Die (re-)konfigurierbare Fahrzeugarchitektur

Zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEURS

von der KIT-Fakultät für
Elektrotechnik und Informationstechnik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

DISSERTATION

von

M.Sc. Hannes Frank Stoll
geb. in Rastatt

Tag der mündlichen Prüfung: 06.07.2021

Hauptreferent: Prof. Dr.-Ing. Eric Sax
Korreferent: Univ.-Prof. Dr. Wolfgang Pree

Kurzfassung

Die Lebenszyklen von Fahrzeugen und die Innovationszyklen zugrundeliegender Technologien laufen auseinander, sehr zum Nachteil der Fahrzeughersteller. Besonders betroffen sind dabei Bereiche, die geringe Stückzahlen mit hoher Variabilität und langen Garantiezeiträumen kombinieren, zum Beispiel Busse.

Dadurch ergibt sich eine Anzahl an Herausforderungen an die Hersteller, die im Rahmen dieser Dissertation herausgearbeitet werden.

Anschließend erfolgt eine Betrachtung des Standes der Wissenschaft und Technik, insbesondere mit Hinblick auf die Fragestellung, wie denn dieser die Herausforderungen adressiert. Dabei werden neben technischer auch rechtliche Aspekte beleuchtet, die ihrerseits neue Herausforderungen hinzufügen.

Gezeigt wird, dass klassische Fahrzeugarchitekturen mit ihren unflexiblen signalbasieren Elektrik-/Elektronik-Architekturen nicht mehr geeignet sind, diesen Herausforderungen zu begegnen. Flexiblere serviceorientierte Architekturen eignen sich weitaus besser, um neue Trends wie das automatisierte Fahren oder fortschrittlichere, kamerabasierte Fahrerassistenzsysteme zu integrieren. Dabei werden verschiedene Ansätze wie die bereits in der Automobilindustrie verbreitete AUTOSAR Adaptive Platform und das bisher hauptsächlich in der Forschung eingesetzte Robot Operating System 2 (ROS2) vorgestellt und miteinander verglichen.

Als Konsequenz wird in dieser Dissertation eine (re-)konfigurierbare Fahrzeugarchitektur entwickelt, die Synergien aus einer Verknüpfung verschiedener Domänen wie Nutzfahrzeuge und PKW, aber auch Informationstechnik nutzt. Dies sind zum einen geringere Stückpreise durch Nutzung von Komponenten aus Domänen mit höheren Stückzahlen, zum anderen durch ein Ersetzen von beispielsweise Sensoren durch günstigere beziehungsweise überhaupt noch verfügbare Exemplare während des Lebenszyklus eines Fahrzeuges.

Basierend auf einer serviceorientierten Architektur in ROS2 und vorher definierter Anforderungen an eine solche Fahrzeugarchitektur, wird ein Konzept entworfen und anschließend prototypisch umgesetzt, um Use-Cases darzustellen, die besonders von dieser neuartigen Architektur profitieren. Ein Beispiel hierfür ist das Austauschen von Steuergeräten, entweder aus den oben angesprochenen Gründen der Verfügbarkeit oder Kosten von Komponenten oder aber zur Erweiterung der Funktionalität. Zur Senkung von Betriebskosten und Verbesserung der Energiebilanz des Fahrzeuges und der Flotte können außerdem Funktionalitäten beziehungsweise Services in das Backend ausgelagert werden.

Die Evaluation dieser prototypisch umgesetzten Architektur und der Use-Cases zeigt, dass der Ansatz grundsätzlich funktioniert und außerdem eine nutzbare Performanz erreicht. Neue Chancen ergeben sich durch eine mögliche Steigerung der Ressourcenauslastung und dynamischer Redundanz, die ausgefallene Komponenten zur Laufzeit des Fahrzeuges ausgleichen kann.

Abstract

The life cycles of vehicles and the innovation cycles of the underlying technology are clearly diverging, much to the detriment of vehicle manufacturers. Particularly affected are areas that combine low volumes with high variability and long warranty periods, for example buses.

For manufacturers, this results in a number of challenges, which are elaborated in this dissertation.

This is followed by an examination of the state of the art, particularly with regard to the question of how it addresses these challenges. In addition to technical aspects, legal ones are also examined, which in turn add new challenges.

It is shown that classic vehicle architectures with their inflexible signal-based electrical/electronic architectures are no longer suitable for meeting these challenges. More flexible service-oriented architectures are far better suited to integrate new trends, like automated driving or camera-based advanced driver assistance systems. Different approaches such as the AUTOSAR Adaptive Platform, which is already widely used in the automotive industry, and the Robot Operating System 2 (ROS2), which has so far mainly been used in research, will be presented and compared.

As a consequence, this dissertation develops a (re-)configurable vehicle architecture that exploits synergies from linking different domains such as commercial vehicles and passenger cars, but also information technology. On the one hand, this means lower unit prices through the use of components from domains with higher quantities, and on the other hand, through the replacement of components like sensors with cheaper ones or alternatives with better availability during the life cycle of a vehicle.

Based on a service-oriented architecture in ROS2 and previously defined requirements for such a vehicle architecture, a concept is designed and then

prototypically implemented to demonstrate use cases that particularly benefit from this novel architecture. Examples include the replacement of ECUs, either for the reasons of component availability or cost discussed above, or to extend functionality. In addition, functionalities or services can be outsourced to the backend to reduce operating costs and improve the energy balance of the vehicle and the fleet.

The evaluation of this prototypically implemented architecture and the use cases shows that the approach basically works and achieves usable performance. New opportunities arise from an increase in resource utilization and dynamic redundancy, which can compensate for failed components at vehicle runtime and is made possible by this architecture.

Vorwort

Die vorliegende Dissertation entstand während meiner Zeit als wissenschaftlicher Mitarbeiter am Institut für Technik der Informationsverarbeitung (ITIV) am Karlsruher Institut für Technologie (KIT), die mir wichtige Einblicke nicht nur in die akademische Welt, sondern auch in die Projektarbeit mit Unternehmen aus dem Automotive-Umfeld und darüber hinaus gegeben hat.

Bedanken möchte ich mich daher besonders bei meinem Doktorvater, Prof. Eric Sax, der mir die Möglichkeit dazu gegeben hat und stets mit hilfreichen Tipps und Feedback zur Seite stand.

Ebenfalls bedanken möchte ich mich bei meinem Korreferenten, Univ.-Prof. Wolfgang Pree, dem Vorsitzenden der Prüfungskommission Prof. Werner Nahm und deren weiteren Teilnehmern Prof. Ulrich Paetzold und Prof. Bryce Richards. Sehr gefreut hat mich auch, dass der ehemalige Leiter des ITIV, Prof. Klaus Müller-Glaser, sich spontan die Zeit genommen hatte, an der Prüfung teilzunehmen.

Auch meinen Kollegen gilt mein Dank, dabei besonders Steffen und Simon für Kino- und Computerspieleabende, Timo und Johannes für ausführliche Diskussionen über Gott und die Welt und diverse Basteleien in Software und Hardware, sowie Marc und Daniel für ausführliches Feedback zu meinen Probevorträgen. Natürlich bedanke ich mich auch besonders bei allen weiteren, die mir zusammen einen großartigen Doktorhut gebaut haben.

Schließlich bedanke ich mich bei meiner Freundin Caroline und meinem Vater Harry für die Jagd auf die letzten verbliebenen Fehlerteufelchen.

Karlsruhe, im Juli 2021
Hannes Stoll

Inhaltsverzeichnis

1 Motivation

1.1 Umfeld dieser Dissertation

Im PKW-Bereich unterstützen moderne Fahrzeug immer mehr Features, die weit über das reine Fahren von Punkt A nach Punkt B hinausgehen. Fahrerassistenzsysteme bis hin zum automatisierten Fahren und Entertainmentsysteme mit Internetanbindung sind Beispiele hierfür. Denkbar und geplant ist es, Fahrzeuge ganz ohne menschlichen Fahrer fahren zu lassen.

In anderen Bereichen ist dies schon länger Realität: In Produktionshallen und auf Werksgeländen sind bereits kleine Transportfahrzeuge völlig selbstständig unterwegs. Eine Erweiterung auf den öffentlichen Nahverkehr und den Transport von Gütern eröffnet neue Möglichkeiten: Fahrzeuge können ihre Fahrgäste selbstständig abholen und sind durch kleinere Gefäßgrößen dabei flexibler als herkömmliche Fahrzeuge mit einem menschlichen Fahrer; Logistikunternehmen sind nicht mehr von den erlaubten Lenkzeiten menschlicher Fahrer abhängig.

Insgesamt betrifft diese fortschreitende Technologisierung also nicht nur den PKW-Bereich. Auch LKW, Busse und andere Fahrzeuge werden neuen Herausforderungen gegenübergestellt.

1.2 Herausforderungen an Fahrzeugarchitekturen in diesem Umfeld

Mit zunehmender Automatisierung und Hinzufügen von weiteren Assistenzfunktionen und Entertainmentfeatures steigt auch der Anteil der Informationstechnologie (s. Abbildung 3.2 auf Seite 40).

Halbleiter dienen zwar als Enabler für neue Funktionen, besitzen aber kurze Lebenszyklen von ein bis zwei Jahren mit noch kürzeren Garantiezeiträumen [Sax18]. Auch wenn die Fahrzeuge im Schnitt mit 10,4 Jahren (s. Tabelle A.4 auf Seite 204, wobei Zugmaschinen im Schnitt deutlich älter sind als die restlichen Fahrzeugtypen) in Deutschland noch vergleichsweise jung sind, sind im Vergleich dazu (s. Tabelle A.3 auf Seite 204) die gesamten Lebenszyklen (von Beginn der Entwicklung bis Ende Aftersales des letzten produzierten Fahrzeuges) wesentlich länger, mit mindestens 24 (PKW) bis hin zu 35 Jahren (Busse), wobei zusätzlich bei LKW und Bussen Garantiezeiträume von mehr als 30 Jahren üblich sind.

Ein Beispiel für die sich hieraus ergebende Problematik ist die Abkündigung von Komponenten wie Sensoren, Aktoren oder auch Teilen von ECUs[1] wie CPUs[2]: Haben die Komponenten ihr EoL[3] erreicht, endet die Massenproduktion und die Unterstützung durch den Hersteller, somit die Lieferbarkeit. Eine eigene Lagerhaltung ist teuer und durch die begrenzte Lebensdauer auch von Ersatzteilen nicht immer bis zum Ende des Lebenszyklus des Fahrzeuges möglich. Bei Neu- oder Weiterentwicklungen werden so gegebenenfalls aufwendige Anpassungen notwendig.

Während der Lebensdauer ergeben sich jedoch nicht nur durch eine veränderte Verfügbarkeit von Komponenten Einsatzgebiete für Änderungen an der Fahrzeugarchitektur. Auch Fortschritte in der Technologie oder Änderungen bei gesetzlichen Vorschriften können Anpassungen erforderlich machen; Aktualisierungen und Erweiterungen der Funktionalität stellen Business-Cases dar.

Ein Beispiel für gestiegene Anforderungen während der Lebensdauer ist ein Upgrade der automatisierten Fahrfunktion von SAE Level 3 zu Level 4 oder 5 (für eine Beschreibung der Level, s. Abschnitt 2.1.1 auf Seite 13).

Dieses Upgrade wird in vielen Fällen auch eine Erweiterung der Sensorik und der dazugehörigen Algorithmen bedingen. Damit zusammen hängen aber auch steigende Anforderungen an Rechenleistung, sodass ECUs hinzugefügt oder gegen leistungsfähigere ausgetauscht werden müssen. Wichtig ist dabei,

[1] Electronic Control Units
[2] Central Processing Units
[3] End of Life

zu tauschende oder zu aktualisierende Komponenten und ihre Kompatibilität untereinander nachverfolgen zu können.

Abseits von gestiegenen Anforderungen kann der technologische Fortschritt aber auch genutzt werden, um auf günstigere oder energiesparendere Komponenten zurückzugreifen, womit sowohl Betriebskosten als auch Herstellungskosten gesenkt werden.

Vergleicht man die Zulassungszahlen in Tabelle A.4 auf Seite 204, die immerhin den Gesamtbestand an diesen Fahrzeugen in Deutschland darstellen (knapp 48 Mio. PKW) oder die weltweiten Neuzulassungen im Jahre 2019 (ca. 75 Mio. PKW [Sco20]) mit dem weltweiten Smartphone-Absatz in einem einzigen Jahr (2019) von über 1,3 Mrd. [IDC20], zeigt sich, dass auch die Stückzahlen eine Herausforderung darstellen. Denn diese haben einen direkten Einfluss auf den Preis und die Verfügbarkeit von notwendigen Komponenten oder gar Sonderanfertigungen und konkurrieren dabei unter anderem mit der Smartphoneherstellung, als Beispiele seien Prozessoren, Sensoren und Displays genannt.

Betrachtet man nicht nur den PKW-Markt, sondern wendet sich den LKW oder gar Bussen zu, verringern sich die Stückzahlen noch weiter, mit 2852 Bussen oder 44,31 % Anteil im Jahre 2019 ist EvoBus dennoch bei den Neuzulassungen in Deutschland führend [KBA19]. Genau entgegengesetzt zu den Stückzahlen sieht es dagegen beim Variantenreichtum und der Individualisierbarkeit aus, die bei Bussen durch stark unterschiedliche Anforderungen der Kunden besonders hoch sind (s. Abbildung A.1 auf Seite 189). Dies bedeutet zum einen, dass sich schwerer günstige Kosten für benötigte Komponenten erreichen lassen, zum anderen aber auch, dass Spezialanfertigungen kostspielig oder (zumindest wirtschaftlich) unmöglich werden, wodurch auf verfügbare Standardkomponenten zurückgegriffen werden muss.

Insgesamt ergeben sich hieraus folgende Herausforderungen:

HF1 Nutzen von Standardkomponenten für Verfügbarkeit und günstige Einkaufspreise

HF2 Anpassungen aufgrund nicht mehr verfügbarer Komponenten wie Sensoren oder Aktoren, aber auch Bauteilen von ECUs wie CPUs

HF3 Anpassungen aufgrund von Aktualisierungen oder Erweiterungen der Funktionalität

HF4 Verwaltung der möglichen Varianten der Komponenten, besonders unter Betrachtung von Aktualisierungen

HF5 Universelle Anwendbarkeit der Architekturkonzepts (auf PKW, LKW, Busse, ...), um Synergien zu nutzen

1.3 (Re-)konfigurierbare Fahrzeugarchitektur

Ein Weg, den im vorhergehenden Abschnitt gezeigten Herausforderungen zu begegnen, ist eine (re-)konfigurierbare (s. Definitionen 1.3.1 und 1.3.2 auf der nächsten Seite) Fahrzeugarchitektur. Diese wird daher im Rahmen dieser Dissertation entwickelt.

Ziel dabei ist, eine möglichst universelle Fahrzeugarchitektur zu konzipieren, die nicht auf einen eingeschränkten Einsatzbereich wie zum Beispiel das automatisierte Fahren bei PKW ausgelegt ist. Eine solche Architektur erlaubt es damit, auch in Bereichen mit geringeren Stückzahlen, zum Beispiel Busse, Synergien aus Bereichen mit höheren Stückzahlen wie den PKW zu nutzen. So kann zum Beispiel von günstigeren Einkaufspreisen profitiert werden.

Darüber hinaus lassen sich im Rahmen der Rekonfiguration Komponenten während des gesamten Lebenszyklus des Fahrzeuges, auch über die Entwicklung und Produktion hinaus, ersetzen oder hinzufügen. So können Komponenten, deren EoL erreicht wurde, gegen modernere Alternativen ausgetauscht werden. Hierdurch können sich von der Verfügbarkeit abgesehen weitere Vorteile ergeben. So sind modernere Komponenten bei ähnlicher Performanz häufig günstiger oder stromsparender beziehungsweise umgekehrt bei ähnlichem Preis leistungsfähiger.

Abgesehen von einem Austausch im Feld im Rahmen der Wartung nach einem Ausfall von Komponenten (zum Beispiel aufgrund mechanischer Beschädigung), lässt sich so die Konfiguration des Fahrzeuges auch nach Auslieferung an den Kunden an veränderte rechtliche Rahmenbedingungen, technologischen Fortschritt (zum Beispiel bei der Fahrautomatisierung) oder an nachträgliche Kundenwünsche anpassen. Damit lassen sich auch neue Business-Cases

realisieren, die über den Verkauf und die gewohnte Wartung von Fahrzeugen hinausgehen.

Definition 1.3.1: Konfiguration

Die ISO 10007:2017 „Qualitätsmanagement – Leitfaden für Konfigurationsmanagement“ [Int17a] definiert Konfiguration als miteinander verbundene funktionale und physikalische Charakteristika eines Produktes oder Services, wie in dessen *Konfigurationsinformationen* beschrieben. Diese umfasst die Anforderungen für Entwurf, Realisierung, Verifizierung, Betrieb und Support von Produkt oder Service.
In der ISO/IEC/IEEE 24765:2017 „Systems and software engineering – Vocabulary“ [Int17b] wird außerdem zusammengefasst die Anordnung oder konkrete Version eines (Computer-)Systems und dessen Software, sowie die Art und Anzahl sowie Verbindungen der Teilkomponenten als Konfiguration bezeichnet.
Eine ebenfalls häufig vorkommende Definition bezieht die Konfiguration auf die Parameter der eingesetzten Software. Staron [Sta17] nennt dies *Laufzeitvariabilität*, das heißt, die Software bleibt selbst unverändert, kann jedoch je nach Parametersatz für unterschiedliche Hardwareausstattungen genutzt werden, indem zum Beispiel ein nicht vorhandener Regensensor über einen Parameter in der Software deaktiviert wird.
Im Rahmen dieser Dissertation wird der Begriff hauptsächlich als die Art und Vernetzung der Fahrzeugkomponenten, wie Sensoren, Aktoren und ECUs, sowie die Verteilung der darauf laufenden Services und Softwarekomponenten gebraucht.

Definition 1.3.2: Rekonfiguration

Matevska [Mat10a] definiert die Rekonfiguration als „die technische Sicht des Prozesses der Veränderung eines bereits entwickelten und operativ eingesetzten Systems […], um es an neue Anforderungen anzupassen, Funktionalität zu erweitern, Fehler zu beseitigen oder die Qualitätseigenschaften zu verbessern.“ Im Kontext dieser Dissertation handelt es sich dabei zum Beispiel, in Anlehnung an Definition 1.3.1, um den Austausch von Hardwarekomponenten wie Sensoren, Aktoren oder ECUs, aber auch

von Softwarekomponenten, die entweder neu hinzugefügt, aktualisiert oder zwischen Steuergeräten verschoben werden.
Bei der *statischen* Rekonfiguration muss das System während dieser gestoppt sein und steht daher solange nicht zur Verfügung. Dies ist bei sicherheitskritischen Anwendungen wie dem automatisierten Fahren nicht immer gewünscht. Die *dynamische* Rekonfiguration oder *Rekonfiguration zur Laufzeit* vermeidet daher diese Ausfallzeit, indem Anpassungen ohne ein Herunterfahren des Systems durchgeführt werden.

1.4 Allgemeine Begriffsdefinitionen

Die folgenden Begriffe sind grundlegend um ein gemeinsames Verständnis im Kontext dieser Dissertation zu schaffen. Dies gilt besonders, da einige mehrere Definitionen besitzen können, von denen nur bestimmte hier verwendet werden.

Besonders der Begriff „System“ (Definition 1.4.1) wird im Rahmen dieser Dissertation mit verschiedener Bedeutung genutzt. Gemeinsam ist allen, dass es aus Komponenten (Definition 1.4.2 auf der nächsten Seite) aufgebaut ist und nicht mit Funktionen im Sinne von mathematischen Funktionen (Definition 1.4.4 auf der nächsten Seite) zu verwechselnde Kundenfunktionen beziehungsweise *Features* (Definition 1.4.3 auf der nächsten Seite) bereitstellt.

Definition 1.4.1: System

Die ISO/IEC/IEEE 24765:2017 „Systems and software engineering — Vocabulary“ [Int17b] definiert System als Kombination von interagierenden Elementen, die so organisiert sind, dass sie einen oder mehrere erklärte Zwecke erreichen.
Im Rahmen dieser Dissertation kann dies, je nach Kontext, verschiedene Dinge bedeuten (vgl. [Sax08]):

- Aus Komponenten wie Elektrik und Antrieb aufgebautes Gesamtfahrzeug.

- Die Architektur dieses Fahrzeuges, bestehend aus Steuergeräten mit Sensoren und Aktoren sowie deren Vernetzung.
- Ein einzelnes dieser Steuergeräte, das aus Mechanik, Hardware und Software aufgebaut ist.
- Ein Teil der Software, aufgebaut aus Applikationssoftware, Laufzeitumgebung und Hardwareabstraktion (s. Abbildung 3.9 auf Seite 50).

Für das Umfeld des Systems, s. Abbildung 3.1 auf Seite 40.

Definition 1.4.2: Komponente

Ein System besteht aus Komponenten, wobei eine Komponente einen funktional und logisch abgegrenzten Teil des Systems darstellt. Eine Komponente kann dabei aus Hardware oder Software bestehen und selbst weiter in Subkomponenten unterteilt werden. (nach ISO/IEC/IEEE 24765:2017 „Systems and software engineering — Vocabulary“ [Int17b])

Definition 1.4.3: Feature

Features, auch *Kundenfunktionen*, sind nicht mit Funktionen im Sinne von Definition 1.4.4 zu verwechseln. Kundenfunktionen stellen abstrakte funktionale Charakteristika eines Systems dar, die für Anwender und andere Stakeholder greifbar sind. (nach ISO/IEC 26550:2015 „Software and systems engineering – Reference model for product line engineering and management“ [Int15])

Definition 1.4.4: Funktion

Im Kontext dieser Dissertation ähnelt die (logische) Funktion einer mathematischen Funktion: Es handelt sich hierbei um eine Softwarekomponente, die eine Eingabewerte empfängt, mit diesen eine spezifische Handlung ausführt und einen einzigen Ausgabewert zurückgibt. (nach ISO/IEC 26514:2008 „Systems and software engineering – Requirements for designers and developers of user documentation“ [Int08])

1.5 Forschungsfragen

Für die Untersuchung und Umsetzung der in Abschnitt 1.3 auf Seite 4 beschriebenen (re-)konfigurierbare Fahrzeugarchitektur ergeben sich folgende Forschungsfragen für diese Arbeit:

Forschungsfrage 1: Anforderungen an die Architektur

Welche besonderen Anforderungen ergeben sich durch die (Re-)Konfigurierbarkeit einer Fahrzeugarchitektur an diese?

Forschungsfrage 2: Umsetzung der Architektur

Wie muss die (re-)konfigurierbare Fahrzeugarchitektur aussehen und wie kann sie umgesetzt werden?

Forschungsfrage 3: Use-Cases und deren Demonstration

Was sind sinnvolle Use-Cases, die durch eine (re-)konfigurierbare Fahrzeugarchitektur möglich werden? Wie lassen sie sich prototypisch darstellen?

1.6 Anforderungen an eine (re-)konfigurierbare Fahrzeugarchitektur

1.6.1 Adaptierbarkeit/dynamische Konfiguration

Um eine Rekonfiguration (s. Definition 1.3.2 auf Seite 5) zu ermöglichen, muss sich die Software an veränderte Konfigurationen, auch der Hardware, anpassen können. Dies kann zum Beispiel der Austausch von abgekündigten Sensoren oder Aktoren, aber auch ECUs gegen aktuelle Komponenten sein. Dazu müssen sich diese Komponenten selbst am System anmelden können. Da hier eine Aufwärtskompatibilität gefordert ist, darf keine besondere Vorkonfiguration des Systems notwendig sein. Das heißt unter anderem, dass

erforderliche Gerätetreiber entweder von der neuen Hardware mitgebracht werden müssen oder dass alternativ über bereits vorhandene und definierte Schnittstellen kommuniziert wird. Dies schließt auch eine Übertragung von Metadaten ein, beispielsweise Reglerparameter oder optische Eigenschaften wie Auflösung, Bildwinkel oder auch Position einer Kamera, auf die sich das System entsprechend anpassen muss.

Zusammengefasst ergeben sich folgende grundlegenden Anforderungen an die Adaptierbarkeit:

AA1 Anpassung an veränderte HW-/SW-Konfigurationen muss automatisiert erfolgen.

AA2 Sensoren, Aktoren sowie neue ECUs müssen sich selbst am System registrieren und damit Services bekanntmachen können.

AA3 Neue Komponenten dürfen keine Vorkonfiguration des Systems erfordern.

AA4 Parameter neuer Komponenten müssen an das System übermittelt werden können.

1.6.2 Portabilität

Da in einem Fahrzeug verschiedene Hardwarearchitekturen zum Einsatz kommen (x86, ARM, ...) und sich diese Diversität mit der Lebensdauer durch Austausch und Hinzufügen von Komponenten noch vergrößern wird, muss die Software auf diesen weiterhin und möglichst unverändert ausführbar sein. Dies ergibt sich auch aus dem Wunsch, dass sich Softwarekomponenten und Services möglichst dynamisch (also auch zur Laufzeit) zwischen den Steuergeräten verschieben lassen. Dieser folgt aus der Forderung einer möglichst unterbrechungsfreien Rekonfiguration und Aktualisierung von Hard- und Software und ist spätestens bei OTA[4]-Updates unabdingbar.

Um dies umzusetzen, muss ein gemeinsames Format zur Beschreibung der Softwarekomponenten und deren Funktionalitäten und Hardwareanforde-

[4] over-the-air

rungen definiert werden, zum Beispiel welche Rechenleistung benötigt wird oder ob Spezialhardware wie FPGAs[5] erforderlich ist. Idealerweise lassen sich die Komponenten so verpacken, dass sie direkt als Einheit verteilt werden können.

Zusammengefasst ergeben sich folgende grundlegenden Anforderungen an die Portabilität:

AP1 Softwarekomponenten müssen auch auf aktualisierten Steuergeräten ausführbar sein. D.h. die Ausführung darf nicht von neueren Treiber- oder Betriebssystemversionen beeinträchtigt werden.

AP2 Softwarekomponenten müssen plattformunabhängig, d.h. z.B. auf verschiedenen Prozessorarchitekturen lauffähig sein.

AP3 Softwarekomponenten und Services müssen sich zur Laufzeit zwischen Steuergeräten verschieben lassen.

AP4 Rekonfiguration und Aktualisierung von HW und SW müssen ausfallfrei erfolgen, dürfen also keine Unterbrechungen zur Laufzeit verursachen.

AP5 Softwarekomponenten sollen als Einheit mit der Beschreibung ihrer Anforderungen verteilt werden können.

1.6.3 Sicherheit

Die Architektur muss verschiedenste Sicherheitsanforderungen erfüllen, also sowohl ausfall- (s. Definition 1.6.3 auf Seite 12) als auch manipulationssicher sein. Im Deutschen werden unter dem Begriff „Sicherheit“ zwei unterschiedliche Definitionen zusammengefasst, welche im Englischen getrennt betrachtet werden: „safety“ (s. Definition 1.6.1 auf der nächsten Seite) und „security“ (s. Definition 1.6.2 auf der nächsten Seite).

[5] Field Programmable Gate Arrays

Definition 1.6.1: Safety

Erwartung, dass ein System unter definierten Bedingungen nicht zu einem Zustand führt, in dem Leben, Gesundheit oder Eigentum von Menschen oder die Umwelt gefährdet sind.
(nach ISO/IEC/IEEE 24765:2017 „Systems and software engineering — Vocabulary“ [Int17b])

Definition 1.6.2: Security

Schutz von Fahrzeughardware oder -software vor versehentlichem oder böswilligem Zugriff, Verwendung, Änderung, Zerstörung oder Offenlegung.
(nach ISO/IEC/IEEE 24765:2017 „Systems and software engineering — Vocabulary“ [Int17b])

Security-Probleme können auch zu Safety-Mängeln werden, indem Fahrzeuge gezielt manipuliert werden, daher müssen stets beide Aspekte zusammen betrachtet werden.

Dabei können zum Beispiel mehrere, parallel auf verschiedenen ECUs laufende, Instanzen die Ausfallsicherheit (im Sinne von *Safety*) gewährleisten (s. Abschnitt 4.2.5 auf Seite 109). Firewalls, Verschlüsselung der Kommunikation über das Netzwerk und Authentifizierung der Teilnehmer erfüllen die *Security*-Anforderungen (s. Abschnitt 4.2.2 auf Seite 104).

Zusammengefasst ergeben sich folgende grundlegenden Anforderungen an die Sicherheit:

AS1 Wichtige Services müssen redundant ausgeführt werden und ausgefallene Services ersetzen können.

AS2 Die Services müssen fail-silent sein und dürfen im Fehlerfall keine fehlerhaften Daten senden.

AS3 Daten im Fahrzeugnetzwerk oder Backend dürfen nicht von unautorisierten Teilnehmern gelesen werden können.

AS4 Daten im Fahrzeugnetzwerk oder Backend dürfen nicht von unautorisierten Teilnehmern gesendet werden können.

Diese Anforderungen müssen je nach Anwendungsfall noch weiter detailliert werden (unter anderem in einer Betrachtung nach *ISO 26262* (s. Abschnitt 2.1.3 auf Seite 18) und stehen nicht im Fokus dieser Dissertation. So hängt vom Einsatzgebiet ab, *welche* Services als wichtig deklariert werden. Für ein nicht automatisiertes Fahrzeug oder niedrige Automatisierungsstufen (s. Abschnitt 2.1.1 auf der nächsten Seite) genügt es, *fail-safe* zu sein, das heißt einen sicheren Zustand einzunehmen und die Kontrolle an einen menschlichen Fahrer zu übergeben. Für höhere Stufen genügt das nicht mehr, hier wird vorausgesetzt, dass das System *fail-operational* ist, also Komponenten im Fehlerfall keine (fehlerhafte) Ausgabe mehr senden (*fail-silent*) und damit durch redundante Services ersetzt werden können. Diese Redundanz und die Betrachtung der Security sind bereits im Konzept der (re-)konfigurierbaren Fahrzeugarchitektur enthalten und werden durch den Orchestrator umgesetzt (s. Abschnitt 4.2.2 auf Seite 104).

Definition 1.6.3: fail-operational vs. fail-silent

Zuerst muss zwischen *fail-safe* und *fail-operational* unterschieden werden: Im erstgenannten Fall wird beim Ausfall von Komponenten ein sicherer Zustand eingenommen. Dabei wird davon ausgegangen, dass noch eine Person im Fahrzeug die Kontrolle übernehmen kann. Im letztgenannten Fall muss ein System den Ausfall von Komponenten ohne menschlichen Fallback beherrschen und dabei weiterhin seine (gegebenenfalls degradierte) Aufgabe erfüllen. Dazu werden Komponenten gegebenfalls redundant ausgeführt. Wichtig ist dabei, um das korrekte Ergebnis auswählen zu können, dass diese *fail-silent* sind. *Fail-silent* bedeutet, dass eine Komponente entweder ihre Aufgabe vollständig erfüllt oder überhaupt nicht. (nach [SSWH19])

2 Grundlagen und Stand der Wissenschaft und Technik

2.1 Allgemeine Grundlagen für das automatisierte Fahren

Das Fahren ohne menschlichen Fahrer oder Überwacher wird häufig „autonomes Fahren“ genannt. Dieser Begriff stammt aus dem Altgriechischen αὐτονομία, „Eigengesetzlichkeit, Selbstständigkeit“, aus αὐτός, „selbst“ und νόμος, „Gesetz“. Hierzu muss das gesamte Leistungsspektrum eines menschlichen Fahrers vom Fahrzeug beherrscht werden, dies ist jedoch derzeit bei Weitem nicht der Fall. Tatsächlich gibt es mehrere Stufen der Automatisierung, von Fahrzeugen ganz ohne Fahrerassistenzfunktionen bis hin zu (teilweise oder vollständig) selbstständig fahrenden Fahrzeugen. Daher und weil es keine allgemeingültige, akzeptierte Definition für „autonomes Fahren“ gibt, werden in dieser Arbeit nur der Begriff „**automatisiertes** Fahren“ und dessen verschiedene Stufen verwendet.

