrig	Mittel	Hoch
Aufwand für Entwickler*innen	Niedrig	Mittel	Hoch
Ebene	Nutzer*innen Ebene	Entwickler*innen Ebene	IT Ebene
Eignung als Plattform für *kognitiven Operator*	×	✓	✓ (Mehraufwand)

Tabelle 4.5: Eignung von SaaS, PaaS und IaaS als Platform für den *kognitiven Operator* (basiert auf [158])

4.4 Schnittstellen

Es gibt zahlreiche Möglichkeiten, die Komponenten eines verteilten Systems miteinander zu verbinden. Schöpping, Korthals, Hesse und Rückert erläutern beispielsweise in [40] Verbindungstypen und Protokolle zur Nutzung von modularen Echtzeit-Systemen in der Robotik. Nachfolgend werden die unterschiedlichen Verbindungstypen auf den verschiedenen Ebenen, bzw. an den drei Kreisen (motorisch, reflektorisch und kognitiv; siehe Abbildung 4.4) des OCM basierten Ansatzes in Abbildung 4.9 dargestellt und erläutert. Das konzeptionelle Modell für eine Systemarchitektur für mobile Roboter sieht eine strenge Trennung der Schichten vor, sodass nicht alle Komponenten aller Schichten direkt miteinander verbunden sind.

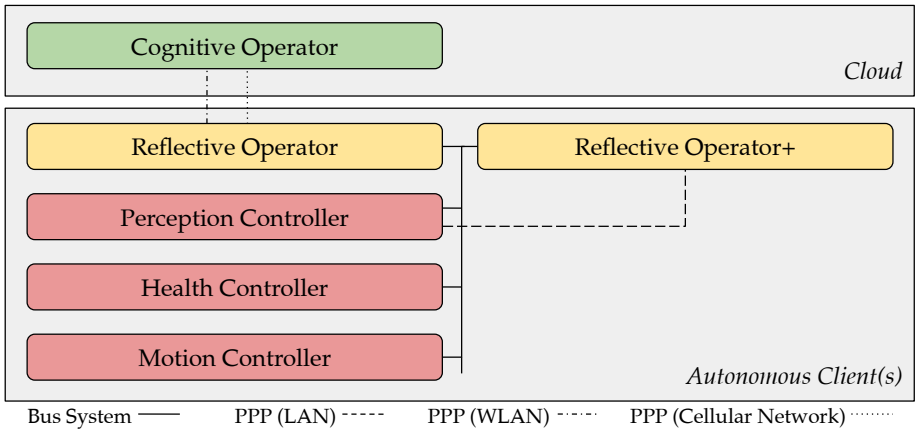


Abbildung 4.9: Schnittstellen im Architekturkonzept

4.4.1 Motorischer Kreis

Die Sensoren und Aktuatoren werden ausschließlich innerhalb des motorischen Kreises an den dazu vorgesehenen *Controller* angeschlossen. Hierbei gibt die Ausstattung der Hardware der Sensoren und Aktuatoren i. d. R. die Verbindungsart vor. Während einfache Aktuatoren, wie Taster direkt über GPIO Pins angeschlossen werden können, werden Sensoren, welche große Datenmengen zur Verfügung stellen, über Verbindungen mit hoher Bandbreite angeschlossen, wie beispielsweise via USB. Sollte ein Sensor oder Aktuator verschiedene Anschlussmöglichkeiten bereitstellen, so sollte stets jene Variante gewählt werden, welche die harten Echtzeitanforderungen der *Controller* am besten erfüllen kann. Weiterhin ist sicherzustellen, dass die Sensoren und Aktuatoren nicht durch andere Komponenten im System erreichbar sind. Falls Datenbus-Systeme für Sensoren oder Aktuatoren eingesetzt werden müssen, weil ein Sensor oder Aktuator nur über Bus-Schnittstellen verfügt, so

sollte dieser Datenbus lokal implementiert werden und nicht mehrere *Controller* verbinden, d. h. der motorische und reflektorische Kreis sollten nicht miteinander verbunden werden. Die *Controller* empfangen Daten der Sensoren typischerweise quasi-kontinuierlich, d. h. der Empfang erfolgt äquidistant mit kurzen Abtastintervallen. Diese Abstände sind ein Teil der Konfiguration der *Controller* und können in unterschiedlichen Modi variieren. Die Abtastrate von Abstandssensoren kann so beispielsweise an die Geschwindigkeit des mobilen Roboters angepasst werden, indem diese bei langsamer Fortbewegung weniger häufig abgefragt werden, als bei schnellerem Tempo. So ist mehr Bandbreite (Kommunikation) und ggf. Rechenleistung (CPU) für priorisierte Nachrichten verfügbar.

4.4.2 Reflektorischer Kreis

Eine Real-Time Communication (RTC)[24] ist auch zwischen den *Controllern* und dem *reflektorischen Operator* vorgesehen. Da der *reflektorische Operator* auf den Eingang neuer Informationen der *Controller* oder des *kognitiven Operators* reagiert, kann der reflektorische Kreis der Systemarchitektur dem Prinzip der ereignisgesteuerten Architektur (EDA) zugeordnet werden (Naumann [32, S. 29] bezeichnet dieses als ereignisorientierte Kommunikation). Typischerweise werden die Komponenten hierbei durch einen Ereignis-Bus verbunden. Da das konzeptionelle Modell aus mehreren *Controllern* besteht, ist die Anbindung der *Controller* an den *reflektorischen Operator* mittels eines Bus-Systems aus praktischer Sicht sinnvoll, weil die Anzahl der physischen Leitungen somit minimiert werden kann. Echtzeitfähige Nachrichten können beispielsweise auf dem CAN-Bus [159] bidirektional verschickt werden, wie es beispielsweise in „RTCAN: A real-time CAN-bus protocol for robotic applications“ [160] erläutert wird. Mögliche Alternativen zur weit verbreiteten CAN-Schnittstelle können aus dem Automobilsektor adaptiert werden. Hierzu zählen u. a. LIN (bei einer kleinen Anzahl von Sensoren/ Aktuatoren) [161] oder FlexRay [162]. Hierbei ist der Aufwand bezüglich der Kosten und der Implementierungszeit zur Realisierung dieser Bus-Systeme auf mobilen Robotern jedoch nicht zu unterschätzen.

Da es bei asynchronen Übertragungen, wie bei CAN und ggf. FlexRay zu Kollisionen von Nachrichten auf dem Bus kommen kann, muss sichergestellt werden, dass die Nachrichten mit der höchsten Priorität auch als Erstes ankommen. Höchste Priorität haben beispielsweise Aktuatorstellwerte, damit der mobile Roboter u. U. mit möglichst wenig Verzögerung stoppen kann. Niedrige Priorität haben beispielsweise Sensordaten, die nicht unmittelbar zur Steuerung des mobilen Roboters benötigt werden. Niedrige Priorität haben zudem Sensordaten, deren dynamisches Verhalten vergleichsweise langsam ist. So kann sich z. B. die Wärme nicht so schnell verändern, wie beispielsweise die Entfernung zu einem Hindernis bei einem sich schnell bewegenden mobilen Roboter. Auf höherer Abstraktionsebene haben Nachrichten zwischen *Controller* und *reflektorischem Operator*

[24] dt. Echtzeitkommunnikation

immer eine höhere Priorität als Nachrichten zwischen dem *reflektorischen Operator* und dem *reflektorischen Operator+*.

Zur Realisierung einer Priorisierung kann das CAN-Protokoll erweitert werden. Bei CAN werden Nachrichten mit einem niedrigen Identifikator (ID) einer höheren ID vorgezogen. Hierzu wird beim CAN-Bus die bitweise Arbitrierung benutzt. Migliavacca, Bonarini und Matteucci nutzen dieses, um drei Stufen zur Priorisierung einzuführen: *hard real-time messages (HRT)*, *soft real-time messages (SRT)* und *non real-time messages (NRT)* [160]. Im Kapitel *Experimentelle Umsetzung der Systemarchitektur für mobile Roboter* (5.2.1) wird dieses Konzept erweitert, sodass diese Priorisierung weitere Abstufungen und dynamische Veränderungen der Priorisierung via CAN erlaubt. Alternativ haben Leen und Heffernan in [163] mit *TTCAN (Time-triggered CAN)* ein alternatives CAN Protokoll vorgestellt, welches besonders gut für periodische Kommunikation geeignet ist, da das Protokoll die zeitlichen Abläufe überwacht und Kollisionen von Nachrichten vermeidet. *FTTCAN (Flexible time-triggered CAN)* von Almeida, Pedreiras und Fonseca [164] verfolgt einen ähnlichen Ansatz, wenngleich das TTCAN-Protokoll weiter verbreitet und im ISO-Standard 11898-4:2004 [165] genormt ist.

Zusammenfassend sind die Eigenschaften der möglichen Bus-Systeme für die Kommunikation innerhalb des motorischen Kreises des konzeptionellen Modells in Tabelle 4.6 dargestellt. Hierbei wird deutlich, dass der LIN-Bus aufgrund der vergleichsweise geringen Bandbreite im Anwendungsgebiet der mobilen Roboter nur bei vergleichsweise einfachen Robotern eingesetzt werden sollte. FlexRay hingegen bietet die meisten Funktionen und die höchste Bandbreite, TTCAN und insbesondere CAN bieten ausreichend Bandbreite und Funktionen für das Übertragen der vorverarbeiteten *Controller*-Nachrichten und der Nachrichten des *reflektorischen Operators* im reflektorischen Kreis.

	LIN	CAN	TTCAN	FlexRay
Nachrichten	Deterministisch, Statisches Scheduling	Ereignis-gesteuert	Ereignis- und zeit- gesteuert	Ereignis- und zeit- gesteuert
Synchronisation	Globale Referenzzeit	Prioritäten basierte Arbitrierung	Globale Referenzzeit	Globale Referenzzeit
Erneutes Übertragen	Eingeschränkt	Wird unterstützt	Wird unterstützt	Wird unterstützt
Fehler Management	CRC Feld, ID Paritätsprüfung, Diagnose Nachrichten	CRC Feld, Bit Überwachung, Bitstopfen, Fehler Nachrichten, Überlast Nachrichten	CRC Feld, Bit Überwachung, Bit stuffing,[1] Fehler Nachrichten, Überlast Nachrichten	2 CRC Felder, (optional) Bus-Guardian[2]
Bandbreite	20 kBit/s max	10 kBit/s - 1 MBit/s max[3]	10 kBit/s - 1 MBit/s max[3]	2 Kanäle: 5 MBit/s, 10 MBit/s
Physischer Aufbau	1 Leitung	2 Leitungen (geschützt vor Interferenzen)	2 Leitungen (geschützt vor Interferenzen)	2 Kanäle, 2 Leitungen (geschützt vor Interferenzen), (optional) Glasfaser
Eignung als Schnittstelle für den reflektorischen Kreis	Eingeschränkt	✓	✓	✓

[1] Füllt Nachrichten mit Füllbits auf (dt. Bitstopfen)
[2] Sorgt für einen kollisionsfreien Kommunikationsablauf
[3] In Abhängigkeit von der Leitungslänge

Tabelle 4.6: Eignung von Übertragungskanälen als Schnittstelle des reflektorischen Kreises (basiert auf [166])

Neben der Auswahl eines geeigneten Bus-Systems zur Übertragung der Nachrichten, ist das Nachrichtenmuster der Kommunikation zu spezifizieren. Da im reflektorischen Kreis die *Controller* und der *reflektorische Operator* asynchron arbeiten, sollte die Kommunikation entsprechend einem asynchronen Nachrichtenmuster folgen. Ein asynchrones

Nachrichtenmuster ist z. B. das Request-Reply[25] Muster. Hierbei fragt der Empfänger das Versenden von Daten am Sender an, welcher darauf mit dem Versenden der angefragten Daten reagiert. Ein weiteres Muster ist das *Publisher-Subscriber Prinzip* [18, S. 54 ff.], welches für den reflektorischen Kreis geeignet ist, in dem viele Sensordaten durch mehrere *Controller* zum *reflektorischen Operator* gesendet werden. Dieses Muster basiert auf der Grundidee, dass ein Publisher Daten oder Ereignisse unter Topics mittels einer Middleware verbreitet (siehe Abbildung 4.10). Subscriber können einzelne oder mehrere Topics abonnieren, werden über neue Daten benachrichtigt (Notification) und können auf deren Eingang reagieren. Im Gegensatz zur direkten Verbindung zweier Rechner zum Datenaustausch, werden Daten oder Ereignisse also nicht direkt an den Empfänger adressiert, sondern dem gesamten Umfeld angeboten. Einzelne Subscriber können unabhängig voneinander und unterschiedlich auf dasselbe Topic eines Publishers reagieren. Der Publisher kann hierbei entweder zyklisch neue Daten senden oder nur bei einer Änderung des Zustands eine Nachricht absetzen. Dieses macht das Anfragen von neuen Daten (Request-Reply) unnötig. Somit entfallen diese Anfragen auf dem Bus-System, wodurch laut Magnenat, Longchamp und Mondada [167] die Latenz zwischen der Messung eines Sensors und dem Empfang an der Zielkomponenten, die Bandbreite und die CPU Auslastung beim Sender und Empfänger verringert werden können. Das Publisher-Subscriber Prinzip ist Stand der Technik in der Robotik und wird beispielsweise in ROS eingesetzt oder bildet die Grundlage speziell für die Robotik angepasster Protokolle, wie z. B. dem embeddedRTPS (Real-Time Publish-Subscribe) Protokoll [168].

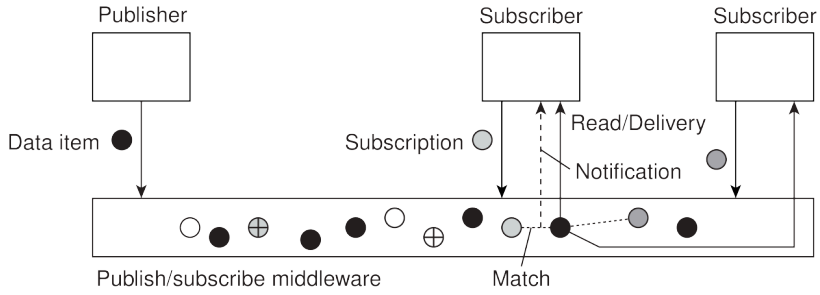


Abbildung 4.10: Prinzip zum Datenaustausch zwischen Publisher und Subscriber [19, S. 71]

In der vorgeschlagenen Systemarchitektur sind alle *Controller* und der *reflektorische Operator* mindestens als Publisher tätig. Bindet ein *Controller* auch Aktuatoren ein, so ist dieser zudem ein Subscriber. Prinzipiell abonniert der *reflektorische Operator* sehr viele Topics, auf dessen Eingang mit einer entsprechenden (parallelen) Datenverarbeitung reagiert wird. Das Publisher-Subscriber Prinzip ermöglicht es dem *reflektorischen Operator*, Nachrichten

[25] dt. Anfrage-Antwort

zu filtern, falls diese aktuell nicht relevant sind. In der eigenen CAN Erweiterung[26] ist es möglich, dass der *reflektorische Operator* jeden einzelnen Publisher konfigurieren kann, indem er die Frequenz der Publisher verändert, diese vollständig ein- und ausschaltet oder eine einmalige Datenübertragung anfordert (einmaliges Request-Reply).

4.4.3 Reflektorischer Operator+ Anbindung

Eine Besonderheit ist die Anbindung des optionalen *reflektorischen Operators+*, welcher außerhalb der drei Kreise (motorisch, reflektorisch und kognitiv) in das System eingebunden wird. Der *reflektorische Operator+* wird einerseits in das Bus-System eingebunden, andererseits ist eine Point-to-Point (PPP)[27] Verbindung zum *reflektorischen Operator* oder zu einem *Controller*, insbesondere zum *Perception Controller* notwendig. Mit der *reflektorischen Operator+* Anbindung an den Kommunikationsbus kann dieser Nachrichten an alle Komponenten der *Controller*- und *reflektorischen Operator*-Ebene abonnieren und verteilen. Da der *reflektorische Operator+* jedoch große Datenmengen parallel verarbeiten soll, reicht diese Verbindung i. d. R. nicht aus. Eine direkte Anbindung an jenen Sensor, welches die zu verarbeiteten Daten liefert, wird hierbei beispielsweise durch eine Local Area Network (LAN) Verbindung implementiert. Prädestiniert ist hierzu eine direkte PPP-Ethernet Kopplung zwischen *reflektorischen Operator+* und *Perception Controller* (siehe Abbildung 4.9), ggf. unter Berücksichtigung der Time-Sensitive Networking (TSN) Protokolle. TSN ermöglicht laut Farkas, Bello und Gunther [169] den Austausch des Datenverkehrs für zeitkritische Anwendungen, u. a. durch die Priorisierung der Daten. Der *Perception Controller* kann via Ethernet sein Video oder die Punktwolke aus einer 3D-Kamera vollständig an den *reflektorischen Operator+* streamen, welcher dann durch eine parallel-verarbeitende GPU-, FPGA- oder ASIC-Anwendung ausgewertet wird. Der *Perception Controller* wertet in dieser Variante weiterhin parallel selbst den Stream aus, um beispielsweise Hindernisse in Echtzeit zu erkennen. Die Verarbeitung des Videostreams durch den *reflektorischen Operator+* hat einen anderen Fokus, wie beispielsweise die Anwendung von rechenintensiven Filtern zur Vorhersage zukünftiger Hindernisse.

4.4.4 Kognitiver Kreis

Die Kopplung zwischen *reflektorischem* und *kognitivem Operator* verbindet den lokalen Klienten oder mehrere lokale Klienten, mit dem cloudbasierten Teil des verteilten Systems im kognitiven Kreis (siehe Abbildung 4.9). Da hierbei keine physische Verbindung möglich ist, erfolgt die Anbindung mit drahtlosen Methoden. Die Topologie der vorgestellten Systemarchitektur macht die Einbindung der Cloud möglich, da der kognitive Kreis keine harte Echtzeit benötigt. Die erste Datenverarbeitung findet lokal auf der Handlungsebene statt (*Controller* und *reflektorischer Operator*), d. h. im Falle des *reflektorischen Operators*

[26] wird nachfolgend in Kapitel 5.2.1 erläutert
[27] dt. Punkt-zu-Punkt

und im Sinne des *Edge Computings*, am Rande des (Cloud) Netzwerkes, sodass die zu versendende Datenmenge in die Cloud reduziert ist. Die Übertragung der Daten kann mit lokalen oder öffentlichen Funknetzen realisiert werden. Öffentliche Funknetze sind zwingend erforderlich, falls der mobile Roboter außerhalb eines lokalen Funknetzes eingesetzt wird. Falls jedoch ein lokales Funknetz dauerhaft und an jedem Ort des Einsatzgebietes des mobilen Roboters zur Verfügung steht, ist die Implementierung einer Anbindung an das öffentliche Funknetz nicht unbedingt erforderlich.

Lokales Funknetz

Das WLAN, bzw. Wi-Fi[28] ist hierbei in den meisten Fällen die zu wählende Technologie zur Einrichtung eines lokalen Funknetzes (siehe Tabelle 4.7). Zigbee, welches typischerweise für Netzwerke mit geringem Datenaufkommen wie Smart-Home Systemen eingesetzt wird, benötigt zwar wenig Energie, kann jedoch nur eine vergleichsweise geringe Datenrate aufbringen. Der Einsatz von Zigbee für das konzeptionelle Modell ist nur bedingt möglich. Eingesetzt werden kann es, falls nur ein Teil der aufkommenden Datenmenge übertragen und analysiert werden soll, beispielsweise zur Überwachung des mobilen Roboters. Die geringe Reichweite von 10 m eines einzelnen Gerätes spricht hierbei ebenfalls gegen den Einsatz, da ein typischer mobiler Roboter tendenziell einen größeren Aktionsradius hat. Zwar kann mit Zigbee eine Reichweite von 100 m erreicht werden, hierzu muss jedoch ein zusammenhängendes Netz an Zigbee Geräten (*Mesh*) eingerichtet werden. Auf Bluetooth und Bluetooth LE trifft in Bezug auf die Datenrate das Gleiche zu. Da hier jedoch auch die Reichweite mit 10 m gering ist, können Bluetooth und Bluetooth LE ebenfalls für die Überwachung eingesetzt werden, indem die Daten lokal gespeichert und bei Verfügbarkeit des Netzwerkes übertragen werden. WLAN hingegen ist als Schnittstelle in einem lokalen Funknetz des kognitiven Kreises geeignet, da hohe Datenraten und eine große Reichweite möglich sind, vorausgesetzt es findet sich ein freier WLAN Kanal im Frequenzband. Der vergleichsweise hohe Energieverbrauch ist ein Nachteil, welcher bei der Spezifikation der Energieversorgung beachtet werden muss.

[28] Wi-Fi ist eine zertifizierte Art eines WLANs (beispielsweise nach IEEE 802.11).

	Zigbee	Wi-Fi	Bluetooth	Bluetooth LE
Standard	IEEE 802.15.4	IEEE 802.11n	IEEE 802.15.1	IEEE 802.15.1
Übertagungs-frequenz	868 MHz, 915 MHz und 2.4 GHz	2.4 und 5 GHz	2.4 GHz	2.4 GHz
Datenrate	20, 40 und 250 kbs	800 Mbs max	1 bis 3 Mbs	1 Mbs
Reichweite	10 bis 100 m	100 m	10 m	10 m
Energie-verbrauch	Sehr niedrig	Hoch	Mittel	Sehr niedrig
Netzwerk-teinehmer	65.000	65	8	*Hängt von der Implementierung ab*
Sicherheit (Security)	128-bit AES	u. a. SSID Authen-tifizierung, AES	64 - 128-bit AES	128-bit AES
Vorteile	Energiesparend, Günstig, Zuverlässig, Skalierbar	Datenrate, Flexibilität	Datenrate	Energiesparend, Günstig
Eignung als Schnittstelle für den kognitiven Kreis (lokales Funknetz)	Eingeschränkt	✓	Eingeschränkt	Eingeschränkt

Tabelle 4.7: Eignung von Netzwerkprotokollen für die Schnittstelle des kognitiven Kreises im lokalen Netz (basiert auf [170])

Öffentliche Funknetze

Zur Anbindung des cloudbasierten *kognitiven Operators* im öffentlichen Funknetz existieren ebenfalls verschiedene Technologien. Hierzu ist die Verwendung des Mobilfunknetzes[29] erforderlich. Hier bietet sich der Einsatz von Mobilfunktechnologien und Protokollen an, welche aus dem IoT-Bereich bekannt sind, da diese wenig Energie umsetzen. Da die für IoT vorgesehenen Mobilfunkstandards heute weitestgehend für Sensornetze mit vergleichsweise niedriger Datendichte vorgesehen sind, ist es auch hier untypisch, den gesamten Datenbestand des autonomen Klienten hochzuladen, sondern nur die

[29] engl. Cellular network

relevanten und vorverarbeiteten Daten, die für die Planung und Optimierung des Systems notwendig sind (*Edge Computing*). In Tabelle 4.8 sind mögliche IoT-Mobilfunktechnologien gegenübergestellt.

	LoRA	**LTE-M1**	**NB-IoT**	**EC-GSM-IoT**	**5G**
Bandbreite	< 500 kHz	1.4 MHz	200 kHz	200 kHz	7 - 900 MHz
Abdeckung	< 15 km	< 11 km	< 35 km	< 35 km	< 15 km
Maximale Datenrate	300 kbps	< 10 Mbps (Download), < 5 Mbps (Upload)	< 170 kbps (Download), < 250 kbps (Upload)	< 140 kbps	> 1 Mbps
Energie-verbrauch	Niedrig	Niedrig	Niedrig	Niedrig	Niedrig
Eignung als Schnittstelle für den kognitiven Kreis (Mobilfunk)	✓	✓	✓	✓	✓ (voraus-sichtlich)

Tabelle 4.8: Eignung von IoT-Technologien als Schnittstelle für den kognitiven Kreis im Mobilfunknetz (Erweiterung von [171])

Long Range (LoRa), bzw. das LoRaWAN-Protokoll sind zur Anbindung von batteriebetriebenen IoT-Anwendungen entwickelt worden. Hierbei liegt der Fokus, wie bei allen nachfolgenden IoT Mobilfunklösungen, auf der Energieeffizienz [172]. Die maximale Datenrate ist vergleichsweise gering. Gleiches gilt für NarrowBand-IoT (NB-IoT), Long Term Evolution (LTE)-M und Global System for Mobile Communications (GSM)-IoT. NB-IoT [173] ist als Standard für IoT Geräte im LTE-Netz verankert. LTE-M ist flexibler und weniger energieeffizient als NB-IoT. GSM-IoT ist eine kostengünstigere Alternative für IoT Geräte, welche auf dem älteren 2G Mobilfunkstandard aufsetzt. Alle dargestellten Protokolle oder Technologien können für die Anbindung des reflektorischen Kreises eingesetzt werden, da keine Echtzeitfähig notwendig ist. Zukünftig kann die Nutzung von 5G [174] als Schnittstelle für den kognitiven Kreis in Betracht gezogen werden. Dieses scheint prädestiniert für den vorgesehenen Einsatz, da im 5G u. a. Bereiche für IoT Geräte mit sehr geringen Latenzen vorgesehen sind. Ebenfalls wird es im 5G voraussichtlich möglich sein, ganze Videostreams in Echtzeit hoch- und herunterzuladen. Aufgrund der aktuell geringen Verfügbarkeit wird 5G nicht weiter betrachtet.

Alternativ kann eine Mobilfunk-Lösung aus dem Endkundenbereich verwendet werden, wenngleich hier der Energieverbrauch ungleich höher ist. Prinzipiell ist die Implementierung einer (zusätzlichen) Schnittstelle via Mobilfunkstandards (öffentliches Funknetz) aufwendiger als die Einbindung von lokalen Netzen, wie z. B. via WLAN. Oft ist auch die benötigte Hardware teurer, zumal Kosten für die Übertragung von Daten im Mobilfunk zu tragen sind. Insbesondere bei der Massenproduktion von mobilen Robotern kann hingegen das lokale Funknetz zur Einbindung des cloudbasierten *kognitiven Operators* eingespart werden. Bei der Entwicklung von Prototypen bietet es sich an, die Kommunikation mit dem *reflektorischen Operator* im lokalen Netzwerk zu evaluieren und erst im zweiten Schritt die Mobilfunkschnittstelle zu implementieren.

Protokolle

Sowohl für die lokale als auch für die öffentliche Anbindung des *kognitiven Operators* sind Protokolle und Methoden zur Übertragung der Daten notwendig. Einen Überblick über typische Protokolle zur Anbindung von IoT Geräten wird in „A Survey of Communication Protocols for Internet of Things and Related Challenges of Fog and Cloud Computing Integration“ [175] dargestellt. In Tabelle 4.9 werden die Ergebnisse aus [175] zusammengefasst und die Eignung als Protokoll für die Schnittstelle des kognitiven Kreises bewertet.

REST ist hierbei nur bedingt geeignet, da die Methode des Abfragens und Antwortens (Polling) eingesetzt wird. Dies ist zwar prinzipiell möglich, führt aber bei großen Datenmengen oder einer großen Anzahl von Sensoren und Aktuatoren, wie es in der Robotik üblich ist, zu einer hohen Auslastung der Schnittstelle. Im lokalen Funknetz ist dieses zu vernachlässigen, für die Datenübertragung über das Mobilfunknetz ist REST in der erläuterten Anwendung nicht optimal. Das Constrained Application Protocol (CoAP) nutzt ebenfalls das Polling, wodurch die Anwendung ebenfalls nur bedingt empfohlen wird. Mit der Erweiterung des Protokolls zur Unterstützung der Publisher-Subscriber Methode, wie sie in [176] erläutert wird, kann CoAP jedoch ebenfalls eingesetzt werden. Message Queuing Telemetry Transport (MQTT) [177] und Advanced Message Queuing Protocol (AMQP) sind gut für die vorgesehene Anwendung geeignet. Beide setzen die Publisher-Subscriber Methode ein, nutzen TCP als Transportprotokoll und bieten damit die Sicherheitsmerkmale (Security) TLS und SSL. Der Energieverbrauch bzw. die Energieeffizienz sind ebenfalls, laut der Recherche in [175], für die vorgesehene Anwendung geeignet. Weiterhin unterstützen beide Protokolle drei Quality of Service (QoS)-Level[30]. Dizdarević, Carpio, Jukan und Masip-

[30] QoS beschreibt die Güte eines Kommunikationsdienstes. Mit den drei QoS Level in MQTT und AMQP kann konfiguriert werden, inwieweit der Erhalt einer Nachricht sichergestellt werden muss. Bei QoS Level 0 werden Nachrichten gesendet, ohne dass der Erhalt der Nachricht quittiert wird. Dieses Verhalten wird mit QoS Level 1 konfiguriert. QoS Level 2 quittiert den Erhalt doppelt mit einem vierfachen Handshake. Je höher der QoS Level gewählt wird, desto länger dauert das Abschließen einer Nachricht und desto größer ist die Garantie, dass die Nachricht korrekt übertragen wird.

Bruin [175] bevorzugen MQTT über AMQP, da dieses gut dokumentiert ist, stabil läuft und einfach zu implementieren ist. Naik [178] zeigt im Vergleich von MQTT zu AMQP ebenfalls, dass MQTT aktuell weiter verbreitet ist. Über die Protokolle Data Distribution Service (DDS), Extensible Messaging and Presence Protocol (XMPP) und HTTP/2.0 kann hier keine endgültige Entscheidung zur Eignung als Protokoll für die Schnittstelle des kognitiven Kreises getroffen werden, da u. a. keine Daten zum Leistungsbedarf vorliegen.

	REST HTTP	**MQTT**	**CoAP**	**AMQP**	**DDS**	**XMPP**	**HTTP / 2.0**
Polling	✓		✓	✓		✓	✓
Publisher-Subscriber		✓	✓	✓	✓	✓	✓
Transport	TCP	TCP	UDP	TCP	TCP/ UDP	TCP	TCP
QoS	-	3 Level	Einge-schränkt	3 Level	Umfang-reich	-	-
Sicherheit (Security)	TLS/ SSL	TLS/ SSL	DTLS	TLS /SSL	TLS/ DTLS/ DDS	TLS/ SSL	TLS/ SSL
Energie-verbrauch[1]	>MQTT >CoAP	<HTTP <CoAP <AMQP	<HTTP >MQTT	>MQTT	N/A	N/A	N/A
Favorit der Autor*innen von [175]	✓	✓					
Eignung[3]	Einge-schränkt	✓	Einge-schränkt[4]	✓	Offen	Offen	Offen

[1] Kein direkter Vergleich von allen dargestellten Protokollen verfügbar. Die Aussage ergibt sich aus der Kombination von mehreren Einzelvergleichen [175]
[2] TCP basierte Protokolle mit höherer Latenz als UDP basierte Lösungen [175]
[3] Als Protokoll für die Schnittstelle des kognitiven Kreises
[4] Mit Publisher-Subscriber Erweiterung

Tabelle 4.9: Eignung von IoT-Protokollen für die Schnittstelle des kognitiven Kreises

4.4.5 Multi-Roboter und Mensch-Roboter Schnittstellen

Wie bereits in Kapitel 4.2.6 gezeigt, ist das OCM basierte Architekturmodell für die Interaktion außerhalb des eigenen verteilten Systems geeignet. Hierbei kann es sich entweder um eine Interaktion mit Menschen (Mensch-Roboter Interaktion (H2M)) oder mit anderen mobilen Robotern (Multi-Roboter Kooperation) handeln. Die benötigte Schnittstelle ist dabei von der Art der Interaktion abhängig und kann im OCM auf den drei Ebenen (Controller, reflektorische oder kognitive Ebene) erfolgen (siehe Abbildung 4.11).

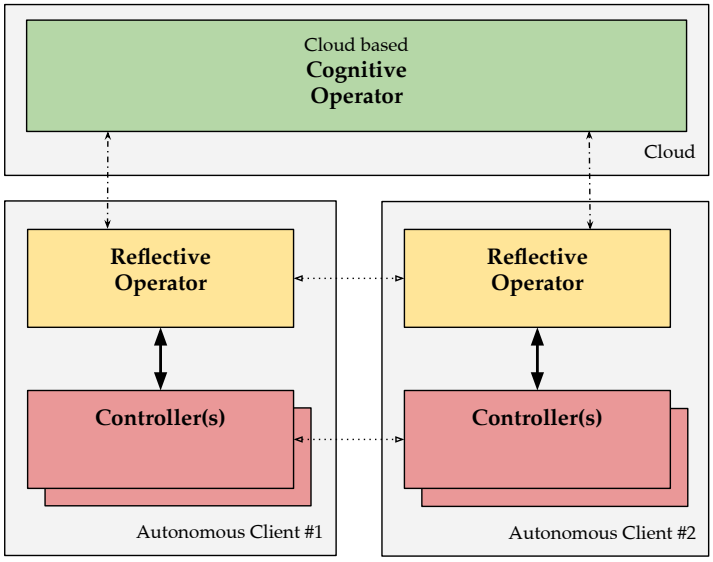


Abbildung 4.11: Interaktion von Robotern im Architekturkonzept

Kognitive Ebene

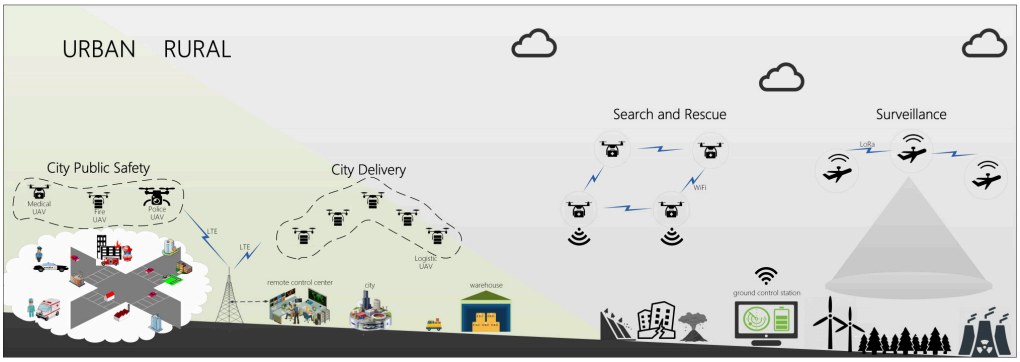
Der gemeinsame *kognitive Operator* ist für die Interaktion mit Menschen oder die Kooperation mit anderen Maschinen vorgesehen. Da der *kognitive Operator* keine harten Echtzeitbedingungen benötigt und die Aktion des Roboters vom *kognitiven Operator* entkoppelt ist, kann hierbei gefahrlos eine Interaktion mit Mensch oder Maschine durchgeführt werden. Bestenfalls nutzen mehrere mobile Roboter die gleiche Datenbasis in der Cloud, sodass ein Austausch von Informationen ohne weitere externe Schnittstelle möglich ist. Die Cloud selbst ist dabei die Schnittstelle. Die Kooperation mit Menschen (H2M) kann ebenfalls direkt über den cloudbasierten *kognitiven Operator* erfolgen. Hierzu greift der Mensch auf die Anwendung in der Cloud zu. Aus Sicherheitsgründen (Security) ist es empfohlen, den *kognitiven Operator* zu schützen und nicht öffentlich zugänglich zu

machen. [158] vergleicht hierbei die Varianten *öffentliche*, *private* und *Community*- Cloud. In der *öffentlichen* Variante kann die Anwendung zwar durch Authentifizierung eingeschränkt werden, prinzipiell ist dieses jedoch die sicherheitskritischste der drei Varianten, da der Server extern verwaltet wird. Bestenfalls wird die *private* Bereitstellung gewählt, falls ein Server im lokalen Netz zur Verfügung steht und ausreichend gegen Angriffe von außen geschützt ist. Nutzer*innen und Roboter müssen sich hierbei im gleichen Netz befinden, Abhilfe schafft hier ggf. die virtuelle Einwahl in das lokale Netz via VPN. Die Realisierung der Cloud auf einem *Community* Server ist für Projekte sinnvoll, bei denen der mobile Roboter von mehreren Partnern entwickelt wird, da hier der Server bei einem der Projektpartner steht und für die beteiligen Partner geöffnet wird.

Reflektorische Ebene

Typischerweise sollte die Koordination von mehreren Robotern über den *kognitiven Operator* ausreichen. Bei großen Schwärmen von Robotern oder nicht verfügbarem *kognitiven Operator* ist es jedoch zudem möglich, eine Schnittstelle zwischen den *reflektorischen Operatoren* von mehreren Robotern zu spezifizieren. Der Autor war hierzu an einer Veröffentlichung beteiligt, welche u. a. die Kommunikation in Multi-Roboter Kooperationen adressiert [A12]. Um die Implementierung einer weiteren Schnittstelle zu vermeiden, kann hierzu die Schnittstelle des kognitiven Kreises genutzt werden (siehe Tabelle 4.7 oder 4.8). Bei einer großen Anzahl von Robotern ist der Aufbau eines lokalen Funknetzes mit WLAN vorzuziehen. Hierbei ist es möglich, die Reichweite durch die Implementierung eines Mesh-Netzes zu erhöhen. Die Mesh-Protokolle open80211s, BATMAN, BATMAN Advanced und Optimized Link State Routing (OLSR) werden in [179] vorgestellt und verglichen. Muss, beispielsweise bei der Erkundung von sehr großen Gebieten mit mehreren Robotern, das Mobilfunknetz eingesetzt werden, so kann hier LTE oder LoRa eingesetzt werden, wie in [180] erläutert und mit WLAN am Beispiel von UAVs verglichen wird. Yuan, Jin, Sun, Chin und Muntean [180] zeigen vier Szenarien in urbanen[31] und ländlichen[32] Regionen und schlagen hierzu je eine Schnittstelle vor (siehe Abbildung 4.12).

[31] engl. Urban
[32] engl. Rural



City Public Safety: Öffentliche Sicherheit in der Stadt durch UAVs, welche via LTE kommunizieren.
City Delivery: Innerstädtische Lieferung durch UAVs, welche via LTE kommunizieren.
Search and Rescue: Rettungsmissionen im ländlichen Gebiet, welche untereinander via WLAN kommunizieren.*
Surveillance: Überwachung von ländlichen Regionen mit UAV Schwärmen, welche via LoRA kommunizieren.*
**Die hier dargestellte Basisstation würde im vorgeschlagenen konzeptionellen Model mit dem kognitiven Operator implementiert werden.*

Abbildung 4.12: Industrielle Anwendungen von UAV Schwärmen in urbanen und ländlichen Regionen [180]

Der *reflektorische Operator* kann weiterhin dazu benutzt werden, um mit dem Menschen in Interaktion zu treten. So ist es beispielsweise möglich, ein (berührungsempfindliches) Display an den *reflektorischen Operator* anzuschließen, welcher aktuelle Informationen des Roboters visualisiert. Dieses Display könnte zudem für Eingaben des Benutzers eingesetzt werden, da die Regelschleife der *Controller* nicht direkt beeinflusst werden kann. So könnten Nutzer*innen direkt am Roboter beispielsweise den Grad der Autonomie anpassen und selbst den Modus des Roboters ändern, z. B. den Energiesparmodus auswählen. Die Eingabe führt nicht zu einer unmittelbaren Änderung des Systems, sondern dient als Eingang für den *reflektorischen Operator*, welcher hieraus, in Abhängigkeit des Zustands des Systems, eine Aktion ausführen kann.

Controller Ebene

Eine Interaktion von mehreren Robotern auf der Ebene der *Controller* basiert auf den Sensoren und Aktuatoren der *Controller*, ohne dass hierzu eine direkte Verbindung zwischen den *Controllern* aufgebaut wird. Weiterhin ist es auch dem menschlichen Kooperationspartnern nicht möglich, direkten Einfluss auf die *Controller* zu nehmen. Dies würde den Sicherheitsbedingungen und der strengen Trennung der einzelnen Schichten widersprechen. Eine Interaktion auf *Controller* Ebene könnte beispielsweise das Erkennen oder Lokalisieren von Interaktionspartnern oder Kooperationssystemen durch die Kameras des *Perception Controllers*, wie es beispielsweise Wang und Olson [181] zeigen, sein. Hier wird jeder Roboter mit AprilTags[33] ausgestattet, welche von anderen Robotern eindeutig erkannt werden können. Aus dem Piktogramm der AprilTags kann weiterhin die Position des Tags in Bezug zur eingesetzten Kamera des *Perception Controllers* bestimmt werden, welche an den *reflektorischen Operator* weitergegeben und dort verarbeitet wird.

[33] AprilTags sind visuelle Hilfsmittel, welche ähnlich zu QR Codes typischerweise ausgedruckt oder mit einem Display angezeigt werden. Im Gegensatz zu QR Codes kann aus den AprilTags die Entfernung zu diesen abgeleitet werden, sodass AprilTags in der mobilen Robotik u. a. zur Lokalisierung verwendet werden.

5 Experimentelle Umsetzung der Systemarchitektur für mobile Roboter

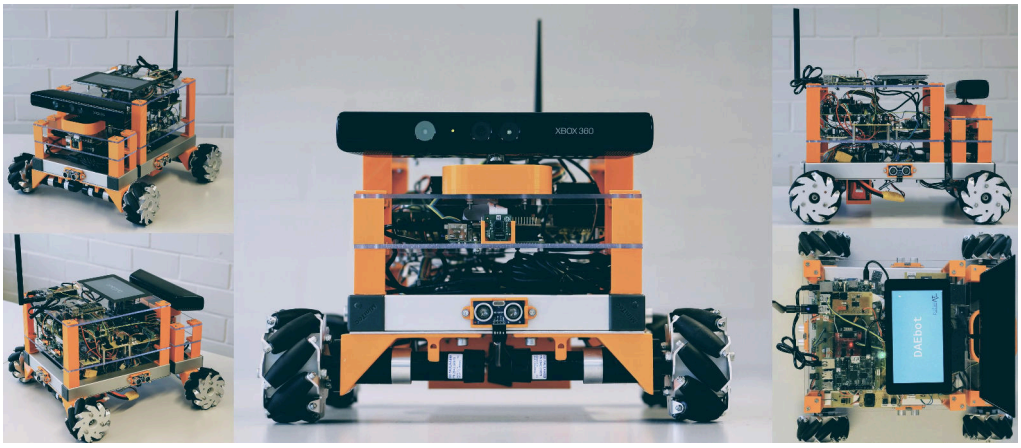


Abbildung 5.1: DAEbot

Dieses Kapitel beschreibt die experimentelle Umsetzung des in Kapitel 4 vorgestellten konzeptionellen Modells einer Systemarchitektur für mobile Roboter. Zur Umsetzung des experimentellen Modells wird ein WMR exklusiv für die Evaluation der Systemarchitektur neu entwickelt. Hierbei wird sich gegen die Nutzung oder Erweiterung einer verfügbaren Roboter-Plattform entschieden, da dies Limitierungen in Bezug auf die Adaption aller im konzeptionellen Modell vorgesehenen Funktionen zur Folge hätte. Durch die Entwicklung eines neuen Roboters bestehen, mit Ausnahme von wirtschaftlichen Faktoren, keine Vorgaben in Bezug auf die Auswahl der Rechner oder die Definition der Komponenten. Es entsteht der „Distributed Architecture Evaluation Robot (DAEbot)“ (siehe Abbildung 5.1). Der Fokus der Entwicklung liegt auf der Evaluation der Architektur in Bezug auf die Praxistauglichkeit des Konzeptes. Hierzu werden die in Kapitel 4.3 vorgestellten Systemkomponenten realisiert. Die Modularität des Systems, insbesondere auch in Bezug auf die softwareseitige Implementierung steht dabei im Vordergrund. Der DAEbot dient als Prototyp, bei welchem sowohl die Änderung der Zuordnung der Sensoren und Aktuatoren

zu den einzelnen Komponenten des verteilten Systems, als auch der Austausch der Komponenten, Rechner und Betriebssysteme unterstützt wird.

Die Entwicklung eines mobilen Roboters ist komplex und umfangreich. Für den hier entwickelten Roboter werden sowohl Hardware als auch Software interdisziplinär entwickelt. Ergänzend zur Implementierung des Autors, wurden mehrere Studenten mit Projekt- und Abschlussarbeiten zur Realisierung von Teilkomponenten eingebunden, welche allesamt vom Autor konzipiert und betreut wurden. Die Ergebnisse dieser Arbeiten werden im folgenden kenntlich gemacht, um sie eindeutig von den Ergebnissen des Autors zu trennen. Weiterhin hat der Autor bereits Ergebnisse aus der Entwicklung des Prototyps publiziert [A8–A10]. Dieses Kapitel zeigt nur diejenigen Aspekte, welche für die Umsetzung des OCM-basierten Ansatzes als besonders relevant erachtet werden und bildet nicht die gesamte Implementierung ab.

Zunächst wird der Systementwurf des DAEbots gezeigt (Kapitel 5.1). Anschließend werden einige Aspekte der entwickelten Middleware erläutert (Kapitel 5.2). Erläuterungen zur Realisierung des DAEbots als Demonstrator folgen in Kapitel 5.3 anhand der einzelnen Komponenten. Hierbei werden die Implementierungen stark zusammengefasst und zudem nicht vollständig abgebildet. Kapitel 5.4 zeigt weitere Demonstratoren, wie den AMiRo, für den die Middleware adaptiert und einiger Teile der Implementierung des DAEbots umgesetzt wurden oder werden.

5.1 Systementwurf

Der Systementwurf des DAEbots resultiert unmittelbar aus dem konzeptionellen Modell (vgl. Kapitel 4). Für die Umsetzung des lokalen Klienten werden Kleinstrechner eingesetzt. Hier werden drei weit verbreitete Kleinstrechner Familien ausgewählt: ein Arduino Mega 2560 wird als *Health Controller*, ein STM32F3 Discovery als *Motion Controller* und ein Raspberry Pi 3 Model B+ als *Perception Controller* verwendet (siehe Abbildung 5.2). Als *reflektorischer Operator* wird ein zusätzlicher Raspberry Pi 3 Model B+ eingesetzt. Die Eignung dieser Kleinstrechner wurde in Kapitel 4.2, u. a. mit Tabelle 4.1, festgestellt. Zudem wird der *reflektorische Operator+* mit einem ZynqBerry 7010 [O37] realisiert. Hierbei handelt es sich um einen HMP, da dieser Kleinstrechner sowohl einen ARM basierten Prozessor, als auch ein FPGA beinhaltet. Die experimentelle Umsetzung dient ebenfalls zur Evaluation der Verwendung von Kleinstrechnern für mobile Roboter in verteilter Systemarchitektur. Die Zuordnung der Kleinstrechner zu den jeweiligen Komponenten, sowie die Auswahl der Aktuatoren und Sensoren werden nachfolgend, u. a. in Kapitel 5.3, erläutert.

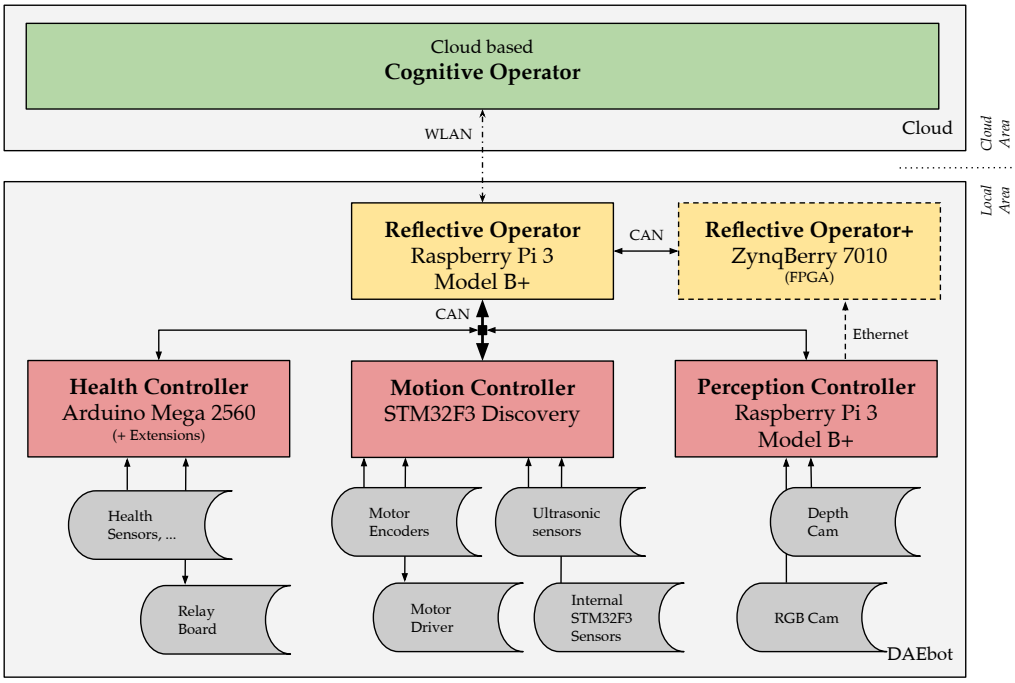


Abbildung 5.2: DAEbot Systemarchitektur

Die Umsetzung der Kommunikation in einem verteilten System kann die Leistungsfähigkeit des gesamten Systems beeinflussen. Ist die Kommunikation nicht bedarfsgerecht realisiert, z. B. wenn die gewählte Bandbreite die Daten der einzelnen Rechner nicht transferieren kann, so kann dies zum Flaschenhals des gesamten verteilten Systems werden. Als Schnittstelle für die Kommunikation im reflektorischen Kreis, wird der CAN-Bus gewählt (siehe Abbildung 5.2). Die Eignung des CAN-Bus wird in Kapitel 4.4.2 erläutert. Des Weiteren unterstützen alle vorgesehenen Kleinstrechner die Anbindung einer CAN-Bus Schnittstelle, entweder durch eine kommerziell erhältliche Erweiterung (im Fall des Arduino Mega 2560) oder durch die Entwicklung einer einfachen Hardwareerweiterung. Zusätzlich wird der *reflektorische Operator+* via Ethernet angebunden (vgl. Kapitel 4.4.3). Die Kommunikation im kognitiven Kreis wird mit WLAN realisiert (vgl. Kapitel 4.4.4).

Die Gestalt eines mobilen Roboters wird maßgeblich von der Antriebsart und dem physischen Aufbau beeinflusst. Der DAEbot ist ein mobiler Roboter der Klasse der WMR und wird somit durch Räder betrieben. Als Basis des DAEbots dient eine Plattform, welche

aus einem omnidirektionalen Antrieb mit vier Mecanum Rädern[1], vier Motoren[2], zwei Motortreibern, vier Motor-Encodern[3] und vier Ultraschallsensoren[4] besteht. Diese Plattform, sowie die Motortreiber wurden im Rahmen des internen Forschungsprojektes FILU [O42] der Fachhochschule Dortmund erstellt und dem Autor zur Entwicklung der Hard- und Software zur Datenverarbeitung zur Verfügung gestellt. Die Plattform ist omnidirektionalen Robotern mit Mecanum Rändern, wie dem omnidirektionalen Roboter von Nexus Robot [O43], nachempfunden.

Der Vorteil eines omnidirektionalen Antriebs ist die Möglichkeit, sich mit drei Freiheitsgraden fortbewegen zu können, d. h. sich in jede beliebige Richtung auf einer Ebene zu bewegen. Diese Antriebstechnik wird laut Lin und Shih [182] u. a. bei intelligenten Rollstühlen oder Gabelstaplern eingesetzt, um die Beweglichkeit in alle Richtungen durchführen zu können, ohne dafür die Ausrichtung ändern zu müssen. Dieses erhöht insbesondere in beengten oder überlasteten Umgebungen die Flexibilität bei der Fortbewegung [182].

Ein omnidirektionaler Antrieb mittels Mecanum Rädern verfügt über mindestens drei Mecanum Räder. Ein einzelnes Mecanum Rad besteht aus mehreren Rollen, meist im Winkel θ von 45° (siehe Abbildung 5.3a). Diese Rollen sind auf der Längsachse $x_{r,i}$ frei beweglich und können somit in Richtung $\pm y_{r,i}$ rotieren. Die frei beweglichen Rollen sind durch eine Narbe verbunden, welche durch einen Motor nach vorne (in Richtung x_R) oder nach hinten (in Richtung $-x_R$) gedreht werden kann. Der Antrieb des DAEbots verfügt über vier Mecanum Räder (siehe Abbildung 5.3b) mit je neun Rollen. Durch die Anordnung der Rollen zueinander ist es möglich, die Bewegung in den folgenden drei Freiheitsgraden zu erreichen.

- $\dot{x}$: Bewegung entlang der x-Achse (gerade Bewegung)
- $\dot{y}$: Bewegung entlang der y-Achse (seitliche Bewegung)
- $\dot{\varphi}$: Bewegung um die z-Achse (Rotation)

Eine seitliche Bewegung nach links (y_R) wird erreicht, indem Rad 1 und Rad 3 vorwärts (φ_1 und φ_3) und Rad 2 und Rad 4 rückwärts ($-\varphi_2$ und $-\varphi_4$) gedreht werden. Für eine Diagonalfahrt nach vorne rechts (x_R, $-y_R$) werden nur Rad 2 und Rad 4 vorwärts betrieben (φ_2 und φ_4), die anderen beiden Räder stehen und rollen über die frei beweglichen Rollen. Ein Nachteil des omnidirektionalen Antriebs mittels Mecanum Rädern zeigt sich besonders bei der Geradeausfahrt, bei welcher alle Räder vorwärts gedreht werden (φ_1, φ_2, φ_3 und φ_4). Durch die in 45° angestellten Rollen, entsteht eine große Reibung und somit eine ineffiziente Ausnutzung der aufgebrachten Energie zur Rotation der Räder. Künemund, Heß und Röhrig zeigen in [184] ein Modell zur Energieoptimierung

[1] Nexus Robot 100 mm 14094 (2xR, 2xL) [O38]
[2] Faulhaber 2642W012CR [O39]
[3] AVAGO HEDM-5500-J02; 1533 A [O40]
[4] OSEPP Electronics HC-SR04 [O41]

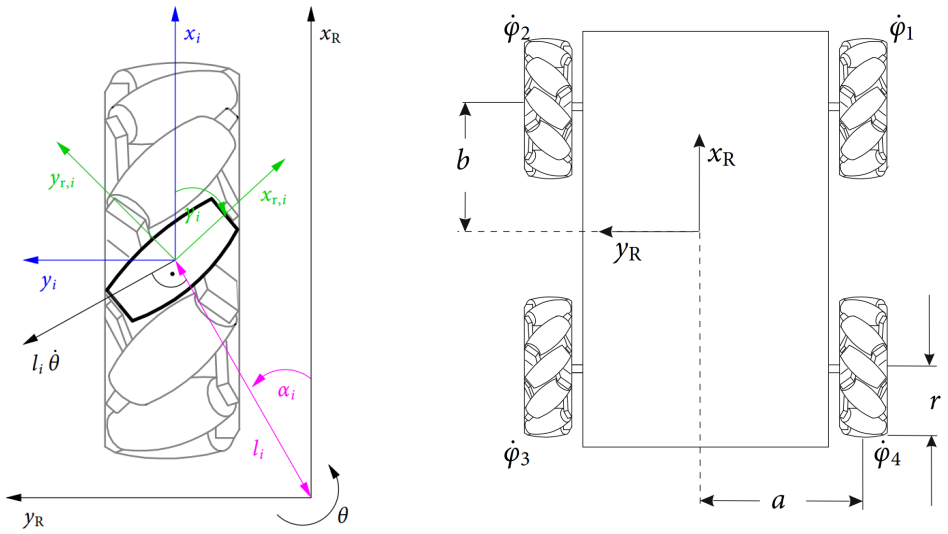


Abbildung 5.3: Omnidirektionaler Antrieb mit Mecanum Rädern [183]

der Bahnplanung. Durch die drei Freiheitsgrade kann die Bahnplanung in unzähligen Varianten erfolgen und ist somit ungleich komplexer, als bei Antrieben mit feststehenden Rädern und einer Lenkachse.

Der Fokus bei der Entwicklung des DAEbots liegt auf der Umsetzung der Schichtenarchitektur als verteiltes System. Der DAEbot ist ein Prototyp und dient als Demonstrator dieser Architektur. Dieses zeigt sich auch beim ungewöhnlichen physischen Aufbau des Roboters. Die Komponenten des Roboters sind je in einer eigenen Schicht implementiert (siehe Abbildung 5.4). Die Schichten sind durch 3D gedruckte (orange) Abstandshalter (siehe Abbildung 5.1) voneinander getrennt und die Kleinstrechner mitsamt der Hardware sind auf transparenten Plexiglasscheiben befestigt, sodass die gesamte Hardware sichtbar ist. Analog zur Systemarchitektur sind hierbei unten die *Controller*, bestehend aus *Motion*, *Health* und *Perception Controller* und darüber der *reflektorische Operator* und der *reflektorische Operator+* zu finden. Der *kognitive Operator* ist zur Verdeutlichung in Abbildung 5.4 über dem mobilen Roboter abgebildet, wenngleich dieser in der Cloud realisiert ist.

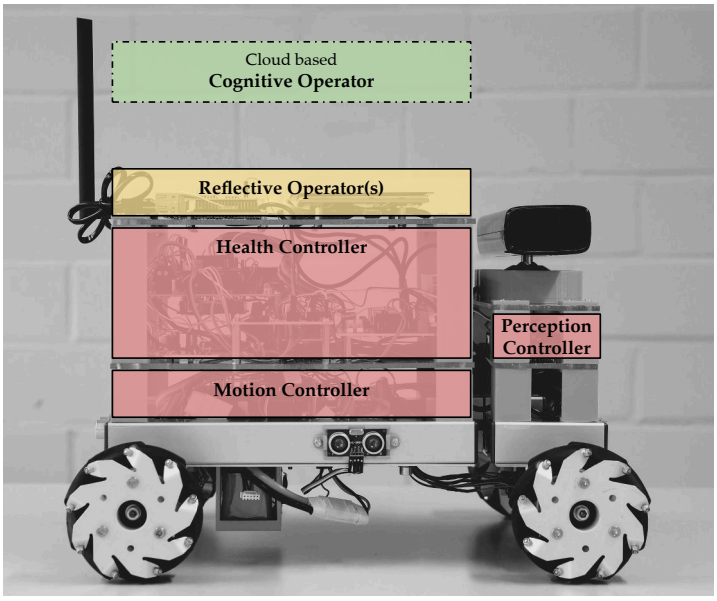


Abbildung 5.4: DAEbot als Demonstrator der Schichtenarchitektur

Die Energieversorgung erfolgt durch einen Lithium-Polymer-Akkumulator[5], welcher unter der Basis des mobilen Roboters in eine Halterung geschoben wird.

5.2 Middleware

Schon im Jahr 1996 schrieb Tresch [185], dass die Middleware die Schlüsseltechnologie zur Entwicklung verteilter (Informations-) Systeme sei. Etzkorn definiert den Begriff der Middleware wie folgt [186, S. 4 ff.]:

[5] Conrad energy 14,8 V 5500 mAh Zellen-Zahl: 4 20 C [O44]

> „Eine Middleware wird verwendet, um die Komplexität der unteren Ebenen von Netzwerk und Betriebssystem vor dem Anwendungsprogrammierer zu verbergen. Sie kann es einem Client, der auf einer Art von Rechner ohne Betriebssystem läuft, erlauben, mit einem Server auf einem anderen Rechner mit einem anderen Betriebssystem zu kommunizieren. Eine Middelware erlaubt die Definition klar definierter Schnittstellen zu Servern, die von den Clients leicht aufgerufen werden können.“

Wenngleich Etzkorn von Server und Client spricht, trifft die Definition gleichermaßen für alle Muster von verteilten System zu. Die Middleware als Zwischenschicht zwischen hardwarenahen Modulen, wie dem Betriebssystem und der Anwendungsschicht stellt Funktionen zur Datenverwaltung und Kommunikation zur Verfügung und vermittelt zwischen den Anwendungen. Bei der Umsetzung der Middleware für den DAEbot steht die Implementierung der Kommunikation (nachfolgend in Kapitel 5.2.1 und Kapitel 5.2.2), der modellbasierten Entwicklung (Kapitel 5.2.3) und der Watchdogs (Kapitel 5.2.4) im Fokus. Nachfolgend wird jeweils erst die Implementierung exemplarisch erläutert und abschließend in einem kurzen Anwendungsbeispiel die Anwendung dessen beschrieben.

5.2.1 CAN-Schnittstelle

Die CAN-Schnittstelle wird im reflektorischen Kreis eingesetzt, welche wie in Kapitel 4.4.2 festgestellt, echtzeitfähig implementiert werden muss. Hierzu kann das CAN-Protokoll, wie ebenfalls in Kapitel 4.4.2 erläutert, erweitert werden [160, 163–165]. Diese Ansätze werden für die Implementierung des DAEbots erweitert und auf das OCM Konzept zugeschnitten.

Priorisierung

Im Vordergrund steht die Priorisierung der Nachrichten im CAN-Protokoll, damit hoch priorisierte Nachrichten sofort bearbeitet werden können. Hierzu wird sich die im CAN-Protokoll genutzte bitweise Arbitrierung zu Nutze gemacht. Die Phase der Arbitrierung bezieht sich auf den Nachrichten-ID des CAN-Daten-Frames. Diese Nachrichten-ID ist Teil des gesamten CAN-Daten-Frames, welche wie in Abbildung 5.5 dargestellt aufgebaut ist. Nach einem dominanten Start-Bit (Start of Frame (SOF)) folgt die besagte Nachrichten-ID. Die Nachrichten-ID besteht aus 11 Bit und kann im sogenannten erweitertem Frame-Format auf 29 Bit erweitert werden. Nachfolgend wird die Nachrichten-ID mit einem Remote Transmission Request (RTR)-Bit abgeschlossen. Es folgt ein 6-Bit langes Kontrollfeld, welches u. a. mit dem Data Length Code (DLC) (vier Bit) die Anzahl der Datenfelder konfiguriert. Die Anzahl der Datenfelder kann hierbei zwischen null und acht Datenfeldern zu je acht Bit betragen, welche im Anschluss übertragen werden. Es folgen die 15 Bit

lange Prüfsumme (Cycle Redundancy Check (CRC)), zwei Bestätigungsfelder (ACK), sieben Ende-Bits (End of Frame (EOF)) und abschließend drei Bits zur Trennung von aufeinander folgenden CAN-Daten-Frames (Intermission Frame Space (IFS)). [187]

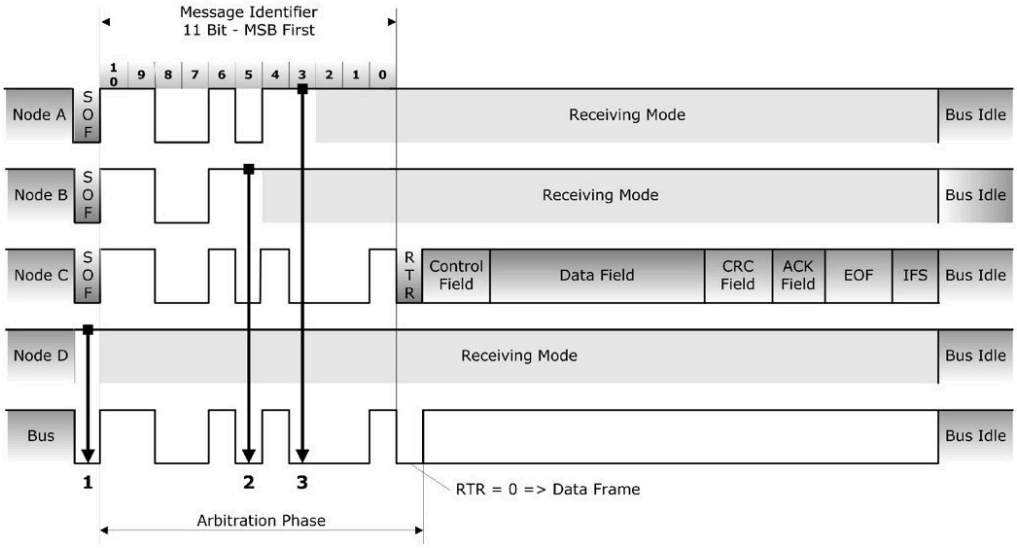


Abbildung 5.5: Bitweise Arbitrierung im CAN-Protokoll [187]

Während der Übertragung der Nachrichten-ID-Bits erfolgt die bitweise Arbitrierung. Hierbei überwacht jeder Sender den Bus, während die Nachrichten-ID gesendet wird. Senden zwei Teilnehmer gleichzeitig die Nachrichten-ID, so wird Bit für Bit geprüft, ob ein dominantes Bit[6] auf ein rezessives[7] Bit trifft und dieses überschreibt. Der Teilnehmer, dessen rezessives Bit überschrieben wird, erkennt dieses und beendet seine Übertragung der Nachricht. Das Beispiel in Abbildung 5.5 zeigt vier Bus Teilnehmer (Node A bis D). Node A, B und C sind im Begriff zu senden, Node D ist bereit, einen CAN-Frame zu empfangen. Die erste Arbitrierung der Nachrichten-ID erfolgt bei „1“. Node A, B und C senden ein dominates Start-Bit, welches das regressive Bit von Node D überschreibt. Node D ist somit im Empfangsmodus. Die nächste Arbitrierung erfolgt beim sechsten übertragenen Nachrichten-ID-Bit („2“). Das rezessive Bit von Node B wird mit den dominanten Bits von Node A und Node C überschrieben. Node B bricht den Sendeversuch ab und wechselt in den Empfangsmodus. Beim achten übertragenem Nachrichten-ID-Bit (siehe „3“) wird das rezessive Bit von Node A vom dominanten Bit von Node C überschrieben. Node A wechselt folgerichtig in den Empfangsmodus und nach der Prüfung

[6] hier eine logische `0`
[7] hier eine logische `1`

der verbleibenden Nachrichten-ID-Bits, sendet Node C die (Kontroll- und) Nachrichten-Bits. [187]

Insofern die Nachrichten-IDs eindeutig vergeben werden, kann durch die Arbitrierung der Verlust von Daten unterbunden werden. CAN-Daten-Frames mit niedriger Nachrichten-ID werden jenen mit höherer Nachrichten-ID vorgezogen und somit priorisiert. Für die Priorisierung der CAN- Nachrichten beim DAEbot, werden die 11 Bit der Nachrichten-ID in drei Teile, bzw. Phasen unterteilt.