2.1.1 Stufen der Automatisierung und Begriffsdefinitionen

Der Verband der Automobilingenieure in den USA (SAE International) unterscheidet sechs verschiedene Stufen der Automatisierung, diese reichen von *keine Automatisierung, Stufe 0* bis *vollautomatisiert, Stufe 5*, die Automatisierung bezieht sich hierbei auf die dynamische Fahraufgabe. Die dynamische Fahraufgabe untergliedert sich in verschiedene Teilbereiche: In diesen ist die laterale und longitudinale Regelung beziehungsweise Steuerung des Fahrzeuges enthalten (über Lenkung und Betätigung von Gas- und Bremspedal), aber auch die Überwachung der Umgebung auf Objekte und Hindernisse sowie die Ausführung einer geeigneten Reaktion. Darüber hinaus gehören auch die

taktische Planung von Fahrmanövern und die korrekte Sicherstellung der Aufälligkeit gegenüber anderen Verkehrsteilnehmern durch Beleuchtung und Signalisierung zur Fahraufgabe. [SAE16]

Hierbei wird immer von motorisierten Fahrzeugen ausgegangen, die sich auf öffentlichen Straßen bewegen. Ein solches Fahrzeug kann durchaus mehrere Fahrautomatisierungssysteme (s. Definition 1.4.1 auf Seite 6 *System*) haben, die sich auf verschiedenen Stufen befinden können. Die aktuelle Stufe wird jedoch nur von den in diesem Augenblick aktivierten Systemen und die dadurch aktiven Rollen definiert. [SAE16]

Die SAE unterscheidet beim Fahren drei verschiedene grundlegende Rollen [SAE16]:

Der (menschliche) Fahrer ist ein Benutzer des Fahrzeuges, der in Echtzeit entweder die dynamische Fahraufgabe voll oder als Fallback für ein Fahrautomatisierungssystem übernimmt. Ein *konventioneller* Fahrer führt hierbei die Lenk- und Beschleunigungs- sowie Gangwechselaufgaben manuell mit den zugehörigen Pedalen und Hebeln aus. Dabei befindet er sich üblicherweise auf dem klassisch als *Fahrersitz* bezeichneten Sitz im Fahrzeug.

Das Fahrautomatisierungssystem ist für die Übernahme der dynamischen Fahraufgabe zuständig. Sein Zusammenspiel mit dem menschlichen Fahrer (falls vorhanden) bestimmt die Zuordnung in die verschiedenen Automatisierungsstufen.

Andere Fahrzeugsysteme und -komponenten umfasst alle Fahrzeugkomponenten und -systeme, die nicht ausschließlich der Fahrautomatisierung zugeordnet werden, auch wenn es mit dieser Überschneidungen geben kann. Die Rollen werden gemäß der aktiven Systemen und ihren Fähigkeiten zugeordnet und bewertet, nicht jedoch basierend auf dem tatsächlichen Verhalten. Ein Beispiel anhand eines menschlichen Fahrers: Eine das Fahrzeug steuernde Person wird auch dann als Fahrer gewertet, wenn sie es versäumt die Straße zu überwachen, während sie ein System wie zum Beispiel einen Abstands-

regeltempomaten (ACC[1]) aktiviert hat, welches sie aufgrund der niedrigen Automatisierungsstufe (hier Stufe 1) nicht von der Überwachungsaufgabe entbindet.

Aufgrund ihrer nur kurzzeitig eingreifenden Wirkung, werden aktive Sicherheitssysteme (im Sinne von *safety*) wie Notbrems- oder Spurhalteassistenten nicht zur Fahrautomatisierung gezählt. Diese System machen die Rolle eines aktiven Fahrers nicht überflüssig oder übernehmen sie teilweise, sondern unterstützen diesen lediglich in einzelnen Situationen. Dennoch sind sie aufgrund der auch für weitergehende Funktionen benötigten und daher verbauten Komponenten nahezu immer in automatisiert fahrenden Fahrzeugen vorhanden.

Folgende Stufen der Automatisierung werden unterschieden (Überblick und Vergleich mit BASt[2]- und NHTSA[3]-Äquivalenten in Abbildung 2.1 auf Seite 17) [SAE16]:

Die niedrigste Stufe (0) **keine Fahrautomatisierung** bedeutet, dass ein menschlicher Fahrer vorhanden ist, der während der gesamten Fahrt ohne Eingreifen des Fahrzeuges die Quer- (Lenken) und Längsführung (Beschleunigen, Bremsen) desselben übernimmt. Hierbei ist kein eingreifendes System aktiv, bis auf aktive Sicherheitssysteme.

Bei der Stufe **Fahrassistenz** (1) übernimmt der Fahrer jeweils entweder die gesamte Längs- **oder** Querregelung, während die andere Aufgabe teilweise vom System ausgeführt wird. Dabei muss der Fahrer das System sowohl dauerhaft überwachen als auch jederzeit zu einer Übernahme der Kontrolle bereit sein. Dabei wird dem Fahrer jedoch ein gewisses Zeitfenster für die Übernahme zugestanden. Ein Beispiel ist das ACC, das die Längsführung mit adaptiver Geschwindigkeits- und Abstandsregelung auf das vorausfahrende Fahrzeug übernimmt.

In der Stufe (2) **Teilautomatisiert** kann das System bereits für einen beschränkten Zeitraum oder in vordefinierten Situationen Längs- **und** Querregelung gleichzeitig übernehmen, während es weiterhin dauerhaft vom Benutzer überwacht werden und dieser zu einem Eingriff bereit sein muss. Erst

[1] Adaptive Cruise Control
[2] Bundesanstalt für Straßenwesen
[3] National Highway Traffic Safety Administration

in diesem Fall wird der Benutzer zum Fahrer. Der menschliche Fahrer ist weiterhin voll für die Objekt- und Ereigniserkennung sowie Reaktion auf diese verantwortlich (OEDR[4]). Beim sogenannten *Autobahnassistenten*, der auf Autobahnen Längs- und Querführung übernehmen kann, handelt es sich um ein solches teilautomatisiertes System.

Bei der **bedingten Fahrautomatisierung** (Stufe 3) führt das System die Fahraufgabe aus, der Fahrer muss aber bereit sein, auf Aufforderung oder bei Ausfall relevanter Systeme zu übernehmen. Die OEDR wird nun vom System übernommen.

Hochautomatisiert (Stufe 4) ist ein System, wenn es die Fahraufgabe dauerhaft ausführen kann, ohne dass damit gerechnet wird, dass ein Benutzer auf Aufforderung die Fahraufgabe übernimmt. Dennoch kann das System den Fahrer auffordern, einzugreifen. Erfolgt dies nicht (beispielsweise, weil gar kein Benutzer an Bord ist, da dieser das System ferngesteuert aktiviert oder das Fahrzeug wieder verlassen hat, was ab Stufe 4 zulässig ist), wird automatisch ein risikominimaler Systemzustand eingenommen.

Den Stufen 1 bis 4 ist dabei gemeinsam, dass sie in der vorgegebenen ODD[5] unterwegs sind. Dies kann die Beschränkung auf bestimmte Fahrsituationen oder Umgebungsverhältnisse sein, außerhalb derer keine automatisierte Fahrfunktion zur Verfügung steht. Ein Beispiel wäre das Fahren auf einer Autobahn, wodurch gewisse Schwierigkeiten wie Fußgänger auf der Straße wegfallen, die in anderen Umgebungen auftreten könnten.

Bei der höchsten Stufe (5) **vollautomatisiert** fällt diese Beschränkung weg, eine Aktivierung ist zu jedem Zeitpunkt möglich, ansonsten verhält sich das System wie ein System der Stufe 4.

2.1.2 Beispiel für die Einordnung in die Stufen der Automatisierung

Der *Tesla Autopilot* besitzt nach vorne und hinten jeweils mehrere redundante Sensoren (s. Abbildung 2.2 auf der nächsten Seite). Nach hinten nur Kame-

[4] Object and Event Detection and Response
[5] Operational Design Domain

SAE-Level	SAE-Bezeichnung	Ausführen von Lenk- und Beschleunigungs-Aufgaben	Überwachen der Fahrumgebung	Fallback für die dynamische Fahraufgabe	vom System beherrschte Fahrmodi	BASt-Level	NHTSA-Level
0	keine Fahrautomatisierung	menschlicher Fahrer	menschlicher Fahrer	menschlicher Fahrer	n/a	Driver only	0
1	Fahrassistenz	menschlicher Fahrer und System	menschlicher Fahrer	menschlicher Fahrer	ausgewählte Modi	assistiert	1
2	teilautomatisiert	System	menschlicher Fahrer	menschlicher Fahrer	ausgewählte Modi	teilautomatisiert	2
3	bedingte Fahrautomatisierung	System	System	menschlicher Fahrer	ausgewählte Modi	hochautomatisiert	3
4	hochautomatisiert	System	System	System	ausgewählte Modi	vollautomatisiert	3/4
5	vollautomatisiert	System	System	System	alle		

Abbildung 2.1: Zusammenfassung der SAE-Level und Vergleich mit den BASt- und NHTSA-Äquivalenten. Nach [SAE16]

Abbildung 2.2: Die Sensoren des Tesla „Autopilot“ [Tes20]. Die Fahrzeugfront ist rechts.

ras, nach vorne mehrere Kameras und ein einziges Radar. Andere optische Sensoren als Kameras, wie zum Beispiel Lidar[6] sind nicht vorhanden.

[6] Light Detection and Ranging

Diese Komponenten sind Teil des sogenannten *Funktionspaketes für autonomes Fahren*. Laut Tesla soll das „autonome Fahren" doppelt so sicher wie ein menschlicher Fahrer sein, jedoch wird dies nicht durch eine Quelle oder weitere Daten belegt. Tesla selbst gibt nicht an, wann dieses System vollständig verfügbar sein wird, jedoch soll auch weiterhin ein Fahrer erforderlich sein, der das System und den Verkehr dauerhaft überwacht und jederzeit zum Eingreifen bereit ist. [Tes20] Nach SAE ist das maximal Stufe 2, da er weiterhin auf Objekte und Ereignisse reagieren können muss. Diese Einschränkung verwehrt die Klassifizierung in Stufe 3. Von tatsächlichem vollautomatisiertem Fahren nach SAE ist das Tesla-System also weit entfernt. Zusätzlich ist fraglich, ob die Leistung einem Stand entspricht, der als voll funktionsfähig bezeichnet werden kann: In einer Umfrage aus 2016 geben 62,4% der befragten Fahrer, die den Autopiloten nutzen, an, schon mindestens einmal unerwartetes oder ungewöhnliches Verhalten im autonomen Fahrmodus erfahren zu haben. Dies umfasst hauptsächlich Schwierigkeiten mit der Spurerkennung, worunter aber auch Situationen fallen, in denen das Fahrzeug versucht hat, in den Gegenverkehr zu fahren. Weitere Probleme betrafen die Längsregelung, wie zum Beispiel plötzliche starke Beschleunigungen (positiv und negativ). [DB16]

Die 33. Zivilkammer des Landgerichts München I hat am 14.07.2020 Tesla verboten, mit dem irreführenden Begriff *Autopilot* für dieses System zu werben [LG 20].

2.1.3 Rechtliche Aspekte des automatisierten Fahrens

Darüber hinaus existieren weitere Normen und Gesetze, die das automatisierte Fahren betreffen und regulieren. Dazu müssen sie jedoch nicht notwendigerweise explizit für dieses erstellt worden sein.

ISO 26262:2018

Die ISO-Norm ISO 26262 mit dem Titel „Road vehicles – Functional safety" [Int18] ist die Anpassung der älteren Norm IEC 61508 speziell auf Serien-Straßenfahrzeuge mit zulässiger Gesamtmasse von maximal 3,5 t, also hauptsächlich PKW. In der ersten Auflage sind Fahrzeuge wie LKW oder Motorräder nicht enthalten, die zweite Auflage adressiert dann auch diese explizit.

Im Gegensatz zu EG-Richtlinien und ECE-Regelungen ist die ISO 26262 in Europa nicht direkt rechtlich verpflichtend. Da jedoch im Rahmen der Produkthaftung eine Entwicklung nach dem Stand der Technik erforderlich ist (zu dem die ISO 26262 auch gehört), ist dadurch mittelbar der Zwang gegeben, nach dieser Norm zu entwickeln.

Funktionale Sicherheit bedeutet, dass ausschließlich Aspekte betrachtet werden, die mit der jeweiligen Funktion zusammenhängen. Beispielsweise werden Verbrennungen an Teilen des Fahrzeuges, die Hitze als Abfallprodukt erzeugen (Motor) daher nicht betrachtet, wohl aber an solchen, bei denen die Erwärmung Teil der Funktion ist (Sitzheizung). Ebenso muss betrachtet werden, ob ein System (zum Beispiel Notbremsassistent) nicht nur diese Funktion ausführt, sondern dies auch zum gewünschten Zeitpunkt tut (bei Annäherung an ein Hindernis).

ISO/PAS 21448:2019

Über die funktionale Sicherheit der Fahrzeugkomponenten hinaus, die von der ISO 26262 abgedeckt wird, behandelt die ISO/PAS 21448:2019 mit dem Titel „Road vehicles – Safety of the intended functionality“ [Int19] die Sicherheit der Sollfunktion. Während die ISO 26262 den Ausfall von Komponenten der E/E-Architektur[7] abdeckt, berücksichtigt die ISO/PAS 21448 auch Performanzeinschränkungen oder unzureichendes Situationsbewusstsein. Dabei spielt es keine Rolle, ob die gewollte Funktion unzulänglich ist oder ein vorhersehbarer Missbrauch vorliegt. Ein Beispiel für einen solchen vorhersehbaren Missbrauch ist mangelnde Aufmerksamkeit des Fahrers beim Nutzen einer Level-2-Fahrautomatisierung (s. Abschnitt 2.1.1 auf Seite 13). Security-Probleme durch Angriffe von außen sind jedoch nicht Teil der Norm.

ISO/SAE DIS 21434

Für Betrachtung der Security des Fahrzeuges ist momentan der Standard ISO/SAE DIS 21434 „Road vehicles – Cybersecurity engineering“ [Int20c] in der Entwicklung. Dabei soll der Standard Maßnahmen für die Entwicklung

[7] Elektrisch/Elektronische Architektur

von Fahrzeugen vorschlagen, die potenziell zukünftig Bedingung für eine Typzulassung sein könnten. Der Aufbau ähnelt dem des Standards ISO 26262.

2.2 Grundlagen zu Wahrnehmung und Navigation

Für die spätere Umsetzung von Use-Case 1 (s. Abschnitt 4.4 auf Seite 121), im Rahmen dessen ein automatisiert fahrendes Fahrzeug auf die (re-)konfigurierbare Architektur umgerüstet wird, spielen Wahrnehmung und Navigation desselben eine gewichtige Rolle. Erstere besteht zum Beispiel aus der Lokalisierung und Objekterkennung, letztere aus der Wegfindung und der Umsetzung des gefundenen Pfades zum Ziel. Damit benötigen beide ein Verständnis der Umgebung, dessen Grundlage ihnen die Sensorik liefert.

2.2.1 Sensoren, die Bilddaten liefern

Im Kontext dieser Dissertation sind zwei Arten von Bilddaten interessant: 2D-Bilder, die sich für die Objekterkennung eignen, und 3D-Punktwolken, die zusätzliche Tiefeninformationen liefern.

Dabei gibt es **2D-Kameras** für verschiedene Wellenlängenbereiche, von denen der sichtbare Bereich und der mittlere Infrarotbereich (3 bis 50 µm) für das automatisierte Fahren relevant sind. Besonderheit der in Fahrerassistenzsystemen verbauten Kameras gegenüber üblicher Nachtsichtkameras, die im nahen Infrarotbereich aufzeichnen, ist, dass Menschen und Tiere durch ihre Wärmestrahlung besonders gut erkannt werden.

Allerdings sind sie aufgrund ihres aufwändigen Aufbaus wesentlich teurer als Kameras für den sichtbaren oder nahen Infrarotbereich. Um eine Signaldrift zu vermeiden, müssen die Bildsensoren stets auf konstanter Temperatur gehalten werden, zum Beispiel durch Peltier-Elemente. Als Sensor kommen sogenannte *Bolometer-Arrays* zum Einsatz, welche aus einem Array von thermisch isolierten Membranen mit Absorbern mit stark temperaturabhängigem Widerstand bestehen. Trifft Wärmestrahlung auf diese, werden sie von dieser erwärmt und die resultierende Widerstandsänderung ausgelesen. Ein mechanischer Verschluss dient der periodischen Abschattung und ermöglicht es, den jeder Membran eigenen spezifischen Widerstand als Referenz zu messen. Durch die-

sen Aufbau ist die mögliche Auflösung gegenüber Kameras für den sichtbaren Bereich gering. [VWE+07] [Max20]

Letztere können wesentlich einfacher aufgebaut werden und basieren heutzutage nahezu ausschließlich auf einem sogenannten *CMOS-Sensor*. Die Bezeichnung rührt daher, dass die für die Aufnahme der Bilddaten zuständigen Photodioden und zugehörigen n-Kanal-MOSFETs in CMOS-Technologie aufgebaut sind. Dabei wird die Sperrschichtkapazität der in Sperrrichtung betriebenen Photodiode zu Beginn der Messung durch den Transistor auf einen definierten Spannungspegel gebracht und anschließend durch eintreffende Photonen entladen. Schließlich wird die resultierende Spannung pixelweise mit ADCs[8] ausgelesen. Die Aufnahme eines Farbbilds erfordert vorgelagerte Farbfilter. [Gil12]

Am Ende geben alle dieser 2D-Kameras ihre Bilder als ein Array aus RGB-Pixeln aus, welches für die Objekterkennung genutzt werden kann. Dabei wird jedem Pixel ein Farbwert aus den Grundfarben rot, grün und blau zugewiesen.

Für die Bestimmung der Lage erkannter Objekte im Raum oder die Erstellung von 3D-Karten werden im Rahmen dieser Dissertation sogenannte *Punktwolken* genutzt. Dabei bildet eine Punktwolke eine räumliche Struktur ab, indem jeder Punkt in der Wolke durch seine Raumkoordinaten definiert ist. Zusätzlich kann jeder Punkt weitere Informationen wie Farbwerte oder, bei einem Lidar, die Intensität des empfangenen Bildpunktes enthalten.

Die zwei für diese Dissertation relevanten Sensoren, die Punktwolken ausgeben beziehungsweise aus deren Daten Punktwolken berechnet werden können, sind **Lidar** und **Stereokamera**.

Lidar Ein Lidar sendet Laserpulse aus, deren von Hindernissen reflektiertes Licht anschließend von einem Photodetektor wieder empfangen wird. Durch die Laufzeit des Signals oder Interferenzen kann auf die Entfernung des Objektes geschlossen werden, der Dopplereffekt ermöglicht über eine Messung der Frequenz die Bestimmung der Geschwindigkeit. Die empfangene Intensität hängt dabei unter anderem vom Material des Objekts ab. Eine Ablenkung des Signals durch Spiegel oder Bewegung von Laserdiode und Sensor ermöglicht es, die Umgebung zu „scannen“

[8] Analog-Digital-Wandler

und so ein dreidimensionales Abbild dieser zu erhalten. Verbreitet sind dabei Systeme, die für die vertikale Auflösung mehrere (üblicherweise im Bereich von 16 bis 128) Laserquellen und zugehörige Sensoren kombinieren, die leicht versetzt angeordnet sind. Für die horizontale Erfassung werden diese um eine senkrechte Achse gedreht. Ein Beispiel hierfür ist das *Velodyne Puck* (früher: *Velodyne VLP-16*), das 16 Laser nutzt, um eine 3D-Punktwolke der Umgebung zu erzeugen [Vel20].

Üblicherweise werden die Daten als Punktwolke, bestehend aus Position der reflektierten Bildpunkte und deren Intensität, ausgegeben.

Stereokamera Eine Stereokamera funktioniert ähnlich wie das räumliche Sehen beim Menschen. Parallele Bildsensoren (oder Augen) nehmen Objekte durch ihre Entfernung zueinander aus leicht verschiedenem Winkel wahr. Die entstehende Parallaxe ändert sich mit der Entfernung des Objekts. Indem die Verschiebung (*disparity*) zusammengehörender Punkte in beiden Bildern geschätzt wird, kann so deren Entfernung berechnet werden. Indem man die Verschiebung für jedes einzelne Pixel berechnet, erhält man eine sogenannte *dense disparity map*. [Hir08]

Herausforderungen dabei sind nicht exakt parallel ausgerichtete Bildsensoren und das Fehlen geeigneter gemeinsamer Punkte in den Bildern. Um ersteres zu lösen, wird eine sogenannte *Rektifizierung* durchgeführt. Dabei werden beide Bilder auf eine gemeinsame Ebene projiziert. Das Fehlen gemeinsamer Punkte kann durch einen Projektor gelöst werden, der ein für das menschliche Auge unsichtbares Muster auf die Umgebung projiziert (*active stereo*). [Ser20]

Ein Beispiel für eine Stereokamera, die sowohl Tiefeninformationen als auch ein 2D-Bild liefert, ist die *Intel RealSense D435* [Int20b]. Dabei wird das Tiefenbild in der Kamera berechnet und als Pixelarray zurückgegeben, bei dem jedem einzelnen Pixel eine Tiefeninformation zugeordnet ist. Zusätzlich stellt Intel mit dem *Intel RealSense SDK 2.0* [Int20a] Bibliotheken zur Verfügung, mit denen unter anderem aus diesen Tiefenbildern eine Punktwolke berechnet werden kann.

2.2.2 Lenkwinkel- und Raddrehzahlsensoren

Zum Verständnis der Umgebung gehört auch die Bestimmung der eigenen Position in dieser. Lenkwinkel- und Raddrehzahlsensoren ermöglichen eine Lokalisierung des Fahrzeuges relativ zum Ausgangspunkt, ohne externe Hilfsmittel. Durch Radschlupf und andere Ungenauigkeiten ist die so ermittelte Position jedoch nur für einen beschränkten Zeitraum genau genug für Anwendungen des automatisierten Fahrens. Ein Beispiel ist das zeitlich begrenzte Durchfahren eines Tunnels.

Inkrementalgeber ermöglichen es, sowohl Winkel als auch, mithilfe einer Zeitmessung, Winkelgeschwindigkeiten und damit Drehzahlen zu bestimmen. Die Bezeichnung *Inkremental*geber stammt daher, dass sie keine absolute Position bestimmen und ausgeben können, der gemessene Winkel ist daher immer relativ zur Ausgangslage. Eine Bestimmung von Drehrichtung und -winkel ist daher über zwei ausgegebene, zueinander um 90° phasenverschobene Signale möglich. [NAS]

Diese Signale können intern auf verschiedenen Wegen generiert werden [NAS]:

Schleifkontakte Eine kostengünstige Variante ist die Realisierung über Schleifkontakte, die auf einer Kreisbahn aufgebracht sind. Nachteil dabei ist der hohe mechanische Verschleiß, wodurch sie sich nicht für einen längeren Einsatz eignen.

Optische Abtastung Bei der optischen Abtastung werden statt der Schleifkontakte Schlitze oder transparente Scheiben mit lichtundurchlässigem Aufdruck verwendet. Ein Beispiel für eine solche Scheibe ist in Abbildung 2.3 auf der nächsten Seite links dargestellt. Ähnlich wie bei einer Lichtschranke werden diese von einer Leuchtdiode beleuchtet, wobei auf der entgegengesetzten Seite das Signal von einer Photodiode wieder aufgenommen wird. Zusätzliche Markierungen (s. Abbildung 2.3 auf der nächsten Seite Mitte und rechts) können verwendet werden, um auch die Absolutposition zu kodieren, damit erhält man einen *Absolutwertgeber*.

Bei dieser Art der Abtastung sind höhere Auflösungen bei niedrigerem Verschleiß möglich, jedoch ist der Stromverbrauch in Ruhe deutlich höher.

Ein Beispiel für die optische Abtastung ist der Inkrementalgeber für den Lenkwinkel in der elektrischen Lenkung des Basisfahrzeuges für die Umsetzung der (re-)konfigurierbaren Fahrzeugarchitektur „Nora“ (s. Abschnitt 5.1.1 auf Seite 131).

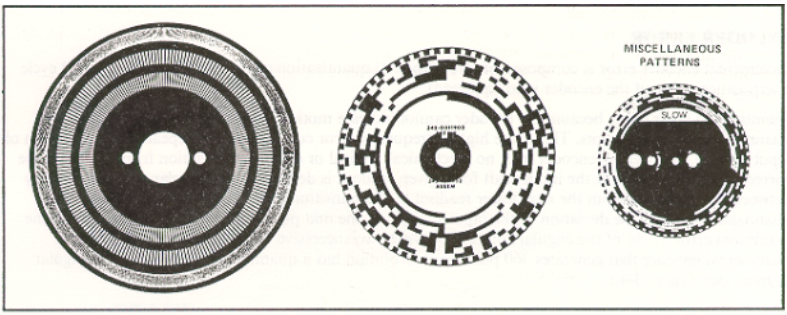


Abbildung 2.3: Verschiedene Encoderscheiben von optischen Drehencodern [NAS]

Magnetische Abtastung Die magnetische Abtastung nutzt ebenfalls ein Rad, in diesem Fall mit magnetisierten Segmenten. Die Erfassung der Werte erfolgt hier über Hall-Sensoren. Im Gegensatz zur optischen Abtastung ist hier der Ruhestrom gering und ein robuster, vor Verschmutzung geschützter, Aufbau in manchen Fällen wirtschaftlicher umzusetzen.

Ein Beispiel für die magnetische Abtastung per Hallsensor ist das Geschwindigkeitssignal aus dem Getriebe von „Nora“ (s. Abschnitt 5.1.1 auf Seite 128).

2.2.3 Sensoren für Beschleunigung und Drehraten

Weitere Informationen über die Lage des Fahrzeuges im Raum können durch Beschleunigungs- und Drehratensensoren gewonnen werden. Durch Integration der Daten lassen sich auch Geschwindigkeit und Position mit ihnen schätzen, aufgrund der sich integrierenden Fehler jedoch wie bei Raddrehzahl- und Lenkwinkelsensoren nur über einen kurzen Zeitraum genau genug für die Lokalisierung.

Im Fahrzeugumfeld kommen nahezu ausschließlich MEMS[9] zum Einsatz. Diese lassen sich mit herkömmlichen Methoden zur Chipfertigung und damit vergleichsweise günstig und in großen Mengen herstellen.

Über kapazitive Effekte können so Sensoren für Beschleunigung und Drehraten gefertigt werden. Dabei kommen bewegliche, gefedert gelagerte, Elemente zum Einsatz, die je nach Lage zueinander die Kapazität verändern.

Beschleunigungssensor Beim Beschleunigungssensor sind das Kämme, die durch die Massenträgheit oder die Erdbeschleunigung ausgelenkt werden (s. Abbildung 2.4: Der weiße Kamm ist gefedert gelagert, die grauen Elektroden sind fest). Für eine Messung im Dreidimensionalen kommen jeweils senkrecht zueinander stehende Sensoren zum Einsatz, deren Ebenen von den Raumachsen aufgespannt werden.

Drehratensensor Bei Drehratensensoren kommt ebenfalls für jede zu messende Richtung ein Sensor zum Einsatz, dabei wird die auf vibrierende kapazitive Elemente wirkende Coriolis-Kraft ausgenutzt.

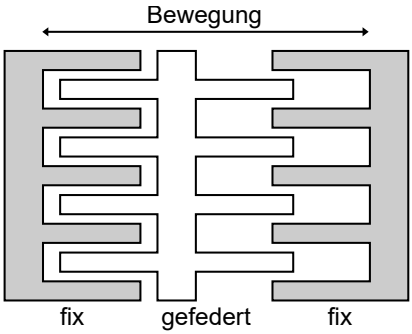


Abbildung 2.4: Funktionsprinzip eines MEMS-Accelerometers

Werden Beschleunigungs- und Drehratensensoren kombiniert, erhält man eine IMU[10].

[9] Micro-Electro-Mechanical Systems
[10] Inertial Measurement Unit

2.2.4 Lokalisierung per GNSS

GNSS[11] ist der Sammelbegriff für Systeme wie GPS (USA), GLONASS (Russland), Galileo (EU) und Beidou (VR China). Hierbei senden Satelliten eine hochpräzise Uhrzeit und ihre genaue Position im Orbit, sowie weitere Korrekturdaten für veränderliche Einflüsse durch die Erdatmosphäre. Ein Empfänger kann daraus über die Bestimmung der Laufzeiten der Signale verschiedener Satelliten seine eigene Position bestimmen. Dabei spielen auch Frequenzveränderungen durch den Dopplereffekt, Einflüsse der Erdatmosphäre auf die Signallaufzeit, relativistische Effekte durch Masse der Erde und Geschwindigkeiten der Satelliten, sowie Uhrenfehler im Empfänger eine Rolle. Dieser Uhrenfehler muss geschätzt werden, um von der empfangenen absoluten Uhrzeit der Satelliten auf die Signallaufzeit schließen zu können. Insgesamt ergeben sich also vier Unbekannte: die Koordinaten der Position im dreidimensionalen Raum und der Uhrenfehler. Daher ist entsprechend das Signal von mindestens vier Satelliten für eine verlässliche Positionsbestimmung erforderlich. Ohne zusätzliche Maßnahmen kann eine Genauigkeit im Meterbereich erreicht werden. [Abe02]

Zusätzliche Korrekturdaten von Bodenstationen (DGPS[12] und RTK[13]-Korrektur) können die Genauigkeit bis auf den niedrigen einstelligen Zentimeterbereich verbessern [mag20].

Bei DGPS wird dabei eine Bodenstation in der Nähe (bis maximal mehrere Kilometer) als Referenz genutzt, deren exakte Position bekannt ist. Es wird davon ausgegangen, dass bei solch niedriger Entfernung dieselben Fehlerquellen auf das empfangene Signal wirken. Daher kann die Bodenstation einen Korrekturfaktor berechnen und an den mobilen Empfänger senden. [mag20]

Mittels RTK-Korrektur kann die Genauigkeit weiter erhöht werden, indem zusätzlich die Phaseninformationen der von den Satelliten empfangenen Signale korreliert und in die Berechnung der Position mit einbezogen werden. [mag20]

[11] Global Navigation Satellite System
[12] Differential Global Positioning System
[13] Real Time Kinematic

Vorteil gegenüber anderen Sensoren wie Beschleunigungs- oder Raddrehzahlsensoren ist, dass GNSS nicht mit einem sich aufintegrierenden Fehler behaftet sind.

2.2.5 GNSS-unabhängige Lokalisierung

In Innenräumen (Garagen etc.) oder auf abgeschatteten Innenhöfen ist eine Lokalisierung per GNSS in den meisten Fällen nicht möglich, es muss daher auf Alternativen zurückgegriffen werden.

Eine Gruppe von GNSS-unabhängigen Lokalisierungslösungen bilden kamerabasierte Systeme. Umsetzungen dieser umfassen sowohl solche, die mit der Erfassung künstlicher Marker [CPL14] oder ohne solche [HKRA] als Referenz für die optische Featureerkennung in den Kamerabildern arbeiten. Auch diese Systeme sind stark von Wetterbedingungen abhängig, zum Beispiel erschweren jahreszeitliche Veränderungen der Optik der Umgebung ihre Arbeit.

Andere Ansätze verwenden auf UHF[14]-Wellen basierende Übertragungsstandards, wie zum Beispiel Bluetooth, Zigbee oder Wireless LAN[15]. Das Funktionsprinzip ist meist die Berechnung der Position über die empfangene Signalstärke von Sendern mit bekannter Position [SGSM11]. Zur Lokalisierung in Innenräumen werden diese häufig mit Methoden wie Magnetic Fingerprinting oder solchen, die vordefinierte Karten der positionsabhängigen Signalstärkesignaturen nutzen, erweitert [SGSM11]. Nachteil dieser Lösungen ist, dass sie weder die Anforderungen an die Aktualisierungsrate der Position noch an die Genauigkeit erfüllen.

In den letzten Jahren wurden UWB[16]-Systeme zur Lokalisierung eingeführt. Aufgrund ihrer sehr hohen Bandbreite ermöglichen sie eine hohe zeitliche Auflösung. Die Position wird über sogenannte time-of-flight Messungen des UWB-Signals bestimmt, sprich die Signallaufzeit von der eigenen Position zu Beacons mit bekannter Position und zurück. Frühere Implementierungen zur Lokalisierung von Robotern nutzten noch Kabel, um die erforderliche

[14] Ultra High Frequency
[15] Local Area Network
[16] Ultra-wideband

genaue zeitliche Synchronisierung der Knoten durchzuführen, was eine solche Umsetzung ungeeignet für den Einsatz in einem Fahrzeug macht [KSGW07].

Eine weitere Steigerung der Genauigkeit kann zum einen durch kalibrierte Tags und Beacons erreicht werden [SPA+14], zum anderen dadurch, dass zusätzlich zu den Laufzeitmessungen der Signale die Ankunftswinkel dieser berücksichtigt werden. Nachteile hiervon sind, dass hierfür verteilte Antennen benötigt werden, was nicht nur erhöhte räumliche Anforderungen stellt, sondern auch die Synchronisierung erschwert [INK12]. Schlussendlich disqualifiziert wird diese Lösung jedoch durch die Nutzung des Bandspektrums zwischen 2 und 6 GHz, welches in Deutschland nicht durchgehend für Signalstärken über $-85\,\mathrm{dBm/MHz}$ freigegeben ist [Bun18].

Die verwendeten Knoten müssen nicht fest positioniert sein, es gibt auch Systeme, bei denen zumindest ein Teil der Knoten mobil ist. Beck et. al. [BBK14] nutzen ein System mit mobilen Ankern und einem multidimensional scaling (MDS) tracking Algorithmus. Dieses Konzept schlägt vor, insgesamt sieben Anker zu nutzen, von denen mindestens drei stationär sein müssen.

Von der Bestimmung der Position von Robotern oder Fahrzeugen abgesehen, gibt es auch andere Anwendungsgebiete: Zum Beispiel die Erfassung von Position und Bewegung von Fußgängern [KHS15] [NAPG15].

2.2.6 Lokalisierung über SLAM

SLAM[17] bezeichnet die Erstellung der Karte der Umgebung und die gleichzeitige Lokalisierung in dieser [TL08].

Dabei werden Daten aus verschiedenen Quellen genutzt: Die Odometrie basierend auf der Radgeschwindigkeit und dem Lenkwinkel, von IMUs gemessene Beschleunigungen und Punktwolken von 3D-Kamera und Lidar. Für die Bestimmung der absoluten Position können außerdem GNSS-Daten zur Korrektur herangezogen werden.

Zu jedem Messzeitpunkt wird die Änderung der Position zum vorhergehenden Zeitpunkt geschätzt, indem zum Beispiel die Punktwolken verglichen und die

[17] Simultaneous Localization and Mapping

Verschiebung bestimmt wird. Dazu ist es erforderlich, dass eine möglichst genaue Lage der z-Achse über die Gravitation bestimmt wird, um so den Rechenaufwand und Fehler aufgrund von Neigungen zu reduzieren und die Lage der Karte im Raum zu bestimmen.

Für die Schätzung der Position können verschiedene Verfahren zum Einsatz kommen, zum Beispiel EKF[18]-, graphen- oder partikelbasierte Methoden.

2.2.7 Lokale und globale Wegfindungsstrategien

Ist die Position geschätzt, muss entschieden werden, wie das Fahrzeug zum Ziel gelangen soll. Hierfür ist die Wegfindung zuständig. Zur Reduktion des benötigten Rechenaufwandes wird üblicherweise zwischen lokaler und globaler Wegfindung unterschieden. Die globale Wegfindung bestimmt dabei nur einen groben Weg vom Start- zum Zielpunkt unter Umfahrung vorher bekannter Hindernisse, während die lokale Wegfindung auch Eigenschaften des Fahrzeuges wie Manövrierfähigkeiten und akut auftretende Hindernisse berücksichtigt und damit den konkreten Weg für die nähere Zukunft (meist wenige Sekunden nach dem aktuellen Zeitpunkt) plant.

Die für die Navigation (und teilweise Lokalisierung) notwendigen Informationen werden in Karten gespeichert. Hierfür kann zum Beispiel ein zellenbasierter Ansatz, eine sogenannte *Occupation Map*, zum Einsatz kommen. Hierbei wird die Umgebung des Fahrzeuges auf ein zweidimensionales Array mit gleich großen Zellen aufgeteilt. Diesen Zellen kann jeweils eine „Farbe“ zugeordnet werden: schwarz bedeutet ein auf jeden Fall zu vermeidendes Hindernis in der Zelle, weiß bedeutet, die Zelle kann und darf befahren werden. Die Kantenlänge der Zellen ist abhängig von benötigter Genauigkeit und zur Verfügung stehender Rechenleistung beziehungsweise Speicherkapazität. Üblicherweise wird auch hier eine feinere Auflösung für die lokale Wegfindung genutzt als für die globale.

Zwei Ansätze, die zur globalen Wegfindung genutzt werden können, sind der Grassfire-Algorithmus und der darauf basierende Potentialfeld-Algorithmus. In dieser Dissertation wird eine solche Occupation Map von einem SLAM-

[18] Extended Kalman-Filter

Algorithmus erstellt (s. Seite 136), von der Objekterkennung mit Hindernissen versehen und anschließend zur Wegfindung und Navigation genutzt (s. Abschnitt 5.1.4 auf Seite 144).

Der **Grassfire-Algorithmus** führt eine Art Breitensuche durch: Vom Zielpunkt ausgehend werden jeweils die benachbarten und erreichbaren Felder besucht und ihre Entfernung zum Zielpunkt gespeichert. Anschließend wird das für die Nachbarn der so markierten Felder durchgeführt, bis der Startpunkt erreicht und somit die kürzeste Route gefunden ist.

Beim **Potentialfeld-Algorithmus** wird eine Art virtuelles Potentialfeld aufgespannt: Es wird davon ausgegangen, dass das Fahrzeug selbst (hier zum Beispiel positiv) geladen ist und das Ziel entsprechend stark negativ. Hindernisse erhalten eine positive Ladung. Damit zieht das Ziel das Fahrzeug an, Hindernisse stoßen es ab. [KB91]

Anders lässt sich das als Gefälle zum Ziel hin darstellen, wobei die Hindernisse Bergen entsprechen.

Dabei zeigt sich, dass lokale Minima nicht überwunden werden können, wenn nur dem Gradienten gefolgt wird. Ein Beispiel sind U-förmige Hindernisse. Damit eignet sich dieser Algorithmus besser für die *lokale* Wegfindung. [KB91]

2.3 Neuronale Netze zur Objekterkennung in Bilddaten

Bereits dargestellt wurde (s. zum Beispiel Abschnitt 2.1.1 auf Seite 13), dass das automatisiert fahrende Fahrzeug Hindernissen, also Objekten im Fahrweg, ausweichen beziehungsweise angemessen auf diese reagieren muss. Um eine angemessene Reaktion finden zu können, müssen diese Objekte erkannt und klassifiziert werden.

Während herkömmliche Methoden zur Bildverarbeitung mit definierten Filtern wie zum Beispiel Kantendetektoren zur Objekterkennung arbeiten, lernen künstliche neuronale Netze hierfür mit manuell annotierten Eingangsdaten. In diesem Fall bestehen diese Daten aus möglichst vielen unterschiedlichen Bilddaten. Vorteil hierbei ist, dass, sobald genug Trainingsdaten vorhanden sind, Ergebnisse erreicht werden können, die die der herkömmlichen Ver-

fahren übertreffen. Für Algorithmen des automatisierten Fahrens gibt es bereits umfangreiche, öffentlich zugängliche Datenbanken (s. Anhang A.2.1 auf Seite 191). In dieser Arbeit werden neuronale Netze zur Objekterkennung beziehungsweise -klassifizierung und semantischen Segmentierung von Bilddaten (Unterteilung des Bildes in verschiedene Bereiche mit gemeinsamem Inhalt oder Bedeutung. Beispielsweise Unterteilung in Straße, Hindernisse und Himmel, anderes Beispiel in Abbildung 2.5) eingesetzt.

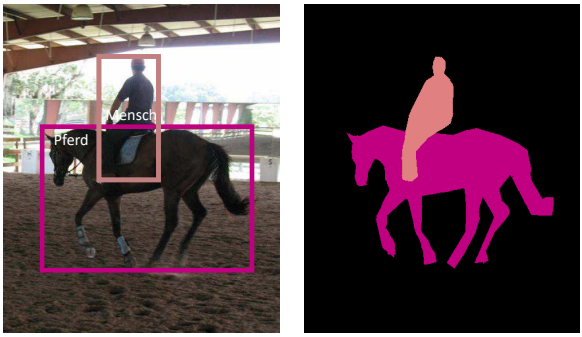


Abbildung 2.5: Beispiele für Bounding Boxes mit Label des Inhalts (a) und semantische Segmentierung nach Bildinhalt (b) [DHS15]

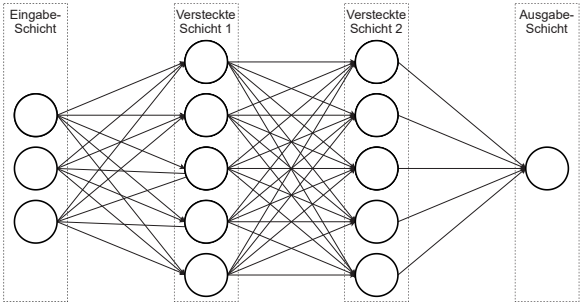


Abbildung 2.6: Beispiel für ein neuronales Netz mit zwei versteckten Schichten (auch als hidden layers bezeichnet)

Ihren Ursprung haben die heutzutage eingesetzten Netze im sogenannten *McCulloch-Pitts-Neuron*. Diese Bezeichnung geht auf ein Neuronenmodell

zurück, das von Warren McCulloch und Walter Pitts im Jahre 1943 vorgestellt wurde [MP43]. Dieses Modell erlaubt es, Vorgänge in biologischen neuronalen Netzen mathematisch abzubilden. Jedes Neuron kann zwar nur eine binäre Entscheidung treffen, diese jedoch aufgrund eines beliebigen reellen Schwellenwerts. Dazu kann jedes Neuron mehrere Eingänge besitzen, denen jeweils ein Gewicht zugeordnet ist. Übertrifft die Summe der gewichteten Eingänge den definierten Schwellenwert, wird eine 1 ausgegeben, sonst eine 0. Eine weitere dem biologischen Neuron ähnliche Eigenschaft ist die sogenannte *Inhibition*. Diese ermöglicht die Hemmung eines Neurons durch eine sogenannte inhibitorische Leitung. Ein rekursives Netz erhält man, indem man Zyklen in einen gerichteten Graphen aus diesen Neuronen einfügt, ansonsten, falls es sich um einen gerichteten Graphen ohne Zyklen handelt, spricht man von einem vorwärtsgerichteten Graphen.

In Abbildung 2.6 auf der vorherigen Seite ist ein Beispiel für ein künstliches neuronales Netz abgebildet, dessen Ebenen aus solchen McCulloch-Pitts-Neuronen bestehen, ein sogenanntes *Perzeptron*. Das Perzeptron ist ein im Jahre 1958 von Frank Rosenblatt vorgestelltes vereinfachtes künstliches neuronales Netz, das aus einem oder mehreren Neuronen mit anpassbaren Gewichten besteht. Je nach Anzahl der Ebenen, abgesehen vom Eingangsvektor, handelt es sich um ein- oder mehrlagige Perzeptren, die Ausgabeschicht (output layer) wird dabei auch als Layer gezählt. Stark vereinfacht lernt ein solches neuronales Netz, indem die Gewichte nach bestimmten Regeln, wie zum Beispiel als Reaktion auf Fehlentscheidungen, angepasst werden. [Ros58]

2.3.1 TensorFlow [AAB+16]

TensorFlow, ein System für maschinelles Lernen im großen Maßstab und in heterogenen Umgebungen, liefert Bibliotheken für dieses und abstrahiert dabei gleichzeitig von der zugrundeliegenden Hardware. So kann ein in TensorFlow definiertes neuronales Netz ohne Detailwissen auf andere Hardwareplattformen portiert werden. Damit ist eine Ausführung desselben Algorithmus auf einem Smartphone oder einem Hochleistungscluster mit GPU[19]-Beschleunigung möglich.

[19] Graphics Processing Unit

Eine Berechnung, zum Beispiel des Modells eines neuronalen Netzes, wird in TensorFlow als gerichteter Datenflussgraph dargestellt, dessen einzelne Knoten mehrere Ein- und Ausgänge haben und jeweils eine Rechenoperation darstellen. Die Kanten zwischen den Knoten stellen *Tensoren* dar, enthalten also die Daten. Diese Netze können sowohl trainiert als auch für die Objekterkennung genutzt werden. Ist ein Netz trainiert, kann sein Zustand „eingefroren“ und weitergegeben werden. Durch die weite Verbreitung von TensorFlow ist das de facto ein Standardweg zur Weitergabe von trainierten neuronalen Netzen und wird daher auch im Rahmen dieser Dissertation zur Ausführung der verwendeten neuronalen Netze eingesetzt.

2.3.2 Metriken zur Bewertung der Ergebnisse der Objekterkennung

Um geeignete Algorithmen für das Fahrzeug auswählen zu können, ist eine Bewertung dieser mittels geeigneter Metriken notwendig.

Die mathematisch einfachste Metrik zur Bewertung einer Klassifizierung ist die *Accuracy* oder *Genauigkeit*. Dabei wird der Quotient aus der Anzahl der korrekten Vorhersagen und der Gesamtanzahl der Vorhersagen gebildet:

$$\text{Accuracy} = \frac{\#\text{korrekte Vorhersagen}}{\#\text{alle Vorhersagen}} \tag{2.1}$$

Diese Metrik ist genau dann sinnvoll, wenn alle Klassen grob dieselbe Anzahl an Elementen enthalten. Ein Beispiel dafür ist eine Klassifikation mit zwei Klassen, *A* und *B*. Enthält *A* 98 Elemente und *B* nur zwei Elemente, erreicht ein Netzwerk, das immer *A* vorhersagt, bei 100 Beobachtungen eine Genauigkeit von $Acc = \frac{98}{100} = 98\%$.

Eine solche Aufteilung ist in der Realität entsprechend selten der Fall, daher existieren diesen Umstand berücksichtigende Metriken wie Präzision beziehungsweise Precison oder Recall.

Die Precision ist dabei der Anteil der korrekt zu einer Klasse zugeordneten Elemente (auch als *true positives, TP* bezeichnet) im Verhältnis zu den insge-

samt dieser Klasse zugeordneten Elementen (Summe aus *true positives, TP* und *false positives, FP*):

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.2}$$

Als Recall wird der Anteil der korrekt klassifizierten Elemente an der Gesamtzahl der tatsächlich bei der Beobachtung vorgekommenen Elemente dieser Klasse bezeichnet, daher werden auch die *false negatives, FN* betracht, also Elemente, die zur Klasse gehören, aber fälschlicherweise als nicht zugehörig vorhergesagt wurden:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2.3}$$

Dabei zeigt sich, dass ein neuronales Netz nicht auf beide Metriken optimiert werden kann, da es für eine gute Precision ausreicht, möglichst wenige Elemente und diese dafür korrekt einer Klasse zuzuordnen, also keine *false positives* zu generieren. In den seltensten Fällen werden durch ein solches Vorgehen jedoch alle Elemente dieser Klasse gefunden. Dies wäre zum Beispiel bei einer Fußgängererkennung eines Notbremsassistenten fatal, wo ein höherer Recall vorzuziehen ist.

Um eine vorhergesagte Bounding Box sinnvoll zu bewerten, sind weitere Anpassungen notwendig. Hierzu wird die sogenannte IoU[20] zur Einteilung verwendet. Diese definiert das Verhältnis der Überlappung der vorhergesagten Bounding Box mit der tatsächlichen Bounding Box der Ground Truth zur vereinigten Fläche der beiden.

$$IoU = \frac{\text{Schnittfläche}}{\text{vereinigte Fläche}} \tag{2.4}$$

Dabei wird eine vorhergesagte Bounding Box als *true positive* bewertet, wenn die IoU größer als 0,5 ist, der richtigen Klasse entspricht und kein Duplikat einer vorhergehenden Vorhersage ist.

Als *false positives* werden Bounding Boxes bezeichnet, die eine IoU kleiner als 0,5 haben oder eine andere Bounding Box derselben Klasse überdecken.

[20] Intersection over Union

Die letzte Möglichkeit stellt eine *false negative* dar, wenn entweder gar kein Objekt erkannt wurde oder die IoU der Bounding Box zwar größer als 0,5 ist, diese aber der falschen Klasse zugeordnet wurde.

True negatives treten nicht auf, da die Anzahl der nicht vorhandenen Bounding Boxes nicht zählbar ist.

Everingham et. al. [EvW+10] führen die AP[21] ein, die die Fläche unter der Precision/Recall-Kurve darstellt. Die Precision für jeden Recall-Wert r kann dazu interpoliert werden, indem die maximale gemessene Precision ausgewählt, für die der Recall r übersteigt und in folgender Gleichung zur Berechnung der interpolierten Precision p_{interp} genutzt wird, wobei $p(\tilde{r})$ die gemessene Precision bei Recall $\tilde{r}$ darstellt:

$$p_{interp}(r) = \max p(\tilde{r}), \text{ mit } \tilde{r} \geq r \tag{2.5}$$

Anschließend wird die AP als Durchschnitt dieser interpolierten Precision für elf gleichmäßig verteilte Recall-Werte zwischen 0 und 1 definiert:

$$AP = \frac{1}{11} \sum_{r \in 0,0.1,\ldots,1} p_{interp}(r) \tag{2.6}$$

Die mAP[22] ist schließlich der Durchschnitt der AP über alle Klassen, dabei werden jedoch häufig mAP und AP synonym verwendet.

2.3.3 Leistung beim Ausführen neuronaler Netze – MLPerf

Eine weitere Metrik zur Bewertung neuronaler Netze beziehungsweise Auswahl geeigneter Hardware sind die Leistungsanforderungen der Netze und damit die Performanz der Hardware bei der Berechnung dieser. MLPerf ist der Versuch, die Performanz verschiedener Systeme im maschinellen Lernen messbar zu machen und damit einen industrieweiten Standard für den Leistungsvergleich dieser zu schaffen. Dazu gehört eine Auswahl möglichst

[21] Average Precision
[22] mean Average Precision

repräsentativer Szenarien und Algorithmen, eine Architekturunabhängigkeit der eingesetzten Algorithmen und eine Reproduzierbarkeit der Ergebnisse. Um diese Ziele umzusetzen, sind Szenarien, verwendete Modelle der neuronalen Netze, zu verarbeitende Bilddaten sowie Qualitätsziele vorgegeben. Die vier Szenarien (*single-stream*, *multi-stream*, *server* und *offline*) decken dabei unterschiedliche Einsatzgebiete ab [RCK+20]:

single-stream In diesem Szenario wird immer nur ein einzelnes Bild verarbeitet, dabei wird die hierfür benötigte Zeit gemessen. Ziel ist die Abdeckung von Anwendungen, in denen die Reaktionszeit eine größere Rolle spielt als der Durchsatz. Bewertet wird das 90 %-Quantil der Latenz.

multi-stream Hier werden zu festen Zeitpunkten (im Bereich 50 bis 100 ms) mehrere Bilder in einer Abfrage auf einmal verarbeitet. Dies soll die parallele Verarbeitung der Daten mehrerer Kameras beim automatisierten Fahren nachbilden (auf dem Demonstrator dieser Dissertation werden vier Kameras parallel eingesetzt, s. Abschnitt *Sensorik* auf Seite 133). Ist die Verarbeitung der vorhergehenden Daten nicht abgeschlossen, wird dieses Intervall übersprungen und die noch nicht verarbeiteten Daten an das nächste angehängt. Dabei darf maximal 1 % der Abfragen übersprungene Intervalle bewirken. Bewertet wird die maximale Anzahl der gleichzeitigen Abfragen, die verarbeitet werden können, während dieses Maximum eingehalten wird.

server Das *server*-Szenario ist dem *multi-stream*-Szenario ähnlich, außer dass die Zeitpunkte der eintreffenden Daten nicht gleich- sondern poissonverteilt sind. Bewertet wird die maximale Anzahl an Abfragen pro Sekunde (queries per second, QPS), während eine vorgegebene Latenz (15 bis 250 ms) für mindestens 99 % der Anfragen eingehalten wird.

offline Im *offline*-Szenario werden so viele Daten wie möglich auf einmal in das System gespeist. Bewertet wird die Anzahl an verarbeiteten Bildern pro Sekunde. Um ein Konfidenzintervall von 99 % für das 99 %-Quantil zu erreichen, müssen aufgerundet 270 000 Anfragen ausgeführt werden. Da im *multi-stream*-Szenario pro Anfrage N Bilder verarbeitet werden, müssen hier

$270\,000 \times N$ Anfragen ausgeführt werden. Dies führt selbst auf leistungsstarken Systemen durch die festen Intervalle zu einer Laufzeit von mindestens 2,5 bis 7 Stunden im *multi-stream*-Szenario. Um Einflüsse von nur kurzzeitig erreichbaren Maximaltaktraten und Ähnlichem zu eliminieren, ist außerdem eine Minimallaufzeit von 60 s vorgegeben.

Modelle und Datensätze sind nach Aufgaben zusammengefasst, so ist außer der für diese Dissertation relevanten Objekterkennung und Bildklassifizierung (jeweils in einer Ausführung für leistungsarme und -starke Geräte) auch die, hier nicht weiter betrachtete, maschinelle Übersetzung im Benchmark enthalten.

Bildklassifizierung Bei der Bildklassifizierung wird ein gesamtes Bild mit einem Label versehen, das die Zugehörigkeit zu einer Klasse enthält, die das Bild am besten beschreibt. In MLPerf werden als Eingangsdaten Bilder aus dem ImageNet-Datensatz [DDS+09] genutzt, die auf eine Auflösung von 224×224 umgewandelt werden und die Accuracy bestimmt. Für das neuronale Netz wurden zwei Modelle ausgewählt: *ResNet-50 v1.5* [HZRS16] und *MobileNet-v1-224* [HZC+17]. Ersteres besitzt dabei eine höhere Accuracy bei wesentlich höheren Rechenleistungsanforderungen (Faktor 6,8 bei Operationen, Faktor 6,1 bei Parametern). Die Länge eines Intervalls ist im *multi-stream*-Szenario für beide 50 ms.

Objekterkennung Bei der Objekterkennung werden Bounding Boxes für einzelne Objekte auf Bildern erstellt, anstatt das gesamte Bild einer Klasse zuzuweisen. Üblicherweise wird dabei dennoch als Vorstufe eine Bildklassifizierung durchgeführt, um den Rechenaufwand der eigentlichen Objekterkennung zu reduzieren. Bei MLPerf wird der COCO-Datensatz [LMB+14] genutzt, für die leichte Version auf 300×300 Pixel reduziert, für die schwere auf 1200×1200 Pixel hochskaliert. Bewertet wird die mAP. Für die Objekterkennung kommt ein Single-Shot-Detektor (SSD) [LAE+16] mit entweder *ResNet-34* (schwer, 66 ms Intervall für *multi-stream*-Szenario) oder *MobileNet-v1-1.0* (leicht, 50 ms Intervall für *multi-stream*-Szenario) zum Einsatz.

3 Fahrzeugarchitektur

Die für diese Dissertation relevante Fahrzeugarchitektur besteht aus der *E/E-Architektur*[1] (s. folgender Abschnitt) und der darauf ausgeführten *SW-Architektur*[2] (s. ab Abschnitt 3.2 auf Seite 49).

3.1 E/E-Architektur

Die ISO/IEC/IEEE 42010:2011 mit dem Titel „Systems and software engineering — Architecture description“ [Int11] definiert den Begriff „Architektur“ wie folgt:

Definition 3.1.1: Architektur

Grundlegende Konzepte und Eigenschaften eines Systems in dessen Umgebung, die in seinen Elementen, Beziehungen und den Prinzipien seines Entwurfs und seiner Weiterentwicklung verkörpert sind.

Diese Definition erstreckt sich auf jegliche Art von Systemarchitektur, hier soll der Umfang jedoch auf die E/E-Architektur im Automotive-Bereich beschränkt werden (s. Abbildung 3.1 auf der nächsten Seite für den Kontext zum System (s. Definition 1.4.1 auf Seite 6) und dessen Umgebung).

Diese ist für einen Großteil der kundenerlebbaren Funktionalität in einem modernen Fahrzeug verantwortlich (laut [BKPS07] stammten bereits im Jahre 2007 über 80% der Innovationen im Fahrzeug von den integrierten Computersystemen). Damit verursacht die E/E-Architektur einen stetig wachsenden Anteil an den Entwicklungs- und Herstellungskosten eines Fahrzeuges und

[1] Elektrisch/Elektronische Architektur
[2] Softwarearchitektur

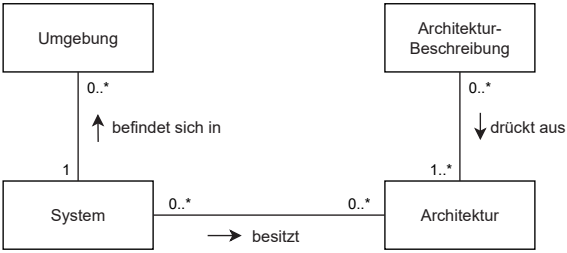


Abbildung 3.1: Kontext der Architektur, nach [Int11]

steigert auch entsprechend ihren Anteil am Gesamtumsatz der Automobilindustrie (eine Prognose bis 2030 ist in Abbildung 3.2 dargestellt). Dies entspricht einem Anteil am globalen Umsatz der Automobilindustrie von 8,64 % in 2020, 11,93 % in 2025 und 12,34 % in 2030 [McK19b].

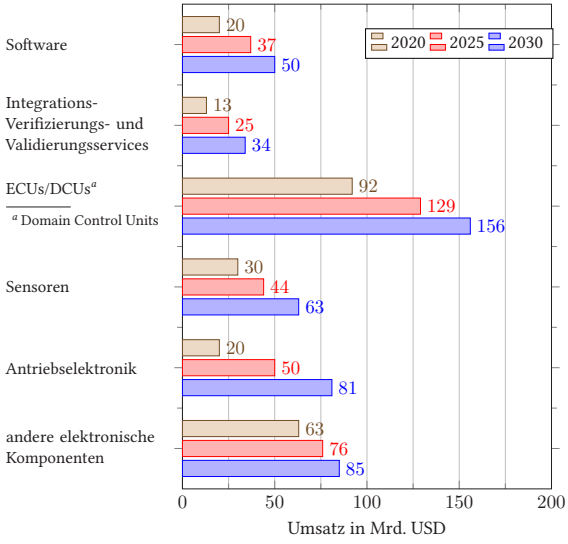


Abbildung 3.2: Umsatzprognose der weltweiten Automobilindustrie im Bereich Software sowie elektrische und elektronische Bauteile nach Segmenten in den Jahren 2020 bis 2030 [McK19a]

3.1.1 Entwicklung der E/E-Architektur

In der klassisch vom Maschinenbau abstammenden Automobilindustrie hat die Verbreitung der Elektronik im Fahrzeug erst relativ spät, das heißt in den Siebzigern des vergangenen Jahrhunderts begonnen [Sta17]. Entsprechend basieren viele E/E-Architekturen auf einem eher evolutionären Ansatz, bei dem das Fahrzeug nach und nach mit immer mehr Elektronik und Elektrik im Sinne von ECUs sowie Sensorik und Aktorik versehen wurde. Ein Beispiel sind hier die Fensterheber: Ursprünglich wurden diese rein mechanisch und manuell über Kurbeln betätigt, mit der Zeit kamen Features wie der elektrische Antrieb und Einklemmschutz hinzu. Ersterer konnte noch über einen Schalter realisiert werden, der den Motor direkt ansteuerte. Für erweiterte Funktionen wie den Einklemmschutz wurde jedoch mehr Intelligenz notwendig. Um die Position der Scheibe bestimmen zu können und damit zum Beispiel zwischen einem Einklemmfall oder dem Erreichen des Endpunktes unterscheiden zu können, ist das Auslesen und Interpretieren der Signale mehrerer Sensoren notwendig. Eine ferngesteuerte Betätigung oder ein Komfortschließen bei Regen erfordern eine Kommunikation mit weiteren Steuergeräten. Insgesamt wurde so mit steigendem Anzahl von Features im Fahrzeug (in Abbildung 3.3 auf der nächsten Seite dargestellt) auch die E/E-Architektur immer umfangreicher: mit der Anzahl der ECUs nahm auch die Anzahl der Busse und Signale auf diesen rapide zu, heutige Premiumfahrzeuge bewegen sich in der Größenordnung von 100 ECUs [Sta17]. Dadurch, dass die Signale zunehmend nicht mehr nur in wenigen Bit kodierbare Informationen wie die oben angesprochene Position des Fensterhebers, einen binären Wert des Regensensors oder Ähnliches enthielten, sondern auch Bilddaten, wuchsen und wachsen die benötigten Bandbreiten nahezu exponentiell an. Um dies in einen Kontext zu setzen, sei auf das schwere Multistream-Szenario aus MLPerf (s. Abschnitt 2.3.3 auf Seite 35) verwiesen: Dabei kommen vier Kameras parallel zum Einsatz, die ca. 15 Bilder in der Sekunde mit 1200×1200 Pixeln und 24 Bit Farbtiefe liefern. Dies entspricht einer Datenrate von rund 2,1 Gigabit in der Sekunde, insgesamt also der Nettodatenrate von über 29 000 Lowspeed- beziehungsweise über 3600 Highspeed-CAN[3]-Bussen.

[3] Controller Area Network

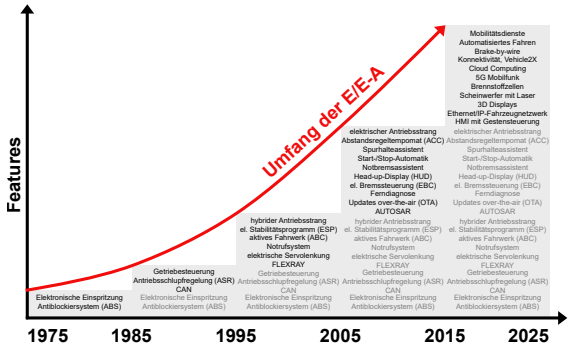


Abbildung 3.3: Umfang der E/E-A über die Zeit, nach [Sta17]

3.1.2 Struktur einer E/E-Architektur

Um diesem steigenden Umfang der E/E-Architekturen Herr zu werden, werden diverse modellbasierte ADLs[4] wie AUTOSAR[5] (s. Abschnitte 3.3.2 und 3.6.2 auf Seite 54 und auf Seite 74), EAST-ADL [EAS19] und EEA-ADL eingesetzt, letztere liefert die Basis für das Ebenenmodell in Vector PREEvision (s. Abbildung A.2 auf Seite 190) [Mat10b]. Eine genauere Untersuchung und einen Vergleich weiterer ADLs unternimmt Matheis [Mat10b]. Allgemeiner unterteilen Streichert und Traub [ST12] dabei in die in Abbildung 3.4 auf der nächsten Seite dargestellten vier Ebenen, die jeweils einer Sicht auf die E/E-Architektur entsprechen (in Abbildung A.2 auf Seite 190 sind die Entsprechungen in Vector PREEvision dargestellt):

Funktionsumfang ist die oberste Ebene und umfasst dabei alle Funktionen, die kundenerlebbar sind, das heißt die Ausstattung des Fahrzeuges. Daher werden diese Funktionen auch *Kundenfunktionen* oder *Features* genannt (s. auch Definition 1.4.3 auf Seite 7). Durch die Möglichkeit, zusätzlich zur Basisausstattung auch kundenspezifische Sonderausstattungen zu wählen, entstehen Abhängigkeiten. So ergibt es keinen Sinn, um beim Beispiel des Fensterhe-

[4] Architecture Description Languages
[5] AUTomotive Open System ARchitecture

bers zu bleiben, eine Komfortschließung bei Regen zu wählen, dabei aber auf den den Regensensor enthaltenden automatischen Scheibenwischer zu verzichten. Ebenso schließen sich auch bestimmte Ausstattungen aus, hier zum Beispiel elektrische und manuelle Fensterheber. Dabei muss jede Funktion nicht nur die kundenerlebbaren Anforderungen erfüllen, sondern auch nicht-kundenerlebbare. Erstere wirken sich direkt auf die Kundenzufriedenheit beziehungsweise dessen Bewertung der Funktion aus, letztere ergeben sich aus technischen oder rechtlichen Bedingungen.