Abbildung 5.6: Angepasste CAN-Nachrichten-ID

Die angepasste CAN-Nachrichten-ID beginnt stets mit dem sogenannten Command/ Sensor (C/S)-Bit (siehe Abbildung 5.6). Bei Nachrichten von einem Sensor ist dieses Bit stets rezessiv. Das C/S-Bit ist dominant, wenn entweder ein Stellwert an einen Aktuator oder eine Konfigurations- oder Befehlsnachricht[8] an einen Sensor gesendet wird. Dieses C/S-Bit wird bei der bitweisen Arbitrierung stets als erstes überprüft (Phase 1), ein Aktuator-Stellwert und Konfigurations-Befehl erhält somit stets den Vorzug über Sensordaten. In Phase 2 wird die Priorität der Nachricht definiert. Hierbei kann mit den zwei zur Verfügung stehenden Bits aus 4 Prioritäten[9] wie folgt gewählt werden (jeweils mit der Bitfolge Bit 9, Bit 8).

- 11_2 $(= 3_{10})$ - Niedrige Priorität
- 10_2 $(= 2_{10})$ - Mittlere Priorität
- 01_2 $(= 1_{10})$ - Hohe Priorität
- 00_2 $(= 0_{10})$ - Höchste Priorität

In Phase 2 wird hierbei die Priorität dynamisch an die Anforderungen im System angepasst, d. h. aktuell kritische Nachrichten werden zeitweise höher priorisiert. In Phase 3 folgt der eigentliche Identifikator aus 8 Bits. Hierbei erhält jeder Sensor und jeder Aktuator eine eindeutige und nicht veränderbare ID. Da durch die vorangegangene Priorisierung durch das C/S-Bit und die Priorität-Bits die Priorisierung bereits erfolgt ist, spielt die mögliche Priorisierung durch die Vergabe von niedrigen IDs für zeitkritische Aktuatoren und Sensoren eine untergeordnete Rolle. Durch die 8-Bit lange ID können $(2^8 =)$ 256 Sensoren/ Aktuatoren eindeutig definiert werden. Sollte dieses nicht ausreichen, so kann das erweiterte CAN-Frame-Format mit einer 29 Bit Nachrichten-ID gewählt werden. Hier würden, bei gleichbleibender Anzahl von C/S- und Priorität-Bits, 26 Bits für die ID und

[8] wird nachfolgend im Abschnitt „Publisher Modi“ erläutert
[9] Anzahl an Möglichkeiten $= 2^{\text{Anzahl der Bits}} = 2^2 = 4$

somit ca. ($2^{26} \approx$) 67 Millionen Möglichkeiten zur Vergabe von eindeutigen IDs zur Verfügung stehen. C/S-Bit, Priorisierung und Nachrichten-ID werden vor dem Versenden zusammengesetzt und beim Empfang decodiert. Die softwaretechnische Realisierung kann in Anhang B (Quelltextauszüge A1 und A2) nachgeschlagen werden.

Publisher Modi

Neben der Priorisierung der CAN-Daten-Frames können vornehmlich die *Controller* über CAN-Nachrichten konfiguriert werden. Hierbei wird das Standard CAN-Format (11 Bit Nachrichten-ID) inklusive der angepassten Nachrichten-ID verwendet. Die Konfiguration bezieht sich insbesondere auf die Anpassung der Publisher/ Subscriber Funktionalität. Diese Methode sieht vor, dass Sensordaten zyklisch durch die *Controller* dem System zur Verfügung gestellt werden. Der *reflektorische Operator* verarbeitet diese. Um die Anzahl an Nachrichten auf dem CAN-Bus zu begrenzen, bzw. nicht unnötig stark zu belasten und die Auslastung der *Controller* zu steuern, kann der *reflektorische Operator* die Publisher jedes einzelnen Sensors (der *Controller*) einstellen. Hierzu werden das C/S-Bit dominant gesetzt, eine Priorität gewählt und der zu konfigurierende Sensor mittels seiner eindeutigen ID adressiert. Durch das dominante C/S-Bit wissen die Bus-Teilnehmer, dass ein Aktuatorstellwert oder eine Konfigurations-Nachricht folgt. Nun erfolgt die Konfiguration anhand der drei in Tabelle 5.1 dargestellten Modi.

Modus	**data[0]**	**data[1] & data[2]**	**data[3]**
Modus 1: Beende Publisher	0	-	-
Modus 2: Starte Publisher	1	[Transferrate] (default: 50)	[Einheit] 0: ms (default) 1: s 2: µs
Modus 3: Anfrage für einmalige Übertragung	2	-	-
Beispiel: Starte Publisher, welcher alle 3 Sekunden sendet	1	3	1

Tabelle 5.1: CAN-Publisher Modi

Enthält die Konfigurations-Nachricht im ersten Datenfeld *(data[0])* eine `0`, so wird der Publisher beendet. Der entsprechende *Controller* beendet hierbei nicht nur die Übertragung, sondern stellt auch die Messungen und die ggf. nachfolgenden Vorverarbeitungen des Sensors aus (mit Ausnahme der Sensoren, welche für die internen Watch-Dogs benötigt werden). Der *Controller* kann die nun nicht mehr benötigte Rechenleistung und Bandbreite der CAN-Schnittstelle für andere Sensoren und Aktuatoren verwenden.

Modus 1, welcher durch einen CAN-Daten-Frame mit einem dominanten C/S-Bit und einer `1` im ersten Datenfeld *(data[0])* ausgewählt wird, ermöglicht das Starten eines Publishers oder das Ändern der aktuellen Transferrate. Ohne weitere Konfiguration wird ein Publisher mit einer Transferrate von 50 ms gestartet. Die Transferrate kann jedoch mittels der nachfolgenden drei Datenfelder spezifiziert werden. Mit *data[3]* wird die Einheit ausgewählt. Hier kann zwischen ms, s und µs gewählt werden. Die Datenfelder *data[1]* und *data[2]* geben die zugehörige Zahl der Transferrate an. Das Beispiel in Tabelle 5.1 zeigt die Aktivierung eines Publishers mit einer Transferrate von 3 s.

Zudem kann mit Modus 3 eine einmalige Übertragung eines Sensors angefragt werden (Polling). Dieser Modus bietet eine zusätzliche Option, wird im normalen Betrieb jedoch nicht verwendet. Modus 3 stoppt den Publisher nicht. Dieses Polling kann z. B. verwendet werden, falls gerade einmalig ein Sensorwert benötigt wird oder die Publisher Transferrate so niedrig ist, dass nicht auf die nächste Übertragung gewartet werden kann (z. B. 10 s).

Mit einer entsprechenden Überwachung der Systemauslastung von jedem Rechner des verteilten Systems können die Transferraten aller Sensoren so eingestellt werden, dass die *Controller* sinnvoll eingesetzt werden können[10]. Der sinnvolle Einsatz bezieht sich hierbei auf die Auslastung der Hardware. So sollte z. B. die CPU weder im Leerlauf, als auch unter Volllast agieren. Um diese Hardware-Auslastung zu erreichen, werden nicht benötigte Publisher herunter getaktet oder ganz deaktiviert und aktuell wichtige Sensoren transferieren ihre Daten öfters. Die Konfiguration der Modi kann jederzeit erfolgen und dynamisch an die aktuelle Situation angepasst werden. Für den Systemstart ist für jeden Sensor eine Voreinstellung[11] hinterlegt, sodass die Sensoren nach dem Systemstart nicht zwangsläufig konfiguriert werden müssen. Diese Voreinstellung ist so gewählt, dass der DAEbot in typischen Szenarien ohne weitere Konfigurationen eingesetzt werden kann.

[10] Zur Überwachung der Systemauslastung wird das nachfolgend in Kapitel 6 vorgestellte Tool *pulseAT* verwendet.
[11] engl. default

Anwendungsbeispiel

Abbildung 5.7 zeigt exemplarisch die Anwendung des selbst angepassten CAN-Nachrichten-ID.

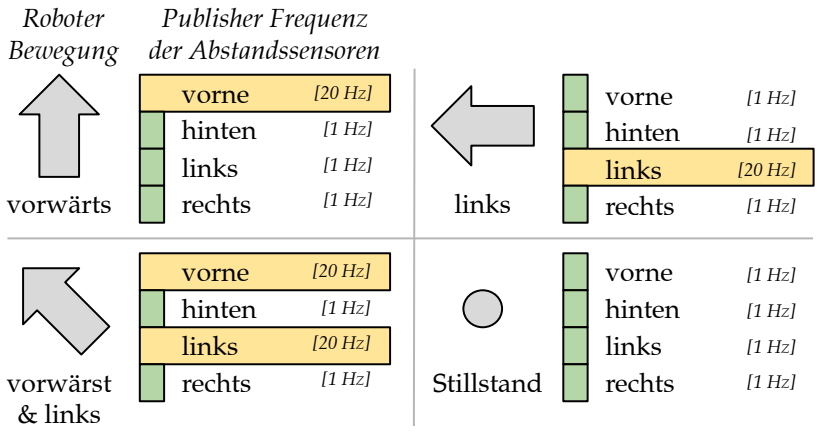


Abbildung 5.7: Adaptive CAN Publisher Transferrate und Priorität

Die vier Abstandssensoren des DAEbots, welche an den vier Seiten des mobilen Roboters verteilt sind (vorne, hinten, links und rechts), werden aufgrund der Fahrtrichtung dynamisch konfiguriert. So kann im ersten Schritt die Priorität von vier Abstandssensoren angepasst werden. Fährt der Roboter nach vorne, so kann die Priorität des nach vorne blickenden Abstandssensors z. B. auf eine hohe Priorität konfiguriert werden (siehe gelbe Balken in Abbildung 5.7), während alle anderen Sensoren auf die niedrigste Priorität wechseln (grüne Balken), da hier die Kollision mit einem (stehendem) Objekt ausgeschlossen ist. Ändert der Roboter nun die Fahrtrichtung in „links und vorne“, so werden entsprechend die Sensoren mit Blick nach vorne und links priorisiert. Die höchste Priorität wird typischerweise nicht vergeben. Diese wird besonderen Ereignissen vorbehalten, wie beispielsweise einem untypisch nahem Abstand zu einem Objekt. Zudem werden die Publisher und somit die Datenverarbeitung auf dem *Motion Controller* dynamisch angepasst. Fährt der Roboter geradeaus, so wird der nach vorne blickende Abstandssensor mit einer Frequenz von 20 Hz und alle anderen Sensoren nur einmal pro Sekunde geprüft (1 Hz). Dieses bezieht sich nicht nur auf die Übertragung der Sensordaten, sondern auch auf die Berechnung dieser. Die Berechnung und Abfrage der Sensoren (hier der Abstandssensoren des *Motion Controllers*) werden analog zur Transferrate verändert und die Sensoren entsprechend der Publisher Konfiguration häufiger oder weniger häufig abgefragt. Diese Methode ermöglicht sehr hohe Abtast- und Transferraten für aktuell „wichtige“ Sensoren, während bei anderen Sensoren

(hier die Abstandssensoren hinten, links und rechts) die Rechneraktivität reduziert bzw. an aktuell „wichtigere“ Sensoren verlagert werden kann. Außerdem wird zudem die Anzahl an Nachrichten auf dem CAN-Bus verringert.

5.2.2 MQTT-Schnittstelle

Während der Datenaustausch im reflektorischen Kreis via CAN erfolgt, werden intern im *reflektorischen* und *kognitiven Operator*, sowie im kognitiven Kreis u. a. MQTT eingesetzt. Das primäre Ziel der Nutzung von MQTT ist die Anbindung der Cloud Dienste und der Tools aus dem TICK Stack[12] [O45], welche auf dem *kognitiven Operator* eingesetzt werden.

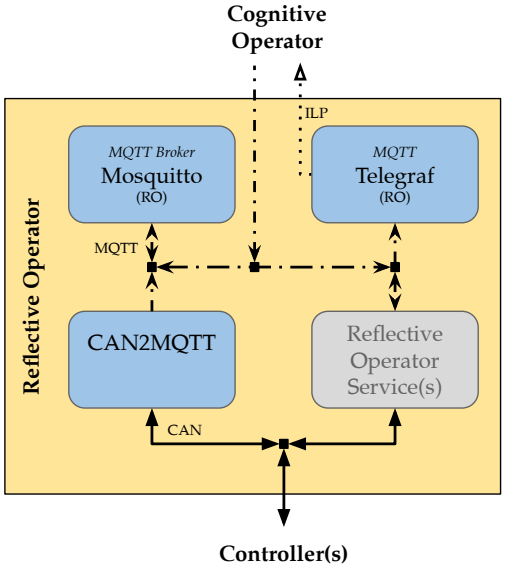


Abbildung 5.8: MQTT Middleware

Die MQTT-Middleware besteht aus mehreren Modulen. Die Beschreibung der MQTT-Middleware wird hier anhand des *reflektorischen Operators* erläutert (siehe Abbildung 5.8). Die Middleware entstand in einer studentischen Arbeit [B10][13], welche vom Autor konzipiert und betreut wurde.

[12] Der TICK Stack umfasst diverse Tools zur Speicherung, Visualisierung und weiteren Verarbeitung von Daten, insbesondere aus der Integration von IoT Geräten. Hierzu zählen Tools zum Einsammeln der Daten (Telegraf), eine Datenbank (InfluxDB), ein Tool zur Visualisierung der Daten (Chronograf) und ein Tool zur weiteren Datenverarbeitung (Kapacitor). Diese Tools werden im Folgenden erläutert.

[13] Referenzen zu betreuten Arbeiten werden mit dem Prefix **B** *(für betreut)* kenntlich gemacht ([B *Nr.*]).

Der *reflektorische Operator* ist als verbindende (und trennende) Komponente zwischen der *Controller-* und der *kognitiven* Ebene auch für den Informationsfluss zwischen diesen Ebenen zuständig. Hierbei kommen die Informationen des reflektorischen Kreises als CAN-Frames an. Die Kommunikation im kognitiven Kreis erfolgt mittels MQTT und InfluxDB Line Protocol (ILP) [O46]. Der Downlink aus der Cloud, d. h. die Verbindung zwischen dem *kognitiven Operator* (Sender) und dem *reflektorischen Operator* (Empfänger), erfolgt direkt als MQTT-Nachrichten. Der Uplink, d. h. die Verbindung zwischen dem *reflektorischen Operator* (Sender) und dem *kognitiven Operator* (Empfänger), erfolgt im ILP-Format, da die Daten so in der Cloud ohne weitere Verarbeitung direkt in eine Datenbank geschrieben werden können.

Mosquitto

Zum Austausch von Daten via MQTT ist die Implementierung eines Brokers unabdingbar. Ein MQTT-Broker koordiniert die Kommunikation zwischen den Klienten, z. B. durch die Verarbeitung der Publisher- und Subscriber-Anfragen [188]. Für die experimentelle Umsetzung wird der Mosquitto MQTT-Broker [189] verwendet. Dieser wird im asynchronen Betrieb ausgeführt, sodass Subscriber und Publisher in einzelnen Threads und somit unabhängig voneinander verarbeitet werden.

MQTT Broker können prinzipiell an verschiedenen Stellen im verteilten System realisiert werden. Banno, Sun, Fujita, Takeuchi und Shudo haben hierzu eine ausführliche Analyse durchgeführt [190]. Hierbei verfolgt die Implementierung des DAEbots und der MQTT Middleware den Ansatz des Edge-Computing. Bei diesem werden lokale Broker verwendet, beim cloudbasierten Ansatz hingegen befindet sich ein einziger Broker in der Cloud [190]. So sieht auch die Realisierung des DAEbots je einen lokalen Broker auf dem *reflektorischen* und dem *kognitiven Operator* vor. Diese Anordnung führt laut [190] insbesondere zu schnellen Übertragungen auf der lokalen Ebene. Diese Einschätzung wird durch die testweise Implementierung eines einzigen Brokers in der Cloud bestätigt. Hierbei ist die Datenübertragung auf dem *reflektorischer Operator* mit zwei Brokern signifikant höher, als mit einem Broker in der Cloud. Zudem bleiben *reflektorischer* und *kognitiver Operator* in Bezug auf den lokalen MQTT Datenfluss unabhängig voneinander. Es können und werden zwar Daten zwischen den Brokern ausgetauscht, ist die Verbindung zur Cloud unterbrochen, ist jedoch weiterhin eine lokale Nutzung der MQTT Daten auf dem *reflektorischen Operator* möglich.

CAN2MQTT

Die CAN-Daten aus dem reflektorischen Kreis werden parallel zur Verarbeitung zur Steuerung des Roboters in MQTT-Nachrichten umgewandelt und teilweise im Rahmen eines MQTT-Publishers veröffentlicht. Hierzu wurde in [B10] die Applikation `CAN2MQTT` implementiert, welche diese Transformation durchführt. Zur Vorbereitung der Überführung in die ILP-Syntax werden die CAN-Daten hier bereits angereichert. Aus den CAN-Daten werden die ID und die Daten in die MQTT Syntax transferiert (siehe Abbildung 5.9). Während der Messwert direkt übernommen wird, wird die CAN-Nachrichten-ID mittels Lookup-Tabellen (LUT)[14] in sein MQTT Äquivalent überführt. Die Umwandlung des C/S-Bits und die Priorität der CAN-Nachrichten aus dem reflektorischen Kreis in MQTT-Nachrichten ist für die exemplarische Umsetzung des DAEbots nicht notwendig. Die MQTT Syntax besteht aus einem Topic und Payload(s)[15]. Dieser Payload wird mit drei Werten gefüllt. Die Measurement ID wird nachfolgend für die Übertragung in das ILP-Format benötigt und ist für den *reflektorischen Operator* fest hinterlegt. Hinzugefügt wird neben dem Messwert, ebenso der Zeitstempel[16], wofür die Applikation `CAN2MQTT` die aktuelle Systemzeit ausliest.

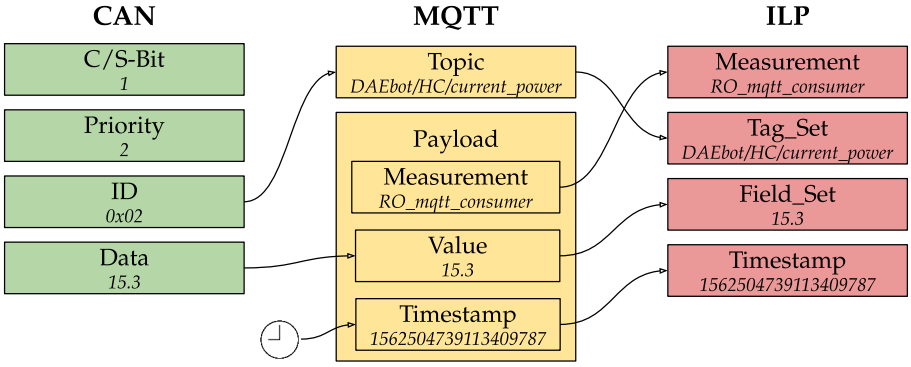


Abbildung 5.9: CAN, MQTT und ILP Syntax und die Übertragung an einem Beispiel

Neben der Transformation der CAN-Daten in ILP konforme MQTT-Nachrichten, kann in der Applikation ausgewählt werden, welche Daten in die Cloud übertragen werden sollen und wie oft diese Übertragung erfolgt. Eine Umwandlung der MQTT-Nachrichten, welche der *kognitive Operator* sendet, erfolgt nicht. Hier ist weiterhin eine strenge Trennung zwischen *kognitivem Operator* und den *Controllern* implementiert. Nachrichten des *kognitiven Operators* werden zunächst vom *reflektorischen Operator* verarbeitet und haben nur einen mittelbaren Einfluss auf die *Controller*.

[14] dt. Umsetzungstabellen
[15] dt. Nutzdaten
[16] engl. timestamp

Telegraf

Die Übertragung der MQTT-Nachrichten auf den cloudbasierten *kognitiven Operator* kann in mehreren Varianten erfolgen. Hierbei ist es beispielsweise möglich, die MQTT-Nachrichten direkt zum *kognitiven Operator* zu senden. Eine andere Variante ist die lokale Umwandlung der Nachrichten in die ILP Syntax. Diese Variante hat den Vorteil, dass ILP-Befehle eine effizientere Übertragung erlauben, als die Übertragung von MQTT-Nachrichten mit anschließendem Schreibbefehl der Daten in eine Datenbank. Analog zur Entscheidung für lokale MQTT-Broker, erfolgt die Umwandlung bei der exemplarischen Umsetzung des mobilen Roboters im Sinne des Edge-Computing lokal und nicht erst in der Cloud. Die Umwandlung erfolgt mit Telegraf [191] aus dem TICK Stack. Telegraf ist ein Tool zur Datensammlung. Das hier verwendete `MQTT Consumer Input Plugin` [O47] sammelt alle anliegenden MQTT-Nachrichten ein, hier insbesondere die MQTT-Nachrichten aus der `CAN2MQTT` Applikation, welche bereits alle notwendigen Felder der ILP-Syntax beinhalten. Die MQTT-Nachrichten werden mittels ILP-Befehlen direkt in die angebundene Datenbank (hier InfluxDB) transferiert. Die entsprechende Datenbank ist hierbei auf dem *kognitiven Operator* implementiert. Sollte die Verbindung zum *kognitiven Operator* nicht verfügbar sein, so können die Befehle kurzzeitig gepuffert werden. Da der Zeitstempel bereits in der MQTT-Nachricht enthalten ist, ist die zu speichernde Information trotz Pufferung mit dem korrekten Zeitstempel versehen (und übernimmt nicht den Zeitstempel zum Zeitpunkt des Schreibens in die Datenbank).

Anwendungsbeispiel

Ein Sensorwert des am *Health Controller* angebundenen Sensors zur Messung der aktuellen elektrischen Leistung liefert einen neuen Messwert von 15,3 W (siehe Abbildung 5.9). Dieser Messwert wird parallel zur Verarbeitung der Steuerung des DAEbots mit der Applikation `CAN2MQTT` in eine ILP konforme MQTT-Nachricht umgewandelt und als Publisher-Event veröffentlicht (siehe Abbildung 5.8). Hierbei werden die CAN-Nachrichten-ID `0x02` mittels LUT in den MQTT Topic `DAEbot/HC/current_power` umgesetzt, die fest hinterlegte Measurement-ID (`RO_mqtt_consumer`) integriert und der aktuelle Zeitstempel (in Nanosekunden) abgerufen und gespeichert. Die Freigabe der Veröffentlichung der MQTT-Nachricht erfolgt anschließend durch den Mosquitto MQTT-Broker. Die nun verfügbare MQTT-Nachricht wird von Telegraf eingesammelt und in einen ILP-Befehl umgewandelt. Durch die (automatische) Ausführung dieses Befehls wird der Messwert an den Cloud-basierten *kognitiven Operator* gesendet und dort in die Datenbank InfluxDB geschrieben.

5.2.3 Toolbox zur modellbasierten Entwicklung

Zur Bewältigung der Komplexität bei der Entwicklung von mobilen Robotern hilft der Einsatz von Methoden und Tools der MBSE [123, S. 46 ff.]. Des Weiteren können die logischen Zusammenhänge und Abläufe des *reflektorischen Operators* in Zustandsautomaten (FSM) modelliert werden. Mit Stateflow, einer Erweiterung für MATLAB Simulink, können diese Zustandsautomaten grafisch modelliert und hieraus Quelltext für die Zielhardware generiert werden. Hierbei handelt es sich beim DAEbot um den *reflektorischen Operator*, welcher auf einem Raspberry Pi 3 Model B+ implementiert ist. Die Generierung des Quelltextes aus MATLAB erfolgt in C/C++. Ein *Support Package* für MATLAB Simulink [O48] ermöglicht neben der Generierung des Quelltextes für den ARM-basierten Prozessor des Raspberry Pi auch die Übertragung der kompilierten Binärdatei und das Starten des Modells auf der Zielhardware. Zudem kann das Modell sowohl eigenständig auf der Zielhardware ausgeführt werden *(„Deploy to Hardware“)* oder, insbesondere zum 'Testen der Applikation, mit dem Entwicklungsrechner auch nach dem Übertragen der Binärdateien verbunden bleiben *(„Run“)*. Während dieser Kopplung wird in Simulink und Stateflow der aktuelle Zustand des Modells angezeigt. Hier können Entwickler*innen z. B. nachvollziehen, in welchem Zustand im Zustandsautomaten sich das Modell aktuell befindet und somit das Verhalten der Zustandsautomaten testen und verifizieren.

Die Nutzung von MATLAB Simulink (Stateflow) ist auch nach Erfahrungen des Autors [A3] zunächst aufwendig. Insbesondere bei der Entwicklung des DAEbots mit seinen über 40 Sensoren und Aktuatoren trifft dies zu. Jeder einzelne Sensor und Aktuator werden mittels mehrerer externer C-Dateien in die MATLAB Simulink Umgebung integriert. In Abbildung 5.10 ist die Einbindung des Gyroskops dargestellt. Hinter den Boxen `gyroscope_polling`, `gyroscope_publisher` und `gyroscope_input` liegen jeweils eine *M-Datei*[17], eine *C-Datei* und eine *Header-Datei*. Das Beispiel zeigt links die Konfiguration des Sensors bzw. des zugehörigen *Controllers*. Hier wird die Priorität eingestellt, sowie die Abtastrate des Sensors übergeben. Innerhalb der extern eingebundenen Quelltexte, werden das C/S-Bit und die Nachrichten-ID hinzugefügt und bei Änderung eines der Eingangsvariablen (Eingang 1 - 3) das Versenden der Konfigurationsnachricht veranlasst. Rechtsseitig werden mit `gyroscope_input` die gemessenen Positionen des Gyroskops für `x`, `y` und `z` in die MATLAB Simulink Umgebung integriert. Hinzu kommt mit `priority_in` die Priorität der empfangenen Daten.

[17] M-Dateien beinhalten MATLAB konforme Skripte, mit denen, z. B. externe Quelltexte eingebunden werden können. Hierbei wird beispielsweise die Darstellung in MATLAB Simulink und die zu übergebenden Datentypen definiert.

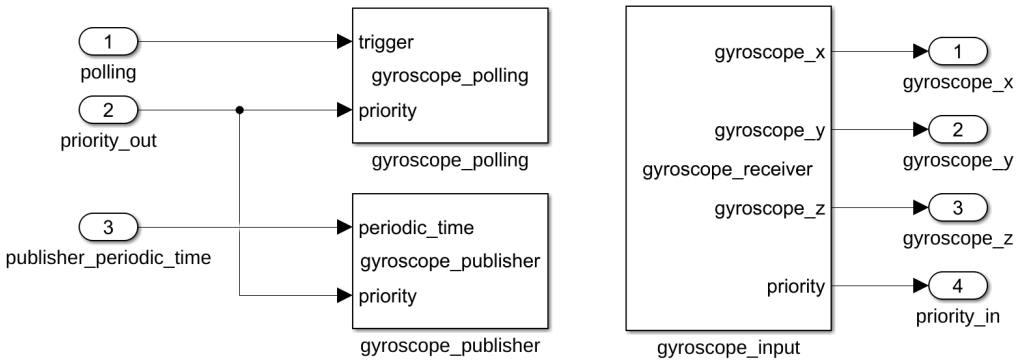


Abbildung 5.10: Beispiel der Einbindung eines Sensors in MATLAB Simulink

Die so integrierten Sensoren und Aktuatoren werden bei der Implementierung des *reflektorischen Operators* in MATLAB Simulink logisch miteinander verknüpft. Hierzu wird insbesondere Stateflow verwendet.

Anwendungsbeispiel

Abbildung 5.11 zeigt die Realisierung eines Zustandsautomaten mit Stateflow am Beispiel der adaptiven CAN-Publisher Transferrate und Priorität. Exemplarisch ist dies für den nach vorne und nach hinten gerichteten Ultraschallsensoren abgebildet. Der hier betrachtete Zustandsautomat mit Zuständen für vorne (`Ultrasonic_Front`) und hinten (`Ultrasonic_Rear`) werden parallel ausgeführt. Die parallele Ausführung wird in Stateflow mittels gestrichelter Linien um die jeweiligen Zustände dargestellt. Zudem zeigen `1` und `2` die Ausführungsreihenfolge, d. h. hier werden zunächst die Transitionen für den nach vorne blickenden Sensor geprüft. Initial befinden sich beide Sensoren im Zustand *Ultrasonic_Front / Rear_Standby*. Die Priorität ist niedrig (`3`), wenn sich der Roboter nicht nach vorne (front Sensor) und respektive nach hinten (hinterer Sensor) bewegt. Zudem beträgt die Transferrate 1 s (`1000 ms`). Sobald sich der Roboter in Blickrichtung des Sensors bewegt `[motor_velocity_x_in > 0]` (für vorne), wechselt die Priorität auf mittel (`2`) und die Transferrate auf 0,1 s (`100 ms`). Dieses findet analog für `[motor_velocity_x_in < 0]`, sowie für die Bewegung des Roboters in Richtung `y` und θ und die Kombination aus diesen statt.

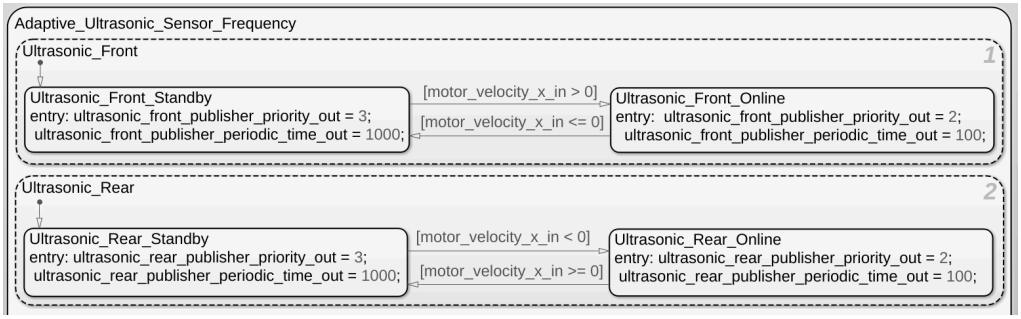


Abbildung 5.11: Implementierung der adaptiven CAN-Publisher-Transferrate und -Priorität mit MATLAB Stateflow

Ein weiteres Beispiel einer Umsetzung eines Zustandsautomaten ist in Anhang B, Abbildung A1 zu finden. Hierbei handelt es sich um die Geschwindigkeitsbegrenzung der Motoren bei der Annäherung an Hindernisse. Beträgt der Abstand zu einem Hindernis nur noch einen Mindestabstand von 0,2 m, so kann sich der Roboter nicht mehr in Richtung des Hindernisses bewegen. Um den Roboter nicht abrupt an diesem Stellwert stoppen zu müssen (hierbei müssten die Motoren der Mecanum Räder nicht in den Leerlauf wechseln, sondern die Räder vollständig blockieren), nimmt die maximal mögliche Geschwindigkeit bereits bei Annäherung an ein Hindernis ab. Hierbei nimmt die maximal mögliche Geschwindigkeit v_{max} zwischen dem Beginn des Annäherungsbereichs d_{an} (hier 0,8 m) und dem Mindestabstand d_{min} (hier 0,2 m) linear ab, sodass z. B. bei einem aktuellen Abstand d_{akt} von 0,4 m, der Roboter maximal mit $33.\overline{3}\,\%$ der möglichen Endgeschwindigkeit bewegt werden kann (siehe Gleichung 5.1).

$$\begin{aligned} v_{max}(d_{akt}) &= \frac{1}{d_{an}-d_{min}} * (d_{akt}-d_{min}) \\ v_{max}(0,4\,m) &= \frac{1}{d_{an}-d_{min}} * (0,4\,m-d_{min}) \\ &= \frac{1}{0,8\,m-0,2\,m} * (0,4\,m-0,2\,m) = \frac{1}{3} = 33.\overline{3}\,\% \end{aligned} \tag{5.1}$$

5.2.4 Watchdogs

Die Realisierung der internen Watchdogs und der Watchdogs für die Überwachung der Kommunikation werden implementiert, damit jeder Rechner des verteilten Systems unabhängig Fehlfunktionen feststellen und darauf reagieren kann. Der Ausfall der Kommunikation wird zeitabhängig, mittels des Analyse-Tools pulseAT[18] geprüft. Es wird ermittelt, wie lange der Erhalt der letzten Nachricht von jedem angebundenen und aktiven Rechner zurückliegt. Überschreitet diese Zeit einen Grenzwert, so wird der *reflektorische Operator* mittels einer Fehlermeldung darauf hingewiesen. Der *reflektorische Operator* unterbricht nun die aktuellen Aufgaben, stoppt jede Bewegung des Roboters und veranlasst den betreffenden Rechner mittels Relais neu zu starten. Werden auch nach dem Neustart des nicht erreichbaren Rechners keine Nachrichten empfangen, so werden die Aufgaben des Roboters abgebrochen und in einen Fehlermodus gewechselt, bis das Problem extern, d. h. von Nutzer*innen behoben wurde.

Die internen Watchdogs überprüfen die Informationen der lokal angebundenen Sensoren und Aktuatoren. So kann z. B. die verbleibende Akkumulatorkapazität durch den *Health Controller* gemessen werden. Sowohl die internen Watchdogs, als auch die Kommunikations-Watchdogs durchlaufen ein dreiphasiges Muster aus Warnung, Fehler und Beobachten, welches durch einen Zustandsautomaten abgebildet werden kann (siehe Abbildung 5.12). Initial beobachtet der Watchdog einen Sensor/ Aktuator oder die Verbindung zu einem anderen Rechner. Wird der Grenzwert für eine Warnung unterschritten, z. B. Akkumulatorkapazität kleiner als 10 %, so wechselt der Watchdog in den Zustand `Warnung`. Hierbei wird beim Eintritt in den Zustand (siehe `E:` in Abbildung 5.12) eine Nachricht nach dem in Tabelle 5.2 gezeigten Aufbau versendet. Die Priorität der Warnung wird entsprechend auf mittlere Priorität (`2`) konfiguriert. Es folgt die Watchdog Komponenten ID. Diese Identifikatoren sind fest definiert und eindeutig vergeben. Jede Komponente hat eine einzige Watchdog Komponenten ID für interne Watchdogs, sowie zusätzliche Identifikatoren für Kommunikations-Watchdogs (je eine für jede Verbindung). Es folgt, in *data[0]* der Fehlertyp, hier eine Warnung (`1`). Bei internen Watchdogs folgt im nächsten Datenfeld (*data[1]*) die bekannte und eindeutige ID des Sensors oder Aktuators, damit das Watchdog Ereignis dem jeweiligen Sensor oder Aktuator zugeordnet werden kann. Beim Unterschreiten des Grenzwertes für einen Fehler, z. B. wenn die Akkumulatorkapazität unter 5 % fällt, wechselt der Watchdog in den Zustand `Fehler`. Dies führt dazu, dass neben dem Versenden der entsprechenden Nachricht, eine aktive Aktion der Komponente folgt. Der *Health Controller* schaltet beispielsweise alle Verbraucher ab, um den Akkumulator nicht vollständig zu entladen und damit einen Schaden an diesem abzuwenden. Dieser Fehler kann nur von Nutzer*innen gelöst werden, da zur Behebung des Problems der Akkumulator aufgeladen werden muss.

[18] wird nachfolgend in Kapitel 6 erläutert

C/S Bit	Priorität	Identifikator	data[0]	data[1]
Sensor (1)	Abhängig vom Fehlertypen: 1 (Hoch): Fehler 2 (Mittel): Warnung 3 (Niedrig): Fehler/ Warnung gelöst	Watchdog Komponenten ID	[Fehlertyp] 1: Fehler 2: Warnung 3: Fehler/ Warnung gelöst	Bei internen Watchdogs: Sensors/ Aktuator ID

Tabelle 5.2: Watchdog CAN-Nachrichten

Im Normalbetrieb sollten die Watchdogs stets im Zustand `Beobachte` sein und somit sollte z. B. die Zwangsabschaltung des *Health Controllers* aufgrund von sehr niedriger Akkumulatorkapazität nie erfolgen müssen. Bereits vor dem Unterschreiten des Grenzwertes für den Wechsel in den Zustand `Warnung` hat der DAEbot das aufkommende Problem erkannt und darauf reagiert, indem z. B. der Roboter bereits bei einer Restkapazität von 15 % die Nutzer*in zum Aufladen des Akkumulators aufgefordert hat.

Anwendungsbeispiel

Die Watchdogs für die Ultraschallsensoren, welche an den *Motion Controller* angebunden sind, wechseln in den Zustand `Warnung`, sobald der Messwert auf unter 20 cm fällt (siehe Abbildung 5.12). Wird diese Warnung missachtet und der Roboter nähert sich weiter, so wird der Zustand `Fehler` bei einem Abstand von unter 10 cm erreicht. Nun werden die Motoren, welche ebenfalls an den *Motion Controller* angeschlossen sind, unabhängig vom *reflektorischen Operator* gestoppt. Es ist ebenso möglich, direkt vom Zustand `Beobachte` in den Zustand `Fehler` zu wechseln, falls der Grenzwert direkt 10 cm unterschreitet. Die `1` im Zustandsdiagramm gibt hierbei an, dass diese Transition als erstes geprüft wird, bevor die Prüfung der Transition `2` erfolgt. Es sei erwähnt, dass sich der DAEbot im Normalfall nie näher als 20 cm einem Hindernis nähert, dass dies bereits von der Geschwindigkeitsbegrenzung, welche der *reflektorische Operator* implementiert, unterbunden wird (siehe Anhang B, Abbildung A1).

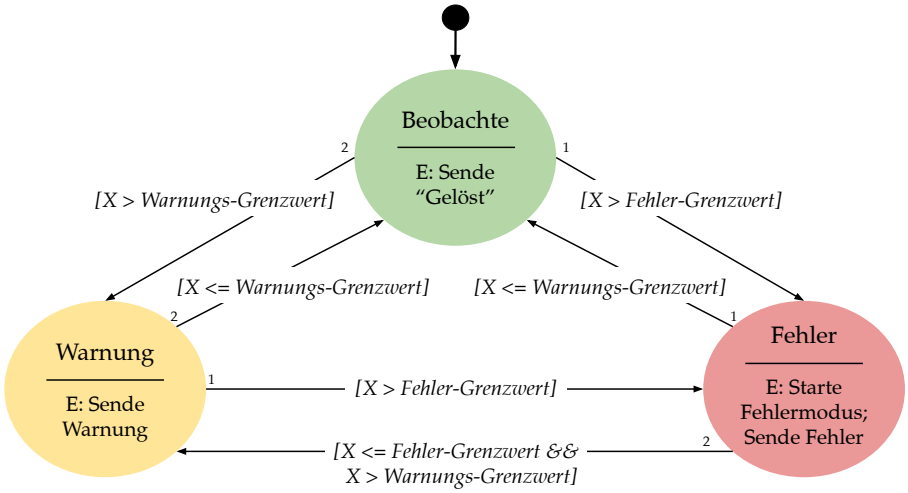


Abbildung 5.12: Watchdog Zustandsautomat

5.3 Implementierung der Komponenten

Der DAEbot als Demonstrator der OCM Schichtenarchitektur besteht aus diversen Komponenten, deren Zwecke und Aufgaben bereits in Kapitel 4.3 dargestellt wurden. Nachfolgend werden beispielhaft einige Implementierungen zu den einzelnen Komponenten des verteilten Systems dargestellt, wenngleich die vollständige Dokumentation der Implementierung des DAEbots nicht Ziel und Teil der vorliegenden Dissertation ist.

5.3.1 Motion Controller

Der *Motion Controller* bindet die Sensoren und Aktuatoren zur Bewegung des Roboters an. Es handelt sich hierbei um eine Komponente der *Controller*-Ebene, welche im hierarchisch unterteilten OCM in der untersten Ebene angesiedelt ist. Entsprechend ist der *Motion Controller* auch physisch direkt auf der Basis des DAEbots zu finden (siehe Abbildung 5.13).

Als Rechner wird ein STM32F3 Discovery SBC eingesetzt, welcher preisgünstig ist, einen internen Beschleunigungssensor, ein Gyroskop, ein Magnetometer und zahlreiche GPIO Pins bereitstellt. Zur Anbindung der Motortreiber, Motor-Encoder, Ultraschallsensoren und einer CAN-Bus-Schnittstelle, wurde ein Trägerboard im Rahmen eines, vom Autor konzipierten und betreuten, studentischen Projektes [B1] entwickelt, dessen Layout in

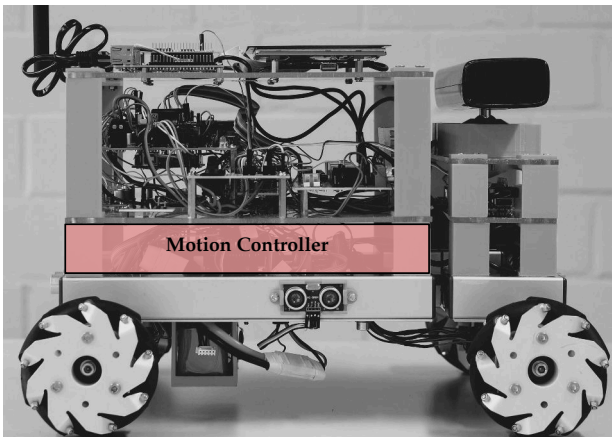


Abbildung 5.13: *Motion Controller* des DAEbots

Anhang C, Abbildung A4 dargestellt ist. Das Trägerboard enthält beispielsweise die notwendigen Transmitter für die CAN Kommunikation.

Die softwareseitige Implementierung erfolgte in zwei studentischen Arbeiten [B4, B6], welche jeweils vom Autor konzipiert und betreut wurden. Ein Fokus der Implementierung des *Motion Controllers* liegt auf der Modularität, auch in Bezug auf das Betriebssystem, da hier mit ChibiOS und FreeRTOS zwei geeignete Varianten zur Verfügung stehen. Die Modularität wird durch die Implementierung einer Software-Zwischenschicht zwischen dem Betriebssystem und den Treibern der angebundenen Sensoren und Aktuatoren erreicht, welche als Peripheral Drivers Abstraction Layer (PDAL) bezeichnet werden. Diese PDAL-Schicht bindet die Hardware Astraction Layer (HAL) der unterschiedlichen Betriebssysteme ein. Da sich der HAL, der hier getesteten Betriebssysteme nur im Detail unterscheidet, prinzipiell jedoch gleich aufgebaut ist, ist die realisierte PDAL-Schicht vergleichsweise simpel. Weiterhin vereinfacht die Trennung der Sensoren vom Betriebssystem auch den Austausch der Sensoren und Aktuatoren auf andere Rechner oder andere Komponenten (mit anderen Rechnern) und steigert somit die Modularität.

Abbildung 5.14 zeigt die Einbindung der PDAL Zwischenschicht zwischen den Treibern für die Hardware Module und den Betriebssystemen. Der *Motion Controller* wird jeweils mit den Betriebssystemen ChibiOS und FreeRTOS getestet. Beide Betriebssysteme sind besonders gut für eingebettete Systeme mit limitierten Hardwareressourcen geeignet. Der Fokus beider Betriebssysteme liegt auf der Echtzeitfunktionalität, insbesondere FreeRTOS wurde diesbezüglich in wissenschaftlichen Veröffentlichungen evaluiert [192]. Zur

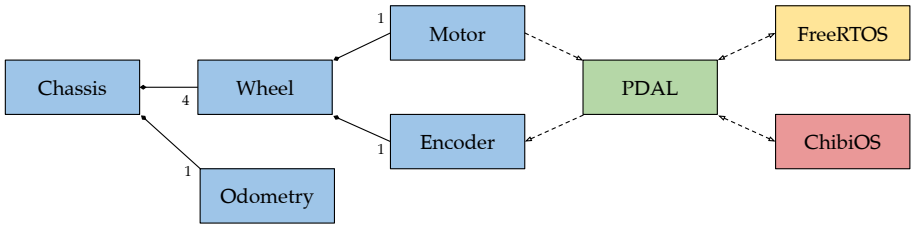


Abbildung 5.14: PDAL am Beispiel des *Motion Controllers*

Erfüllung der Echtzeitanforderungen bieten beide Betriebssystem-Grundfunktionen, wie z. B. Scheduler. Beispielprojekte für übliche CPUs oder Entwicklungsboards sind ebenfalls sowohl für ChibiOS, als auch für FreeRTOS verfügbar. Die Implementierung und der Test beider Betriebssysteme waren erfolgreich. Beide Betriebssysteme können uneingeschränkt für die Realisierung des *Motion Controllers* verwendet werden, sodass die Auswahl von FreeROTS letztendlich eine Frage der persönlichen Präferenz ist.

Das Konzept der Implementierung der Software des *Motion Controllers* zielt auf die Erfüllung der harten Echtzeitanforderungen im OCM ab. Hierbei basieren die logische Verknüpfung bzw. die Ab- und Reihenfolge der Software Klassen und Tasks auf dem Publisher-Subscriber Prinzip und der Umsetzung der entsprechenden Middleware der CAN-Schnittstelle. Hierbei ist die CPU des STM32F3 Discovery ein limitierender Faktor. Da nur ein Prozessorkern zur Verfügung steht, ist die Konfiguration der Prozesse im Scheduling von übergeordneter Bedeutung. Hier wird der Task zur zyklischen Abfrage von neuen CAN-Daten-Frames priorisiert, da der *Motion Controller* auf Befehlsnachrichten zum Anpassen der Fahrgeschwindigkeit schnellstmöglich reagieren muss.

Die Implementierung der Softwareklassen zum Auslesen der Sensoren und zum Stellen der Aktuatoren ist Stand der Technik. So wird beispielsweise eine inverse Kinematik zur Berechnung der Drehbewegung der einzelnen Mecanum-Ränder mittels Jacobi-Matrix und Odometrie berechnet [184]. Die `Chassis` Klasse (siehe Abbildung 5.14) berechnet jene inverse Kinematik unter Berücksichtigung der aktuellen Messwerte der `Odometrie`. Die `Chassis` Klasse bindet die vier Mecanum Räder (`Wheel`) ein. Die Klasse `Wheel` wiederum bestimmt mit der `Motor` Klasse die Beschleunigung eines Rades mittels Pulsweitenmodulation (PWM). Des Weiteren werden die Encoder Messwerte (`Encoder`) eingelesen. Die `Motor` und `Encoder` Klasse binden jeweils via PDAL die entsprechenden Motortreiber und Motor-Encoder unter Zuhilfenahme der HAL Betriebssystemfunktionen (wie beispielsweise die GPIO Softwarebibliotheken) ein.

Parallel erfolgen die Berechnung der Messwerte der Ultraschallsensoren, sowie die Abfrage der Messergebnisse des Beschleunigungssensors, des Gyroskops und des Magnetometers. Diverse Watchdogs überwachen die CAN Verbindungen zum *reflektorischen Operator* und, während der Fahrt, die Abstände zu Hindernissen.

5.3.2 Health Controller

Dem *Health Controller* des DAEbots kommt eine besondere Bedeutung zu, da dieser im Vergleich zum *Motion* und *Perception Controller* untypisch ist, d. h. aktuell in dieser Form nicht im typischen mobilen Roboter zu finden ist. Der *Health Controller* erfasst mittels Sensorik den aktuellen Zustand des DAEbots und kann zudem mittels Relais einzelne Komponenten des verteilten Systems ein- und ausschalten. Weiterhin implementiert der *Health Controller* diverse unterschiedliche Spannungsausgänge, z. B. für die Kinect (12 V) oder die SBC (5 V). Das Konzept des *Health Controllers* stammt vom Autor, die Realisierung der prototypischen Hardware wurde in einer studentischen Arbeit [B2] durchgeführt. Die Software entstand im Rahmen einer weiteren studentischen Arbeit [B3], welche ebenfalls vom Autor betreut wurde.

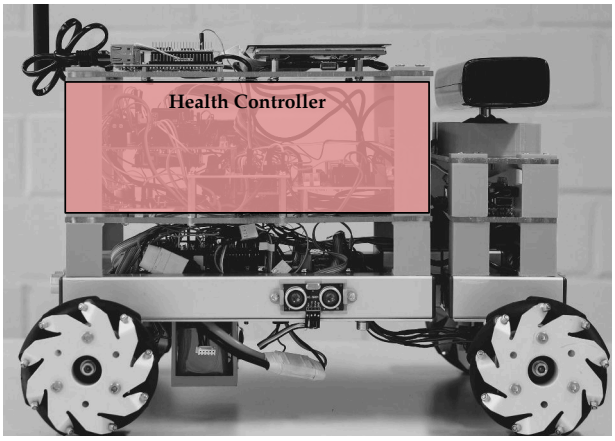


Abbildung 5.15: *Health Controller* des DAEbots

Der *Health Controller* macht aufgrund einer Ausstattung mit Relais und Leistungselektronik einen beträchtlichen Teil des Gesamtvolumens des DAEbots aus (siehe Abbildung 5.15). Als Rechner kommt ein Arduino Mega 2560 zum Einsatz, da dieser über viele GPIO Pins verfügt und somit besonders gut für Erweiterung durch zusätzliche Hardware geeignet ist. Diese

Hardware Erweiterungen des *Health Controllers* sind umfangreich (siehe Abbildung 5.16). Das Arduino Mega 2560 Board wird durch ein kommerziell erhältliches CAN Shield erweitert [O49].

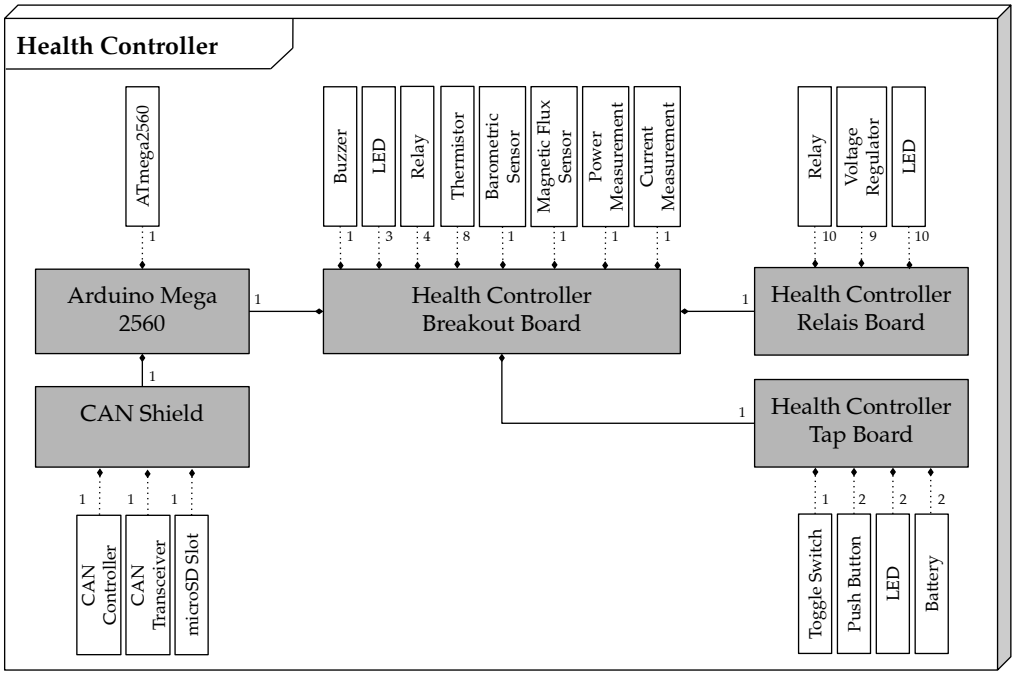


Abbildung 5.16: Hardwarekomponenten des *Health Controllers*

Die Einbindung aller anderen Sensoren und Aktuatoren an das Arduino-Board erfolgt über ein Trägerboard, welches im Rahmen des DAEbots entstanden und dessen Layout in Anhang C, Abbildung A5 abgebildet ist. Zur Sicherung der Bauteile gegen Überhitzungen und zur Analyse der Hitzeentwicklungen sind acht Thermistoren[19] auf dem DAEbot verteilt und an den *Health Controller*, bzw. dessen Trägerboard, angeschlossen. Diese befinden sich an potenziell kritischen Punkten, wie z. B. den Motortreibern. Zur Überwachung der Umgebung, welche den Zustand des DAEbots beeinflussen könnte, sind Sensoren zum Erfassen der Barometrie, d. h. der Luftfeuchtigkeit, des atmosphärischen Drucks, der

[19] Ein Thermistor ist ein temperaturabhängiger elektrischer Widerstand, aus dessen reproduzierbarer Widerstandsveränderung eine Temperatur abgeleitet werden kann.

Umgebungstemperatur[20] und des magnetischen Feldes[21], Bestandteil des *Health Controllers*. Etwaige Warnungen oder Signale werden mittels diverser LEDs und einem Summer kommuniziert. Zudem verfügt das Trägerboard des *Health Controllers* über eine Überwachung der Energieversorgung. Hierbei können sowohl die aktuelle Spannung, als auch die aktuell abgerufene Leistung gemessen werden. Durch die hieraus abgeleitete Restkapazität und die aktuelle Leistung kann der *Health Controller* eine Aussage über den Status der Energieversorgung treffen.

Mittels einer weiteren Hardware Erweiterung, dem Relais-Board[22], stellt der *Health Controller* die Spannungsversorgung mittels Spannungsreglern für andere Rechner des verteilten Systems und weitere Peripherie zur Verfügung (siehe Abbildung 5.16). Hierbei handelt es sich beim Großteil der Versorgungsanschlüsse um USB-Buchsen, welche eine Spannung von 5 V liefern und zur Versorgung aller (Kleinst-) Rechner verwendet werden. Zudem versorgt ein 12 V-Spannungsausgang die Kinect Stereokamera, welche am *Perception Controller* angebunden ist. Weitere Ausgänge sind für zukünftige Erweiterungen verfügbar und lassen sich mittels Jumpern auf 5 V, 12 V oder die Akkumulatorspannung (14,8 V) einstellen. Ein Großteil des Relais-Boards wird für die Unterbringung von zehn Relais[23] benötigt. Mit diesen können die Versorgungs-Ausgänge ein- und ausgeschaltet werden. Der aktuelle Status eines jeden Relais wird über eine LED Leiste angezeigt. Die für die Spannungsregulierung benötigten Kühlkörper sind ebenfalls auf dem Relais-Board realisiert.

An das sogenannte Tap[24]-Board wird der Akkumulator des DAEbots angeschlossen[25]. Durch eine Sicherung wird die Stromversorgung gegen einen Kurzschluss geschützt. Das Tap Board greift den Akkumulator über den Hauptstecker ab (4 Zellen, 14,8 V) und nutzt außerdem den Balancer-Stecker[26] zum Abgreifen der einzelnen Zellen (je Zelle 3,7 V). Somit werden die 5 V USB-Buchsen des Relais-Boards, welche vergleichsweise wenig elektrische Leistung zur Verfügung stellen müssen, aus der Spannung von zwei Zellen (7,4 V) gespeist. Die Spannungswandlung von 7,4 V auf 5 V ist hierbei effizienter als das Wandeln der gesamten Akkumulatorspannung (14,8 V) auf 5 V, wenngleich die Gefahr von unterschiedlicher Zellspannung im Akkumulator zu einer Beschädigung führen kann. Im Einsatz des DAEbots hat sich jedoch gezeigt, dass die geringe benötigte elektrische Leistung zu keinen Beeinträchtigungen des Akkumulators führt. Verbraucher mit einem

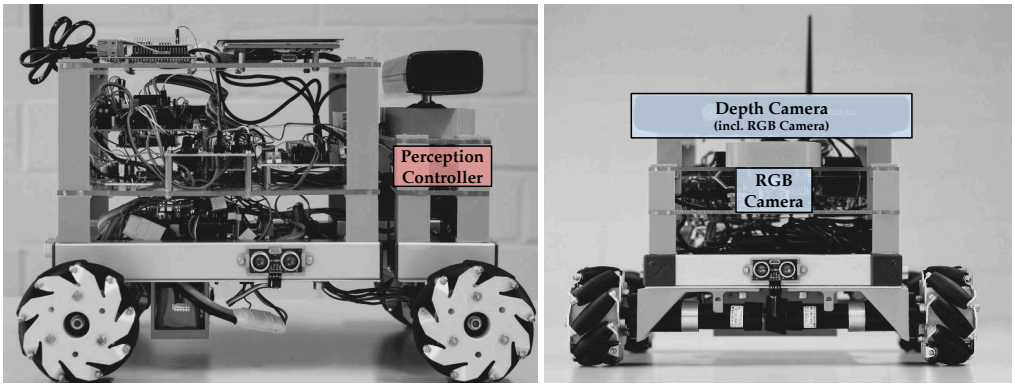
[20] DEBO BME280 [O50]
[21] Honeywell SS49E [O51]
[22] Das Layout des *Health Controller* Relais-Boards ist in Anhang C, Abbildung A6 abgebildet.
[23] Omron G5LE-1-VD [O52]
[24] engl. abgreifen
[25] Das Layout des *Health Controller* Tap Boards ist in Anhang C, Abbildung A7 abgebildet.
[26] Über den Balancer (dt. Ausgleichsregler), u. a, eines Lithium-Polymer-Akkumulators wird die Zellspannung der einzelnen Zellen geregelt. Hierzu wird der Balancer-Stecker zusätzlich an das Ladegerät angeschlossen, welches die Spannungen der einzelnen Zellen lesen kann und einzelne Zellen laden/ entladen kann.

höheren benötigten Leistungsbedarf, wie z. B. die Motoren, werden stets aus allen vier verfügbaren Zellen gespeist. Mittels eines Hauptschalters kann die Verbindung des Akkumulators zum Tap-Board getrennt werden. Ein Taster führt Startroutinen durch den *Health Controller* durch. Ein kurzer Klick startet nur das Relais des *reflektorischen Operators*, mit einem langen Klick werden alle Relais und somit alle Komponenten des DAEbots eingeschaltet. Der zweite Taster wird zum Ein- und Ausschalten des am *reflektorischen Operator* angeschlossenen Displays verwendet.

Die Software der Rechner der Arduino Familie kommt typischerweise ohne Betriebssystem aus, indem nach dem Start des Rechners einmalig ein Setup durchgeführt wird und nachfolgend alle Befehle in einer Schleife wiederholt werden. Die beim OCM benötige harte Echtzeitfähigkeit ist hierbei nur bedingt erfüllt, da im kritischsten Fall auf die Ausführung eines Programmbefehls nahezu ein gesamter Zyklus gewartet werden muss. Buonocunto, Biondi, Pagani, Marinoni und Buttazzo zeigen mit ARTE [193] die Anwendung einer echtzeitfähigen Erweiterung für Multitasking Anwendungen mit Arduinos auf. ARTE dient der vom Autor konzipierten und betreuten studentischen Arbeit [B7] als Inspiration zur Implementierung eines eigenen Arduino Schedulers. Dieser Scheduler basiert auf dem Konzept der Protothreads [194], welche das Springen zwischen einzelnen Threads an fest definierten Programmstellen erlaubt. Die zeitliche Basis des Schedulers bildet ein interner ATmega2560 Timer. Tasks wird eine Priorität und eine Periodizität, d. h. die Anzahl an Aufrufen pro Zyklus, zugeordnet, sodass Software Tasks mit hoher Priorität, wie der Task zum Abrufen der Befehlsnachrichten, häufiger aufgerufen werden können. So wird z. B. der Befehl zum Ein- und Ausschalten eines Relais schnell erkannt und priorisiert ausgeführt.

5.3.3 Perception Controller

Der *Perception Controller* ist eine weitere Komponente der *Controller* Ebene. Das Ziel des *Perception Controllers* ist die Wahrnehmung der Umgebung. Der *Perception Controller* befindet sich an der Front des DAEbots (siehe Abbildung 5.17a). Dieser *Controller* ist nicht zwingend erforderlich, denn der Roboter kann rudimentär auch allein mit den Abstandssensoren des *Motion Controllers* fortbewegt werden. Eine Erweiterung des Systems um leistungsfähigere Sensoren zur Erfassung der Umgebung, ist jedoch üblich und sinnvoll. Bei jenen Sensoren handelt es sich um eine RGB Kamera [O53] und eine Kinect Stereokamera von Microsoft (siehe Abbildung 5.17b). Die Verbindung der Kameras erfolgt mittels USB (Kinect) und der 15 poligen seriellen Camera Serial Interface (CSI)-Schnittstelle (RGB Kamera). Die bedarfsgesteuerte Nutzung der Kamera hängt von der Situation ab (bedarfsgesteuerte Komponentennutzung). Als Basis des *Perception Controllers* wird ein Raspberry Pi 3 Model B+ eingesetzt, welches um ein CAN Erweiterungsmodul auf Basis eines MCP2515 [O54] ergänzt wird. Die Herausforderung bei der Implementierung des *Perception Controllers* liegt in der Erfüllung der Echtzeitfähigkeit bei der vergleichsweise wenig leistungsstarken Hardware für die Anbindung von visuellen Sensoren.



(a) (b)

Abbildung 5.17: *Perception Controller* des DAEbots

Die Implementierung der Software erfolgte in einer studentischen Arbeit [B6], welche vom Autor konzipiert und betreut wurde. Eine Herausforderung bei der Umsetzung des *Perception Controllers* liegt in der Realisierung der Echtzeitfähigkeit auf dem Raspberry Pi SBC, welcher typischerweise mit nicht echtzeitfähigem Raspberry Pi OS [O55] (ehemals Raspbian) verwendet wird. Zur Realisierung dieser Echtzeitfähigkeit wird eine Linux Distribution mittels des *Yocto* Projektes [195, O56] erstellt. *Yocto* ist ein Open-Source Projekt, welches Entwickler*innen die Erstellung einer eigenen Linux Distribution stark vereinfacht. Hierzu wird zunächst ein Linux Kernel ausgewählt. Anschließend erfolgt die Konfiguration der notwendigen Bibliotheken, welche der eigenen Distribution hinzugefügt werden sollen. Zudem können eigene Konfigurationen, z. B. die Boot-Sequenz für das MCP2515 Modul, eingebracht werden. Zusätzlich zur so entstehenden Linux Distribution, welche nachfolgend als `DAEbot_Perception_CO_OS` bezeichnet wird, wird ein SDK extrahiert, welches zur Entwicklung der Applikation für das `DAEbot_Perception_CO_OS` benötigt wird.

Abbildung 5.18 zeigt die wichtigsten Module und Komponenten des *Perception Controllers* als Komponentendiagramm. Das `DAEbot_Perception_CO_OS` enthält Erweiterungen zur Erfüllung der Echtzeitfähigkeit durch die Nutzung eines Echtzeit Linux Kernels [196]. Für die beiden visuellen Sensoren sind entsprechende Module in das Betriebssystem integriert. Darauf aufbauend liegt die Middleware, welche die softwareseitige Integration der CAN-Schnittstelle und der Kamera-Streams beinhaltet.

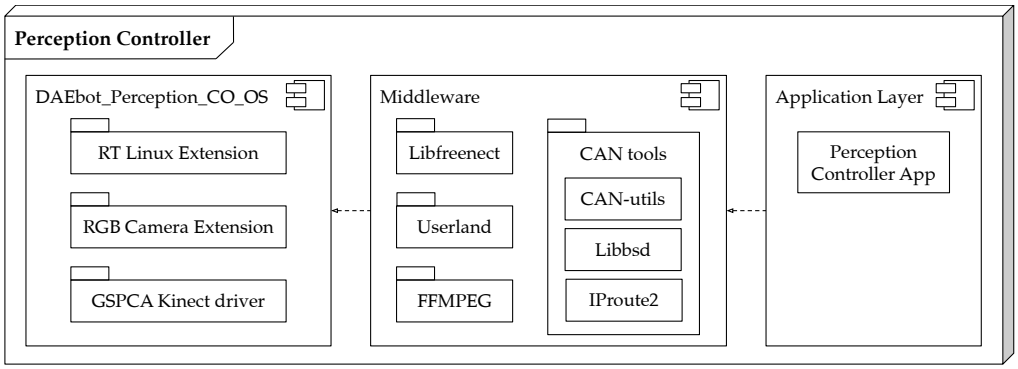


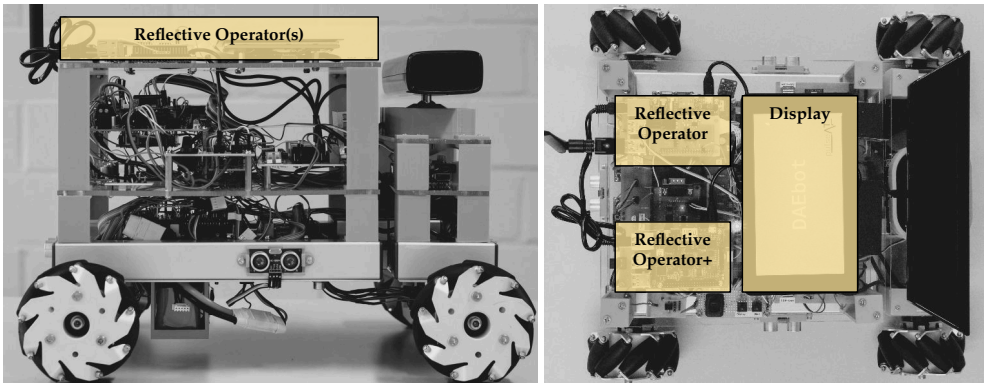
Abbildung 5.18: Komponentendiagramm des *Perception Controllers* (basiert auf [B6])

`Libfreenect` [O57] stellt die Treiber zur Anbindung der Kinect bereit. Dieses beinhaltet nicht nur die Videostreams (RGB und 3D-Video), sondern auch die Anbindung der Mikrofone, des integrierten Beschleunigungssensors, der LED und des integrierten Motors zum Ausrichten der Kamera. Die Raspberry Pi `Userland` Bibliothek [O58] stellt u. a. Interfaces für das hardwarebeschleunigte Video Encoding bereit. `FFMPEG` [O59] wird zur Aufnahme und Konvertierung der Video- und Audio-Streams verwendet. Hierzu wird zudem die Erweiterung `OpenMAX` verwendet, welches die CPU Auslastung signifikant verringert. Die benötigten Treiber für die CAN Erweiterung sind in `CAN-utils` [O60] vorhanden. `Libusb` [O61] wird ebenso für die CAN-Schnittstelle benötigt, wie `IProute2`, welches Teil des Linux Kernels ist.