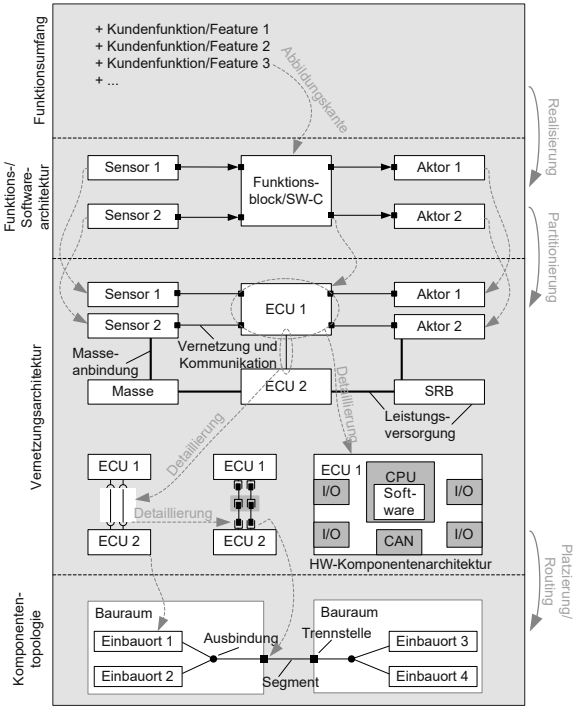


Abbildung 3.4: Die vier Systemebenen der E/E-Architektur, leicht angepasst nach [ST12]

Funktions-/Softwarearchitektur stellt die nächste Ebene dar, die den Funktionsumfang durch logische Verknüpfungen über Verbindungen von Sensor-, Aktor- und Funktionsblöcken abbildet. Letztere funktionieren hier ähnlich

wie eine mathematische Funktion und werden auch als *logische Funktionen* bezeichnet (s. auch Definition 1.4.4 auf Seite 7): Eingabedaten werden verarbeitet und das Ergebnis ausgegeben. Dabei können auch Funktionsblöcke von mehreren kundenerlebbaren Funktionen wiederverwendet werden (als Beispiel sei hier der Regensensor für die Komfortschließung der Fenster und den automatischen Scheibenwischer genannt). Auf dieser Abstraktionsebene spielt die konkrete Umsetzung dabei noch keine Rolle, nur die Schnittstelle ist wichtig. Diese beschreibt die Datenformate der Ein- und Ausgabedaten, die benötigte Bandbreite und die Art der zugehörigen Ports (Eingang oder Ausgang). Darüber hinaus kann jedoch bereits grob über die Verbindungen die Kommunikationsarchitektur definiert werden. Beispiele hierfür sind *Client-Server-Kommunikation*, hier werden einem Client die Daten auf Anfrage durch einen Server zur Verfügung gestellt, oder die *Sender-Receiver-Kommunikation*, hier werden, analog zu einer klassischen Buskommunikation, die Daten von einem Sender „ungefragt" versendet und vom Empfänger empfangen. Klassisch wird die endgültige Partitionierung auf ECUs oder Technologien wie CPU, GPU oder FPGA noch nicht abgebildet. Die beschriebene Funktions-/Softwarearchitektur entspricht daher grob dem AUTOSAR-Modell (s. Abschnitt 3.3.2 auf Seite 54), bei dem die Funktionsblöcke durch sogenannte Softwarekomponenten abgebildet werden, beziehungsweise ROS[6] mit den ROS-Nodes (vergleiche Abschnitt 3.3.3 auf Seite 56).

Vernetzungsarchitektur ist eine Ebene darunter für die Darstellung der die Softwarekomponenten der Funktions-/Softwarearchitektur ausführenden ECUs, Sensoren und Aktoren verantwortlich. Außerdem ist deren Energieversorgung und Kommunikationsstruktur abgebildet. Dabei ist die Vernetzungsarchitektur selbst weiter hierarchisch untergliedert:

- **Kommunikationsstruktur** Die Kommunikationsstruktur beschreibt die Netzwerke aus den Komponenten wie Sensoren, Aktoren und ECUs. Dabei kommen Busse wie FlexRay, CAN, LIN oder andere Netzwerktechnologien wie Ethernet, aber auch proprietäre oder dedizierte Anbindungen zum Einsatz. Dies kann zum Beispiel die Anbindung eines Sensors oder Aktors sein. Dabei vermitteln Gateways gegebenenfalls zwischen verschiedenen getrennten Bussystemen (Beispiele: Zwischen

[6] Robot Operating System

Ethernet und CAN beziehungsweise Comfort und Body CAN) (verschiedene Topologien im Laufe der Zeit, s. Abschnitt 3.1.3 auf der nächsten Seite).

- **Leistungsversorgung** Hier wird dargestellt, wie die einzelnen Komponenten mit Leistung versorgt werden. Dazu müssen der Leistungsbedarf der einzelnen Komponenten sowie die Quelle der Leistung (Batterie, Generator oder aber anderes Steuergerät) definiert werden. Dabei schließt die Leistungsverteilung auch Komponenten wie Leistungsverteiler, Massestellen und Sicherungs-/Relaisboxen ein.
- **Komponentenarchitektur** In der Komponentenarchitektur wird der Aufbau der Komponenten wie Sensoren, Aktoren oder Steuergeräten detaillierter beschrieben. Dabei liegt noch kein konkreter Schaltplan vor, es werden aber die Baublöcke der Komponenten dargestellt. Dies umfasst CPU und Speicher wie RAM[7] und ROM[8], aber auch Bauteile zur Leistungsversorgung, Betriebssystem und Kommunikationsinterfaces wie CAN. Dabei können auf dieser Ebene Varianten der Bestückung angegeben werden, um entweder verschiedene Ausstattungsumfänge zu erlauben oder auch die Abhängigkeit von einem einzelnen Zulieferer zu verringern.
- **Leitungssatz** Der Leitungssatz stellt den Übergang der Abstraktion der Kommunikationsstruktur und Leistungsversorgung von den logischen Verbindungen hinab zur konkreten Darstellung der elektrischen Verbindungen dar. Dabei bestimmt die elektrische Verbindung die zur Umsetzung der logischen Verbindung notwendigen Parameter wie Anzahl der Pins, Leitungs- und Kabeltypen sowie -längen. Schließlich wird im Leitungssatz die Zuordnung der elektrischen Verbindungen auf die einzelnen Stecker des Leitungssatzes vorgenommen.

Ein Beispiel für das resultierende Bordnetz ist in Abbildung 3.5 auf der nächsten Seite dargestellt.

[7] Random Access Memory
[8] Read-only Memory

Abbildung 3.5: Bordnetz einer E/E-Architektur [MBB18]

Komponententopologie ist schlussendlich die niedrigste Ebene. Sie definiert Bauräume für sowohl die Komponenten als auch die Platzierung des Leitungssatzes, letztere werden dabei als Segmente bezeichnet. Dabei können Segmente durch Parameter wie die Länge und den maximal nutzbaren Bündeldurchmesser des Leitungssatzes beschrieben werden.

3.1.3 Topologien von E/E-Architekturen

Im Laufe der Zeit haben sich mit der Entwicklung der Technologie und der im Fahrzeug vorhandenen Funktionalitäten auch unterschiedliche Netzwerktopologien der E/E-Architektur entwickelt.

Klassisch – Verteilte Architektur

Der „klassische“ Ansatz besteht darin, jede einzelne Funktion auf ein eigenes Steuergerät zu partitionieren (s. Abbildung 3.6 auf der nächsten Seite). Die einzelnen ECUs sind über einen gemeinsamen Bus verbunden (wie zum Beispiel CAN), interagieren dabei aber meist nicht miteinander. Ein Beispiel hierfür ist der elektrische Fensterheber (s. Abschnitt 3.1.1 auf Seite 41). Vorteil dieser Topologie ist der einfache Aufbau, da sie durch die Trennung der Funktionen sehr modular ist. Allerdings steigt hierdurch der Umfang der E/E-Architektur und die Anzahl der nötigen Steuergeräte mit neu hinzugefügten Funktionalitäten stark an. [BP20]

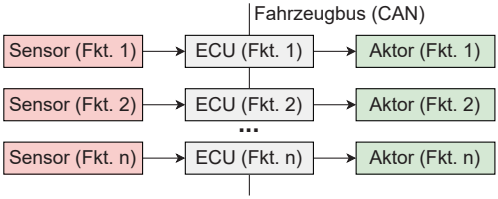


Abbildung 3.6: Verteilte E/E-Architektur

Heute – Domänenorientierte Architektur

Da heutzutage vermehrt Features implementiert werden, die aus zusammengesetzten Funktionen auf mehreren ECUs bestehen, wird eine domänenorientierte Topologie eingesetzt. Damit geht die Entwicklung in die Richtung einer *Funktionsbasierung*: Die Software wird von der Hardware abstrahiert, es wird logisch nach Funktionen unterteilt, die gegebenenfalls aus der Zusammenarbeit mehrerer ECUs bestehen können, anstatt nach einzelnen Steuergeräten. [BP20]

Abbildung 3.7 auf der nächsten Seite zeigt eine solche Kommunikationsstruktur, ein zentrales Gateways verbindet die einzelnen Domänen miteinander. Die Domänen selbst sind jeweils entweder direkt oder über ein als Gateway fungierendes Steuergerät (dem Domänencontroller) an einen Domänen-eigenen Bus angebunden: Einfache Sensorik und Aktorik wie Regensensor oder Sitzverstellung sind über einen LIN-Bus mit dem Body-Steuergerät verbunden, das mitsamt der anderen Steuergeräte in der Body-Domäne über den Body-CAN (B-CAN in der Abbildung) mit dem zentralen Gateway und damit dem Rest des Fahrzeuges verbunden ist.

Der im vorherigen Abschnitt genannte elektrische Fensterheber kann so zum Beispiel auf Sensordaten des Regensensors für den automatischen Scheibenwischer zugreifen und damit bei Regen selbstständig das Fenster schließen.

Hierdurch kann der Umfang des Kabelbaums gegenüber der verteilten Architektur verkleinert werden, über entsprechende Gateways wird auch eine Kommunikation mit dem Internet ermöglicht.

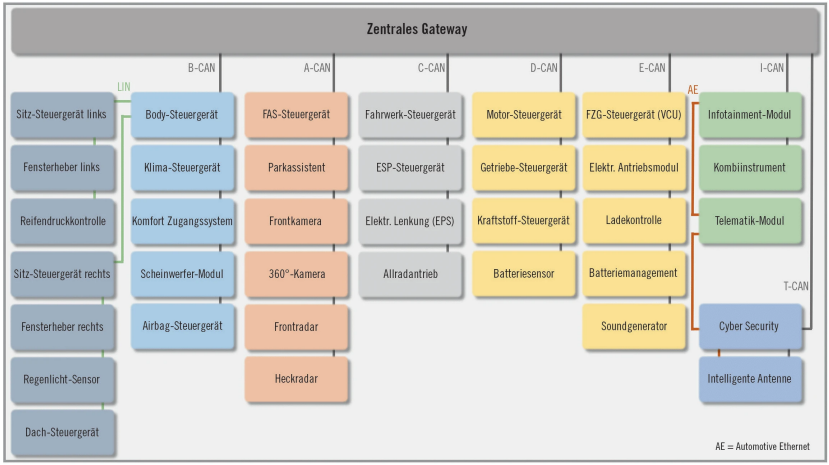


Abbildung 3.7: Topologie der domänenorientierten EE-Architektur „Base2017“ von EDAG [MBB18]

Zukunft – Zentralisierte Architektur

Zukünftig werden vermehrt zentralisierte Architekturen eingesetzt werden (wie auch im Rahmen dieser Dissertation, s. Abschnitt 4.2.1 auf Seite 100). Die Fusion von Sensordaten aus unterschiedlichen Quellen ermöglicht ein systemisches Verständnis und dient als Grundlage für die Umsetzung fortgeschrittener Features wie dem automatisierten Fahren. Dabei werden Konzepte aus der IT auf den Automotive-Bereich angewandt: Weniger, dafür aber leistungsfähigere ECUs führen die der Funktionalität zugrundeliegenden Softwarekomponenten aus, diese können je nach Bedarf und Leistungsanforderungen verteilt und ausgeführt werden. [BP20] Mit der dynamischen Verteilung von Softwarekomponenten werden neuartige Redundanzkonzepte und Skalierungsmöglichkeiten erschlossen, wie sie auch für die Realisierung der (re-)konfigurierbaren Fahrzeugarchitektur eingesetzt werden.

Anders als bei den verteilten und domänenorientierten Topologien, die größtenteils eine signalbasierte SW-Architektur einsetzen, kommt bei einer zen-

tralisierten Architektur vorzugsweise eine SOA[9] zum Einsatz (s. Abschnitt 3.4 auf Seite 58).

Ein Beispiel für eine Zwischenstufe aus domänenorientierter und zentralisierter Architektur ist die *Zonenarchitektur* (s. Abbildung 3.8). Anders als bei der domänenorientierten Architektur erfolgt die Gruppierung der Komponenten nicht auf funktionaler Basis, sondern entsprechend ihrer Position im Fahrzeug (die Zonen sind in der Grafik durch gestrichelte Linien dargestellt). Um Kommunikationsoverhead zu verringern, erfolgen alle Berechnungen zentral auf einem leistungsstarken Steuergerät, dem sogenannten *Server*. Die Zonencontroller selbst dienen nur dem Empfang von Daten der angeschlossenen Sensoren und der Weiterleitung dieser an den Server, beziehungsweise umgekehrt der Ansteuerung der Aktorik basierend auf Anweisungen des Servers. Durch die Verwendung möglichst identischer Zonencontroller, im Gegensatz zu spezialisierten ECUs bei einer verteilten oder domänenorientierten Architektur, sollen durch höhere Stückzahlen Kosten gesenkt werden. [BRKW17]

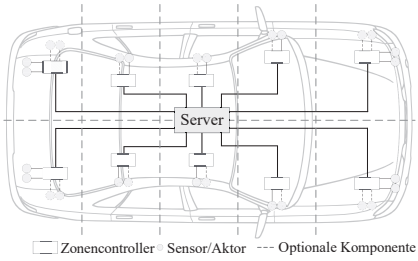


Abbildung 3.8: Die Zonenarchitektur, nach [BRKW17]

3.2 Software-Architektur

Im Rahmen dieser Dissertation ist die E/E-Architektur nicht mehr strikt in *Funktions-/Software-Architektur* und *Vernetzungsarchitektur* getrennt, da die Softwarearchitektur beziehungsweise SW-Architektur viele der Partitionie-

[9] serviceorientierte Architektur

rungsaufgaben und damit das Zuweisen von Softwarekomponenten zu ECUs und das Herstellen von Kommunikationsverbindungen selbst dynamisch übernimmt.

Bass et al. [BCK13] definieren eine SW-Architektur wie folgt und damit als konsequente Erweiterung der allgemeinen Definition einer Architektur (s. Definition 3.1.1 auf Seite 39):

Definition 3.2.1: Software-Architektur

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

In Abbildung 3.9 ist der grundlegende Aufbau einer SW-Architektur dargestellt. Dabei erfolgt der Aufbau schichtweise, mit steigender Abstraktion von der ECU-Hardware ausgehend. Die Hardwareabstraktionsebene erlaubt es, dass die Laufzeitumgebung möglichst generisch für unterschiedliche Hardwareplattformen gebaut werden kann. Die Laufzeitumgebung beinhaltet die sogenannte *Middleware* (s. nächster Abschnitt. Realisierungen, s. Anhang A.2.2 auf Seite 196), die von der Netzwerktopologie abstrahiert und so die Kommunikation von Komponenten auf Netzwerk- und ECU-Ebene erlaubt. Auf der Laufzeitumgebung läuft schließlich die Applikations-Software, die die kundenerlebbare Funktionalität zur Verfügung stellt.

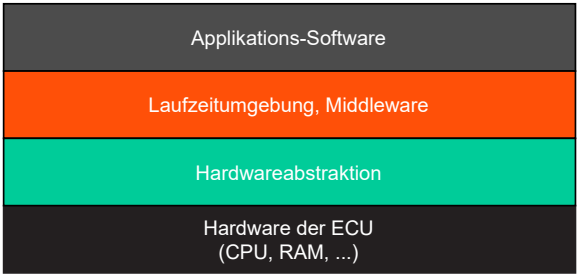


Abbildung 3.9: Grundaufbau einer Softwarearchitektur

3.2.1 Das OSI-Schichtenmodell und die Middleware

Das OSI[10]-Referenzmodell wurde als Basis entwickelt, die Kommunikation von Systemen durch eine Schichtenarchitektur zu beschreiben und die Entwicklung von Standards so zu erleichtern. Es existieren sieben Schich-

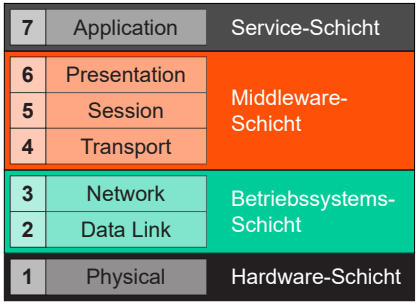


Abbildung 3.10: OSI-Schichten, nach [ITU94]

ten, mit steigendem Abstraktionsgrad von Schicht 1 bis Schicht 7 (s. Abbildung 3.10): [ITU94][Wik21]

Dabei dienen die unteren vier Schichten der Übertragung von Informationen, vom *physical layer*, der die Übertragung auf unterster Ebene darstellt, zum Beispiel in Form von elektrischen Signalen auf einem Leiter, bis hin zur Transportschicht. Diese segmentiert den Datenstrom und vermeidet Staus, insgesamt wird so eine fehlerfreie Übertragung sichergestellt.

Die oberen drei Schichten werden auch als anwendungsorientierte Schichten bezeichnet. Ihre Aufgaben umfassen die Kommunikation und den Datenaustausch von Anwendungen untereinander, bis hin zur obersten Schicht, dem *application layer*. Diese stellt Funktionen für Anwendungen zur Verfügung, wie zum Beispiel die Ein- und Ausgabe.

Dabei kann das Modell auf unterschiedlichste Systeme angewandt werden, also auch auf die SW-Architektur. Dazu werden hier die Schichten so zusam-

[10] Open Systems Interconnection Model

mengeführt, dass sie auf die Ebenen in Abbildung 3.9 auf Seite 50 abgebildet werden können (s. Tabelle 3.1).

OSI-Schichten	zusammengefasste Schicht	SWA-Ebene (Abbildung 3.9 auf Seite 50)
7 (Application)	Service-Schicht	Applikations-Software
6 (Presentation), 5 (Session), 4 (Transport)	Middleware-Schicht	Laufzeitumgebung, Middleware
3 (Network), 2 (Data Link)	Betriebssystems-Schicht	Hardwareabstraktion
1 (Physical)	Hardware-Schicht	Hardware der ECU

Tabelle 3.1: Mapping der OSI-Schichten auf die Ebenen einer SWA

Auf der obersten Schicht, der *Service*-Schicht, werden die Softwarekomponenten oder im Falle der serviceorientierten (re-)konfigurierbaren Fahrzeugarchitektur die Services ausgeführt. Darunter befindet sich die Middleware-Schicht mit der Middleware. Zum Umfang dieser Schicht existieren unterschiedliche Auffassungen, im Kontext dieser Dissertation reicht sie von Schicht 4 bis Schicht 6.

Hier wird die Transportschicht (4) auch noch zur Middleware gezählt, da letztere auch für im Embedded-Bereich essentielle zur Transportschicht gehörende Funktionalitäten zuständig ist, zum Beispiel QoS[11] mit unterschiedlichen Profilen (s. Abschnitt 3.6.1 auf Seite 72), um die Zuverlässigkeit der Datenübertragung sicherzustellen. Auf der *Session*-Schicht (5) erfüllt die Middleware Aufgaben wie das Verwalten von Sitzungen und Verbindungen der Kommunikationsteilnehmer, wodurch diese miteinander kommunizieren können. Die oberste Schicht der Middleware (*Presentation*, 6) erfolgt die (De-)Serialisierung von Daten, um sie über das Netzwerk übertragen zu können. Dabei erfolgt die Übersetzung in die abstrakten Nachrichtentypen, wie sie von den auf der Service-Schicht laufenden Anwendungen und Services verwendet werden (Beispiel ROS2[12] in Abschnitt 3.6.1 auf Seite 68).

[11] Quality of Service
[12] Robot Operating System 2

3.3 Signalbasierte Software-Architekturen

3.3.1 OSEK/VDX [KJ]

Das Projekt OSEK[13] wurde im Jahre 1993 von der deutschen Automobilindustrie als der erste gemeinsame Standard für eine fahrzeugweite Softwarearchitektur unter Berücksichtigung der Verteiltheit der einzelnen Steuergeräte gestartet. Dabei werden Kommunikationsprotokolle und -services (*OSEK-COM*), das Netzwerkmanagement (*OSEK-NM*) und außerdem die dazugehörigen Betriebssysteme *OSEK-TIME* (Echtzeit) und *OSEK-OS* (keine Echtzeit) spezifiziert. Mit dem Beitritt von Peugeot und Renault erfolgte im Jahre 1994 die Umbenennung in OSEK/VDX.

Ziel von OSEK ist es, basierend auf Standardschnittstellen und -protokollen eine strukturierte und modulare Software so zu entwickeln, dass sie gleichzeitig portabel, wiederverwendbar und erweiterbar ist. Dabei bedeutet Portabilität, dass die Software ohne umfangreiche Anpassungen auf einer anderen Zielhardware lauffähig ist, zum Beispiel wenn auf eine neue Hardware-Revision gewechselt werden soll. Während Wiederverwendbarkeit die Möglichkeit zur Weiterverwendung existierender Software meint, bezieht sich die Erweiterbarkeit hier darauf, dass zusätzliche Funktionen auf bestehenden Steuergeräten implementiert werden können. Eine der Hauptanforderungen des Konsortiums ist dabei, dass mehrere Funktionen konkurrierender Zulieferer gemeinsamsam ungehindert zeitgleich auf derselben ECU lauffähig sein müssen. Von diesen Konzepten ist das Zeitverhalten explizit ausgenommen, sie beschränken sich alleine auf die funktionale Ebene.

Um eine Echtzeitfähigkeit zu gewährleisten, sind dabei alle OSEK-Module statisch konfiguriert, skaliert und auf die Hardware partitioniert. Drei Ausführungsebenen (Interruptebene, Betriebssystemebene und Taskebene, mit absteigender Priorität) sind vorgesehen. Dabei verfügen die Tasks insgesamt über die geringste Priorität, können daher den Rest nicht blockieren, aber noch weiter individuell manuell priorisiert werden. Für den Zugriff auf gemeinsam genutzte Ressourcen gibt es dabei Funktionen des Betriebssystems, ebenso

[13] Offene Systeme und deren Schnittstellen für die Elektronik im KFZ

wie für das Management der Tasks. Durch diesen statischen Ansatz lässt sich die Echtzeitfähigkeit auch auf schwächerer Hardware gewährleisten.

Der Aufbau von OSEK ist in Abbildung 3.11 dargestellt. Dabei ist OSEK nach dem OSI-Referenzmodell in Schichten aufgebaut (s. Abschnitt 3.2.1 auf Seite 51).

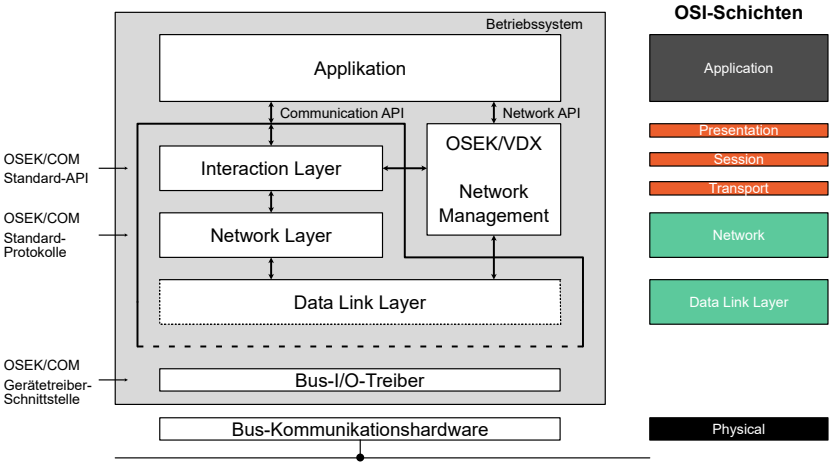


Abbildung 3.11: Ebenen der OSEK/VDX Architektur, nach [KJ]

3.3.2 AUTOSAR Classic [AUT19a]

Die AUTOSAR Classic Platform [AUT19a] basiert auf den OSEK-Prinzipien und ist dabei ähnlich statisch aufgebaut und auf die Hardware partitioniert wie dieses.

Dabei liegt bei AUTOSAR Classic[14] anders als bei AUTOSAR Adaptive[15] (s. Abschnitt 3.6.2 auf Seite 74) der Fokus auf eher einfachen, echtzeitfähigen ECUs. Im Kontext von AUTOSAR Classic ist dabei eine eng mit Sensorik und

[14] AUTOSAR Classic Platform
[15] AUTOSAR Adaptive Platform

Aktorik verbundene Mikrocontroller-Plattform gemeint, mit typischerweise 16 oder 32 Bit-CPU und Verbindung zum Fahrzeugnetzwerk. Dabei kommt ein Echtzeitbetriebssystem zum Einsatz, dessen Programme von internem oder externem Flash-Speicher ausgeführt werden. Die Standardmodule können in der Funktionalität erweitert werden, dazu muss die Konfiguration von AUTOSAR Classic entsprechend angepasst werden. Weitere Ebenen können jedoch nicht hinzugefügt werden. Die Einteilung in Abstraktionsebenen ist in Abbildung 3.12 dargestellt.

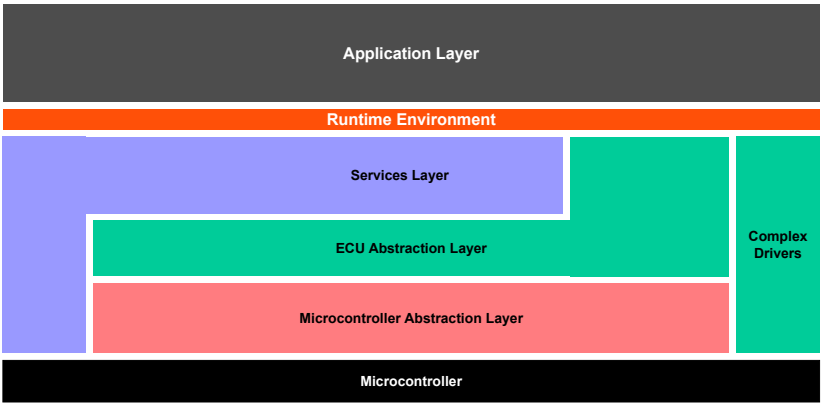


Abbildung 3.12: Aufbau von AUTOSAR Classic [AUT19a]

Services Layer, *ECU Abstraction Layer*, *Microcontroller Abstraction Layer* und *Complex Drivers* bilden zusammen die BSW[16]. Dabei ist zu betonen, dass es sich trotz der irreführenden Bezeichnung nicht um Services im Sinne von SOA handelt, sondern eher um allgemeine Systemdienste.

Der *Microcontroller Abstraction Layer* stellt die unterste Ebene dar und enthält Gerätetreiber zur Ansteuerung der Peripherie des Mikrocontrollers. Diese Peripherie beinhaltet interne Komponenten wie Verschlüsselungsfunktionen, aber auch Hardware für ECU-interne (zum Beispiel per SPI) Kommunikation oder externe Kommunikation mit dem Fahrzeug (zum Beispiel per CAN).

[16] AUTOSAR Basic Software

Damit ist diese Ebene selbst stark hardwareabhängig und stellt nach oben ein Mikrocontroller-unabhängiges Interface zur Verfügung.

Auf der Ebene *ECU Abstraction Layer* werden die Treiber des *Microcontroller Abstraction Layers* mit den Treibern der anderen auf der ECU verbauten Komponenten verbunden und nach oben hin abstrahiert. Hierdurch ist die darüber liegende Ebene von der ECU-Hardware unabhängig.

Der *Complex Drivers* Layer befindet sich parallel zu den beiden vorgenannten Ebenen bis hoch zum *Runtime Environment*. Ziel ist es, Geräte ohne Umweg über die AUTOSAR-Abstraktion einzubinden. Dies kann bei sehr hohen Echtzeitanforderungen oder bei noch nicht auf AUTOSAR portierten Treibern notwendig sein.

Die oberste Ebene *Services Layer* stellt Betriebssystemfunktionen zur Verfügung und übernimmt die Verwaltung des ECU-Zustands, des Fahrzeugnetzwerkes und die Kommunikation. Davon abgesehen werden den Ebenen darüber Basisdienste für Anwendungen und das *Runtime Environment* zur Verfügung gestellt.

Das *Runtime Environment (RTE)* stellt Kommunikationsdienste für die Applikationssoftware zur Verfügung, diese besteht aus AUTOSAR Softwarekomponenten und/oder Sensor-/Aktor-Komponenten. Diese können über das *RTE* miteinander oder mit anderen Diensten, sowohl innerhalb von als auch zwischen ECUs, kommunizieren.

3.3.3 Robot Operating System (ROS)

Anders als sein Nachfolger ROS2 (s. Abschnitt 3.6.1 auf Seite 65), ist das Software-Framework ROS [ros18] hauptsächlich auf den Betrieb von Robotern ausgelegt. Der Begriff „Roboter“ reicht dabei von Industrierobotern bis zu humanoiden Forschungsrobotern. Für den Einsatz im Automotive-Bereich ist es zum Beispiel wegen der fehlenden Echtzeitfähigkeit und anderer Limitierungen nur eingeschränkt geeignet (s. auch Abschnitt 3.6.1 auf Seite 65).

Die Schichten von ROS sind in Abbildung 3.13 dargestellt. Auf der Applikations-Ebene werden die ROS-Nodes und der ROS-Master ausgeführt. Letzterer ist dabei für die Koordination der Nodes und der Kommunikation zuständig. In der Middlewareebene befinden sich die Komponenten von ROS, die für

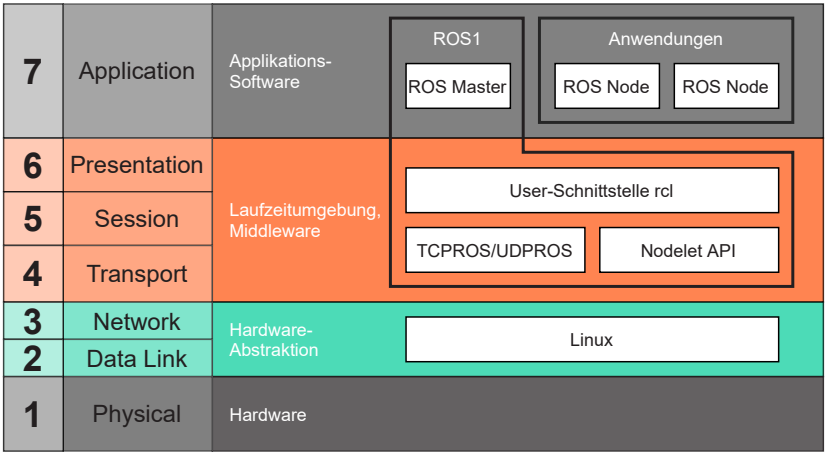


Abbildung 3.13: Ebenen von ROS1 und die darin enthaltenen OSI-Schichten, nach [MKA16]

die Kommunikation zuständig sind und nicht nur erlauben, Nachrichten zwischen den Nodes zu versenden, sondern auch diese aufzuzeichnen (in sogenannten *rosbags*) und zu einem späteren Zeitpunkt wieder abzuspielen. Das ist besonders hilfreich, wenn zum Beispiel Algorithmen und Änderungen an diesen mit zuvor aufgezeichneten Realdaten überprüft werden sollen, ohne ansonsten etwas am System zu ändern. Die Nachrichtenübermittlung folgt dabei dem *publish-subscribe-Prinzip*: Eine Softwarekomponente, der sogenannte *ROS-Node*, stellt einen Nachrichtenkanal zur Verfügung, den andere Nodes abonnieren können. Ähnlich wie bei einer signalbasierten Architektur werden Nachrichten so gleichzeitig asynchron an alle Abonnenten verteilt. Außer den Nachrichten können RPCs[17] durchgeführt werden, die zusätzlich zum asynchronen publish-subscribe-Prinzip der Nachrichten eine synchrone Anfrage/Antwort-Interaktion zwischen Prozessen erlaubt. Zur Aufbewahrung von gemeinsam genutzten Werten existiert ein Parameterserver, auf dem die Systemparameter global gespeichert und angepasst werden können. Dabei stellt die *Client Library* eine Abstraktion von den zugrundeliegenden Transportmechanismen *TCPROS* und *UDPROS* sowie der *Nodelet API* zum effizienteren Datenaustausch innerhalb von Prozessen dar [MKA16].