Die Applikation des *Perception Controllers* hat drei Hauptaufgaben. Die Erste ist die Verarbeitung des CAN-Nachrichten Frames. Hierüber sendet der *Perception Controller* u. a. die Position eines detektierten AprilTags an den *reflektorische Operator*. Die zweite Aufgabe ist entsprechend die Auswertung des Video-Streams mit OpenCV [116]. Die dritte Aufgabe ist die Verbindung des *Perception Controllers* zum *reflektorischen Operator+* via Ethernet und die Übertragung des Video-Streams an den *reflektorischen Operator+*, welcher den Video-Stream parallel mittels einer parallel verarbeitenden FPGA Anwendung auswertet. Aufgrund der begrenzten Bandbreite der Ethernet Verbindung von 100 Mbit/s, muss der Video-Stream vor dem Bereitstellen komprimiert werden. Dieses erfolgt mit dem Videokompressionsstandard H.264 [197], welches die Hardwarebeschleunigung des Raspberry Pi unterstützt und somit nur eine geringe CPU-Auslastung zur Folge hat.

5.3.4 Reflektorischer Operator

Der *reflektorische Operator* ist auf der mittleren Ebene des OCM und der höchsten Ebene des lokalen Klienten zu finden. Der *reflektorische Operator* ist somit auch physisch oben auf dem DAEbot zu finden (siehe Abbildung 5.19a). Die Ebene der *reflektorischen Operatoren* besteht aus dem *reflektorischen Operator+*, einem Display und dem hier beschriebenen *reflektorischen Operator* (siehe Abbildung 5.19b). Als Rechner kommt ebenfalls ein Raspberry Pi 3 Model B+ inkl. CAN Erweiterungsmodul und einem WLAN USB Stick mit großer Reichweite zum Einsatz, wobei hier keine harte Echtzeitfähigkeit notwendig ist und somit auf das Raspberry Pi OS zurückgegriffen wird. Beim Display handelt es sich um ein 7" LCD-Touch-Display [O62], welches direkt über das Display Serial Interface (DSI) des Raspberry Pi verbunden ist.



(a) (b)

Abbildung 5.19: *Reflektorische Operatoren* des DAEbots

Der *reflektorische Operator* implementiert die Koordination aller *Controller*, sowie die Schnittstelle zum *kognitiven Operator*. Die Auswertung aller Sensor-Werte der *Controller* erfolgt in einem `MATLAB Simulink (Stateflow) Model` (siehe Abbildung 5.20), welche die Toolbox zur modellbasierten Entwicklung (siehe Kapitel 5.2.3) einsetzt. Das Abrufen von CAN-Daten Frames und MQTT-Nachrichten erfolgt jeweils in einem eigenen Thread, welche, aufgrund des Mehrkernprozessors des Rechners, parallel ausgeführt werden. Einfache logische Verknüpfungen, insbesondere auf Hardware-Ebene werden in Simulink modelliert (siehe Beispiel in Anhang B, Abbildung A2). Die Realisierung der Zustandsautomaten erfolgt in Stateflow. Das Basismodell umfasst hierbei Funktionen, welche in jedem Use-Case des DAEbots benötigt werden. Hierzu zählen u. a. die adaptive CAN-Publisher Transferrate und Priorität, Safety Funktionen wie z. B. die Geschwindigkeitsregulierung in Bezug zu einem Hindernis, die bedarfsgesteuerte

Komponentennutzung, sowie verschiedene Roboter Modi (z. B. Standby, Energie sparen, volle Leistung). Use-Case spezifische Funktionen ergänzen das MATLAB Simulink Model oder nutzen weitere Dienste außerhalb der MATLAB Umgebung.

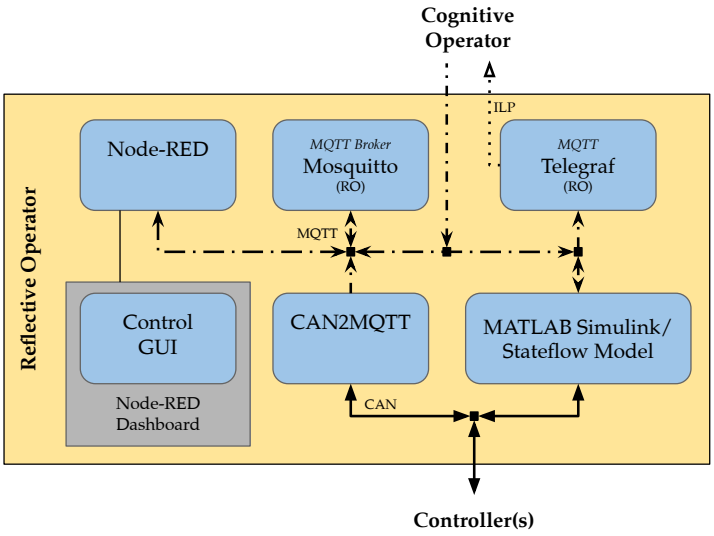


Abbildung 5.20: Dienste des *reflektorischen Operators*

Parallel zum MATLAB Simulink (Stateflow) Model werden Sensor und Aktuator-Daten mittels der in [B10] realisierten MQTT-Schnittstelle (siehe Kapitel 5.2.2) mit `CAN2MQTT` in MQTT-Nachrichten umgewandelt, von `Mosquitto` verwaltet, mit `Telegraf` eingesammelt und an den *kognitiven Operator* versendet (siehe Abbildung 5.20).

Zudem wird mit `Node-RED` [O63] ein Programmier-Tool für IoT Geräte verwendet, welches u. a. in [198] in Verbindung mit einem Raspberry Pi SBC eingesetzt wird. `Node-RED` ist ein Tool zur modellbasierten Entwicklung von IoT Geräten. Verschiedene Geräte können mit einem Baukastenprinzip zusammen geschaltet werden (siehe Beispiel in Anhang B, Abbildung A3). Hierbei werden in `Node-RED` alle in der IoT üblichen Dienste und Technologien unterstützt. Der Datentransfer zwischen `Node-RED` und den anderen Diensten des *reflektorischen Operators* erfolgt hierbei mittels MQTT-Nachrichten.

Bei der Realisierung des Demonstrators wird zudem das in [B10] entwickelte `Node-RED Dashboard` eingesetzt, welches eine GUI für einfache Ein- und Ausgaben bereitstellt. Das Dashboard wird als Webserver implementiert. Diese GUI wird beim DAEbot als Schnittstelle zur Mensch-Roboter Interaktion auf der Ebene des *reflektorischen Operators* verwendet.

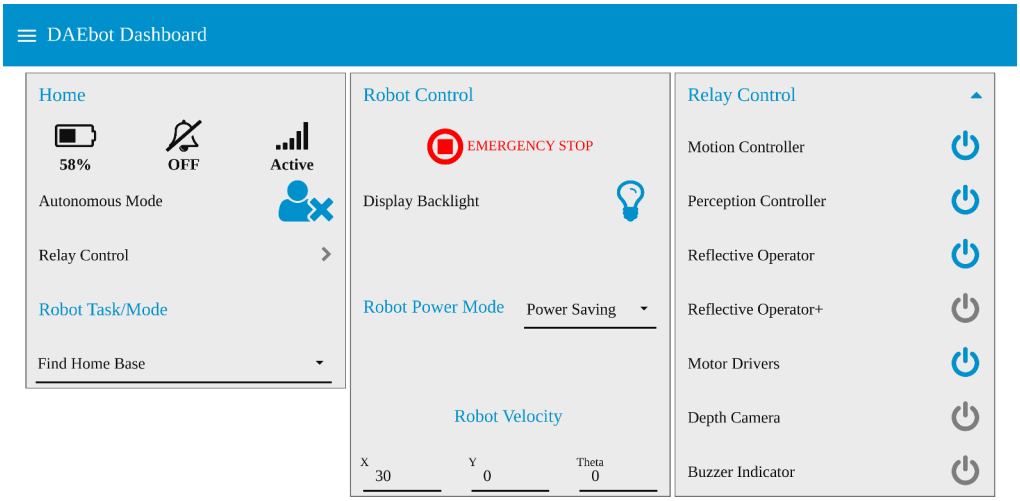


Abbildung 5.21: Lokales Dashboard des DAEbots (basiert auf [B10])

Diese H2M Schnittstelle wird auf dem Touch-Display des *reflektorischen Operators* angezeigt und kann zudem innerhalb des lokalen Netzwerkes per Fernzugriff angezeigt und bedient werden. Sie besteht zunächst prinzipiell aus drei Bereichen (siehe Abbildung 5.21). Die linke Spalte *Home* stellt den aktuellen Zustand des DAEbots in Form der Batteriekapazität und dem Status der Verbindung zur Datenbasis dar. Darunter kann ausgewählt werden, ob der DAEbot im autonomen oder manuellen Betrieb gesteuert wird. Im autonomen Betrieb erfolgen alle Einstellungen, Konfigurationen und letztendlich auch die Aufgabe des Roboters vom *reflektorischen* respektive *kognitiven Operator*. Hier kann im Dashboard eine Aufgabe ausgewählt werden, wie hier „Finde den Heim-Standort“. Der manuelle Betrieb öffnet die mittlere und rechte Spalte (welche im autonomen Betrieb ausgeblendet werden). Die mittlere Spalte dient zur Steuerung des DAEbots. Neben einem „Notaus“ kann die Geschwindigkeit des Roboters zu Testzwecken eingegeben werden. Zudem erfolgt die Auswahl des „Roboter Energiemodus“. Im „Standby“ werden alle Komponenten außer dem *Health Controller* und dem *reflektorischen Operator* ausgeschaltet. Im Modus „Volle Leistung“ sind alle Komponenten eingeschaltet. Typischerweise wird der „Energiesparmodus“ genutzt, welcher die Komponenten je nach Bedarf mittels der Relais aktiviert. Die Relais können zudem in der rechten Spalte manuell ein- und ausgeschaltet werden. Im dargestellten Beispiel bewegt sich der DAEbot in einer hellen Umgebung, sodass der *Motion*, *Health* und *Perception Controller* eingeschaltet und die 3D Stereokamera ausgeschaltet ist. Zudem werden die Motortreiber und der *reflektorische Operator* benötigt.

Insbesondere für den *reflektorischen Operator* werden zahlreiche weitere Tools zum Testen des DAEbots implementiert. Zu nennen sei hierbei u. a. die manuelle Steuerung des DAEbots mit einem Gamepad im teilautonomen Betrieb. Dabei sind die Safety Funktionen, beispielsweise zum Vermeiden von Kollisionen weiterhin aktiviert. Des Weiteren wird intensiv die Funktionalität des Publisher-Subscriber Prinzips der CAN-Schnittstelle mit eigenen Debug-Tools, insbesondere in Bezug auf die Einhaltung der zeitlichen Rahmenbedingungen, evaluiert.

5.3.5 Reflektorischer Operator+

Das Konzept, den *reflektorischen Operator* durch eine weitere (heterogene) Recheneinheit zu erweitern, soll mit dem *reflektorischen Operator+* evaluiert werden. Sowohl die Inbetriebnahme der Hardware [B5], als auch die Implementierung [B8] des *reflektorischen Operators+* erfolgte in zwei studentischen Arbeiten, welche vom Autor konzipiert und betreut wurden. Als Koprozessor zum *reflektorischen Operator* werden heterogene Rechner verwendet. Neben dem ARM-basierten Raspberry Pi SBC als *reflektorischer Operator* kommt als *reflektorischer Operator+* ein ZynqBerry 7010 SBC zum Einsatz, welches aus einem frei verfügbaren FPGA Bereich (Xilinx Programmable Logic (PL)[27]) und einem ARM Cortex-A9 CPU Bereich (Processing System (PS)) innerhalb des Zynq XC7Z010 SoPC besteht. Das ZynqBerry-Board wird durch CAN Transmitter (MCP2515) erweitert.

Das *Processing System* wird zur Anbindung der Kommunikations-Schnittstellen genutzt (siehe `System Handler` in Abbildung 5.22). Hier kommt mit PetaLinux [O64] von Xilinx ein auf Yocto basierendes Tool zum Einsatz, mit welchem eine eigene Linux Distribution erstellt wird. Das *Processing System* ist hauptsächlich für die Kommunikation mit dem *reflektorischen Operator* zuständig. Hierbei werden zwischen den beiden *reflektorischen Operatoren* Steuerbefehle und Ergebnisse aus der Bildverarbeitung der *programmierbaren Logik* via CAN versendet.

Das Konzept sieht vor, dass die *programmierbare Logik* den Videostream des *Perception Controllers* via Ethernet erhält und dort unmittelbar die Bildverarbeitung in der *programmierbaren Logik* des Zynq ausgeführt wird. So existieren im DAEbot zwei parallel und auf unterschiedlichen Komponenten ausgeführte Module zur Bildverarbeitung. Hierbei findet eine vergleichsweise einfache Bildverarbeitung im *Perception Controller* statt, dessen Ergebnisse via CAN an den *reflektorischen Operator* übertragen werden. Bei der über CSI verbundenen RGB Kamera liefert die Bildverarbeitung des *Perception Controllers* brauchbare Ergebnisse. Wird jedoch die über USB verbundene Kinect aktiviert, kann die 3D-Punktewolke nicht schnell genug verarbeitet werden. In diesem Fall oder auch wenn komplexere

[27] dt. programmierbare Logik

Bilderverarbeitungsalgorithmen ausgeführt werden sollen, wird der *reflektorische Operator+* aktiviert, die Bildverarbeitung des *Perception Controllers* unterbrochen und der Videostream des *Perception Controllers* an den *reflektorischen Operator+* weitergeleitet.

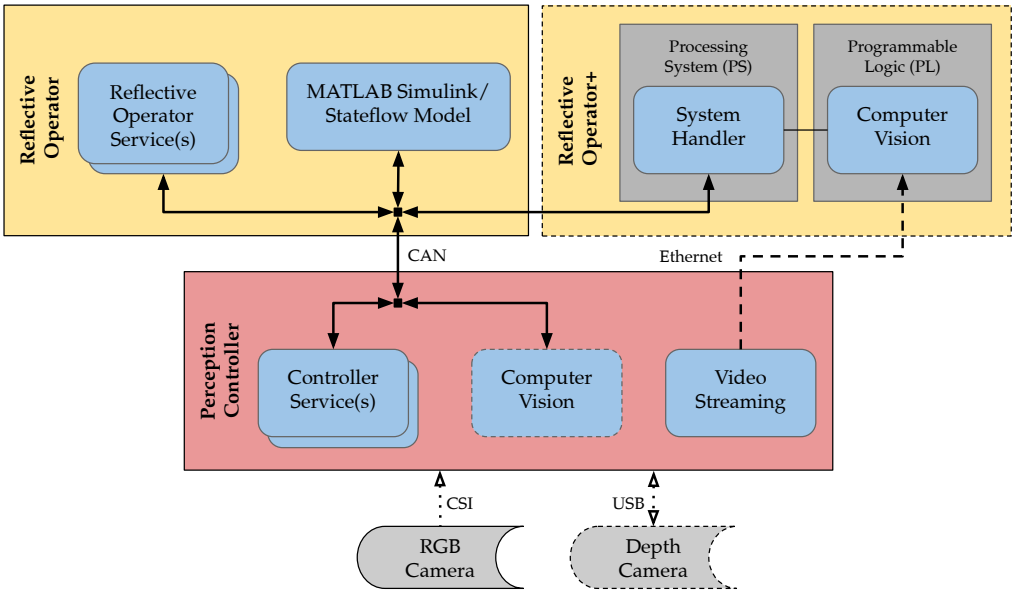


Abbildung 5.22: Einbindung des *reflektorischen Operator+*

Die exemplarische Implementierung der Bildverabeitung in der *programmierbaren Logik* des ZynqBerry mittels der von Xilinx veröffentlichen IP-Cores[28], scheiterte im Rahmen der studentischen Arbeit [B8] jedoch daran, dass die Einbindung des Videostreams über Ethernet/ TCP aufgrund von fehlenden oder fehlerhaften Bibliotheken und Treibern nicht erfolgen kann. Zwar kann das *Processing System* als Server für den Erhalt des Videostreams dienen und diesen an die *programmierbare Logik* weitergeben, hierbei bremst die notwendige Vorverarbeitung des *Processing Systems* die Bildverarbeitung des *reflektorischen Operators+* jedoch aus.

[28] Ein IP-Core ist ein vorgefertigter Funktionsblock im Chipdesign, welcher u. a. von Herstellen von FPGAs Entwickler*innen zur Verfügung gestellt werden kann. Als Hard-Cores werden fertige und unveränderbare Schaltung des Chips bezeichnet, während die hier als IP-Core bezeichneten Soft-Cores als Quelltext (Netzliste) vorliegen und zur Konfiguation u. a. von FPGAs verwendet werden.

Letztendlich kann das Konzept des *reflektorischen Operators+* u. a. aufgrund der ausgewählten Hardware nicht vollständig evaluiert werden, wenngleich in der Literatur prinzipiell Beispiele für eine erfolgreiche Hardwarebeschleunigung von Bildverarbeitung mit FPGA in Zusammenarbeit mit einer ARM basierten CPU erfolgversprechend sind [153, 154].

5.3.6 Kognitiver Operator

Das Verschieben des *kognitiven Operators* in die Cloud ermöglicht die Implementierung von Software und Tools, welche hohe Hardwareanforderungen benötigen. Zudem ist die Cloud, im Gegensatz zum lokalen Klienten, dauerhaft verfügbar. Des Weiteren spielt die Energieversorgung in der Planung der Software beim *kognitiven Operator* eine untergeordnete Rolle. Der *kognitive Operator* des DAEbots nutzt ebenfalls die MQTT Schnittstelle (siehe Kapitel 5.2.2). Sowohl die Middleware, als auch die Implementierung bzw. Konfiguration der nachfolgend gezeigten Software erfolgte in einer studentischen Arbeit [B10], welche vom Autor konzipiert und betreut wurde.

Bei der experimentellen Umsetzung der OCM Architektur für mobile Roboter liegt der Fokus bei der Realisierung des *kognitiven Operators* zunächst auf der Bereitstellung der notwendigen Datenbasis und Tools für die Optimierung und Missionsplanung des Systems. Hierbei bildet eine Time Series Database (TSDB)[29] den Mittelpunkt der Software Infrastruktur. Eine TSDB speichert zu jedem Datensatz zusätzlich einen Zeitstempel [199]. Diese TSDBs werden laut Nasar und Kausar [200] insbesondere im Bereich der IoT verwendet und sind zudem laut Bonnet, Gehrke und Seshadri [201] ebenfalls für Systeme mit vielen Sensoren geeignet.

Stand der Technik für TSDB ist aktuell die InfluxDB-Datenbank, welche Teil des TICK Stacks ist. Jener TICK Stack wird u. a. zur Überwachung von großen Datenmengen im ATLAS Projekt im Kernforschungszentrum CERN verwendet [202]. Der TICK Stack ist Open-Source und besteht in der Basis aus vier Modulen (siehe Abbildung 5.23), wenngleich diese nicht zwangsläufig alle verwendet werden müssen. `Telegraf` sammelt Daten aus diversen Quellen ein, transformiert diese in das ILP-Format und stellt die Daten u. a. der Datenbank `InfluxDB` zur Verfügung. `Chronograf` ist ein Interface zur Visualisierung der Datensätze. Mit `Kapacitor` steht ein Werkzeug für die Datenverarbeitung im TICK Stack zur Verfügung, mit dem einfache Algorithmen zur Benachrichtigung oder komplexere Algorithmen des maschinellen Lernens implementiert werden können.

[29] dt. Zeitreihen-Datenbank

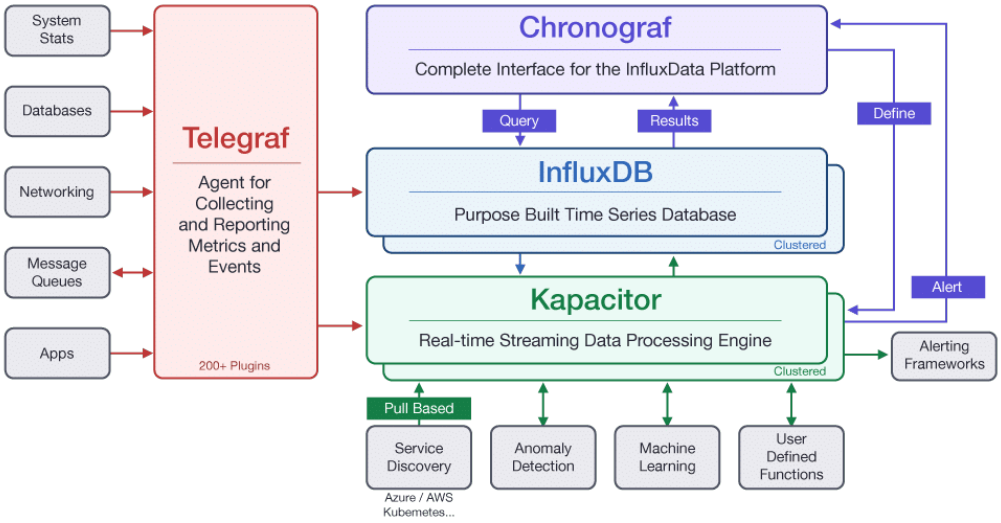


Abbildung 5.23: TICK Stack [O45]

Der *kognitive Operator* des DAEbots nutzt große Teile des TICK Stack und setzt zur Visualisierung neben `Chronograf` das Open-Source Tool `Grafana` [O65] ein, welches unabhängig vom TICK Stack ist, aber dennoch kompatibel zu diesem ist und im Vergleich zu `Chronograf` aktuell Vorteile in der Benutzerfreundlichkeit und dem Leistungsumfang hat. Die Implementierung des *kognitiven Operators* erfolgt für die exemplarische Umsetzung auf dem am Standort verfügbaren Server und wird als IaaS umgesetzt, wenngleich dies dem Umstand geschuldet ist, dass das Aufsetzen auf der PaaS-Ebene am Standort nicht vorgesehen ist. Somit dient eine Linux-basierte Ubuntu Server Distribution als Basis für die PaaS Implementierungen des *kognitiven Operators*.

Die Kommunikation mit dem *reflektorischen Operator* erfolgt beim DAEbot via WLAN. Die Daten vom *reflektorischen Operator* zum *kognitiven Operator* werden als ILP-Befehl zum Schreiben in die `InfluxDB`-Datenbank versendet (siehe Abbildung 5.24). Nachrichten des *kognitiven Operators* an den lokalen Klienten erfolgen als MQTT-Nachricht. Die Verwaltung der MQTT-Nachrichten erfolgen über den `Mosquitto` MQTT-Broker. Da dieser sowohl lokal, als auch auf dem cloudbasierten *kognitiven Operator* verfügbar ist, können MQTT-Nachrichten lokal auch ohne den DAEbot ausgetauscht werden. Das Einsammeln der MQTT-Nachrichten erfolgt mit `Telegraf`, welches die Nachrichten in die `InfluxDB`-Datenbank speichert.

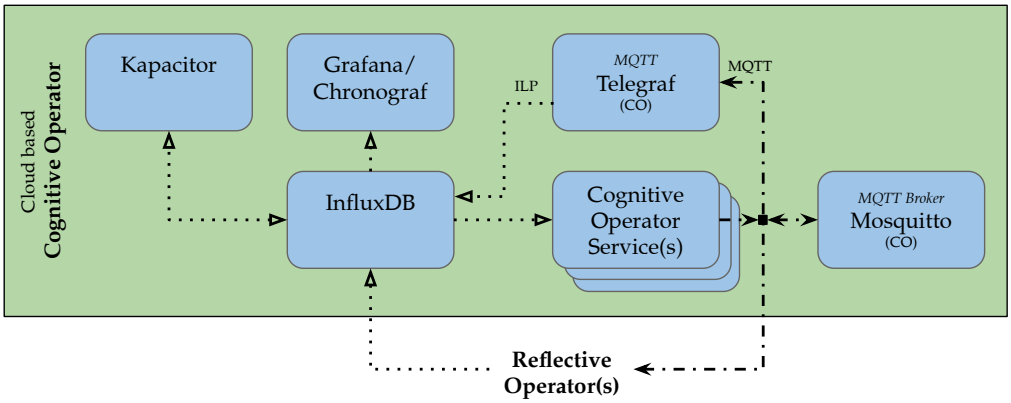


Abbildung 5.24: Dienste des *kognitiven Operators*

Mittels `Grafana` werden Sensorwerte der lokalen Komponenten des DAEbots dargestellt. Zu Testzwecken werden ebenfalls alle Befehlsnachrichten, wie die Publisher Konfigurationen an den *kognitiven Operator* übertragen und in der Datenbank geloggt. Vorteilhaft ist hierbei die Speicherung in der Cloud insbesondere deswegen, weil diese Befehle und deren Auswirkungen im Nachgang der Nutzung des DAEbots analysiert werden können. So wird `Grafana` insbesondere für die manuelle Analyse des Systems genutzt, indem u. a. Messwerte und Aktuator-Stellwerte visuell in Zusammenhang gebracht werden. Das in Abbildung 5.25 gezeigte Beispiel zeigt die Visualisierung der Stellwerte für die Geschwindigkeit des Roboters[30] und die u. a. davon abhängigen Messergebnisse für die elektrische Leistung[31] und den Temperaturen der Motortreiber. Die Korrelation wird insbesondere durch den Vergleich des Betrages der Stellwerte für die Geschwindigkeit (siehe zweiter Graph in Abbildung 5.25) deutlich. Im ersten Testlauf wird der Roboter mit maximal 50 cm/s bewegt, welches zu einer maximalen elektrischen Leistungsaufnahme von 41 W führt. Die Temperatur steigt von unter 20 °C auf ca. 30 °C an. Bei dem hier dargestellten zweiten Testlauf wird der Roboter mit seiner Maximalgeschwindigkeit von 100 cm/s (3,6 km/h) bewegt, welches zu einem weiteren Anstieg des elektrischen Leistungsbedarfs auf maximal 46 W führt. Die höchste Temperatur der Motortreiber von 33 °C wird bei einer vergleichsweise langsamen Bewegung des Roboters mit durchschnittlich unter 30 cm/s erreicht (siehe dritter Testlauf). Im Leerlauf kühlen die Motortreiber deutlich ab, die elektrische Leistung liegt bei etwa 17 W.

[30] engl. Velocity
[31] engl. Power consumption

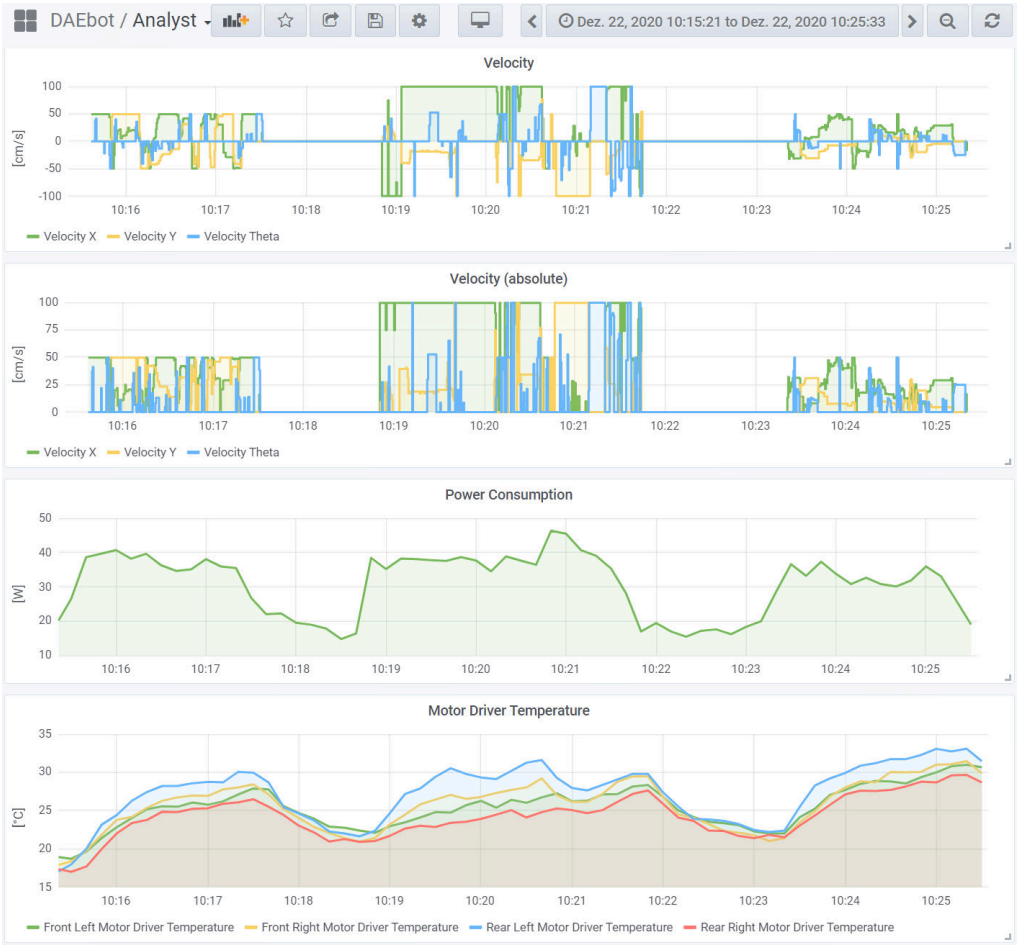


Abbildung 5.25: Visualisierung von Sensorwerten und Aktuatordaten mit Grafana

Während `Grafana` vornehmlich zur manuellen Durchsicht der Systemdaten genutzt wird, wird `Kapacitor` zudem für automatische Analysen genutzt. Hierzu bietet `Kapacitor` diverse Tools (siehe Abbildung 5.23). Testweise werden Benachrichtigungen bei kritischen Werten, wie beispielsweise zu hohe Temperaturen der Motortreiber implementiert. Zudem wird die Detektion von Anomalien testweise anhand der Messwerte

der Ultraschallsensoren implementiert. Hier wird automatisch erkannt, wenn ein Messwert stark von der bisherigen Messreihe abweicht.

Neben den genannten Tools werden eigene Skripte und Tools realisiert, welche in Abbildung 5.24 als `Cognitive Operator Service(s)` bezeichnet werden. Hierbei ist beispielsweise das auf [O66] basierte Skript zum Import der `InfluxDB` Datensätze in MATLAB zu nennen, womit die Daten in MATLAB weiter analysiert werden können.

Die experimentelle Umsetzung des *kognitiven Operators* des DAEbots stellt somit, neben der rudimentären Implementierung einiger Funktionalitäten, die Infrastruktur für zukünftige Methoden zur Optimierung des verteilten Systems zur Verfügung. Hierbei steht auch die Infrastruktur für die gemeinsame Nutzung eines einzelnen *kognitiven Operators* für die Koordination von mehreren mobilen Robotern in Multi-Roboter Kooperationen bereit, wenngleich dieses im Rahmen dieses Projektes nicht evaluiert wird.

5.4 Weitere Demonstratoren

Während der DAEbot exklusiv für die experimentelle Umsetzung der OCM-basierten Systemarchitektur entwickelt wird, werden bereits während der Entwicklung einige Kernkonzepte, u. a. die DAEbot Middleware auf einen weiteren Demonstrator, den mobilen Roboter AMiRo übertragen. Hierbei wird evaluiert, ob und wie das konzeptionelle Modell auf einen bereits entwickelten mobilen Roboter übertragen werden kann.

Die Adaption der Systemarchitektur auf den AMiRo liegt nahe, da der mobile Roboter ebenfalls als verteiltes und modulares System innerhalb eines Roboters realisiert ist (siehe Kapitel 3.2.3, Abbildung 3.4) und dem Autor zur Verfügung steht. Die Adaption der DAEbot Middleware für den AMiRo erfolgt in einer studentischen Arbeit [B9], welche vom Autor konzipiert und betreut wurde.

Die drei Basismodule des AMiRos (*DiWheelDrive*, *PowerManagement* und *LightRing*) dienen als *Controller*-Ebene der OCM-Architektur (siehe Abbildung 5.26). Die *reflektorischen Operatoren* bestehen aus dem Linux-basierten *Cognition* Modul und dem *Image Processing* FPGA Modul als *reflektorischer Operator+*, welcher hier nicht realisiert wird. Als cloudbasierter *kognitiver Operator* wird eine weitere *influxDB*-Datenbank parallel zur Datenbank des DAEbots in derselben virtuellen Maschine eingesetzt, sodass hier keine weiteren größeren Implementierung vonnöten sind. Ebenfalls adaptiert werden, kann die CAN-Schnittstelle bzw. die CAN-Middleware des DAEbots, da diese Schnittstelle auch im

AMiRo verfügbar ist. Umgesetzt wird zudem das Watchdog Konzept des DAEbots. Andere Konzepte, wie die bedarfgesteuerte Komponentennutzung, können nicht realisiert werden, da der AMiRo das Ein- und Ausschalten von einzelnen Modulen nicht unterstützt.

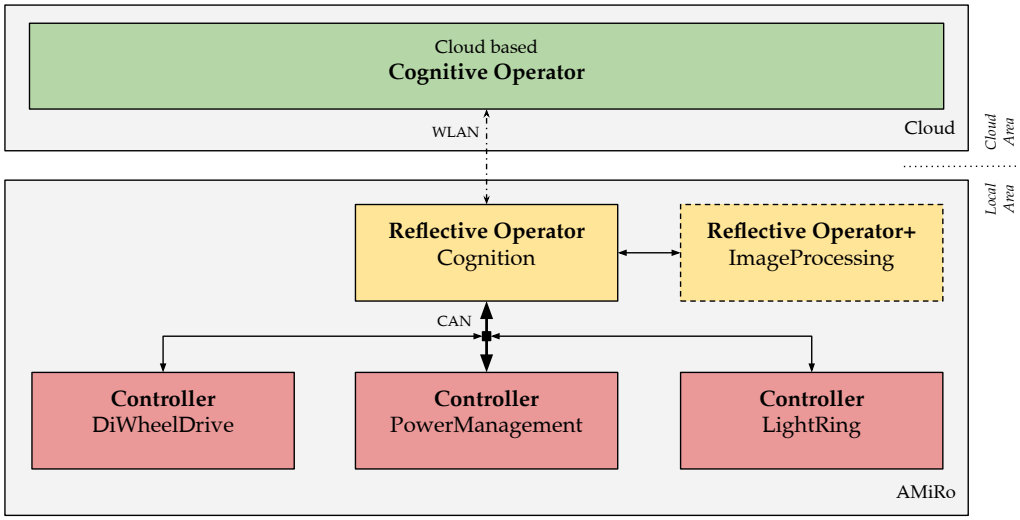


Abbildung 5.26: Umsetzung des Architekturkonzeptes mit dem AMiRo

Des Weiteren ist die Übertragung der hier gezeigten verteilten Systemarchitektur aktuell im Rahmen des Forschungsprojektes S4R in Arbeit. Die vorgeschlagene Systemarchitektur wurde hierfür als geeignet evaluiert[32]. Somit wird das konzeptionelle Modell für mobile Roboter im S4R-Kontext außerhalb der WMR erprobt. Zudem liegt der Fokus zusätzlich auf der in der Praxis bisher nicht evaluierten Multi-Roboter Kooperation. Bei den mobilen Robotern des Forschungsprojektes S4R handelt es sich um mehrere UAVs, welche in einer Multi-Roboter Kooperation als UAS eingesetzt werden sollen. Dieses Multi-UAV System wird für die Überwachung und Kartierung für Notfalleinsätze in Zusammenarbeit mit der Feuerwehr entwickelt [A12]. Der Entwicklungsprozess eines solchen Systems und die zu implementierende OCM-basierte Systemarchitektur wurden zudem in [A13] veröffentlicht.

[32]Die Evaluation wird nachfolgend in Kapitel 7.2 erläutert.

6 Entwicklung eines Tools zur Analyse von verteilten Systemen

Dieses Kapitel beschreibt die Entwicklung eines Tools zur Analyse von verteilten Systemen. Nach der einleitenden Motivation (Kapitel 6.1) und der Beschreibung der Anforderungen an das Analyse-Tool in Kapitel 6.2, folgt der Stand der Technik in Bezug auf die Überwachung und Analyse von technischen Systemen (Kapitel 6.3). Kapitel 6.4 beschreibt das Konzept von pulseAT. Einige Funktionen werden beispielhaft in Kapitel 6.5 erläutert, wenngleich hier keine vollständige Dokumentation der Implementierung erfolgt. Das Kapitel schließt mit der Beschreibung der Integration von pulseAT in die Systemarchitektur für mobile Roboter (Kapitel 6.6) ab. Der Autor hat das Analyse-Tool pulseAT bereits in [A11] veröffentlicht.

6.1 Motivation

Bei der Entwicklung von technischen Systemen ist die Auswahl der Hardware ein entscheidender Faktor, u. a. für die Leistungsfähigkeit des Systems. Dabei stehen meist viele Auswahlmöglichkeiten zur Verfügung. Insbesondere seit der weiten Verbreitung von eingebetteten Systemen, hat sich das Angebot der verfügbaren Hardware, wie z. B. SBC als Recheneinheit, weiter vergrößert. Bei der Auswahl der Hardware müssen zahlreiche Systemanforderungen beachtet und ggf. priorisiert werden. Bei Rechnern für mobile Roboter als verteiltes System sind z. B. die Prozessorarchitektur, die Leistungsfähigkeit der CPU oder die Energieeffizienz wichtig. Die initiale Auswahl der Hardware erfolgt oft aufgrund der Erfahrungen der Entwickler*innen aus vorangegangenen Projekten oder anhand von Referenzbeispielen.

Um die Eignung der Hardware und Software zu evaluieren, ist es während der Entwicklung üblich, die Software mit HiL-Tests auf der ausgewählten Hardware zu prüfen. Hierbei wird validiert, ob die Software sich erwartungsgemäß verhält oder ob beispielsweise die zur Verfügung stehenden Hardware-Ressourcen ausreichen. Die Auslastung bzw. Nutzung der Hardware-Ressourcen kann bei monolithischen Systemen und Standard-PCs mit den üblichen Betriebssystemen, wie Windows, macOS oder Linux vergleichsweise einfach geprüft werden. Hierzu stehen i. d. R. Betriebssystem-Tools zur Verfügung, welche die sogenannte Systemauslastung[1] aufzeichnen und visualisieren. So

[1] engl. System utilization

zeigen beispielsweise bei Windows der `Task Manager` oder bei Linux das Tool `Top` u. a. die CPU-, RAM- oder Netzwerk-Auslastung an.

Die Systemauslastung eines gesamten verteilten Systems kann mit System-in-the-Loop (SYSiL) Methoden [203] anhand der Systemauslastung der einzelnen Rechner geprüft werden, insofern diese Informationen für jeden Rechner des verteilten Systems vorliegen. Da die Kenntnis über die Systemauslastung in verteilten Systemen nicht an zentraler Stelle vorhanden ist, ist die Verifikation der Systemauslastung hier aufwendig und umständlich. Zudem sind, insbesondere für einige eingebettete Echtzeit-Systeme, wie z. B. FreeRTOS oder ChibiOS, keine vergleichbaren Tools wie der `Task Manager` oder `Top` verfügbar. Auch bei betriebssystemlosen Systemen, wie üblicherweise bei Rechnern der Arduino Familie, sind keine Daten zur Systemauslastung verfügbar.

Bei mobilen Robotern sind das Überwachen und Analysieren der Systemauslastung sowohl zum Entwurf, als auch zur Laufzeit besonders wichtig, da es sich bei mobilen Robotern um sicherheitskritische Systeme handelt, welche laut Guiochet, Machin und Waeselynck [104] eine potenzielle Gefahr für die Nutzer, die Umwelt und sich selbst sind[2]. Um den sicheren und zuverlässigen Betrieb eines mobilen Roboters zu gewährleisten, müssen die harten Echtzeitbedingungen, beispielsweise in Bezug auf die Aktuatoren und Sensoren des *Motion Controllers*, unbedingt erfüllt werden. Diese harten Echtzeitbedingungen müssen hierbei in jedem Fall eingehalten werden, auch wenn sich beispielsweise der Use-Case des mobilen Roboters ändert. Diese Use-Cases sind in der mobilen Robotik divers. Eine Roboter-Plattform kann nach der Entwicklung in verschiedenen Use-Cases eingesetzt werden. Hierbei wirken Umwelteinflüsse, im Sinne eines CPS, sowohl auf die Hardware, als auch auf die Ausführung der Software. So muss ein mobiler Roboter in langsamer Bewegung genauso sicher vor einem Hindernis stoppen, wie ein Roboter, der sich sehr schnell bewegt. Zudem ist es möglich, dass Sensoren, Aktuatoren oder Rechner gegen andere Modelle ausgetauscht werden, insofern der mobile Roboter oder das verteilte System modular aufgebaut ist. Dieses hat selbstverständlich auch Einfluss auf die Systemauslastung und die Kommunikation im verteilten System.

[2] vgl. *Safety* in Kapitel 3.2.5

6.2 Anforderungen

Zur Analyse der Daten zur Systemauslastung von verteilten Systemen wird ein Tool benötigt, welches die nachfolgenden Anforderungen erfüllt. Das Tool sollte alle Daten zur Systemauslastung von allen Rechnern im verteilten System einheitlich und an zentraler Stelle bereitstellen. Hierbei ist es notwendig, dass sowohl der aktuelle Zustand des Systems, als auch die Historie der Daten zur Systemauslastung aus diesen Daten extrahiert werden können. Zudem ist es notwendig, dass das Analyse-Tool Informationen zur Analyse der Echtzeitfähigkeit einzelner Rechner bereitstellt. Die Daten zur Systemauslastung werden sowohl für klassische PCs, als auch für eingebettete Systeme benötigt. Das Tool soll dabei sowohl bei der Entwicklung des technischen Systems, als auch während des Betriebs eingesetzt werden.

Das Verifizieren der aktuellen Systemauslastung während der Phase der Entwicklung eines technischen Systems gibt Aufschluss über den aktuellen Zustand des Rechners. Hierbei kommt es nicht selten zur Anpassung der Hardware, beispielsweise zu einem Austausch eines Rechners oder zu einer veränderten Verteilung der Software auf die Rechner innerhalb eines verteilten Systems[3]. Nach Beendigung der Entwicklung eines technischen Systems folgt der Betrieb des Systems. In dieser Phase[4] wird die Systemauslastung oft nicht weiter untersucht, wenngleich diese Systemauslastung nicht zwingend konstant ist. So kann sich z. B. eine ereignisgesteuerte Software je nach Situation (und Ereignis) anders verhalten und ggf. eine höhere Systemauslastung zur Folge haben. Probleme bezüglich der Systemauslastung, wie beispielsweise eine dauerhaft hohe CPU-Auslastung, können zu schwerwiegenden Problemen, wie dem Verzögern von echtzeitfähigen Tasks führen. Die Überwachung und nachfolgende Analyse der Systemauslastung kann weiterhin dazu dienen, das System zu optimieren, indem beispielsweise Rechner mit dauerhaft geringer CPU-Auslastung gegen weniger leistungsstarke aber energieeffizientere Rechner ausgetauscht werden oder die Softwareverteilung innerhalb eines verteilten Systems optimiert wird. Holmström Olsson und Bosch [12] sprechen hierbei von *Post-deployment data usage*, welches laut den Autor*innen auch zur Verbesserung von nachfolgenden Produkten und Systemen verwendet werden kann.

Neben der Verifikation der Hardware-Auswahl anhand der Systemauslastung, sowie der Analyse dieser Informationen im Betrieb des technischen Systems, ist auch die Voraussage von zukünftigen Zuständen der Rechner von Vorteil. Diese Vorhersage von zukünftigen Zuständen wird insbesondere in Bezug auf die Hardware von Maschinen als *Predictive Maintenance*[5] bezeichnet [204]. Hiermit können u. a. frühzeitig Fehler erkannt bzw. vorausgesagt und ggf. umgangen werden. Bei der Fusion von den Informationen der

[3] vgl. Phase I in Kapitel 1.2.5 und Abbildungen 1.7 und 1.8
[4] vgl. Phase II in Kapitel 1.2.5 und Abbildungen 1.7 und 1.8
[5] dt. Prädiktive Instandhaltung

Systemauslastung aller Rechner im verteilten System mit den Messwerten des *Health Controllers*, stünden dem mobilen Roboter sowohl der interne Zustand, als auch die externen Einflüsse, wie z. B. die Umgebungstemperatur zur Verfügung, anhand derer der mobile Roboter optimiert werden könnte.

Nachdem die Recherche nach einem Tool, welches die beschriebenen Anforderungen an die Analyse der Systemauslastung von verteilten Systemen erfüllt, erfolglos geblieben ist, wird ein eigenes Analyse-Tool entwickelt. Dieses Tool, welches als „pulse Analysis Tool (pulseAT)“ bezeichnet wird, soll die Analyse der Systemauslastung der einzelnen Rechner im verteilten System deutlich vereinfachen. Zudem soll pulseAT Möglichkeiten zum Erhalt von Informationen zur Systemauslastung für verschiedene eingebettete Systeme, hier vordergründig die im DAEbot verwendeten Rechner bzw. Betriebssysteme wie FreeRTOS und ChibiOS, realisieren. Zusätzlich zu den üblichen Daten zur Systemauslastung, wie der CPU- oder RAM-Auslastung, sollen weitere relevante Informationen erhoben werden. Bei verteilten Systemen ist u. a. der Zustand der Kommunikation maßgeblich. Bei echtzeitfähigen Systemen ist das Validieren von der Einhaltung der Echtzeit relevant. Sowohl in der Phase der Entwicklung von verteilten Systemen soll pulseAT hilfreich sein, als auch im Betrieb des verteilten Systems, Nutzer*innen Informationen zum Zustand der Systemauslastung bieten. Da diese Nutzer*innen nicht zwangsläufig selbst Entwickler*innen sind, muss der Zustand des verteilten Systems insofern aufgearbeitet werden, dass auch ohne Expertenwissen eine erste Auswertung der Informationen möglich ist, u. a. durch das automatische Versenden von Benachrichtigungen im Fehlerfall.

Das Tool pulseAT entstand im Rahmen dieser Dissertation und dort insbesondere während der Entwicklung des DAEbots. Nichtsdestotrotz soll pulseAT ein eigenständiges Tool für die Analyse von verteilten Systemen bleiben, welches in jedem Architekturmuster verwendbar ist. Ein modularer Aufbau der Software soll zudem die Erweiterung des Analyse-Tools um weitere Daten zum Zustands des Systems, wie z. B. beim DAEbot die Integration der Daten des *Health Controllers* in pulseAT möglich machen.

6.3 Stand der Technik

Die Überwachung oder Analyse des Zustands eines technischen Systems wird unter verschiedenen Begriffen geführt. Einige der üblichen Begriffe, deren Suchmatrix in IEEE Xplore und die dort erzielten Suchtreffer, sind in Tabelle 6.1 dargestellt.

Begriff	**Suchmatrix**	**Suchtreffer**
System Monitoring	(“All Metadata“:system monitoring)	179.689
System Analysis	(“All Metadata“:system (status OR state) analysis)	168.068
System Verification	(“All Metadata“:system verification)	58.540
Condition Monitoring	(“All Metadata“:condition monitoring)	49.693
Fault Detection	(“All Metadata“:fault detection)	42.139
System Validation	(“All Metadata“:system validation)	39.961
Time Verification	(“All Metadata“:verification AND (timing OR time))	20.950
Anomaly Detection	(“All Metadata“:anomaly detection)	12.150
System Health	(“All Metadata“:“system health“)	3.285
Runtime Verification	(“All Metadata“:runtime verification)	1.831
Self-Awareness	(“All Metadata“:self awareness)	1.665
System Status	(“All Metadata“:“system status“)	583
System Diagnosis	(“All Metadata“:“system diagnosis“)	488

Alle Suchbegriffe wurden am 29.12.2020 in IEEE Xplore angewendet

Tabelle 6.1: Begriffe, Suchmatrix und Suchtreffer zur Überwachung oder Analyse

Unter dem Begriff *System Monitoring* werden mit etwa 180.000 Suchtreffern die meisten Treffer zu diesen Suchanfragen in IEEE Xplore erzielt. Das Monitoring, d. h. die Überwachung des Systems ist der erste Schritt zur Analyse des Zustands des Systems. Dieser Ist-Zustand ist auch unter dem Begriff *Condition Monitoring* (ca. 50.000 Suchtreffer) zu finden. Unter dem Begriff „System Status“ (ca. 600 Suchtreffer) sind ebenfalls vereinzelt Ergebnisse in der Literatur zum Status eines Systems zu finden. Die *System Analyse* (ca. 170.000 Suchtreffer) geht einen Schritt weiter als die Anzeige oder Betrachtung des Ist-Zustands und setzt beispielsweise Methoden ein, um Fehler zu erkennen. Diese Fehlererkennung (*Fault Detection*, ca. 42.000 Suchtreffer) ist Stand der Technik für die Analyse des Zustands eines technischen Systems. Die Erkennung von Anomalien (*Anomaly Detection*, ca. 12.000 Suchtreffer) ist ebenfalls eine Art der Fehlererkennung, wenngleich diese zum Ziel hat, vor dem Eintreten eines Fehlers, eine Anomalie zu erkennen und bestenfalls zu beheben. Eine Übersicht über Techniken zur Erkennung von Netzwerk-Anomalien zeigt u. a. [205]. Ehlers, Hoorn, Waller und

Hasselbring zeigen eine Anomalie-Detektion für das Zeitverhalten von Rechnern [206]. Der Begriff *System Diagnose* (488 Suchtreffer) impliziert den aktuellen Zustand des Systems und die Analyse dessen, wenngleich die System-Diagnose nicht zwangsläufig dauerhaft ausgeführt sind, wie es beim *Monitoring*, d. h. der Überwachung typisch ist. Gleiches gilt für den „Gesundheitszustand" des Systems (*System Health*, ca. 3.000 Suchtreffer). Ein Ansatz, den Gesundheitszustand eines autonomen Systems durch ein künstliches Immunsystem zu erhalten und daraus die Fähigkeit der Selbstheilung zu entwickeln, wird im Forschungsprojekt *KI4AS* [O67] untersucht. Die Selbstwahrnehmung (*self-awareness*, ca. 1.700 Suchtreffer) von technischen Systemen beschreiben Lewis, Chandra, Parsons, Robinson, Glette, Bahsoon, Torresen und Yao [207] als die Fähigkeit, den eigenen Zustand zu erkennen. Dieses sei laut den Autor*innen in immer größer, komplexer und autonomer werdenden Systemen notwendig, damit das System selbst ausdrücken und entscheiden kann, wenn z. B. ein Rechner eigenständig erkennt, dass eine Aufgabe nicht erfüllt werden kann. Die Validierung (ca. 40.000 Suchtreffer) und Verifikation (ca. 60.000 Suchtreffer) von technischen Systemen bezieht sich meist nicht auf den internen Zustand des Systems, sondern generell auf das Überprüfen der Erfüllung des Zwecks (Validierung) und der korrekten Umsetzung (Verifikation). Nichtsdestotrotz fallen unter diese Begriffe auch das Prüfen des Systemzustands, wie beispielsweise die Verifikation der Einhaltung der zeitlichen Rahmenbedingungen (*Time Verification*, ca. 21.000 Suchtreffer) oder die Laufzeit-Verifikation (*Runtime Verification*, ca. 1.800 Suchtreffer) welche insbesondere bei Echtzeit-Systemen und Multi-Core Systemen eine große Bedeutung haben [208]. Eine Einführung und Übersicht über die Laufzeit-Verifikation von verteilten oder dezentralen technischen System zeigt [209]. Hierbei erläutern Francalanza, Pérez und Sánchez zudem unterschiedliche Formen von Strukturen zur Verifikation von verteilten Systemen, wie beispielsweise das zentralisierte Monitoring mittels eines einzelnen Moduls zur Überwachung oder dem dezentralen Ansatz durch die Überwachung von mehreren Modulen.

Neben mobilen Robotern sind praktisch alle weiteren Fahrzeuge ebenfalls sicherheitskritische Systeme, wie z. B. Flugzeuge oder Fahrzeuge aus der Automotive Domäne. Bodensohn, Haueis, Mäckel, Pulvermüller und Schreiber zeigen in [210] die Vision eines „Gesundheitsmanagement-Systems für Automobile". Dieses besteht aus vier Phasen, welche einen unendlichen Kreislauf ergeben. Aus der Diagnose erfolgt eine Therapie. Die Therapie wird zur Prävention genutzt. Diese Prävention wird in einer Fall-Historie gespeichert, welche wiederum als Eingang für die Diagnostik dient. Wenngleich sich die gezeigten Beispiele auf die Zustände von Sensoren und Aktuatoren beziehen, zeigt es insgesamt die Notwendigkeit einer Überwachung und die daraus folgenden Rückschlüsse und Aktionen. [211] und [212] zeigen ebenfalls die Notwendigkeit für die Überwachung und Analyse des Zustands im Automotive-Bereich. Hierbei verwenden die Autor*innen die Begriffe „vorausschauende Wartung" (vgl. *Predictive Maintenance* [204]) und Prognose. Pandian und Ali zeigen hierzu in [211] einen Überblick

über Methoden und Algorithmen. Giobergia, Baralis, Camuglia, Cerquitelli, Mellia, Neri, Tricarico und Tuninetti [212] erläutern die Anwendung dessen für den Use-Case eines Sauerstoffsensors.

Adamides, Christou, Katsanos, Xenos und Hadzilacos [81] zeigen in ihrer Taxonomie für die Teleoperation von mobilen Robotern u. a. die Notwendigkeit einer Systemüberwachung (hier als *Robot State Awareness* bezeichnet) in der Robotik. Hierbei nennen die Autoren u. a. die Notwendigkeit von Indikatoren über den Gesundheitszustand/ Status des Roboters, das Ermöglichen, dass (menschliche) Nutzer*innen den Status eines Roboters und die Sichtbarkeit des Systemstatus verstehen [81]. Als Beispiel für die Realisierung der genannten *Robot State Awareness* Faktoren werden in [81] die Quellen [213–216] angegeben. Clarkson und Arkin erläutern in [213], dass die zu visualisierenden Systeminformationen gerade so viele Informationen enthalten sollten, dass menschliche Benutzer*innen erkennen können, ob eingegriffen werden muss ohne dabei durch zu detaillierte Informationen überfordert zu werden. Elara, Acosta Calderon, Zhou, Yue und Hu beschreiben in [214], dass der Systemstatus menschlichen Nutzer*innen vollständig präsentiert werden sollte. Die Autor*innen weisen hierbei auf die Rückmeldung des Roboters in einem angemessenen Zeitrahmen hin. Keyes, Micire, L. und A. zeigen in [215] fünf Roboter-Dimensionen, welche Nutzer*innen präsentiert werden. Eine davon ist der Status des Systems, wobei hierbei nicht die Systemauslastung, sondern Informationen zum Zustand der Hardware, wie z. B. der Batteriezustand, gemeint sind. Labonte, Boissy und Michaud nutzen ihr HMI ebenfalls zur Anzeige des Zustands der Hardware [216], welche bei der Implementierung des DAEbots vom *Health Controller* übernommen wird.

Zahlreiche Tools nutzen Betriebssystemfunktionen von Windows, macOS oder Linux, um CPU-Metriken zu überwachen und bieten hierbei teilweise auch Dienste zur Benachrichtigung an. Zu nennen seien hierbei neben den bereits erwähnten Betriebssystemtools `Task Manager` (Windows) und `Top` (Linux) z. B. die teilweise kommerziellen Tools zur Überwachung von (verteilten) Systemen SolarWinds Engineer's Toolset [O68], Paessler PRTG Network Monitor [O69] oder Nagios XI [O70]. Mit Simple Network Management Protocol (SNMP) [217] steht zudem seit 1988 ein Protokoll zur Verwaltung von Netzwerkgeräten, wie Computern oder Servern zur Verfügung. SNMP ermöglicht neben der Steuerung, die Überwachung der Geräte und ist ein de-facto Standard für Router, Switches und Server. Hierbei besteht ein System aus einem *Manager*, welcher Anfragen nach Systeminformationen, wie die CPU-Auslastung, an die Netzwerkgeräte, welche als *Agenten* bezeichnet werden, stellt. Die *Agenten* beantworten diese Anfrage oder senden eigenständig eine Nachricht an den Manager, falls selbstständig ein Fehler im eigenen System festgestellt wird.

In der Literatur sind zudem weitere Tools zur Analyse des Systemzustands verfügbar. So beschreiben z. B. Emmanouilidis, Jantunen und MacIntyre in [218] eine Software zur Diagnose und Zustandsüberwachung. Der Ansatz unterstützt die Fehlererkennung mittels einer Fehlerbaumanalyse für ein monolithisches System, enthält jedoch keine Funktionen für verteilte Systeme oder Systeme unter Echtzeitbedingungen.

[219] beschäftigt sich mit der Fehlerdiagnose auf Sensordaten unter Verwendung von künstlicher Intelligenz für die Signalverarbeitungstechnik. Wenjun Xi, Yuguang Feng, Yu Zhou und Bo Xu beschreiben verschiedene Phasen der Diagnose, zeigen jedoch nicht die zugrunde liegende Architektur oder Implementierungen.

Rocha und Brandão [220] schlagen eine Multi-Agenten-Architektur für die Überwachung von IoT-Geräten vor. Dieses Modell konzentriert sich darauf, die Gesamtstruktur so zu organisieren, dass eine effiziente und robuste Skalierbarkeit des Systems gewährleistet ist. Das Modell unterteilt die Agenten nicht in Ebenen oder Schichten, welches die Nutzung des Tools in einem System mit einer starken und definierten Hierarchie erschwert.

Das von Rocha und Brandão [220] vorgeschlagene Modell ähnelt dem von Salkenov und Bagchi [221] entworfenen cloudbasierten Überwachungssystem, welches zur Überwachung und Verwaltung heterogener verteilter Systeme unter Verwendung mobiler Agenten verwendet wird. Obwohl das Modell eine Architektur für verteilte Systeme bietet, ist es ebenfalls auf flache Systeme ohne Hierarchie fokussiert und daher nicht direkt auf die OCM-basierte hierarchische Struktur anwendbar.

Das Kieker Framework [222, O71] bietet u. a. eine Antwortzeit-Analyse, eine Trace-Analyse und die Visualisierung von diversen Monitoring- und Analyse-Informationen. Das Eclipse-basierte Framework basiert auf Java, weshalb z. B. die Nutzung mit dem eingesetzten Arduino und den Betriebssystemen FreeRTOS und ChibiOS nicht möglich ist. Zudem zielt Kieker auf monolithische Systeme ab.

6.4 Konzept

Das Konzept des Analyse-Tools pulseAT entstand im Rahmen der Umsetzung des DAEbots. Entsprechend ist die Struktur der Software auf die OCM-basierte Systemarchitektur des konzeptionellen Modells für mobile Roboter abgestimmt. Prinzipiell kann pulseAT jedoch auch in Systemen mit anderen Architektur-Mustern eingesetzt werden. Unter anderem um Informationen über die Einhaltung der Echtzeit zu evaluieren, ist die Integration von pulseAT auch in monolithische Systeme sinnvoll.

Prinzipiell wird pulseAT entwickelt, um die Systemauslastung und damit den Zustand der Rechner in einem verteilten System zu analysieren. Typische Daten zur Systemauslastung sind hierbei die CPU-Auslastung, CPU-Temperatur oder die Anzahl der aktuell ausgeführten Prozesse. Im Vordergrund steht hierbei, dass alle diese im System verteilten Informationen, an einem Ort zusammengeführt, in Zusammenhang gebracht, analysiert und visualisiert werden. Die Software selbst sollte hierbei die Hauptanwendungen der Rechner so wenig wie möglich beeinflussen, weshalb das Softwaretool in verschiedene Module unterteilt ist. Weiterhin bietet pulseAT Schnittstellen zur Erweiterung der Software für eigene Anwendungen und Anforderungen.

Das Analyse-Tool pulseAT besteht aus drei Komponenten (siehe Abbildung 6.1). Jeder zu analysierende oder zu überwachende Rechner verfügt über den *pulseAT Agenten*. Zudem sammelt ein *pulseAT Manager* alle Informationen der im lokalen verteilten System verfügbaren *pulseAT Agenten* ein und wertet diese in einem ersten Schritt aus. Der *pulseAT Manager* ermittelt den Ist-Zustand des verteilten Systems. Des Weiteren sendet der *pulseAT Manager* seine vorliegenden Daten an einen *pulseAT Analyzer*. Dieser wird, analog zur Systemarchitektur für mobile Roboter, in der Cloud realisiert. Dieses hat entsprechend die bereits mehrfach in den Kapiteln 4 und 5 erläuterten Vorteile, wie beispielsweise die Verfügbarkeit der Daten im ausgeschalteten Zustand des lokalen verteilten Systems, die in der Cloud verfügbare Rechenleistung für komplexere Analysen, Methoden und Algorithmen, sowie die Möglichkeit, der gemeinsamen Verwaltung von mehreren lokalen Klienten, sodass ggf. ein Synergieeffekt bzw. eine vergleichbare Analyse baugleicher Roboter der Systemzustände genutzt werden kann. Hierbei wäre es beispielsweise möglich, dass ein mobiler Roboter seine Aufgabe an einen anderen Roboter abgibt, falls festgestellt wird, dass ein Fehler beim Roboter vorliegt oder sich der Zustand verschlechtert. Durch die Ausführung des *pulseAT Analyzers* in der Cloud wird zudem sichergestellt, dass die Aufgaben des lokalen Klienten so wenig wie möglich beeinflusst werden. Es findet eine physische Entkopplung zwischen dem Sammeln der Daten, der Berechnung des Ist-Zustands, möglicher komplexerer Analysen und dem langfristigen Speichern der pulseAT Informationen statt.

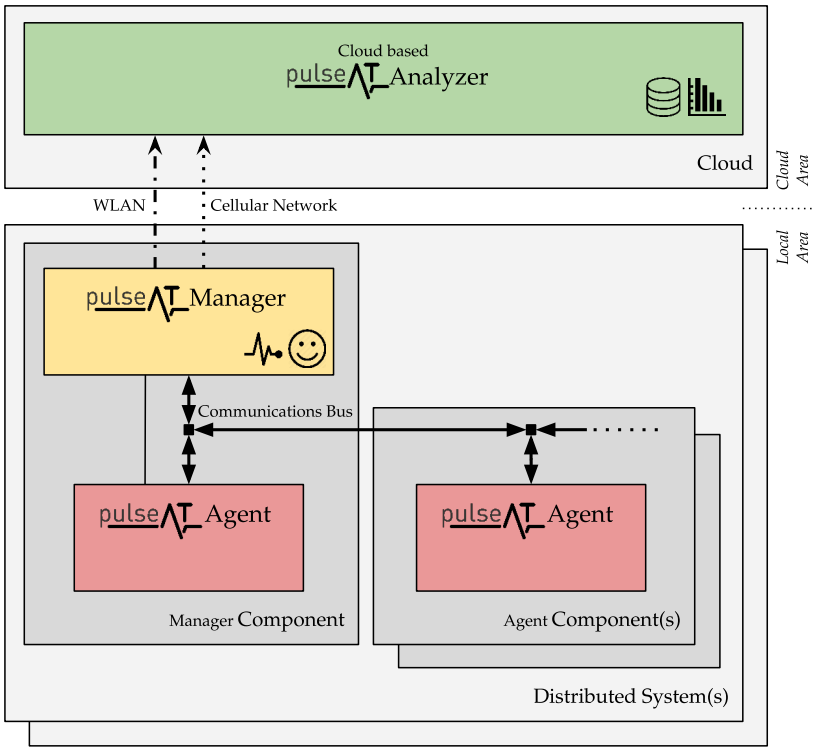


Abbildung 6.1: pulseAT Architektur

Jene Komponente, welche zusätzlich zum *pulseAT Agenten* den *pulseAT Manager* ausführt, wird als Manager-Komponente des lokalen Klienten bezeichnet (siehe Abbildung 6.1). Die Agenten-Komponenten führen je einen *pulseAT Agenten* aus und haben somit einen deutlich reduzierten Leistungsumfang. Diese Struktur wird als agentenbasiertes Modell bezeichnet [223], welche einer service- bzw. dienst-orientierten Architektur entspricht (vgl. SOA in Kapitel 2.2.1 und [18, S. 202]). Handelt es sich um mehrere Agenten, wird das Architekturmodell auch als Multi-Agenten System bezeichnet [56].

Die Agenten sind hierbei eigenständige Softwaremodule, die eine eigene Aufgabe erfüllen. Bei pulseAT besteht die Aufgabe im Sammeln der Daten zur Systemauslastung, sowie in der eigenständigen Benachrichtigung des *pulseAT Managers* bei kritischen Zuständen. Der Manager verwaltet die Agenten und stellt eine Schnittstelle für weitergehende Aufgaben dar. Diese Komponente wird in [56] auch als *Application Environment* bezeichnet. Diese Art

der Modellierung wird ebenfalls in SNMP zur Steuerung und Überwachung von Geräten verwendet. Zudem zeigt z. B. auch [224] die Nutzung von Agenten für das Überwachen von Prozessen und Benachrichtigen bei Auffälligkeiten.

6.4.1 pulseAT Agent(en)

Aufgaben des *pulseAT Agenten* sind das Sammeln und Berechnen der lokal verfügbaren Daten zur Systemauslastung. Hierbei werden aktuell folgende *pulseAT Agenten*-Informationen erfasst, falls diese lokal verfügbar sind:

- pulseAT ID
- CPU-Auslastung (je Kern)
- CPU-Temperatur
- RAM-Auslastung
- Anzahl Prozesse
- Anzahl Threads
- Verpasste (Echtzeit-) Deadline(s)[6]

Die pulseAT ID wird mit jeder neuen Nachricht des *pulseAT Agenten* inkrementiert. Dies hilft dem *pulseAT Manager* bei der Zuordnung der Nachrichten. Zudem verfügt jeder *pulseAT Agent* über einen im System eindeutig vergebenen Identifikator für jeden *pulseAT Agent*, der sich aus der ID des Rechners ergibt (bei CAN beispielsweise die CAN-ID und bei TCP-basierten Schnittstellen die IP). Der pulseAT Identifikator ermöglicht dem *pulseAT Manager* die Identifikation der *pulseAT Agenten*. Die CPU-Auslastung wird möglichst für alle Kerne von Multi-Core Rechnern und die CPU-Temperatur bei vorhandenen internen Temperatursensor(en) übertragen. Die RAM-Auslastung, sowie die aktuell ausgeführten Prozesse und Threads werden ebenfalls ermittelt. Bei Echtzeit-Systemen überprüft ein pulseAT-Watchdog, ob die Echtzeit eingehalten wird und überträgt einen Fehlercode bei Missachtung oder Überfälligkeit eines Prozesses.