[17] Remote Procedure Calls

Von dieser Middleware abgesehen stellt ROS einige weitere Komponenten speziell für Roboter zur Verfügung. Beispielsweise werden Standardnachrichten zur Kommunikation vordefiniert, die bereits einen breiten Anwendungsbereich abdecken. Zusätzlich existieren Bibliotheken und Komponenten, die geometrische Aufgaben vereinfachen, zum Beispiel das Umrechnen der Sensordaten aus und in verschiedene Koordinatensysteme, die sich durch die (veränderliche) Lage der Sensoren zueinander ergeben. Eine Beschreibungssprache für Roboter (*Unified Robot Description Format*) ermöglicht es, den Roboter oder das Fahrzeug für alle Komponenten nutzbar zentral zu definieren. Auch mechanische beziehungsweise physikalische Eigenschaften lassen sich so darstellen. Diese Abstraktionen ermöglichen es, Nodes mit vergleichsweise geringen Anpassungen auszutauschen. Eine umfangreiche Sammlung solcher Softwarekomponenten wird kostenlos im Quellcode von der ROS-Community bereitgestellt: zum Beispiel Lokalisierung-, Kartographierungs- oder Navigationsstacks, aber auch Treiber für verschiedenste Hardwarekomponenten. Nachteil ist allerdings, dass meist kein (professioneller) Support verfügbar ist.

3.4 Von der signalbasierten funktionalen Architektur zur serviceorientierten Architektur (SOA)

Wie in Abschnitt 3.1.2 auf Seite 42 beschrieben, wird die Software-Architektur klassisch von einer funktionalen Sicht bestimmt. In dieser werden die einzelnen Funktionen (s. Definition 1.4.4 auf Seite 7) des Fahrzeuges und ihre Beziehungen untereinander betrachtet. Hierbei kann die Anzahl der Funktionen schnell die Tausendermarke erreichen und überschreiten [Sta17]. Ein großer Teil der neu hinzukommenden Funktionen stammt dabei von der Umsetzung neuer Technologien im Automobil wie dem automatisierten Fahren oder der Elektrifizierung, aber auch der Vernetzung des Fahrzeuges mit seiner Umwelt. Heutzutage ist die Software-Architektur eines einzelnen Steuergerätes schichtweise aufgebaut, die Applikations-Software enthält als oberste Schicht schließlich die von der Hardware abstrahierten kundenerlebbaren Funktionen (vgl. Abschnitt 3.2 auf Seite 49).

Dabei geht diese Hardwareabstraktion jedoch noch nicht notwendigerweise so weit, dass die Funktionen auf verschiedenen ECUs ausgeführt werden kön-

nen, zum Beispiel aufgrund unterschiedlicher Prozessorarchitekturen. Auch findet die Kommunikation größtenteils noch signalbasiert statt. Das heißt, ein bestimmter Wert wird erfasst und per Broadcast an die anderen ECUs ausgegeben. Das ist solange effizient genug, wie nur zyklische Daten begrenzter Größe zirkulieren. Außerdem ist die *Kommunikationsmatrix* hier statisch festgelegt und nicht zur Laufzeit anpassbar. Diese Form der Kommunikation wird noch bei OSEK (s. Abschnitt 3.3.1 auf Seite 53) und AUTOSAR Classic (s. Abschnitt 3.3.2 auf Seite 54) eingesetzt, da diese auf dafür entwickelte Bussysteme wie CAN oder FlexRay ausgelegt sind.

Leistungsfähigere Kommunikationssysteme wie Ethernet, die wesentlich höhere Bandbreiten liefern können und eine echte Punkt-zu-Punkt-Kommunikation erlauben, ermöglichen neue Ansätze. Dabei fallen (s. Abschnitt 3.1.1 auf Seite 41) immer größere Datenmengen an, die effizient verteilt werden müssen. Ebenso stammen die Daten immer häufiger nicht nur aus einer einzelnen Quelle und die einzelnen Datenpakete können dynamische Größen und Inhalte besitzen. Daher muss auch die Kommunikationsmatrix entsprechend flexibel sein, besonders wenn neue Komponenten dynamisch hinzukommen.

Um diese Herausforderungen zu bewältigen, eignen sich SOA. Sie erlauben dabei Steuergeräten oder Sensoren, sogenannte *services* anzubieten (sie sind dann Service-*Anbieter* beziehungsweise *-Provider*) und Interessenten (Service-*Konsumenten* beziehungsweise *-Consumer*), diese zu abonnieren. Dabei werden die Daten nur an diese Abonnenten gesendet und dadurch überflüssiger Overhead vermieden. Dies ermöglicht darüber hinaus weitere Konzepte wie QoS, wobei Informationen über die Qualität der Daten aus einer Quelle vorhanden sind und gegebenenfalls, bei einem Ausfall oder nicht ausreichender Qualität, auf eine andere Quelle ausgewichen werden kann, die dieselben oder ähnliche Daten liefert. Die Services können dabei dynamisch angeboten und zur Laufzeit des Systems konfiguriert und potentiellen Abonnenten bekannt gemacht werden. Dadurch ist es sogar möglich, Funktionen dynamisch auf andere Steuergeräte zu verlagern. SW-Architekturen, die das SOA-Konzept unterstützen, sind AUTOSAR Adaptive (s. Abschnitt 3.6.2 auf Seite 74) und ROS2 (s. Abschnitt 3.6.1 auf Seite 65).

Als weitere Spezialisierung der Definitionen 3.1.1 und 3.2.1, definieren Krafzig et. al. [KBS09] wie folgt:

Definition 3.4.1: SOA

A Service-Oriented Architecture (SOA) is a software architecture that is based on the key concepts of an application frontend, service, service repository, and service bus.

Eine mathematisch tiefergehende Definition und einen Ansatz, allgemeine SOA formal zu verifizieren, präsentieren Malkis und Marmsoler in [MM15]. Dabei unterteilen sie die Architektur in semantisch unabhängige Schichten, um die dadurch entstehende Komplexität zu reduzieren.

Für den Automotive-Bereich stellen Kugele et. al. in [KOB+17] die αSOA vor. Dort wird, anders als in dieser Dissertation, wo der Fokus auf einer universell anwendbaren, dafür (re-)konfigurierbaren Architektur liegt, der Schwerpunkt auf die Safety-Kritikalität von automatisiertem Fahren gelegt. Dazu wurde, basierend auf Erkenntnissen aus einer Befragung von Systemarchitekten ein formales Service-Modell und ein Framework für eine Automotive SOA erstellt, basierend auf den semantisch unabhängigen Schichten aus [MM15] (s. vorheriger Absatz). Zwei Drittel der im Jahre 2016 interviewten Systemarchitekten hielten eine vollständige Umstellung auf Serviceorientierung in den folgenden 3 bis 8 Jahren für denkbar, ein Drittel rechnete mit mindestens 8 Jahren.

3.5 Ebenen und Komponenten einer serviceorientierten Architektur

Die grundlegenden Ebenen einer SW-Architektur (s. Abbildung 3.9 auf Seite 50) bleiben auch bei einer SOA bestehen, allerdings kann hier die Ebene der Applikations-Software noch weiter unterteilt werden (s. Abbildung 3.14 auf der nächsten Seite). Der Konsument greift auf die Services zu, die schlussendlich von den Softwarekomponenten ausgeführt werden. Von diesen grundlegenden Softwarekomponenten (in ROS2 sind das die ROS-Nodes) abstrahiert die Service-Komponenten-Schicht, über der dann die eigentlichen Services liegen, dabei können diese atomar oder zusammengesetzt aus mehreren atomaren Services sein. Auf der Prozessebene erfolgt die Verschaltung und Verteilung der Prozesse zur Gesamtfunktionalität, die den Konsumenten angeboten wird.

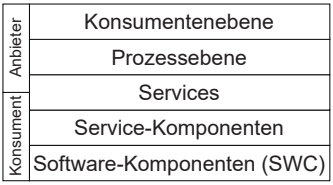


Abbildung 3.14: Ebenen einer SOA, angelehnt an [The21]

Die Komponenten der SOA nach Krafzig et. al. [KBS09] sind dabei wie in Abbildung 3.15 dargestellt verknüpft, in Klammern die Verweise auf die entsprechenden auf dieser Quelle basierenden Abschnitte.

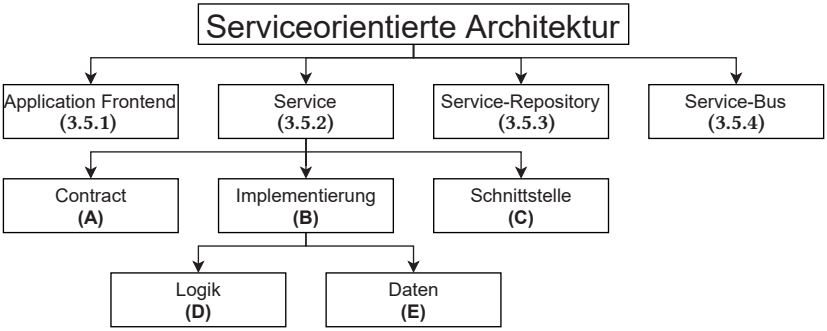


Abbildung 3.15: Komponenten einer SOA, nach Krafzig et. al. [KBS09]

3.5.1 Application Frontend

Das Application Frontend stellt eine Schnittstelle der Architektur zur Außenwelt dar. Dazu initiiert es Prozesse und empfängt deren Resultat. In einem Fahrzeug entspricht das Application Frontend den Schnittstellen zum Fahrer, also Pedale, Lenkrad oder Benutzeroberflächen auf Displays.

3.5.2 Service

Der Service ist eine logische Repräsentation einer wiederholbaren Aktivität mit einem spezifizierten Ergebnis, die in sich geschlossen ist. Dazu besteht der Service aus den Schnittstellen, dem Contract, der Implementierung, der Logik und gegebenenfalls aus zugehörigen Daten (s. Abbildung 3.16).

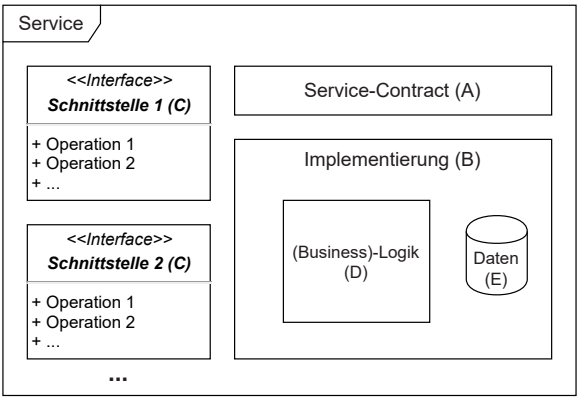


Abbildung 3.16: Komponenten eines Services, nach Krafzig et. al. [KBS09]

Die Funktionalität des Services wird durch die Schnittstelle **(C)** nach außen zum Client hin dargestellt und abstrahiert. Die Beschreibung an sich ist im Contract **(A)** definiert (Beispiel .msg-Datei bei ROS2, s. Erläuterung 3.6.1 auf Seite 68. In AUTOSAR Adaptive vom *Communication Management* definiert, s. Seite 80), zusätzlich muss jedoch auch getrennt von diesem die physikalische Implementierung der Schnittstelle definiert werden.

Der Contract definiert die Schnittstelle idealerweise möglichst formal, in einer Sprache wie IDL[18] (bei DDS[19] und damit auch ROS2 und AUTOSAR Adaptive, s. Anhang A.2.2 auf Seite 196) oder WSDL[20]. Durch diese Formalisierung wird der Grad der Abstraktion erhöht und eine optimale Technologieabhängigkeit erzielt. Dies bedeutet, dass keine Abhängigkeiten von einer bestimmten

[18] Interface Definition Language
[19] Data Distribution Service
[20] Web Service Description Language

Programmiersprache, Middleware, Laufzeitumgebung oder einem definierten Netzwerkprotokoll besteht. Außerdem sind im Contract Informationen zur Semantik von Funktionalität und Parametern enthalten.

Durch unterschiedliche Entwicklungsstände auf den Fahrzeugkomponenten kann es geschehen, dass Contracts nicht mehr kompatibel sind. Als Lösungsansatz bietet AUTOSAR Adaptive eine Versionierung von Contracts mit in einer Version einheitlichen Schnittstellen und Funktionalitäten, wobei sowohl Services als auch Komponenten mehrere Versionen unterstützen können (s. *Communication Management* in Abschnitt 3.6.2 auf Seite 78).

Als Implementierung **(B)** wird die technische Realisierung bezeichnet, die den Contract eines Service erfüllt, zum Beispiel in ROS2 die *Nodes* (s. Abschnitt 3.3.3 auf Seite 56). In der Implementierung ist die Logik enthalten, wobei diese durch Service-Schnittstellen verfügbar gemacht wird. Ist der Service *stateful*, sind auch die Daten **(E)** in der Implementierung enthalten. Häufig sind Services jedoch *stateless*, das heißt, dass sie keinen internen Speicher besitzen und daher bei jedem Aufruf mit demselben Zustand starten. Dabei kann jedoch ein anderer Service die benötigten Daten zur Verfügung stellen. Sobald ein Service dazu dient, anderen Services Daten zur Verfügung zu stellen oder selbst Daten verarbeitet, sind auch die Daten Teil des Services.

Dies ist besonders bei sogenannten *Microservices* relevant. Diese stellen eine Spezialisierung der SOA mit feinerer Granularität dar. Dabei sind die Services voneinander unabhängig, wodurch die Ausfallsicherheit des Gesamtsystems erhöht werden kann. Diese Unabhängigkeit bedingt aber auch, dass Datenspeicher nicht oder nur eingeschränkt gemeinsam genutzt werden können und Services daher über lokale Datenspeicher verfügen. Die lokalen Datenspeicher erschweren dabei die Sicherstellung der Datenkonsistenz zwischen verschiedenen gleichartigen und parallel ausgeführten Service-Instanzen, wenn beispielsweise eine Instanz ausfällt oder die Kommunikation gestört ist. Jedoch wird eine schnelle, dynamische Bereitstellung benötigter Dienste erleichtert, dafür werden häufig *Virtualisierung* (s. Abschnitt 3.7.1 auf Seite 86) und *Containerisierung* (s. Abschnitt 3.7.1 auf Seite 88) eingesetzt. [Tal21]

Ein spezieller Typ von Service, der später in dieser Arbeit (s. Abschnitt 4.4.1 auf Seite 121) bei der Verwendung von ROS-Komponenten in ROS2 eine Rolle spielt, ist das Technologie-Gateway. Dieses dient dazu, vorhandenen und nicht

SOA-kompatiblen Code nutzbar zu machen, indem dieser gekapselt und nach außen mit einer Service-Schnittstelle versehen wird.

3.5.3 Service-Repository

Das Service-Repository stellt Informationen zur Verfügung, die es ermöglichen, Services zu entdecken und zu nutzen. Das schließt auch über den Contract des Services hinausgehende Informationen ein, wie zum Beispiel Kontaktpersonen und verfügbare Service-Levels. Im einfachsten Fall ist dies eine gedruckte Dokumentation. Eine solche ist nicht für dynamische Zuordnungen oder eine automatisierte Nutzung geeignet, womit sie für den Einsatz im Fahrzeug nicht in Frage kommt.

Eine Lösung hierfür ist der GDS[21] von DDS und damit auch ROS2, bei dem die Informationen auf alle Kommunikationsteilnehmer verteilt werden und Services damit dynamisch an- und abgemeldet werden können, ohne dass ein zentrales Service-Repository benötigt wird (s. Anhang A.2.2 auf Seite 196).

Ähnlich geht AUTOSAR Adaptive vor, wo es als eine sogenannte *Service Registry* Teil des Communication Managements (s. in Abschnitt 3.6.2 auf Seite 80) ist, das auf jedem Steuergerät im Fahrzeug vorhanden ist. In der feinsten Granularität teilen sich alle Publisher und Subscriber auf einer ECU ein gemeinsames, lokales Service-Repository, je nach Konfiguration kann jedoch auch ein Service-Repository mehrere ECUs bedienen. Anders ist das bei ROS2 gelöst, wo jeder Publisher und Subscriber die Informationen dezentral lokal verwaltet.

Wünschenswert, beziehungsweise im Fahrzeug aus Sicherheitsgründen erforderlich, ist außerdem eine Definition von Zugriffsrechten im Service-Repository. Informationen über Performanz, wie mittlere Reaktionszeiten und Durchsatzlimitierungen sowie Skalierbarkeit des Services sind als Teil eines allgemeinen SLA[22] definiert.

[21] Global Data Space
[22] Service Level Agreement

3.5.4 Service-Bus

Die Teilnehmer der SOA, Services und Application Frontends werden durch den Service-Bus miteinander verbunden. Dabei handelt es sich nicht um einen einzelnen physikalischen Bus, sondern um eine Verbindung mehrerer heterogener Medien. Dies ist notwendig, da auch eine Heterogenität der Clients berücksichtigt werden muss. Bei einer (re-)konfigurierbaren Architektur sind das zum Beispiel neuartige Vernetzungstechnologien, die im Rahmen von Upgrades integriert werden, zum Beispiel neue und schnellere Netzwerkprotokolle. Die Verbindung wird dabei durch eine oder mehrere Middlewares hergestellt (s. Abschnitt 3.2.1 auf Seite 51).

Ein Beispiel für die Funktionsweise der Kommunikation ist in Abbildung 3.17 dargestellt. Eine Anwendung, die einen Service bereitstellt, registriert diesen im *Service-Repository*. Eine Anwendung, die den Service nutzen will, kann ihn dort entdecken und um ihn zu nutzen direkt mit der den Service anbietenden Anwendung kommunizieren.

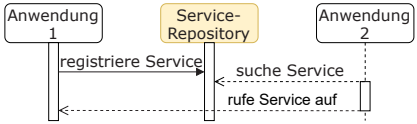


Abbildung 3.17: Beispiel für das Entdecken eines Services

3.6 Beispiele für serviceorientierte Architekturen (SOA)

3.6.1 ROS2

Die erste und immer noch sehr weit verbreitete Version des Robot Operating Systems, ROS1 oder einfach nur ROS (s. Abschnitt 3.3.3 auf Seite 56), wurde bei Willow Garage für den Forschungsroboter PR2 entwickelt. Bei diesem handelt es sich noch um eine Entwicklungsplattform für einen mobilen, durch zwei Greifarme leicht humanoid wirkenden Roboter und kein automatisiertes

Fahrzeug. Im Gegensatz zum Einsatz in einem serienreifen automatisierten Fahrzeug mit begrenzten Ressourcen (Bauraum, Kühlung, Stromversorgung, Kosten pro Stück), konnte hier daher auf sehr leistungsfähige Hardware aus dem Workstationbereich und eine zuverlässige und performante Netzwerkinfrastruktur zurückgegriffen werden, da solche Ressourcen weniger beschränkt als in Serienfahrzeugen sind. Dies rührt zum einen daher, dass es sich um die Prototypenentwicklung mit sehr geringen Stückzahlen handelt, aber auch daher, dass die Rechnereinheiten außerhalb des Roboters platziert werden können und dieser die unmittelbare Nähe des Labors üblicherweise nicht verlässt. Ebenso konnten, anders als bei Fahrzeugen, Echtzeitanforderungen vernachlässigt werden. [Bri20]

Diese für den Einsatz im automatisierten Fahren gravierenden Nachteile, keine Echtzeitfähigkeiten, keine Unterstützung nicht-idealer Netzwerke wie Mobilfunk sowie die Ausrichtung auf den Forschungsbereich, werden mit der zweiten Generation, ROS2, adressiert. Dabei fallen die Änderungen so umfangreich aus, dass keine unmittelbare Abwärtskompatibilität gegeben ist. Wie damit umgegangen werden kann, ist Teil der Untersuchung von Usecase 1 „Erweiterung einer bestehenden Architektur“ (s. Abschnitt 4.4 auf Seite 121).

Struktur der ROS2-API [Ope20a]

In Abbildung 3.18 auf der nächsten Seite sind die Ebenen von ROS2 abgebildet (vgl. Abbildung 3.13 auf Seite 57 für Ebenen in ROS). Im Vergleich zu ROS (s. Abschnitt 3.3.3 auf Seite 56) fehlt der ROS-Master, die Kommunikation wird nun dezentral verwaltet. Die ROS-Nodes bieten ihre Services selbstständig an beziehungsweise abonnieren andere. Statt einer eigenen Middleware kommt in ROS2 DDS (s. Anhang A.2.2 auf Seite 196) zum Einsatz.

Im Vergleich zu ROS wurde in ROS2 außerdem die Anzahl der unterstützten Betriebssysteme erhöht: Anstatt nur Linux wird nun jedes POSIX[23]-kompatible Betriebssystem (wie zum Beispiel *MacOS*) unterstützt, außerdem kann ROS2 unter Windows ausgeführt werden.

[23] Portable Operating System Interface

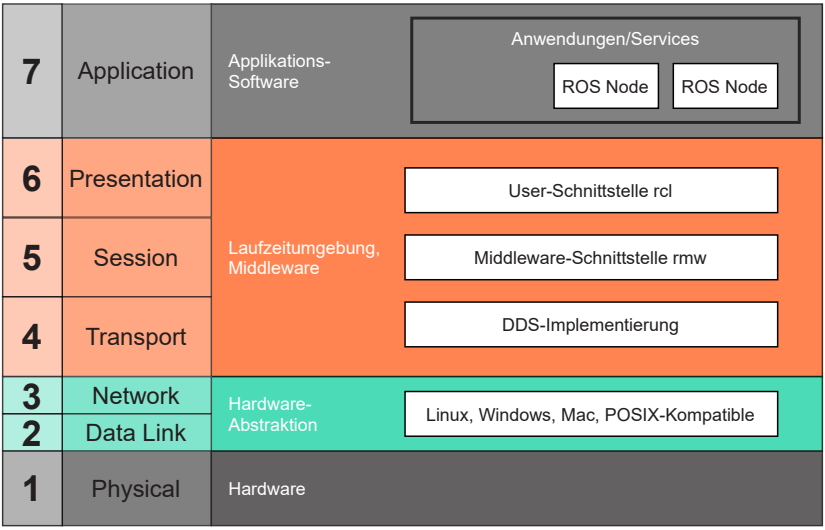


Abbildung 3.18: Ebenen von ROS2 und die darin enthaltenen OSI-Schichten

Wie in Abbildung 3.19 auf Seite 69 detaillierter dargestellt, gibt es zwei wichtige Schnittstellen: die Middleware-Schnittstelle *rmw* und die User-Schnittstelle *rcl*. Ziel dieser Abstraktionsebenen ist, dass Anwender der Plattform, die die eigentliche Funktionalität implementieren, nie direkt mit der konkreten Umsetzung der Basis interagieren müssen und so zum Beispiel ein Austausch der Middleware mit möglichst wenigen Änderungen zu bewältigen ist.

Erstere stellt dabei die Schnittstelle zur zugrundeliegenden Middleware-Implementierung dar. Diese ist bei ROS2 im Gegensatz zu ROS modular austauschbar, zum Beispiel kann entweder DDS (s. Anhang A.2.2 auf Seite 196) oder RTPS[24] (s. Anhang A.2.2 auf Seite 196) genutzt werden, außerdem existieren Umsetzungen verschiedener Hersteller. Damit stellt sie Funktionen zur Verfügung, die dem Entdecken, Abonnieren und Nutzen von Services und dem Austausch von Nachrichten dienen (zum Beispiel die Publish-Subscribe-Funktionalität auf Nachrichtenkanälen).

[24] Real-time Publish-Subscribe Protocol

Letztere baut auf dieser Abstraktionsebene der Middleware-Schnittstelle *rmw* auf und stellt die Schnittstelle zu den Nutzeranwendungen dar. Dazu existieren verschiedene, sprachspezifische APIs[25], die die Schnittstelle in Sprachen wie C++, Python oder Java nutzbar machen. Wichtig ist hierbei, dass, wie im nächsten Abschnitt genauer beschrieben, transparent zwischen den unterschiedlichen Implementierungen von Datentypen in den verschiedenen Programmiersprachen übersetzt wird. Aufgabe der *rcl* ist dabei, von komplexeren zugrundeliegenden ROS-Funktionalitäten zu abstrahieren, wie der Verwaltung von Services, deren Parametern und zum Beispiel dem Logging.

Die links in Abbildung 3.19 auf der nächsten Seite dargestellte DDS-Abstraktions-Schnittstelle ermöglicht es Anwendungen, falls notwendig, auf Parameter der DDS-Implementierung zuzugreifen und dennoch eine bestmögliche Isolation von der konkreten DDS-Implementierung zu ermöglichen. Idealerweise hat ein Austausch der Middleware so keinerlei Einfluss auf die Applikations-Software.

Unterstützung von Nachrichtentypen in ROS2 [Ope20a]

Die konkrete Umsetzung der Datentypen in Nachrichten und damit des Interfaces ist schlussendlich von der zur Implementierung der Applikations-Software verwendeten Programmiersprache abhängig. Um eine Unabhängigkeit der Nachrichtentypdefinition und der Middleware von der Applikations-Software zu erreichen (und umgekehrt), stellt ROS2 Mechanismen zur Übersetzung bereit. Dies umfasst den gesamten Weg von der Definition des Nachrichtenformats, zum Beispiel in Form von .msg-Dateien (s. Erläuterung 3.6.1), bis hin zur sprachspezifischen Umsetzung in Form von Programmcode und Header-Files. Dabei kann diese Übersetzung statisch oder dynamisch erfolgen, jeweils mit eigenen Vor- und Nachteilen.

Erläuterung 3.6.1: .msg-Datei

Messages stellen einen Teil des Kommunikationsinterfaces in ROS2 dar. Basierend auf der Nachrichtentypdefinition in *.msg*-Dateien erzeugen

[25] Application Programming Interfaces

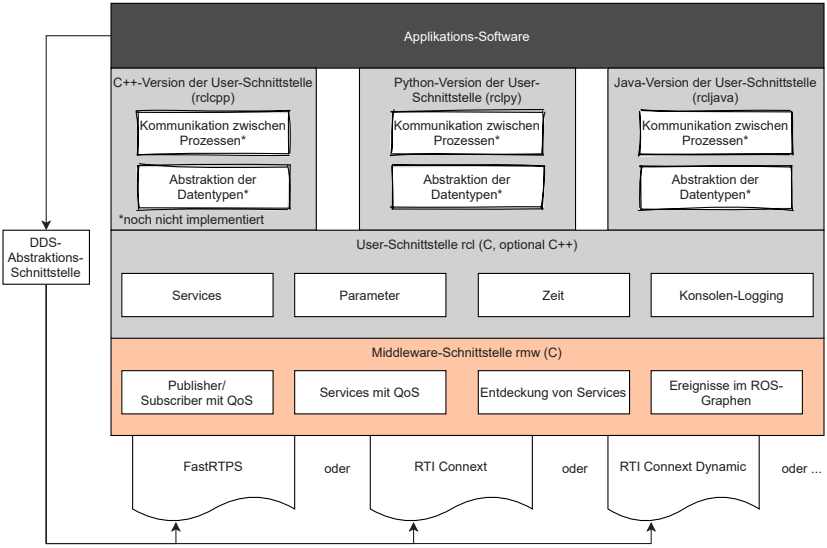


Abbildung 3.19: ROS2-API-Überblick, nach [Ope20a]

verschiedene Tools automatisch den Quelltext in verschiedenen Programmiersprachen, der benötigt wird, um die Nachrichten interpretieren zu können. Eine *.msg*-Datei ist dabei aus sogenannten *Feldern* und *Konstanten* aufgebaut, welche jeweils eine Zeile umfassen. Felder bestehen aus einem Datentyp, einer Bezeichnung und gegebenenfalls einer Angabe eines Standardwertes oder des gültigen Wertebereichs beziehungsweise der erlaubten Länge im Falle von Zeichenketten. Die Datentypen entsprechen den Basisdatentypen gängiger Programmiersprachen wie C++ und Arrays aus diesen, aber auch Strings und dynamische Arrays werden unterstützt. Dabei können so definierte Nachrichtentypdefinitionen auch selbst wieder Datentypen darstellen und somit verschachtelt werden. Ein Beispiel ist in Listing 1 auf der nächsten Seite dargestellt. Dieses enthält ein Feld vom Typen Array aus Integern, das maximal 5 Elemente enthalten darf, die bereits vordefiniert sind.

Ähnlich sind Konstanten aufgebaut: sie entsprechen Feldern mit einem Standardwert, der zur Laufzeit nicht geändert werden kann.

```
1 int32[<=5] werte [-200, -100, 0, 100, 200]
```

Listing 1: Beispiel für eine .msg-Datei

Die statische Übersetzung, in Abbildung 3.20 auf der nächsten Seite dargestellt, hat den Vorteil, dass Overhead reduziert wird, solange ausschließlich zur Implementierungszeit definierte Datentypen zum Einsatz kommen. Hierfür werden Funktionen entsprechend speziell für diese Datentypen implementiert, können daher auch ohne Änderungen nicht mit geänderten Nachrichtenformaten umgehen.

Das wird problematisch, sobald neue Komponenten hinzugefügt werden, da die bereits auf dem Fahrzeug vorhandenen Nachrichtentypen unterstützt werden müssen und die neuen Komponenten ohne Änderungen beziehungsweise Vorkonfiguration am bestehenden System (wie in Anforderung AA3 gefordert) auf diese beschränkt wären. Dies ist auch ein Nachteil der Umsetzung des RACE-Projekts, s. Anhang A.2.3 auf Seite 199.

Bei der statischen Übersetzung werden die im ROS2-Format definierten Nachrichtendefinitionen im .msg-Format direkt durch sprachspezifische Codegeneratoren in die Zielsprachen, wie zum Beispiel C++ oder Python, übersetzt (s. rechte Seite in Abbildung 3.20 auf der nächsten Seite). Dabei ist der Codegenerator selbst von der eingesetzten Middleware unabhängig.

Auf der linken Seite der Abbildung ist dargestellt, wie die .msg-Dateien genutzt werden, um spezifischen Code zur Unterstützung der Datentypen zu generieren. Dieser enthält Funktionen, die für einen bestimmten Datentyp spezifisch sind und für diesen angepasste Aufgaben ausführen. Zum Beispiel kann das Versenden einer Nachricht mit diesem Datentyp bestimmte Umwandlungen erfordern. Ebenso kann es passieren, dass für die verwendete Middleware-Implementierung spezifische Anpassungen durchgeführt werden müssen, da sich deren Schnittstellen von denen anderer Implementierungen unterscheiden.

Dazu werden zuerst die ROS-spezifischen Nachrichtentypdefinition im .msg-Format in das DDS-Format .idl übersetzt und anschließend Code für die Unterstützung der verwendeten Datentypen generiert, der speziell auf verwendete Middleware und Programmiersprachen angepasst ist.

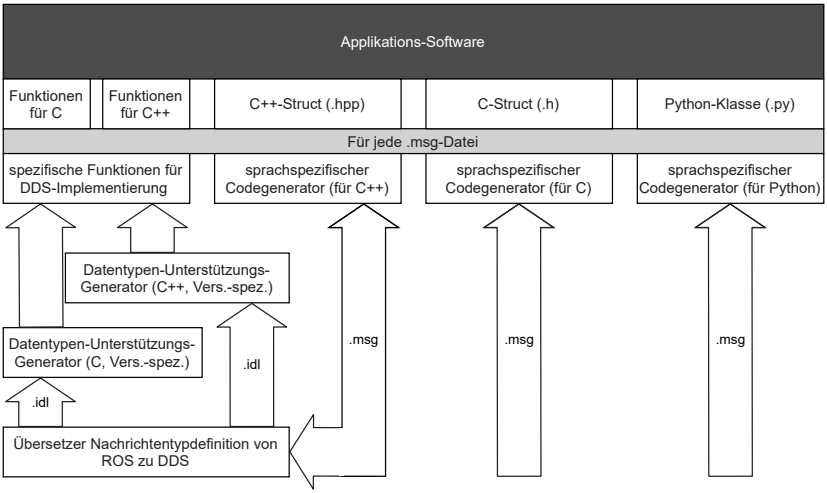


Abbildung 3.20: Statische Datentypenunterstützung in ROS2, nach [Ope20a]

Bei der dynamischen Übersetzung (s. Abbildung 3.21 auf der nächsten Seite) ist hingegen keine frühzeitige Festlegung auf bestimmte Datentypen notwendig, Funktionen sind so implementiert, dass sie mit unterschiedlichen Datentypen umgehen können, womit auch neue Komponenten ohne Änderungen am bestehenden System mitsamt neuer Nachrichtendatentypen hinzugefügt werden können. Dazu erhalten diese Funktionen zu jedem Nachrichtentyp Metadaten, welche Informationen enthalten, wie dieser zu interpretieren ist. Vorteil dieses Vorgehens ist die Flexibilität, Middleware-Unabhängigkeit des erzeugten Codes und generell verringerte Menge erzeugten Codes, Nachteil jedoch der dadurch entstehende Overhead zur Laufzeit und die erhöhte Komplexität, womit eine höhere Fehleranfälligkeit einher geht.