Der *pulseAT Agent* wird, wenn möglich, in Threads mit niedriger Priorität ausgeführt, damit die sonstige Software des Rechners so wenig wie möglich beeinflusst wird.

6.4.2 pulseAT Manager

Der *pulseAT Manager* sammelt alle Informationen der *pulseAT Agenten* des lokalen verteilten Systems ein und bringt die Daten in Zusammenhang. Hieraus berechnet der *pulseAT Manager* den Ist-Zustand des Systems und ermöglicht die Visualisierung dieses Zustands. Der *pulseAT Manager* berechnet hierbei aktuell folgende Informationen für jeden *pulseAT Agenten*, zusätzlich zu den eingesammelten Daten dieser Agenten:

[6] dt. Frist

- Betriebszustand
- Antwortzeit
- Anzahl von Timeouts[7]
- Anzahl verpasster (Echtzeit-) Deadlines
- Durchschnittliche CPU-Auslastung
- Gesundheitszustand

Eine Hauptfunktion von pulseAT ist die Berechnung der Antwortzeit für jeden einzelnen *pulseAT Agenten*. Hierzu sendet der *pulseAT Manager* eine Anfrage[8] an jeden *pulseAT Agenten* und berechnet die Zeit bis zur Beantwortung der Anfrage mit den *pulseAT Agent*-Daten. Die pulseAT-Daten werden stets mit niedriger Priorität versendet und reihen sich somit ggf. hinter andere Nachrichten des verteilten Systems ein. Die pulseAT-Antwortzeit enthält entsprechend mögliche Netzwerk-Latenzen. Zudem werden die *pulseAT Agenten* typischerweise nur zyklisch aktiviert, um Hardwareressourcen zu sparen. So ist es beispielsweise möglich, dass die Antwortzeit vergleichsweise lang ist, da die Manager-Anfrage erst nach Beendigung aller anderen Prozesse bearbeitet wird. Bei der Auswertung der Antwortzeit wird somit nicht nur auf die Länge der aktuellen Antwortzeit geachtet, sondern auch durch den *pulseAT Analyzer* geprüft, ob die aktuelle Antwortzeit abnormal ist, d. h. stark von vorangegangenen abweicht. Der *pulseAT Manager* hingegen prüft, ob die Antwortzeit einen Grenzwert, wie z. B. 250 ms nicht überschreitet.

Die Messung der Antwortzeit ist namensgebend für das Analyse-Tool. Man stelle sich vor, der *pulseAT Manager* würde alle Agenten zeitgleich anfragen (siehe Manager Request in Abbildung 6.2) und somit praktisch einen Im**puls** in das System schicken. Die Antworten der *pulseAT Agent* (Agent Responses) treffen zeitlich nach und nach ein (t_1 bis t_4, zur Illustration sind die Antworten der *pulseAT Agenten* unterschiedlich hoch dargestellt). Analog zur Signalverarbeitung kann der Impuls des *pulseAT Managers* auch als Dirac-Impuls $\delta(t)$ bezeichnet werden. Die Antworten der *pulseAT Agent* sind in dieser Analogie die „Impulsantworten“ auf das Eingangssignal (der Manager-Anfrage). Der Begriff „Puls“ soll zudem eine Analogie zu dem in der Biologie umgangssprachlich verwendeten „Puls“ von Lebewesen verdeutlichen. Dieser aus der Elektrokardiographie bekannte Sinusryhtmus („Puls“) ist ein Symbol zum Gesundheitszustand von Lebewesen. Dieses Ziel verfolgt auch das „pulse Analysis Tool (pulseAT)“ in Bezug auf technische Systeme, wie z. B. bei Robotern.

[7] dt. Zeitüberschreitungen
[8] engl. Query

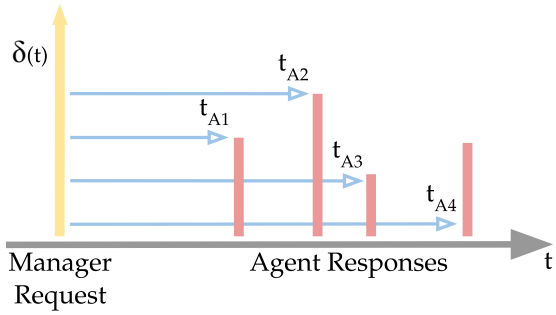


Abbildung 6.2: „Puls“ Analogie zur Impulsantwort durch die pulseAT Antwortzeiten

Antwortet ein *pulseAT Agent* zu spät oder gar nicht auf die Anfrage des *pulseAT Managers*, so wird dieses dem *pulseAT Analyzer* mitgeteilt, indem die Anzahl an Timeouts für jenen Agenten steigt. Der Betriebszustand gibt an, ob der *pulseAT Agent* verfügbar ist oder ggf. der zugehörige Rechner inaktiv ist. Dieser wird auf *inaktiv* gesetzt, falls der *pulseAT Agent* einen Schwellwert von Timeouts überschreitet (wie z. B. nach drei Timeouts beim DAEbot). Zudem zählt der *pulseAT Manager* die Anzahl an verpassten (Echtzeit-) Deadlines und berechnet die durchschnittliche CPU-Auslastung über alle Rechenkerne von Multi-Core Prozessoren. Die Kombination der Daten von allen *pulseAT Agenten* und des *pulseAT Managers* lassen einen Rückschluss über den Ist-Zustand des Gesamtsystems zu, welcher in Form des Gesundheitszustands des Systems dargestellt wird.

Der Gesundheitszustand konzentriert die gewonnenen Daten über alle Agenten in folgende Merkmale:

- CPU-Zustand
- RAM-Zustand
- Netzwerk-Zustand
- Anomalieerkennung
- Gesamtzustand

Der CPU-Zustand beinhaltet die Informationen über die Auslastung und Temperatur. Der Netzwerk-Zustand wird aus den Antwortzeiten und ggf. Timeouts berechnet. Die Anomalieerkennung bezieht sich sowohl auf das Verpassen von (Echtzeit-) Deadlines, als auch auf die Antwortzeiten. Dieser Gesundheitszustand soll es auch nicht-technischen Nutzer*innen möglich machen, den Zustand des Systems zu erkennen. Die Visualisierung des Systemzustands ist beispielsweise laut [213–216] für Mensch-Roboter Interaktionen hilfreich. Hierzu wird dieser Gesundheitszustand möglichst einfach kenntlich gemacht und enthält je nur drei Zustände, analog zur Ampel-Symbolik (siehe Abbildung 6.3), grün (guter Zustand), gelb (kritischer Zustand) und rot (schlechter Zustand), welche beispielsweise durch ein lokales Display angezeigt werden können. Das Beispiel in

Abbildung 6.3 zeigt Auffälligkeiten bei der Anomalieerkennung, welches z. B. durch das Verpassen von (Echtzeit-) Deadlines ausgelöst werden kann. Dadurch ist auch der allgemeine Gesundheitszustand in Rot dargestellt, was Nutzer*innen ein generelles Problem anzeigt. Der kritische Zustand von CPU und RAM deutet auf eine kritische Auslastung der Rechenkapazität des Rechners hin, welche das Verpassen von (Echzeit-) Deadlines bedingen kann.

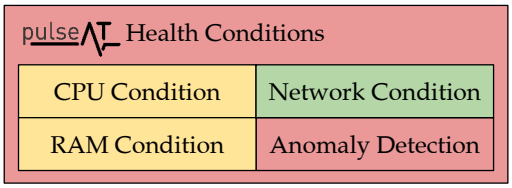


Abbildung 6.3: pulseAT Gesundheitszustand

Um die Hardware-Ressourcen des eingesetzten Rechners so wenig wie möglich zu belasten, speichert der *pulseAT Manager* keine vergangenen Daten und führt keine komplexen Berechnungen durch. Zudem wird der *pulseAT Manager* typischerweise in Prozessen mit geringer Priorität ausgeführt, sodass die sonstigen Berechnungen des Rechners Vorrang haben. Dies kann z. B. bei hoher Auslastung der CPU dazu führen, dass die Threads des *pulseAT Managers* pausieren und somit die Antwort der *pulseAT Agenten* mit Verzögerung erkannt wird. Die Antwortzeiten der *pulseAT Agenten* enthalten somit die Verzögerungen durch den *pulseAT Manager* und bilden damit die gesamte Kommikationsstrecke vom *pulseAT Manager* zum *pulseAT Agent* (inklusive der Berechnung der lokalen Daten zur Systemauslastung) und zurück zum *pulseAT Manager* ab.

6.4.3 pulseAT Analyzer

Der cloudbasierte *pulseAT Analyzer* speichert alle lokal verfügbaren Daten und stellt diese für weitere Analysen zur Verfügung. Im Gegensatz zur Berechnung des Ist-Zustands des *pulseAT Managers* kann der *pulseAT Analyzer* u. a. für Langzeitanalysen genutzt werden. So kann der *pulseAT Analyzer* beispielsweise feststellen, wie hoch die CPU-Auslastung eines Rechners über den gesamten bisherigen Betrieb war. Ist die CPU-Auslastung über einen langen Zeitraum, z. B. mehrere Betriebswochen, gering, so könnte das System optimiert werden. Diese Optimierung kann entweder dazu führen, den Rechner gegen einen Rechner mit weniger leistungsstarker CPU und typischerweise höherer Energieeffizienz zu tauschen oder dem aktuellen Rechner weitere Aufgaben zu übertragen. Die Betrachtung von kurzen Zeiträumen ist insbesondere während der Entwicklung sinnvoll, beispielsweise um zu prüfen, ob die Daten zur Systemauslastung der Erwartung entspricht. Abbildung 6.4 zeigt einige Daten zur Systemauslastung des *reflektorischen Operators* des DAEbots. Die

CPU-Auslastung liegt im dargestellten Zeitraum im Bereich von 25 % bis 60 % und ist somit im unkritischen Bereich mit Potenzial zur Übernahme von weiteren Aufgaben. Die CPU-Temperatur (ca. 42 °C - 46 °C) befindet sich ebenfalls im unkritischen Bereich. Die RAM-Auslastung (ca. 35 %) und Anzahl an Prozessen (170) sind konstant, was darauf schließen lässt, dass die Prozesse ordnungsgemäß ausgeführt werden.

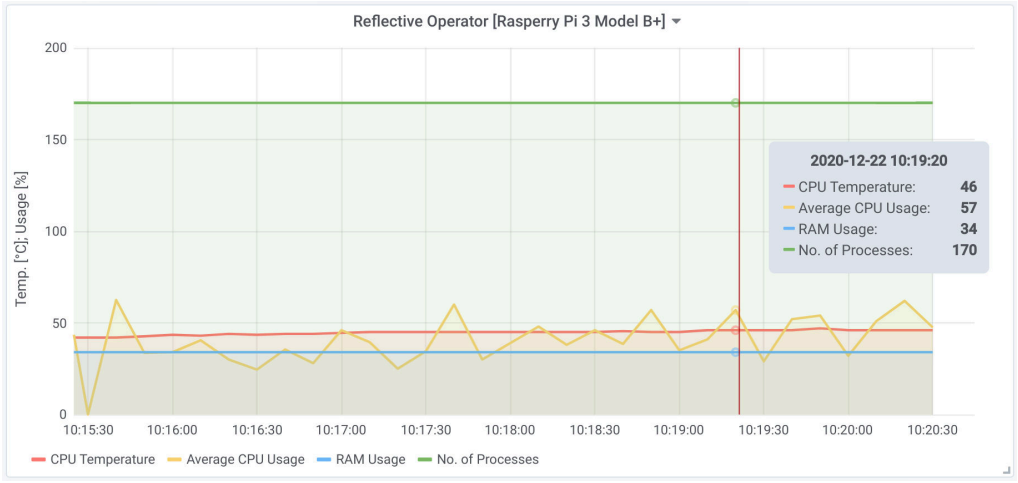


Abbildung 6.4: pulseAT Visualisierung mit Grafana

Analog zum *kognitiven Operator* des konzeptionellen Modells für eine Systemarchitektur für mobile Roboter ist es möglich, mehrere *pulseAT Manager* anzubinden und auszuwerten. Durch den cloudbasierten Ansatz kann die Rechenkapazität in Abhängigkeit der zu analysierenden Systeme und der Komplexität der Methoden und Algorithmen zur Analyse, skaliert werden.

6.4.4 Manager-Anfrage und Ereignis-Benachrichtigung

Die Kommunikation bzw. die Zusammenarbeit zwischen den Komponenten des Analyse-Tools und insbesondere zwischen dem *pulseAT Manager* und den *pulseAT Agenten* basiert auf zwei Methoden, welche aus SNMP übernommen werden.

Manager-Anfrage

Zur Berechnung der Antwortzeit der *pulseAT Agenten* wird ein *Polling* in Form einer *Manager-Anfrage* verwendet. Hierbei fragt der *pulseAT Manager* jeden einzelnen *pulseAT Agenten* unabhängig voneinander an. Der Ablauf ist als vereinfachtes Sequenzdiagramm in Abbildung 6.5 dargestellt.

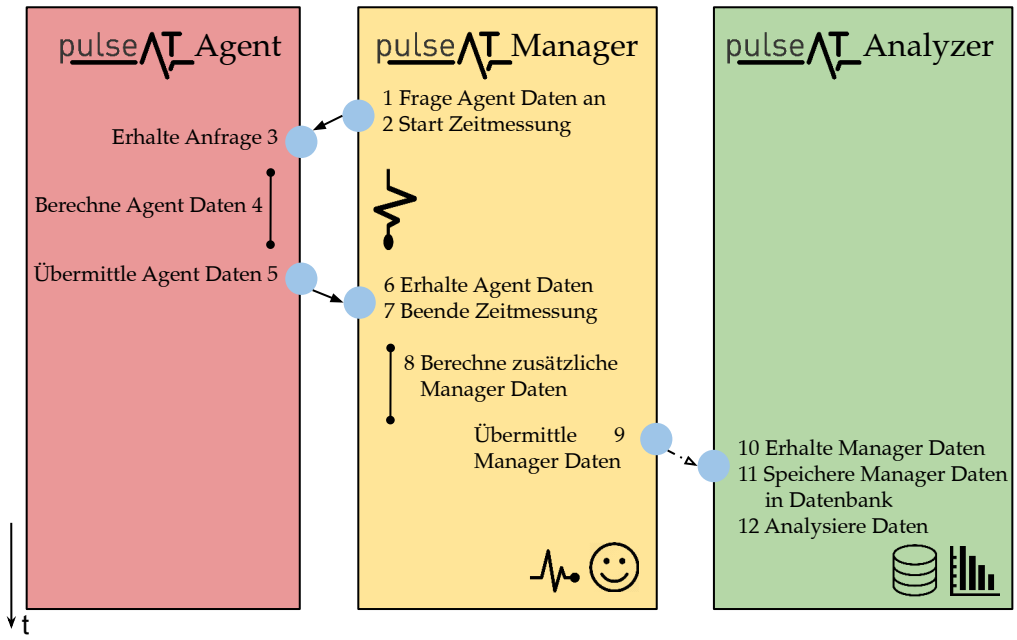


Abbildung 6.5: pulseAT Sequenzdiagramm

Die Sequenz beginnt mit der Anfage an den *pulseAT Agenten* durch den *pulseAT Manager* (1). Unmittelbar nach dieser Anfrage startet der *pulseAT Manager* eine Zeitmessung (2). Sobald der *pulseAT Agent* die Anfrage erhält (3), berechnet dieser die Daten zur Systemauslastung, wie z. B. die CPU- oder RAM-Auslastung (4) und übermittelt diese an den *pulseAT Manager* (5). Sobald die *pulseAT Agenten*-Daten beim *pulseAT Manager* angekommen sind (6),

wird die in (2) gestartete Zeitmessung beendet (7) (symbolisiert durch den „Puls“). Die daraus resultierende gestoppte Zeit wird als Antwortzeit gespeichert. Nachfolgend berechnet der *pulseAT Manager* die Manager-Daten (8). Hierbei wird auch der Gesundheitszustand berechnet und codiert (symbolisiert durch das Gesichts-Piktogramm). Die Manager-Daten werden nachfolgend an den *pulseAT Analyzer* übermittelt (9), wobei diese Übermittlung nicht zeitkritisch ist und ggf. gepuffert werden kann. Der cloudbasierte *pulseAT Analyzer* erhält die Daten des lokalen Klienten (10) und speichert diese in eine (Zeitreihen-) Datenbank (11, symbolisiert durch das „Datenbank“-Symbol). Die Analyse der Daten kann im letzten Schritt erfolgen (12, symbolisiert durch das „Graph“-Symbol), wobei diese nicht synchron mit jedem neuen Datensatz erfolgen muss, sondern auch asynchron und in größeren Zeitabständen erfolgen kann.

Ereignis-Benachrichtigung

Die Ereignis-Benachrichtigung[9] ist die zweite Methode zur Zusammenarbeit zwischen den Komponenten von pulseAT. Eine Ereignis-Benachrichtigung kann jeder *pulseAT Agent* selbstständig an den *pulseAT Manager* versenden, falls kritische Daten zur Systemauslastung gemessen werden. Hierzu überwacht ein Watchdog Grenzwerte, welche für die *pulseAT Agenten*-Daten hinterlegt sind, wie z. B. eine CPU-Auslastung von über 90 %. Diese Grenzwerte können zwischen den *pulseAT Agenten* variieren, typischerweise sind unterschiedliche Prozessor-Typen für unterschiedliche CPU-Temperaturen ausgelegt. Um möglichst wenig Rechenkapazität zu beanspruchen, erfolgt die Überprüfung der Grenzwerte nicht kontinuierlich, sondern in regelmäßigen Abständen (beim DAEbot mit einer Frequenz von 1 Hz). In der Zwischenzeit wird der *pulseAT Agent* pausiert und z. B. durch zeitgesteuerte Interrupts reaktiviert.

6.4.5 Verknüpfung mit dem Health Controller

Verfügt das technische System neben pulseAT über Informationen zum Zustand der Hardware und der Umgebung, in der sich das System befindet, so können Synergien genutzt werden. So können beispielsweise Informationen zum Zustand der Hardware, wie z. B. der Akkumulator-Zustand, mit den Daten zur Systemauslastung in Zusammenhang gebracht werden. Zudem können Umwelteinflüsse, die auch die Rechner beeinflussen könnten, wie z. B. die Luftfeuchtigkeit, mit in die pulseAT-Daten einfließen. Das konzeptionelle Modell für mobile Roboter sieht mit dem *Health Controller* (vgl. Kapitel 4.3.2) eine Komponente vor, welche diese Hardware-Zustände und Umwelteinflüsse erhebt. Diese wird zudem beim DAEbot umgesetzt (vgl. Kapitel 5.3.2). Die Messwerte des *Health Controllers* fließen in die Visualisierung des Gesundheitszustands des pulseAT mit ein und erweitern diese. Somit erhalten Nutzer*innen neben den Daten zur Systemauslastung auf einen Blick einen Eindruck über den Zustand des Gesamtsystems. Werden Anomalien und Fehler

[9] engl. Trap Notification

durch pulseAT festgestellt, beispielsweise ein dauerhaft vollständig ausgelasteter CPU-Kern, welcher nicht mehr reagiert, so kann dieser Rechner mittels der Relais des *Health Controllers* neugestartet werden. Dies führt zum Neustart aller Software Module des Rechners. Insofern der Rechner nachfolgend nicht wieder das gleiche Fehlerbild zeigt, konnte das System zumindest kurzfristig zurück in Betrieb gebracht werden. Laut [225] ist dieser Neustart der Software Module eine Methode zur Selbstheilung.

6.4.6 Analyse während der Implementierung

Die Nutzung von pulseAT während der Entwicklung und Implementierung eines technischen Systems ermöglicht den Entwickler*innen eine einfache, aber trotzdem detaillierte Einsicht in die Daten zur Systemauslastung der eingesetzten Rechner. Diese Daten können beispielsweise bei der Optimierung des Systems während der Entwicklung helfen, welches auch als *Profiling* bezeichnet wird [226]. Besonders in verteilten Systemen erleichtert es die Auswahl der Rechner und die Verteilung der Software auf diese. Funktionen, wie die Erkennung des Verpassens von (Echtzeit-) Deadlines sind besonders bei der Entwicklung von sicherheitskritischen Systemen sinnvoll. Typischerweise wird pulseAT während der Implementierung aktiv benutzt, d. h. die Entwickler*innen überprüfen während oder nach einem Test die Daten zur Systemauslastung.

6.4.7 Analyse während des Betriebs

Nach der Implementierung ist es vorgesehen, dass pulseAT auch während des Betriebs eingesetzt wird. Nicht-technische Nutzer*innen können den Zustand des Systems durch die Systemüberwachung mit einfachen und eindeutigen Visualisierungen prüfen. Ebenso ist der passive Einsatz von pulseAT während des Betriebs vorgesehen. Hierbei werden während der Entwicklung Benachrichtigungen konfiguriert, welche bei kritischen oder abnormalen Situationen, Nutzer*innen oder ggf. auch direkt Entwickler*innen benachrichtigen. Ebenso ist die Übertragung aller Daten zur Systemauslastung (nicht nur im Fehlerfall) an Entwickler*innen möglich, insofern dies mit der Datenschutzbestimmungen vereinbar und gewünscht ist. Dies hilft den Entwickler*innen dauerhaft, ggf. auch durch mehrere Systeme des gleichen Typs, ihr System zu prüfen und dabei Rückschlüsse für Weiterentwicklungen zu ziehen (vgl. *Post-deployment data usage* in [12]).

6.5 Implementierung

Die Implementierung des Analyse-Tools startet parallel zur Implementierung des *reflektorischen* und *kognitiven Operators* des DAEbots. Große Teile der Infrastruktur werden für pulseAT übernommen und deshalb in diesem Kapitel nicht erneut im Detail erläutert, wie z. B. die Cloud Infrastruktur. Das Analyse-Tool ist modular aufgebaut und wird in die vorhandene Architektur integriert. Diese Integration betrifft auch die Kommunikation zwischen *pulseAT Agenten* und dem *pulseAT Manager*.

6.5.1 Kommunikation

Die Kommunikation innerhalb der pulseAT Struktur wird, analog zum OCM, in drei Ebenen und somit zwei Schnittstellen unterschieden. Zur bidirektionalen Kommunikation zwischen *pulseAT Agenten* und *pulseAT Manager* wird nach Möglichkeit die bereits vorhandene Schnittstelle im verteilten System mitgenutzt. Hierbei handelt es sich oft um Bus-Systeme, wie z. B. den CAN-Bus (siehe Abbildung 6.1). Möglich ist jedoch auch eine drahtlose oder kabelgebundene PPP-Verbindung. Die Nutzung des selben Kommunikationskanals, welcher auch zum Transfer von Sensor- und Aktuator-Daten genutzt wird, ist notwendig, um die Latenz im Netzwerk durch die Antwortzeit der *pulseAT Agenten* zu berechnen. Wichtig ist hierbei jedoch, dass das gesamte Datenaufkommen von pulseAT die sonstige Kommunikation so wenig wie möglich beeinflusst. Stellt die Schnittstelle eine Priorisierung zur Verfügung, so sollte stets die niedrigste Priorität für den pulseAT-Datentransfer gewählt werden. Bei der Implementierung von pulseAT für die mobilen Roboter AMiRo und DAEbot wird die angepasste CAN-Nachrichten-ID verwendet,[10] mit der die Priorität stets als „niedrig“ konfiguriert wird. Die einzige Ausnahme bei der Kommunikation zwischen den *pulseAT Agenten* und dem *pulseAT Manager* kann bei der Manager-Komponente vorliegen. Diese führt lokal sowohl den *pulseAT Manager* als auch einen *pulseAT Agenten* aus, welche entweder softwaretechnisch verbunden werden können oder den Loopback[11] der Schnittstelle nutzen. Der Austausch via Software kann hier beispielsweise über Sockets[12] erfolgen.

Die Schnittstelle zwischen *pulseAT Manager(n)* und *pulseAT Analyzer* nutzt ILP im Uplink, ebenfalls analog zur Schnittstelle zwischen dem *reflektorischen* und dem *kognitiven Operator* im OCM-basierten konzeptionellen Modell für mobile Roboter. Die Kommunikation in die Cloud ist unidirektional. Nachrichten vom *pulseAT Analyzer* an den *pulseAT Manager* sind aktuell nicht vorgesehen. Die Nachrichten, bzw. Kommandos im ILP-Format schreiben die

[10] vgl. Kapitel 5.2.1

[11] Als Loopback wird eine Nachricht eines Kommunikationskanals bezeichnet, bei dem der Sender und Empfänger identisch sind.

[12] Sockets sind eine vom Betriebssystem bereitgestellte Software zur bidirektionalen Kommunikation. Sockets können sowohl intern auf einem Rechner, als auch im Netzwerk zum Datenaustausch zwischen Programmen genutzt werden.

Daten des *pulseAT Managers* direkt in die Datenbank des *pulseAT Analyzers*. ILP konforme Kommandos können im lokalen Funknetz via WLAN oder via z. B. NB-IoT (öffentliches Funknetz) versendet werden.

6.5.2 pulseAT Agenten

Die Aufgabe der *pulseAT Agenten* ist das Sammeln der lokal verfügbaren Daten zur Systemauslastung. Diese Funktion soll für diverse Plattformen und Betriebssysteme bereitgestellt werden. Wenngleich die Implementierung der *pulseAT Agenten* für unterschiedliche Rechner und Betriebssysteme differenziert betrachtet werden muss, sind das zugrunde liegende Prinzip und der Ablauf identisch.

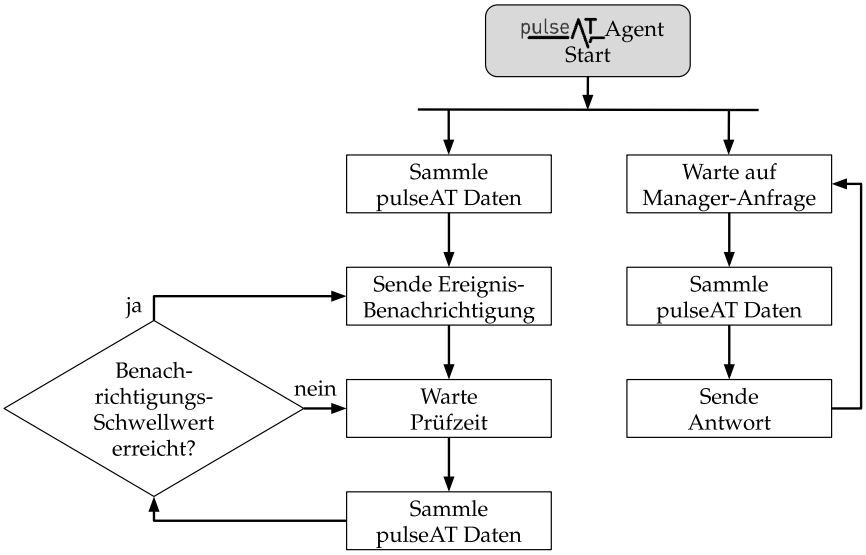


Abbildung 6.6: Ablauf der pulseAT Ereignis-Benachrichtigung und Manager-Anfrage

Jeder *pulseAT Agent* führt zwei unabhängige Tasks aus (siehe Abbildung 6.6). Der rechte Zweig zeigt den Ablauf bei der Manager-Anfrage (*Polling*). Hierbei wartet der *pulseAT Agent* zunächst auf den Eingang der Anfrage des *pulseAT Managers*. Nach Empfang der Anfrage sammelt der *pulseAT Agent* die Daten zur Systemauslastung und sendet diese als Antwort zurück an den *pulseAT Manager*. Nun wartet der *pulseAT Agent* auf die nächste Manager-Anfrage. Parallel dazu läuft ein Thread zur Prüfung von Ereignissen (*Watchdog*). Dieser Thread sammelt nach Programmstart zunächst einmalig die Daten zur Systemauslastung und sendet das Ereignis „Aktivierung“, um dem *pulseAT Manager* zu zeigen, dass der *pulseAT Agent* ab sofort aktiv und erreichbar ist (dieses startet dann die

zyklische Manager-Anfrage). Anschließend wartet der *pulseAT Agent* eine vordefinierte und konfigurierbare `Prüfzeit` (beim DAEbot beträgt die `Prüfzeit` beispielsweise eine Sekunde). In der Zwischenzeit wird der Thread pausiert, damit keine Hardwareressourcen gebunden werden. Nach der `Prüfzeit`, welche z. B. durch einen Timer-Interrupt beendet werden kann, sammelt der *pulseAT Agent* die Daten zur Systemauslastung und prüft alle Schwellwerte, wie beispielsweise die maximal zulässige CPU-Temperatur. Wird einer dieser Benachrichtungs-Schwellwerte überschritten, so sendet der *pulseAT Agent* ein Ereignis zusammen mit den aktuellen Daten zur Systemauslastung und wartet anschließend mit der nächsten Überprüfung auf den Ablauf der `Prüfzeit`. Wird kein Benachrichtungs-Schwellwerte überschritten, so versendet der *pulseAT Agent* keine Nachricht und wartet auf die nächste Überprüfung nach Ablauf der `Prüfzeit`.

Einige Aspekte der softwaretechnischen Umsetzung der *pulseAT Agenten* werden nachfolgend am Beispiel von Agenten für Linux, FreeRTOS, ChibiOS und Arduino erläutert.

Linux

Die Implementierung des *pulseAT Agenten* für das Betriebssystem Linux ist vergleichsweise einfach. Sowohl das Linux-basierte Raspberry Pi OS für die Raspberry Pi Familie, als auch die verbreiteten Linux-Betriebssysteme für PCs wie Ubuntu, Debian oder Mint bringen Betriebssystemfunktionen mit, welche übliche Daten zur Systemauslastung zur Verfügung stellen.

Die aktuelle RAM-Auslastung wird z. B. in `/proc/meminfo` im Dateisystem abgelegt. Das Auslesen dieser Information erfolgt mit einem `FileHandler`, welcher u. a. folgende Messungen ausliest[13]:

- $MemTotal$: Gesamter zur Verfügung stehender Speicher im RAM
- $MemFree$: Freier Speicher im RAM
- $MemBuffers$: Gepufferter Speicher im RAM
- $MemCached$: Zwischengespeicherter Speicher im RAM

Die Berechnung des belegten RAMs (in %) erfolgt mit Gleichung 6.1.

$$RAMusage = \frac{MemTotal - MemFree - MemBuffers - MemCached}{MemTotal} * 100 \tag{6.1}$$

[13] Der gesamte Funktion kann in Anhang D, Quelltextauszug A3 nachgeschlagen werden.

Für die Berechnung der CPU-Auslastung (in %) für jeden Kern i[14] wird die Kernel-Aktivität aus `/proc/stat` ausgelesen. Die Kernel-Aktivität zeichnet den Zeitraum in diversen Modi seit Systemstart für jeden Prozessorkern auf. Hierbei sind folgende Informationen für die Berechnung der CPU-Auslastung relevant:

- $IdleTime$: Kernel-Zeit im Leerlauf
- $IOWait$: Kernel-Zeit durch das Warten auf I/O Ausführungen
- $UserTime$: Kernel-Zeit für ausgeführte Prozesse des Nutzer
- $NiceTime$: Kernel-Zeit für ausgeführte Prozesse mit niedriger Priorität des Nutzer
- $StealTime$: Kernel-Zeit in der die (virtuelle) CPU einer virtuellen Maschine auf eine andere virtuelle CPU wartet
- $SystemTime$: Kernel-Zeit für ausgeführte Prozesse des Betriebssystems
- $IRQTime$: Kernel-Zeit in Bearbeitung von Interrupts
- $SoftIRQTime$: Kernel-Zeit in Bearbeitung von Software-Interrupts

Zunächst wird hieraus die gesamte Kernel-Zeit im Leerlauf ($Idle$) für den Prozessorkern (i) berechnet. Hierzu werden $IdleTime$ und $IOWait$ addiert (siehe Gleichung 6.2).

$$Idle(i) = IdleTime(i) + IOWait(i) \tag{6.2}$$

Für die Berechnung der Kernel-Gesamtzeit ($Total$) in allen Modi für den Prozessorkern (i) wird die Zeit im Leerlauf ($Idle$ aus Gleichung 6.2) mit den Kernel-Zeiten addiert, indem der Prozessorkern Prozesse ausführt (siehe Gleichung 6.3).

$$\begin{aligned} Total(i) = & Idle(i) + UserTime(i) + NiceTime(i) + StealTime(i) \\ & + SystemTime(i) + IRQTime(i) + SoftIRQTime(i) \end{aligned} \tag{6.3}$$

Die CPU-Auslastung (in %) seit Systemstart kann nun durch das Verhältnis von Gesamtzeit ($Total$ aus Gleichung 6.3) und der Zeit im Leerlauf ($Idle$ aus Gleichung 6.2) berechnet werden (siehe Gleichung 6.4).

$$CPUusage(i) = \frac{Total_t(i) - Idle_t(i)}{Total_t(i)} * 100 \tag{6.4}$$

Um die aktuelle CPU-Auslastung zu erhalten, wird ein Intervall aus zwei Zeitpunkten (t und $t-1$) bestimmt, zu denen die Kernel Aktivitäten aus `/proc/stat` herangezogen werden (siehe Gleichung 6.5). Der Zeitpunkt t ist hierbei der aktuelle Zeitpunkt des Empfangs der Manager-Anfrage. Nun erfolgt umgehend die Berechnung der CPU-Auslastung. Als Zeitpunkt $t-1$ werden die Kernel Aktivitäten zum Zeitpunkt der letzten Manager-Anfrage oder der letzten Überprüfung für die Ereignis-Benachrichtigung herangezogen. Somit zeigt die CPU-Auslastung die Auslastung zwischen zwei Manager-Anfragen für den

[14] $i = 1, 2, ..., n$, mit n als Anzahl der Prozessorkerne des Rechners

Prozessorkern (i). Die CPU-Auslastung zwischen zwei Prüfungen (t und $t-1$) für die Ereignis-Benachrichtigung zeigt entsprechend die CPU-Auslastung zwischen zwei Prüfungen (nach Ablauf der `Prüfzeit`).

$$CPUusage_{[t-1;t]}(i) = \frac{[Total_t(i) - Total_{t-1}(i)] - [Idle_t(i) - Idle_{t-1}(i)]}{Total_t(i) - Total_{t-1}(i)} * 100 \qquad (6.5)$$

FreeRTOS und ChibiOS

Die Implementierung des *pulseAT Agenten* für FreeRTOS und ChibiOS, am Beispiel des SMT32F3 Discovery, erfolgte im Rahmen der Weiterentwicklung des *Motion Controllers* des DAEbots in einer studentischen Arbeit [B6], welche vom Autor betreut wurde.

FreeRTOS und ChibiOS nutzen teilweise die gleichen Prinzipien, sodass die grundsätzliche Programmstruktur für beide *pulseAT Agenten* gleich sind. Im Detail ist das Abrufen der Daten zur Systemauslastung jedoch unterschiedlich. So kann beispielsweise die aktuelle RAM-Auslastung in FreeRTOS aus `configTOTAL_HEAP_SIZE` (verfügbarer RAM-Speicher) und `xPortGetFreeHeapSize` (freier Speicher im RAM) berechnet werden. Die Äquivalente dazu in ChibiOS lauten `CH_CFG_MEMCORE_SIZE`, `chCoreGetStatusX` (freier Speicher im RAM) und `configTOTAL_HEAP_SIZE` (verfügbarer RAM-Speicher). Analog wird die CPU-Auslastung aus den verfügbaren Betriebssystem-Werkzeugen ermittelt. Als Beispiel ist die Funktion für die Berechnung der CPU-Auslastung in FreeRTOS in Anhang A3, Quelltextauszug A4 dokumentiert.

Zur Integration der Möglichkeit, verpasste (Echtzeit-) Deadlines zu erkennen, wird die Funktion `pulseAT_check_deadline_missed` in relevanten Funktionen mit Echtzeitbedingungen aufgerufen. Die Funktion (dokumentiert in Anhang A3, Quelltextauszug A5) prüft bei jeder Bearbeitung eines Prozesses mit Echtzeitbedingungen, ob die Ausführung innerhalb eines definierten Zeitfensters abgearbeitet wird. Im Falle der Bewegungssteuerung des DAEbots lässt das maximal zulässige Zeitfenster eine Bearbeitung des Prozesses innerhalb von 20 ms zu (siehe Anhang A3, Quelltextauszug A6). Wird diese 20 ms Deadline übertroffen, so werden die Daten zur Systemauslastung des *pulseAT Agenten* aktualisiert und als Ereignis-Benachrichtigung an den *pulseAT Manager* transferiert.

Während u. a. die Funktion zur Erkennung von verpassten (Echtzeit-) Deadlines direkt in die Anwendungssoftware von FreeRTOS eingebunden wird, erfolgt das Überprüfen auf neue Manager-Anfragen in diskreten Abständen, damit die Hardware-Ressourcen des einzigen Rechenkerns des beim DAEbot verwendeten STM32F3 Discovery möglichst wenig belastet werden. Bei dem *Motion Controller* des DAEbots erfolgt dieses Überprüfen auf neue Manager-Anfragen alle 50 ms. Somit liegt auch die zu erwartende minimale Antwortzeit des *pulseAT Agenten* des *Motion Controllers* bei ungefähr 50 ms.

Arduino

Die Implementierung des *pulseAT Agenten* für Arduino, am Beispiel des Arduino Mega REV3, erfolgte in einer studentischen Arbeit [B7] im Rahmen der Implementierung des *Health Controllers* für den DAEbot, welche vom Autor konzipiert und betreut wurde.

Da die Software für Rechner der Arduino Familie, wie bereits in Kapitel 5.3.2 erwähnt, ohne Betriebssystem auskommt, sind nicht alle Daten zur Systemauslastung durch Protokoll-Dateien oder andere Betriebssystem-Werkzeuge verfügbar. Auch bei der Recherche für solche Informationen für den Mikrocontroller ATmega2560, welcher vom Arduino Mega 2560 verwendet wird, gibt es keine direkte Möglichkeit, die CPU-Auslastung zu erhalten. Die Funktionen zur Berechnung werden im Rahmen der Implementierung des *Health Controllers* für den DAEbot implementiert. Hierzu wird der von [B7] entwickelte Arduino Scheduler verwendet (vgl. Kapitel 5.3.2). Zunächst wird bestimmt, wie viele System-Ticks[15] die CPU maximal in einem Umlauf der Hauptschleife durchlaufen kann. Hierzu wird eine Kalibrierung durchgeführt, indem der ATmega2560 in einem definierten Zeitraum keine Prozesse ausführt, sondern lediglich die Anzahl an System-Ticks zählt. Hierbei ergibt sich beim ATmega2560 eine Anzahl von 276.440 `SystemTicks`. Nun werden während des Betriebs alle Ticks im Leerlauf mitgezählt. Bei dem Abruf der Daten zur Systemauslastung des *pulseAT Agenten* wird die CPU-Auslastung durch das Verhältnis von Ticks im Leerlauf (`IdleTicks`) zu maximal möglichen System-Ticks (`SystemTicks`) berechnet.

Die Berechnung der RAM-Auslastung basiert darauf, dass der freie und nicht genutzte Speicher im RAM zwischen Heap[16] und Stapelspeicher[17] liegt. Da die Positionen der Speicherbereiche Heap und Stapelspeicher mittels `__heap_start` und `__brkval` ermittelt werden können, kann der freie RAM-Speicher zwischen den Bereichen Heap und Stapelspeicher berechnet werden. Die Berechnung der RAM-Auslastung ist in Anhang D, Quelltextauszug A7 dokumentiert.

Der *pulseAT Agent*, welcher auf dem *Health Controller* des DAEbots ausgeführt wird, läuft mit niedriger Priorität. Trotzdem wird auf den Eingang neuer CAN-Nachrichten jede Millisekunde, d. h. mit einer Frequenz von 1 kHz geprüft. Wird eine neue Manager-Anfrage empfangen, so wird diese im Millisekunden-Bereich erkannt. Die zu erwartende minimale Antwortzeit des *pulseAT Agenten* des *Health Controllers* liegt somit ebenfalls im Bereich von Millisekunden.

[15] Ein System-Tick ist eine interne Zeiteinheit eines Rechners, welche je nach CPU und Betriebssystem variieren kann. Der hier verwendete System-Tick des Arduinos wird aufgrund von Timer-Interrupts generiert.

[16] Der Heap ist ein dynamischer Bereich im RAM-Speicher, welcher mittels einer Bibliothek verwaltet wird. Speicherbedarfe können z. B. mit `malloc` angefordert werden.

[17] Der Stapelspeicher (engl. Stack) ist ein Bereich im RAM-Speicher, wo der Prozessor z. B. Register zwischenspeichert.

6.5.3 pulseAT Manager

Der *pulseAT Manager* sammelt die Informationen der *pulseAT Agenten* ein, generiert hieraus den Ist-Zustand des Systems und versendet alle Informationen zur weiteren Verarbeitung an den cloudbasierten *pulseAT Analyzer*. Der *pulseAT Manager* wird für den DAEbot auf dem *reflektorischen Operator* ausgeführt. Als Rechner wird also der SBC Raspberry Pi 3 Model B+ eingesetzt.

Das Sammeln der Informationen der *pulseAT Agenten* erfolgt in je einem Thread für jeden Agenten. Diese werden zunächst initialisiert (siehe Quelltextauszug A8 in Anhang D). Bei der Initialisierung werden die einzelnen Threads gestartet. Die Sammler-Threads arbeiten ein Schema aus sechs Schritten ab (der Quelltext der Sammler-Threads ist in Anhang D, Quelltextauszug A9 dokumentiert). Im ersten Schritt sendet der Manager eine Anfrage an den *pulseAT Agenten*. Anschließend wird der Zeitpunkt der Manager-Anfrage als `start_time` gespeichert. Hierzu wird auf die Betriebssystem-Funktion `gettimeofday` des verwendeten linux-basierten Raspberry Pi OS zurückgegriffen. Nun wartet der *pulseAT Manager* auf die Antwort des Agenten. Wird diese Antwort empfangen, so wird die Zeitmessung gestoppt und die Zeit von der Manager-Anfrage bis zur Antwort (`elapsed_time`) als Antwortzeit gespeichert. Es folgt die Berechnung des Ist-Zustands.

Zur Berechnung des Ist-Zustands werden die Daten zur Systemauslastung des *pulseAT Agenten* bewertet. Diese Bewertung erfolgt für jede Information des *pulseAT Agenten* und führt zu einem der drei Zustände „gut“, „kritisch“ oder „schlecht“. So wird beispielsweise bei der RAM-Auslastung der Zustand als „gut“ bezeichnet, wenn der RAM mit weniger als 80 % belegt ist. Wenn der RAM-Speicher droht vollzulaufen, wird der „kritische“ Zustand zwischen einer Auslastung von 80 % und 90 % erreicht. Ab 91 % RAM-Auslastung sollte die Software des Rechners näher untersucht werden und wechselt den Zustand in „schlecht“. Der Gesamtzustand des *pulseAT Agenten* entspricht dem am schlechtesten bewerteten Zustand eines der Teilzustände (wie z. B. der CPU- oder RAM-Auslastung). Ist beispielsweise nur die CPU-Temperatur im „schlechten“ Zustand, so wechselt der Gesamtzustand der Komponenten ebenfalls auf „schlecht“. So kann auch durch nicht-technische Nutzer*innen lokal am *pulseAT Manager* festgestellt werden, dass etwas nicht in Ordnung ist. Analog wird so auch auf der nächsten Ebene der Gesamtzustand des Systems berechnet. Der *pulseAT Analyzer* übernimmt hier den schlechtesten Gesamtzustand einer Komponente als Zustand für das gesamte System.

Im letzten Schritt werden alle Informationen eines *pulseAT Agenten* in MQTT-Nachrichten umgewandelt. Somit stehen lokal auf der Manager-Komponente (welche den *pulseAT Manager* ausführt) nicht nur die Informationen des *pulseAT Agenten* zur Verfügung, sondern auch die Daten nach der Auswertung des Ist-Zustands durch den *pulseAT Manager*. Analog zur Implementierung der MQTT Schnittstelle für den DAEbot (vgl. Kapitel 5.2.2) werden

diese MQTT-Nachrichten durch den bereits vorhandenen `Mosquitto`-Broker verwaltet, mittels `Telegraf` eingesammelt und im ILP-Format an den *pulseAT Analyzer* transferiert.

Nach der Manager-Anfrage wird zyklisch, mit einer Frequenz von 1 kHz, die vergangene Zeit (`elapsed_time`) geprüft. Erfolgt nach einem definierten Timeout (`pulseAT_agent_timeout_maxtime`) (beim DAEbot drei Sekunden) keine Rückmeldung des *pulseAT Agenten*, so wird eine weitere Manager-Anfrage versendet. Nach mehreren nicht beantworteten Anfragen (`pulseAT_agent_timeout_max_errors`, drei Stück beim DAEbot) wird die Komponente als inaktiv angesehen und nicht weiter angefragt. Der Sammler-Thread prüft alle fünf Sekunden, ob der *pulseAT Agent* wieder aktiv ist. Hierzu versendet ein *pulseAT Agent* beim Start immer eine Ereignis-Benachrichtigung.

Zur Visualisierung des in Abbildung 6.3 dargestellten Gesundheitszustands kann das für den DAEbot implementierte Dashboard erweitert werden (vgl. Abbildung 5.21). Dieses basiert auf `Node-RED` (vgl. Kapitel 5.3.4). `Node-RED` wird dazu genutzt, um die lokal verfügbaren MQTT-Nachrichten des *pulseAT Managers* zu visualisieren und am Display des *reflektorischen Operators* anzuzeigen.

6.5.4 pulseAT Analyzer

Der *pulseAT Analyzer* ist für die Analyse und Speicherung aller Daten zur Systemauslastung eines verteilten Systems zuständig. Bei der Entwicklung des DAEbots wird der *pulseAT Analyzer* in die Infrastruktur des ebenfalls cloudbasierten *kognitiven Operators* integriert (vgl. Kapitel 5.3.6). Analog zum *kognitiven Operator* werden für den *pulseAT Analyzer* ebenfalls der TICK Stack und `Grafana` verwendet. Die Einbindung der Cloud kann auf der PaaS Schicht erfolgen. Den Mittelpunkt bildet die TSDB `InfluxDB` (siehe Abbildung 6.7). Diese empfängt Befehle zum Speichern der pulseAT Daten zur Systemauslastung und des Ist-Zustands vom *pulseAT Manager* im ILP Format. Für jeden Messwert wird der zugehörige Zeitstempel abgelegt, sodass Messungen jederzeit zugeordnet werden können. Eine bidirektionale Verbindung vom *pulseAT Analyzer* zum *pulseAT Manager* ist aktuell nicht vorgesehen.

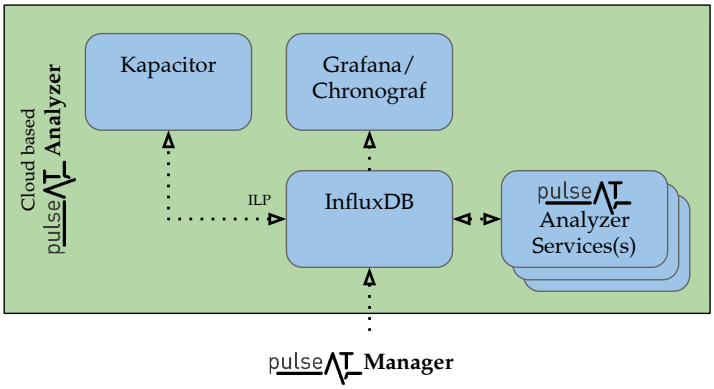


Abbildung 6.7: Dienste des *pulseAT Analyzers*

Die Anbindung des *pulseAT Analyzers* erfolgt im Falle des DAEbots via WLAN. Da das Analyse-Tool jedoch lokal vergleichsweise wenige Daten produziert, die an den *pulseAT Analyzer* gesendet werden müssen, bietet sich auch der Einsatz von IoT typischen Anbindungen, wie NB-IoT, im öffentlichen Funknetz an. Die Datenmenge kann abermals reduziert werden, wenn die Agenten weniger häufig angefragt werden. Insbesondere bei, im Vergleich zu mobilen Robotern, weniger agilen Systemen, kann diese Abfrage beispielsweise auch im Stundentakt oder gar im Abstand von Tagen erfolgen (z. B. bei einem verteilten System aus Geräten, wie u. a. intelligenten Laternen [227], im Bereich *Smart City*).

Die Visualisierung der pulseAT Daten erfolgt mittels `Grafana` (siehe Abbildung 6.4). `Kapacitor` wird zur Analyse der Messdaten verwendet. Hierbei wird u. a. die Anomalie-Detektion verwendet, um Anomalien in der Antwortzeit der *pulseAT Agenten* festzustellen. Während der *pulseAT Manager* nur den Ist-Zustand betrachtet, ist der *pulseAT Analyzer* in der Lage, u. a. mit `Kapacitor` eine Langzeitanalyse durchzuführen. Alle bisherigen Messwerte liegen in der TSDB `InfluxDB` und zeigen die Historie der Daten zur Systemauslastung von allen Rechnern im verteilten System. Hierbei kann ein Ergebnis sein, dass eine Komponente niemals zu mehr als 30 % ausgelastet wird. Während der *pulseAT Manager* dies als einen „guten“ Ist-Zustand betrachtet, kann der *pulseAT Analyzer* feststellen, dass der Rechner dieser Komponente überdimensioniert ist.

Ein Konzept des *pulseAT Analyzers* ist die Modularität der Software. Eigene Skripte und Dienste können in die TICK Infrastruktur integriert werden. Hierbei kann es sich sowohl um Methoden des maschinellen Lernens, als auch um Methoden zur Vorhersage von zukünftigen

Zuständen, beispielsweise mittels Kalman-Filtern, handeln. Die einzige Voraussetzung ist die Integration in die TICK Infrastruktur. Diese Dienste werden als *pulseAT Analyzer Services* bezeichnet (siehe Abbildung 6.7).

6.6 Integration in die vorgeschlagene Systemarchitektur

Wie bereits in den vorangegangenen Kapiteln erwähnt, nutzen sowohl die Implementierung des konzeptionellen Modells für den DAEbot, als auch die Implementierung des Analyse-Tools pulseAT gleiche Tools, Protokolle und Applikationen. Zudem lässt sich die Struktur von pulseAT gut auf die OCM-basierte Systemarchitektur abbilden. Hierbei entstehen viele Synergien, welche den Implementierungsaufwand zur Integration von pulseAT stark mindern. Da die Umsetzung des konzeptionellen Modells für den AMiRo ebenfalls die Middleware des DAEbots einsetzt, sind auch hier die grundsätzliche Struktur und viele der eingesetzten Tools und Protokolle identisch. Im Rahmen einer studentischen Arbeit [B11], welche vom Autor konzipiert und betreut wurde, erfolgte die teilweise Integration von pulseAT in die Software des AMiRo.

Bei den Synergien zwischen DAEbot, AMiRo und pulseAT seien zunächst die Kommunikation und die Schnittstellen zu nennen (siehe Abbildung 6.8). Die Anbindung der *Controller* an die *reflektorischen Operatoren* erfolgt sowohl beim DAEbot, als auch beim AMiRo via CAN-Bus. Dieser wird auch zur Übertragung der Daten zur Systemauslastung des *pulseAT Agenten* an den *pulseAT Manager* verwendet. Hierbei verschickt ein *pulseAT Agent* je zwei CAN-Nachrichten, in denen alle notwendigen Daten zur Systemauslastung enthalten sind. Die Kommunikation innerhalb des *reflektorischen Operators* des DAEbots, u. a. zwischen dem `MATLAB Simulink Stateflow Model` und `Node-RED`, findet via MQTT statt. Die Schnittstelle zwischen *pulseAT Manager* und `Node-RED` nutzt ebenfalls MQTT. Die gesamte Infrastruktur der *reflektorischen Operatoren* rund um MQTT (`CAN2MQTT`, `Telegraf` und `Mosquitto`) wird von pulseAT mitgenutzt und nicht parallel implementiert. Der Uplink vom *pulseAT Manager* zum cloudbasierten *pulseAT Analyzer* erfolgt, ebenfalls analog zum *reflektorischen* und *kognitiven Operator* mittels der ILP Syntax. Die Infrastruktur des cloudbasierten *kognitiven Operators* und des ebenfalls cloudbasierten *pulseAT Analyzers* nutzt jeweils den TICK Stack. Da der TICK Stack bereits für die Speicherung and Analyse der Sensor- und Aktuator-Daten des DAEbots vorhanden ist, ist es nicht notwendig, die entsprechenden Tools, Dienste und Applikationen parallel zu implementieren. Vielmehr werden die Tools des TICK Stacks für beide Anwendungszwecke verwendet. Eine klare Trennung erfolgt durch das Anlegen einer weiteren Datenbank in `InfluxDB` (*DAEbot pulseAT* neben *DAEbot Control*).

Prinzipiell kann jede beliebige Komponente zur Manager-Komponente werden, indem der *pulseAT Manager* auf dieser Komponente bzw. diesem Rechner ausgeführt wird. Bei der Auswahl des Rechners sollte jedoch beachtet werden, dass ausreichend Rechenleistung und die entsprechende Schnittstelle für die Cloud Anbindung vorhanden sind. Bei der Integration von pulseAT in die mobilen Roboter AMiRo und DAEbot liegt es aus den genannten Synergieeffekten nahe, den *pulseAT Manager* demselben Rechner zuzuordnen, welcher im verteilten System auch als *reflektorischer Operator* eingesetzt wird. Alle *Controller*, sowohl des AMiRos als auch des DAEbots, führen einen *pulseAT Agenten* aus. Dies trifft auch auf die *reflektorischen Operatoren* zu, welche neben dem *pulseAT Manager* lokal einen *pulseAT Agenten* zur Berechnung der Daten zur Systemauslastung ausführen.

Abbildung 6.8 zeigt abschließend alle Software-Module des *reflektorischen Operators* und des *kognitiven Operators*, welche im Laufe der Kapitel 5 und 6 vorgestellt wurden. Im Vergleich mit den Software-Modulen des *reflektorischen Operators* (vgl. Abbildung 5.20) und des *kognitiven Operators* (vgl. Abbildung 5.24) ohne pulseAT wird nochmals die starke Integration der pulseAT Module deutlich.

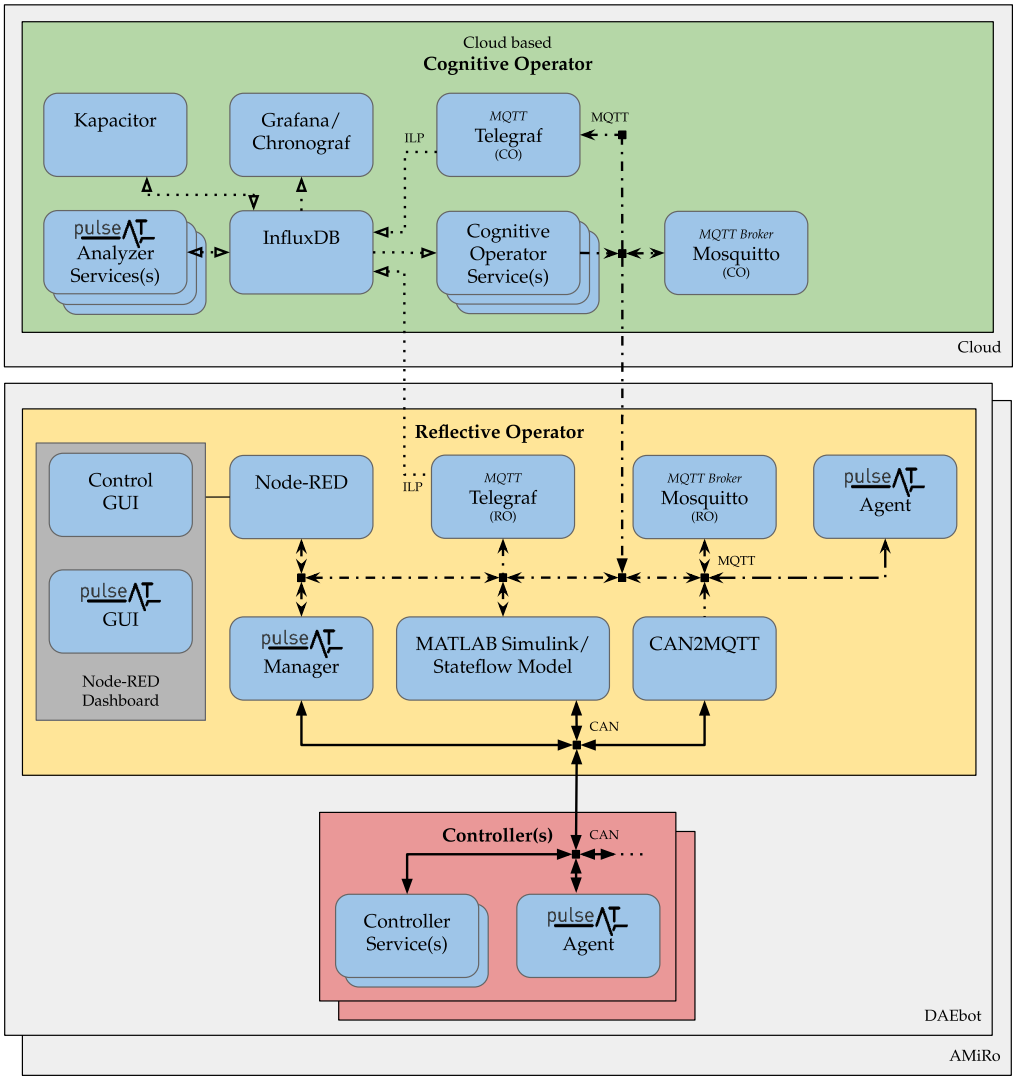


Abbildung 6.8: Integration von pulseAT in die Software des DAEbots

7 Evaluation

Dieses Kapitel beschreibt die Evaluation, der im Rahmen der Dissertation durchgeführten und in den vorherigen Kapiteln erläuterten Arbeiten. Die Auswertung der experimentellen Umsetzung erfolgt in 7.1. Hierbei werden die Experimente erläutert und die Realisierung der Schlüsselmerkmale betrachtet. Die Evaluation der vorgeschlagenen Systemarchitektur wird in Kapitel 7.2 anhand der in Kapitel 3 definierten Anforderungen und Herausforderungen an die Systemarchitektur von mobilen Robotern durchgeführt. Hierbei wird zudem die vorgeschlagene Systemarchitektur in den Stand der Technik eingeordnet. Die Evaluation wird mit einer abschließenden Bewertung der Systemarchitektur und deren Eignung für mobile Roboter beendet (Kapitel 7.3).

Die Evaluation wurde in Teilen bereits in mehreren Veröffentlichungen des Autors publiziert [A3, A8, A9, A11].

7.1 Experimentelle Umsetzung

Die Praxistauglichkeit des in Kapitel 4 beschriebenen, konzeptionellen Modells einer Systemarchitektur für mobile Roboter wurde bereits in Kapitel 5 mit der Entwicklung des DAEbots detailliert evaluiert. Hierbei wurde das konzeptionelle Modell während der Entwicklung des DAEbots verfeinert, erweitert und angepasst.

7.1.1 Entwicklungs-Framework DAEbot

In der zeitaufwendigen experimentellen Umsetzung des DAEbots werden Details des konzeptionellen Modells getestet. So werden z. B. die Definition der Schnittstellen oder die Verteilung der Software auf die Rechner des verteilten Systems evaluiert. Hierbei steht die Praxistauglichkeit des Prototyps im Vordergrund. Der fertig entwickelte DAEbot stellt sicher, dass das konzeptionelle Modell kein theoretisches Konzept bleibt. Durch die Entwicklung eines gänzlich neuen Roboters bestehen u. a. keine Vorgaben in Bezug auf die Auswahl der Komponenten oder die Verteilung der Funktionen auf das verteilte System. Der DAEbot als Demonstrator für verteilte Systeme innerhalb eines Roboters kann hierbei auch für die Evaluation weiterer Architekturkonzepte eingesetzt werden.

Durch die Implementierung von Schlüsselmerkmalen wie der Einbindung in die Cloud, welche u. a. die dauerhafte Verfügbarkeit der Sensor- und Aktuatordaten möglich macht, sowie die Umsetzung einer Toolbox zur modellbasierten Entwicklung mit MATLAB Simulink (Stateflow), kann die Infrastruktur des DAEbots nicht nur als SYSiL, sondern beispielsweise auch zur Simulation von Anwendungen eingesetzt werden (Software-in-the-Loop (SiL)). Diese unterschiedlichen *X-in-the-Loop* Phasen werden laut Garousi, Felderer, Karapıçak und Yılmaz beispielsweise in der Automotive-Domäne analog zum V-Modell eingesetzt [203, S. 21 ff]. Der DAEbot als Entwicklungs-Framework wird auch außerhalb der vorliegenden Arbeit für verschiedene derartige Tests verwendet. Beispielsweise wird die Model-in-the-Loop (MiL)-Methode zum Testen von Nachrichten und Events verwendet. Dieses unterstützt der DAEbot auf Ebene der Simulation des MATLAB Simulink Modells. SiL findet ebenfalls außerhalb der Hardware statt. Processor-in-the-Loop (PiL) Methoden werden zur Überprüfung der Software auf dem Rechner durchgeführt. Typischerweise werden hier die Daten zur Systemauslastung geprüft, was beim DAEbot mit pulseAT erfolgt. Die HiL-Tests erfolgen direkt auf der Hardware. Das Prüfen der Integration des gesamten Systems erfolgt als SYSiL-Tests. Hierzu bietet sich der DAEbot besonders an, da dieser u. a. durch die offene Bauweise und großzügige Bauform auch den Austausch von Komponenten erleichtert.

Während der Entwicklung des DAEbots im Rahmen dieser Dissertation wurden zahlreiche studentische Projekt- und Abschlussarbeiten (u. a. [B1–B8, B10]) durchgeführt. Die Student*innen nutzten den DAEbot hierbei für Experimente als Entwicklungs-Framework.

7.1.2 Analyse mit pulseAT

Das in Kapitel 6 vorgestellte Analyse-Tool pulseAT wurde zur Analyse von Daten zur Systemauslastung in verteilten Systemen entwickelt. Die Evaluation der Systemarchitektur wurde durch pulseAT stark vereinfacht, insbesondere durch die dauerhafte Verfügbarkeit der Daten zur Systemauslastung und die Visualisierung mit `Grafana`. So können beispielsweise im Nachgang eines Feldversuchs mit dem DAEbot die einzelnen Daten zur Systemauslastung aller Komponenten an zentraler Stelle betrachtet werden. Da beim DAEbot allein auf der lokalen Ebene fünf Komponenten mit 12 CPU-Kernen zum Einsatz kommen, ist die Analyse während des Einsatzes für einzelne Entwickler*innen oder kleine Entwickler-Teams praktisch unmöglich. So wird durch pulseAT z. B. der in Abbildung 7.1 gezeigte Fehler des *Perception Controllers* gefunden. Während des Feldversuchs liefert die Applikation zur Erkennung von AprilTags nach kurzer Zeit keine Messwerte an den *reflektorischen Operator* zurück. Durch die Analyse der CPU-Auslastung des *Perception Controllers* zeigt sich, dass mindestens zwei der vier CPU-Kerne für ein paar Minuten nahezu ausgelastet sind und dann plötzlich in den Leerlauf wechseln. Erst durch die Betrachtung der RAM-Ausnutzung

im gleichen Zeitraum wird klar, dass die RAM-Auslastung stetig steigt, bis der RAM nahezu komplett gefüllt ist. Anschließend wird die Applikation gestoppt, d. h. es liegt offenbar ein Fehler in Bezug auf die Speicherverwaltung vor.

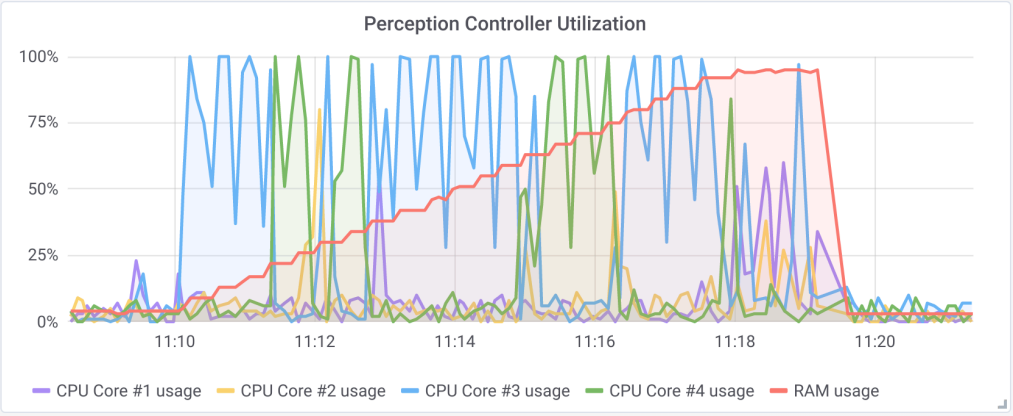


Abbildung 7.1: Fehlersuche mit pulseAT

Das Analyse-Tool pulseAT kommt nachfolgend in dieser Evaluation mehrfach zur Anwendung. Zudem wurden bereits zuvor einige mittels pulseAT erzielte Ergebnisse dargestellt (vgl. Abbildungen 5.25, 6.4, 7.2, 7.3, 7.4, 7.5 und A8, sowie Tabellen 7.1 und 7.2).

7.1.3 Definition der Systemkomponenten

Die in Kapitel 4.3 erläuterte Definition der Systemkomponenten und die damit verbundene Verteilung der Funktionen auf die einzelnen Komponenten, haben sich bei der experimentellen Umsetzung als praxistauglich erwiesen. Hierbei ist besonders die Definition der *Controller* zu nennen, welche gut auf die Vorteile des verteilten Systems abgestimmt ist. So ist der für mobile Roboter zwangsläufig notwendige *Motion Controller* typischerweise immer aktiv, um die Bewegung des Roboters zu steuern. Der *Perception Controller* hingegen fügt dem System typische Funktionen, wie die Bildverarbeitung, hinzu. Um Energie zu sparen, kann dieser je nach Anwendungsfall, jedoch auch ausgeschaltet werden (vgl. bedarfsgesteuerte Komponentennutzung in Kapitel 4.2.5). Für die sichere Bewegung des mobilen Roboters haben beim Demonstrator der *Motion Controller* und die dort angebundenen Abstandssensoren ausgereicht. Sobald ein Ziel wie z. B. ein AprilTag erreicht werden soll, wird der *Perception Controller* hinzugeschaltet. Die Informationen des *Health Controllers* haben sich bei der experimentellen Umsetzung ebenfalls als sinnvoll herausgestellt. Insbesondere die Informationen zum Zustand der Akkumulatoren und zum

aktuellen Leistungsbedarf sind hierbei zu nennen. Das am *Health Controller* angeschlossene Relais-Board ermöglicht Funktionen wie die bedarfsgesteuerte Komponentennutzung. Die übergeordnete Steuerung und Konfiguration der *Controller* durch den *reflektorischen Operator* hat sich beim Betrieb des DAEbots bewährt. Insbesondere aus Sicht der Entwickler*innen ist die modellbasierte Entwicklung der Steuerung in einem zentralen (MATLAB Simulink) Modell von Vorteil. Der *reflektorische Operator+* ist zwar theoretisch vorteilhaft, konnte mit dem Demonstrator jedoch nicht fertig entwickelt werden. Die Erweiterung des verteilten Systems um einen Cloud-basierten *kognitiven Operator* ist insbesondere durch die dauerhafte Verfügbarkeit der lokalen Daten in einer leistungsfähigen Cloud Umgebung von Vorteil.

7.1.4 Auswahl der Rechner

Neben der Definition der Komponenten, ist die Auswahl der Rechner des verteilten Systems ausschlaggebend für die Leistungsfähigkeit eines mobilen Roboters wie dem DAEbot. Hierbei sieht die vorgeschlagene Systemarchitektur den Einsatz von SoC-Systemen, wie SBC vor[1]. Dieses soll die Energieeffizienz des verteilten Systems erhöhen. Während die Energieeffizienz in Kapitel 7.1.7 detailliert betrachtet wird, bezieht sich dieser Abschnitt auf die Leitungsfähigkeit der ausgewählten Rechner und die Eignung dieser Auswahl für die OCM-basierte Systemarchitektur für mobile Roboter. Als Grundlage für die Evaluation dienen, neben der analytischen Evaluation des Autors, die pulseAT Daten zur Systemauslastung. Hier werden exemplarisch die CPU-Auslastung und die pulseAT Antwortzeit als Indiz zur Auslastung der einzelnen lokalen Rechner betrachtet. Auf die Informationen zur RAM-Auslastung, CPU-Temperatur und Anzahl an aktiven Prozessen (wie beispielsweise in Kapitel 6.4.3, Abbildung 6.4 dargestellt), wird nicht näher eingegangen. Nicht evaluiert wird hierbei der *reflektorische Operator+*, da dieser, wie in Kapitel 5.3.5 beschrieben, nicht abschließend getestet wird.

CPU-Auslastung

Abbildung 7.2 zeigt die durchschnittliche CPU-Auslastung des *Motion*, *Health* und *Perception Controllers*, sowie des *reflektorischen Operators*, jeweils über alle Rechenkerne. Hierbei wird schon bei der ersten Betrachtung deutlich, dass die CPU des *Motion Controllers* (Rot) nahezu unbelastet ist. Der *Health Controller* (Blau) hingegen, ist nahe am Maximum. Bei den beiden Mehrkern-Rechnern des *Perception Controllers* (Gelb) und *reflektorischen Operators* (Grün) variiert die CPU-Auslastung deutlich.

[1] vgl. Kapitel 4.2.1

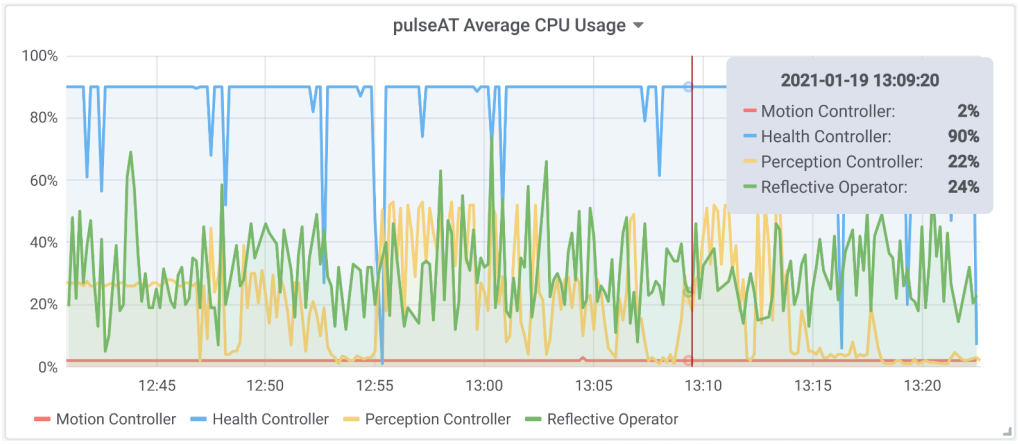


Abbildung 7.2: Evaluation der CPU-Auslastung des DAEbots mit pulseAT

Zur weiteren Analyse der CPU-Auslastung im dargestellten Zeitraum (siehe Abbildung 7.2) werden die Messdaten statistisch ausgewertet. Die Ergebnisse dieser Auswertung sind als Box-Plot-Diagramme[2] in Abbildung 7.3 und in tabellarischer Form in Tabelle 7.1 dokumentiert. Als Vergleichswerte sind hier ebenfalls die Daten zur CPU-Auslastung des AMiRos vermerkt. Für jede dargestellte Komponente stehen mindestens 250 Messwerte (N) zur Verfügung. Die ersten beiden Spalten zur CPU-Auslastung zeigen den Durchschnitt $\varnothing$ und den Median Med. Das untere Quartil $Q1$ und das obere Quartil $Q3$ helfen bei der Einordnung aller Messwerte im Datensatz in Bezug auf die Streuung der Daten. Der jeweilige kleinste Messwert Min und größte Messwert Max zeigen die Extrema des Datensatzes. Die Standardabweichung[3] σ gibt ebenfalls Aufschluss über die Streuung der Messwerte.

Der Box-Plot des *Motion Controllers* in Abbildung 7.3 verdeutlicht, dass die CPU-Auslastung des *Motion Controllers* nur im Bereich von 2 % (siehe Min Tabelle 7.1) bis 3 % (Max) liegt. Die Standardabweichung ist mit unter 0,1 % (σ) ebenfalls sehr gering. Der *Health Controller*

[2] Box-Plots (dt. Kastengrafiken) werden zur übersichtlichen Darstellung der Verteilung von Messwerten in einem Datensatz verwendet. Hier zeigt die sogenannte „Box“ im Diagramm den Bereich, in dem 50 % der Werte des Datensatzes liegen. Das untere Quartil $Q1$ liegt bei 25 % eines nach der Größe sortierten Datensatzes und stellt im Box-Plot das untere Ende der Box dar. Das obere Quartil $Q3$ zeigt den oberen Rand der Box und liegt bei 75 % eines nach der Größe sortierten Datensatzes. Die Markierung in der Mitte der Box zeigt den Median Med, welcher der Datenwert in der Mitte eines nach der Größe sortierten Datensatzes ist. Weitere Visualisierungen der Extrema zeigen zudem das gesamte Spektrum des Datensatzes. Hierbei ist die Markierung links außen der kleinste Wert Min im Datensatz und wird auch als unterer Whisker bezeichnet. Die Markierung rechts außen zeigt den größten Wert Max im Datensatz und wird auch als oberer Whisker bezeichnet.

[3] Die Standardabweichung σ ist ein Maß der Streuung von Datenwerten in einem Datensatz. Je größer die Standardabweichung σ, desto größer ist die Streuung der Datenwerte.

hingegen hat seinen Median in Bezug auf die CPU-Auslastung bei 90 % (Med). Auffällig ist hier jedoch die breite Streuung im Bereich von 1 % (Min) und 90 % (Max), welches auch die Standardabweichung von ca. 13 % (σ) zeigt. Die durchschnittliche CPU-Auslastung über alle Kerne des Multi-Core *Perception Controllers* liegt hauptsächlich im Bereich von 4 % ($Q1$) bis 29 % ($Q3$). Analog liegen die Messwerte des *reflektorischen Operators* hauptsächlich im Bereich von 21 % ($Q1$) bis 37 % ($Q3$).

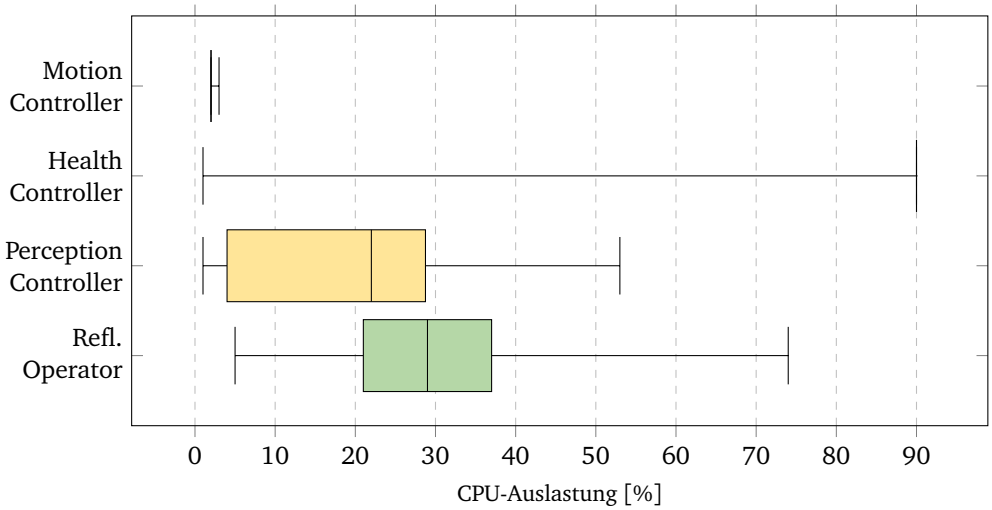


Abbildung 7.3: Evaluation der CPU-Auslastungen am Beispiel des DAEbots

Roboter	**Komponente**	**CPU-Auslastung**						
		$\varnothing$ [%]	Med [%]	$Q1$ [%]	$Q3$ [%]	Min [%]	Max [%]	σ [%]
DAEbot	*Motion Controller*	2	2	2	2	2	3	0,064
	Health Controller	87	90	90	90	1	90	12,901
	Perception Controller	20,8	22	4	28,8	1	53	15,99
	Reflektorischer Operator	30,5	29	21	37	5	74	12,43
AMiRo	*Di Wheel Drive*	1,3	1	1	1	1	29	2,149
	Power Management	1,2	1	1	1	1	9	0,901
	Light Ring	1,9	1	1	2	1	14	2,099

Tabelle 7.1: Evaluation der CPU-Auslastungen mit pulseAT

Antwortzeit

Die pulseAT Antwortzeit wird zwischen dem Versenden der *Manager-Anfrage* und dem Empfang der Antwort des *pulseAT Agenten* gemessen[4]. Die Antwortzeit enthält den (zweifachen) Weg über den Kommunikationskanal, sowie die Berechnung der internen Daten zur Systemauslastung der angefragten Komponente durch den *pulseAT Agenten*. Da diese *pulseAT Agenten* als Tasks mit niedriger Priorität ausgeführt werden, sind auch Verzögerungen durch die Hauptaufgaben der Rechner enthalten, sodass die Antwortzeit auch Hinweise auf eine zu hohe Auslastung der Komponente liefern kann.

Abbildung 7.4 zeigt die pulseAT Antwortzeit des *Motion*, *Health* und *Perception Controllers*, sowie des *reflektorischen Operators*. Hierbei wird schon bei der ersten Betrachtung deutlich, dass die Antwortzeiten des *Motion Controllers* (Rot) und des *Health Controllers* (Blau) nahezu konstant sind. Die pulseAT Antwortzeit des *Perception Controllers* (Gelb) hingegen variiert leicht und weist zudem starke Ausreißer auf. Diese Ausreißer liegen teilweise bei einer Sekunde und sind aus Gründen der Darstellung bei 800 ms abgeschnitten. Die Antwortzeit des *reflektorischen Operators* (Grün) ist ebenfalls weniger konstant als die des *Motion* und *Health Controllers*, streut jedoch nicht aus einem vergleichsweise kleinen Bereich heraus. Generell liegen die Antwortzeiten der Komponenten des DAEbots weit auseinander. Während der *Health Controller* in unter 5 ms antwortet, liegt die Antwortzeit des *reflektorischen Operators* bei etwa 330 ms.

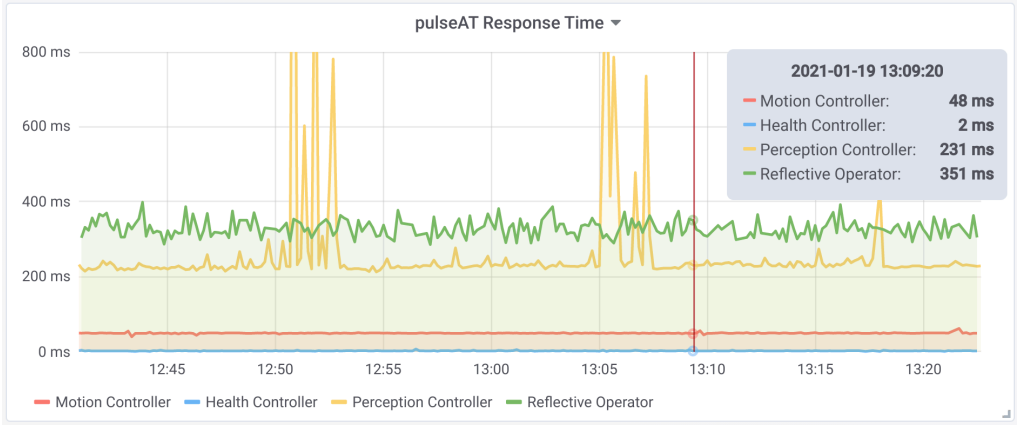


Abbildung 7.4: Evaluation der Antwortzeiten des DAEbots mit pulseAT

[4] vgl. Abbildung 6.5 in Kapitel 6.4.4

Im Detail zeigen auch die in Abbildung 7.5 exemplarisch dargestellten Box-Plots des *Motion Controllers* und des *reflektorischen Operators*[5], sowie die Berechnung der statistischen Werte in Tabelle 7.2 unterschiedliche Ergebnisse bei der Analyse der pulseAT Antwortzeit in Bezug auf die Streuung und den Median der Messwerte. Während die Antwortzeit des *Motion Controllers* im Bereich von 40 ms (siehe Min in Tabelle 7.2) und 56 ms (Max) liegt, liegen diese Werte für den *reflektorischen Operator* bei 286 ms (Min) und 399 ms (Max). Die Box-Plots verdeutlichen zudem, dass 50 % der Werte des *Motion Controllers* innerhalb von einer Millisekunde liegen (zwischen 49 ms ($Q1$) und 50 ms ($Q3$)), während die Box des *reflektorischen Operators* sich über 32 ms erstreckt (zwischen 343 ms ($Q1$) und 311 ms ($Q3$)). Durch die sehr großen Ausreißer der pulseAT Antwortzeiten des *Perception Controllers* ist auch die Standardabweichung mit 104 ms (σ) sehr groß. Die Standardabweichung der Antwortzeit des *Health Controllers* hingegen liegt bei unter einer Millisekunde.

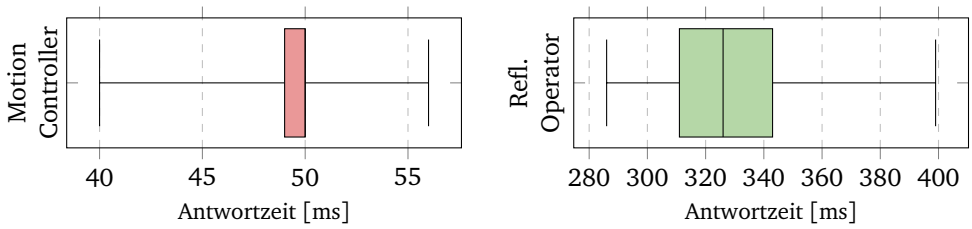


Abbildung 7.5: Evaluation der pulseAT Antwortzeiten am Beispiel des *Motion Controllers* und *reflektorischen Operators*

Im Vergleich mit den ebenfalls evaluierten pulseAT Antwortzeiten der AMiRo-Controller (siehe Tabelle 7.2) sind die Antwortzeiten des DAEbots in drei von vier Fällen deutlich länger. Dies ist auf die Implementierung zurückzuführen. Während die *Controller* des AMiRos häufiger auf den Erhalt neuer CAN-Nachrichten prüfen, variiert diese Prüfzeit bei den Komponenten des DAEbots. So prüft der *Motion Controller* alle 50 ms, ob eine neue *Manager-Anfrage* des *pulseAT Managers* vorliegt. Der *Health Controller* hingegen führt diese Prüfung viel häufiger durch. Ausschlaggebend für die Beurteilung der pulseAT Antwortzeit ist somit die Streuung der Messwerte. Diese ist insbesondere beim *Motion* und *Health Controller* im etwa gleichen Bereich wie die Streuung der Antwortzeiten der AMiRo-Controller. Die Streuung des *reflektorischen Operators* ist insofern weniger ausschlaggebend, da hier keine harte Echtzeit notwendig ist. Einzig die Streuung der Antwortzeiten des *Perception Controllers* ist, mit einer Standardabweichung von über 100 ms und Ausreißern mit bis zu einer Sekunde, auffällig.

[5] Die Box-Plots der anderen Komponenten sind in Anhang E, Abbildung A8 dargestellt.

Roboter	Komponente	Antwortzeit						
		$\varnothing$ [ms]	Med [ms]	$Q1$ [ms]	$Q3$ [ms]	Min [ms]	Max [ms]	σ [ms]
DAEbot	*Motion Controller*	49,5	50	49	50	40	56	1,255
	Health Controller	2,4	2	2	3	1	7	0,709
	Perception Controller	252	229	225	235	213	1000	104
	*Reflektorischer Operator**	328,6	326	311,5	343	286	399	23,11
AMiRo	*Di Wheel Drive*	4,3	4	4	5	3	11	1,051
	Power Management	4,3	4	4	5	3	11	0,997
	Light Ring	4,2	4	3	5	3	15	1,267

*Hardware CAN-Loopback

Tabelle 7.2: Evaluation der Antwortzeiten mit pulseAT

Auswertung der Analysen

Der ***Motion Controller*** nutzt den ARMv7 Cortex-M4 [O28] Einkernprozessor der verwendeten STM32F3 Discovery CPU mit durchschnittlich 2 % nur sehr wenig aus. Im Vergleich mit der CPU-Auslastung der ebenfalls ARM-basierten Einkernprozessoren des AMiRos (ARM Cortex-M3 für das *Di Wheel Drive*- und *Light Ring*-Modul, sowie der ARM Cortex-M4F für das *Power Management*)-Modul [94]) zeigt sich jedoch, dass auch die Auslastung der AMiRo-Controller bei durchschnittlich unter 2 % liegt. Zudem ist die Antwortzeit sehr konstant, ohne das Ausreißer aufgezeichnet werden. Der *Motion Controller* prüft alle 50 ms, ob eine neue *Manager-Anfrage* des *pulseAT Managers* vorliegt. Somit ist die Antwortzeit mit dem Median von 50 ms plausibel. Beim Demonstrator sind zudem keine Auffälligkeiten bezüglich der Eignung des STM323F3 Discovery als *Motion Controller* aufgetreten. Die Nutzung einer niedriger getakteten CPU für den *Motion Controller* ist möglich. Eine weitere Alternative ist die Optimierung der Software auf schnellere, bzw. häufigere Beantwortung von CAN-Nachrichten. Hiervon wurde jedoch abgesehen, da die erzielten Antwortzeiten bei der experimentellen Umsetzung zufriedenstellend sind.

Die Implementierung des ***Health Controllers*** hingegen ist so optimiert, dass die maximale Leistung aus dem vergleichsweise langsam getakteten Einkernprozessor des Arduino Mega 2560 abgerufen wird. Die CPU-Auslastung ist mit durchschnittlich 88 % nahe am Grenzbereich zur Überlastung. Nichtsdestotrotz ist die Antwortzeit extrem gering und stabil. Der Arduino Mega 2560 wurde insbesondere aufgrund der großen Anzahl von 54 GPIOs [O24] ausgewählt, da der *Health Controller* eine große Anzahl an Sensoren und

Aktuatoren einbindet. Beim Demonstrator hat der Einsatz des Arduino Rechners für den *Health Controller* gut funktioniert. Um die Verarbeitung der zahlreichen Sensoren und Aktoren des *Health Controllers* zu optimieren, kann jedoch die Auswahl eines eingebetteten Systems mit einem Mehrkernprozessor sinnvoll sein.

Der ***Perception Controller*** des DAEbots nutzt einen Mehrkernprozessor (ARMv8 [O7]). Wenngleich die durchschnittliche CPU-Auslastung über alle Rechenkerne bei etwa 21 % liegt, ist mindestens ein Kern des Prozessors bei der Ausführung von Bildverarbeitungsalgorithmen ausgelastet. Dies führt dazu, dass die pulseAT Antwortzeit Ausreißer bis in den inakzeptablen Sekundenbereich aufzeigt. Hierbei sei zu erwähnen, dass die Einbindung der 3D-Tiefenkamera (Kinect) zu einer weit höheren Auslastung führt, als die Nutzung der RGB-Kamera. Bei der experimentellen Umsetzung ist dies spürbar. So ist die Framerate der bearbeiteten Bilder im Videostream sehr gering (ca. 5-10 fps). Ein Raspberry Pi 3 Model B+ als *Perception Controller* kann zwar brauchbare Ergebnisse liefern, ein leistungsstärkerer Rechner könnte diese Ergebnisse jedoch stark verbessern und den *Perception Controller* insgesamt aufwerten.

Für den ***reflektorischen Operator*** wird ebenfalls ein Raspberry Pi 3 Model B+ eingesetzt. Im Gegensatz zum *Perception Controller* arbeitet dieser jedoch nicht an der Leistungsgrenze. Zudem muss beim *reflektorischen Operator* keine harte Echtzeit eingehalten werden. Die CPU-Auslastung ist zwar nicht konstant, hat jedoch keine große Varianz (vgl. Standardabweichung von ca. 13 %). Die durchschnittliche pulseAT Antwortzeit ist mit 329 ms zwar vergleichsweise hoch, die Streuung ist jedoch gering. Hierbei sei zu erwähnen, dass der *pulseAT Agent* des *reflektorischen Operators* und der *pulseAT Manager* zwar auf dem gleichen Rechner implementiert sind, die *Manager-Anfrage* und die Antwort des *pulseAT Agenten* jedoch über den CAN-Bus als Hardware-Loopback laufen. Beim DAEbot hat sich diese Auswahl für den *reflektorischen Operator* bewährt. Die CPU-Auslastung kann sich durch komplexere und umfassendere Verarbeitungen im MATLAB Simulink (Stateflow) Model noch erhöhen. Hierbei ist jedoch aktuell nicht abzusehen, dass die Leistungsfähigkeit des Raspberry Pi 3 Model B+ die höhere Last nicht verarbeiten könnte. Die Mehrkern-Architektur des ARMv8-Prozessors führt dazu, dass die diversen Dienste und Applikationen, wie u. a. das erwähnte MATLAB Modell, die Verarbeitung der MQTT Schnittstelle und die Anbindung der Cloud parallel verarbeitet werden können. Nicht empfehlenswert ist die Nutzung eines Einkern-Prozessors für den *reflektorischen Operator*, da die Parallelisierung nur eingeschränkt möglich ist.

7.1.5 Integration der Cloud

Die Integration des Cloud Computing in die Systemarchitektur des DAEbots war erwartungsgemäß aufwendig. Die Umsetzung der Infrastruktur in der Cloud ist dabei aus Sicht der Entwickler*innen nicht besonders komplex, bedarf jedoch Expertise in einem

anderen Bereich als bei der hardwarenahen Programmierung der *Controller* und des *reflektorischen Operators*. Werden bei der hardwarenahen Programmierung typischerweise Applikationen mit C/C++ entwickelt und Kenntnisse im Bereich der Hardware benötigt, so erfordert die Entwicklung von Cloud-Applikationen Kenntnisse im Bereich von virtuellen Maschinen, Mircoservices und Datenbanken. Die genannten Vorteile, wie die dauerhafte Verfügbarkeit der Daten, sowie die nahezu uneingeschränkte Leistungsfähigkeit der Cloud-Rechner, überwiegen die Nachteile durch den Implementierungsaufwand deutlich. Weiterhin kann der Implementierungsaufwand dadurch gesenkt werden, dass verfügbare und weit verbreitete Dienste, wie der TICK Stack verwendet werden. Ein weiterer Nachteil ist die Notwendigkeit, das System gegen Angriffe von außen (Security) zu schützen, insbesondere falls die Cloud über ein öffentliches Funknetz eingebunden wird [25]. Dieses wird bei der Umsetzung des Prototyps nicht weiter betrachtet, da das System bisher ausschließlich im lokalen Funknetz eingesetzt wurde. Ein weiteres Argument für die Integration der Cloud in die Systemarchitektur, ist die Reduktion der lokal auf dem Roboter verfügbaren Rechenleistung und die damit einhergehende Reduktion der elektrischen Leistung. Im Falle des DAEbots beträgt der lokale elektrische Leistungsbedarf im Stand und im Energiesparmodus ca. 20 W. Bei Aktivierung aller Komponenten werden ca. 40 W benötigt, wobei die Systemkomponenten des DAEbots potenziell noch deutlich weniger elektrische Energie benötigen könnten (wird nachfolgend in Kapitel 7.1.7, u. a. mit Tabelle 7.3 erläutert). Würde man die verteilte Systemarchitektur durch ein monolithisches System mit einem Intel NUC (NUC8i5BEK) Rechner ersetzen, so kann laut [O27] allein der Rechner eine elektrische Leistung von 70 W benötigen[6]. Die Realisierung des *kognitiven Operators* auf einer lokalen Komponente wie dem Jetson TX2 könnte laut [20] die lokal aufzubringende elektrische Leistung um 15 W erhöhen.

7.1.6 Implementierung der Schnittstellen

Die Auswahl und Implementierung der Schnittstellen[7] werden nach der Evaluation mit dem Demonstrator ebenfalls als geeignet evaluiert. Die lokale Kommunikation mittels CAN-Bus und adaptiertem Header funktioniert zuverlässig und effizient. Die Analyse der Antwortzeiten der lokalen Komponenten des DAEbots und des AMiRos in Kapitel 7.1.4 zeigen keine ungeklärten Auffälligkeiten. Die Realisierung der MQTT-Schnittstellen auf den höheren Ebenen des OCM sind ebenso positiv zu bewerten. Einzig bei der Übertragung in die Cloud kommt es sowohl bei MQTT-Nachrichten, als auch bei ILP-Befehlen bei hohem Datenaufkommen zum zeitversetzten, aber zuverlässigem Empfang der Daten am *kognitiven Operator*. Da im kognitiven Kreis[8] jedoch keine echtzeitfähige Übertragung notwendig ist,

[6] vgl. Kapitel 4.2.1, Tabelle 4.1
[7] vgl. Kapitel 4.4
[8] vgl. Kapitel 4.4.4

stellt das kein Problem dar. Die Zeitverzögerung durch die Pufferung der Daten könnte vermieden werden, indem z. B. das QoS-Level verringert wird. Hierbei besteht jedoch die Gefahr, dass dann der rechtzeitige Empfang der Daten nicht gesichert ist.

7.1.7 Realisierung von Schlüsselmerkmalen

Die in Kapitel 4.2 vorgestellten Schlüsselmerkmale zeigen einen Auszug der wichtigen Aspekte des konzeptionellen Modells. Nachfolgend werden einige dieser Schlüsselmerkmale und deren Umsetzung evaluiert.

Modularität

Verteilte Systeme sind, wie in Kapitel 4.2.2 erläutert, aufgrund der zugrunde liegenden Architektur modular aufgebaut. Die vorgeschlagene Systemarchitektur erhöht diese Modularität u. a. durch die klar definierte Abtrennung zwischen den Aufgaben der einzelnen Komponenten. Die Modularität in Bezug auf die Rechner des DAEbots ist zudem durch die Verwendung eines Bus-Systems gewährleistet. Rechner können ausgetauscht und neue Komponenten hinzugefügt werden, indem die Middleware der CAN-Schnittstelle[9] verwendet wird. So wurde im Laufe der Entwicklung des DAEbots beispielsweise der Rechner des *Perception Controllers* ausgetauscht. Zur Evaluation von verschiedenen Betriebssystemen für den STM32F3 Discovery SBC des *Motion Controllers* wurde eine Abstraktionsschicht zwischen dem Betriebssystem und der eingebundenen Peripherie namens PDAL[10] entwickelt. Nichtsdestotrotz lag insbesondere bei der Implementierung des *reflektorischen Operators* der Fokus nicht auf der Modularität. So basiert beispielsweise die Generierung des Quelltextes aus dem MATLAB Simulink (Stateflow) Modell auf einem *Support Package* für den dort eingesetzten Raspberry Pi. Bei der Adaption des Modells auf das *Cognition*-Board des AMiRos musste deshalb viel Arbeit in die Adaption der Generierung des Quelltextes investiert werden. Da die Code-Generierung für das Gumstix-Modul des *Cognition*-Boards direkt aus MATLAB heraus nicht verfügbar ist, muss das Generieren des Quelltextes manuell durch das Ausführen eines Bash-Skriptes erfolgen. Da der Quelltext außerhalb von MATLAB generiert wird, ist somit auch die Simulink-Simulation des Modells auf der Hardware (HiL) nicht möglich.

Entkopplung

Die Entkopplung der einzelnen Komponenten innerhalb eines verteilten Systems, wird in Kapitel 4.2.3 als einer der Hauptvorteile eines verteilten Systems und insbesondere der OCM-Architektur beschrieben. Die experimentelle Umsetzung des DAEbots bestätigt dieses. Die klare Trennung der einzelnen Ebenen im OCM führt dazu, dass Aufgaben eindeutig

[9] vgl. Kapitel 5.2.1
[10] vgl. Abbildung 5.14 in Kapitel 5.3.1

auf die einzelnen Komponenten verteilt werden. So ist beispielsweise klar definiert, dass der *kognitive Operator* keinen direkten Einfluss auf die *Controller* nehmen kann. So hatten Verzögerungen bei der Datenübertragung in und aus der Cloud beispielsweise keine Auswirkungen auf die sichere Bewegung des mobilen Roboters. Auch die Trennung innerhalb einer Ebene, welches durch die klare Kapselung der Aufgaben auf verschiedene Komponenten in der vorgeschlagenen Systemarchitektur vorgegeben ist, erhöht die Sicherheit (Safety) und Zuverlässigkeit des Systems. So ist z. B. der *Motion Controller* nur für die Bewegung des Roboters zuständig. Softwarefehler, wie der in Kapitel 7.1.2, Abbildung 7.1 beschriebene Fehler des *Perception Controllers*, hatten ebenfalls keine Auswirkungen auf die anderen Komponenten des Systems.

Sicherheit (Safety) und Zuverlässigkeit

Die Zuverlässigkeit und Sicherheit (Safety) von mobilen Robotern ist, wie in Kapitel 4.2.4 erläutert, von großer Bedeutung. Bei der Umsetzung des DAEbots wird dieser Anforderung insbesondere mit Watchdogs[11] begegnet. Diese Watchdogs haben sich bei den Experimenten bewährt: selbst bei einem mit Absicht hervorgerufenen Absturz des Rechners des *reflektorischen Operators* hielt der sich bewegende DAEbot aufgrund des *Motion Controller*-Watchdogs an. Je nach Abstand zu einem Hindernis hat entweder der Kommunikations-Watchdog[12] die Unterbrechung der Verbindungen zum *reflektorischen Operator* erkannt und den mobilen Roboter gestoppt oder der interne Watchdog des *Motion Controllers* hat das Hindernis erkannt und ebenfalls den DAEbot sicher gestoppt. Die Zuverlässigkeit des DAEbots wurde unter anderem durch den Einsatz des Analyse-Tools pulseAT erhöht. Während der Entwicklung wird durch pulseAT sichergestellt, dass die ausgewählten Rechner die Anforderungen erfüllen und z. B. Deadlines für Echtzeitfunktionen eingehalten werden. Während des Betriebs werden diese Echtzeitfunktionen und die Systemauslastung der einzelnen Rechner weiterhin überwacht. Etwaige Probleme, wie die hohe Antwortzeit des *Perception Controllers* wurden so erkannt und die Software des *Perception Controllers* entsprechend angepasst. Diese Überwachung erhöht die Zuverlässigkeit, da Fehler früh erkannt werden und Anomalien ggf. schon vor dem Eintreten eines Fehlers gelöst werden können. Zudem wird die Zuverlässigkeit des DAEbots erhöht, indem einzelne Komponenten bei Problemen durch die Relais des *Health Controllers* neu gestartet werden können (vgl. Selbstheilung in Kapitel 4.2.4).

Bedarfsgesteuerte Komponentennutzung

Ein weiteres Schlüsselmerkmal des konzeptionellen Modells ist die bedarfsgesteuerte Komponentennutzung, welche in Kapitel 4.2.5 eingeführt wurde. Diese sieht vor, dass Rechner und Peripherie, wie z. B. Sensoren nur dann eingeschaltet werden, wenn sie

[11] vgl. Kapitel 5.2.4
[12] vgl. Abbildung 4.6 in Kapitel 4.2.4

benötigt werden. Dieses soll die Energieeffizienz des Gesamtsystems, neben der Verwendung von energieeffizienten SBC[13], weiter steigern. Im Feldversuch werden hierbei die drei implementierten Energie-Modi (Standby, Energie sparen und volle Leistung) des DAEbots evaluiert und die Leistungsdaten mithilfe des *Health Controllers* gemessen.

Komponente (zusätzlich) eingeschaltet	**Modus**	**Leistungs-bedarf**	**Differenz zur vorherigen Messung**	**Differenz zum Modus „Energie sparen“**	
		[W]	[W]	[W]	[%]
(Nur Health Controller eingeschaltet)	Standby	14,8	-2,7	-4,8	-27
Reflektorischer Operator		17,5	-2,1	-2,1	-11
Motion Controller	**Energie sparen**	**19,6**			
Perception Controller		22	2,4	2,4	14
Display		24,8	2,8	5,2	30
3D-Tiefenkamera		36,8	12	17,2	98
Reflektorischer Operator+		38,2	1,4	18,6	106
Motortreiber	volle Leistung	40,3	2,1	20,7	118
(Fahren mit 60 cm/s)		51,5	11,2	31,9	182

Tabelle 7.3: Leistungsmessung des DAEbots

In Tabelle 7.3 ist der Leistungsbedarf in Abhängigkeit des Zustands der Komponenten (an/aus) dokumentiert. Im Standby-Modus ist nur der *Health Controller* des DAEbots aktiviert. Der *Health Controller* wird für die Aktivierung aller anderen Komponenten durch das dort angebundene Relais-Board benötigt. Da der *Health Controller* zudem die Spannungsregelung durch diverse Regler (und Kühlkörper) beinhaltet, ist der Leistungsbedarf dieser Komponente und damit die Grundleistung des Gesamtsystems mit 14,8 W sehr hoch. Da der Arduino Rechner selbst laut [O24] einen Leistungsbedarf von unter 1 W haben sollte, erscheinen die 14,8 W selbst unter Berücksichtigung der Regelung der Spannungsversorgung sehr groß. Die in der studentischen Arbeit [B2] realisierte Erweiterung der Hardware des *Health Controllers*, kann hier in Bezug auf die Energieeffizienz noch optimiert werden. Das Einschalten des *reflektorischen Operators*

[13] vgl. Kapitel 4.2.1

erhöht die Leistung um 2,7 W. Mit der Aktivierung des *Motion Controllers* wird der Energiesparmodus des DAEbots erreicht. Hierbei kann der DAEbot eigenständig und sicher bewegt werden. Dieser Energiesparmodus wird als Bezugspunkt definiert, da dieser die Mindestanforderungen an die Handlungsfähigkeit eines mobilen Roboters erfüllt. Im Energiesparmodus beträgt der Leistungsbedarf 19,6 W (vgl. -27 % Leistung bezogen auf den Bezugspunkt im Standy Modus). Der *Perception Controller* wird eingeschaltet, wenn beispielsweise AprilTags erkannt werden sollen. Dies steigert den Leistungsbedarf in Bezug auf den Bezugspunkt um 14 %. Der Leistungsbedarf erhöht sich um weitere 2,8 W, wenn das Display des *reflektorischen Operators* aktiviert wird, um beispielsweise Einstellungen am DAEbot vorzunehmen oder den Status des Roboters zu überprüfen. Für die erweiterte Bildverarbeitung mittels 3D-Tiefenkamera (Kinect) und *reflektorischen Operator+*, erreicht der DAEbot einen Leistungsbedarf von 38,2 W. Mit eingeschalteten Motortreibern ist der Energiemodus „volle Leistung“ aktiviert. Hierbei liegt der Leistungsbedarf um 118 % über der Leistung im Energiesparmodus (Bezugspunkt). Den Spitzenwert erreicht der DAEbot, wenn der Roboter zudem mit etwa 60 cm/s fortbewegt wird[14].

Die bedarfsgesteuerte Komponentennutzung wurde mit dem DAEbot positiv evaluiert. Typischerweise wird der DAEbot im Energiesparmodus betrieben. Der Leistungsbedarf kann mit dem Energiesparmodus deutlich gesenkt werden, obwohl der mobile Roboter selbst noch die Grundfunktionen, wie das sichere Fortbewegen, ausführen kann. Eine weitere Optimierung der Hardware des *Health Controllers* würde die in Tabelle 7.3 dargestellte Differenz zum Energiesparmodus voraussichtlich noch weiter verbessern und den Nutzen der bedarfsgesteuerten Komponentennutzung noch deutlich erhöhen.

Softwareentwicklung

Wie in Kapitel 4.2.7 beschrieben, ist die Software des DAEbots umfangreich. Die Entwicklung der Software des DAEbots und des Analyse-Tools pulseAT war sehr zeitaufwendig. Je nach Ebene des OCM sind unterschiedliche Expertisen der Entwickler*innen gefragt. Auf der *Controller*-Ebene wurden typische regelungstechnische Methoden, mit den Programmiersprachen C und selten C++ umgesetzt. Die Herausforderung bei der Entwicklung auf der *Controller*-Ebene sind die hardwarenahe und effiziente Programmierung. Bei der Entwicklung des *reflektorischen Operators* kommen ebenfalls hauptsächlich die hardwarenahen Programmiersprachen C und C++ zum Einsatz. Der größte Aufwand steckt beim *reflektorischen Operator* jedoch in der Entwicklung des MATLAB Simulink Modells[15]. Wenngleich die Generierung des Quelltextes aus MATLAB heraus sehr lange dauert (ca. 5 Minuten), erleichtert die modellbasierte Entwicklung die Programmierung des DAEbots sehr. Kenntnisse aus dem Bereich IoT Entwicklung sind bei der Realisierung der MQTT-Schnittstelle und der Anbindung des Cloud-basierten *kognitiven*

[14] vgl. Abbildung 5.25 in Kapitel 5.3.6
[15] vgl. Kapitel 5.2.3

Operators gefragt. Gleiches gilt für den *kognitiven Operator* selbst, welcher größtenteils durch eine typische IoT-Infrastruktur implementiert wird. Da die Softwareentwicklung zur Umsetzung des gesamten konzeptionellen Modells, inklusive der Softwareentwicklung für den DAEbot, pulseAT und die Adaptierung der Software für den AMiRo, für den Autor allein zu umfangreich ist, werden viele Softwaremodule in diversen, vom Autor betreuten, studentischen Arbeiten ([B3–B11]) umgesetzt. Hierbei ist die Nutzung eines Tools zur Versionierung unabdingbar. Für die Softwareentwicklung im Rahmen dieser Dissertation wurden durchgehend GitLab [O72] und die dazugehörigen Funktionen, wie das Wiki oder die Issues als Kanban-Board [O73] eingesetzt.

7.1.8 Adaption an die bestehende Software des AMiRo

Während der Implementierung des DAEbots startet die Adaption der Software des DAEbots in die bestehende Software des AMiRos. Ziel der Adaption ist die Evaluation, ob die vorgeschlagene Systemarchitektur auch auf andere mobile Roboter übertragbar ist und ob hierbei die Konzepte die Funktion des mobilen Roboters verbessern. Diese Adaption erfolgt in zwei studentischen Arbeiten. Zunächst wird die CAN-Infrastruktur[16] des DAEbots an die Software des AMiRo adaptiert [B9]. Hierbei wird die Software des AMiRo-OS so angepasst, dass Sensordaten in Abhängigkeit der Konfiguration des zugehörigen CAN-Publishers berechnet werden. Die CAN-Publisher können dann durch das ebenfalls migrierte MATLAB Simulink (Stateflow) Modell konfiguriert werden. Später erfolgt darauf aufbauend die Integration von pulseAT in die Software des AMiRos [B11]. Als Grundlage der studentischen Arbeiten dient das AMiRo-OS in der Version 1.0_stable[17], welche, ebenso wie die Hardware des AMiRo, am CITEC der Universität Bielefeld entwickelt wird.

Zur Evaluation werden die Daten zur Systemauslastung des AMiRo-OS 1.0 mit der Erweiterung durch die Teile der Middleware des DAEbots verglichen. Hierbei wird das AMiRo-OS 1.0 durch die *pulseAT Agenten* erweitert, da die Daten zur Systemauslastung ansonsten nicht zur Verfügung ständen. Ziel der Integration der CAN-Middleware des DAEbots in die Software des AMiRo-OS ist es, die Konfiguration der AMiRo-*Controller* in Bezug auf die Häufigkeit der Übertragung der CAN-Publisher zu optimieren.

Die Ergebnisse dieser Evaluation sind in Tabelle 7.4 dokumentiert. Hierbei wird deutlich, dass die CPU-Auslastung mit AMiRo-OS für alle drei Rechner sehr niedrig ist. In AMiRo-OS 1.0 werden alle Sensordaten mit 16 Hz an das *Cognition*-Modul übertragen. Mit der Integration der DAEbot CAN-Middleware ist es nun möglich, die Publisher der Sensoren einzeln über das *Cognition*-Modul als *reflektorischer Operator* zu konfigurieren. Zudem können nun Nachrichten, u. a. mit dem MATLAB Simulink (Stateflow) Modell, dynamisch priorisiert werden, sodass z. B. Nachrichten zur Steuerung der Motoren mit hoher Priorität

[16] vgl. Kapitel 5.2.1
[17] Commit `5b1b6715` in Git-Repository `amiro-os` vom 11.04.2018

Firmware	Publisher Frequenz [Hz]	Di Wheel Drive				Power Management				Light Ring			
		CPU-Auslastung [%]	RAM-Auslastung [%]	Anzahl Threads	Antwortzeit [ms]	CPU-Auslastung [%]	RAM-Auslastung [%]	Anzahl Threads	Antwortzeit [ms]	CPU-Auslastung [%]	RAM-Auslastung [%]	Anzahl Threads	Antwortzeit [ms]
AMiRo-OS	16	1	31	18	3,8	1	25	26	3,8	1,3	24	9	3,8
DAEbot-Middleware	0	1	37	22	3,8	1,1	30	28	3,7	1	31	9	3,6
	16	1	37	22	3,8	1,1	30	28	3,7	1	31	9	3,9
	160	1,3	37	22	4,3	1,2	30	28	4,3	1,9	31	9	4,2
	500	1,1	37	22	14	1,7	30	28	10	2,8	31	9	12

Tabelle 7.4: Evaluation der entwickelten Middleware anhand der Systemauslastung des AMiRo

gesendet werden. Zunächst werden alle Publisher ausgeschaltet (0 Hz). Die CPU-Auslastung liegt bei allen Modulen bei 1 %. Im Vergleich zum AMiRo-OS sind die RAM-Auslastung und die Anzahl an Threads leicht erhöht, da zusätzliche Threads die Konfiguration der Publisher ausführen. Bei einer Frequenz von 16 Hz für alle Publisher ist die CPU-Auslastung weiterhin niedrig. Die pulseAT Antwortzeiten sind nahezu identisch mit den Antwortzeiten bei 0 Hz und der Vergleichsmessung mit dem AMiRo-OS. Erst bei der Erhöhung aller Publisher-Frequenzen auf 500 Hz sind Auswirkungen auf die Systemauslastung auffällig. Die pulseAT Antwortzeit ist für alle Rechner deutlich höher. Hier ist der CAN-Bus bzw. dessen Konfiguration der Flaschenhals des Systems. Dass die CPU-Auslastung weiterhin gering ist, kann auch daran liegen, dass die Anzahl der Sensoren und Aktuatoren im Vergleich zum DAEbot geringer ist. Bei einer Frequenz von 160 Hz sind die pulseAT Antwortzeiten wieder gering und weisen, wie im Detail in Tabelle 7.2 (Kapitel 7.1.4), sowie als Box-Plot in Anhang E in Abbildung A8 dargestellt, kaum Streuung auf. Die CPU-Auslastung ist weiterhin gering[18]. Die Frequenz von 160 Hz für alle Publisher hat somit keine negativen Auswirkungen auf die Funktion der *Controller* des AMiRo. Trotzdem konnte die Frequenz der Übertragung der Sensordaten durch die Integration der CAN-Schnittstelle des DAEbots von 16 Hz auf 160 Hz verzehnfacht werden. Hinzu kommen die dynamische Anpassung der

[18] Details zur CPU-Auslastung bei einer Frequenz von 160 Hz sind in Tabelle 7.1 (Kapitel 7.1.4) dargestellt.

Prioritäten der Nachrichten, sowie die Analyse der Daten zur Systemauslastung mit pulseAT. Die Realisierung des Cloud-basierten *kognitiven Operators* wurde nicht realisiert und kann somit nicht im Zusammenhang mit dem AMiRo evaluiert werden.

7.2 Evaluation der Systemarchitektur anhand der Anforderungen und Herausforderungen

Die Evaluation der Systemarchitektur anhand der Anforderungen und Herausforderungen beginnt mit der Erläuterung der Methodik (Kapitel 7.2.1). Es folgt eine Übersicht über das Evaluationsergebnis in Kapitel 7.2.2. Abschließend wird die Bewertung anhand der Anforderungen (Kapitel 7.2.3) und Herausforderungen (Kapitel 7.2.4) begründet.

7.2.1 Methodik

Die Evaluation von System- oder Softwarearchitekturen erfolgt oftmals anhand des subjektiven Eindrucks der Entwickler*innen und ggf. der Nutzer*innen eines technischen Systems. So wird die Architektur i. d. R. während der Entwicklung angepasst und optimiert, bis die Anforderungen an das System erfüllt und somit auch die gewählte Architekturform positiv bewertet werden kann. Umfragen unter den Nutzer*innen eines Systems können dabei helfen, dass die Entwickler*innen Rückschlüsse auf die Architektur des Systems und der Software ziehen. Diese Rückmeldungen sind jedoch subjektiv. Die subjektiven Eindrücke der Nutzer*innen und Entwickler*innen sollen in diesem Kapitel formalisiert werden, um somit eine objektive Beurteilung der vorgestellten Systemarchitektur zu ermöglichen. Dieses soll zudem sicherstellen, dass die Eignung der Systemarchitektur für den in Kapitel 3, in Form einer Taxonomie, definierten archetypischen mobilen Roboter allgemeingültig ist. Hierzu wird eine Recherche nach Bewertungskriterien für die Evaluation einer System- und oder Softwarearchitektur durchgeführt.

Bass, Clements und Kazman sprechen in ihrem Buch „*Software Architecture in Practice*“ [6, S. 397 ff.] von drei Formen der Evaluation von Architekturen. Die erste Form ist die Evaluation durch den*die Designer*in der Architektur während des Entwicklungsprozesses. Die zweite Form ist die Evaluation der Architektur durch Begutachter*innen im Entwicklungsprozess (*Peer Review*). Bei der dritten Form analysieren außenstehende Gutachter*innen die Architektur nach der Fertigstellung. In allen drei Formen wird die Analyse-Methode „Architecture Tradeoff Analysis Method (ATAM)“ verwendet. Diese Methode gibt einen Rahmen vor, in dem die Evaluation einer Architektur erfolgt. Dieser Rahmen sieht die in [6, S. 403 ff.] definierten Schritte vor:

1. ATAM präsentieren - Stellt den Stakeholder*innen das Konzept von ATAM vor und beantwortet alle Fragen zum Prozess.
2. Geschäftstreiber präsentieren - Jede*r im Prozess präsentiert und bewertet die Geschäftstreiber für das betreffende System.
3. Die Architektur präsentieren - Der*die Architekt*in stellt dem Team die High-Level-Architektur mit einem angemessenen Detaillierungsgrad vor.
4. Identifizieren von Architekturansätzen - Verschiedene Architekturansätze für das System werden vom Team vorgestellt und diskutiert.
5. Generieren von Qualitätsmerkmalen - Definiert die zentralen geschäftlichen und technischen Anforderungen des Systems und ordnet diese einer geeigneten Architektureigenschaft zu. Präsentiert ein Szenario für diese gegebene Anforderung.
6. Analysieren der Architekturansätze - Analysiert jedes Szenario und bewertet es nach Priorität. Die Architektur wird dann anhand der einzelnen Szenarien bewertet.
7. Brainstorming und Priorisierung der Szenarien - Präsentiert den Stakeholder*innen die aktuellen Szenarien und erweitert diese.
8. Analysieren der Architekturansätze - Erneute Durchführung von Schritt 6 mit dem zusätzlichen Wissen der größeren Stakeholder-Gruppe.
9. Ergebnisse präsentieren - Stellt alle Unterlagen den Stakeholder*innen zur Verfügung.

Das Ergebnis des ATAM Ansatzes ist eine Szenario-basierte Evaluation der Architektur, welche je nach Anwendungsfall bzw. Szenario unterschiedlich ausfallen kann. Das ATAM Modell kann somit die Allgemeingültigkeit einer System- oder Softwarearchitektur nur bedingt belegen.

Rozanski und Woods zeigen in ihrem Buch „*Software Systems Architecture: Working With Stakeholders using Viewpoints and Perspectives*“ [8, S. 217 ff.], neben der Szenario-basierten Evaluation, u. a. die Evaluation durch Prototypen und *Proof-of-Concept* Systemen. Dieser Nachweis wurde mit der experimentellen Umsetzung des DAEbots erbracht. Eine weitere Option nennt Rozanski und Woods mit Evaluation eines sogenannten *Skeleton*[19]-Systems. Ziel dieses *Skeleton*-Systems ist die Evaluation eines technischen Systems in einer ersten Version, welche die Architektur realisiert, daneben jedoch nur die Minimalfunktionen des Systems implementiert. Die Evaluation des *Skeleton*-Systems erfolgte ebenfalls bei der Entwicklung des DAEbots. Rozanski und Woods ordnen in Abbildung 7.6 die verschiedenen Methoden zur Evaluation einer (Software-) Architektur in den Lebenszyklus einer Software ein. So ist die Evaluation durch Überprüfungen durch mehrere Personen (*Reviews and Walkthroughs*) schon vor der Entwicklung der Software möglich. Der Szenario-basierte Ansatz, wie z. B. durch das ATAM Modell, findet laut [8] ebenfalls bereits vor der

[19] dt. Gerüst

Fertigstellung des Systems statt. Die Abbildung zeigt nicht die Validierung einer Architektur eines bestehenden Systems. Hierzu nennen die Autoren u. a. die Evaluation aufgrund der Einhaltung der Anforderungen an das System. Sind diese erfüllt, so ist prinzipiell auch die Architektur geeignet [8, S. 233 ff.]. Als Ergebnis aller Evaluationsmethoden nennen Rozanski und Woods Gesprächs- und Entscheidungs-Protokolle, sowie Prüf- und Evaluationsberichte.

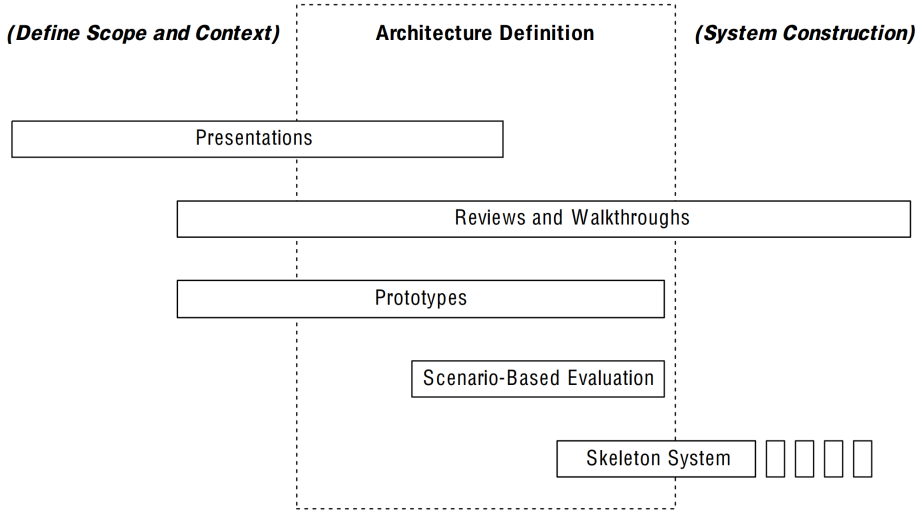


Abbildung 7.6: Evaluationsmethoden von Architekturen an verschiedenen Punkten des Lebenszyklus [8, S. 231]

Balzert spricht in seinem „*Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*“ im Kapitel „Qualitätssicherung der Architektur“ [18, S. 487 ff.] neben Szenario-basierten Verfahren von Verfahren auf Basis von Metriken, wie z. B. die statische Codeanalyse. Zudem erläutert der Autor die Gruppe der Nachweisverfahren, welche u. a. Simulationen, Prototypen und Experimente beinhalten.

Die Literaturrecherche im Bereich Software- oder Systemarchitektur-Analyse oder Evaluation zeigt diverse Übersichtspapiere mit Szenario-basierten Methoden. Patidar und Suman betrachten und vergleichen in [228] diverse Methoden, die Ähnlichkeiten zum gezeigten ATAM Ansatz aufweisen. Hierbei werden im Jahr 2015 acht Methoden miteinander verglichen, welche alle zwischen den Jahren 1994 und 1999 veröffentlicht wurden. Die in [228] beschriebenen offenen Probleme, wie die Abdeckung aller möglichen Szenarien durch die Szenario-basierten Methoden, scheinen bis heute nicht gelöst zu sein. Als weitere offene Fragestellungen benennen die Autoren den großen Zeitaufwand und die fehlenden Tools zur Unterstützung des Evaluations-Prozesses. Zu einem ähnlichen

Ergebnis kommen bereits im Jahr 2002 Patidar und Suman in [228]. Dieses wird später durch Dobrica und Niemela mit [229] erneut bestätigt. In [230] fokussieren sich die Autoren nur auf Szenario-basierte Ansätze zur Evaluation von Softwarearchitekturen. Ihr Ergebnis bestätigt die Ergebnisse aus [228] und [229]. Zwar bietet ATAM einen definierten Prozess, doch auch hier bemängeln Babar und Gorton die fehlende Unterstützung durch Tools [230].

Die Bewertung und Beurteilung der vorgeschlagenen Systemarchitektur erfolgte bereits während der Entwicklung des DAEbots und der Adaption der Software an den AMiRo in Form von Interviews mit den sieben Masterstudenten der Projekt- und Abschlussarbeiten [B1–B11]). Hierbei werden die Kernmerkmale der vorgestellten Systemarchitektur evaluiert. Die Bewertung fiel dabei stets positiv aus. Interviews zur Eignung der vorgeschlagenen Systemarchitektur finden zudem häufig unter den Mitarbeitern des Software for Robots (S4R)-Forschungsprojektes statt. Der Autor dieser Dissertation ist ebenfalls Teil der S4R-Mitarbeiter. Im Rahmen des S4R-Projekts wird ein UAS als Use-Case für die Etablierung eines Entwicklungsprozesses [A13] für UAVs implementiert. Das Forschungsprojekt beinhaltet die Definition oder Entwicklung von Tools und einer Systemarchitektur zur Entwicklung von UAVs. Die Begutachtung in Form von Interviews und Diskussionen der S4R-Mitarbeiter bezieht sich nach Bass, Clements und Kazman [6] auf zwei Phasen. Auf der einen Seite wird begutachtet, wie die Architektur nach Fertigstellung des DAEbots zu bewerten ist. Auf der anderen Seite findet diese Begutachtung vor der Implementierung der S4R-UAVs statt. So wird bewertet, ob die vorgeschlagene Systemarchitektur auch bei den UAVs eingesetzt werden soll. Diese Interviews und Diskussionen starten bereits in frühen Phasen des S4R-Projekts und sind ein fortlaufender Prozess.

Um die Evaluation der vorgestellten Systemarchitektur zu formalisieren und abzuschließen, wurde zusätzlich zu den Interviews die ATAM-Methode eingesetzt. Die in ATAM definierte strukturierte Diskussion über System- und Softwarearchitekturen zur Evaluation der Architekturen durch Gutachter*innen erfolgt im März 2021 in Form eines Peer-Reviews mit drei S4R-Mitarbeitern (inklusive des Autors). Die am Review teilnehmenden S4R-Mitarbeiter sind allesamt Promovenden und Experten im Bereich der mobilen Robotik. Teilnehmer „1“ forscht an der Navigation von Robotern. Teilnehmer „2“ beschäftigt sich in seinem Promotionsprojekt im weitesten Sinne mit der Bildverarbeitung von UAVs. Der Autor (Teilnehmer „3“) forscht bekanntlich an einer Systemarchitektur für mobile Roboter.

Das Gutachten der vorgeschlagenen Systemarchitektur mit der ATAM-Methode ist dabei stark an die in [6, S. 403 ff.] definierten Schritte angelehnt, wobei einzelne Schritte aufgrund des Vorwissens der Teilnehmer übersprungen wurden. So stellt der Autor zwar die ATAM-Methode vor (Schritt 1), die in Schritt 2 zu präsentierenden Geschäftstreiber

sind den Teilnehmern jedoch grundsätzlich bekannt. Die Präsentation der Architektur (Schritt 3) erfolgt in Form von Kapitel 1 bis Kapitel 6 dieser Dissertation, wobei sich die Teilnehmer selbstverständlich bereits im Laufes des Forschungsprojektes mit der Architektur vertraut gemacht haben. Die in Schritt 5 zu definierenden Qualitätsmerkmale werden den in Kapitel 3.4 definierten Anforderungen und Herausforderungen entnommen. Diese Anforderungen und Herausforderungen sind aus der systematischen Literaturrecherche zur Bestimmung einer Taxonomie (vgl. Tabelle 3.5 in Kapitel 3.3) für einen archetypischen mobilen Roboter abgeleitet worden. Die Analyse des Architekturansatzes (Schritt 6) erfolgt anhand von verschiedenen Szenarien, welche für jede Anforderung und Herausforderung diskutiert wurden. Diese Analyse fand in einer offenen Diskussion, mit allen Teilnehmern gleichzeitig statt. Jeder Teilnehmer hat hierbei die nachfolgende Tabelle 7.5 ausgefüllt und somit die vorgeschlagene Systemarchitektur einzeln bewertet. Es kam zu keinen maßgeblichen Abweichungen in der Bewertung. Die Analyse wurde mit der Zustimmung aller Teilnehmer in Form einer Videoaufzeichnung protokolliert. Die Präsentation des Ergebnisses der Analyse (Schritt 7 bzw. 9) ist in Tabelle 7.5 dokumentiert. Außerdem werden die Kernpunkte der Diskussion nachfolgend in Kapitel 7.2.3 und Kapitel 7.2.4 zusammenfassend dokumentiert. Grundlage der Dokumentation, ist die Videoaufzeichnung der Analyse. Die Teilnehmer haben dem Ergebnis und der nachfolgenden Dokumentation zugestimmt, sodass das Ergebnis der Evaluation als Konsens aller Teilnehmer bezeichnet werden kann.

Die Allgemeingültigkeit der Architektur wurde u. a. bewertet, indem die vorgeschlagene Systemarchitektur in den Stand der Technik eingeordnet wurde. Hierbei wurde auf die in Kapitel 2.5 gezeigten Architekturen und die in Kapitel 3.2.3 dargestellten Referenzsysteme aus der mobilen Robotik Bezug genommen. Weiterhin wird die Eignung der Systemarchitektur auch für weitere typische Anwendungsgebiete (siehe Taxonomie 3.5 in Kapitel 3.2.4) und weitere Klassen (ebenfalls in der Taxonomie enthalten), außerhalb des DAEbots (WMR) und der S4R-UAVs diskutiert und in der Bewertung berücksichtigt.

Als Grundlage dienten die Erfahrungen mit dem DAEbot und die Diskussion über die Eignung der Architektur für die Anwendungsszenarien des S4R-Forschungsprojektes. Hierbei wurden in Kapitel 7.1 bereits einige Evaluationsergebnisse erläutert. Zudem wurden zu den Anforderungen und Herausforderungen, welche im Rahmen dieses Promotionsprojektes nicht praktisch evaluiert werden konnten, Literaturquellen als Grundlage der Bewertungen genutzt. So wird beispielsweise anhand von Veröffentlichungen evaluiert, dass aufgrund der zur Verfügung stehenden Hardware (dem *kognitiven Operator*) das maschinelle Lernen in der vorgeschlagenen Systemarchitektur eingesetzt werden kann.

Für die Bewertung, ob und wie z. B. die Realisierung der Autonomie beim DAEbot mit der vorgeschlagenen Systemarchitektur unterstützt wird, liegen hingegen praktische Erfahrungen mit dem DAEbot vor. So wird bewertet, ob die Autonomie in der vorgeschlagenen Systemarchitektur besser unterstützt wird, als bei anderen Architekturen aus dem Stand der Technik und wie sich die Realisierung von den Referenzsystemen unterscheidet. Zudem wird evaluiert, ob die Systemarchitektur auch dazu geeignet ist, die vergleichsweise hohen Anforderungen an die Autonomie bei den S4R-UAVs zu implementieren.

Nachfolgend werden zunächst eine tabellarische Übersicht und das Ergebnis des Review-Prozesses gezeigt. Im Anschluss daran erfolgt eine zusammenfassende Erläuterung und Begründung der jeweiligen Bewertungen.

7.2.2 Übersicht

Tabelle 7.5 zeigt die Resultate der Evaluation der vorgeschlagenen Systemarchitektur. Hierbei wird eine fünfstufige Bewertung gewählt. Die bestmögliche Bewertung ist hierbei $++$ und zeigt, dass die vorgeschlagene Systemarchitektur einen Fortschritt bzw. große Vorteile gegenüber dem Stand der Technik in Bezug auf die Anforderung oder Herausforderung hat. Die Bewertung $--$ würde bedeuten, dass die Systemarchitektur die Anforderung oder Herausforderung nicht erfüllt. Die mit 0 bewerteten Anforderungen oder Herausforderungen sind in etwa gleich auf mit dem Stand der Technik. Die Bewertungen $+$ und $-$ stellen leichte Vor- oder Nachteile in Bezug auf den Stand der Technik dar. Die mit $\checkmark$markierten Anforderungen und Herausforderungen wurden im Rahmen dieser Dissertation praktisch, d. h. beispielsweise mit dem DAEbot geprüft. Die mit $(\checkmark)$ markierten Anforderungen oder Herausforderungen wurden nur zum Teil praktisch evaluiert.

Die vorgeschlagene Systemarchitektur erfüllt alle Anforderungen und Herausforderungen und ist laut des Gutachtens der S4R-Mitarbeiter in vielen Fällen besser geeignet als der Stand der Technik. Hierbei sind Architekturen oder Referenzsysteme aus dem Stand der Technik zwar in Teilbereichen ähnlich gut aufgestellt, jedoch erfüllt die vorgeschlagene Systemarchitektur alle vogestellten Merkmale. Die S4R-Mitarbeiter entscheiden sich für die Realisierung der vorgeschlagenen Systemarchitektur für die S4R-UAVs. Die Begründung dazu kann den nachfolgenden Betrachtungen der jeweiligen Anforderungen und Herausforderungen entnommen werden.

	#	Anforderung/ Herausforderung	Bewertung	Praktische Evaluation
Anforderungen	A1	Wartbarkeit	−	
	A2	Weiterentwickelbarkeit	0	(✓)
	A3	Sicherheit (Safety)	++	✓
	A4	Sicherheit (Security)	−	
	A5	Zuverlässigkeit	++	✓
	A6	Leistung und Effizienz	+ bis −	✓
	A7	Benutzbarkeit	++	✓
	A8	Portabilität	0	(✓)
	A9	Navigation	+	✓
	A10	Autonomie	+	✓
	A11	Optimierung/ Lernen	++	✓
	A12	Multi-Roboter Kooperation	+	
	A13	Mensch-Roboter Interaktion	++	✓
	A14	Echtzeitfähigkeit	++	✓
	A15	Maschinelles Lernen	+	
	A16	Bildverarbeitung	+	✓
	A17	Cloud Computing	++	✓
	A18	Systemüberwachung	++	✓
	A19	Modularität	+	✓
	A20	Modellbasierte Entwicklung	0	✓
Herausforderungen	H1	Selbstheilung	++	✓
	H2	Redundanz	+	
	H3	Rekonfigurierbarkeit	++	✓
	H4	Verteiltes System	+	✓
	H5	Edge Computing	+	✓
	H6	Einplatinenrechner	++	✓

++ ≙ Große Vorteile (Fortschritt)[1]; + ≙ Vorteile[1]; 0 ≙ Gleichauf[1]; − ≙ Nachteile[1]; −− ≙ Nicht erfüllt[2]
[1] gegenüber/ mit dem Stand der Technik
[2] Anforderung/ Herausforderung wird durch die vorgeschlagene Systemarchitektur nicht erfüllt

Tabelle 7.5: Evaluation anhand der Anforderungen und Herausforderungen an die Systemarchitektur von mobilen Robotern

7.2.3 Anforderungen

Die Anforderungen an die Systemarchitektur für mobile Roboter aus Kapitel 3.4.3 zeigen die aktuellen Anforderungen an die Systemarchitektur. Diese sind in die Kategorien „Allgemeine Anforderungen“ und „Anforderungen aus der mobilen Robotik“ unterteilt.