Die rechte Seite der Abbildung 3.21 auf der nächsten Seite ist dabei identisch zu Abbildung 3.20. Hauptunterschied ist das Vorgehen, wie die Unterstützung der Datentypen umgesetzt wird.

Bei der dynamischen Übersetzung entfällt dabei die Umwandlung der ROS-spezifischen .msg-Dateien in .idl-Dateien im DDS-kompatiblen Format. Dadurch wird der Einsatz von DDS-Implementierungen notwendig, die eine solche dynamische Datentypenunterstützung enthalten. Gleichzeitig wird dadurch der Generator für die Datentypenunterstützung Middleware-agnostisch.

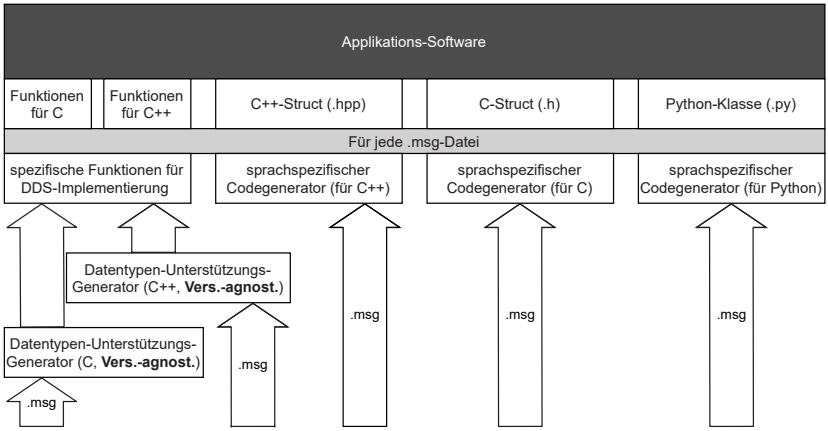


Abbildung 3.21: Dynamische Datentypenunterstützung in ROS2, nach [Ope20a]

Quality of Service (QoS) in ROS2

Um die Kommunikation zu optimieren, verfügt ROS2 über eine Vielzahl an QoS-Policies. Anwendungsbeispiele sind der Einsatz in verlustbehafteten Netzwerken wie Mobilfunk, wo nicht davon ausgegangen werden kann, dass jedes Datenpaket sein Ziel erreicht, aber auch in Echtzeitsystemen, wo die Übertragung innerhalb spezifizierter Deadlines erfolgen muss. [ros20c] Im Kontext dieser Dissertation tritt ein solcher Fall zum Beispiel bei der Verlagerung eines Services in das entfernte Backend auf (s. Use-Case 4 *Auslagern von Services ins Backend* auf Seite 123).

Dabei werden in ROS2 mehrere QoS-Policies zu einem QoS-Profil zusammengefasst. Einige solcher Profile werden vordefiniert mitgeliefert, zum Beispiel für häufige Use-Cases wie die Übertragung von Sensordaten (hierbei wird es als wichtiger betrachtet, dass die Daten *so schnell wie möglich* ankommen

anstatt dass *alle* ihr Ziel erreichen). Profile können sowohl auf die Anbieter als auch die Konsumenten von Services und Daten angewandt werden. Zu berücksichtigen ist hierbei, dass nicht alle Profile untereinander kompatibel sind. Während sich in ROS alle Subscriber auf ein Topic „subscriben" konnten, wenn nur der Datentyp identisch war, ist dies in ROS2 nicht mehr unbedingt der Fall. [ros20c]

In Tabelle A.5 auf Seite 205 sind die unterstützten Policies des Basis-QoS-Profils in ROS2 aufgeführt und beschrieben. Dabei gibt es für jede Policy einen Defaultwert, der, wenn es sich um keine Zeitdauer handelt, von der darunterliegenden Middleware vorgegeben wird, andernfalls üblicherweise einer unendlichen Dauer entspricht.

Aus diesen Policies resultieren einige, hauptsächlich für Entwickler interessante, Ereignisse, die von ROS2 zu Verfügung gestellt werden. Dazu gehören zum Beispiel verpasste Deadlines, *verstorbene* (ein Publisher wird als tot betrachtet, sobald eine definierte Zeit lang keine Nachricht von ihm empfangen wurde) Publisher oder inkompatible QoS-Policies. [ros20c]

Managed Nodes in ROS2

Ähnlich wie AUTOSAR Adaptive mit dem *Execution Management* (s. Abschnitt 3.6.2 auf Seite 79) unterstützt ROS2 bei der Verwaltung der Lebenszyklen von Nodes. Hierzu dient das Konzept der *Managed Nodes* [BF20].

Diese erlauben es dem System bei dem Start des Fahrzeuges alle Komponenten, wie ROS2-Nodes und die dazugehörigen Services, in einer sinnvollen Reihenfolge zu starten und damit sicherzustellen, dass alle für die Funktionalität notwendigen Komponenten und Abhängigkeiten korrekt gestartet wurden, bevor diese vollständig ausgeführt wird. Dabei besitzt jeder *Managed Node* eine genau spezifierte und bei allen Nodes identische Schnittstelle, mit der er verwaltet werden kann. Diese Verwaltung umfasst zum Beispiel das Neustarten oder Ersetzen des Nodes zur Laufzeit des Systems.

3.6.2 AUTOSAR Adaptive [AUT19b]

Der Standard AUTOSAR Adaptive Platform [AUT19b] definiert die Architektur des Laufzeitsystems, woraus die Plattform besteht und welche Funktionalitäten und Schnittstellen sie zur Verfügung stellt. Dabei werden auch maschinenlesbare Modelle definiert, die zur Entwicklung eines solchen Systems genutzt werden.

Von AUTOSAR Classic zu AUTOSAR Adaptive

Die AUTOSAR Classic Platform (s. Abschnitt 3.3.2 auf Seite 54) hat hauptsächlich klassische ECUs im Fokus, die relativ einfache und über den Lebenszyklus eines Fahrzeuges konstante Funktionalitäten besitzen und dabei hauptsächlich vormals elektromechanische Systeme ersetzen, AUTOSAR Adaptive adressiert hingegen sich durch neue Fahrzeugfunktionen wie automatisiertes Fahren ergebende Systeme mit neuen Anforderungen an OTA-Aktualisierbarkeit der Software und Security, aber auch an die Performanz.

Während AUTOSAR Classic mit Hinblick auf kleinere Mikrocontroller entwickelt wurde, die üblicherweise einen einzigen CPU-Kern besitzen, unterstützt AUTOSAR Adaptive die immer weiter voranschreitende Parallelisierung durch Erhöhung der Kernanzahl durch Multi- bis Manycore-Prozessoren mit mehreren Dutzend und mehr CPU-Kernen. Auch dedizierte Beschleuniger, FPGAs oder die Berechnung auf GPUs (*heterogenous computing*) waren in AUTOSAR Classic noch nicht verfügbar und daher nicht berücksichtigt. Ein weiterer Technologietreiber bei der Entwicklung von AUTOSAR Adaptive war der Einsatz von Ethernet im Fahrzeug, wodurch sich durch die wesentlich höhere Bandbreite neue Kommunikationsmöglichkeiten ergeben, die mit AUTOSAR Classic nicht oder nur eingeschränkt umsetzbar wären, trotz dessen grundsätzlicher Unterstützung von Ethernet, ein Beispiel ist der Aufbau als SOA.

AUTOSAR Adaptive unterstützt das *inkrementelle Deployment* von Anwendungen. Dies bedeutet, dass nur veränderte Anwendungen neu ausgerollt und Ressourcen und Kommunikation anschließend dynamisch zur Laufzeit zugewiesen werden, im Gegensatz zu AUTOSAR Classic, wo solche Dinge statisch konfiguriert sind und im Ganzen verteilt werden. Jedoch lässt sich auch diese Dynamik aus Safety-Gründen im sogenannten *Execution Manifest*

(s. Abschnitt 3.6.2) begrenzen, sodass zum Beispiel die Entdeckung von Services vordefiniert werden kann oder Speicher und CPU-Leistung nur in der Startphase dynamisch zugewiesen werden. Dabei werden auszurollende Softwarekomponenten und Abhängigkeiten automatisch erkannt und aufgelöst.

Bei ROS2 ist für ROS-Nodes ein ähnlich dynamisches Verteilen vorgesehen, jedoch nicht für die Zuweisung von Hardwareressourcen. In dieser Dissertation kommt zu diesem Zweck daher ein eigener Orchestrator (s. Abschnitt 4.2.2 auf Seite 104) zum Einsatz, der dies ermöglicht.

Dabei ist in AUTOSAR Adaptive durchaus vorgesehen, dass eine gemischte Architektur aus AUTOSAR Classic-Komponenten (für die klassischen Funktionen) und AUTOSAR Adaptive-Komponenten (für neuartige Funktionen wie automatisiertes Fahren) zum Einsatz kommt (s. Abbildung 3.22 auf der nächsten Seite: AUTOSAR Classic, in grün, wird für vorhandene Fahrfunktionen und die Sensorik genutzt, während AUTOSAR Adaptive, in hellblau, für die automatisierte Fahrfunktion zum Einsatz kommt.). Dies ergibt sich auch aus der Tatsache, dass speziell in AUTOSAR Classic definierte Features wie der direkte Zugriff auf elektrische Signale und Automotive Bussysteme wie CAN oder LIN[26] nicht im Fokus der Standardisierung von AUTOSAR Adaptive sind, auch wenn sie integriert werden könnten. Dazu werden bestimmte Protokolle wie SOME/IP[27] (s. Anhang A.2.2 auf Seite 197) sowohl von AUTOSAR Adaptive als auch AUTOSAR Classic unterstützt.

In Tabelle A.6 auf Seite 206 sind einige dieser Unterschiede und weitere Eigenschaften gegenübergestellt.

Entwicklungsprozess und Manifeste in AUTOSAR Adaptive

Die Standardisierung des Entwicklungsvorgehens ist auch Bestandteil von AUTOSAR Adaptive. Dabei wird auch die Beschreibung von Artefakten wie Services, Anwendungen, Maschinen und deren Konfiguration sowie die ihrer Zusammenarbeit behandelt.

[26] Local Interconnect Network
[27] Scalable service-Oriented MiddlewarE over IP

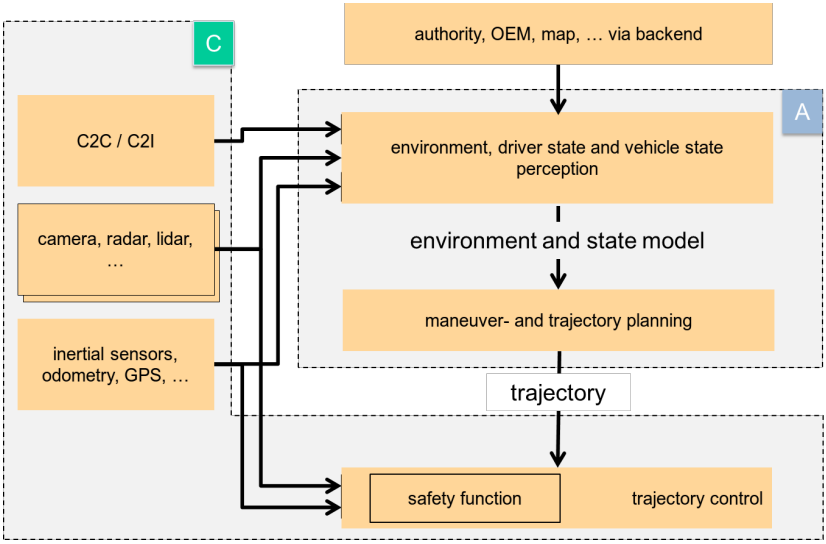


Abbildung 3.22: Beispiel für den gemeinsamen Einsatz von AUTOSAR Classic und AUTOSAR Adaptive in einem Fahrzeug [AUT19b]

In Abbildung 3.23 auf der nächsten Seite ist das Vorgehen dabei schematisch dargestellt, ausführbare Anwendungen sind orange gefärbt, Beschreibungen und *Manifeste* hellorange, sowie Arbeitsschritte des Prozesses in grau. Sogenannte *Manifeste* stellen einen Teil der Modellbeschreibung eines einzelnen Produkts dar und enthalten die entsprechende Konfiguration.

Im *Application Design* werden alle entwurfsspezifischen Details für die Erstellung von Applikationssoftware für AUTOSAR Adaptive spezifiziert. Dabei wird es zur Laufzeit nicht mehr benötigt und muss daher auch nicht auf dem Fahrzeug vorhanden sein. Außerdem dient es als Basis für die Verteilung der Applikationssoftware in *Execution Manifest* und *Service Instance Manifest*. Dazu gehört die Spezifikation der Datentypen und Schnittstellen zur serviceorientierten Kommunikation, aber auch Schnittstellen zu den anderen Komponenten von AUTOSAR Adaptive (s. Abschnitt 3.6.2 auf Seite 78).

Das *Execution Manifest* enthält Informationen über das Ausrollen von Applikationen, die auf dem Fahrzeug laufen. Es wird zusammen mit dem ausführbaren

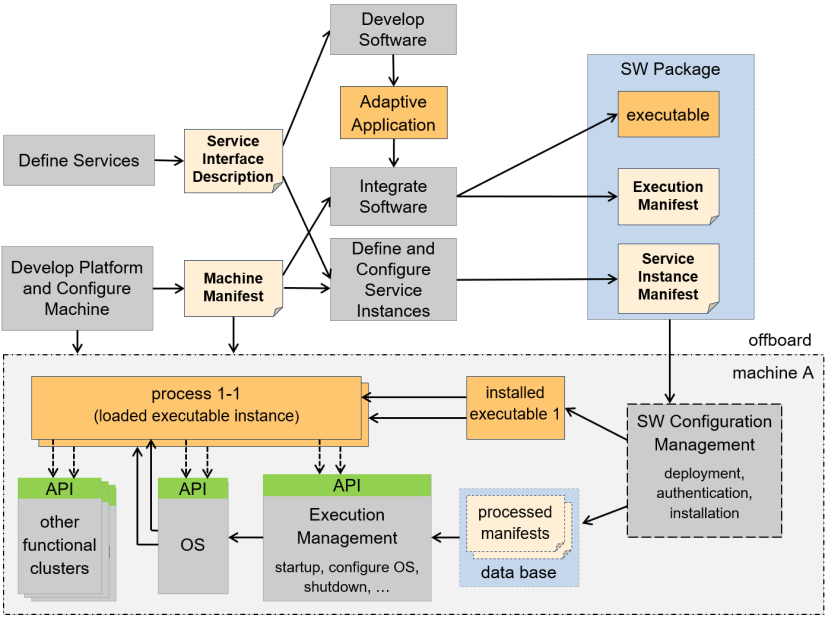


Abbildung 3.23: Entwicklungsprozess in AUTOSAR Adaptive [AUT19b]

Binärcode gebündelt und unterstützt damit dessen Integration auf dem Fahrzeug (entsprechend Anforderung AP5). Dazu wird definiert, wie oft und auf welchen ECUs die Applikation instanziiert werden soll, wie sie gestartet werden soll und welche Ressourcen sie zugewiesen bekommt.

Die serviceorientierte Kommunikation und die ihr zugrundeliegenden Kommunikationsprotokolle werden vom *Service Instance Manifest* spezifiziert. Dabei wird es mit dem ausführbaren Code gebündelt, der die Implementierung der serviceorientierten Kommunikation enthält. Diese umfasst auch die Konfiguration von Security-Aspekten und Logging.

Im *Machine Manifest* ist ausschließlich die Konfiguration der Hardwarebasis unter AUTOSAR Adaptive beschrieben. Damit dient dieses der Einrichtung und Anpassung von AUTOSAR Adaptive selbst. Konfiguriert werden dabei zum Beispiel die Netzwerkanbindung und welche Hardwareressourcen in Form von CPU-Kernen oder RAM verfügbar sind.

Logische Struktur von AUTOSAR Adaptive

Basis der logischen Struktur von AUTOSAR Adaptive (s. Abbildung 3.24 auf der nächsten Seite) kann direkt die Hardware der ECU sein, aber auch eine virtuelle Maschine oder ein Container wie zum Beispiel ein Docker-Container (s. Abschnitt 3.7.1 auf Seite 89).

Auf das auf dieser Basis laufende POSIX-kompatible (s. Definition 3.6.1) Betriebssystem, wie zum Beispiel Linux, setzt die ARA[28] über das *Operating System Interface* auf. Dabei kann auf die objektorientierte C++-Standardlibrary oder PSE51, eine Untermenge des POSIX-Standards zurückgegriffen werden. Letztere ist dabei empfohlen, da auf Freiheit von Wechselwirkungen zwischen den laufenden Anwendungen optimiert.

Definition 3.6.1: POSIX

In der IEEE 1003.1 „IEEE Standard for Information Technology – Portable Operating System Interface (POSIX) Base Specifications“ [IEE17] definiert, handelt es sich bei POSIX um eine Sammlung von Standards, die die Kompatibilität zwischen verschiedenen Betriebssystemen gewährleisten sollen. Dazu werden eine Standardschnittstelle zum Betriebssystem und eine passende Umgebung wie zum Beispiel ein Befehlsinterpreter sowie einige Dienstprogramme definiert. Ziel ist, dass Programme auf Quellcode-Ebene zwischen verschiedenen Betriebssystemen portabel sind, das heißt, dass sie nach einer Kompilierung auf dem Zielbetriebssystem ohne vorherige Anpassungen auf diesem ausführbar sind.
Beispiele sind: zertifiziert: macOS, QNX, größtenteils kompatibel: Android, Linux.

Über diesem *Operating System Interface* befinden sich die FCs[29], die Anwendungsschnittstellen zu den AUTOSAR Adaptive-Services zur Verfügung stellen. Dabei wird zwischen *Adaptive Platform Foundation* und *Adaptive Platform Services* unterschieden. Erstere umfasst fundamentale Funktionalitäten von AUTOSAR Adaptive, während letztere Standard-Services von AUTOSAR Ad-

[28] AUTOSAR Runtime for Adaptive Applications
[29] Functional Cluster

aptive darstellen. Die für diese Dissertation relevanten Komponenten werden im folgenden Abschnitt genauer beschrieben.

Auf der obersten Ebene laufen schließlich die AA[30] als sogenannte *User Applications*. Auch AA können Services zur Verfügung stellen, diese sind in der Abbildung als *non-platform (non-PF) Services* dargestellt.

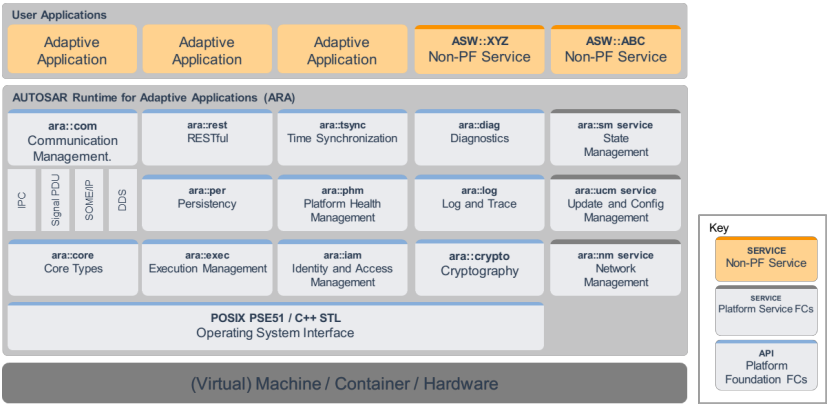


Abbildung 3.24: Logische Struktur von AUTOSAR Adaptive [AUT19b]

Functional Clusters der AUTOSAR Adaptive Platform

Execution Management Das *Execution Management* ist für alle Tätigkeiten im Rahmen der Ausführung des Systems zuständig. Dies umfasst die Initialisierung der Plattform sowie Starten und Beenden der Anwendungen (bei ROS2 existieren dazu sogenante *Managed Nodes*, s. Abschnitt 3.6.1 auf Seite 73). Dazu gehört auch die Authentifizierung von Anwendungen, sodass nur autorisierte Anwendungen ausgeführt werden können. Dabei ist wichtig, dass eine deterministische Ausführung gewährleistet wird, sowohl im Sinne von Zeit- als auch Datendeterminiertheit. Erstere ist erforderlich, um Echtzeitanforderungen einhalten zu können, und sagt aus, dass Ergebnisse immer in der vorgesehenen Zeitspanne verfügbar sind. Letztere bedeutet, dass aus

[30] Adaptive Applications

identischen Eingangsdaten und im selben internen Zustand immer dasselbe Ergebnis erzeugt werden muss.

Da davon ausgegangen wird, dass sich die Einhaltung von zeitlichen Deadlines durch Bereitstellung ausreichender Ressourcen bewerkstelligen lässt, konzentriert sich das *Execution Management* auf die Datendeterminiertheit und stellt dazu eigene Programmierschnittstellen zur Verfügung.

Gleichzeitig muss aber auch beachtet werden, dass sich fehlerhaft verhaltende Anwendungen andere Anwendungen in Mitleidenschaft ziehen können, daher können Anwendungsprozesse sogenannten *RessourceGroups* mit definierten und beschränkten Ressourcen zugewiesen werden, die vom *Execution Management* überwacht werden. Entsprechend werden auch sich nicht nach Spezifikation verhaltende Prozesse von Anwendungen durch das *Platform Health Management* (s. Seite 82) erkannt und vom *Execution Management* beendet und gegebenenfalls neu gestartet. Dieses Verhalten ist im *Execution Manifest* (s. Seite 76) festgelegt.

Communication Management Von besonderer Wichtigkeit für die Service-Orientierung ist das *Communication Management*, das für die serviceorientierte Kommunikation zwischen Anwendungen in einer verteilten und eingebetteten Echtzeitumgebung verantwortlich ist. Das heißt, dass nicht nur die Kommunikation innerhalb oder zwischen Anwendungen auf derselben ECU möglich ist, sondern auch zwischen Anwendungen auf verschiedenen ECUs. Ein Beispiel für die Funktionsweise der Kommunikation ist in Abbildung 3.17 auf Seite 65 dargestellt.

In AUTOSAR Adaptive können diese Kommunikationspfade entweder zur Entwurfszeit, beim Start oder zur Laufzeit des Systems festgelegt werden. Damit kann deren Konfiguration entweder statisch oder dynamisch stattfinden, von der voll statischen Kommunikation, bei der alle Service-Nutzer alle Service-Anbieter kennen, bis zur vollständig dynamischen Variante, in der zu Beginn keine Pfade bekannt sind und Services zur Laufzeit von den Nutzern entdeckt werden. Bei AUTOSAR Adaptive ist dafür die, zum *Communication Management* gehörende, sogenannte *Service Registry* als Service-Repository (s. Abschnitt 3.5.3 auf Seite 64) verantwortlich.

Momentan (AUTOSAR Adaptive R19-11) werden von AUTOSAR Adaptive SOME/IP (s. Anhang A.2.2 auf Seite 197), DDS (s. Anhang A.2.2 auf Seite 196), IPC[31] und Signal PDU (als signalbasierte Kommunikationsform) unterstützt. Der Service-Contract (s. in Abschnitt 3.5.2 auf Seite 62) ist in AUTOSAR Adaptive um eine Versionierung erweitert. Damit ist sichergestellt, dass zwischen den während der Entwicklung entstehenden Versionen von Funktionalität und Schnittstellen der Services eindeutig unterschiedenen werden kann. Möglich ist so auch die Bereitstellung von mehreren Versionen zur selben Zeit, um eine Kompatibilität mit alten Legacy-Komponenten und aktuellen, neuen sicher zu stellen.

Update and Config Management AUTOSAR Adaptive soll Aktualisierungen der Software außerhalb von Werkstätten, über Mobilfunk und Ähnliches erlauben (OTA). Zu diesem Zweck gibt es mit dem *Update and Configuration Manager (UCM)* einen Service, der Anfragen für Softwareupdates bedient. Zu seinen Aufgaben gehört das Installieren, Entfernen und Protokollieren von Software im Fahrzeug. Damit ist er ähnlich den in der IT-Welt verbreiteten Linux-Paketmanagern. Dabei wird sichergestellt, dass nur signierte und autorisierte Sofwarepakete installiert werden können. Außerdem wird die Konfiguration der Pakete bei der Installation ensprechend dem Fahrzeugmodell und seiner konkreten Ausstattung durchgeführt. Aufgabe des OEM[32] ist es, zu garantieren, dass während des Updates beziehungsweise der Installation von Software ein sicherer Zustand eingehalten wird. Dies schließt ein, dass im Anschluss an das Update überprüft wird, ob alle relevanten Prozesse weiterhin funktionsfähig sind und gestartet werden.

Bei der (re-)konfigurierbaren Fahrzeugarchitektur wird eine ähnliche Funktionalität von der CI/CD[33]-Pipeline mit anschließendem Deployment durch den Orchestrator ins Fahrzeug umgesetzt (s. Abschnitt 4.2.8 auf Seite 117).

Identity and Access Management Das *Identity and Access Management (IAM)* dient der Security im Fahrzeug. Dazu gehört, dass beschränkte Zugriffsrechte

[31] Inter-Process Communication
[32] Original Equipment Manufacturer
[33] Continuous Integration, Continuous Delivery/Deployment

vergeben werden können und verhindert werden muss, dass Angreifer aus diesen Einschränkungen ausbrechen können (Privilege Escalation). Hierzu wird eine Identifikation der laufenden Anwendungen über Signaturen durchgeführt, damit sichergestellt werden kann, dass nur berechtige Zugriffe möglich sind. Hiermit werden die Securityanforderungen auf Seite 10 AS3 und vor allem AS4 erfüllt.

Cryptography Die Kryptographie übernimmt als weiteren Aspekt der Security das Erstellen und Verwalten sicherer Schlüssel und weiterer kryptografischer Anwendungen. Dies kann dynamisch zur Laufzeit erfolgen und unterstützt die Verwendung spezieller *Hardware Security Module (HSM)*. Die erstellten beziehungsweise gespeicherten Schlüssel dienen dazu, Daten sicher verschlüsselt übertragen (Anforderung AS3) und wieder dekodieren zu können. Ziel ist es, sicherheitskritische Funktionen in einer separaten Komponente zu kapseln.

Bei der (re-)konfigurierbaren Fahrzeugarchitektur erfüllen Orchestrator und Zertifizierungsstelle (beide s. Abschnitt 4.2.2 auf Seite 104) diese Aufgaben.

Platform Health Management Das *Platform Health Management* dient dazu, die Ausführung der Softwarekomponenten zu überwachen, dies umfasst beispielsweise Frequenz der Ausführung und Einhalten von Deadlines.

Zusätzlich können aber auch Informationen über den Gesundheitszustand der Hardware wie zum Beispiel Temperaturen oder Betriebsspannungen in das *Platform Health Management* eingespeist und von diesem überwacht werden.

Falls in einer der überwachten Komponenten ein Fehler auftritt, kann eine ensprechende Recovery-Aktion ausgeführt werden, um diesen zu beheben oder die Auswirkungen zu minimieren, entsprechend der Safetyanforderungen auf Seite 10 AS1 und AS2.

Das Konzept der sogenannten *Managed Nodes* wird bei der (re-)konfigurierbaren Fahrzeugarchitektur in einer erweiterten Form dazu genutzt, dies umzusetzen (s. Abschnitt 4.2.7 auf Seite 112).

3.6.3 Vergleich der serviceorientierten Architekturen

Zu vergleichende Eigenschaften

Aus den Herausforderungen (s. Abschnitt 1.2 auf Seite 1) und den daraus folgenden Anforderungen (s. Abschnitt 1.6 auf Seite 8) an die (re-)konfigurierbare Fahrzeugarchitektur lassen sich Eigenschaften ableiten, die eine SOA erfüllen muss, um für eine Umsetzung in Frage zu kommen.

Anpassbarkeit Da die (re-)konfigurierbare Fahrzeugarchitektur noch nicht Stand der Technik ist, ist eine Anpassbarkeit der verwendeten SOA-Basis essentiell. Dies umfasst besonders die Anforderungen an die Adaptierbarkeit und dynamische Konfiguration (AA1, AA2, AA3, AA4), aber auch die Portabilität (AP2 und AP4). Dabei muss auch auf geeignete Lizenzen geachtet werden, die entsprechende Änderungen erlauben.

Betriebssystembasis Um von der verwendeten (SW/HW)-Plattform unabhängig zu sein (AP1 und AP2), ist eine möglichst breite unterstützte Betriebssystemsbasis wünschenswert.

Service-Discovery Die Entdeckung von Services ist ein grundlegender Bestandteil des Verschiebens von Services (AP3) und der Rekonfiguration und Aktualisierung von HW und SW zur Laufzeit (AP4). Zusätzlich spielt sie bei der Adaptierbarkeit und dynamischen Konfiguration (AA2, AA3 und AA4) eine Rolle.

Deployment Das Starten und Verteilen von Services und Softwarekomponenten ist notwendig, um diese zur Laufzeit verschieben (AP3) und aktualisieren (AP4) zu können.

HW-Konfiguration Die Definition und Zuweisung von Ressourcen ist für Umsetzung von Safety-Ansätzen wie Redundanz (AS1) oder beim Verschieben von Services (AP3) relevant.

Security Security-Features dienen der Umsetzung der Anforderungen an die Security (AS3 und AS4).

Vergleich der SOA bezüglich dieser Eigenschaften

Ein Vergleich soll zeigen (s. auch Tabelle A.7 auf Seite 207), welche der Lösungen sich am besten eignet, die in dieser Dissertation gestellten Herausforderungen (s. Abschnitt 1.2 auf Seite 1) und Anforderungen (s. Abschnitt 1.6 auf Seite 8) umzusetzen. Einige Eigenschaften werden dabei auch von der eingesetzten Middleware (s. Anhang A.2.2 auf Seite 196) definiert, dabei können zum Beispiel sowohl ROS2 als auch AUTOSAR Adaptive auf DDS setzen. Daher werden die in diesem Bereich meistverbreiteten Middlewares getrennt in Anhang A.2.2 auf Seite 197 verglichen.

Besonders wichtig ist die Flexibilität und Anpassbarkeit auf neue Konzepte wie das der (re-)konfigurierbaren Architektur (besonders die Herausforderungen HF2 und HF3), gleichzeitig sollte nicht alles von Grund auf neu entwickelt werden müssen (Forschungsfrage 2 *Umsetzung der Architektur* auf Seite 8). Das außer im Automotive-Bereich auch in Robotik, Luftfahrt und Forschung eingesetzte ROS2 ist hier im Vorteil, da die Hauptkomponenten an sich bereits verfügbar sind, während das ausschließlich für den Automotive-Bereich entwickelte AUTOSAR Adaptive nur einen Standard darstellt, dessen Implementierung entweder in Form eines Stacks hinzugekauft oder selbst durchgeführt werden muss. Dafür ist in AUTOSAR Adaptive auch der Entwicklungsprozess spezifiziert.

Die Umsetzung mit ROS2 ist wesentlich günstiger, da keinerlei Lizenzkosten anfallen, obwohl die eigene Implementierung unter eine beliebige Lizenz gestellt und sogar verkauft werden darf. Bei AUTOSAR Adaptive muss entweder ein Stack eingekauft werden (zum Beispiel Vector Adaptive MICROSAR) oder es ist eine kostenpflichtige AUTOSAR-Partnerschaft notwendig, um eigene auf AUTOSAR Adaptive basierende Produkte veröffentlichen zu dürfen.

Die (statische) Vorkonfigurierbarkeit von AUTOSAR Adaptive ist an vielen Stellen größer als in ROS2, welches auf eine dynamische Konfiguration zur Laufzeit setzt. So ist die Service-Entdeckung und Verwaltung der Kommunikation in ROS2 immer dynamisch und daher nicht vor dem Start des Systems bekannt. Dies umfasst auch Service-eigene Parameter. Bei AUTOSAR Adaptive können Services bereits zur Entwicklungszeit des Systems definiert und bekannt gemacht werden, aber auch zur Startzeit oder dynamisch während der Laufzeit wie bei ROS2. Ebenso können in AUTOSAR Adaptive die dazu verwendeten Kommunikationspfade entweder statisch vordefiniert oder

erst zur Laufzeit bekannt gemacht werden. Ein von AUTOSAR Adaptive außerdem unterstützter Mittelweg ist, dass Applikationen keine dynamische Entdeckung durchführen können, ihnen die angebotenen Services also vorab bekannt sind, die Anbieter der Services jedoch diese Konsumenten nicht im Vorhinein kennen.

Weiterhin sieht AUTOSAR Adaptive eine detaillierte Definition des Systems in sogenannten *Manifesten* vor: Bis hinab zur Anpassung von AUTOSAR Adaptive auf die verwendete Hardwarekonfiguration wird dieses hier beschrieben. In ROS2 findet diese keine Beachtung; diese Aufgabe wird an das zugrundeliegende Betriebssystem abgegeben. Dafür ist ROS2 in der Unterstützung von Betriebssystemen flexibler, außer POSIX-kompatiblen wie Linux und MacOS wird auch Windows unterstützt, AUTOSAR Adaptive ist ausschließlich auf einem POSIX-kompatiblem Betriebssystem ausführbar.

Dabei sieht AUTOSAR Adaptive anders als ROS2 auch die Verwaltung von Ressourcenzuweisungen und -beschränkungen im sogenannten *Execution Manifest* vor. Beide unterstützen jedoch die Definition von Abhängigkeiten und zu startenden Komponenten, AUTOSAR Adaptive ebenfalls im *Execution Manifest*, ROS2 in sogenannten *launch*-Files. Die fehlende Vordefinition und Verteilung von Ressourcen stellt hier jedoch keinen Nachteil dar, da diese in dieser Dissertation von Kubernetes übernommen wird (s. Abschnitt 4.2.5 auf Seite 109).

Während AUTOSAR Adaptive Module zur Authentifizierung und Authorisierung (*Identity and Access Management*) und Verschlüsselung mitbringt, verlässt sich ROS2 ausschließlich auf die Security-Features seiner Middleware DDS, welche manuell aktiviert werden müssen. Beide profitieren von einer gegebenfalls von der Middleware unterstützten Transportverschlüsselung auf Netzwerkprotokollebene.

Bei der (re-)konfigurierbaren Fahrzeugarchitektur erstellt und verwaltet die Zertifizierungsstelle (s. Abschnitt 4.2.2 auf Seite 104) die dazu benötigten Zertifikate.

3.7 Weitere Grundlagen für die (re-)konfigurierbare Fahrzeugarchitektur

3.7.1 Virtualisierung

Für die Umsetzung der (re-)konfigurierbaren Architektur selbst ist die *Virtualisierung* relevant: Sie stellt eine der Möglichkeiten dar, die Portabilitätsanforderungen an die Architektur zu erfüllen (s. Abschnitt 1.6.2 auf Seite 9).

Dies wird durch die Abstraktion von der Steuergeräte-Hardware (AP2) und die Containerisierung der Softwarekomponenten (AP5) erreicht, diese können anschließend vom Orchestrator im Fahrzeug verteilt werden (AP3).

Kommen verschiedene Hardwarearchitekturen zu Einsatz, bleiben die Softwarekomponenten durch eine Emulation dennoch lauffähig.

Virtualisierung

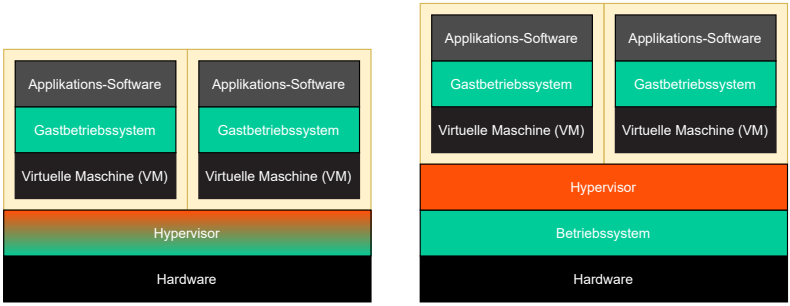
Popek und Goldberg [PG74] definieren eine virtuelle Maschine als „effizientes, isoliertes Duplikat der realen Maschine“.

Dazu stellt die Virtualisierung eine solche Abstraktion von der realen Maschine (Host) her, dass ausgeführte Softwarekomponenten eine ähnliche Umgebung wie auf dieser vorfinden (Duplikat der realen Maschine). Dabei können mehrere solcher virtuellen Maschinen (Gäste) auf einer realen laufen, ohne dass die ausgeführte Software dadurch beeinflusst wird (Isolation).

Die zweite von Popek und Goldberg definierte Charakteristik, Effizienz, bedeutet, dass so viele Instruktionen wie möglich direkt auf dem Host ausgeführt werden, ohne Übersetzung oder Ähnliches. Dies erhöht die zur Verfügung stehende Performanz maßgeblich, allerdings auf Kosten der Plattformunabhängigkeit. Durch den Verzicht einer Übersetzung beziehungsweise Emulation der Befehle müssen Host und Gäste dieselbe Prozessorarchitektur besitzen, jedoch können die Gäste auch ohne Modifikation auf anderen Hosts ausgeführt werden.

Diese Abstraktion erfolgt durch den sogenannten *Hypervisor*, der bei Popek und Goldberg noch als *Virtual Machine Monitor (VMM)* bezeichnet wurde.

Dabei werden zwei Typen von Hypervisoren unterschieden: Typ 1 und Typ 2 (s. Abbildung 3.25).



(a) Hypervisor Typ 1 (b) Hypervisor Typ 2

Abbildung 3.25: Vergleich der beiden Hypervisoren-Typen

Ersterer benötigt kein darunterliegendes Betriebssystem, da er direkt auf der Hardware des Hosts aufsetzt. Dazu muss der Hypervisor aber die entsprechenden Gerätetreiber selbst mitbringen. Verfügbare Produkte sind *XEN*, *Microsoft Hyper-V*, *KVM*[34] (bereits in den Linuxkernel integriert und damit auf jeder Linuxmaschine verfügbar, Mischung aus Typ 1 und Typ 2) und *VMware vSphere*.

Letzterer nutzt die Abstraktion durch ein darunterliegendes Betriebssystem und kann daher auf dessen Hardwareunterstützung in Form von Gerätetreibern aufbauen. Beispiele sind *Oracle VirtualBox* und *VMware Workstation*.

Gemeinsam ist beiden Typen von Hypervisoren, dass in jeder virtuellen Maschine ein komplettes Betriebssystem ausgeführt wird. Dies erleichtert die Trennung der Gäste, was auch Security-Vorteile mit sich bringt, sorgt jedoch für Overhead. Besonders im Embedded-Bereich mit begrenzten Ressourcen wie Rechenleistung oder Arbeitsspeicher ist dieser Overhead unerwünscht und daher zu reduzieren.

[34] Kernel-based Virtual Machine

Containerisierung

Eine Möglichkeit, den Overhead zu reduzieren, stellt die *containerbasierte Virtualisierung* dar. Bei dieser teilen sich alle Gäste den selben Betriebssystemkernel mit dem Host. Entsprechend erfordert dies besonders aufwändige Separierungsmaßnahmen und erhöht die Anfälligkeit für Sicherheitslücken im Kernel beziehungsweise auch deren Auswirkungen.

Ihren Ursprung hat diese Form der Virtualisierung in der *chroot*-Umgebung, die einen Teil des Dateisystems isoliert. Nicht vertrauenswürdige Software kann so nicht auf den Rest des Systems zugreifen. Jedoch ist ein Ausbrechen relativ einfach, selbst in der zugehörigen Dokumentation wird ein möglicher Weg beschrieben [deb20]. Damit ist *chroot* in diesem Kontext unbrauchbar, da die Sicherheitsanforderungen AS3 und AS4 verletzt werden.

Die vier heutzutage noch vertretenen und am weitesten verbreiteten Alternativen sind, zumindest ihrer Popularität bei Suchanfragen zufolge (s. Abbildung 3.26 und 3.27), *LXC*, *LXD*, *Docker* und seit Kurzem *Podman*. Dabei ist die Anzahl der Suchanfragen auf 100 normiert. Mit über 90% Anteil ist dabei Docker am stärksten verbreitet. Die Anteile der restlichen Konkurrenten sind ähnlich groß, jedoch wächst der von Podman am stärksten.

Abbildung 3.26: Containerbasierte Virtualisierung, Anteil an Suchanfragen bei Google im Zeitraum von Mitte August 2015 bis Mitte August 2020. Docker in grün, LXC in blau, LXD in rot und Podman in gelb. [Goo20a]

LXC und LXD LXC (kurz für „Linux Containers“) und LXD werden mittlerweile im *linuxcontainers.org*-Projekt zusammengefasst [Lin20].

Beide bilden eine Schnittstelle zu den im Linux-Kernel enthaltenen Kapselungsfähigkeiten (engl.: *containment features*). Dabei wird unter anderem auch

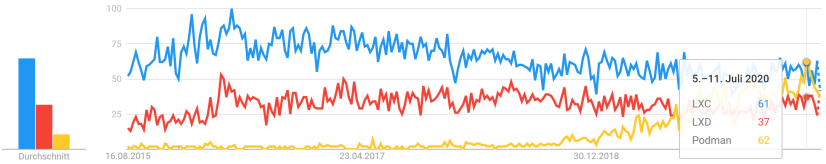


Abbildung 3.27: Containerbasierte Virtualisierung, Anteil an Suchanfragen bei Google im Zeitraum von Mitte August 2015 bis Mitte August 2020. Ohne Docker, LXC in blau, LXD in rot und Podman in gelb. [Goo20b]

chroot in einer moderneren Form genutzt. Ein so gekapselter Gast wird als *Container* bezeichnet.

LXD setzt auf LXC auf und ergänzt dieses um erweiterte Funktionalitäten und verbesserte Performanz sowie Sicherheit.

Während jeder einzelne Container bei LXC in einem eigenen Prozess läuft, existiert bei LXD ein sogenannter *Daemon*, der mehrere Container in einem Prozess ausführt und so den Overhead reduziert. Durch diesen Daemon sind außerdem erweiterte Sicherheitsfunktionen wie Einschränkungen von Rechten und Ressourcen möglich.

Erweiterungen der Nutzbarkeit von LXD gegenüber LXC umfassen Dinge wie Unterstützung mehrerer, miteinander vernetzter Hosts. Zusätzlich wird eine Migration der Container zwischen diesen Hosts zur Laufzeit unterstützt. Im Rahmen dieser Funktionalität ist auch ein Backup von Containern zur Laufzeit möglich, ohne diese stoppen zu müssen. Außerdem wird das Durchreichen von GPUs in LXD direkt unterstützt, bei LXD nur über inoffizielle Workarounds. Dies ist wichtig, um zum Beispiel die Ausführung von neuronalen Netzen für die Objekterkennung zu beschleunigen. Größtes Manko dieser Lösungen ist die eingeschränkte Möglichkeit zur Definition der Softwarekonfiguration eines Containers, mit der dieser ausgehend von einem Basisimage gebaut werden kann.

Docker Im ursprünglich auch auf LXC basierenden und deutlich weiter verbreiteten Docker wird diese Konfiguration über sogenannte *Dockerfiles* unterstützt. Diese dienen der Erzeugung von Docker-Images, basierend auf einem Basis-Image. Diesem können weitere Dateien hinzugefügt und auch

Softwarepakete installiert werden. Ein so erstelltes Image kann anschließend wieder als Basis-Image genutzt werden.

Docker stellt außerdem Möglichkeiten zur Verfügung, GPUs an den Container weiterzuleiten.

Podman Im Jahre 2018 gestartet, ist Podman deutlich jünger als Docker und damit weniger verbreitet. Um einen Umstieg attraktiver zu gestalten, ist es jedoch weitgehend zur Docker-Syntax kompatibel. Dabei können sogar vorhandene Registries und Dockerfiles weitergenutzt werden. Auch das Durchreichen von GPUs ist auf ähnliche Weise möglich. [Pod20]

Das Aufspannen eines Netzwerks von Containern über mehrere Hosts hinweg ist jedoch, anders als bei Docker, nicht möglich.

Aus Anwendersicht ist eines der Hauptfeatures von Podman das Ausführen von Containern als unprivilegisierter Benutzer, also *rootless*. Damit können auch nicht vertrauenswürdige Benutzer eines Hosts Container starten, ohne dass sie Administratorrechte benötigen. Dies erhöht maßgeblich die Sicherheit eines Mehrbenutzersystems und erschwert Manipulationen. Aktuelle Versionen von Docker unterstützen dieses Feature jedoch mittlerweile auch.

Orchestrierung

Der Begriff *Orchestrierung* bedeutet in der Musik die Verteilung der Stimmen eines Stücks auf die einzelnen Instrumente eines Orchesters. Im Kontext dieser Dissertation wird analog dazu die Verteilung von Services und Softwarekomponenten auf die einzelnen Steuergeräte und die Verknüpfung zu einem Gesamtsystem mit diesem Begriff bezeichnet.

Diese Verteilung ermöglicht eine Skalierung der angebotenen Services, die parallele Ausführung mehrerer Instanzen desselben Services auf verschiedenen Steuergeräten erhöht die Performanz. Außerdem wird die Verfügbarkeit verbessert, indem diese parallel laufenden Instanzen bei einem Ausfall den fehlerhaften Service nahtlos zur Laufzeit ersetzen können. Dies kann entsprechend auch für das Ausrollen neuer Versionen genutzt werden. Dazu werden diese parallel gestartet und, sobald dies erfolgreich war, die älteren Versionen ersetzt.

Die Verbindung über ein gemeinsames Netzwerk bewirkt, dass von Anforderungen an Spezialhardware oder Hardwareschnittstellen abgesehen, der tatsächliche Ausführungsort der einzelnen Container keine Rolle spielt.

Lösungen wie OpenStack [Ope20b] werden hier nicht näher behandelt, da sie hauptsächlich auf Cloud- und Webdienste fokussiert sind, auch wenn sie sich einige Features mit Kubernetes und Docker Swarm teilen.

Docker Swarm Docker Swarm ist die Erweiterung des Docker-Konzepts auf einen Verbund aus mehreren Hosts, die über ein gemeinsames Netzwerk verbunden sind und zentral verwaltet werden. Vorteil gegenüber anderen Lösungen ist, dass diese Features bereits mit der Docker-Installation mitgeliefert werden und keine separate Einrichtung benötigen. [Doc20]

Standardmäßig sind nur die Hosts über das gemeinsame Netzwerk verbunden, die Container verfügen nur über eine Schnittstelle zum jeweiligen Host. Daher ist dieser für die Weiterleitung der Pakete und damit die Vernetzung der Container untereinander verantwortlich. Andererseits hat das auch den Vorteil der im Vorfeld besseren Trennung der Netzwerkteilnehmer voneinander.

Für die Verwaltung wird keine grafische Benutzeroberfläche mitgeliefert, ebenso sind keine Tools für Monitoring oder Logging enthalten.

Insgesamt ist Docker Swarm im Vergleich zu Kubernetes also eher auf kleinere, schnell einzurichtende Installationen spezialisiert.

Kubernetes Kubernetes [Kub20b] wird auch von der U.S. Airforce auf dem sich in Entwicklung befindenden strategischen Langstreckenbombers *Northrop Grumman B-21* eingesetzt [Rop20], mit der Luftfahrt ist Kubernetes also schon in einer dem Umfeld dieser Dissertation von den Anforderungen her ähnlichen Domäne im Einsatz; auch die Deutsche Bahn plant, zukünftig Kubernetes einzusetzen [JK20].

Dabei ist der Funktionsumfang umfangreicher, aber auch die Einrichtung aufwendiger als bei Docker. Außerdem ist die Nomenklatur eine andere (s. Tabelle 3.2 auf der nächsten Seite).

Mehrere Container lassen sich zu sogenannten *Pods* zusammenfassen, die sich gemeinsame Ressourcen wie IP-Adresse oder Dateisystem teilen. In *De-*

ployments werden replizierte Pods verwaltet, wodurch Redundanzen oder Skalierungen definiert werden können.

Ohne weitere Konfiguration sind alle Pods und Container über das Netzwerk verbunden und können miteinander kommunizieren.

Im Gegensatz zu Docker enthält Kubernetes im Auslieferungszustand bereits Tools für Monitoring und Logging, sowie eine grafische Benutzeroberfläche, das sogenannte *Dashboard*. Außerdem lässt es sich direkt in Gitlab (s. Abschnitt 3.7.2 auf Seite 94) integrieren.

Bezeichnung	Beschreibung
Container	Ausführbares Image mit einer atomaren Softwarekomponente und deren Abhängigkeiten.
Node	Eine virtuelle oder physische Maschine, auf der die einzelnen Container ausgeführt werden. Der *Master Node* überwacht und verwaltet den Cluster.
Cluster	Verbund von Nodes.
Pod	Das kleinste Objekt in Kubernetes, dabei kann ein Pod auch mehrere Container enthalten. Container in einem Pod teilen sich dabei Ressourcen wie Dateisystem oder IP-Adresse.
Deployment	Verwaltet mehrere replizierte Pods, um die Redundanz zu erhöhen oder die Performanz zu skalieren, können einem Deployment weitere Pods hinzugefügt werden.
Label	Unter einem Label können mehrere Kubernetes-Objekte wie Pods oder Services zusammengefasst und darüber ausgewählt werden.
Selector	Ein Selector wählt alle Objekte mit identischem Label aus
Replikation	Im Falle eines Updates oder Ausfalls eines Containers, wird vom *replication controller* ein neuer Pod mit den entsprechenden Containern gestartet und nahtlos auf diesen gewechselt, sobald dieser bereit ist.

Tabelle 3.2: Begriffsdefinitionen für Kubernetes [JAX19]

Emulation

Der Begriff Emulator stammt vom lateinischen *aemulatur* („er/sie/es wird nachgeahmt“) und bedeutet, dass bestimmte Hardware nachgebildet wird.

Im Umfeld dieser Dissertation ist dies die Prozessorarchitektur (Anforderung AP2), da zum Beispiel x86-CPUs und ARM-CPUs nicht miteinander *binärkompatibel* sind, die eine CPU also nicht für die andere kompilierten Code ausführen kann.

Die Nachahmung erfolgt daher über eine dynamische Übersetzung der Instruktionen im Binärcode in den Instruktionssatz der Zielarchitektur, vergleichbar zum Simultandolmetschen. Dies ist der Hauptunterschied zur Virtualisierung, bei der eine virtuelle Umgebung erzeugt, jedoch nicht zwischen Instruktionssätzen übersetzt wird. Daher ermöglicht eine Virtualisierung ohne Emulation kein Ausführen von Code einer anderen Prozessorarchitektur.

Einziger verbreiteter Emulator, der zwischen mehreren modernen Prozessorarchitekturen in beide Richtungen übersetzen kann, ist *QEMU* [QEM20]. Dabei ist die Besonderheit, dass es sich bei QEMU sowohl um einen Emulator als auch einen Virtualisierer handelt.

3.7.2 Continuous Integration, Delivery und Deployment

Eine CI/CD-Pipeline dient dazu, den Prozess in der Softwareentwicklung von der Integrations- und Testphase bis zum finalen Verteilen der Softwarekomponenten zu automatisieren [Red20].

Im Rahmen dieser Dissertation bedeutet dies, dass Änderungen an Softwarekomponenten direkt kompiliert, mittels automatischer Tests getestet und als Updates zur Laufzeit des Systems auf die verschiedenen ECUs ausgerollt werden können (Anforderungen AP3 und AP5). Dazu ist es wichtig, dass das CI/CD-System seine Ergebnisse im fahrzeugeigenen Software-Repository (s. Abschnitt 4.2.2 auf Seite 104) zur Verfügung stellt und nahtlos mit dem Orchestrator zusammenarbeitet.

Bei der *Continuous Integration* werden dazu automatisiert Änderungen am Quellcode in einem Versionsverwaltungssystem wie zum Beispiel *git* erfasst, kompiliert und durch automatische Tests auf die Funktionsfähigkeit untersucht. Erst dann werden diese Änderungen in die Codebasis übernommen. Dies ist zum Beispiel sinnvoll, wenn mehrerer Entwickler an einem gemeinsamen Projekt arbeiten und dabei zu verhindern ist, dass parallele Änderungen die Lauffähigkeit der entwickelten Software beeinträchtigen.

Continuous Delivery ist der nächste Schritt. In diesem werden die Änderungen, falls erfolgreich getestet, in einem Repository (zum Beispiel für Docker-Images, s. Abschnitt 3.7.1 auf Seite 89) bereitgestellt.

Die letzte Instanz wird als *Continuous Deployment* bezeichnet. Dieses umfasst das finale Ausrollen der Softwarekomponente auf das Produktivsystem.

Dadurch, dass alle diese Schritte vollkommen automatisiert ablaufen, ist kein manuelles Quality-Gate erforderlich oder vorhanden, dieses wird vollständig durch automatisiert durchgeführte Tests ersetzt. [Red20]

Gitlab

Gitlab unterstützt den gesamten CI/CD-Prozess. Ursprünglich eine webbasierte grafische Benutzeroberfläche für die Versionsverwaltung mit *git*, wurde es im Laufe der Zeit um weitere Funktionalitäten erweitert. [Git20]

Dazu gehört mit *Gitlab CI* ein System für CI/CD, zusätzlich werden praktisch alle im CI/CD-Umfeld notwendigen Dienste integriert, anders als in anderen CI/CD-Lösungen wie zum Beispiel *Jenkins* [Jen20].

Dies umfasst nicht nur die Versionsverwaltung, sondern auch den sogenannten *Gitlab Runner*, eine Registry für Docker-Images und eine Integration in Kubernetes.

Der *Gitlab Runner* wird von *Gitlab CI* genutzt, um Kompilieraufgaben und Softwaretests automatisch und auf verschiedenen sogenannten *Runnern*, die auf unterschiedlichen physischen Maschinen laufen können, auszuführen.

Das Ergebnis der CI kann anschließend in der integrierten Docker-Registry gespeichert und durch die Kubernetes-Integration auf ein eingebundenes Kubernetes-Cluster deployed werden. Damit kann die Gitlab-interne Docker-Registry die Aufgabe des Software-Repositories des Fahrzeuges erfüllen.

3.7.3 Related Work zur Rekonfiguration

Außerhalb der Automotive-Domäne wird Rekonfiguration zum Beispiel in der IT, in Sensornetzwerken oder in der Industrieautomatisierung eingesetzt, jeweils mit unterschiedlichen Zielsetzungen.

Gomaa et. al. stellen in [GH04] eine Möglichkeit vor, eine Softwarearchitektur zur Laufzeit zu rekonfigurieren, indem jede einzelne Softwarekomponente so angepasst wird, dass ein Zustand existiert, in welchem sie rekonfiguriert werden kann. Somit kann die Verschaltung der Softwarekomponenten und die Zustandsmaschine des Gesamtsystems dynamisch angepasst werden. Dabei handelt es sich hier um einen rein theoretischen Ansatz ohne eine tatsächliche praktische Umsetzung. Eine Erweiterung auf die Betrachtung der Echtzeitfähigkeit führen Rasche et. al. in [RP05] durch.

Die Rekonfiguration von Sensornetzwerken (zum Beispiel in Gebäuden) wird von Balani et. al. in [BHR+06] betrachtet. Dabei liegt der Fokus auf einem bereits existierenden Sensornetzwerk, dessen Knoten im Betrieb mit Softwareupdates versorgt werden, wodurch die Funktionalität angepasst werden kann. Das eigentliche Netzwerk wird nicht geändert, also keine Sensorknoten hinzugefügt oder entfernt. Einen ähnlichen Ansatz zeigen Mottola et. al. in [MPA08] für *kabellose* Sensornetzwerke; in [KHO14] diskutieren Kouche et. al. die Erweiterung der einzelnen Sensorknoten um zusätzliche Hardware-Module. Dabei muss die Software des Sensorknotens jedoch weiterhin angepasst werden, um diese neuen Module zu unterstützen.

Für Industrierobotor wird eine rekonfigurierbare Zelle in [GRB+17] vorgestellt. Dabei setzen Gaspar et. al. auf eine modulare Bauweise für die Rekonfigurierbarkeit der Hardware; die Modularität der Software wird über den Einsatz von ROS (s. Abschnitt 3.3.3 auf Seite 56) und dessen Nodes erzielt. Dabei ist der Aufbau nur statisch rekonfigurierbar und erfordert dazu manuelle Eingriffe, die Modularität ist hauptsächlich auf die Verringerung des manuellen Anpassungsaufwand an eine neue Produktionsstätte ausgelegt.

Ansätze zur Rekonfiguration im Automotive-Umfeld existieren bereits seit Anfang des Jahrtausends [UHGB04]. Ullmann et. al. nutzen allerdings rekonfigurierbare Hardware (FPGAs), um verschiedene (Hardware-)Funktionen zur Laufzeit auszutauschen und damit die Anzahl benötigter ECUs in einer verteilten E/E-Architektur (s. Abschnitt 3.1.3 auf Seite 46) zu reduzieren, indem nicht

alle Funktionen gleichzeitig ausgerollt sind. Damit eignet sich der Ansatz nicht für kritische Funktionen, die dauerhaft ausgeführt werden müssen.

Einen anderen Ansatz, bei dem die Verschaltung der ECUs untereinander anstatt nur der Hardware eines einzelnen Steuergeräts dynamisch rekonfiguriert wird, stellen Oszwald et. al. in [OBOT18] vor. Dabei geht es um die Ansteuerung Safety-kritischer Aktoren, die im Fehlerfall umgeschaltet werden muss. Dazu existieren zwei (oder mehr) ECUs, die den entsprechenden Aktor ansteuern können. Im Fehlerfall wird mechanisch über elektrische Schalter zwischen diesen Steuergeräten gewechselt. Dabei muss die Zeit bis umgeschaltet wird (FRT[35]) unter der Zeitspanne liegen, die zwischen Auftreten eines Defekts bis zu einem resultierenden gefährlichen Ereignis liegt. Oszwald et. al. berechnen dieses FTTI[36] gemäß ISO26262 (s. Abschnitt 2.1.3 auf Seite 18) für zwei Situationen: Dem Verlust der Lenkfunktion (FTTI: 130 ms) und dem Ausfall der Bremsfunktion (FTTI: 160 ms). Das Problem der Erhaltung des Kommunikationszustands wird von denselben Autoren in [OOTB19] gelöst, indem dafür ein Hardwaremodul (wie in [UHGB04] auf einem FPGA) genutzt wird, das die Kommunikation kapselt und somit beim Umschalten den Zustand erhalten kann.

In [FK16] stellen Farzaneh und Knoll die dynamische Integration von neuen Hardware-Komponenten in das Fahrzeugnetzwerk vor, dabei wird die Middleware *CHROMOSOME* aus dem RACE-Projekt (s. Anhang A.2.3 auf Seite 199) eingesetzt. Die zugrundeliegende Ontologie wird dabei manuell von einem entsprechenden Experten erstellt.

Die Abstraktion der Software von der Hardware behandeln Lin et. al. in [LKS20]. Dabei wird wie in dieser Dissertation eine Virtualisierung (s. Abschnitt 3.7.1 auf Seite 86) genutzt, um eine Portabilität in den verschiedenen Lebenszyklusphasen eines Fahrzeuges zu ermöglichen. Die Veröffentlichung beschränkt sich auf einer theoretischen Ebene mit einem einzelnen Steuergerät mit Virtualisierungsschicht, das Zusammenspiel eines gesamten dynamischen Fahrzeugnetzwerkes und damit das Verschieben von Softwarekomponenten zwischen ECUs wird nicht betrachtet.

[35] Fault Reaction Time
[36] Failure Tolerance Time Interval

Die Vision des Hinzufügen von (Safety-)Features auf Kundenwunsch diskutieren Gangadharan et. al. in [GKS+16]. Auch hier soll eine Abstraktion von der zugrundeliegenden Hardware das Hinzufügen von zusätzlichen Softwarekomponenten erleichtern. Außerdem werden Herausforderungen aufgezeigt, diese beinhalten unter anderem eine geeignete Spezifikation der Plattform, sodass sie mit zukünftigen Features kompatibel bleibt, Safety- und Securityaspekte sowie das Garantieren von benötigten Ressourcen zur Laufzeit. Dennoch ist es nicht vorgesehen, dass zusätzliche Hardwareressourcen in Form von intelligenten Sensoren beziehungsweise Aktoren oder neuen Steuergeräten hinzugefügt werden können.

In [KHS18] stellen Kugele et. al. eine dynamische Auslagerung von Services ins Backend abhängig von Betriebszustand und Umgebung eines Fahrzeuges vor, um die Beschränkung der Rechenleistung im Fahrzeug zu überwinden. Dabei kommt wie in dieser Dissertation Docker zur Containerisierung zum Einsatz, einzelne Services werden so voneinander getrennt, allerdings wird statt Kubernetes ein leichtgewichtigerer Orchestrator genutzt. Außerdem werden Strategien definiert, um die Verteilung basierend auf bestimmten Regeln oder Ereignissen durchzuführen.

3.8 Lücken des Standes der Wissenschaft und Technik

Im letzten Abschnitt wurde gezeigt, dass Rekonfiguration bereits in den verschiedensten Domänen Teil des Standes der Wissenschaft ist. Auch im Automobilbereich existieren Ansätze, die Software von der Hardware zu abstrahieren und damit eine Austauschbarkeit von Software- oder Hardwarekomponenten zu ermöglichen, zum Beispiel um Services ins Backend zu verlagern. Was jedoch fehlt, ist die dynamische Skalierung zur Erhöhung der Leistungsfähigkeit oder zur Verbesserung der Zuverlässigkeit *innerhalb* des Fahrzeuges während der Laufzeit des Fahrzeuges. Zusätzlich ist eine Betrachtung der vollständigen Architektur über den gesamten Lebenszyklus hinweg und ihre Anpassbarkeit auf unterschiedliche Fahrzeugtypen nicht Teil des Standes der Wissenschaft.

Untersucht man daher die Abdeckung der zu Beginn aufgestellten Herausforderungen (s. Abschnitt 1.2 auf Seite 1) durch die im Stand der Technik

verfügbaren Fahrzeugarchitekturen (s. Anhang A.2.3 auf Seite 198), zeigt sich, dass keine der Realisierungen alle Herausforderungen abdeckt. Manche Herausforderungen werden überhaupt nicht abgedeckt. Eine tabellarische Gegenüberstellung der Abdeckung der Herausforderungen des Standes der Technik mit der (re-)konfigurierbaren Fahrzeugarchitektur befindet sich in Tabelle A.9 auf Seite 209.

Die Ansätze aus der Automobilindustrie sind zu spezifisch auf herkömmliche PKW ausgelegt und sehen keine Konfigurierbarkeit der Komponenten nach Auslieferung vor. Auch die vorgestellten Projekte (s. Anhang A.2.3 auf Seite 199) decken jeweils nur einen Teil der Herausforderungen ab. UNICARagil beschränkt sich dabei zu sehr auf den Individualverkehr mit einer **vor** der Entwicklung des Fahrzeuges modularen und skalierbaren Architektur, die außerdem im jetzigen Zustand aufgrund ihrer Herkunft aus einem Forschungsprojekt teuren Overhead mit sich bringt. Beispielsweise enthalten die Sensorboxen vergleichsweise teure Sensorik, die nicht immer benötigt wird und deren individuelle Konfiguration nicht vorgesehen ist.

AKIT beschreitet dagegen einen fast entgegengesetzten Weg: Die Architektur wird auf bereits fertige Fahrzeuge angepasst, dabei werden aber nur sehr individuelle Einsatzbereiche für einzelne Fahrzeuge betrachtet. Damit ist der Ansatz für eine Serienproduktion völlig ungeeignet, da unwirtschaftlich, und selbst auch nicht mehr skalier- oder rekonfigurierbar sobald einmal für dieses Fahrzeug konzipiert.

RACE scheint auf den ersten Blick die meisten zu Beginn gestellten Herausforderungen zu erfüllen, geht jedoch nicht weiter auf Lösungen für die in Abschnitt 1.6 auf Seite 8 identifizierten weitergehenden Anforderungen ein, ebenso existieren die meisten der Lösungen nur in der Theorie und wurden nie weiter detailliert. So wird ein dynamisches Verschieben von Softwarekomponenten auf andere Steuergeräte oder Ausweichen auf andere Sensoren während der Fahrt aufgrund von Ausfällen nicht berücksichtigt, ebenso wenig Updates zur Laufzeit oder unterschiedliche Hardwarearchitekturen.

4 Idee und Konzept

4.1 Idee und Beitrag dieser Dissertation

Der Stand der Wissenschaft und Technik ist bisher stark auf einzelne Domänen (beispielsweise PKW) oder einzelne Anwendungsbereiche (beispielsweise automatisiertes Fahren) fokussiert. Weniger stückzahlenreiche Bereiche wie Busse oder die Lücke zwischen Produktlebenszyklen in der IT und in der Automobilbranche erfahren wenig Beachtung. So ist es aktuell nicht vorgesehen, Fahrzeuge während ihres Lebenszyklusses an neue Herausforderungen wie eine notwendige beziehungsweise gewünschte Erhöhung der Funktionalität oder fehlende Verfügbarkeit von Komponenten anzupassen, zum Beispiel durch den Austausch gegen modernere oder das Hinzufügen zusätzlicher Komponenten.