Allgemeine Anforderungen

Die allgemeinen Anforderungen (A1 bis A8) stammen aus Balzerts Lehrbuch für Softwaretechnik [18, S. 109 ff.]. Bei diesen Anforderungen handelt es sich um nichtfunktionale Anforderungen (NFR), welche generell bei der Entwicklung einer Architektur eingehalten werden sollten. Die Bewertung dieser NFR ist aufgrund der wenig klar definierten, spezifischen Kriterien nicht immer eindeutig. Die Bewertung hängt von sehr vielen Faktoren, wie z. B. vom Anwendungsgebiet oder der Komplexität des Systems ab. So kann z. B. die Wartbarkeit schon wegen des begrenzten Zeitraums (indem keine vollständige Wartung des Systems notwendig ist) im Rahmen dieser Dissertation kaum evaluiert werden. Die Bewertungen durch die S4R-Mitarbeiter erfolgt deshalb teilweise anhand der von Balzert [18] erläuterten Kriterien zur Erfüllung der jeweiligen Anforderung. Diese Kriterien wurden zusammen mit der Erläuterung der allgemeinen und nichtfunktionalen Anforderungen in Kapitel 3.4.1 vorgestellt.

Die **Wartbarkeit (A1)** eines softwareintensiven Systems kann dadurch vorbereitet werden, dass sich später ändernde Komponenten und Funktionen früh identifiziert und geeignete generische Schnittstellen definiert werden und das System analysierbar ist [18, S. 116 ff.]. Der modulare Aufbau der Systemarchitektur fördert den Austausch der Hardware [44]. Wenngleich die Wartung bei der Entwicklung des Prototyps des DAEbots nicht notwendig ist und somit nicht praktisch evaluiert wurde, zeigt beispielsweise die Implementierung der PDAL-Abstraktionsschicht zum Austausch z. B. von den Betriebssystemen FreeRTOS und ChibiOS des *Motion Controllers*[20] einen möglichen Ansatz zur Erhöhung der Wartbarkeit durch die Abstraktion der Software. Zudem kann durch Tools, wie pulseAT die Analysierbarkeit der verteilten Systemarchitektur realisiert werden. Im Gegensatz zu monolithischen Systemen wird die Wartbarkeit bei verteilten Systemen jedoch prinzipiell durch die Verteilung auf verschiedene Rechner erschwert. So sollten z. B. alle Betriebssysteme aller Komponenten neue Sicherheitsupdates bekommen. Beim DAEbot muss hierzu zu jedem Rechner eine (teilweise physische) Verbindung aufgebaut werden. Um Fehlerkorrekturen oder Verbesserungen in der Software zu implementieren, werden zudem unterschiedliche Entwicklungsumgebungen benötigt. Die Wartbarkeit wird somit

[20] vgl. Kapitel 5.3.1

als nachteilig im Vergleich zum Stand der Technik bewertet, da die aufgezählten Nachteile die Vorteile überwiegen.

Die **Weiterentwickelbarkeit (A2)** hängt laut Balzert ebenfalls u. a. von der Analysierbarkeit und Änderbarkeit ab [18, S. 119 ff.]. Die Weiterentwickelbarkeit hängt zudem eng mit der Wartbarkeit von Software zusammen, sodass auch die Nachteile, wie die erhöhte Komplexität durch den Einsatz eines verteilten Systems, in die Diskussion um die Bewertung eingehen. Im Vergleich zur Wartbarkeit wird die Weiterentwickelbarkeit jedoch als gleichauf mit dem Stand der Technik bewertet, da die klare hierarchische Trennung im OCM und die Abkapslung der Rechner, die Erweiterung des Systems stark vereinfachen (vgl. Modularität (A19)). Unterstützt wird die Weiterentwickelbarkeit zudem durch die Middleware (vgl. Kapitel 5.2) und die klare Definition von einheitlichen Schnittstellen, welche dem modularen Konzept von ROS [4] nicht unähnlich ist. Die Realisierung des *Perception Controllers* des DAEbots kann zumindest teilweise als Evaluation der Weiterentwickelbarkeit betrachtet werden, da der *Perception Controller* erst nach der Fertigstellung einer ersten Version des Demonstrators entwickelt und in den DAEbot integriert wurde. Insbesondere aufgrund der Komplexität der vorgeschlagenen Architektur als Nachteil und der Modularität als Vorteil, bewerten die S4R-Mitarbeiter die Weiterentwickelbarkeit als gleichauf mit dem Stand der Technik.

Die Anforderung an die **Sicherheit (Safety) (A3)** ist sowohl eine allgemeine Anforderung aus Balzert [18, S. 121 ff.] (hier als Funktionssicherheit bezeichnet), als auch eine spezifische Anforderung an die Systemarchitektur von mobilen Robotern, da mobile Roboter sicherheitskritische Systeme sind [104]. Balzert benennt die Entkopplung von Komponenten als einen Ansatz zur Erhöhung der Sicherheit (Safety). Die Kapselung von sicherheitskritischen Funktionen wird auch in der ACROSET Architektur [54] als notwendiges Merkmal genannt. Diese Entkopplung bzw. Kapselung ist eines der Hauptmerkmale der vorgeschlagenen Systemarchitektur. Zudem werden u. a. im Abschnitt „Safety“ in Kapitel 4.2.4 weitere Vorteile und Maßnahmen zur Erhöhung der Sicherheit (Safety), wie die Nutzung von Watchdogs (vgl. Kapitel 5.2.4) der vorgeschlagenen Systemarchitektur betrachtet und in Kapitel 7.1.7 evaluiert. Dabei zeigte sich auch bei der experimentellen Umsetzung mit dem DAEbot, dass u. a. durch die Entkopplung von sicherheitskritischen Funktionen von der Planungsebene der Systemarchitektur, die Systemarchitektur die Anforderungen an die Sicherheit in Bezug auf *Safety* erfüllt. Zudem ist die vorgeschlagene Systemarchitektur monolithischen Strukturen, welche u. a. beim Khepera IV [85] und WolfBot [87] eingesetzt werden, in Bezug auf die Sicherheit (Safety) überlegen, da der SPoF von monolithischen Systemen durch die Nutzung von mehreren Rechnern behoben werden kann (vgl. Abschnitt „Safety“ in Kapitel 4.2.4). Die vorgeschlagene Systemarchitektur ist nach der Prüfung der S4R-Mitarbeiter ein Fortschritt gegenüber dem Stand der Technik.

Die Anforderung an die **Sicherheit (Security) (A4)** ist ebenfalls sowohl eine allgemeine Anforderung aus Balzert [18, S. 121 ff.] (hier als Betriebssicherheit bezeichnet), als auch eine spezifische Anforderung an die Systemarchitektur von mobilen Robotern. Bei der vorgeschlagenen Systemarchitektur besteht die Gefahr, dass die Sicherheit (Security) nicht erfüllt wird, falls die in Abschnitt „Security“ in Kapitel 4.2.4 erläuterten Maßnahmen zur Erhöhung der Sicherheit (Security) nicht beachtet würden. Die potenziell größte Schwachstelle der Systemarchitektur ist der Cloud-basierte *kognitive Operator*. Wie bei allen anderen mobilen Robotern mit Cloud-Anbindung (vgl. [119]), muss die Einbindung in die Cloud gegen Angriffe von außen geschützt werden. Hierbei sollten Protokolle zur Erhöhung der Sicherheit (Security) wie TLS eingesetzt werden. Zudem sollte die Cloud-Anbindung ggf. über VPN oder ausschließlich in einer privaten Cloud erfolgen. Eine Authentifizierung vor dem Zugang in das Netzwerk erfolgt beispielsweise auch in [43]. Ggf. kann in einigen Anwendungsszenarien, z. B. bei UAVs, welche geheime oder geschützte Bereiche filmen, der Cloud-basierte *kognitive Operator* durch einen lokalen Rechner als *kognitiven Operator* ersetzt werden. Eine weitere Schwachstelle ist die Struktur als verteiltes System, da mehrere Rechner potenziell mehrere Angriffspunkte ermöglichen. Die strenge Trennung im OCM verhindert zwar, dass z. B. der *kognitive Operator* direkt auf die *Controller* zugreifen kann, trotzdem müssen sich die Entwickler*innen dieser Gefahr bewusst sein und nicht benötigte Schnittstellen deaktivieren (wie im Abschnitt „Security“ in Kapitel 4.2.4 beschrieben). Die Sicherheit (Security) wurde nicht praktisch evaluiert und wird insbesondere im Vergleich zu monolithischen Systemen und aufgrund der Cloud Anbindung als nachteilig in Bezug auf den Stand der Technik bewertet.

Die Fähigkeit, die Bedingungen an die **Zuverlässigkeit (A5)** eines technischen Systems zu erfüllen, ist sowohl eine allgemeine Anforderung nach Balzert [18, S. 124 ff.], als auch eine Anforderung aus der mobilen Robotik. Analog zur Sicherheit (Safety) spielt die Zuverlässigkeit bei mobilen Robotern eine große Rolle. Ein zuverlässiges System kann auch die Anforderungen an die Sicherheit (Safety) erfüllen. Die Zuverlässigkeit kann im vorgeschlagenen Konzept z. B. durch die Analyse des Zustands mit pulseAT erhöht werden. Anomalien können mit pulseAT erkannt und ggf. vor dem Auftreten eines Fehlers behandelt werden. In Kombination mit dem *Health Controller* können so z. B. einzelne Komponenten neu gestartet und somit eine Selbstheilung durchgeführt werden (vgl. Kapitel 6.4.5). Durch die klare Verteilung der Funktionen und der angesprochenen Möglichkeit der Selbstheilung, kann die Systemarchitektur nachweislich zuverlässig realisiert werden. Schon der DAEbot als experimentelle Umsetzung der Architektur war äußerst zuverlässig. Hierbei ist die vorgeschlagene Systemarchitektur auch für weitere Klassen mobiler Roboter geeignet, wie z. B. für UAVs, bei denen ein zuverlässiger Betrieb aufgrund des hohen Grads an Autonomie unabdingbar ist. Analog zur Sicherheit (Safety, A3) betrachten die S4R-Mitarbeiter die Zuverlässigkeit als Fortschritt und damit großen Vorteil gegenüber dem Stand der Technik.

Die **Leistung und Effizienz (A6)** erfüllt ein System nach Balzert u. a. dadurch, dass eine angemessene Leistung für die volle Funktionalität unter den definierten Bedingungen zur Verfügung steht [18, S. 128 ff.]. Diese Leistungsfähigkeit wurde mittels des DAEbots und dem Analyse-Tool pulseAT detailliert evaluiert, indem beispielsweise das Zeitverhalten, der Leistungsbedarf und die Antwortzeiten analysiert wurden (vgl. Kapitel 7.1.4). Die Komponenten können ebenfalls aufgrund der Analyse mit pulseAT in Bezug auf die Effizienz in Bezug auf die Auslastung der Rechner optimiert werden (vgl. Optimierung der *Controller* des AMiRo, erläutert in Kapitel 7.1.8). Zudem wurde die Energieeffizienz detailliert evaluiert. Die Fähigkeit eines mobilen Roboters, die meist sehr limitierte verfügbare Energie effizient zu nutzen, ist zudem eine spezifische Teilanforderung aus der mobilen Robotik an die Leistung und Effizienz. So ist die Nutzung von SBC ein positiv evaluierter Ansatz, die Energieeffizienz zu verbessern (vgl. Kapitel 4.2.1). Zudem kann die bedarfsgesteuerte Komponentennutzung den Leistungsbedarf weiter optimierten (vgl. Kapitel 7.1.7). Dies ist u. a. möglich, da die vorgeschlagene Systemarchitektur eine klare Trennung der Funktionen vorsieht. Durch diese Trennung können Funktionen auf unterschiedliche Rechner des Systems verteilt und nicht notwendige Komponenten ausgeschaltet werden. Dennoch ist die Leistungsfähigkeit und Effizienz stark von der Komplexität des Systems abhängig. Einfache mobile Roboter, wie beispielsweise Staubsaugerroboter oder der ArEduBot [90] kommen tendenziell auch mit einem monolithischen System mit einem einzigen SBC aus. Erst bei etwas komplexeren Systemen, also typischen Robotern mit Kameras und oder anderen Umfeldsensoren, überwiegt der Vorteil eines verteilten Systems gegenüber einem monolithischen System mit einem einzigen leistungsstarken Rechner (vgl. Tabelle 4.1 in Kapitel 4.2.1). Werden komplexere Berechnungen benötigt, beispielsweise für Lernalgorithmen oder zur Optimierung des Systems, ist die vorgeschlagene Systemarchitektur durch die Integration der Cloud und die damit verbundene Rechenkapazität ohne lokalen Energiebedarf (mit Ausnahme der Cloud Verbindung) effektiv. So kann die lokale Energie mit energieeffizienten SBCs effizient genutzt und die benötige Rechenleistung in der Cloud bereitgestellt werden. Vergleichbar sind hierbei Server-Client Architekturen, wie [43], wenngleich diese im Vergleich zur vorgeschlagenen Systemarchitektur eine dauerhafte Verbindung zum Server benötigen. Insofern liegt die Bewertung zwischen vorteilhaft (bei komplexen mobilen Robotern) bis hin zu nachteilig (bei einfachen mobilen Robotern) in Bezug auf den Stand der Technik.

Die **Benutzbarkeit (A7)**, also laut Balzert die Anforderung an die Benutzer-Interaktion mit dem System, kann dadurch erhöht werden, dass Funktionen, wie die Eingabe von Benutzern und die Ausgabe von Daten, in Subsystemen vom restlichen System abgekoppelt werden [18, S. 130 ff.]. Dies soll laut Balzert vermeiden, dass beispielsweise Nutzer auf falsche Systemstrukturen zugreifen. Zudem hat die Art der Kommunikation mit dem Nutzer, z. B. die Form der Aufbereitung der Daten, laut Balzert Einfluss auf die Benutzbarkeit. Die vorgeschlagene Systemarchitektur sieht die Entkopplung der Mensch-Roboter Interaktion auf den Ebenen des *reflektorischen* und *kognitiven Operators*

vor (vgl. Kapitel 4.2.6). Die Daten können für die Nutzer*innen auf verschiedene Detailgrade abstrahiert werden. Informationen für nicht-technische Nutzer werden z. B. mit pulseAT aufbereitet und als Gesundheitszustand visualisiert (vgl. Kapitel 6.4.2). Detaillierte Informationen können z. B. mit `Grafana` betrachtet werden (vgl. Kapitel 5.3.6) oder als Rohdaten der Datenbank entnommen werden. Hierzu sind keine Kenntnisse der Software der darunterliegenden Ebene, wie z. B. der *Controller*-Ebene notwendig. Wenngleich auch Referenzsysteme, wie Spot [O13], den Nutzer*innen Informationen in Form von Visualisierungen zur Verfügung stellen, ist dies aktuell nicht Stand der Technik. Zudem geht die Aufbereitung der Daten mit pulseAT über eine Visualisierung hinaus, sodass die Benutzbarkeit als Fortschritt gegenüber dem Stand der Technik bewertet wird.

Die **Portabilität (A8)**, also laut Balzert die Anforderung, dass ein Softwareprodukt in andere Umgebungen übertragen werden kann, kann durch die Modularität eines Systems vorbereitet werden [18, S. 132 ff.]. Die vorgeschlagene Systemarchitektur ist als verteiltes System modular aufgebaut. Zudem werden die Modulation und Portabilität durch die Realisierung einer Middleware (vgl. Kapitel 5.2) erhöht bzw. erleichtert. Teile der Software der Umsetzung der vorgeschlagenen Systemarchitektur in Form des Prototyps DAEbot wurden experimentell auf den mobilen Roboter AMiRo übertragen (vgl. Kapitel 5.4). Wie bereits im Abschnitt „Softwareentwicklung“ in Kapitel 7.1.7 erläutert, war beispielsweise die Adaption des MATLAB Simulink (Stateflow) Modells nur bedingt möglich, da unter anderem die Codegenerierung aus MATLAB heraus nur über Umwege möglich ist. Das zeigt auch einen grundsätzlichen Nachteil der vorgeschlagenen Systemarchitektur in Bezug auf die Portabilität. Durch die Verteilung der Software auf verschiedene Rechner werden auf der einen Seite verschiedene Entwicklungsumgebungen benötigt und zum anderen werden typischerweise auch unterschiedliche Programmiersprachen eingesetzt. Bei einem monolithischen System hingegen, ist die gesamte Software an einem Ort und entsprechend einfacher zu portieren. Auf der anderen Seite erhöht die Modularität des Systems die Portabilität durch die erwähnte Middleware. Ein gutes Beispiel ist das modulare Framework ROS [4], bei dem die ROS Packages von Entwickler*innen bereitgestellt werden und auf diversen Robotern integriert und portiert werden können. Das ORCA Framework [44] ist ebenfalls modular aufgebaut und ist damit laut der Autoren gut auf andere Systeme portierbar. Die Begutachtung der S4R-Mitarbeiter sieht keinen klaren Vor- und Nachteil in Bezug auf die Portabilität und bewertet diese Anforderungen als gleichauf mit dem Stand der Technik.

Anforderungen aus der mobilen Robotik

Die Anforderungen aus der mobilen Robotik wurden in Kapitel 3.4.2 definiert und sind in *Fähigkeiten* (A9 bis A13) und *technische Realisierungen* (A14 bis A20) unterteilt. In Tabelle 3.3 in Kapitel 3.2.5 ist zudem dokumentiert, welches Referenzsystem welche der nachfolgend betrachteten *Fähigkeiten* adressiert. Analog zeigt Tabelle 3.4 in Kapitel 3.2.6, welche Referenzsysteme die *technischen Realisierungen* implementiert. Die Ergebnisse der Evaluation anhand der Anforderungen aus der mobilen Robotik sind in Tabelle 7.5 enthalten.

Die Fähigkeit der **Navigation (A9)** eines mobilen Roboters wird in der vorgeschlagenen Systemarchitektur auf die verschiedenen Ebenen verteilt. Mit dem Cloud-basierten *kognitiven Operator* steht dem System ausreichend Rechenleistung zur Verfügung, um auch komplexe Planungen zur Navigation umzusetzen (vgl. hybrides deliberatives/ reaktives System, u. a. in Kapitel 4.1). Andere Schichtenarchitekturen für mobile Roboter hingegen führen die Navigation nur auf einer Ebene aus (siehe Component Control Layer bei der SERA Architektur [46]). Das hat den Nachteil, dass die (Bewegungs-) Planung und Anbindung der Sensoren und Aktuatoren vermischt werden und somit Funktionen mit harten und weichen Echtzeitanforderungen nicht voneinander gekapselt sind. Um die Navigation effizient zu planen, stehen dem mobilen Roboter durch den *Perception Controller* und *Health Controller* Umweltinformationen und Informationen zum Zustand des Systems zur Verfügung. Der Bewegung des Roboters wird mit dem *Motion Controller* eine dedizierte Komponenten gewidmet, wodurch die Erfüllung der harten Echtzeitbedingungen gefördert wird. Bei monolithischen Systemen, wie dem Khepera IV [85], WolfBot [87] oder ArEduBot [90], wird durch das Teilen der verfügbaren Rechner (inkl. der Speicher) die Einhaltung der Echtzeit erschwert. Dem *Motion Controller* stehen durch die Sensorik, wie z. B Abstandsensoren (z. B. bei WMRs Ultraschallsensoren und bei UAVs eher Sensoren mit größerer Reichweite) zur Verfügung. Die Sensoren können in Abhängigkeit zur Situation mit unterschiedlichen Abtastraten konfiguriert und einzelnen Nachrichten priorisiert werden (vgl. Kapitel 5.2.1). Zudem prüft der *Motion Controller* die neuen Stellwerte intern (vgl. Abschnitt „Safety“ in Kapitel 4.2.4) bevor die Bewegungsänderung umgesetzt wird. Dieses sind allesamt Vorteile gegenüber dem Stand der Technik, wobei in der Mehrschichtenarchitektur [50] zumindest softwareseitig die Navigation und Steuerung der Roboter ausgelagert werden. Bei anderen Architekturen, Referenzsystemen und monolithischen Systemen findet diese Trennung jedoch nicht statt, sodass die S4R-Mitarbeiter die Navigation in der vorgeschlagenen Systemarchitektur als vorteilhaft gegenüber dem Stand der Technik bewerten.

Mobile Roboter haben i. d. R. einen hohen Grad an **Autonomie (A10)**. Die vorgeschlagene Systemarchitektur sieht einige Merkmale vor, welche mit der experimentellen Umsetzung des WMR DAEbots evaluiert wurden. Zunächst sei hierbei die Aufteilung in einen autonomen Klienten und einen Cloud-basierten *kognitiven Operator* anzuführen. Nach dem hybriden deliberativen/ reaktivem Paradigma der Robotik (vgl. Kapitel 3.1.1 und 4.1), bei dem die Planung des Systems auch asynchron zum reaktiven System erfolgen kann, ist der lokale Klient autonom nutzbar, selbst wenn der Cloud-basierte *kognitive Operator* kurzzeitig nicht zur Verfügung steht. Ist der *kognitive Operator* langfristig nicht verfügbar, kann jedoch die Handlungsfähigkeit des mobilen Roboters eingeschränkt sein, sodass bei Robotern ohne Zugang zur Cloud der *kognitive Operator* auch lokal implementiert werden kann. Dies ist beispielsweise bei der Klasse der UMVs notwendig. Referenzsysteme, wie der TurtleBot3 [O14] oder der OmniMan [89] sind ohne den Remote-PC nicht autonom handlungsfähig. Um die Autonomie eines mobilen Roboters zu gewährleisten, muss dieser zudem sicher, zuverlässig und effizient sein. Diese Anforderungen wurden bereits äußerst positiv evaluiert (A3, A5 und A6). Wie die praktische Evaluation mit dem DAEbot gezeigt hat, wird Autonomie zudem dadurch gestärkt, dass durch pulseAT und den *Health Controller* der Status des Systems bekannt ist (vgl. Kapitel 6.4.5), Anomalien erkannt und einzelne Komponenten zur Laufzeit neu gestartet werden können. Aus Sicht der Gutachter ist die Autonomie als vorteilhaft gegenüber dem Stand der Technik zu werten und ist auch für den autonomen Betrieb der S4R-UAVs geeignet.

Zur Realisierung der Fähigkeit der **Optimierung** und des **Lernens (A11)** bedarf es mindestens eines leistungsstarken Rechners. Die vorgeschlagene Systemarchitektur sieht hier mit dem *kognitiven Operator* eine Komponente auf einer hierarchisch übergeordneten Ebene vor. Dieser *kognitive Operator*, bzw. die Definition einer eigenen Schicht zur Optimierung des Systems ist eine Besonderheit des OCM und unterscheidet sich hier von anderen Schichtenarchitekturen wie [45, 46, 50]. Im konzeptionellen Modell ist die Realisierung des *kognitiven Operators* als Cloud Applikation eine Weiterentwicklung des *SFB-614-OCM*. Hier wird der *kognitive Operator* von mehreren mobilen Robotern verwendet und verbindet diese. Somit können Synergien genutzt werden und nicht nur der mobile Roboter selbst, sondern das Gesamtsystem aufeinander abgestimmt und optimiert werden. Für jeden Roboter müssen dazu der Zustand und die Zustands-Historie des Roboters bekannt sein. Hierzu sieht die vorgeschlagene Systemarchitektur den *Health Controller* und das Analyse-Tool pulseAT vor (vgl. Kapitel 6.4.5). Sowohl bei der experimentellen Umsetzung mit dem DAEbot, als auch bei der Adaption des konzeptionellen Modells auf den AMiRo werden u. a. die Rechenauslastung aufgrund der Ergebnisse von pulseAT optimiert, indem die Publisher-Frequenzen optimiert wurden (vgl. Kapitel 7.1.8). Die Möglichkeiten, einen mobilen Roboter zu optimieren und Methoden des Lernens einzusetzen, werden als Fortschritt und somit als großer Vorteil im Vergleich zum

Stand der Technik beurteilt, zumal die Referenzsysteme die Optimierung und das Lernen in der Mehrheit nicht umsetzen (vgl. Tabelle 3.3 in Kapitel 3.2.5).

Die Fähigkeit, dass mehrere (mobile) Roboter in einer **Multi-Roboter Kooperation (A12)** zusammenarbeiten, ist in der OCM-basierten Systemarchitektur auf mehreren Ebenen möglich (vgl. Kapitel 4.2.6). Mit dem gemeinsamen Cloud-basierten *kognitiven Operator* ist die oberste Ebene der Schichtenarchitektur prädestiniert für die Koordination (und Optimierung) von mehreren Robotern in einem Schwarm oder Multi-Roboter System. Ähnlich zu der DDX-Architektur [42] und der in [49] veröffentlichten Architektur, ist eine gemeinsame Datenbasis vorhanden. Im Gegensatz zum DDX-Ansatz ist in der vorgeschlagenen Architektur der Standort des *kognitiven Operators* jedoch unabhängig vom jeweiligen Roboter, weshalb dieser auch beim Ausfall eines der Roboter verfügbar ist. Einen ähnlichen Ansatz verfolgt erfolgreich auch [44]. Zudem kann in der vorgestellten Systemarchitektur eine Interaktion auf den anderen Ebenen, insbesondere auf Ebene des *reflektorischen Operators* stattfinden. Hierbei ist durch die strenge Trennung der Schichten gewährleistet, dass die Interaktion von Robotern, den Betrieb der *Controller* nicht beeinflussen kann. Multi-Roboter Kooperation wurde im Zuge dieser Dissertation nicht praktisch evaluiert, ist aber essenzieller Bestandteil im S4R-Forschungsprojekt. Hierbei müssen mehrere UAVs in einem UAS interagieren, beispielsweise um gemeinsam eine Karte eines Lagerhallen-Komplexes zu erstellen. Nach Begutachtung der Systemarchitektur entscheiden die S4R-Mitarbeiter, dass die vorgeschlagene Systemarchitektur auch für diesen Anwendungsfall bestens geeignet ist und im Rahmen des Projektes umgesetzt wird. Hierbei wird der *kognitive Operator* als Basisstation zur Planung und Optimierung des Gesamtsystems verwendet. Andere Ansätze nutzen ebenfalls eine Basisstation für die Koordination von mobilen Robotern [43, 44, 50]. Die (kurzfristige) Koordination der UAVs in der Luft findet über die *reflektorischen Operatoren* statt. Die vorgeschlagene Systemarchitektur ist u. a. durch die Koordination auf mehreren Ebenen gegenüber den Architekturen aus dem Stand der Technik im Vorteil.

Die Fähigkeit der **Mensch-Roboter Interaktion (A13)** ist zum Teil redundant zur Benutzbarkeit (A7), welche eine von Balzert [18, S. 130 ff.] definierte nichtfunktionale Anforderung ist. Die Benutzbarkeit wurde bereits mit $++$ bewertet, u. a. aufgrund der Aufarbeitung der Roboter-Daten und des -Zustands mit pulseAT. Die Mensch-Roboter-Interaktion als spezifische Fähigkeit von mobilen Robotern wird in der vorgeschlagenen Systemarchitektur zudem durch die Kapselung der einzelnen Schichten unterstützt. So können Nutzer-Interaktionen auf der Ebene des *reflektorischen* oder *kognitiven Operators* die Ausführung der Prozesse der *Controller* nicht beeinflussen. Das erhöht die Sicherheit (Safety) und Zuverlässigkeit von mobilen Robotern. Das ORCA Framework [44] sieht ebenfalls einen dedizierten Rechner als Schnittstelle für die Mensch-Roboter-Interaktion vor. Die Interaktion von Mensch und Maschine wurde mit der experimentellen Umsetzung des DAEbots positiv evaluiert. Im Review-Prozess werden

diverse Szenarien von unterschiedlichen Klassen diskutiert. Hierbei kommen die S4R-Mitarbeiter zu dem Ergebnis, dass die Multi-Roboter-Kooperation durch die vorgeschlagene Systemarchitektur bestens unterstützt wird und große Vorteile gegenüber dem Stand der Technik bietet.

Die Anforderung, dass mobile Roboter echtzeitfähig realisiert werden müssen, bzw. dass die Systemarchitektur das Erfüllen der **Echtzeitfähigkeit (A14)** ermöglicht, ist mit der vorgeschlagenen Systemarchitektur eindeutig gegeben. Die Einhaltung der Echtzeitfähigkeit gelingt u. a. durch die Trennung in Handlungs- und Planungsebene (vgl. Abbildung 4.4 in Kapitel 4.1) und die hardwareseitige Verteilung auf mehrere Rechner. Die *Controller* müssen harte Echtzeitfähigkeit garantieren, hier werden Echtzeit-Betriebssysteme eingesetzt und die Einhaltung der Echtzeit wird mit pulseAT überwacht. Auf der Ebene des *reflektorischen Operators* hingegen sind die Anforderungen an die Echtzeit geringer, d. h. hier wurde auch bei der Umsetzung des DAEbots ein Standard-Betriebssystem verwendet. Das vereinfacht die Implementierung und ermöglicht die Nutzung von nicht-echtzeitfähigen Tools oder Funktionen, wie z. B. eine GUI zur Mensch-Roboter Interaktion. Auf der Ebene des *kognitiven Operators* bedarf es keiner Echtzeit, sodass eine Realisierung in der Cloud möglich ist. Die Einhaltung der Echtzeitfähigkeit wird u. a. in der JAUS Architektur [53] als problematisch angesehen. Erfolgversprechend ist im Bereich Echtzeitfähigkeit zudem die Arbeit an ROS 2 [O9], bei dessen Vorgänger ROS [4] die Einhaltung der Echtzeitfähigkeit nur bedingt möglich ist. Dieses trifft ebenso auf die monolithisch aufgebauten mobilen Roboter Khepera IV [85], WolfBot [87] oder ArEduBot [90] zu. Wenngleich die Echtzeitfähigkeit Stand der Technik bei mobilen Robotern ist, ist die Realisierung der Echtzeitfähigkeit in der vorgeschlagenen Systemarchitektur ein Fortschritt zum Stand der Technik.

Zur Realisierung des **maschinellen Lernens (A15)** wird laut Sze, Chen, Emer, Suleiman und Zhang [140] vergleichsweise viel Rechenleistung benötigt. Diese stellt in der vorgeschlagenen Systemarchitektur der Cloud-basierte *kognitive Operator* bereit. Somit kann maschinelles Lernen auf der einen Seite bei der Planung eingesetzt werden (vgl. hybrides deliberatives/ reaktives System, u. a. in Kapitel 4.1). Auf der anderen Seite können Methoden des maschinellen Lernens laut Jordan und Mitchell [155] von massiv parallelen Rechnern profitieren. Die Integration eines solchen Rechners ist in der vorgeschlagenen Systemarchitektur mit dem *reflektorischen Operator+* vorgesehen (vgl. Kapitel 4.3.5). Wenngleich das maschinelle Lernen im Zuge dieser Dissertation nicht praktisch evaluiert wurde, sieht das Gutachten der S4R-Mitarbeiter die technischen Voraussetzungen zur Realisierung vom maschinellen Lernen in der vorgeschlagenen Systemarchitektur als gegeben an. Durch die Möglichkeit, die Cloud als *kognitiven Operator* oder einen FPGA, eine GPU oder ein ASIC als *reflektorischen Operator+* einzusetzen, hat die vorgeschlagene Systemarchitektur Vorteile gegenüber dem Stand der Technik, da die

Referenzsysteme das maschinelle Lernen in der Mehrheit bisher nicht umsetzen (vgl. Tabelle 3.4 in Kapitel 3.2.6).

Für die Realisierung von Algorithmen und Methoden zur **Bildverarbeitung (A16)** ist in der vorgeschlagenen Systemarchitektur mit dem *Perception Controller* eine dedizierte Komponente vorgesehen. Die Ausarbeitung dieser Komponente kann an die Anforderungen des Systems, beispielsweise in Bezug auf die Auflösung oder Bildwiederholrate, angepasst werden. Mit dem *reflektorischen Operator+* kann der mobile Roboter mit einer Komponente zur Beschleunigung der Bildverarbeitung erweitert werden (vgl. Kapitel 4.3.5). Im Vergleich zu monolithischen Systemen, wie dem Khepera IV [85], WolfBot [87] oder ArEduBot [90], ist die Bildverarbeitung in der vorgeschlagenen Systemarchitektur hardwareseitig von den anderen Rechnern des verteilten Systems entkoppelt. Insbesondere bei SBCs ist nicht die Bildwiederholrate der Kamera, sondern die Verarbeitung der Bilder der limitierende Faktor. Die Entkopplung hat den Vorteil, dass die Bildverarbeitung die gesamte verfügbare Rechenleistung beanspruchen kann, damit die Bilder so schnell wie möglich verarbeitet werden. Der *Perception Controller* des DAEbots ist beispielsweise zwischenzeitlich so stark ausgelastet, dass die mit pulseAT gemessene Antwortzeit sehr lang wird (vgl. Kapitel 7.1.4). Dieses hat jedoch keinen Einfluss auf die Handlungsfähigkeit des mobilen Roboters. Die vorgeschlagene Systemarchitektur hat somit alle Möglichkeiten, die Bildverarbeitung für mobile Roboter zu realisieren. Da die Bildverarbeitung selbst jedoch Stand der Technik und somit ein eigenes Forschungsfeld ist, werden die Möglichkeiten zur Bildverarbeitung in der vorgeschlagenen Systemarchitektur von den S4R-Mitarbeitern als vorteilhaft in Bezug auf den Stand der Technik bewertet.

Cloud Computing (A17) wird in der verteilten Systemarchitektur für mobile Roboter durch die Auslagerung des *kognitiven Operators* in die Cloud eingesetzt. Dies wird durch die OCM-Struktur ermöglicht, da hier eine strenge Trennung zwischen den Schichten implementiert ist, welche verhindert, dass die Cloud direkt auf die *Controller* zugreifen kann. Zudem sieht das OCM eine Trennung von der Handlungs- und Planungsebene und die damit verbundene Unterscheidung von Komponenten mit harten Echtzeitanforderungen (*Controller*) und Komponenten ohne diese Anforderungen an die Echtzeit vor. Die Anbindung einer Cloud ist dabei aktuell nicht prinzipiell echtzeitfähig. Im Gegensatz zu [43, 44, 50] findet die Anbindung jedoch über eine Ebene ohne harte Echtzeitanforderungen statt. Die Aufgaben sind dabei klar verteilt. Die Planung kann nach dem hybrid deliberatives/ reaktives Paradigma der Robotik in der Cloud stattfinden, während die Reaktion immer nur lokal ausgeführt wird (vgl. hybrides deliberatives/ reaktives System, u. a. in Kapitel 4.1). Wenngleich die Nutzung der Cloud in der mobilen Robotik nicht neu ist, sind die Einbindung und Aufteilung der Aufgaben in den lokalen und Cloud-basierten Bereich in der vorgeschlagenen Systemarchitektur nach dem Gutachten der S4R-Mitarbeiter ein großer Fortschritt und Vorteil gegenüber dem Stand der Technik.

In anderen Architekturen wird die Cloud vermehrt zur Koordination mit anderen Robotern oder Menschen genutzt [43, 44, 50].

Die Realisierung der **Systemüberwachung (A18)** ist eine Anforderung an die Systemarchitektur für mobile Roboter. Diese Überwachung und Analyse sind ein Kernthema dieser Dissertation. Der aktuelle Zustand der Hardware und der Umgebung des Systems wird mit einer dedizierten Komponente, dem *Health Controller* überwacht (vgl. Kapitel 4.3.2). Zudem wird im Rahmen dieser Dissertation mit dem Analyse-Tool pulseAT eine Lösung zur Systemüberwachung der vorgestellten Systemarchitektur erläutert und evaluiert (vgl. Kapitel 6 und Kapitel 7.1.2). [81, 213–216] zeigen, wie der Zustand von Robotern dargestellt werden kann, ohne selbst die technische Implementierung im Detail zu erläutern. Die Referenzsysteme bieten diese Funktionalität nicht (vgl. Tabelle 3.4 und Kapitel 3.2.6). Die Realisierung der Systemüberwachung ist somit ein klarer Vorteil der vorgeschlagenen Systemarchitektur gegenüber dem Stand der Technik.

Die Notwendigkeit und das Konzept der **Modularität (A19)** sind in Kapitel 4.2.2 erläutert. Im Vergleich zu monolithischen Systemen, ist die Modularität in Bezug auf die Hardware in verteilten Systemen höher. Die vorgeschlagene Systemarchitektur geht hier über die inhärente Modularität durch die Verteilung des Systems auf mehrere Komponenten hinaus. So steht eine Middleware zur Verfügung, welche die Einbindung neuer Rechner in das System vereinfacht (vgl. Kapitel 5.2). Zudem wird der DAEbot auch softwareseitig abstrahiert, sodass u. a. im Falle des *Motion Controllers* durch die PDAL-Abstraktionsschicht die Einbindung bzw. der Austausch von verschiedenen Betriebssystemen vereinfacht wurde (vgl. Kapitel 5.3.1). Ähnlich wird die Modularität u. a. in ROS [4] umgesetzt. Hier werden die Schnittstellen zur Kommunikation stark abstrahiert, sodass sich die Entwickler*innen praktisch selbst kaum um die Integration von ROS Packages kümmern müssen. Der mobile Roboter AMiRo [86] ist ebenfalls als verteiltes System mit abstrahierten Schichten in der Architektur [40] sehr modular aufgebaut. Andere Referenzsysteme, wie z. B. der OmniMan [89] bieten das in dieser Form nicht, sodass die Bewertung der Modularität in der vorgestellten Systemarchitektur als Vorteil im Vergleich zum Stand der Technik bewertet wird.

Die **modellbasierte Entwicklung (A20)** wird bei der experimentellen Umsetzung der vorgeschlagenen Systemarchitektur mit dem DAEbot bei der Entwicklung der Applikation zur Steuerung des Roboters auf dem *reflektorischen Operator* angewendet. Hierzu wird, vergleichbar zum ArEduBot [90], eine eigene MATLAB Simulink Toolbox entwickelt (vgl. Kapitel 5.2.3). Mit der Simulink Erweiterung Stateflow wird die ereignisgesteuerte modellbasierte Entwickelung des *reflektorischen Operators* auf Basis von FSMs ermöglicht. Des Weiteren hat die vorgeschlagene Systemarchitektur den Vorteil, dass die hierarchische Dekomposition ermöglicht, dass die *Controller* im Modell abstrahiert werden können. Die Entwickler*innen des *reflektorischen Operators* können die *Controller* konfigurieren, ohne die

Hard- und Software der *Controller* verstehen zu müssen. Zudem ist auch die modellbasierte Entwicklung auf anderen Ebenen, insbesondere den *Controllern* möglich. Da die *Controller* u. a. Methoden der Regelungstechnik verwenden, kann hier MATLAB Simulink eingesetzt werden. Durch die Verteilung der Software auf verschiedene Rechner ergibt sich jedoch der Nachteil, dass kein ganzheitliches Modell in einem einzigen Tool abbildbar ist, wie es beispielsweise bei monolithischen Systemen möglich ist. Die Referenzsysteme adressieren die modellbasierte Entwicklung mit Ausnahme des ArEduBots [90] nicht (vgl. Tabelle 3.4 in Kapitel 3.2.6). Da die modellbasierte Entwicklung jedoch prinzipiell in nahezu allen Architekturmustern möglich ist, bewerten die S4R-Mitarbeiter dieses Merkmal als gleichauf mit dem Stand der Technik.

7.2.4 Herausforderungen

Die Herausforderungen an die Systemarchitektur von mobilen Robotern beschreiben sowohl eine *Fähigkeit* (H1), als auch *technische Realisierungen* (H2 bis H6) und wurden ebenfalls in Kapitel 3.4.3 definiert. Nachfolgend wird die Systemarchitektur anhand dieser *Fähigkeit* und den *technischen Realisierungen* evaluiert. Die *Fähigkeiten* und *technischen Realisierungen* der Referenzsysteme sind in den Tabellen 3.3 und 3.4 in Kapitel 3.2.5 und 3.2.6 dokumentiert. Die Zusammenfassung der Evaluation der Systemarchitektur ist in Tabelle 7.5 enthalten.

Wie bereits in der Anforderung an die Zuverlässigkeit (A5) erläutert, ist die **Selbstheilung (H1)** nur möglich, wenn der Zustand des Systems bekannt ist. Die vorgeschlagene Systemarchitektur sieht hierfür den *Health Controller* (vgl. Kapitel 4.3.2) und Analyse-Tool pulseAT vor. Bei der experimentellen Umsetzung besitzt der *Health Controller* ein Relais-Board, mit denen einzelne Komponenten ein- und wieder ausgeschaltet werden können. Wird ein Fehler festgestellt (z. B. durch pulseAT), kann laut [225] das System geheilt werden, indem die Komponente neugestartet wird. Wenngleich dieser Neustart prinzipiell in jedem verteilten System möglich ist, ermöglicht die verteilte Struktur des OCM dieses in der vorgeschlagenen Systemarchitektur erst. Durch den hierarchischen Ansatz ist es möglich, dass die höheren Ebenen die darunter liegenden Ebenen überwachen. Da die Informationen der *Controller* im *reflektorischen Operator* zusammen kommen, kann dieser beispielsweise bei Problemen mit dem *Perception Controller* in einen Notfallmodus wechseln. Im DAEbot wurde das so implementiert, dass dann der *Motion Controller* die Motoren des mobilen Roboters stoppt und erst weiter fahren kann, wenn das Problem durch einen Neustart des *Perception Controllers* behoben wurde. Zudem ist es möglich, dass der *reflektorische Operator* die *Controller* rekonfiguriert. Ist beispielsweise ein Ultraschallsensor defekt, so wird der Publisher des Sensors ausgeschaltet und die Tiefenkamera aktiviert, um das Erkennen von Hindernissen weiterhin zu gewährleisten. Diese oder andere Methoden zur Selbstheilung werden von keinem der Referenzsysteme adressiert (vgl. Tabelle 3.3 in Kapitel 3.2.5). Die Möglichkeit

zur Selbstheilung in der vorgestellten Systemarchitektur wird demnach als Fortschritt im Vergleich zum Stand der Technik bewertet.

Die vorgeschlagene Systemarchitektur für mobile Roboter kann prinzipiell durch **Redundanzen (H2)** erweitert werden. Hierbei ist die Integration von (Hardware-) Redundanzen im sicherheitskritischen Bereich der *Controller*, aufgrund des in der vorgeschlagenen Systemarchitektur vorgesehenen Bus-Systems, im reflektorischen Kreis (vgl. Kapitel 4.4.2) möglich. Hierbei könnten redundante Komponenten, wie z. B. der *Perception Controller* parallel ausgeführt werden. In diesem Fall könnte der *reflektorische Operator* prüfen, ob die Ergebnisse von beiden *Perception Controllern* gleich sind. Weichen die Ergebnisse voneinander ab, so könnte der *reflektorische Operator* prüfen, welcher der beiden *Perception Controller* valide Ergebnisse liefert. Eine redundante Ausführung des *reflektorischen Operators* ist prinzipiell ebenfalls möglich. Eine parallele Ausführung von redundanten *reflektorischen Operatoren* ist jedoch nicht sinnvoll, da die Nachrichten des *reflektorischen Operators* eindeutig sein sollten. Zudem ist die Integration eines redundanten *reflektorischen Operator* nicht so einfach möglich, wie bei den via Daten-Bus eingebundenen *Controllern*, da der *reflektorische Operator* eine direkte (PPP-) Verbindung zum *kognitivem Operator* aufbaut (vgl. Kapitel 4.4.4). Die Evaluation mit dem DAEbot hat gezeigt, dass die Sicherheit (Safety) und Zuverlässigkeit auch ohne Redundanzen gegeben sind. Somit wurde die Realisierung von Redundanzen nicht praktisch evaluiert. Die vorgestellte Systemarchitektur kann demnach prinzipiell zumindest auf der Ebene der *Controller* redundant implementiert werden. Auf den höheren Ebenen ist das mit vergleichsweise viel Aufwand verbunden. Die S4R-Mitarbeiter bewerten die Redundanz als vorteilhaft im Vergleich zum Stand der Technik.

Eines der Schlüsselmerkmale der vorgeschlagenen Systemarchitektur ist die **Rekonfigurierbarkeit (H3)** der *Controller* durch den *reflektorischen Operator*. Die *Controller* führen zwar ihre Aufgaben selbstständig aus, werden aber durch den *reflektorischen Operator* an die aktuelle Situation angepasst. Diese Rekonfiguration kann mitunter sehr dynamisch sein, wie beispielsweise die adaptive CAN Publisher Transferrate und Priorität in Abbildung 5.7 in Kapitel 5.2.1 zeigen. Hier wird der nach vorne blickende Abstandssensor viel öfter abgetastet, als die zur Seite und nach hinten blickenden Abstandssensoren. Die Rekonfigurierbarkeit der *Controller* wurden mit dem DAEbot und dem AMiRo evaluiert. Diese Art der dynamischen Rekonfiguration ist in keinem der Referenzsysteme implementiert (vgl. Tabelle 3.4 in Kapitel 3.2.6), sodass Realisierung in der vorgeschlagenen Systemarchitektur als Fortschritt bewertet wird.

Die OCM Struktur ist in der vorgeschlagenen Systemarchitektur als **verteiltes System (H4)** realisiert. Die Vorteile der Verteilung der Aufgaben auf verschiedene Komponenten, welche jeweils über einen dedizierten Rechner verfügen, wurden bereits mehrfach angeführt. Unter anderem unterstützt die Verteilung die Entkopplung der einzelnen Ebenen und die Einhaltung der Echtzeitfähigkeit an den notwendigen Stellen. Einen mobilen Roboter als verteiltes System zu realisieren, ist hierbei nicht neu, so ist beispielsweise der AMiRo sehr ähnlich aufgebaut. Die mobilen Roboter TurtleBot3 [O14], e-puck2 [88] und *Savvy* [91] bestehen ebenfalls aus mehreren Recheneinheiten. Die Kombination eines lokal verteilten Systems mit einer Cloud-Komponente hingegen, ist in dieser Form in keinem der Referenzsysteme oder den vorgestellten Architekturen vorzufinden. Die S4R-Mitarbeiter sehen die Kombination aus lokalen verteilten System mit Cloud Komponente als vorteilhaft im Vergleich zum Stand der Technik an.

Die Realisierung von **Edge Computing (H5)** unterstützt die vorgeschlagene Systemarchitektur auf mehreren Ebenen. Sowohl im motorischen, als auch reflektorischen Kreis werden die Daten vor dem Versenden vorverarbeitet (vgl. Kapitel 4.4). Die *Controller* führen beispielsweise Umrechnungen durch, bevor die Daten via CAN an die nächsthöhere Ebene versendet werden. Hauptsächlich jedoch findet das Edge Computing auf der Ebene des *reflektorischen Operators* statt. Dieser sammelt und verarbeitet die Informationen aller *Controller* im System. Die hierarchisch übergeordnete Ebene des *kognitiven Operators* benötigt jedoch nicht alle Informationen der *Controller*. Der *reflektorische Operator* als Schnittstelle zum Cloud-basierten *kognitiven Operator* filtert unnötige Daten und sendet typischerweise nur die Daten in die Cloud, welche zur Planung und Optimierung des Systems benötigt werden (vgl. Kapitel 4.4.4). Während das Edge Computing bei den Referenzsystemen nicht adressiert wird (vgl. Tabelle 3.4 in Kapitel 3.2.6), gibt es in der Literatur Ansätze, wie z. B. [127], die über die Realisierung in der vorgeschlagenen Systemarchitektur hinausgehen. Die Realisierung des Edge Computing wird als vorteilhaft gegenüber dem Stand der Technik bewertet.

Die vorgeschlagene Systemarchitektur sieht den Einsatz von **Einplatinenrechnern (H6)** vor (vgl. Kapitel 4.2.1). Der Einsatz von SBC bei der Realisierung von mobilen Robotern ist nicht unüblich. So nutzen beispielsweise auch der ArEduBot [90] und der *Arduino Robot* [92] je einen Arduino SBC als zentrale Steuereinheit, wenngleich die Leistungsfähigkeit mit diesen Rechnern begrenzt ist. Die Realisierung als verteiltes System in der vorgeschlagenen Systemarchitektur und insbesondere die Integration der Cloud ermöglicht es jedoch, dass auch komplexe Roboter lokal mit der Rechenleistung von mehreren kombinierten SBCs auskommen. Nach dem hybrid deliberativem/ reaktivem Paradigma der Robotik, können weitergehende Planungen asynchron mittels des *kognitiven Operators* erfolgen (vgl. hybrides deliberatives/ reaktives System, u. a. in Kapitel 4.1). Während der lokal ausgeführte reaktive Teil des Systems (Handlungsebene) vergleichsweise wenig Rechenleistung benötigt und auf SBCs ausgeführt werden kann, finden aufwendige Berechnungen der Planungsebene in

der Cloud statt. Die Evaluation in Kapitel 7.1.4 bestätigt, dass die Auswahl von SBCs als lokale Rechner sinnvoll war. Im Vergleich zu den erwähnten Referenzsystemen bei denen SBC eingesetzt werden, zeigt die praktische Evaluation, dass auch die Realisierung von komplexen Robotern wie der DAEbot mit SBCs möglich und sinnvoll ist. Die vorgestellte Systemarchitektur ist hierbei für den Einsatz von energieeffizienten SBCs geeignet und nutzt die Möglichkeiten der SBC durch Methoden wie die bedarfsgesteuerte Komponentennutzung (vgl. Kapitel 4.2.5 und Kapitel 7.1.7) aus. Die S4R-Mitarbeiter bewerten die Nutzung von SBCs in der vorgeschlagenen Systemarchitektur als Fortschritt im Vergleich zum Stand der Technik.

7.3 Abschließende Bewertung der Systemarchitektur

Abschließend werden die Vor- und Nachteile der in Kapitel 4 vorgestellten Systemarchitektur für mobile Roboter zusammengefasst und bewertet.

7.3.1 Verteilte Systemarchitektur für mobile Roboter

Die experimentelle Umsetzung der vorgeschlagenen Systemarchitektur mit der Entwicklung des DAEbots und die Evaluation haben gezeigt, dass die Nutzung einer verteilten Systemarchitektur innerhalb eines Roboters sinnvoll ist. Die Nutzung von energieeffizienten Rechnern, wie z. B. SBCs kann die Energieeffizienz eines mobilen Roboters verbessern. Durch die Kombination von mehreren dieser Rechner steht für die sichere und autonome Handlung des Roboters ausreichend Rechenkapazität zur Verfügung. Eine verteilte Architektur erleichtert zudem die Umsetzung eines modularen Ansatzes. Neue Komponenten können vergleichsweise einfach integriert werden, da diese nahezu unabhängig agieren und die Integration fast ausschließlich über die Schnittstellen der Kommunikation erfolgt. Die Verteilung der Software erhöht zudem die Sicherheit (Safety), u. a. durch die Entkopplung der Rechner voneinander. Diese ermöglicht, dass z. B. Funktionen mit harten Echtzeitanforderungen, wie die Steuerung der Motoren, nicht durch weniger zeitkritische Anforderungen, wie die Mensch-Roboter Interaktion, beeinflusst werden. Zudem ist ein mobiler Roboter mit mehreren Rechnern nicht von einer Recheneinheit abhängig. So kann beispielsweise der *Motion Controller* des DAEbots die Motoren selbstständig stoppen, wenn ein Hindernis erkannt wird oder der *reflektorische Operator* nicht zur Verfügung steht. Sollte der *Motion Controller* ausfallen, so kann der *reflektorische Operator* den *Health Controller* anweisen, die Relais der Motortreiber auszuschalten, welches ebenfalls zum Stoppen des Roboters führt. Der erhöhte Implementierungsaufwand eines verteilten Systems ist hingegen ein Nachteil gegenüber einem monolithischen System. Zudem kann es insbesondere bei mobilen Robotern der Klasse der Micro/ Nano Roboter aus Platzgründen unmöglich sein, mehrere Rechner zu integrieren.

7.3.2 OCM Struktur für mobile Roboter

Die OCM als strukturgebendes Muster für die Systemarchitektur von mobilen Robotern wurde mit Demonstratoren positiv evaluiert und erfüllt die Anforderungen und Herausforderungen an die Systemarchitektur von mobilen Robotern. Die klare hierarchische Trennung zwischen *Controllern* und *Operatoren* erlaubt die Trennung zwischen Funktionen mit harter und weicher Echtzeit. Diese Trennung ermöglicht den Einsatz von Komfort-Funktionen, wie die Darstellung des aktuellen Zustands des Roboters auf der Planungsebene, ohne dass die Handlungsebene dadurch beeinflusst wird. Zudem erlaubte diese Trennung, dass für die *Operatoren* leicht zu implementierende Standard-Betriebssysteme und Tools verwendet werden können, während die Implementierung der *Controller* auf Basis von Echtzeit-Betriebssystemen erfolgen muss. Die klare Trennung der Schichten im OCM erhöht zudem die Sicherheit (Safety). So ist es u. a. nicht vorgesehen, dass der *reflektorische Operator* direkt Stellwerte von Aktuatoren verändert. Diese werden als Konfigurationen an die *Controller* gesendet, welche lokal diese Konfiguration erst nach einer internen Prüfung umsetzen. Die Multi-Roboter-Kooperation und die Mensch-Roboter Interaktion finden hauptsächlich auf den oberen Schichten des OCM statt. Die strenge Trennung zwischen den Schichten wird in der vorgeschlagenen Systemarchitektur nochmals durch die klare Definition von typischen *Controllern* von mobilen Robotern gestärkt. So wird bei einem mobilen Roboter stets ein *Motion Controller* benötigt, meistens auch ein *Perception Controller*. Diese Definition der typischen *Controller*, wie auch die Umsetzung der Schichten des *reflektorischen* und *kognitiven Operators*, ist allgemeingültig für mobile Roboter. Durch die Kapselung der Funktionen auf verschiedene *Controller* kann die Energieeffizienz des Roboters nochmals durch das Ausschalten von nicht benötigten Komponenten im Sinne der vorgestellten bedarfsgesteuerten Komponentennutzung verbessert werden.

7.3.3 Integration von Cloud Computing in das OCM

Die Integration der Cloud in Form des *kognitiven Operators* in die OCM-Architektur ist eine der Erweiterungen des *SFB-614-OCM*. Diese Erweiterung wird in den Experimenten und der Evaluation positiv bewertet. Zwar erhöht die Integration der Cloud den Implementierungsaufwand, die Vorteile überwiegen diesen Aufwand jedoch. Durch die Integration der Cloud stehen dem lokalen Klienten zumindest zeitweise eine große Rechenkapazität zur Verfügung, beispielsweise für das Durchführen von Methoden zur Optimierung und dem maschinellen Lernen. Da diese Rechenkapazität nicht auf dem lokalen Klienten bereitgestellt werden muss, kann der lokale Klient energieeffiziente Rechner mit weniger Rechenkapazität einsetzen. Möglich ist dies, da die Planungen nach dem hybrid deliberativen/ reaktiven Paradigma der Robotik, asynchron zur Reaktion des Roboters erfolgen können. Zur Reduzierung der Datenmenge wird Edge Computing eingesetzt, indem der *reflektorische Operator* bereits die erste Datenverarbeitung übernimmt und die zu versendende Datenmenge in die Cloud reduziert. Bei den Experimenten war zudem das Speichern von Sensor- und Aktuatordaten in der Cloud ein großer Vorteil bei der Analyse

des Systems. Zusammen mit der Analyse des Systemzustands mit pulseAT, konnte eine detaillierte Analyse eines Experiments im Nachgang mit den Daten aus der Cloud erfolgen. Die Möglichkeit, mehrere Roboter anzubinden und zu koordinieren, ist ein weiterer Vorteil des Cloud-basierten Ansatzes. Neben dem hohen Aufwand durch die Implementierung einer weiteren Schnittstelle, bedarf es auch für die Absicherung dieser Schnittstelle eines erhöhten Aufwands. Insbesondere wenn die Kommunikation über das öffentliche Funknetz erfolgt, muss die Schnittstelle besonders gut abgesichert werden, da diese eine mögliche Schwachstelle der Sicherheit (Security) darstellt.

8 Zusammenfassung und Ausblick

Mobile Roboter sind komplexe technische Systeme, deren Entwicklung aufwendig ist und interdisziplinär erfolgt. Die Auswahl und Definition einer geeigneten Systemarchitektur sind ausschlaggebend für die erfolgreiche Entwicklung eines solch komplexen Systems. Hierbei gibt es in der mobilen Robotik diverse Ansätze zur Umsetzung der Architektur. Ziel dieser Dissertation ist es, eine allgemeingültige Systemarchitektur für einen typischen mobilen Roboter zu definieren und zu evaluieren. Diese allgemeingültige Systemarchitektur sollte hierbei nicht nur die grobe Struktur, sondern eine Definition der Komponenten, Schnittstellen und weiterer Schlüsselmerkmale im Detail enthalten.

Der Stand der Technik bezüglich Architekturen für mobile Roboter umfasst sowohl monolithische Systeme, d. h. Systeme, die von einem einzigen Rechner gesteuert werden, als auch Systeme aus mehreren Rechnern. Hierbei werden insbesondere Systeme in Form einer Server-Client Architektur eingesetzt, wobei der Server i. d. R. die Methoden zur Planung des Systems übernimmt. Nach dem hybrid deliberativen/ reaktiven Paradigma der Robotik, kann die Reaktion des Systems von der Planung (der Kognition) des Systems entkoppelt werden, sodass die Planung weniger häufig und asynchron zur Reaktion des Systems erfolgen kann. Die Softwarearchitektur innerhalb eines Rechners eines monolithischen Systems setzt teilweise mehrere Schichten im sogenannten Mehrschicht-Muster ein. Wenige mobile Roboter setzen diese Mehrschichten-Architektur als verteiltes System um. Das OCM ist ein solches Muster aus mehreren Schichten. Der OCM Ansatz wurde bisher vorrangig in großen mechatronischen Systemen, wie z. B. bei einem Forschungsprojekt zur Entwicklung eines Systems mit Schienenfahrzeugen verwendet. Die grundlegende Struktur des OCM scheint jedoch auch für mobile Roboter geeignet, sodass dieser Ansatz für die Entwicklung einer Systemarchitektur für mobile Roboter übernommen wurde.

Um die Allgemeingültigkeit einer Systemarchitektur zu prüfen, wurde mit einer systematischen Literaturrecherche evaluiert, welche Merkmale aktuell einen typischen mobilen Roboter auszeichnen. Hierzu wurden sowohl Referenzsysteme, als auch typische Fähigkeiten und technische Realisierungen von mobilen Robotern anhand von populären Suchbegriffen in der Literatur betrachtet. Als Ergebnis wurde eine Taxonomie für mobile Roboter erstellt, welche typische Klassen, Anwendungsgebiete, Fähigkeiten und technische Realisierungen definiert. Anhand dieser Taxonomie, sowie allgemeinen Anforderungen an

Architekturen für technische Systeme, wurden die Anforderungen und Herausforderungen an eine Systemarchitektur für einen archetypischen mobilen Roboter erarbeitet.

Nach der Spezifikation der Anforderungen an die Systemarchitektur für mobile Roboter, wurde die Systemarchitektur in Form eines konzeptionellen Modells definiert. Das konzeptionelle Modell der Systemarchitektur basiert auf der OCM Struktur, deren Umsetzung als verteiltes System vorgesehen ist. Das OCM besteht aus drei hierarchisch angeordneten Schichten, den *Controllern*, dem *reflektorischen Operator* und dem *kognitiven Operator*. Die *Controller* sind hierbei die unterste Ebene des OCM und haben direkten Kontakt zum physischen System, d. h. zu den Sensoren und Aktuatoren. Da (mobile) Roboter sicherheitskritische Systeme sind, ist es notwendig, dass z. B. Motoren rechtzeitig vor Hindernissen stoppen. Auf Ebene der *Controller* ist somit das Einhalten von harten Echtzeitanforderungen unbedingt erforderlich. Der *reflektorische Operator* sammelt alle Informationen der *Controller* und wertet diese aus. Aus dem Ergebnis dieser Auswertung werden die *Controller* konfiguriert, beispielsweise indem neue Sollwerte für die Motoren übertragen werden. Der *kognitive Operator* ist, hierarchisch betrachtet, die oberste Komponente des OCM und wird u. a. zur Optimierung des Roboters eingesetzt. Da hierzu eine hohe Rechenkapazität erforderlich ist, wird der *kognitive Operator* in der vorgeschlagenen Systemarchitektur als cloudbasierte Komponente umgesetzt. Da die Leistungsanforderungen der *Controller* und des *reflektorischen Operators* begrenzt sind und die lokale Energieeffizienz eine der Anforderungen an einen mobilen Roboter ist, wurden hier SBC als energieeffiziente Recheneinheiten eingesetzt. Neben dieser grundlegenden Architektur enthält das konzeptionelle Modell weitere Details zur Realisierung der Architektur. Hierzu wurden beispielsweise die Schnittstellen des Systems definiert, indem detailliert verschiedene Varianten diskutiert wurden. Weiterhin werden Funktionen erläutert, welche die Sicherheit (Safety und Security), die Zuverlässigkeit, die Energieeffizienz oder die Modularität des Roboters erhöhen oder die Multi-Roboter Kooperation oder die Mensch-Roboter Interaktion ermöglichen. Zudem wurden Systemkomponenten für mobile Roboter definiert, welche als *Motion*, *Health* und *Perception Controller* bezeichnet werden. Mit dem *Health Controller* wurde ein neuer *Controller* eingeführt, welcher den aktuellen Zustand des Systems messen kann und zudem in der Lage ist, einzelne Rechner des verteilten Systems ein- und auszuschalten.

Nach der Definition des konzeptionellen Modells erfolgt die experimentelle Umsetzung der Systemarchitektur. Diese Umsetzung sollte sicherstellen, dass die vorgeschlagene Systemarchitektur sowohl für die Entwicklung, als auch für den Betrieb effektiv und effizient ist. Um bei der Umsetzung frei von Vorgaben, wie z. B. in Bezug auf die Auswahl der Rechner, zu sein, wird kein existierender mobiler Roboter und keine Roboter-Plattform verwendet. Es wird ein Prototyp, der DAEbot entwickelt, welcher zudem als Entwicklungs-Framework für mobile und verteilte Roboter eingesetzt wird. Unterstützung erhielt der Autor in Form von einigen studentischen Arbeiten im Rahmen der Entwicklung

des DAEbots. Der DAEbot besteht aus fünf SBCs, wie beispielsweise zwei Raspberry Pis, einem STM32F3 Discovery und einem Arduino Mega 2560. Für diese SBCs wurden im Rahmen dieser Dissertation diverse Tools, Applikationen, Dienste und Betriebssysteme entwickelt oder konfiguriert. Zudem wurde eine Middleware entwickelt, die u. a. eine Toolbox zur modellbasierten Entwicklung mit MATLAB Simulink (Stateflow), eine Erweiterung der CAN-Schnittstelle, Watchdogs zur Überwachung der sicherheitskritischen Funktionen und die Einbindung der Cloud Dienste (u. a. den TICK Stack) enthält. Um zu prüfen, ob das Konzept allgemeingültig ist, wurde die Systemarchitektur auch auf den mobilen Roboter AMiRo übertragen und wird aktuell auf den UAVs des Forschungsprojektes S4R implementiert.

Zur Unterstützung des Hardware-Auswahl-Prozesses und zur Überprüfung des Systemzustands der Rechner in einem verteilten System, wurde das Analyse-Tool pulseAT entwickelt. Ein solches Analyse-Tool wird bei der Umsetzung des DAEbots und insbesondere bei der Evaluation des DAEbots benötigt. Das Analyse-Tool pulseAT wertet in einem dreistufigen Konzept die Daten zur Systemauslastung aller Rechner im verteilten System aus. Ein *pulseAT Agent* sammelt auf jedem Rechner die aktuellen Daten zur Systemauslastung. Hierzu werden auch *pulseAT Agenten* für die SBC erstellt, deren Daten zur Systemauslastung typischerweise nicht verfügbar sind. Zudem kann mit pulseAT geprüft werden, ob Deadlines von Funktionen mit Echtzeitanforderungen eingehalten werden. Im System existiert ein einziger *pulseAT Manager*, welcher alle Daten der *pulseAT Agenten* einsammelt. Zudem fragt der *pulseAT Manager* die Daten an und berechnet aus dem zeitlichen Versatz der Antwort, die Antwortzeit für jeden Rechner. Der *pulseAT Manager* wertet den Ist-Zustand des Systems aus. Zudem sendet der *pulseAT Manager* die Daten zur weiteren Analyse an einen cloudbasierten *pulseAT Analyzer*. Der *pulseAT Analyzer* hat Zugriff auf die Historie der Daten, kann diese in einer Langzeitanalyse betrachten und die Ergebnisse visualisieren. Zusammen mit den Daten des *Health Controllers* stehen somit wichtige Informationen zum Zustand der Hardware und der Software des DAEbots zur Verfügung.

Die Eignung der vorgeschlagenen Systemarchitektur wurde mit der Umsetzung des DAEbots und der teilweisen Adaption an die Software des AMiRos nachgewiesen. In einer Evaluation wurden die Ergebnisse des konzeptionellen Modells und dessen Implementierung genauer betrachtet. Diese Evaluation bestätigte u. a. die Definition der Komponenten und die Auswahl der Rechner und der Schnittstellen. Ebenfalls evaluiert werden einige der Schlüsselmerkmale, wie z. B. die bedarfsgesteuerte Komponentennutzung. Zudem wurde die Systemarchitektur anhand der zu Beginn definierten Anforderungen und Herausforderungen evaluiert. Abschließend wurde die Systemarchitektur bewertet. Hierbei wurde festgestellt, dass sowohl die Realisierung eines

verteilten Systems, als auch die OCM Struktur und die Einbindung der Cloud für mobile Roboter sehr gut geeignet sind.

Zu Beginn dieser Arbeit wurden die Kernaussage (vgl. Kapitel 1.3) und Hypothesen (vgl. Kapitel 1.4) in Form eines Hypothesenbaums (vgl. Abbildung 1.9) definiert. Tabelle 8.1 zeigt in welchem Abschnitt dieser Dissertation die Hypothesen u. a. diskutiert und nachweislich verifiziert wurden. Da alle Teilhypothesen (siehe Hyp 1.1 bis Hyp 3.2 in Abbildung 1.9) verifiziert wurden, sind auch die drei Hypothesen (Hyp 1, 2 und 3) haltbar. Die Kernaussage (K) wird somit ebenfalls als verifiziert betrachtet.

#	**Kernaussage/ Hypothese**	**Verifiziert/ Diskutiert in**
K	Die Anforderungen an eine allgemeingültige Architektur für mobile Roboter werden durch eine dauerhaft überwachte und hierarchisch verteilte Systemarchitektur erfüllt.	*Hyp 1, 2 und 3*
Hyp 1	Die Anforderungen an die Systemarchitektur sind auf typische mobile Roboter übertragbar.	*Hyp 1.1 und 1.2*
Hyp 1.1	Ein archetypischer mobiler Roboter kann mittels einer Taxonomie definiert werden.	Kapitel 3.3
Hyp 1.2	Die Anforderungen an eine Systemarchitektur können aus einer Taxonomie abgeleitet werden.	Kapitel 3.4
Hyp 2	Ein hierarchisch verteiltes System erfüllt die Anforderungen an die Systemarchitektur von mobilen Robotern.	*Hyp 2.1 und 2.2*
Hyp 2.1	Die Verteilung des Systems auf mehrere Rechner ist für mobile Roboter besser geeignet als monolithische Strukturen.	Kapitel 4.2 und 7.3.1
Hyp 2.2	Die Anforderungen an die Architektur mobiler Roboter werden durch eine verteilte Schichtenarchitektur erfüllt.	Kapitel 7.2 und 7.3.2
Hyp 3	Bei mobilen Robotern ist eine dauerhafte Analyse des Systemzustands notwendig.	*Hyp 3.1 und 3.2*
Hyp 3.1	Mobile Roboter sind sicherheitskritische Systeme.	Kapitel 6.1
Hyp 3.2	Eine dauerhafte Analyse des Systemzustands ist bei sicherheitskritischen Systemen notwendig.	Kapitel 6.2 und 6.4.5

Tabelle 8.1: Evaluation der Kernaussage und Hypothesen

Durch die Entwicklung des DAEbots wurde ein Entwicklungs-Framework realisiert, welches für weitere Aufgaben im Bereich Systemarchitekturen und der mobilen Robotik eingesetzt werden kann. So steht die Infrastruktur sowohl auf dem DAEbot, als auch in der Cloud bereit, um die im Rahmen dieser Dissertation gezeigten Forschungen weiterzuführen. Hervorzuheben ist hierbei die Umsetzung der Multi-Roboter Kooperation, welche im Rahmen des S4R-Forschungsprojektes realisiert wird. Durch die Verfügbarkeit des DAEbots, des AMiRos und der benötigten Infrastruktur kann hier direkt an das konzeptionellen Modell und dessen experimentellen Umsetzung angeknüpft werden. Offen sind zudem die bisher nicht praktisch evaluierten Anforderungen und Herausforderungen an die vorgeschlagene Systemarchitektur in praxisnahen Experimenten (siehe Tabelle 7.5 in Kapitel 7.2).

Mit der Infrastruktur rund um den DAEbot wurde zudem die Grundlage für die Forschung an aktuellen Themen der mobilen Robotik geschaffen. Hier sei beispielsweise die Umsetzung des maschinellen Lernens auf dem *kognitiven Operator* zu nennen. Im nächsten Schritt könnte zudem die Integration von ROS/ ROS 2 in den *reflektorischen* und oder *kognitiven Operator* erfolgen. Dieses würde die Arbeit mit dem DAEbot voraussichtlich weiter vereinfachen, da fertige ROS Packages, z. B. zur Navigation des Roboters verwendet werden könnten.

Grundsätzlich kann der DAEbot auch zur Evaluation von weiteren Rechnern verwendet werden. Die Energieeffizienz des DAEbots kann zudem durch eine Weiterentwicklung des *Health Controllers* stark verbessert werden. Weiterhin gilt es, das Konzept des *reflektorischen Operators+* zu evaluieren. Begrüßenswert wäre es zudem, wenn die vorgestellte Systemarchitektur für weitere Systeme, außer dem DAEbot, AMiRo und den im Rahmen des Forschungsprojektes S4R entstehenden UAVs, umgesetzt würde. Hierbei ist auch eine Adaption der Systemarchitektur in anderen Domänen, wie z. B. im Bereich Automotive wünschenswert.

Das Analyse-Tool pulseAT ist grundsätzlich einsatzbereit, kann jedoch noch erweitert werden. Hierbei sind sowohl die Adaption des *pulseAT Agenten* für weitere SBC sinnvoll, als auch die Erweiterung der Funktionalität des *pulseAT Analyzers*. Bestenfalls entstehen eine Reihe von Erweiterungen des *pulseAT Analyzers*, welche je nach Bedarf integriert werden können. Um diese Erweiterungen voranzutreiben, ist eine Veröffentlichung des Analyse-Tools als Open-Source Projekt angedacht.

Abkürzungsverzeichnis

AMiRo Autonomous Mini Robot
AMQP Advanced Message Queuing Protocol
AMR Autonomous Mobile Robot
ANR Autonomous Networked Robots
API Application Programming Interface
ASIC Application-Specific Integrated Circuit
ATAM Architecture Tradeoff Analysis Method
CAN Controler Area Network
CITEC Center for Cognitive Interaction Technology
CPS Cyber-Physical Systems
CPU Central Processing Unit
CSI Camera Serial Interface
CV Computer Vision
DAEbot Distributed Architecture Evaluation Robot
DSP Digitaler Signal Prozessor
EDA Event-driven Architecture
EE Execution Environment
FPGA Field Programmable Gate Array
FR Functional Requirement
FSM Finite State Machine
GPIO General Purpose Input/ Output
GPU Graphics Processing Unit
GSM Global System for Mobile Communications
GUI Graphical User Interface
H2M Human to Machine
HAL Hardware Astraction Layer
HiL Hardware-in-the-Loop
HMI Human Machine Interface
HMP Heterogeneous Multiprocessing
IaaS Infrastructure as a Service
ID Identifikator
IDiAL Institut für die Digitalisierung von Arbeits- und Lebenswelten
ILP InfluxDB Line Protocol
IMU Inertial Measurement Unit
IoT Internet of Things
ISA Instruction Set Architecture
KI Künstliche Intelligenz
KNN Künstliche Neuronale Netze
LAN Local Area Network
LE Low Energy
LiDAR Light Detection and Ranging
LoRa Long Range
LTE Long Term Evolution
LUT Lookup-Tabellen
M2H Machine to Human
M2M Machine to Machine
MBSE Model-based Software Engineering
MCU Micro Controller Unit
MDSE Model-driven Software Engineering
MiL Model-in-the-Loop
MQTT Message Queuing Telemetry Transport
NB-IoT NarrowBand-IoT
NFR Nonfunctional Requirement
OCM Operator-Controller-Modul
PaaS Platform as a Service
PDAL Peripheral Drivers Abstraction Layer
PE Programming Environment
PiL Processor-in-the-Loop
PL Programmable Logic
PPP Point-to-Point
PS Processing System
pulseAT pulse Analysis Tool
PWM Pulsweitenmodulation

QoS Quality of Service
RAM Random-Access Memory
RE Runtime Environment
REST Representational State Transfer
RFID Radio-Frequency Identification
RGB Red Green Blue
RISC Reduced Instruction Set Computer
ROA Ressourcenorientierte Architektur
ROS Robot Operation System
ROV Remotly Operated Vehicle
RSB Robotics Service Bus
RTC Real-Time Communication
S4R Software for Robots
SaaS Software as a Service
SBC Single-Board Computer
SDK Software Development Kit
SFB Sonderforschungsbereich
SiL Software-in-the-Loop
SLAM Simultaneous Localization and Mapping
SNMP Simple Network Management Protocol
SOA Systemorientierte Architektur
SoC System on Chip
SoPC System on a Programmable Chip
SoS System of Systems
SPoF Single Point of Failure
SSL Secure Socket Layer
SYSiL System-in-the-Loop
TCP Transport Control Protocol
TLS Transport Layer Security
ToF Time of Flight
TSDB Time Series Database
TSN Time-Sensitive Networking
UAS Unmanned Aircraft System
UAV Unmanned Arial Vehicle
UDP User Datagram Protocol
UML Unified Modeling Language
UMV Unmanned Marine Vehicle
USB Universal Serial Bus
UUV Unmanned Underwater Vehicle
VM Virtuelle Maschine
VPN Virtual Private Network
WLAN Wireless Local Area Network
WMR Wheeled Mobile Robots

Tabellenverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

Literaturverzeichnis

[1] *NAO 6 - Preliminary Marketing Datasheet*. Documentation Index: 01. SoftBank Robotics. Feb. 2018.

[2] David Seal. *ARM Architecture Reference Manual*. 2nd. USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 0201737191.

[3] S. Herbrechtsmeier, T. Korthals, T. Schopping und U. Rückert. „AMiRo: A modular customizable open-source mini robot platform“. In: *2016 20th International Conference on System Theory, Control and Computing (ICSTCC)*. 2016, S. 687–692. DOI: 10.1109/ICSTCC.2016.7790746.

[4] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler und Andrew Y Ng. „ROS: an open-source Robot Operating System“. In: *ICRA workshop on open source software*. 2009.

[5] Mark W Maier. „Architecting principles for systems-of-systems“. In: *Systems Engineering: The Journal of the International Council on Systems Engineering* 1.4 (1998), S. 267–284. DOI: 10.1002/(SICI)1520-6858(1998)1:4%3C267::AID-SYS3%3E3.0.CO;2-D.

[6] Len Bass, Paul Clements und Rick Kazman. *Software Architecture in Practice*. Third Edition. Pearson Education, Inc., 2013. ISBN: 978-0-321-81573-6.

[7] K. Jo, J. Kim, D. Kim, C. Jang und M. Sunwoo. „Development of Autonomous Car—Part I: Distributed System Architecture and Development Process“. In: *IEEE Transactions on Industrial Electronics* 61.12 (2014), S. 7131–7140. DOI: 10.1109/TIE.2014.2321342.

[8] Nick Rozanski und Eoin Woods. *Software Systems Architecture: Working With Stakeholders using Viewpoints and Perspectives*. 2. Aufl. Pearson Education, Inc., 2012. ISBN: 978-0-321-71833-4.

[9] Bashar Nuseibeh. „Weaving Together Requirements and Architectures“. In: *Computer* 34.3 (2001), S. 115–119. DOI: 10.1109/2.910904.

[10] Ralf Kneuper. *Software Processes and Life Cycle Models*. Springer, 2018. DOI: 10.1007/978-3-319-98845-0.

[11] Jakob Engblom. „Continuous integration for embedded systems using simulation“. In: *Embedded World 2015 Congress*. 2015.

[12] Helena Holmström Olsson und Jan Bosch. „Towards Data-Driven Product Development: A Multiple Case Study on Post-deployment Data Usage in Software-Intensive Embedded Systems“. In: *Lean Enterprise Software and Systems*. Hrsg. von Brian Fitzgerald, Kieran Conboy, Ken Power, Ricardo Valerdi, Lorraine Morgan und Klaas-Jan Stol. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, S. 152–164. ISBN: 978-3-642-44930-7.

[13] Aline Dresch, Daniel Pacheco Lacerda und José Antônio Valle Antunes. *Design science research: A method for science and technology advancement*. 2015, S. 1–161. DOI: 10.1007/978-3-319-07374-3.

[14] Hannu Jaakkola und Bernhard Thalheim. „Architecture-Driven Modelling Methodologies“. In: (2011), S. 97–116.

[15] „IEEE Recommended Practice for Architectural Description for Software-Intensive Systems“. In: *IEEE Std 1471-2000* (2000), S. 1–30. DOI: 10.1109/IEEESTD.2000.91944.

[16] Christion Haubelt und Jürgen Teich. *Digitale Hardware/Software-Systeme - Spezifikation und Verifikation*. Springer-Verlag, Berlin/Heidelberg, Germany, 2010. DOI: 10.1007/978-3-642-05356-6.

[17] Andrew S. Tanenbaum. *Structured Computer Organization*. 5. Edition. Prentice-Hall, Inc., 2005. ISBN: 0-13-148521-0.

[18] H. Balzert. *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*. Lehrbücher der Informatik. Spektrum Akademischer Verlag, 2011. ISBN: 9783827422460.

[19] Maarten van Steen und Andrew S. Tanenbaum. *Distributed Systems - Principles and Paradigms*. Version 3.01. 2017, S. 582. ISBN: 978-15-430573-8-6.

[20] A. A. Süzen, B. Duman und B. Şen. „Benchmark Analysis of Jetson TX2, Jetson Nano and Raspberry PI using Deep-CNN“. In: *2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*. 2020, S. 1–5. DOI: 10.1109/HORA49412.2020.9152915.

[21] Louise H Crockett, Ross A Elliot, Martin A Enderwitz und Robert W Stewart. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014. ISBN: 099297870X.

[22] Christian Baun, Marcel Kunze, Jens Nimis und Stefan Tai. *Cloud Computing - Web basierte dynamische IT-Services*. Springer Berlin Heidelberg, 2010. DOI: 10.1007/978-9-642-01594-6.

[23] F Lui, Jin Tong, Jian Mao, Robert Bohn, John Messina, Lee Badger und Dawn Leaf. In: *National Institute of Standard and Technology* NIST Specication (2011).

[24] C. N. Höfer und G. Karagiannis. „Cloud computing services: Taxonomy and comparison“. In: *Journal of Internet Services and Applications* 2.2 (2011), S. 81–94. DOI: 10.1007/s13174-011-0027-x.

[25] Edward Griffor. *Handbook of System Safety and Security*. Elsevier Inc., 2017. ISBN: 978-0-12-803773-7.

[26] Tharam Dillon, Chen Wu und Elizabeth Chang. „Cloud computing: Issues and challenges“. In: *Proceedings - International Conference on Advanced Information Networking and Applications, AINA* (2010), S. 27–33. DOI: 10.1109/AINA.2010.187.

[27] Philipp Adelt, Jörg Donoth, Jürgen Gausemeier, Jens Geisler, Stefan Henkler, Sascha Kahl, Benjamin Klöpper, Alexander Krupp, Eckehard Münch, Simon Oberthür, Carlos Paiz, Mario Porrmann, Rafael Radkowski, Christoph Romaus, Alexander Schmidt, Bernd Schulz, Henner Voecking, Ulf Witkowski, Katrin Witting und Alex Znamenshchykov. „Selbstoptimierende Systeme des Maschinenbaus: Definitionen, Anwendungen, Konzepte“. In: *HNI-Verlagsschriftenreihe* Band 234 (2009).

[28] Jürgen Gausemeier und Sascha Kahl. „Architecture and Design Methodology of Self-Optimizing Mechatronic Systems“. In: *Mechtronic Systems, Simulation, Modeling and Control*. Vucovar: InTech Open Access Publisher, 2010, S. 255–282.

[29] Günter Ropohl. *Allgemeine Technologie: Systemtheorie der Technik*. 3. Auflage. Universitätsverlag Karlsruhe, Karlsruhe, 2009, S. 360. DOI: 10.5445/KSP/1000011529.

[30] Gerhard Strube. „Modelling Motivation and Action Control in Cognitive Systems“. In: *Mind Modelling* January 1998 (1998), S. 98–108.

[31] J. Frolik, M. Abdelrahman und P. Kandasamy. „A confidence-based approach to the self-validation, fusion and reconstruction of quasi-redundant sensor data“. In: *IEEE Transactions on Instrumentation and Measurement* 50.6 (2001), S. 1761–1769. DOI: 10.1109/19.982977.

[32] Rolf Naumann. *Modellierung und Verarbeitung vernetzter intelligenter mechatronischer Systeme*. Fortschritt. VDI-Verl., 2000, S. 180. ISBN: 3-18-331820-2.

[33] Oliver Oberschelp, Thorsten Hestermeyer und Holger Giese. „Strukturierte Informationsverarbeitung für selbstoptimierende mechatronische Systeme“. In: *2. Paderborner Workshop Intelligente Mechatronische Systeme*. Verlagsschriftenreihe des Heinz Nixdorf Instituts, Paderborn. Heinz Nixdorf Institut. Paderborn, 2004.

[34] Thorsten Hestermeyer. „Strukturierte Entwicklung der Informationsverarbeitung für die aktive Federung eines Schienenfahrzeugs“. Dissertation. Universität Paderborn, 2006.

[35] Oliver Oberschelp. „Strukturierter Entwurf selbstoptimierender mechatronischer Systeme“. Dissertation. Universität Paderborn, 2008.

[36] Eckehard Münch. „Selbstoptimierung verteilter mechatronischer Systeme auf Basis paretooptimaler Systemkonfigurationen“. Dissertation. Universität Paderborn, 2012.

[37] Roman Dumitrescu. „Entwicklungssystematik zur Integration kognitiver Funktionen in fortgeschrittene mechatronische Systeme“. Dissertation. Universität Paderborn, 2010.

[38] Francisco Javier, Rodríguez Lera, Jesús Balsa, Fernando Casado, Camino Fernández, Francisco Martín Rico und Vicente Matellán. „Cybersecurity in Autonomous Systems: Evaluating the performance of hardening ROS“. In: *Proceedings of the Waf2016* June (2016), S. 1–7.

[39] Yuya Maruyama, Shinpei Kato und Takuya Azumi. „Exploring the Performance of ROS2“. In: *Proceedings of the 13th International Conference on Embedded Software*. EMSOFT '16. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2016. DOI: 10.1145/2968478.2968502.

[40] Thomas Schöpping, Timo Korthals, Marc Hesse und Ulrich Rückert. „Generic Architecture for Modular Real-time Systems in Robotics“. In: *ICINCO 2018 - Proceedings of the 15th International Conference on Informatics in Control, Automation and Robotics* 2 (2018), S. 413–420. DOI: 10.5220/0006899304130420.

[41] Aakash Ahmad und Muhammad Ali Babar. „Software architectures for robotic systems: A systematic mapping study“. In: *Journal of Systems and Software* 122 (2016), S. 16–39. DOI: 10.1016/j.jss.2016.08.039.

[42] Peter Corke, Pavan Sikka, Jonathan M. Roberts und Elliot Duff. „DDX : A distributed software architecture for robotic systems“. In: *Australasian Conference on Robotics and Automation 2004 ACRA2004*. Dez. 2004. URL: https://eprints.qut.edu.au/33835/.

[43] Antti Tikanmäki und Juha Röning. „Property service architecture for distributed robotic and sensor systems“. In: *International Conference on Informatics in Control, Automation and Robotics* (2007), S. 226–233. DOI: 10.5220/0001650002260233.

[44] Tobias Kaupp, Alex Brooks, Ben Upcroft und Alexei Makarenko. „Building a software architecture for a human-robot team using the orca framework“. In: *Proceedings - IEEE International Conference on Robotics and Automation* April (2007), S. 3736–3741. DOI: 10.1109/ROBOT.2007.364051.

[45] Shuo Yang, Xinjun Mao, Sen Yang und Zhe Liu. „Towards a hybrid software architecture and multi-agent approach for autonomous robot software“. In: *International Journal of Advanced Robotic Systems* 14.4 (2017), S. 1–15. DOI: 10.1177/1729881417716088.

[46] Sergio Garcia, Claudio Menghi, Patrizio Pelliccione, Thorsten Berger und Rebekka Wohlrab. „An Architecture for Decentralized, Collaborative, and Autonomous Robots“. In: *Proceedings - 2018 IEEE 15th International Conference on Software Architecture, ICSA 2018* (2018), S. 75–84. DOI: 10.1109/ICSA.2018.00017.

[47] Jeff Kramer und Jeff Magee. „Self-managed systems: An architectural challenge“. In: *FoSE 2007: Future of Software Engineering* (2007), S. 259–268. DOI: 10.1109/FOSE.2007.19.

[48] D. Paul Benjamin, Damian Lyons und Deryle Lonsdale. „ADAPT: A Cognitive Architecture for Robotics“. In: *Proceedings of the International Conference on Cognitive Modelling, ICCM 2004*. 2004.

[49] Catherina Burghart, Ralf Mikut, Rainer Stiefelhagen, Tamim Asfour, Hartwig Holzapfel, Peter Steinhaus und Ruediger Dillmann. „A cognitive architecture for a humanoid robot: A first approach“. In: *Proceedings of 2005 5th IEEE-RAS International Conference on Humanoid Robots* 2005.Humanoids (2005), S. 357–362. DOI: 10.1109/ICHR.2005.1573593.

[50] Jose Luis Sanchez-Lopez, Ramón A Suárez Fernández, Hriday Bavle, Carlos Sampedro, Martin Molina, Jesus Pestana und Pascual Campoy. „Aerostack: An architecture and open-source software framework for aerial robotics“. In: *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE. 2016, S. 332–341. DOI: 10.1109/ICUAS.2016.7502591.

[51] Daniel Feitosa und Elisa Yumi Nakagawa. „An Investigation into Reference Architectures for Mobile Robotic Systems“. In: *Proceedings of the Seventh International Conference on Software Engineering Advances (ICSEA'12)* c (2012), S. 465–471.

[52] James S. Albus. „4D/RCS A Reference Model Architecture for Intelligent Unmanned Ground Vehicles“. In: *Unmanned Ground Vehicle Technology IV* 4715 (2002), S. 303–310. DOI: 10.1117/12.474462.

[53] Steve Rowe und Christopher R. Wagner. „An introduction to the Joint Architecture for Unmanned Systems (JAUS)“. In: *Fall Simulation Interoperability Workshop 2007* 2 (2007), S. 1645–1652.

[54] Francisco Ortiz, Diego Alonso, Juan Pastor, Barbara Alvarez und Andres Iborr. „A Reference Control Architecture for Service Robots as applied to a Climbing Vehicle“. In: *Bioinspiration and Robotics Walking and Climbing Robots*. 2007. DOI: 10.5772/5501.

[55] L. Peters, M. Pauly und A. Arghir. „Servicebots - a scalable architecture for autonomous service robots“. In: *IEEE International Conference on Fuzzy Systems*. 2000. DOI: 10.1109/fuzzy.2000.839187.

[56] Danny Weyns und Tom Holvoet. „A reference architecture for situated multiagent systems“. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2007. DOI: 10.1007/978-3-540-71103-2_1.

[57] Bárbara Álvarez, Andrés Iborra, Alejandro Alonso und Juan Antonio De la Puente. „Reference architecture for robot teleoperation: Development details and practical use“. In: *Control Engineering Practice* (2001). DOI: 10.1016/S0967-0661(00)00121-0.

[58] Barbara Hayes-Roth, Karl Pfleger, Philippe Lalanda, Philippe Morignot und Marko Balabanovic. „A Domain-Specific Software Architecture for Adaptive Intelligent Systems“. In: *IEEE Transactions on Software Engineering* (1995). DOI: 10.1109/32.385968.

[59] Stuart Russell und Peter Norvig. *Artificial Intelligence: A Modern Approach*. Dritte Auflage. Pearson Education Limited, 2016. ISBN: 9781292153964.

[60] Sebastian Thrun, Wolfram Burgard und Dieter Fox. *Probabilistic Robotics*. The MIT Press, 2006. ISBN: 978-0-262-20162-9.

[61] Bruno Siciliano und Oussama Khatib. *Springer Handbook of Robotics*. 2. Auflage. Springer Publishing Company, Incorporated, 2016. ISBN: 3319325507.

[62] Peter Corke. *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*. Zweite Auflage. Springer Publishing Company, Incorporated, 2016. ISBN: 978-3-319-54412-0.

[63] Robin R Murphy. *Introduction to AI Robotics*. The MIT Press, 2000. ISBN: 9780262133838.

[64] Nicole Lazzeri, Daniele Mazzei, Lorenzo Cominelli, Antonio Cisternino und Danilo Emilio De Rossi. „Designing the mind of a social robot“. In: *Applied Sciences* 8.2 (2018), S. 302. DOI: 10.3390/app8020302.

[65] Robin R Murphy, Eric Steimle, Michael Hall, Michael Lindemuth, David Trejo, Stefan Hurlebaus, Zenon Medina-Cetina und Daryl Slocum. „Robot-assisted bridge inspection“. In: *Journal of Intelligent & Robotic Systems* 64.1 (2011), S. 77–95. DOI: 0.1007/s10846-010-9514-8.

[66] Vitor A. M. Jorge, Roger Granada, Renan G. Maidana, Darlan A. Jurak, Guilherme Heck, Alvaro P. F. Negreiros, Davi H. dos Santos, Luiz M. G. Gonçalves und Alexandre M. Amory. „A Survey on Unmanned Surface Vehicles for Disaster Robotics: Main Challenges and Directions“. In: *Sensors* 19.3 (2019). DOI: 10.3390/s19030702.

[67] Michael Gerke, Ulrich Borgolte, Ivan Masár, František Jelenciak, Pavol Bahnik und Naef Al-Rashedi. „Lighter-than-air UAVs for surveillance and environmental monitoring“. In: *Future Security Research Conference*. Springer. Springer Berlin Heidelberg, 2012, S. 480–483. ISBN: 978-3-642-33161-9.

[68] Cornelius A. Thiels, Johnathon M. Aho, Scott P. Zietlow und Donald H. Jenkins. *Use of Unmanned Aerial Vehicles for Medical Product Transport*. 2015. DOI: 10.1016/j.amj.2014.10.011.

[69] Jennifer Goetz, Sara Kiesler und Aaron Powers. „Matching robot appearance and behavior to tasks to improve human-robot cooperation“. In: *The 12th IEEE International Workshop on Robot and Human Interactive Communication, 2003. Proceedings. ROMAN 2003.* 2003, S. 55–60. DOI: 10.1109/ROMAN.2003.1251796.

[70] Jinxing Li, Berta Esteban-Fernández de Ávila, Wei Gao, Liangfang Zhang und Joseph Wang. „Micro/Nanorobots for Biomedicine: Delivery, surgery, sensing, and detoxification“. In: *Science Robotics* 2.4 (2017). DOI: 10.1126/scirobotics.aam6431.

[71] A. M. Turing. „Computing Machinery and Intelligence“. In: *Mind* LIX.236 (Okt. 1950), S. 433–460. DOI: 10.1093/mind/LIX.236.433.

[72] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, Kenny Lau, Celia Oakley, Mark Palatucci, Vaughan Pratt, Pascal Stang, Sven Strohband, Cedric Dupont, Lars-Erik Jendrossek, Christian Koelen, Charles Markey, Carlo Rummel, Joe van Niekerk, Eric Jensen, Philippe Alessandrini, Gary Bradski, Bob Davies, Scott Ettinger, Adrian Kaehler, Ara Nefian und Pamela Mahoney. „Stanley: The robot that won the DARPA Grand Challenge“. In: *Journal of Field Robotics* 23.9 (2006), S. 661–692. DOI: 10.1002/rob.20147.

[73] Justus J. Randolph. „A guide to writing the dissertation literature review“. In: *Practical Assessment, Research and Evaluation* 14.13 (2009). DOI: 10.7275/b0az-8t74.

[74] Jan vom Brocke, Alexander Simons, Bjoern Niehaves, Bjorn Niehaves, Kai Riemer, Ralf Plattfaut und Anne Cleven. „Reconstructing the Giant: On the Importance of Rigour in Documenting the Literature Search Process“. In: *Information systems in a globalising world : challenges, ethics and practices ; ECIS 2009, 17th European Conference on Information Systems*. Hrsg. von Susan Newell, Edgar Whitley, Nancy Pouloudi, Jonathan Wareham und Lars Mathiassen. Verona: Università di Verona, Facoltà di Economia, Departimento de Economia Aziendale, 2009, S. 2206–2217. URL: https://www.alexandria.unisg.ch/213419/.

[75] Roland Siegwart, Illah R. Nourbakhsh und Davide Scaramuzza. *Introduction to Autonomous Mobile Robots*. 2. Auflage. The MIT Press, 2011. ISBN: 0262015358.

[76] Thomas Bräunl. *Embedded Robotics*. Third Edition. Springer-Verlag, Berlin/Heidelberg, Germany, 2008. DOI: 10.1007/978-3-662-05099-6.

[77] Mark Yim, Wei Min Shen, Benham Salemi, Daniela Rus, Mark Moll, Hod Lipson, Eric Klavins und Gregory S. Chirikjian. „Modular self-reconfigurable robot systems [Grand challenges of robotics]“. In: *IEEE Robotics and Automation Magazine* 14.1 (2007), S. 43–52. DOI: 10.1109/MRA.2007.339623.

[78] Guang Zhong Yang, Jim Bellingham, Pierre E. Dupont, Peer Fischer, Luciano Floridi, Robert Full, Neil Jacobstein, Vijay Kumar, Marcia McNutt, Robert Merrifield, Bradley J. Nelson, Brian Scassellati, Mariarosaria Taddeo, Russell Taylor, Manuela Veloso, Zhong Lin Wang und Robert Wood. „The grand challenges of science robotics“. In: *Science Robotics* 3.14 (2018). DOI: 10.1126/scirobotics.aar7650.

[79] Cyril Robin und Simon Lacroix. „Multi-robot target detection and tracking: taxonomy and survey“. In: *Autonomous Robots* 40.4 (2016), S. 729–760. DOI: 10.1007/s10514-015-9491-7.

[80] Terry Huntsberger, Guillermo Rodriguez und Paul S. Schenker. „Robotics challenges for robotic and human Mars exploration“. In: *Proceedings of the 4th International Conference and Exposition on Robotics for Challenging Situations and Environments - Robotics 2000* 299 (2000), S. 340–346. DOI: 10.1061/40476(299)45.

[81] George Adamides, Georgios Christou, Christos Katsanos, Michalis Xenos und Thanasis Hadzilacos. „Usability guidelines for the design of robot teleoperation: A taxonomy“. In: *IEEE Transactions on Human-Machine Systems* 45.2 (2015), S. 256–262. DOI: 10.1109/THMS.2014.2371048.

[82] Francisco Rubio, Francisco Valero und Carlos Llopis-Albert. „A review of mobile robots: Concepts, methods, theoretical framework, and applications“. In: *International Journal of Advanced Robotic Systems* 16.2 (2019), S. 1–22. DOI: 10.1177/1729881419839596.

[83] Chimsom Chukwuemeka und Maki Habib. „Development of autonomous networked robots (ANR) for surveillance: Conceptual design and requirements“. In: *Proceedings: IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society* (2018), S. 3757–3763. DOI: 10.1109/IECON.2018.8591452.

[84] Mary B. Alatise und Gerhard P. Hancke. „A Review on Challenges of Autonomous Mobile Robot and Sensor Fusion Methods“. In: *IEEE Access* 8 (2020), S. 39830–39846. DOI: 10.1109/ACCESS.2020.2975643.

[85] Jorge M. Soares, Iñaki Navarro und Alcherio Martinoli. „The Khepera IV Mobile Robot: Performance Evaluation, Sensory Data and Software Toolbox“. In: *Robot 2015: Second Iberian Robotics Conference*. Hrsg. von Luís Paulo Reis, António Paulo Moreira, Pedro U. Lima, Luis Montano und Victor Muñoz-Martinez. Cham: Springer International Publishing, 2016, S. 767–781. ISBN: 978-3-319-27146-0.

[86] Thomas Schöpping, Timo Korthals, Marc Hesse und Ulrich Rückert. „AMiRo: A Mini Robot as Versatile Teaching Platform“. In: *Advances in Intelligent Systems and Computing* 829 (2019), S. 177–188. DOI: 10.1007/978-3-319-97085-1_18.

[87] Joseph Betthauser, Daniel Benavides, Jeff Schornick, Neal O'Hara, Jimit Patel, Jeremy Cole und Edgar Lobaton. „WolfBot: A distributed mobile sensing platform for research and education“. In: *Proceedings of the 2014 Zone 1 Conference of the American Society for Engineering Education - Ëngineering Education: Industry Involvement and Interdisciplinary Trends", ASEE Zone 1 2014* (2014), S. 1–8. DOI: 10.1109/ASEEZone1.2014.6820632.

[88] Francesco Mondada, Michael Bonani, Xavier Raemy, James Pugh, Christopher Cianci, Adam Klaptocz, Steffane Magnenat, Jean-Christophe Zufferey, Dario Floreano und Alcherio Martinoli. „The e-puck, a robot designed for education in engineering“. In: *Proceedings of the 9th conference on autonomous robot systems and competitions* 1.1 (2009), S. 59–65.

[89] Christof Röhrig und Daniel Heß. „Mobile Manipulation for Human-Robot Collaboration in Intralogistics“. In: *IAENG Transactions on Engineering Sciences - Special Issue for the International Association of Engineers Conferences* (2020), S. 1–20. DOI: 10.1142/9789811215094_0001.

[90] Ilya B Gartseev, Leng-feng Lee und Venkat N Krovi. „A Low-Cost Real-Time Mobile Robot Platform (ArEduBot) to support Project-Based Learning in Robotics & Mechatronics“. In: *2nd International Conference on Robotics in Education (RiE 2011)* (2011), S. 117–124.

[91] Jingyang Wu, Chaoshun Lv, Lijun Zhao, Ruifeng Li und Guanglin Wang. „Design and implementation of an omnidirectional mobile robot platform with unified I/O interfaces“. In: *2017 IEEE International Conference on Mechatronics and Automation, ICMA 2017* (2017), S. 410–415. DOI: 10.1109/ICMA.2017.8015852.

[92] S. Meghana, Teja V. Nikhil, Raghuveer Murali, S. Sanjana, R. Vidhya und Khurram J. Mohammed. „Design and implementation of surveillance robot for outdoor security“. In: *RTEICT 2017 - 2nd IEEE International Conference on Recent Trends in Electronics, Information and Communication Technology, Proceedings* 2018-Janua (2018), S. 1679–1682. DOI: 10.1109/RTEICT.2017.8256885.

[93] Omar Yaseen Ismael und John Hedley. „Analysis, Design, and Implementation of an Omnidirectional Mobile Robot Platform“. In: *American Scientific Research Journal for Engineering* 22.1 (2016), S. 195–209. ISSN: 2313-4402.

[94] Stefan Herbrechtsmeier. „Modell eines agilen Leiterplattenentwurfsprozesses basierend auf der interdisziplinären Entwicklung eines modularen autonomen Miniroboters“. Dissertation. Universität Bielefeld, 2016.

[95] Johannes Wienke und Sebastian Wrede. „A middleware for collaborative research in experimental robotics“. In: *2011 IEEE/SICE International Symposium on System Integration, SII 2011* (2011), S. 1183–1190. DOI: 10.1109/SII.2011.6147617.

[96] Brian Paden, Michal Čáp, Sze Zheng Yong, Dmitry Yershov und Emilio Frazzoli. „A survey of motion planning and control techniques for self-driving urban vehicles“. In: *IEEE Transactions on Intelligent Vehicles* 1.1 (2016), S. 33–55. DOI: 10.1109/TIV.2016.2578706.

[97] M. Bajracharya, M. W. Maimone und D. Helmick. „Autonomy for Mars Rovers: Past, Present, and Future“. In: *Computer* 41.12 (2008), S. 44–50. DOI: 10.1109/MC.2008.479.

[98] Håvard F. Grip, Johnny Lam, David S. Bayard, Dylan T. Conway, Gurkirpal Singh, Roland Brockers, Jeff H. Delaune, Larry H. Matthies, Carlos Malpica, Travis L. Brown, Abhinandan Jain, Alejandro M. San Martin und Gene B. Merewether. „Flight Control System for NASA's Mars Helicopter“. In: *AIAA Scitech 2019 Forum*. DOI: 10.2514/6.2019-1289.

[99] Jenay M. Beer, Arthur D. Fisk und Wendy A. Rogers. „Toward a Framework for Levels of Robot Autonomy in Human-Robot Interaction“. In: *Journal of Human-Robot Interaction* 3.2 (Juli 2014), S. 74–99. DOI: 10.5898/JHRI.3.2.Beer.

[100] E. Sholes. „Evolution of a UAV Autonomy Classification Taxonomy“. In: *2007 IEEE Aerospace Conference*. 2007, S. 1–16. DOI: 10.1109/AERO.2007.352738.

[101] Leonel Rozo, Pablo Jiménez und Carme Torras. „A robot learning from demonstration framework to perform force-based manipulation tasks“. In: *Intelligent service robotics* 6.1 (2013), S. 33–51. DOI: 10.1007/s11370-012-0128-9.

[102] Jesse Levinson, Jake Askeland, Jan Becker, Jennifer Dolson, David Held, Soeren Kammel, J. Zico Kolter, Dirk Langer, Oliver Pink, Vaughan Pratt, Michael Sokolsky, Ganymed Stanek, David Stavens, Alex Teichman, Moritz Werling und Sebastian Thrun. „Towards fully autonomous driving: Systems and algorithms“. In: *IEEE Intelligent Vehicles Symposium, Proceedings* Iv (2011), S. 163–168. DOI: 10.1109/IVS.2011.5940562.

[103] Marco Dorigo, Dario Floreano, Luca Maria Gambardella, Francesco Mondada, Stefano Nolfi, Tarek Baaboura, Mauro Birattari, Michael Bonani, Manuele Brambilla, Arne Brutschy, Daniel Burnier, Alexandre Campo, Anders Lyhne Christensen, Antal Decugniere, Gianni Di Caro,

Frederick Ducatelle, Eliseo Ferrante, Alexander Förster, Javier Martinez Gonzales, Jerome Guzzi, Valentin Longchamp, Stephane Magnenat, Nithin Mathews, Marco Montes De Oca, Rehan O'Grady, Carlo Pinciroli, Giovanni Pini, Philippe Rétornaz, James Roberts, Valerio Sperati, Timothy Stirling, Alessandro Stranieri, Thomas Stützle, Vito Trianni, Elio Tuci, Ali Emre Turgut und Florian Vaussard. „Swarmanoid: A novel concept for the study of heterogeneous robotic swarms". In: *IEEE Robotics and Automation Magazine* 20.4 (2013), S. 60–71. DOI: 10.1109/MRA.2013.2252996.

[104] Jérémie Guiochet, Mathilde Machin und Hélène Waeselynck. „Safety-critical advanced robots: A survey". In: *Robotics and Autonomous Systems* 94 (2017), S. 43–52. ISSN: 0921-8890. DOI: 10.1016/j.robot.2017.04.004.

[105] Alberto Broggi, Michele Buzzoni, Stefano Debattisti, Paolo Grisleri, Maria Chiara Laghi, Paolo Medici und Pietro Versari. „Extensive tests of autonomous driving technologies". In: *IEEE Transactions on Intelligent Transportation Systems* 14.3 (2013), S. 1403–1415. DOI: 10.1109/TITS.2013.2262331.

[106] M. Vasic und A. Billard. „Safety issues in human-robot interactions". In: *2013 IEEE International Conference on Robotics and Automation*. 2013, S. 197–204. DOI: 10.1109/ICRA.2013.6630576.

[107] Jessie Y.C. Chen, Ellen C. Haas und Michael J. Barnes. „Human performance issues and user interface design for teleoperated robots". In: *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews* 37.6 (2007), S. 1231–1245. DOI: 10.1109/TSMCC.2007.905819.

[108] Laurel D Riek und Senior Member. „Movement Coordination in Human – Robot Teams :" In: *IEEE Transactions on Robotics* 32.4 (2016), S. 909–919. DOI: 10.1109/TRO.2016.2570240.

[109] Nico Hochgeschwender, Gary Cornelius und Holger Voos. „Arguing Security of Autonomous Robots". In: *IEEE International Conference on Intelligent Robots and Systems* (2019), S. 7791–7797. DOI: 10.1109/IROS40897.2019.8967670.

[110] Jennifer Carlson und Robin R. Murphy. „Reliability analysis of mobile robots". In: *Proceedings - IEEE International Conference on Robotics and Automation* 1 (2003), S. 274–281. DOI: 10.1109/robot.2003.1241608.

[111] Frank Künemund, Daniel Hess und Christof Röhrig. „Energy efficient kinodynamic motion planning for holonomic AGVs in industrial applications using state lattices". In: *47th International Symposium on Robotics, ISR 2016* 2016 (2016), S. 459–466.

[112] Jürgen Pripfl, Tobias Körtner, Daliah Batko-Klein, Denise Hebesberger, Markus Weninger, Christoph Gisinger, Susanne Frennert, Hakan Eftring, Margarita Antona, Ilia Adami, Astrid Weiss, Markus Bajones und Markus Vincze. „Results of a real world trial with a mobile social service robot for older adults". In: *ACM/IEEE International Conference on Human-Robot Interaction* 2016-April (2016), S. 497–498. DOI: 10.1109/HRI.2016.7451824.

[113] Seppe Terryn, Joost Brancart, Dirk Lefeber, Guy Van Assche und Bram Vanderborght. „Self-healing soft pneumatic robots“. In: *Science Robotics* 2.9 (2017), S. 1–13. DOI: 10.1126/scirobotics.aan4268.

[114] O. Khatib. „Real-time obstacle avoidance for manipulators and mobile robots“. In: *Proceedings. 1985 IEEE International Conference on Robotics and Automation*. Bd. 2. März 1985, S. 500–505. DOI: 10.1109/ROBOT.1985.1087247.

[115] An Min Zou, Zeng Guang Hou, Si Yao Fu und Min Tan. „Neural networks for mobile robot navigation: A survey“. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 3972 LNCS (2006), S. 1218–1226. DOI: 10.1007/11760023_177.

[116] Gary Bradski und Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Inc., 2008.

[117] Eduardo Arnold, Omar Y. Al-Jarrah, Mehrdad Dianati, Saber Fallah, David Oxtoby und Alex Mouzakitis. „A Survey on 3D Object Detection Methods for Autonomous Driving Applications“. In: *IEEE Transactions on Intelligent Transportation Systems* 20.10 (2019), S. 3782–3795. DOI: 10.1109/TITS.2019.2892405.

[118] E. Guizzo. „Robots with their heads in the clouds“. In: *IEEE Spectrum* 48.3 (2011), S. 16–18. DOI: 10.1109/MSPEC.2011.5719709.

[119] Jiafu Wan, Shenglong Tang, Hehua Yan, Di Li, Shiyong Wang und Athanasios V. Vasilakos. „Cloud robotics: Current status and open issues“. In: *IEEE Access* 4 (2016), S. 2797–2807. DOI: 10.1109/ACCESS.2016.2574979.

[120] A. Roennau, G. Heppner, T. Kerscher und R. Dillmann. „Fault diagnosis and system status monitoring for a six-legged walking robot“. In: *IEEE/ASME International Conference on Advanced Intelligent Mechatronics, AIM* (2011), S. 874–879. DOI: 10.1109/AIM.2011.6027107.

[121] Raphael Golombek, Sebastian Wrede, Marc Hanheide und Martin Heckmann. „Learning a probabilistic self-awareness model for robotic systems“. In: *IEEE/RSJ 2010 International Conference on Intelligent Robots and Systems, IROS 2010 - Conference Proceedings* (2010), S. 2745–2750. DOI: 10.1109/IROS.2010.5651095.

[122] S. Mellah, G. Graton, E. M. E. Adell, M. Ouladsine und A. Planchais. „Mobile Robot Additive Fault Diagnosis and Accommodation“. In: *2019 8th International Conference on Systems and Control (ICSC)*. 2019, S. 241–246. DOI: 10.1109/ICSC47195.2019.8950504.

[123] Henning Kagermann, Wolfgang Wahlster und Johannes Helbig. *Umsetzungsempfehlungen für das Zukunftsprojekt Industrie 4.0*. Techn. Ber. 5. 2013.

[124] Juan Pablo Mendoza und Reid Simmons. „Mobile Robot Fault Detection based on Redundant Information Statistics“. In: *2012 International Conference on Intelligent Robots and Systems* (2012). DOI: 10.1184/R1/6607376.

[125] Yan Meng, Yuyang Zhang und Yaochu Jin. „Autonomous self-reconfiguration of modular robots by evolving a hierarchical mechanochemical model“. In: *IEEE Computational Intelligence Magazine* 6.1 (2011), S. 43–54. DOI: 10.1109/MCI.2010.939579.

[126] Fang Tang und Lynne E. Parker. „ASyMTRe: Automated synthesis of multi-robot task solutions through software reconfiguration“. In: *Proceedings - IEEE International Conference on Robotics and Automation* 2005.April (2005), S. 1501–1508. DOI: 10.1109/ROBOT.2005.1570327.

[127] V. K. Sarker, J. Pena Queralta, T. N. Gia, H. Tenhunen und T. Westerlund. „Offloading SLAM for Indoor Mobile Robots with Edge-Fog-Cloud Computing“. In: *1st International Conference on Advances in Science, Engineering and Robotics Technology 2019, ICASERT 2019* 2019.Icasert (2019). DOI: 10.1109/ICASERT.2019.8934466.

[128] Ryan Krauss. „Combining Raspberry Pi and Arduino to form a low-cost, real-time autonomous vehicle platform“. In: *Proceedings of the American Control Conference* 2016-July (2016), S. 6628–6633. DOI: 10.1109/ACC.2016.7526714.

[129] Helmut Krcmar. „Informationsmanagement“. In: *Informationsmanagement*. 6. Auflage. Springer, 2015, S. 85–111. DOI: 10.1007/978-3-662-45863-1.

[130] Holly A Yanco und Jill L Drury. „A taxonomy for human-robot interaction“. In: *Proceedings of the AAAI Fall Symposium on Human-Robot Interaction*. sn. 2002, S. 111–119.

[131] Gregory Dudek, Michael Jenkin, Evangelos Milios und David Wilkes. „A taxonomy for swarm robots“. In: *Proceedings of 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'93)*. Bd. 1. IEEE. 1993, S. 441–447. DOI: 10.1109/IROS.1993.583135.

[132] Gregory Dudek, Michael RM Jenkin, Evangelos Milios und David Wilkes. „A taxonomy for multi-agent robotics“. In: *Autonomous Robots* 3.4 (1996), S. 375–397. DOI: 10.1007/BF00240651.

[133] Steffen Steinert. „The five robots—a taxonomy for roboethics“. In: *International Journal of Social Robotics* 6.2 (2014), S. 249–260. DOI: 10.1007/s12369-013-0221-z.

[134] Matthias Kramer James und Scheutz. „Development environments for autonomous mobile robots: A survey“. In: *Autonome Roboter* 22.2 (2007), S. 101–132. DOI: 10.1007/s10514-006-9013-8.

[135] Nader Mohamed, Jameela Al-Jaroodi und Imad Jawhar. „Middleware for robotics: A survey“. In: *2008 IEEE Conference on Robotics, Automation and Mechatronics*. Ieee. 2008, S. 736–742. DOI: 10.1109/RAMECH.2008.4681485.

[136] Karl Wiegers und Joy Beatty. *Software requirements*. Third Edition. Pearson Education, 2013. ISBN: 978-0-7356-7966-5.

[137] ISO/IEC. *ISO/IEC 9126-1:2001. Software engineering - Product quality*. ISO/IEC, 2001.

[138] A. Birolini. *Zuverlässigkeit von Geräten und Systemen*. Springer Berlin Heidelberg, 2013. ISBN: 9783642603990.

[139] Dewi Mairiza, Didar Zowghi und Nurie Nurmuliani. „An Investigation into the Notion of Non-Functional Requirements“. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. SAC '10. Sierre, Switzerland: Association for Computing Machinery, 2010, S. 311–317. DOI: 10.1145/1774088.1774153.

[140] V. Sze, Y. Chen, J. Emer, A. Suleiman und Z. Zhang. „Hardware for machine learning: Challenges and opportunities“. In: *2017 IEEE Custom Integrated Circuits Conference (CICC)*. 2017, S. 1–8. DOI: 10.1109/CICC.2017.7993626.

[141] S. Jain, A. Vaibhav und L. Goyal. „Raspberry Pi based interactive home automation system through E-mail“. In: *2014 International Conference on Reliability Optimization and Information Technology (ICROIT)*. 2014, S. 277–280. DOI: 10.1109/ICROIT.2014.6798330.

[142] K. Krinkin, E. Stotskaya und Y. Stotskiy. „Design and implementation Raspberry Pi-based omni-wheel mobile robot“. In: *2015 Artificial Intelligence and Natural Language and Information Extraction, Social Media and Web Search FRUCT Conference (AINL-ISMW FRUCT)*. 2015, S. 39–45. DOI: 10.1109/AINL-ISMW-FRUCT.2015.7382967.

[143] Isaac Jesus Da Silva, Claudio O. Vilao, Anna H.R. Costa und Reinaldo A.C. Bianchi. „Towards robotic cognition using deep neural network applied in a goalkeeper robot“. In: *Proceedings - 2017 LARS 14th Latin American Robotics Symposium and 2017 5th SBR Brazilian Symposium on Robotics, LARS-SBR 2017 - Part of the Robotics Conference 2017* 2017-December (2017), S. 1–6. DOI: 10.1109/SBR-LARS-R.2017.8319463.

[144] Jahanzaib Shabbir und Tarique Anwer. „A Survey of Deep Learning Techniques for Mobile Robot Applications“. In: 14.8 (2018), S. 1–10. arXiv: 1803.07608.

[145] Reem J Alattas, Sarosh Patel und Tarek M Sobh. „Evolutionary modular robotics: Survey and analysis“. In: *Journal of Intelligent & Robotic Systems* 95.3-4 (2019), S. 815–828. DOI: 10.1007/s10846-018-0902-9.

[146] Marina Thottan, Guanglei Liu und Chuanyi Ji. „Anomaly detection approaches for communication networks“. In: *Algorithms for next generation networks*. Springer, 2010, S. 239–261. DOI: 10.1007/978-1-84882-765-3_11.

[147] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić und Leonid Ryzhyk. „System Programming in Rust: Beyond Safety“. In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. HotOS '17. Whistler, BC, Canada: Association for Computing Machinery, 2017, S. 156–161. DOI: 10.1145/3102980.3103006.

[148] Anne Angermann, Michael Beuschel, Martin Rau und Ulrich Wohlfarth. *MATLAB-Simulink-Stateflow: Grundlagen, Toolboxen, Beispiele*. 10. Auflage. Walter de Gruyter GmbH & Co KG, 2021. ISBN: 978-3-11-064107-3.

[149] Jürgen Cito, Philipp Leitner, Thomas Fritz und Harald C. Gall. „The Making of Cloud Applications: An Empirical Study on Software Development for the Cloud“. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, S. 393–403. DOI: 10.1145/2786805.2786826.

[150] Alexander Bergmayr, Uwe Breitenbücher, Nicolas Ferry, Alessandro Rossini, Arnor Solberg, Manuel Wimmer, Gerti Kappel und Frank Leymann. „A Systematic Review of Cloud Modeling Languages“. In: *ACM Comput. Surv.* 51.1 (Feb. 2018). DOI: 10.1145/3150227.

[151] Richard I. Hartley und Peter Sturm. „Triangulation“. In: *Computer Vision and Image Understanding* 68.2 (1997), S. 146–157. DOI: 10.1006/cviu.1997.0547.

[152] Jan Smisek, Michal Jancosek und Tomas Pajdla. „3D with Kinect“. In: *Consumer Depth Cameras for Computer Vision: Research Topics and Applications*. Hrsg. von Andrea Fossati, Juergen Gall, Helmut Grabner, Xiaofeng Ren und Kurt Konolige. London: Springer London, 2013, S. 3–25. DOI: 10.1007/978-1-4471-4640-7_1.

[153] D. Honegger, H. Oleynikova und M. Pollefeys. „Real-time and low latency embedded computer vision hardware based on a combination of FPGA and mobile CPU“. In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2014, S. 4930–4935. DOI: 10.1109/IROS.2014.6943263.

[154] S. K. Rethinagiri, O. Palomar, J. A. Moreno, O. Unsal und A. Cristal. „An energy efficient hybrid FPGA-GPU based embedded platform to accelerate face recognition application“. In: *2015 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XVIII)*. 2015, S. 1–3. DOI: 10.1109/CoolChips.2015.7158532.

[155] M. I. Jordan und T. M. Mitchell. „Machine learning: Trends, perspectives, and prospects“. In: *Science* 349.6245 (2015), S. 255–260. DOI: 10.1126/science.aaa8415. URL: https://science.sciencemag.org/content/349/6245/255.

[156] David Barrie Thomas, Lee Howes und Wayne Luk. „A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation“. In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '09. Monterey, California, USA: Association for Computing Machinery, 2009, S. 63–72. DOI: 10.1145/1508128.1508139.

[157] Elias Vansteenkiste. „New FPGA design tools and architectures“. Dissertation. Ghent University, 2016.

[158] Mohammad Ubaidullah Bokhari, Qahtan Makki und Yahya Kord Tamandani. „A Survey on Cloud Computing“. In: *Big Data Analytics*. Hrsg. von V. B. Aggarwal, Vasudha Bhatnagar und Durgesh Kumar Mishra. Singapore: Springer Singapore, 2018, S. 149–164. ISBN: 978-981-10-6620-7.

[159] Ken Tindell, H Hanssmon und Andy J Wellings. „Analysing Real-Time Communications: Controller Area Network (CAN).“ In: *RTSS*. Citeseer. 1994, S. 259–263.

[160] Martino Migliavacca, Andrea Bonarini und Matteo Matteucci. „RTCAN: A real-time CAN-bus protocol for robotic applications“. In: *ICINCO 2013 - Proceedings of the 10th International Conference on Informatics in Control, Automation and Robotics* 2.January 2016 (2013), S. 353–360. DOI: 10.5220/0004484303530360.

[161] M. Ruff. „Evolution of local interconnect network (LIN) solutions“. In: *2003 IEEE 58th Vehicular Technology Conference. VTC 2003-Fall (IEEE Cat. No.03CH37484)*. Bd. 5. 2003, 3382–3389 Vol.5. DOI: 10.1109/VETECF.2003.1286317.

[162] Traian Pop, Paul Pop, Petru Eles, Zebo Peng und Alexandru Andrei. „Timing analysis of the FlexRay communication protocol“. In: *Real-time systems* 39.1-3 (2008), S. 205–235. DOI: 10.1007/s11241-007-9040-3.

[163] G Leen und D Heffernan. „TTCAN: a new time-triggered controller area network“. In: *Microprocessors and Microsystems* 26.2 (2002), S. 77–94. DOI: 10.1016/S0141-9331(01)00148-X.

[164] L. Almeida, P. Pedreiras und J. A. G. Fonseca. „The FTT-CAN protocol: why and how“. In: *IEEE Transactions on Industrial Electronics* 49.6 (2002), S. 1189–1201. DOI: 10.1109/TIE.2002.804967.

[165] ISO Central Secretary. *Road vehicles — Controller area network (CAN) — Part 4: Time-triggered communication*. en. Standard ISO TR 11898-4:2004. International Organization for Standardization, 2004. URL: https://www.iso.org/standard/36306.html.

[166] S. C. Talbot und S. Ren. „Comparision of FieldBus Systems CAN, TTCAN, FlexRay and LIN in Passenger Vehicles“. In: *2009 29th IEEE International Conference on Distributed Computing Systems Workshops*. 2009, S. 26–31. DOI: 10.1109/ICDCSW.2009.15.

[167] Stéphane Magnenat, Valentin Longchamp und F. Mondada. „ASEBA, an event-based middleware for distributed robot control“. In: IEEE Press, 2007. URL: http://infoscience.epfl.ch/record/111860.

[168] A. Kampmann, A. Wüstenberg, B. Alrifaee und S. Kowalewski. „A Portable Implementation of the Real-Time Publish-Subscribe Protocol for Microcontrollers in Distributed Robotic Applications“. In: *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*. Okt. 2019, S. 443–448. DOI: 10.1109/ITSC.2019.8916835.

[169] J. Farkas, L. L. Bello und C. Gunther. „Time-Sensitive Networking Standards“. In: *IEEE Communications Standards Magazine* 2.2 (2018), S. 20–21. DOI: 10.1109/MCOMSTD.2018.8412457.

[170] Haider Mshali, Tayeb Lemlouma, Maria Moloney und Damien Magoni. „A survey on health monitoring systems for health smart homes“. In: *International Journal of Industrial Ergonomics* 66 (2018), S. 26–56. DOI: 10.1016/j.ergon.2018.02.002.

[171] Jinseong Lee und Jaiyong Lee. „Prediction-based energy saving mechanism in 3GPP NB-IoT networks“. In: *Sensors* 17.9 (2017), S. 2008. DOI: 10.3390/s17092008.

[172] Jetmir Haxhibeqiri, Eli De Poorter, Ingrid Moerman und Jeroen Hoebeke. „A survey of LoRaWAN for IoT: From technology to application“. In: *Sensors* 18.11 (2018), S. 3995.

[173] R. Ratasuk, B. Vejlgaard, N. Mangalvedhe und A. Ghosh. „NB-IoT system for M2M communication“. In: *2016 IEEE Wireless Communications and Networking Conference*. 2016, S. 1–5. DOI: 10.1109/WCNC.2016.7564708.

[174] J. G. Andrews, S. Buzzi, W. Choi, S. V. Hanly, A. Lozano, A. C. K. Soong und J. C. Zhang. „What Will 5G Be?“ In: *IEEE Journal on Selected Areas in Communications* 32.6 (2014), S. 1065–1082. DOI: 10.1109/JSAC.2014.2328098.

[175] Jasenka Dizdarević, Francisco Carpio, Admela Jukan und Xavi Masip-Bruin. „A Survey of Communication Protocols for Internet of Things and Related Challenges of Fog and Cloud Computing Integration“. In: *ACM Comput. Surv.* 51.6 (Jan. 2019). DOI: 10.1145/3292674.

[176] Michael Koster, Ari Keranen und Jaime Jimenez. „Publish-subscribe broker for the constrained application protocol (CoAP)“. Version draft-ietf-core-coap-pubsub-09. In: (2019). draft. URL: https://datatracker.ietf.org/doc/draft-ietf-core-coap-pubsub/.

[177] U. Hunkeler, H. L. Truong und A. Stanford-Clark. „MQTT-S — A publish/subscribe protocol for Wireless Sensor Networks“. In: *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE '08)*. 2008, S. 791–798. DOI: 10.1109/COMSWA.2008.4554519.

[178] N. Naik. „Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP“. In: *2017 IEEE International Systems Engineering Symposium (ISSE)*. 2017, S. 1–7. DOI: 10.1109/SysEng.2017.8088251.

[179] J. Pojda, A. Wolff, M. Sbeiti und C. Wietfeld. „Performance analysis of mesh routing protocols for UAV swarming applications“. In: *2011 8th International Symposium on Wireless Communication Systems*. 2011, S. 317–321. DOI: 10.1109/ISWCS.2011.6125375.

[180] Z. Yuan, J. Jin, L. Sun, K. Chin und G. Muntean. „Ultra-Reliable IoT Communications with UAVs: A Swarm Use Case“. In: *IEEE Communications Magazine* 56.12 (2018), S. 90–96. DOI: 10.1109/MCOM.2018.1800161.

[181] J. Wang und E. Olson. „AprilTag 2: Efficient and robust fiducial detection“. In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016, S. 4193–4198. DOI: 10.1109/IROS.2016.7759617.

[182] Lih-Chang Lin und Hao-Yin Shih. „Modeling and adaptive control of an omni-mecanum-wheeled robot“. In: (2013). DOI: 10.4236/ica.2013.42021.

[183] C. Röhrig, D. Heß und F. Künemund. „Motion controller design for a mecanum wheeled mobile manipulator“. In: *2017 IEEE Conference on Control Technology and Applications (CCTA)*. 2017, S. 444–449. DOI: 10.1109/CCTA.2017.8062502.

[184] F. Künemund, D. Heß und C. Röhrig. „Energy Efficient Kinodynamic Motion Planning for Holonomic AGVs in Industrial Applications using State Lattices“. In: *Proceedings of ISR 2016: 47st International Symposium on Robotics*. 2016, S. 1–8.

[185] Markus Tresch. „Middleware: Schlüsseltechnologie zur Entwicklung verteilter Informationssysteme“. In: *Informatik-Spektrum* 19.5 (1996), S. 249–256. DOI: 10.1007/s002870050035.

[186] Letha Hughes Etzkorn. *Introduction to Middleware: Web Services, Object Components, and Cloud Computing*. CRC Press, 2017. ISBN: 978-0367573591.

[187] Wilfried Voss. *A Comprehensible Guide to Controller Area Network*. Copperhill Media, 2008. ISBN: 0-9765116-0-6.

[188] Dipa Soni und Ashwin Makwana. „A Survey on MQTT: A Protocol of Internet of Things (IoT)“. In: *International Conference On Telecommunication, Power Analysis And Computing Techniques (ICTPACT-2017)*. 2017.

[189] Roger A. Light. „Mosquitto: server and client implementation of the MQTT protocol“. In: *Journal of Open Source Software* 2.13 (2017), S. 265. DOI: 10.21105/joss.00265.

[190] R. Banno, J. Sun, M. Fujita, S. Takeuchi und K. Shudo. „Dissemination of edge-heavy data on heterogeneous MQTT brokers“. In: *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*. 2017, S. 1–7. DOI: 10.1109/CloudNet.2017.8071523.

[191] Nicolas Chan. „A Resource Utilization Analytics Platform Using Grafana and Telegraf for the Savio Supercluster“. In: *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*. PEARC '19. Chicago, IL, USA: Association for Computing Machinery, 2019. ISBN: 9781450372275. DOI: 10.1145/3332186.3333053.

[192] Fei Guan, Long Peng, Luc Perneel und Martin Timmerma. „Open source FreeRTOS as a case study in real-time operating system evolution“. In: *Journal of Systems and Software* 118 (2016), 19–35. DOI: 10.1016/j.jss.2016.04.063.

[193] Pasquale Buonocunto, Alessandro Biondi, Marco Pagani, Mauro Marinoni und Giorgio Buttazzo. „ARTE: Arduino Real-Time Extension for Programming Multitasking Applications“. In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. SAC '16. Pisa, Italy: Association for Computing Machinery, 2016, S. 1724–1731. DOI: 10.1145/2851613.2851672.

[194] Adam Dunkels, Oliver Schmidt, Thiemo Voigt und Muneeb Ali. „Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems“. In: *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*. SenSys '06. Boulder, Colorado, USA: Association for Computing Machinery, 2006, S. 29–42. DOI: 10.1145/1182807.1182811.

[195] Christian Charreyre. „Using Yocto Project to build rich and reliable embedded Linux distributions“. In: *Embedded Real Time Software (ERTS'14)*. TOULOUSE, France, Feb. 2014. URL: https://hal.archives-ouvertes.fr/hal-02271300.

[196] Federico Reghenzani, Giuseppe Massari und William Fornaciari. „The Real-Time Linux Kernel: A Survey on PREEMPT_RT“. In: *ACM Comput. Surv.* 52.1 (Feb. 2019). DOI: 10.1145/3297714.

[197] Ulf Jennehag, Stefan Forsstrom und Federico Fiordigigli. „Low Delay Video Streaming on the Internet of Things Using Raspberry Pi“. In: *Electronics* 5.4 (Sep. 2016), S. 60. DOI: 10.3390/electronics5030060.

[198] M. Lekić und G. Gardašević. „IoT sensor integration to Node-RED platform“. In: *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)*. 2018, S. 1–5. DOI: 10.1109/INFOTEH.2018.8345544.

[199] Brian McBride und Dave Reynolds. „Survey of time series database technology“. UKCEH reviewers: Matt Fry, Oliver Swain, Simon Stanley, Mike Brown. Wallingford, März 2020. URL: http://nora.nerc.ac.uk/id/eprint/527832/.

[200] Mohammad Nasar und Mohammad Abu Kausar. „Suitability Of Influxdb Database For Iot Applications“. In: *International Journal of Innovative Technology and Exploring Engineering* 8.10 (2019), S. 1850–1857. ISSN: 2278-3075.

[201] Philippe Bonnet, Johannes Gehrke und Praveen Seshadri. „Towards Sensor Database Systems“. In: *Mobile Data Management*. Hrsg. von Kian-Lee Tan, Michael J. Franklin und John Chi-Shing Lui. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, S. 3–14. ISBN: 978-3-540-44498-5.

[202] Thomas Alfons Beermann, Aleksandr Alekseev, Dario Barberis, Sabine Chrystel Crepe-Renaudin, Johannes Elmsheuser, Ivan Glushkov, Michal Svatos, Armen Vartapetian, Petr Vokac und Helmut Wolters. *Implementation of ATLAS Distributed Computing monitoring dashboards using InfluxDB and Grafana*. Techn. Ber. ATL-SOFT-PROC-2020-019. Geneva: CERN, März 2020. URL: https://cds.cern.ch/record/2712177.

[203] Vahid Garousi, Michael Felderer, Çağrı Murat Karapıçak und Uğur Yılmaz. „Testing embedded software: A survey of the literature“. In: *Information and Software Technology* 104 (2018), S. 14–45. DOI: https://doi.org/10.1016/j.infsof.2018.06.016.

[204] B. Lu, D. B. Durocher und P. Stemper. „Predictive maintenance techniques“. In: *IEEE Industry Applications Magazine* 15.6 (2009), S. 52–60. DOI: 10.1109/MIAS.2009.934444.

[205] Mohiuddin Ahmed, Abdun Naser Mahmood und Jiankun Hu. „A survey of network anomaly detection techniques“. In: *Journal of Network and Computer Applications* 60 (2016), S. 19–31. DOI: 10.1016/j.jnca.2015.11.016.

[206] Jens Ehlers, Andre van Hoorn, Jan Waller und Wilhelm Hasselbring. „Self-Adaptive Software System Monitoring for Performance Anomaly Localization“. In: *Proceedings of the 8th ACM International Conference on Autonomic Computing*. ICAC '11. Karlsruhe, Germany: Association for Computing Machinery, 2011, S. 197–200. DOI: 10.1145/1998582.1998628.

[207] P. R. Lewis, A. Chandra, S. Parsons, E. Robinson, K. Glette, R. Bahsoon, J. Torresen und X. Yao. „A Survey of Self-Awareness and Its Application in Computing Systems“. In: *2011 Fifth IEEE Conference on Self-Adaptive and Self-Organizing Systems Workshops*. 2011, S. 102–107. DOI: 10.1109/SASOW.2011.25.

[208] Claire Maiza, Hamza Rihani, Juan M. Rivas, Joël Goossens, Sebastian Altmeyer und Robert I. Davis. „A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems“. In: *ACM Comput. Surv.* 52.3 (Juni 2019). DOI: 10.1145/3323212.

[209] Adrian Francalanza, Jorge A. Pérez und César Sánchez. „Runtime Verification for Decentralised and Distributed Systems“. In: *Lectures on Runtime Verification: Introductory and Advanced Topics*. Hrsg. von Ezio Bartocci und Yliès Falcone. Cham: Springer International Publishing, 2018, S. 176–210. DOI: 10.1007/978-3-319-75632-5_6.

[210] A. Bodensohn, M. Haueis, R. Mäckel, M. Pulvermüller und T. Schreiber. „System Monitoring for Lifetime Prediction in Automotive Industry“. In: *Advanced Microsystems for Automotive Applications 2005*. Hrsg. von Jürgen Valldorf und Wolfgang Gessner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, S. 149–158. DOI: 10.1007/3-540-27463-4_11.

[211] A. Pandian und A. Ali. „A review of recent trends in machine diagnosis and prognosis algorithms“. In: *2009 World Congress on Nature Biologically Inspired Computing (NaBIC)*. 2009, S. 1731–1736. DOI: 10.1109/NABIC.2009.5393625.

[212] F. Giobergia, E. Baralis, M. Camuglia, T. Cerquitelli, M. Mellia, A. Neri, D. Tricarico und A. Tuninetti. „Mining Sensor Data for Predictive Maintenance in the Automotive Industry“. In: *2018 IEEE 5th International Conference on Data Science and Advanced Analytics (DSAA)*. 2018, S. 351–360. DOI: 10.1109/DSAA.2018.00046.

[213] Edward Clarkson und Ronald C. Arkin. „Applying heuristic evaluation to human-robot interaction systems“. In: *Proceedings of the Twentieth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2007* (2007), S. 44–49.

[214] Mohan Rajesh Elara, CA Acosta Calderon, Changjiu Zhou, Pik Kong Yue und Lingyun Hu. „Using heuristic evaluation for human-humanoid robot interaction in the soccer robotics domain“. In: *IEEE-RAS International Conference on Humanoid Robots (Humanoids 2007), Pittsburgh, USA, Nov*. 2007.

[215] Brenden Keyes, Mark Micire, Jill L. und Holly A. „Improving Human-Robot Interaction through Interface Evolution“. In: *Human-Robot Interaction* February (2010). DOI: 10.5772/8140.

[216] Daniel Labonte, Patrick Boissy und François Michaud. „Comparative analysis of 3-D robot teleoperation interfaces with novice users“. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics* 40.5 (2010), S. 1331–1342. DOI: 10.1109/TSMCB.2009.2038357.

[217] David Harrington, Randy Presuhn und Bert Wijnen. *RFC3411: An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks*. 2002.

[218] Christos Emmanouilidis, Erkki Jantunen und John MacIntyre. „Flexible software for condition monitoring, incorporating novelty detection and diagnostics“. In: *Computers in Industry* 57.6 (2006). E-maintenance Special Issue, S. 516–527. DOI: 10.1016/j.compind.2006.02.012.

[219] Wenjun Xi, Yuguang Feng, Yu Zhou und Bo Xu. „Condition monitoring and plenary diagnostics strategy based on event driven threads“. In: *2008 International Conference on Condition Monitoring and Diagnosis*. 2008, S. 576–578. DOI: 10.1109/CMD.2008.4580353.

[220] Vladimir Rocha und Anarosa Alves Franco Brandão. „A scalable multiagent architecture for monitoring IoT devices“. In: *Journal of Network and Computer Applications* 139 (2019), S. 1–14. DOI: 10.1016/j.jnca.2019.04.017.

[221] Aldiyar Salkenov und Susmit Bagchi. „Cloud based autonomous monitoring and administration of heterogeneous distributed systems using mobile agents“. In: *Future Generation Computer Systems* 99 (2019), S. 527–557. DOI: 10.1016/j.future.2019.04.047.

[222] André van Hoorn, Jan Waller und Wilhelm Hasselbring. „Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis“. In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ICPE '12. Boston, Massachusetts, USA: Association for Computing Machinery, 2012, S. 247–248. ISBN: 9781450312028. DOI: 10.1145/2188286.2188326.

[223] Dirk Helbing. „Agent-based modeling“. In: *Social self-organization*. Springer, 2012, S. 25–70. DOI: 10.1007/978-3-642-24004-1.

[224] Larry Bunch, Maggie Breedy, Jeffrey M. Bradshaw, Marco Carvalho, Niranjan Suri, Andrzej Uszok, Jack Hansen, Michal Pechoucek und Vladimir Marik. „Software Agents for Process Monitoring and Notification“. In: *Proceedings of the 2004 ACM Symposium on Applied Computing*. SAC '04. Nicosia, Cyprus: Association for Computing Machinery, 2004, S. 94–100. ISBN: 1581138121. DOI: 10.1145/967900.967921.

[225] F. M. David und R. H. Campbell. „Building a Self-Healing Operating System“. In: *Third IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC 2007)*. 2007, S. 3–10. DOI: 10.1109/DASC.2007.22.

[226] Gao, Lei and Huang, Jia and Ceng, Jianjiang and Leupers, Rainer and Ascheid, Gerd and Meyr, Heinrich. „TotalProf: A Fast and Accurate Retargetable Source Code Profiler“. In: *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '09. Grenoble, France: Association for Computing Machinery, 2009, S. 305–314. DOI: 10.1145/1629435.1629477.

[227] D. Tukymbekov, A. Saymbetov, M. Nurgaliyev, N. Kuttybay, Y. Nalibayev und G. Dosymbetova. „Intelligent energy efficient street lighting system with predictive energy consumption“. In: *2019 International Conference on Smart Energy Systems and Technologies (SEST)*. 2019, S. 1–5. DOI: 10.1109/SEST.2019.8849023.

[228] A. Patidar und U. Suman. „A survey on software architecture evaluation methods“. In: *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*. 2015, S. 967–972.

[229] L. Dobrica und E. Niemela. „A survey on software architecture analysis methods“. In: *IEEE Transactions on Software Engineering* 28.7 (2002), S. 638–653. DOI: 10.1109/TSE.2002.1019479.

[230] M. A. Babar und I. Gorton. „Comparison of scenario-based software architecture evaluation methods“. In: *11th Asia-Pacific Software Engineering Conference*. 2004, S. 600–607. DOI: 10.1109/APSEC.2004.38.

Verzeichnis der Online Referenzen

[O1] *IEEE Xplore - Advanced Search*. IEEE. 2020. URL: https://ieeexplore.ieee.org/search/advanced (besucht am 26. 08. 2020).

[O2] *NAO 6*. SoftBank Robotics. 2020. URL: https://www.softbankrobotics.com/emea/en/nao (besucht am 25. 08. 2020).

[O3] *S6 MaxV*. Roborock. 2020. URL: https://us.roborock.com/pages/roborock-s6-maxv (besucht am 25. 08. 2020).

[O4] arm Community. *ARMv6-M vs ARMv7-M - Unpacking the Microcontrollers*. 2013. URL: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/armv6-m-vs-armv7-m---unpacking-the-microcontrollers (besucht am 13. 02. 2020).

[O5] Smart Wireless Computing. *INFORCE 6640™ SINGLE BOARD COMPUTER (SBC)*. 2018. URL: https://www.inforcecomputing.com/products/single-board-computers-sbc/qualcomm-snapdragon-820-inforce-6640-sbc (besucht am 14. 02. 2020).

[O6] *CPU und GPU: beide optimal einsetzen*. intel Cooperation. 2020. URL: https://www.intel.de/content/www/de/de/products/docs/processors/cpu-vs-gpu.html (besucht am 21. 09. 2020).

[O7] *Raspberry Pi 3 Model B+*. Raspberry Pi Foundation. 2020. URL: https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/ (besucht am 01. 10. 2020).

[O8] Heinz Nixdorf Institut Universtät Paderborn. *SONDERFORSCHUNGSBEREICH 614*. 2020. URL: https://www.hni.uni-paderborn.de/forschung/schwerpunktprojekte/abgeschlossen/sonderforschungsbereich-614/ (besucht am 24. 02. 2020).

[O9] *ROS 2*. GitHub Inc. 2021. URL: https://github.com/ros2 (besucht am 26. 02. 2021).

[O10] AUTOSAR. *Classic Platform - AUTOSAR*. 2020. URL: https://www.autosar.org/standards/classic-platform/ (besucht am 04. 03. 2020).

[O11] IEEE. *What Is a Robot?: ROBOTS: Your Guide to the World of Robotics*. 2020. URL: https://robots.ieee.org/learn/what-is-a-robot/ (besucht am 10. 03. 2020).

[O12] *Google Scholar - Advanced Search*. Google LLC. 2020. URL: https://scholar.google.com/?hl=en&as_sdt=0,5#d=gs_asd (besucht am 26. 08. 2020).

[O13] Boston Dynamics. *Spot Boston Dynamics*. 2020. URL: https://www.bostondynamics.com/spot (besucht am 10. 03. 2020).

[O14] Inc. Open Source Robotics Foundation. *Turtlebot*. 2017. URL: https://www.turtlebot.com/ (besucht am 13. 03. 2020).

[O15] *MATLAB*. The Mathworks Inc. 2020. URL: https://www.mathworks.com/products/matlab.html (besucht am 06. 10. 2020).

[O16] *Simulation and Model-Based Design*. The Mathworks Inc. 2020. URL: https://www.mathworks.com/products/simulink.html (besucht am 06. 10. 2020).

[O17] IEEE Spectrum. *Boston Dynamics' Spot Robot Dog Goes on Sale*. 2019. URL: https://spectrum.ieee.org/automaton/robotics/industrial-robots/boston-dynamics-spot-robot-dog-goes-on-sale (besucht am 12. 03. 2020).

[O18] *BeagleBoard bone*. The BeagleBoard.org Foundation. 2020. URL: https://beagleboard.org/bone (besucht am 13. 10. 2020).

[O19] GCtronic. *e-puck2*. 2018. URL: https://www.gctronic.com/doc/index.php/e-puck2 (besucht am 20. 03. 2020).

[O20] *Create 2 Robot*. iRobot Corp. 2020. URL: https://edu.irobot.com/what-we-offer/create-robot (besucht am 13. 10. 2020).

[O21] Arduino. *Arduino Home*. 2020. URL: https://www.arduino.cc/ (besucht am 19. 03. 2020).

[O22] IEEE. *Types of Robots: ROBOTS: Your Guide to the World of Robotics*. 2020. URL: https://robots.ieee.org/learn/types-of-robots/ (besucht am 10. 03. 2020).

[O23] SAE International. *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. 15. Juni 2018. URL: https://www.sae.org/standards/content/j3016%5C_201806 (besucht am 11. 08. 2020).

[O24] *Arduino Mega 2560 Rev3*. Arduino CC. 2020. URL: https://store.arduino.cc/arduino-mega-2560-rev3 (besucht am 01. 10. 2020).

[O25] *Intel NUC-Kit NUC8i5BEK*. intel. 2020. URL: https://ark.intel.com/content/www/de/de/ark/products/126147/intel-nuc-kit-nuc8i5bek.html (besucht am 01. 10. 2020).

[O26] *Arduino: Power Consumption Compared*. Thomas Lextrait. 2016. URL: https://tlextrait.svbtle.com/arduino-power-consumption-compared (besucht am 01. 10. 2020).

[O27] *Coffee Lake i5 NUC Review (NUC8i5BEK / NUC8i5BEH)*. The NUC Blog. 2020. URL: https://nucblog.net/2018/11/coffee-lake-i5-nuc-review-nuc8i5bek-nuc8i5beh/3/ (besucht am 01. 10. 2020).

[O28] *Discovery kit with STM32F303VC MCU*. STMicroelectronics. 2020. URL: https://www.st.com/en/evaluation-tools/stm32f3discovery.html (besucht am 01. 10. 2020).

[O29] *ChibiOS Homepage*. chibiOS. 2020. URL: https://www.chibios.org/ (besucht am 01. 10. 2020).

[O30] *FreeRTOS*. Amazon. 2020. URL: https://www.freertos.org/ (besucht am 01. 10. 2020).

[O31] *Ten millionth Raspberry Pi, and a new kit*. Raspberry Pi Foundation. 2016. URL: https://www.raspberrypi.org/blog/ten-millionth-raspberry-pi-new-kit/ (besucht am 02. 10. 2020).

[O32] *Kinect teardown: two cameras, four microphones, 12 watts of power, no controller*. Verizon Media. 2010. URL: https://www.engadget.com/2010-11-04-kinect-teardown-two-cameras-four-microphones-12-watts-of-powe.html (besucht am 05. 10. 2020).

[O33] *Rust Programming Language*. Rust Project Developers. 2020. URL: https://www.rust-lang.org/ (besucht am 06. 10. 2020).

[O34] *Model and simulate decision logic using state machines and flow charts*. The Mathworks Inc. 2020. URL: https://www.mathworks.com/products/stateflow.html (besucht am 06. 10. 2020).

[O35] *Empowering App Development for Developers*. Docker Inc. 2020. URL: https://www.docker.com/ (besucht am 07. 10. 2020).

[O36] *.NET | Cross-platofrm. Open Source.* Microsoft Cooperation. 2020. URL: https://dotnet.microsoft.com/ (besucht am 07. 10. 2020).

[O37] *ZynqBerry Module with Xilinx Zynq-7010 in Raspberry Pi Form Faktor*. trenz electronic. 2020. URL: https://shop.trenz-electronic.de/en/TE0726-03M-ZynqBerry-Module-with-Xilinx-Zynq-7010-in-Raspberry-Pi-Form-Faktor (besucht am 02. 11. 2020).

[O38] *(4 INCH) 100MM MECANUM WHEEL RIGHT/BEARING ROLLERS 14094R*. Nexus Robot. 2020. URL: http://www.nexusrobot.com/product/4-inch-100mm-mecanum-wheel-rightbearing-rollers-14094r.html (besucht am 05. 11. 2020).

[O39] *DC-Kleinstmotoren Serie 2642 ... CR*. DR. FRITZ FAULHABER GMBH & CO. KG. 2020. URL: https://www.faulhaber.com/de/produkte/serie/2642cr/ (besucht am 05. 11. 2020).

[O40] *HEDM-5500 J02*. Mouser Electronics. 2020. URL: https://www.mouser.de/ProductDetail/Broadcom-Avago/HEDM-5500J02?qs=RuhU64sK2%252BvVO9yTwp2Qow%3D%3D (besucht am 05. 11. 2020).

[O41] *HC-SR04*. Mouser Electronics. 2020. URL: https://www.mouser.de/ProductDetail/OSEPP-Electronics/HC-SR04?qs=sGAEpiMZZMu3sxpa5v1qrp3Jl9HJt%252BrQXKKGUGBAP6Q%3D (besucht am 05. 11. 2020).

[O42] *Abgeschlossene Projekte*. Fachhochschule Dortmund. 2021. URL: https://www.fh-dortmund.de/de/ftransfer/profil/abgeschlosseneprojekte.php (besucht am 17. 02. 2021).

[O43] *4WD MECANUM WHEEL MOBILE ARDUINO ROBOTICS CAR 10011*. Nexus Robot. 2020. URL: https://www.nexusrobot.com/product/4wd-mecanum-wheel-mobile-arduino-robotics-car-10011.html (besucht am 03. 11. 2020).

[O44] *Conrad energy Modellbau-Akkupack*. Conrad Electronic SE. 2020. URL: https://www.conrad.de/de/p/conrad-energy-modellbau-akkupack-lipo-14-8-v-5500-mah-zellen-zahl-4-20-c-softcase-xt90-1344151.html (besucht am 05. 11. 2020).

[O45] *InfluxDB 1.X*. InfluxData Inc. 2020. URL: https://www.influxdata.com/time-series-platform/ (besucht am 03. 12. 2020).

[O46] *InfluxDB line protocoll tutorial*. InfluxData Inc. 2020. URL: https://docs.influxdata.com/influxdb/v1.8/write_protocols/line_protocol_tutorial/ (besucht am 03. 12. 2020).

[O47] *MQTT Consumer Input Plugin*. GitHub Inc. 2020. URL: https://github.com/influxdata/telegraf/tree/master/plugins/inputs/mqtt_consumer (besucht am 03. 12. 2020).

[O48] *Raspberry Pi Support from Simulink*. The Mathworks Inc. 2020. URL: https://www.mathworks.com/hardware-support/raspberry-pi-simulink.html (besucht am 16. 11. 2020).

[O49] *CAN-BUS Shield*. SparkFun Electronics. 2020. URL: https://www.sparkfun.com/products/13262 (besucht am 10. 12. 2020).

[O50] *DEBO BME280*. reichelt elektronik GmbH & Co. KG. 2020. URL: https://www.reichelt.de/entwicklerboards-temperatur-feuchtigkeits-und-drucksensor--debo-bme280-p253982.html?&nbc=1 (besucht am 10. 12. 2020).

[O51] *SS49E*. Mouser Electronics. 2020. URL: https://www.mouser.de/ProductDetail/?qs=%2Ffq2y7sSKcJBD3o5K2Vcgg== (besucht am 11. 12. 2020).

[O52] *G5V-1-DC5*. Mouser Electronics. 2020. URL: https://www.mouser.de/ProductDetail/Omron-Electronics/G5V-1-DC5?qs=Pjd0UV7BHP%2FGqZ0zy8DG3Q== (besucht am 10. 12. 2020).

[O53] *Camera Module V2*. Raspberry Pi Foundation. 2020. URL: https://www.raspberrypi.org/products/camera-module-v2/ (besucht am 08. 12. 2020).

[O54] *Microchip Technology MCP2515-I/SO*. Mouser Electronics. 2020. URL: https://www.mouser.de/ProductDetail/Microchip-Technology/MCP2515-I-SO?qs=KwArPi4cUogGDbGnphOvtQ%3D%3D (besucht am 08. 12. 2020).

[O55] *Raspberry Pi OS*. Raspberry Pi Foundation. 2020. URL: https://www.raspberrypi.org/software/ (besucht am 08. 12. 2020).

[O56] *Yocto Project*. Yocto Project. 2020. URL: https://www.yoctoproject.org/ (besucht am 08. 12. 2020).

[O57] *OpenKinect/libfreenect*. GitHub Inc. 2020. URL: https://github.com/OpenKinect/libfreenect (besucht am 08. 12. 2020).

[O58] *raspberrypi/userland*. GitHub Inc. 2020. URL: https://github.com/raspberrypi/userland (besucht am 08. 12. 2020).

[O59] *FFmpeg*. Fabrice Bellard. 2020. URL: https://ffmpeg.org/ (besucht am 08. 12. 2020).

[O60] *linux-can/can-utils*. GitHub Inc. 2020. URL: https://github.com/linux-can/can-utils (besucht am 08. 12. 2020).

[O61] *libusb/libusb*. GitHub Inc. 2020. URL: https://github.com/libusb/libusb (besucht am 08. 12. 2020).

[O62] *Raspberry Pi Shield - Display LCD-Touch, 7", 800x480 Pixel*. reichelt elektronik GmbH & Co. KG. 2020. URL: https://www.reichelt.de/raspberry-pi-shield-display-lcd-touch-7-800x480-pixel-raspberry-pi-7td-p159859.html?&nbc=1 (besucht am 14. 12. 2020).

[O63] *Node-RED*. OpenJS Foundation. 2020. URL: https://nodered.org/ (besucht am 15. 12. 2020).

[O64] *PetaLinux Tools*. Xilinx Inc. 2020. URL: https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html (besucht am 16. 12. 2020).

[O65] *Grafana Features*. Grafana Labs. 2020. URL: https://grafana.com/grafana/ (besucht am 18. 12. 2020).

[O66] *EnricSala/influxdb-matlab*. GitHub Inc. 2020. URL: https://github.com/EnricSala/influxdb-matlab (besucht am 22. 12. 2020).

[O67] *SICP KI4AS*. Universität Kiel. 2020. URL: https://www.sicp.de/projekte/ki4as/ (besucht am 30. 12. 2020).

[O68] *Softwaretools zur Remoteverwaltung*. SolarWinds Worldwide LLC. 2020. URL: https://www.solarwinds.com/de/engineers-toolset (besucht am 29. 12. 2020).

[O69] *PRTG Network Monitor*. Paessler AG. 2020. URL: https://www.de.paessler.com/prtg (besucht am 29. 12. 2020).

[O70] *Nagios XI*. Nagios Enterprises LLC. 2020. URL: https://www.nagios.com/products/nagios-xi/ (besucht am 29. 12. 2020).

[O71] *Kieker Application Performance Monitoring*. Universität Kiel. 2020. URL: http://kieker-monitoring.net/ (besucht am 30. 12. 2020).

[O72] *DevOps Platform GitLab*. GitLab Inc. 2021. URL: https://about.gitlab.com/ (besucht am 27. 01. 2021).

[O73] *Kanban-Boards*. Kanbanize. 2021. URL: https://kanbanize.com/de/kanban-boards-de (besucht am 27. 01. 2021).

Eigene Veröffentlichungen

[A1] Carsten Wolff, Christopher Brink, Robert Höttger, Burkhard Igel, Erik Kamsties, Lukas Krawczyk und **Uwe Lauschner**[1]. „Automotive Software Development With AMALTHEA“. In: *International Scientific Conference on Project Management in the Baltic Countries*. Riga, Latvia, 2015, S. 432–442.

[A2] Carsten Wolff, Lukas Krawczyk, Robert Hottger, Christopher Brink, **Uwe Lauschner**, Daniel Fruhner, Erik Kamsties und Burkhard Igel. „AMALTHEA - Tailoring tools to projects in automotive software development“. In: *Proceedings of the 2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, IDAACS 2015*. Bd. 2. September. Warsaw, Poland, 2015, S. 515–520. ISBN: 9781467383615. DOI: 10.1109/IDAACS.2015.7341359.

[A3] **Uwe Lauschner**, Burkhard Igel, Lukas Krawczyk und Carsten Wolff. „Applying model-based principles on a distributed robotic system application“. In: *Proceedings of the 2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, IDAACS 2015*. Bd. 2. September. Warsaw, Poland, 2015, S. 893–897. ISBN: 9781467383615. DOI: 10.1109/IDAACS.2015.7341432.

[A4] Carsten Wolff, Lukas Krawczyk, Robert Höttger, Christopher Brink und **Uwe Lauschner**. „Parallel Software for Embedded Systems“. In: *XXXI. Kandò Conference 2015*. Budapest, Hungary, 2015.

[A5] Robert Hoettger, Lukas Krawczyk, **Uwe Lauschner**, Phil Naerdemann, Philipp Heisig, Carsten Wolff, Erik Kamsties und Burkhard Igel. „Teaching Distributed and Parallel Systems with APP4MC“. In: *Desire Symposium - International Symposium on Embedded Systems and Trends in Teaching Engineering*. 09013. Nitra, Slovenia, 2016.

[A6] Carsten Wolff, Torben Lippmann und **Uwe Lauschner**. „Systems engineering for metropolitan energy systems- Ruhrvalley“. In: *Proceedings of the 2017 IEEE 9th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, IDAACS 2017*. Bd. 1. Bucharest, Romania, 2017, S. 425–430. ISBN: 9781538606971. DOI: 10.1109/IDAACS.2017.8095117.

[A7] Carsten Wolff, Torben Lippmann und **Uwe Jahn**. „Holistic systems engineering methodology for intelligent energy systems - with a Case Study from "ruhrvalley"“. In: *Dependable IoT for Human and Industry: Modeling, Architecting, Implementation*. 2018, S. 331–350. ISBN: 9788770220132.

[1] Veröffentlichungen vor 2018 unter Geburstname *Uwe Lauschner*

[A8] **Uwe Jahn**, Carsten Wolff und Peter Schulz. „Design of an Operator-Controller Based Distributed Robotic System“. In: *Communications in Computer and Information Science*. Bd. 920. Vilnius, Lithuania, 2018, S. 59–70. ISBN: 9783319999715. DOI: 10.1007/978-3-319-99972-2_5.

[A9] **Uwe Jahn**, Carsten Wolff und Peter Schulz. „Concepts of a Modular System Architecture for Distributed Robotic Systems“. In: *Computers* 8.1 (2019), S. 25. ISSN: 2073-431X. DOI: 10.3390/computers8010025.

[A10] **Uwe Jahn** und Peter Schulz. „Modular and Distributed System Architecture for Mobile Robots (Poster)“. In: *Dortmund Applied Research and Transfer (DART) Symposium*. Dortmund, Germany, 2019.

[A11] **Uwe Jahn**, Vladimir Poliakov, Meghadoot Gardi, Peter Schulz und Carsten Wolff. „Introducing pulseAT: A tool for analyzing system utilization in distributed systems“. In: *Proceedings - 2019 20th International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2019* (2019), S. 271–276. DOI: 10.1109/PDCAT46702.2019.00057.

[A12] Merlin Stampa, Andreas Sutorma, **Uwe Jahn**, Felix Willich, Sylvia Pratzler-Wanczura, Jörg Thiem, Christof Röhrig und Carsten Wolff. „A Scenario for a Multi-UAV Mapping and Surveillance System in Emergency Response Applications“. In: *2020 IEEE 5th International Symposium on Smart and Wireless Systems within the Conferences on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS-SWS)*. 2020, S. 1–6. DOI: 10.1109/IDAACS-SWS50031.2020.9297053.

[A13] **Uwe Jahn**, Merlin Stampa, Andreas Sutorma, Felix Willich, Jörg Thiem, Christof Röhrig und Carsten Wolff. „A Recommendation for a Systems Engineering Process and System Architecture for UAS“. In: *2020 IEEE 3rd International Conference and Workshop in Óbuda on Electrical and Power Engineering (CANDO-EPE)*. 2020, S. 000091–000096. DOI: 10.1109/CANDO-EPE51100.2020.9337752.

[A14] **Uwe Jahn**, Daniel Heß, Merlin Stampa, Andreas Sutorma, Christof Röhrig, Peter Schulz und Carsten Wolff. „A Taxonomy for Mobile Robots: Types, Applications, Capabilities, Implementations, Requirements, and Challenges“. In: *Robotics* 9.4 (2020), S. 109. ISSN: 2218-6581. DOI: 10.3390/robotics9040109.

[A15] Felix Willich, Carsten Wolff, Andreas Sutorma, **Uwe Jahn** und Merlin Stampa. „Model-based Systems Engineering of an Active, Oleo-Pneumatic Damper for a CS-23 General Aviation Aircraft Landing Gear“. In: *IEEE European Technology and Engineering Management Summit (ETEMS)*. 2021.

[A16] Merlin Stampa, Andreas Sutorma, **Uwe Jahn**, Jörg Thiem, Carsten Wolff und Christof Röhrig. „Maturity Levels of Public Safety Applications using Unmanned Aerial Systems: a Review“. In: *Journal of Intelligent & Robotic Systems* (2021). DOI: 10.1007/s10846-021-01462-7.

Betreute Arbeiten

[B1] Raphael Schrader. „Developement of an Breakout-Board for an STM32F3 running on a mobile robot“. Seminararbeit. Fachhochschule Dortmund, 2016.

[B2] Gaurav Pradeep Kulkarni. „Design of Health controller for Distributed Architecture Evaluation Robot (DAEbot)“. Seminararbeit. Fachhochschule Dortmund, 2018.

[B3] Meghadoot Gardi. „Design and Development of Health Controller (Health Monitoring System) For Distributed Architecture Evaluation Bot (DAEbot)“. Seminararbeit. Fachhochschule Dortmund, 2018.

[B4] Hector Gerardo Munoz. „Implementation of an Internal Controller for a mobile robot“. Projektarbeit. Fachhochschule Dortmund, 2018.

[B5] Javier Reyes. „Concept of an FPGA based Reflective Operator Accelerator“. Seminararbeit. Fachhochschule Dortmund, 2018.

[B6] Vladimir Poliakov. „Design and Implementation of DAEbot controller layer components“. Projektarbeit. Fachhochschule Dortmund, 2018.

[B7] Meghadoot Gardi. „Design and Development of RTOS (Scheduler) framework with CPU components analysis (pulseAT implementation) of DAEbot’s Health Controller Project“. Projektarbeit. Fachhochschule Dortmund, 2018.

[B8] Javier Reyes. „Implementation of an FPGA based Reflective Operator Accelerator“. Projektarbeit. Fachhochschule Dortmund, 2018.

[B9] Hector Gerardo Munoz. „Development of a real-time software architecture for AMiRo robot based on the operator controller-module“. Masterthesis. Fachhochschule Dortmund, 2018.

[B10] Meghadoot Gardi. „Design and Implementation of a Cloud-Connectivity Framework for the DAEbot“. Masterthesis. Fachhochschule Dortmund, 2019.

[B11] Jean Herrera. „Adaption of pulseAT for AMiRo robot“. Seminararbeit. Fachhochschule Dortmund, 2021.

Verzeichnis der Anhänge

Anhang A: Suchbegriffe und Syntax der Literaturrecherche

Typ	Referenz Titel/ Bezeichnung [Quelle]	Suchmatrix
Referenzsysteme aus 3.2.3	Spot [O13]	spot AND “boston dynamics“
	Khepera IV [85]	(“Khepera IV“) AND robot
	TurtleBot3 [O14]	“turtlebot 3“ AND robot
	AMiRo [86]	amiro AND robot
	WolfBot [87]	Wolfbot AND robot
	e-puck2 [88]	(e-puck2 OR “e-puck 2“) AND robot
	OmniMan [89]	Omniman AND robot
	ArEduBot [90]	ArEduBot AND robot
	Savvy [91]	*Anzahl Zitate*
	Arduino Robot [92]	*Anzahl Zitate*
	Omnidirectional Mobile Robot [93]	*Anzahl Zitate*

Alle Suchbegriffe wurden im Juli und August 2020 angewendet

Tabelle A1: Suchmatrix zur Relevanz der ausgewählten Quellen

Betreff	Suchmatrix
Navigation	(“All Metadata“:mobile AND robot AND (navigation OR mapping OR slam OR “collision avoidance“ OR “path planning“))
Autonomie	(“All Metadata“:mobile AND robot AND (autonomy OR autonomous))
Optimierung/ Lernen	(“All Metadata“:mobile AND robot AND (learning OR optimizing))
Multi-Roboter Kooperation	(“All Metadata“:mobile AND robot AND (swarm OR “multi robot“ OR “networked robots“))
Sicherheit (Safety)	(“All Metadata“:mobile AND robot AND safety)
Mensch-Roboter Interaktion	(“All Metadata“:mobile AND robot AND (“human machine“ OR “machine human“ OR “human robot“ OR “robot human“))
Sicherheit (Security)	(“All Metadata“:mobile AND robot AND security)
Zuverlässigkeit	(“All Metadata“:mobile AND robot AND reliability)
Energieeffizienz	(“All Metadata“:mobile AND robot AND (energy AND (efficient OR efficiency)))
Benutzerfreundlichkeit	(“All Metadata“:mobile AND robot AND usability)
Selbstheilung	(“All Metadata“:mobile AND robot AND (“self healing“ OR „self repairing“))

Alle Suchbegriffe wurden im Juli und August 2020 angewendet

Tabelle A2: Suchmatrix zu *Fähigkeiten* von mobilen Robotern

Betreff	Suchmatrix
Echtzeitfähigkeit	(“All Metadata“:mobile AND robot AND (“real time“ OR realtime))
Maschinelles Lernen	(“All Metadata“:mobile AND robot AND (“neural network“ OR “neural networks“ OR “machine learning“ OR “deep learning“))
Bildverarbeitung	(“All Metadata“:mobile AND robot AND (“computer vision“ OR cv OR “object recognition“))
Cloud Computing	(“All Metadata“:mobile AND robot AND (cloud OR server))
Systemüberwachung	(“All Metadata“:mobile AND robot AND (verification OR validation OR diagnosis OR “self awareness“ OR “system monitoring“ OR “fault detection“))
Modularität	(“All Metadata“:mobile AND robot AND modular)
Modellbasierte Entwicklung	(“All Metadata“:mobile AND robot AND (“model based“ OR “model driven“))
Redundanz	(“All Metadata“:mobile AND robot AND redundancy)
Rekonfigurierbarkeit	(“All Metadata“:mobile AND robot AND (reconfiguration OR reconfigure))
Verteiltes System	(“All Metadata“:mobile AND robot AND (“distributed system“ OR “distributed systems“))
Edge Computing	(“All Metadata“:mobile AND robot AND (“edge computing“ OR “fog computing“))
Einplatinenrechner	(“All Metadata“:mobile AND robot AND (sbc OR “single board computer“))

Alle Suchbegriffe wurden im Juli und August 2020 angewendet

Tabelle A3: Suchmatrix zu *technischen Realisierungen* von mobilen Robotern

Anhang B: DAEbot Software Implementierungen

```
inline uint16_t can_code_identifier_pd(priority_can_id_t *priority_can_id
    , c_s_bit_can_id_t *c_s_bit_can_id, topic_can_id_t *topic_can_id) {
   /* Combines c_s_bit and priority bits */
   uint8_t significant_bits = *priority_can_id + (*c_s_bit_can_id * 4);
   /* Combines MSB and topic ID and returns it */
   return ((significant_bits * 0x100) + *topic_can_id);
}
```

Quelltextauszug A1: Codierung der CAN-Nachrichten-ID

```
inline can_frame_types_t can_decode_identifier_pd(canid_t *canid) {
   can_frame_types_t can_frame_types;
   /* Get the "significant bits" by bit-shifting */
   uint8_t significant_bits = *canid >> 8;
   /* CONTROLL Bit */
   if (significant_bits < 0x4)
   {
      can_frame_types.c_s_bit_id = CONTROLL_DATA_BIT;
      can_frame_types.priority_id = (priority_can_id_t)significant_bits;
      can_frame_types.topic_id = (topic_can_id_t) (*canid - (
    significant_bits * 0x100));
   }
   /* SENSOR Bit */
   else
   {
      can_frame_types.c_s_bit_id = SENSOR_DATA_BIT;
      can_frame_types.priority_id = (priority_can_id_t)(significant_bits
    - 0x4);
      can_frame_types.topic_id = (topic_can_id_t) (*canid - (
    significant_bits * 0x100));
   }
   return can_frame_types;
}
```

Quelltextauszug A2: Decodierung der CAN-Nachrichten-ID

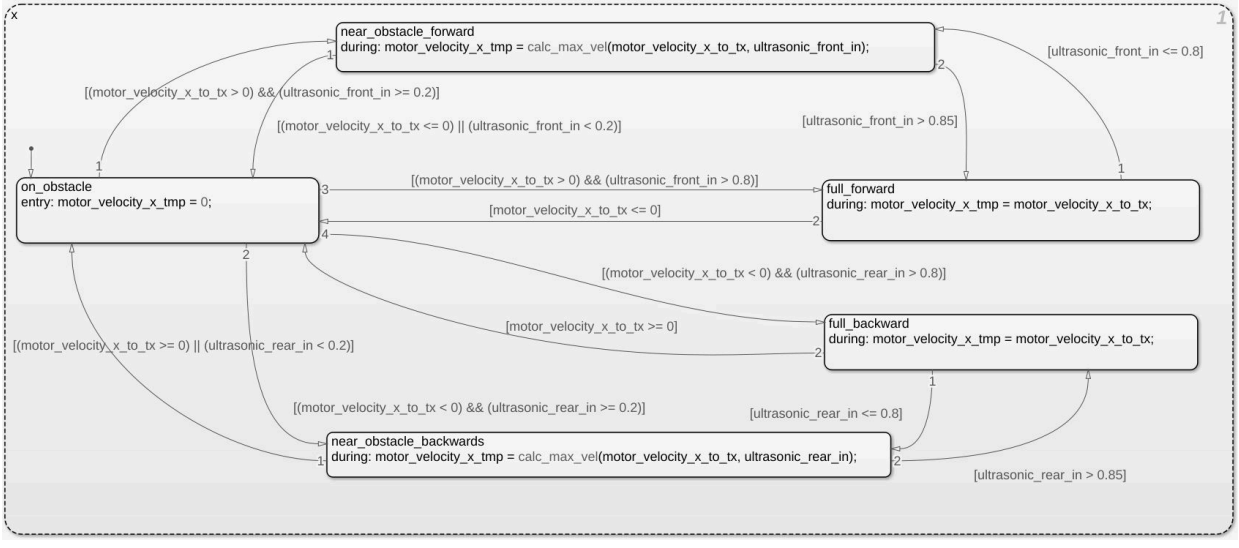


Abbildung A1: Geschwindigkeitsbegrenzung als FSM in MATLAB Simulink Stateflow

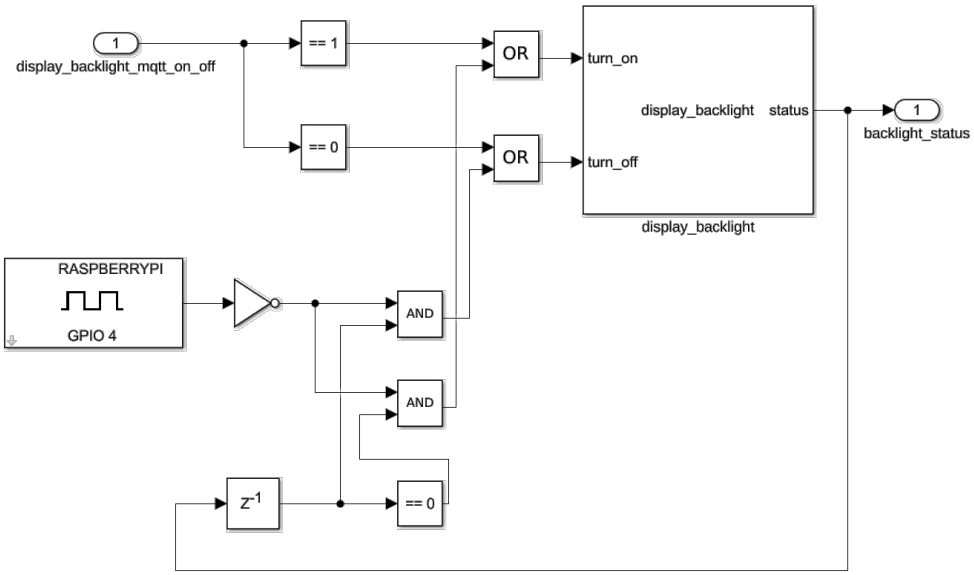
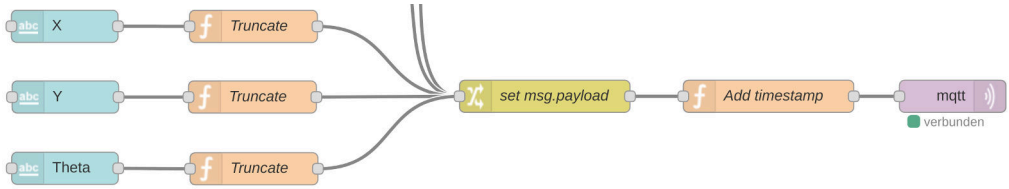
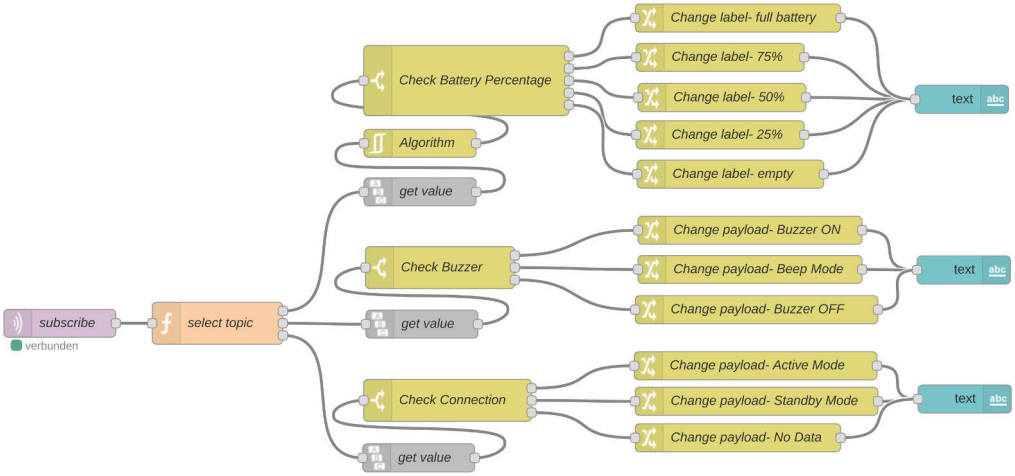


Abbildung A2: Display Hintergrundbeleuchtung Logik in MATLAB Simulink



(a) Publisher



(b) Subscriber

Abbildung A3: MQTT Implementierung in Node-RED

Anhang C: DAEbot Hardware Implementierungen

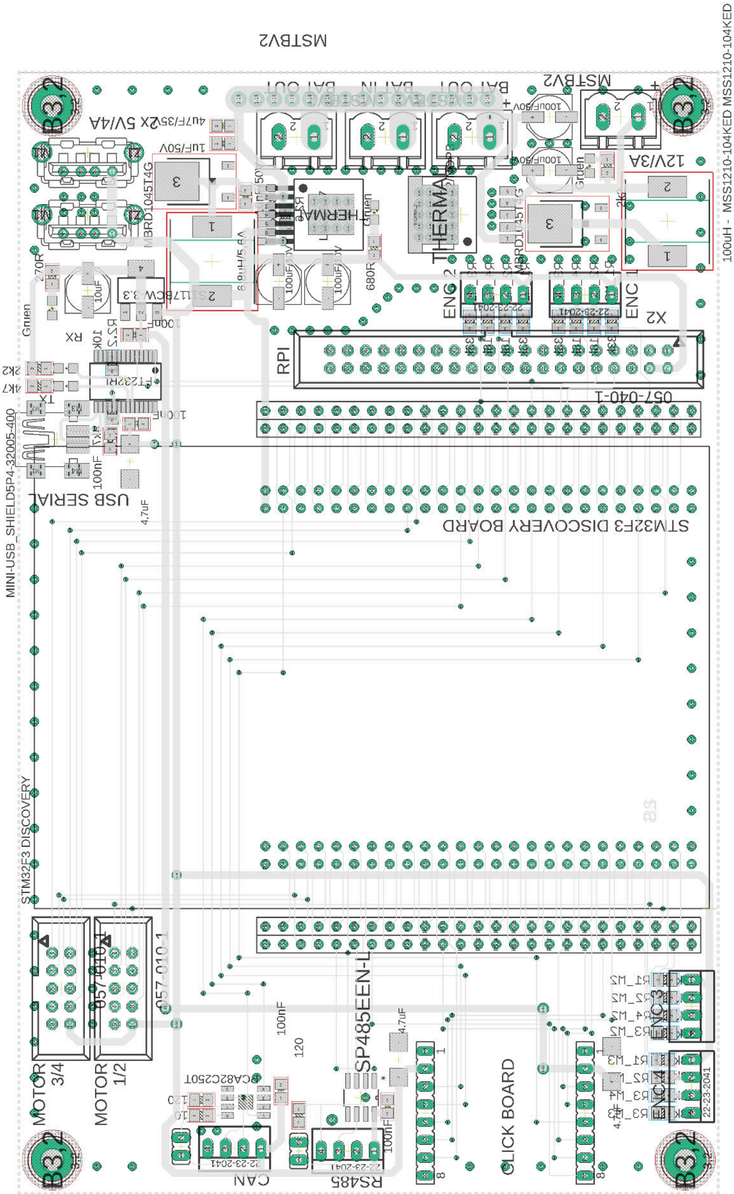


Abbildung A4: Layout des *Motion Controller* Trägerboards [B1]

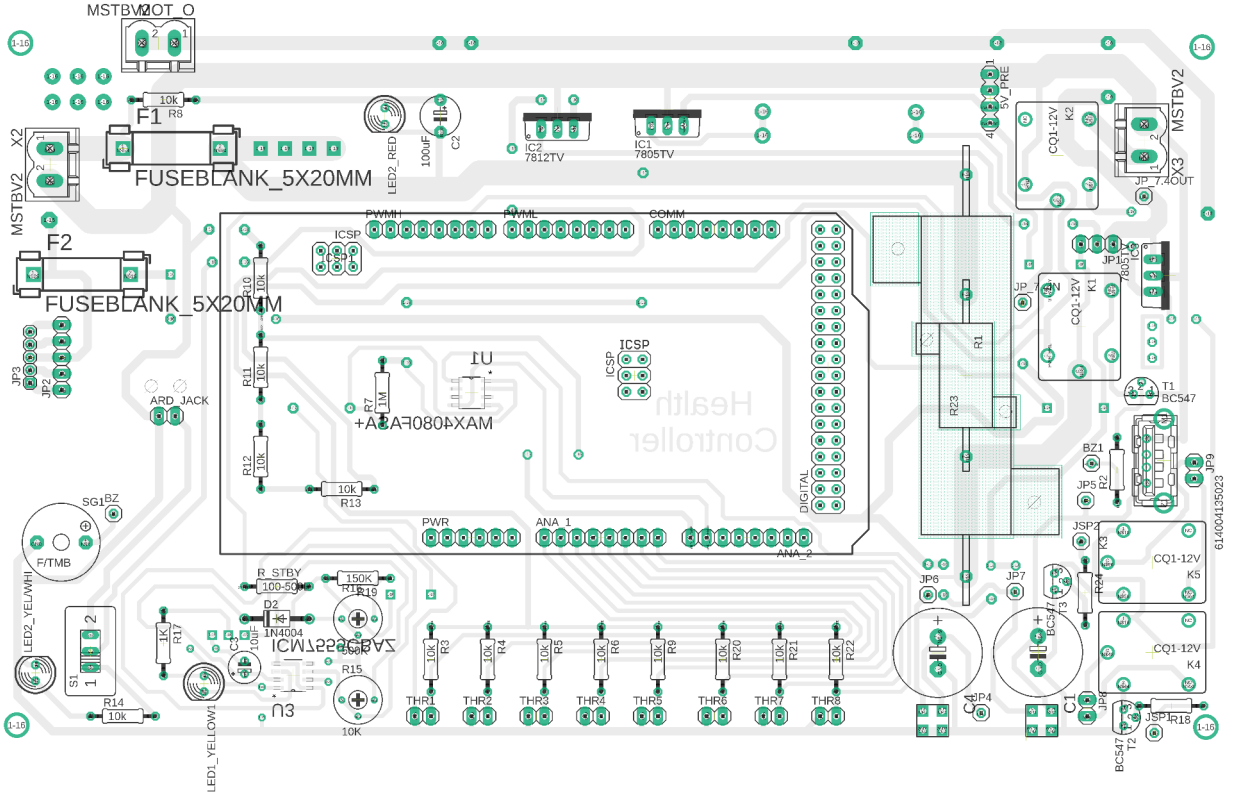


Abbildung A5: Layout des *Health Controller* Trägerboards [B2]

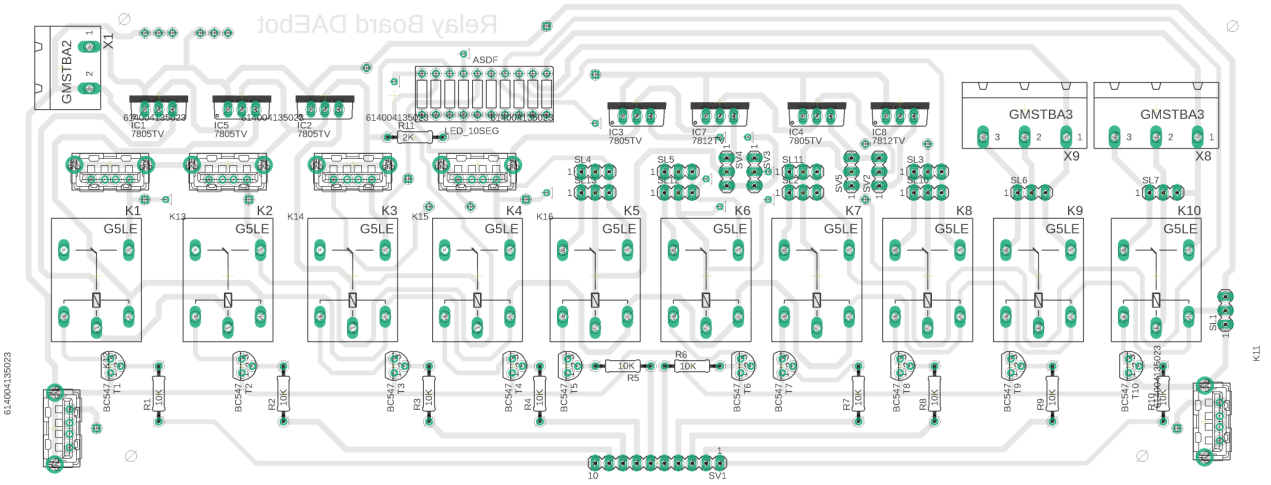


Abbildung A6: Layout des *Health Controller* Relais-Boards [B2]

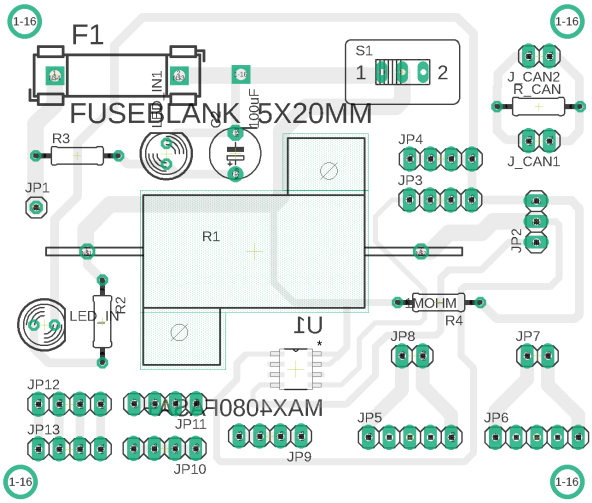


Abbildung A7: Layout des *Health Controller* Tap Boards [B2]

Anhang D: pulseAT Implementierungen

```
inline uint8_t pulseAT_get_RAM_usage() {
   uint32_t FileHandler;
   char FileBuffer[256];
   uint32_t memtotal, memfree, memavai, membuffers, memcached;
   uint8_t ram_usage;
   /* Open file with FileHandler */
   FileHandler = open("/proc/meminfo", O_RDONLY);
   if (FileHandler < 0) {
      printf("Error: file handler in RAM usage\n");
      return -1;
   }
   /* Read values necessary for the calulation */
   read(FileHandler, FileBuffer, sizeof(FileBuffer) - 1);
   sscanf(FileBuffer,
   "MemTotal: %d kB MemFree: %d kB MemAvailable: %d kB Buffers: %d kB
 Cached: %d kB",
   &memtotal, &memfree, &memavai, &membuffers, &memcached);
   /* Close file */
   close(FileHandler);
   /* Calculate and return percentage of used RAM */
   ram_usage = (memtotal - memfree - membuffers - memcached) * 100 /
 memtotal;
   return ram_usage;
}
```

Quelltextauszug A3: Berechnung der RAM-Auslastung in Linux

```c
void pulseAT_get_CPU_core_usage(uint8_t CPUcount,
pulseAT_agent_data_types_t *pulseAT_agent_data)
{
   (void)(CPUcount);
   /* Get total number of total and idle ticks since last call */
   TickType_t period = xTaskGetTickCount() - prev_tick_count;
   UBaseType_t status = taskENTER_CRITICAL_FROM_ISR();
   TickType_t idle = idle_tick_count;
   taskEXIT_CRITICAL_FROM_ISR(status);
   /* Calculate usage in relative value and in percent */
   float usage = (float)(period - idle) / (float)period;
   uint8_t usage_in_percent = usage * 100;
   /* Update time variables */
   prev_tick_count = xTaskGetTickCount();
   status = taskENTER_CRITICAL_FROM_ISR();
   idle_tick_count = 0;
   taskEXIT_CRITICAL_FROM_ISR(status);
   /* Map data on pulseAT package */
   pulseAT_agent_data->core1_usage =  usage_in_percent;
}
```

Quelltextauszug A4: Berechnung der CPU-Auslastung in FreeRTOS [B6]

```c
/* pulseAT check deadline missed function */
inline uint8_t pulseAT_check_deadline_missed(pulseAT_agent_data_types_t *
    pulseAT_agent_data_ptr,
pulseAT_tick_t start,
pulseAT_tick_t period)
{
   /* Check if the deadline is (already) missed
   * start: last wake up time
   * period: period of repetition of the task */
   uint8_t missed = (start + period <= pulseAT_get_current_time());
   // Deadline is missed -> Trap Notification to pulseAT Manager
   if(missed)
   {
      /* Get task ID */
      pulseAT_agent_data_ptr->deadline_missed = (uint8_t)
    pulseAT_get_current_task_id();
      /* Update pulseAT data and transmit it to the manager */
      pulseAT_update_and_transmit_agent_data(pulseAT_agent_data_ptr);
      /* Reset deadline_missed byte after transmission */
      pulseAT_agent_data_ptr->deadline_missed = 0;
   }
   return missed;
}
```

Quelltextauszug A5: Prüfung der Einhaltung der (Echtzeit-) Deadline in FreeRTOS [B6]

```
/* Definition of the chassis control loop period in ms */
#define CHASSIS_CONTROL_LOOP_PERIOD_MS      20

/* Chassis control loop */
void chassisControlLoop(void *params)
{
   (void)params;
   /* Initialize timing variables */
   TickType_t last_wake_up = xTaskGetTickCount();
   TickType_t period = CHASSIS_CONTROL_LOOP_PERIOD_MS /
    portTICK_PERIOD_MS;
   initChassis();
   /* Wake time */
   vTaskDelay(CHASSIS_RELAY_DELAY / portTICK_PERIOD_MS);
   while(1)
   {
      /* Call update each CHASSIS_CONTROL_LOOP_PERIOD_MS */
      taskENTER_CRITICAL();
      updateChassisControlLoop(period * portTICK_PERIOD_MS);
      taskEXIT_CRITICAL();
      /* Check deadline and update wake up time
      * if the deadline is missed */
      if(pulseAT_check_deadline_missed(&pulseAT_agent_data, last_wake_up,
     period))
       last_wake_up = xTaskGetTickCount();
       /* Unblock */
       vTaskDelayUntil(&last_wake_up, period);
   }
}
```

Quelltextauszug A6: Einbindung der (Echtzeit-) Deadline Funktion am Beispiel der DAEbot Bewegungssteuerung [B6]

```
/* Definition of ATmega2560 RAM size in Byte */
#define RAM_Size 8192

/* RAM utilization calculation */
uint8_t pAG_system_info::pulseAT_get_RAM_usage()
{
   /* Get position of heap and stack */
   extern int __heap_start, *__brkval;
   int v;
   /* Calculate percent of used RAM */
   uint8_t  = unusedRAM  &v - (__brkval == 0 ? (int) &__heap_start : (int
    ) __brkval);
   return (RAM_Size - unusedRAM) / (RAM_Size / 100);
}
```

Quelltextauszug A7: Berechnung der RAM-Auslastung eines Ardunio [B7]

```
inline pulseAT_ExitStatus_types_t pulseAT_init_agent_collector(
pulseAT_manager_data_types_t *manager_data) {
   /* Use default sample time if the sample has not been defined */
   if (manager_data->tmp.sample_time == 0) {
      manager_data->tmp.sample_time = pulseAT_agent_default_sample_time;
   }
   /* Set up initial manager data */
   manager_data->tmp.active = 1;
   manager_data->tmp.query_response = 0;
   manager_data->timeout_counter = 0;
   /* Declare thread IDs */
   pthread_t id_op, id_opp, id_hc, id_mc, id_pc;
   /* Handle each component (this function will be called for each agent
    in main),
   * set up IDs and device type and create threads */
   switch (manager_data->component) {
      case pulseAT_component_Refl_Op:
      manager_data->tmp.topic_id = REFL_OPERATOR_ID_PULSEAT;
      manager_data->agent_data.device_type = rPI_3B;
      pthread_create(&id_op, NULL, pulseAT_collector_thread, manager_data
    );
      break;
      case pulseAT_component_Refl_Op_plus:
      manager_data->tmp.topic_id = REFL_OPERATOR_PLUS_ID_PULSEAT;
      manager_data->agent_data.device_type = ZYNQBERRY7010;
      pthread_create(&id_opp, NULL, pulseAT_collector_thread,
    manager_data);
      break;
      case pulseAT_component_HC:
      manager_data->tmp.topic_id = HEALTH_CTRL_ID_PULSEAT;
      manager_data->agent_data.device_type = ARDUINO_MEGA;
      pthread_create(&id_hc, NULL, pulseAT_collector_thread, manager_data
    );
      break;
      case pulseAT_component_MC:
      manager_data->tmp.topic_id = MOTION_CTRL_ID_PULSEAT;
      manager_data->agent_data.device_type = SMT32F3_DISCOVERY;
      pthread_create(&id_mc, NULL, pulseAT_collector_thread, manager_data
    );
      break;
      case pulseAT_component_PC:
      manager_data->tmp.topic_id = PERCEPTION_CTRL_ID_PULSEAT;
      manager_data->agent_data.device_type = rPI_3B;
      pthread_create(&id_pc, NULL, pulseAT_collector_thread, manager_data
    );
      break;
   }
   return pulseAT_STATUS_OK;
}
```

Quelltextauszug A8: Initialisierung der pulseAT Sammler-Threads für die Agenten des DAEbots

```
/* Get start time */
uint32_t start_time_measurement() {
   uint32_t start_time;
   struct timeval timecheck;
   /* Use Linux functions to get the current time */
   gettimeofday(&timecheck, NULL);
   start_time = (uint32_t) timecheck.tv_sec * 1000
   + (uint32_t) timecheck.tv_usec / 1000;
   return start_time;
}

/* Get elapsed time from start to now */
uint32_t elapsed_time_measurement(long start_time) {
   uint32_t current_time;
   struct timeval timecheck;
   /* Use Linux functions to get the current time */
   gettimeofday(&timecheck, NULL);
   current_time = (uint32_t) timecheck.tv_sec * 1000
   + (uint32_t) timecheck.tv_usec / 1000;
   /* elapsed time = current_time - start_time */
   uint32_t elapsed_time = current_time - start_time;
   return elapsed_time;
}

/* pulseAT collector thread */
void *pulseAT_collector_thread(void *pulseAT_manager_data) {
   /* Step 1: Send query
   * Step 2: Start time measurement
   * Step 3: Receive agent response
   * Step 4: Stop time measurement & get response time
   * Step 5: Calculate Health Condition
   * Step 6: Publish data to MQTT */
   /* Flags and data types */
   uint8_t first_in_inactive = 0;
   uint8_t first_in_active = 0;
   pulseAT_manager_data_types_t *manager_data =
   ((pulseAT_manager_data_types_t *) pulseAT_manager_data);
   /* The collector thread for each agent is always running */
   while (1) {
      /* Step 1 - 6 is running, if the agent is active */
      while (manager_data->tmp.active) {
         if (first_in_active == 0) {
            first_in_active = 1;
            first_in_inactive = 0;
            /* First time acitve: set mode to active */
            manager_data->health_condition.mode =
            health_condition_mode_active;l
         }

         /* Step 1: Send query */
         pulseAT_send_query(manager_data->tmp.topic_id,
```

```
52          &manager_data->tmp.can_socket_tx);
53          /* pulseAT trigger (will be set to "1" in main after
54          * receiving data from agent */
55          manager_data->tmp.query_response = 0;
56
57          /* Step 2: Start time measurement */
58          uint32_t start_time = start_time_measurement();
59          /* Stay in this loop as long as the agent is active */
60          while (1) {
61             /* Check elapsed time */
62             uint32_t elapsed_time = current_time(start_time);
63
64             /* Step 3: Received agent response */
65             if (manager_data->tmp.query_response) {
66
67                /* Step 4: Stop time measurement & get response time */
68                manager_data->response_time = elapsed_time;
69                /* Set back timeout_counter,
70                * as the agent is active (again)  */
71                manager_data->timeout_counter = 0;
72
73                /* Step 5: Calculate Health Condition */
74                pulseAT_calculate_health_condition(manager_data);
75
76                /* Step 6: Publish data to MQTT */
77                /********** Critical Section ******************/
78                pthread_mutex_lock(&mutex);
79                {
80                   pulseAT_manager_data_types_t pulsedata = *manager_data;
81                   bufferWrite(pub_buffer_ptr, pulsedata);
82                }
83                pthread_mutex_unlock(&mutex);
84                /********** Critical Section ******************/
85                break;
86                }
87             /* no response from agent in step 3 */
88             else
89             {
90                /* timeout */
91                if (elapsed_time
92                >= (pulseAT_agent_timeout_maxtime * 1000)) {
93                   manager_data->timeout_counter++;
94                   /* Try pulseAT_agent_timeout_max_errors
95                   * times to get a response from the agent */
96                   if (manager_data->timeout_counter
97                   >= pulseAT_agent_timeout_max_errors) {
98                      manager_data->timeout_counter = 0;
99                      manager_data->tmp.active = 0;
100                  }
101                  /* switch to inactive and break the loop */
102                  break;
```

```
                }
            }
            /* as long as the agent is active, the loop checks if
            * new agent data is received.
            * between those checks, a sleep of 1 ms is done
            * (affects presicion of time measurement) */
            usleep(1000);
         }
         /* Sleep for sample time until starting
         * over with sending the next manager query */
         usleep(manager_data->tmp.sample_time * 1000);
      }
      /* Out of the "ative" loop, as the agent is inactive */
      if (manager_data->tmp.active == 0) {
         /* inactive status just recognized ->
         * set mode to inactive and
         * pusblish it via MQTT */
         if (first_in_inactive == 0) {
            /* Set mode to standby */
            manager_data->health_condition.mode =
            health_condition_mode_standby;
            /* publish inactive status */
            /********** Critical Section ******************/
            pthread_mutex_lock(&mutex);
            {
               pulseAT_manager_data_types_t pulsedata = *manager_data;
               bufferWrite(pub_buffer_ptr,pulsedata);
            }
            pthread_mutex_unlock(&mutex);
            /********** Critical Section ******************/
            /* reset flags */
            first_in_inactive = 1;
            first_in_active = 0;
         }
         /* Sleep (5 seconds) until the agents is active again
         * (A starting agent transmits a trap notification) */
         sleep(5);
      }
   }
   return NULL;
}
```

Quelltextauszug A9: pulseAT Sammler-Thread

Anhang E: Weitere Evaluationsdaten

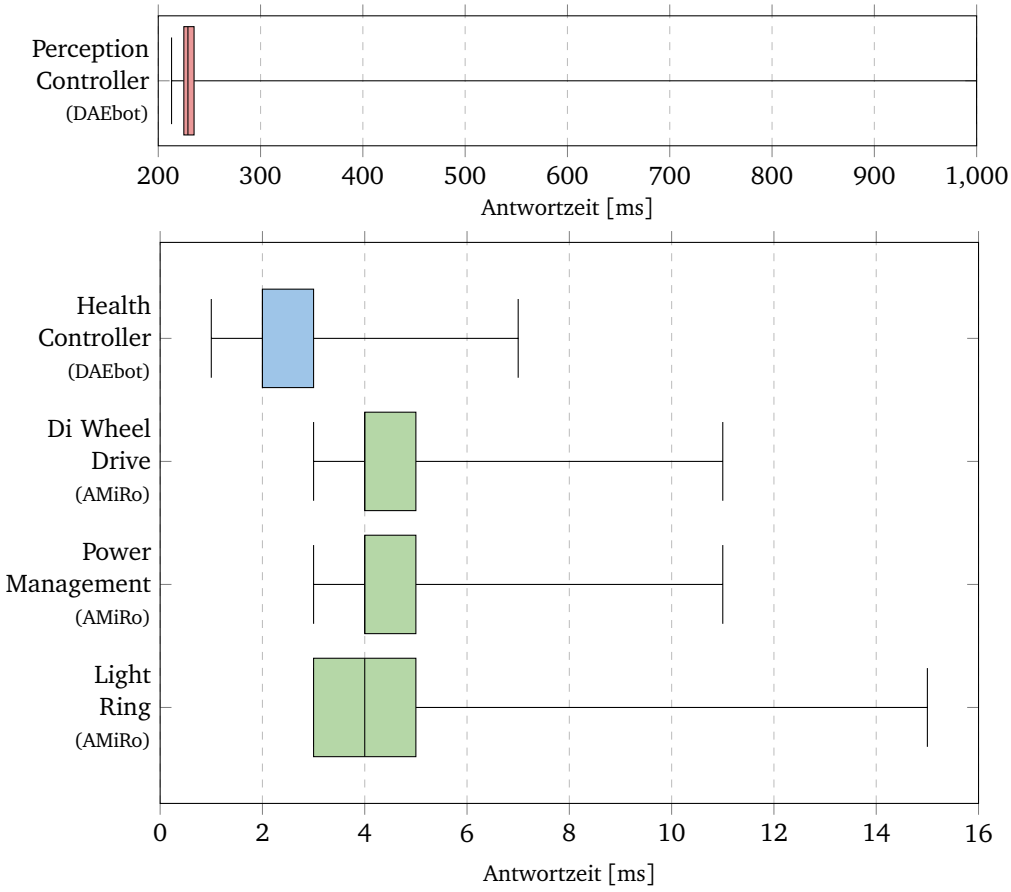


Abbildung A8: Evaluation weiterer pulseAT Antwortzeiten