Gerade dies bietet jedoch, von der Erhöhung der Funktionalität oder der Verbesserung der Verfügbarkeit von Komponenten abgesehen, auch die Chance, Kosten zu sparen: Bei der Entwicklung, da nicht der gesamte Entwicklungsprozess erneut durchlaufen werden muss, und indem auf Standardkomponenten ausgewichen werden kann, die durch hohe Stückzahlen geringere Einkaufskosten mit besserer Verfügbarkeit kombinieren. Eine Lagerhaltung von Komponenten über den gesamten Lebenszyklus eines Fahrzeuges hinweg kann so zumindest teilweise entfallen. Damit können sowohl Lagerhaltungs- als auch Teilekosten reduziert werden, da nicht aufgrund unbekannter Stückzahlen für Wartung und Reparatur benötigter Teile zu große Mengen bevorratet werden müssen.

Diese Dissertation liefert daher einen Beitrag, diese Lücken im Stand der Wissenschaft und Technik zu schließen: Die (re-)konfigurierbare Fahrzeugarchitektur. Dabei werden die Begriffe *Konfiguration* und *Rekonfiguration* gemäß der Definitionen 1.3.1 und 1.3.2 auf Seite 5 verwendet, um ein gemeinsames Bild zu schaffen und Mehrdeutigkeiten zu vermeiden.

Eine moderne SOA stellt eine geeignete Basis dar, um die notwendige Flexibilität zu ermöglichen und es zu erlauben, Komponenten und Services zur Laufzeit hinzufügen und entfernen zu können. Die Abstraktion der Funktionalität beziehungsweise Software des Fahrzeuges von der zugrundeliegenden Hardware schafft die Grundlage, um die definierten Portabilitätsanforderungen zu erfüllen (s. Abschnitt 1.6.2 auf Seite 9). Eine Zentralisierung der Rechenleistung reduziert den Kommunikationsoverhead im Fahrzeug und vereinfacht die Kommunikationsinfrastruktur. Konsequenz daraus ist außerdem, dass bisher nicht verfügbare dynamische Skalierungs- und Redundanzkonzepte möglich werden. Es können also Services und Softwarekomponenten auf mehreren Steuergeräten gleichzeitig ausgeführt werden, um entweder die Leistungsfähigkeit beziehungsweise den Durchsatz von beispielsweise Algorithmen zu erhöhen, beziehungsweise um die Zuverlässigkeit von sicherheitskritischer Funktionalität zu steigern.

Für die der (re-)konfigurierbaren Fahrzeugarchitektur zugrundeliegende Hardware wird auf Standardkomponenten (COTS[1]) aus dem IT-Bereich gesetzt. Dadurch können auch in Domänen mit geringen Stückzahlen (zum Beispiel Busse) geringe Kosten erzielt werden. Neu ist hier, dass Komponenten durch neu verfügbare ersetzt werden können, wodurch die kürzeren Lebenszyklen der IT-Industrie nicht negativ ins Gewicht fallen, sondern sogar im Positiven für einen Funktionalitätshub oder eine Energieeffizienzsteigerung genutzt werden können.

4.2 Konzept für die Umsetzung einer (re-) konfigurierbaren Fahrzeugarchitektur

4.2.1 Die (re-)konfigurierbare Fahrzeugarchitektur

Ausgehend von der Idee dieser Dissertation werden im Folgenden die aufgestellten Anforderungen konzeptionell umgesetzt. In den Tabellen A.17 bis A.19 auf den Seiten 216–218 ist die Abdeckung der Anforderungen mit Kurzbe-

[1] Commercial off-the-shelf

schreibung des Lösungsansatzes und Verweis auf die entsprechenden Abschnitte dargestellt.

Die Definition 1.3.2 auf Seite 5 versteht unter Rekonfiguration im Rahmen dieser Dissertation *„den Austausch von Hardwarekomponenten wie Sensoren, Aktoren oder ECUs, aber auch von Softwarekomponenten, die entweder neu hinzugefügt, aktualisiert oder zwischen Steuergeräten verschoben werden“*.

Eine mögliche Struktur für eine solche (re-)konfigurierbare Fahrzeugarchitektur ist in Abbildung 4.1 auf der nächsten Seite dargestellt. Softwarekomponenten können frei zwischen den Steuergeräten verschoben werden und auch redundant ausgeführt werden, um Safetyanforderungen (AS1) zu erfüllen. Steuergeräte mit unterschiedlichen Komponenten (wie zum Beispiel GPUs oder FPGAs) und Sensoren oder Aktoren können dynamisch hinzugefügt beziehungsweise entfernt werden[2].

Das für die prototypische Umsetzung genutzte *Kubernetes* (s. Abschnitt 5.4 auf Seite 150) unterstützt bis zu 5000 Nodes und damit ECUs beziehungsweise intelligente Sensoren oder Aktoren [Kub20a], stellt also keine realistische Beschränkung der maximalen Komponentenanzahl dar.

Um Softwarekomponenten auf die einzelnen ECUs verschieben zu können, muss eine zentrale Ablage für diese eingerichtet werden, das sogenannte *Software-Repository* (D). Gleichzeitig muss darauf geachtet werden, dass die Softwarekomponente nur auf solche Steuergeräte verschoben werden, die die benötigten Ressourcen zur Verfügung stellen können.

Diese Aufgabe übernimmt der sogenannte *Orchestrator* (C) (s. Seite 90), der den Verbund der heterogenen Steuergeräte verwaltet. Zusammen mit dem Software-Repository übernimmt er die Konfiguration der Fahrzeugsoftware und bestimmt damit, welche Komponente auf welcher ECU ausgeführt wird. Dies umfasst auch das Ausrollen von Sicherheitsupdates und die Sicherstellung der Redundanz safety-relevanter Komponenten.

Orchestrator und Software-Repository bilden im Rahmen dieser Dissertation eine Einheit und werden in Abschnitt 4.2.2 auf Seite 104 genauer beschrieben.

[2] Dies ist außerdem ein essentieller Teil von Use-Case 2: „Austausch von Komponenten“ und wird daher in Abschnitt 4.5 auf Seite 122 detaillierter behandelt.

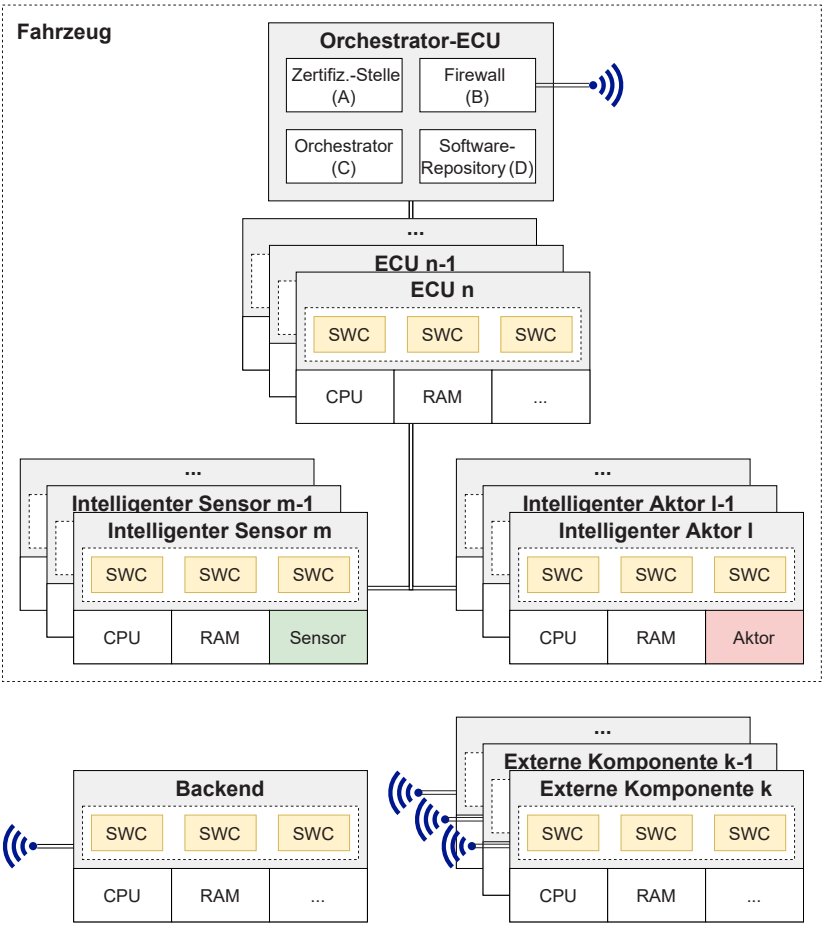


Abbildung 4.1: Mögliche Struktur der (re-)konfigurierbaren Architektur.

Außerdem muss sichergestellt sein, dass die Softwarekomponente und die Hardwarearchitektur der Ziel-ECU kompatibel sind. Dies kann auf zwei Wegen erfolgen, die in dieser Arbeit beide evaluiert werden:

1. Emulation (s. Seite 92) einer zur Softwarekomponente kompatiblen Plattform

2. Erstellen beziehungsweise Kompilieren der Softwarekomponente für jede möglicherweise eingesetzte Hardwareplattform

Beide dieser Wege haben individuelle Vor- und Nachteile. Bei der Emulation muss daher zum Beispiel der zwangsweise vorhandene Overhead und damit Mehrverbrauch an Rechenleistung sowie Arbeitsspeicherkapazität untersucht und mit dem Mehraufwand der Mehrfachkompilierung verglichen werden. Dies schließt nicht nur die eigentliche Kompilierzeit, sondern auch gegebenenfalls notwendige Modifikationen der Software und die Problematik von bei der Erstellung der Softwarekomponente noch nicht verfügbaren Architekturen, ein.

Bei der Abwägung dieser Vor- und Nachteile muss außerdem berücksichtigt werden, in welchem Abschnitt des Lebenszyklusses des Fahrzeuges Emulation und Kompilierung stattfinden. Erstere kann an zwei Punkten mit unterschiedlichen Auswirkungen zum Einsatz kommen: Während der Entwicklung des Fahrzeuges, um die Softwarekomponenten auf unterschiedlichen nicht real vorhandenen Hardwarearchitekturen ausführen zu können und nach der Auslieferung während der Laufzeit des Fahrzeuges beim Kunden, zum Beispiel um die Ausführung von Softwarekomponenten auf einer bestimmten Hardwareplattform zu ermöglichen. In letzterem Fall ergeben sich direkt Auswirkung auf Performanz und Energieverbrauch des Fahrzeuges, die kritisch sein können. Die Kompilierung findet üblicherweise ausschließlich in der Entwicklungsphase von Fahrzeug oder neuen Komponenten statt, hat also keine Auswirkungen auf Performanz oder Energieverbrauch beim Kunden. Allerdings kann es denkbar sein, dass Softwarekomponenten zum Beispiel bei einem Update einmalig beim Kunden kompiliert werden, um Bandbreite bei der Übertragung einsparen und Anpassungen an die Hardwareplattform des Fahrzeugs vornehmen zu können.

Eine zentrale Firewall dient dazu, dass die Verbindung des Systems zur Außenwelt (zum Beispiel zum Backend) abgesichert ist. Das Backend stellt dabei eine räumlich vom Fahrzeug getrennte und üblicherweise unbewegliche Infrastruktur dar, die Services für das Fahrzeug beziehungsweise die gesamte Flotte bereitstellen kann. Mögliche Betreiber des Backends sind der Fahrzeughersteller im Rahmen eines Servicevertrages mit dem Kunden oder der Betreiber der Straßeninfrastruktur zur Optimierung der Verkehrsführung. So können zum Beispiel von Umfeldsensorik erstellte Objektlisten oder Softwareupdates dem Fahrzeug zur Verfügung gestellt werden.

4.2.2 Die Orchestrator-ECU

Da beide Komponenten für eine korrekte Funktion essentiell sind, bietet es sich an, Orchestrator und Software-Repository gemeinsam auf einem besonders zuverlässigen (im Idealfall redundant ausgelegten) Steuergerät auszuführen. Dieses wird im Folgenden als *Orchestrator-ECU* bezeichnet.

Eine ebenso wichtige, grundlegende Funktionalität, die hier ausgeführt wird, ist die der **zentralen Zertifizierungsstelle (A)**. Diese verwaltet Zertifikate für die Komponenten im Fahrzeug, erstellt neue und kann auch gegebenenfalls kompromittierte Zertifikate zurückziehen. Dieses Vorgehen ist sicherer als feste Passwörter, da einzelne Komponenten ausgeschlossenen werden können und die Zertifikate schwerer für Unbefugte zu erlangen und verbreiten sind. Damit wird sichergestellt, dass alle Komponenten authentifiziert werden können und kein unautorisierter Zugriff auf Orchestrator und Software-Repository oder andere wichtige Komponenten erfolgen kann. Somit kann weder die Konfiguration manipuliert noch fremder Code eingeschleust werden. Zusätzlich werden die Zertifikate zur Transportverschlüsselung genutzt, Unbefugte können also weder Nachrichten senden noch lesen (Sicherheitsanforderungen AS3 und AS4). Um das Problem zu umgehen, dass es möglich sein muss, neue Komponenten hinzufügen, die der zentralen Zertifizierungsstelle des Fahrzeuges nicht bekannt sind, lassen sich weitere Zertifizierungsstellen hinzufügen. So können zum Beispiel Fahrzeug- und Komponentenhersteller ihre Komponenten signieren und das Fahrzeug akzeptiert auch diese Zertifikate. Neue Zertifizierungsstellen (zum Beispiel ein neuer Hersteller) oder Listen mit zurückgezogenen Zertifikaten (im Falle einer Kompromittierung, zum Beispiel durch Veröffentlichung des privaten Schlüssels) müssen anschließend über (OTA)-Updates verteilt werden.

Damit auch auf dem Weg ins Backend keine Daten mitgelesen oder manipuliert werden können, wird über die **Firewall (B)** ein verschlüsselter Tunnel, ein sogenanntes VPN[3] zum Backend aufgebaut. Dieses stellt die Verbindung zwischen den in Abbildung 4.1 auf Seite 102 als Antennen dargestellten Schnittstellen des Fahrzeuges und Backends zu einem äußeren Netzwerk (zum Beispiel über Mobilfunk) dar. Alle anderen Zugriffe von außerhalb ins Fahrzeug oder umgekehrt werden unterbunden. Außerdem verwaltet die Firewall die

[3] Virtual Private Network

Verteilung von IP-Adressen und die Namensauflösung (DNS[4]). Damit ist sichergestellt, dass alle Komponenten miteinander kommunizieren können. Zusätzlich stellt der Orchestrator darauf basierend ein Softwarekomponentenübergreifendes Netzwerk zur Verfügung, sodass es für die Kommunikation keine Rolle spielt, welche Softwarekomponente auf welcher ECU oder gar im Backend ausgeführt wird.

Hauptaufgabe des **Orchestrators (C)** ist die Verteilung der Softwarekomponenten auf die einzelnen ECUs (s. Abschnitt 4.2.5 auf Seite 109), zum einen einmalig beim Start des Fahrzeuges, aber auch bei Änderungen zur Laufzeit, zum Beispiel bei Ausfall einer Komponente. Safety-relevante Softwarekomponenten werden dabei mehrfach parallel auf unterschiedlichen ECUs ausgeführt (s. Abschnitt 4.2.5 auf Seite 109). Ähnlich erfolgt auch das Updaten von einzelnen Softwarekomponenten (s. Abschnitt 4.2.8 auf Seite 117).

Quelle dieser Softwarekomponenten ist das **Software-Repository (D)** (für die unterschiedlichen Anwendungsgebiete in den Lebenszyklusphasen eines Fahrzeuges, s. Abschnitt 4.2.3 auf der nächsten Seite). Dieses enthält alle bekannten Softwarekomponenten, gegebenenfalls in verschiedenen Varianten für unterschiedliche Zielhardware oder in mehreren Versionen, um bei fehlgeschlagenen Updates zur Vorgängerversion zurückkehren zu können. Die bekannten Softwarekomponenten umfassen dabei die mit dem Fahrzeug mitgelieferten und außerdem die durch zusätzlich verbaute Komponenten hinzugekommenen Softwarekomponenten. Zusätzlich hält das Software-Repository auch die bereits angesprochenen Anforderungen der Softwarekomponenten bereit, die der Orchestrator zur Verteilung auf die Steuergeräte nutzt.

Außerdem ist es möglich, externe Software-Repositories (zum Beispiel beim Fahrzeughersteller) einzubinden, von welchen Softwarekomponenten bezogen werden können. Damit wird vermieden, dass Fahrzeuge alle zur Verfügung stehenden Softwarekomponenten lokal vorhalten müssen. Sinnvoll ist ein solches Vorgehen zur Verteilung von Softwareupdates oder vom Kunden nachträglich gekaufter und in Software realisierter zusätzlicher Funktionalität.

[4] Domain Name System

4.2.3 Das Software-Repository in den verschiedenen Lebenszyklusphasen des Fahrzeuges

In den unterschiedlichen Lebenszyklusphasen eines Fahrzeuges (s. Abbildung 4.2) werden dem Software-Repository verschiedene Aufgaben zuteil.

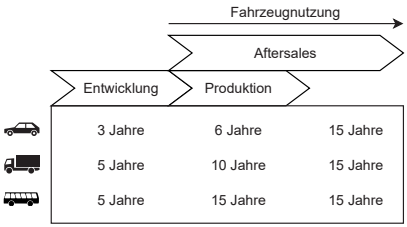


Abbildung 4.2: Lebenszyklusphasen eines Fahrzeuges (nach Tabelle A.3 auf Seite 204, [SGS+21])

Entwicklung

Während der Entwicklungsphase kann die CI/CD-Pipeline (s. Abschnitt 3.7.2 auf Seite 93) genutzt werden, um den in Entwicklung befindlichen Quellcode von Softwarekomponenten des Fahrzeuges automatisiert zu kompilieren, mittels automatisierter Tests zu testen und auf Prototypen auszurollen. Dazu wird diese Pipeline automatisch gestartet, sobald Änderungen vorgenommen wurden. Die entstehenden Softwarekomponenten werden in die Software-Repositories der Prototypen hochgeladen (*Continuous Delivery*) und von dort aus verteilt (*Continuous Deployment*).

Produktion

Ändern sich gesetzliche Rahmenbedingungen, die Verfügbarkeit von Komponenten oder Technologien, kann auch während der Produktionsphase eine Anpassung der Fahrzeugsoftware notwendig werden. Dabei wird wie in der Entwicklungsphase vorgegangen, um die Software im Repository des Herstellers zu aktualisieren, sodass beim „Flashen am Bandende“ die aktualisierten Softwarekomponenten in das fahrzeuginterne Software-Repository übertragen werden können.

Aftersales

Befindet sich das Fahrzeug beim Kunden, wird das interne Repository für die dynamische Umverteilung von Softwarekomponenten auf die Steuergeräte genutzt, A) um Redundanzen zur Erhöhung der Zuverlässigkeit zu schaffen (s. Abschnitt 4.2.5 auf Seite 109) oder B) beim Hinzufügen neuer Komponenten und Funktionalitäten und dem Verteilen der zugehörigen Software (s. Abschnitt 4.2.4 auf der nächsten Seite). Außerdem kommt es bei Softwareupdates zum Einsatz (s. Abschnitt 4.2.8 auf Seite 117).

4.2.4 Einbinden und Entfernen von ECUs

Neu hinzugefügte Steuergeräte, darunter auch intelligente Sensoren und Aktoren, müssen sich am System anmelden können, ohne Vorkenntnisse über dieses zu haben (Anforderung AA2) oder solche vom System zu erfordern (Anforderung AA3). Ebenso wie der Zugriff auf das Software-Repository oder die Kommunikation muss dieser Anmeldeprozess entsprechend sicher sein, nur autorisierte ECUs dürfen sich am System anmelden können. Daher muss jedes neu hinzugefügte Steuergerät ein von der zentralen Zertifizierungsstelle oder einer anderen vertrauenswürdigen Stelle signiertes Zertifikat besitzen. Mit diesem kann es sich an der Orchestrator-ECU authentifizieren und erhält von dieser weitere Instruktionen zum Anmeldevorgang. Damit ist die Kompatibilität der Anmeldung über verschiedene Softwaregenerationen hinweg sichergestellt.

Anschließend wird die Anmeldung vollautomatisch durchgeführt. Nach der Anmeldung werden Informationen über Fähigkeiten und Anforderungen des Steuergerätes oder Sensors beziehungsweise Aktors übertragen, gegebenenfalls auch Softwarekomponenten ins Software-Repository übermittelt (s. auch Abbildung 4.3).

Die Metadaten der Fähigkeiten umfassen bei Sensoren Eigenschaften wie Auflösung oder Bildwinkel und Positionen und ermöglichen damit beispielsweise den durch die Sensorik abgedeckten Bereich der Umgebung zu bestimmen. Bei Aktoren werden Informationen wie mögliche Lenkwinkel oder Drehmomente übermittelt, bei ECUs Eigenschaften wie verfügbare Rechenkapazitäten oder enthaltene Spezialhardware. Außerdem kann so gegebenenfalls die Kompati-

bilität mit dem Fahrzeug und den bereits verbauten Komponenten überprüft werden. Im Idealfall ist aber durch die Abstraktion der Softwarekomponenten von der Hardwareplattform ein reibungsloses Zusammenspiel gewährleistet.

Ähnlich wie die Anmeldung erfolgt auch das Abmelden von nicht mehr vorhandenen oder defekten ECUs, aber auch bei Inkompatibilitäten: Entweder vom zu entfernenden Steuergerät initiiert oder wenn erkannt wird, dass die ECU eine bestimmte Zeitspanne lang nicht mehr erreichbar war.

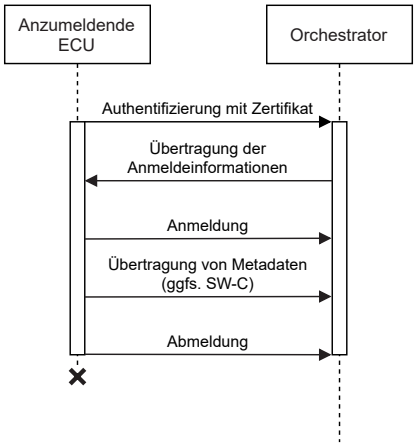


Abbildung 4.3: Anmelde- und Abmeldevorgang eines neuen Steuergerätes im Fahrzeug [SGS+21]

Hinzufügen der Software zum Software-Repository

Für die Funktionalität des hinzugefügten Steuergerätes beziehungsweise Sensors oder Aktors ist üblicherweise spezielle Software notwendig. Um die Software gegebenenfalls auf andere ECUs verteilen zu können, zum Beispiel um sie redundant auszuführen, muss sie in das Software-Repository des Fahrzeuges übertragen werden. In diesem verbleibt sie bis zum endgültigen Entfernen der zugehörigen Komponente. Auch hier muss sichergestellt sein, dass Angreifer diesen Weg nicht nutzen können, um Schadsoftware in das System zu schleusen. Dazu werden nur Softwarekomponenten aus vertrauenswürdigen Quellen akzeptiert, die von der zentralen Zertifizierungsstelle akzeptierte Zertifikate zur Authentifizierung besitzen.

4.2.5 Verteilung von Services auf die ECUs

Spätestens beim ersten Starten des Fahrzeuges müssen die Softwarekomponenten und Services aus dem Software-Repository auf die ECUs ausgerollt werden. Ebenso müssen aktualisierte oder neu hinzugefügte Softwarekomponenten verteilt werden. Um die Anforderungen an die Portabilität (Anforderungen AP1, AP2, AP3, AP4, AP5) umzusetzen, werden die einzelnen Softwarekomponenten von der zugrundeliegenden Hardwareplattform abstrahiert und vom Orchestrator als gebündelte Einheiten auf die verschiedenen ECUs verteilt.

Dies schließt ein, dass Anforderungen der Softwarekomponenten beachtet werden, welche zum einen Ressourcen wie Speicher und CPU-Leistung, aber auch Spezialhardware wie GPUs oder Sensoren und Aktoren umfassen. Um die Effizienz zu maximieren, werden mehrere Softwarekomponenten gemeinsam auf einem Steuergerät ausgeführt. Dabei ist es essentiell, dass vermieden wird, dass fehlerhafte Softwarekomponenten so viele Ressourcen auslasten können, dass andere nicht mehr ihre Anforderungen an Qualität oder Echtzeitfähigkeit erfüllen können. Dazu ist der Orchestrator in der Lage, individuelle und feste Ressourcenbeschränkungen zuzuweisen und zu überwachen (Ähnliches wird von AUTOSAR Adaptive unterstützt, nicht aber von ROS2). Gegebenenfalls müssen die Softwarekomponenten auf eine andere ECU verschoben werden.

Außerdem wird, indem dieselbe Softwarekomponente auf mehreren Steuergeräten parallel ausgeführt wird, so auch die Safetyanforderung AS1 in Form einer Hardwareredundanz abgedeckt (s. folgenden Abschnitt).

Redundanz durch Verteilung der Services auf mehrere ECUs

Um eine Redundanz zur Erhöhung der Zuverlässigkeit durch geeignetes Verteilen auf mehrere ECUs zu erreichen, gibt es zwei Möglichkeiten:

- Redundantes Anbieten von Services basierend auf Fähigkeiten und Umschalten im Service-Konsumenten (s. Abschnitt 4.2.7 auf Seite 112)
- Nachbilden von Hardwareredundanz-Konzepten wie Triple Modular Redundancy (s. im Folgenden)

Da die Services nach Anforderung AS2 *fail-silent* sind, kann erstere Möglichkeit genutzt werden, um Redundanz mit geringem zusätzlichen Ressourcenbedarf umzusetzen.

Sollte diese Anforderung nicht umsetzbar sein, mehrere Komponenten denselben redundant auszulegenden Service benötigen oder höhere Anforderungen an die Zuverlässigkeit bestehen, können mit der (re-)konfigurierbaren Fahrzeugarchitektur auch andere Redundanzkonzepte umgesetzt werden. Bei TMR[5] werden hier dabei identische Softwarekomponenten beziehungsweise solche mit identischen oder vergleichbaren Ein- und Ausgabedaten dreifach redundant ausgeführt. Ein sogenannter *Voter* entscheidet per Mehrheitsentscheid über das gültige Ergebnis (s. Abbildung 4.4).

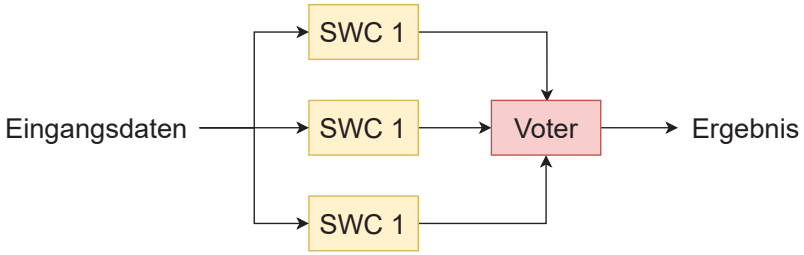


Abbildung 4.4: Triple Modular Redundancy (TMR)

Im stark vereinfachten Fall, dass die Softwarekomponente, die ausführende Hardware und deren Ressourcenauslastung für alle drei Instanzen identisch sind (mit der Überlebenswahrscheinlichkeit $R_{\text{SWC 1}}$), ergibt sich für die Wahrscheinlichkeit, ein korrektes Ergebnis zu erhalten (R_{TMR}), mit der Überlebenswahrscheinlichkeit des Voters (R_{Voter}) der in Gleichung 4.1 dargestellte Zusammenhang:

$$R_{\text{TMR}} = R_{\text{Voter}} \cdot (3\, R^2_{\text{SWC 1}} - 2\, R^3_{\text{SWC 1}}) \tag{4.1}$$

Damit ist also der Einfluss der Zuverlässigkeit des Voters auf die gesamte Zuverlässigkeit höher als die der einzelnen Softwarekomponenten, wodurch

[5] Triple Modular Redundancy

der Voter eine deutlich höhere Überlebenswahrscheinlichkeit aufweisen muss, um sinnvoll zu sein.

Angelehnt an Abbildung 4.1 auf Seite 102 sieht die Verteilung in einer beispielhaften Architektur mit zwei ECUs und intelligentem Sensor und Aktor anschließend wie in Abbildung 4.5 dargestellt aus. Dabei wird dieselbe Softwarekomponente mehrfach auf unterschiedlichen Steuergeräten oder im Backend ausgeführt. Der Voter befindet sich davon getrennt auf einer weiteren ECU mit geringerer Ausfallwahrscheinlichkeit. In diesem Fall bietet also der Voter die Fähigkeiten an, die Service-Konsumenten nutzen können (s. Abschnitt 4.2.7 auf der nächsten Seite).

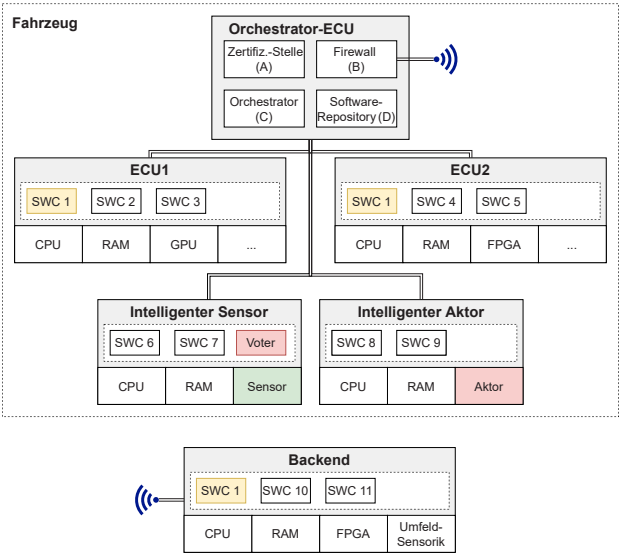


Abbildung 4.5: Beispiel für Einsatz von Triple Modular Redundancy (TMR)

4.2.6 Anbieten und Nutzen neuer Services

In dieser Dissertation wird eine Art erweitertes Plug-and-Play-System eingesetzt, um automatisch auf optimale Services umschalten zu können. Dabei werden Komponenten nicht nur über angebotene Services, sondern auch über

ihre sogenannten *Fähigkeiten* charakterisiert. Diese verknüpfen einen Service (zum Beispiel das Erfassen eines Bildes der Fahrzeugumgebung) mit durch Metriken automatisiert bewertbaren (Güte-)Eigenschaften. Dies kann zum Beispiel die Auflösung oder der Blickwinkel einer Kamera sein, aber auch die Möglichkeit ein Farbbild aufzunehmen gegenüber einer Schwarzweißkamera.

Ist eine neue Fähigkeit verfügbar, muss sie dem System mitsamt ihrer Eigenschaften bekannt gemacht werden. Dazu bietet es sich an, einen allgemeinen Kanal zu erstellen, den alle interessierten Konsumenten abonnieren und neue Fähigkeiten entsprechend auf diesem anzukündigen. Außerdem muss der entgegengesetzte Fall berücksichtigt werden, die *Nicht*verfügbarkeit dieser Fähigkeit, zum Beispiel durch den Ausfall eines Sensors oder Steuergeräts.

Da mehrere Aspekte zu einer fehlenden Verfügbarkeit führen können und der Service-Anbieter nicht in allen Fällen in der Lage ist, diesen Ausfall selbst anzukündigen, wird hier ein sogenannter *Heartbeat* eingesetzt: Auf einem dafür vorgesehen Kanal senden alle Service-Anbieter mit festgelegter Frequenz Lebenszeichen. Bleiben diese aus, gilt der Service als ausgefallen.

Im Rahmen einer maximalen Flexibilität wird auch die Verfügbarkeit dezentral verwaltet, jeder Service-Konsument überwacht selbstständig die Verfügbarkeit der benötigten Fähigkeiten. Somit kann auch individuell für jeden Konsumenten die Anzahl tolerierter ausbleibender Heartbeats festgelegt werden. Dies ermöglicht es, den Ressourcenverbrauch bei weniger zeitkritischen Aufgaben zu reduzieren.

4.2.7 Umschaltung auf neue oder redundant verfügbare Services

„Managed Nodes“ und ihre Zustände

Das Konzept der *Managed Nodes* (s. Abschnitt 3.6.1 auf Seite 73) ist in ROS2 bereits im Ansatz verfügbar, wird aber stark erweitert, um die Anforderungen dieser Dissertation zu erfüllen (s. auch [SGS+21]).

Ziel ist es, den Lebenszyklus der Softwarekomponenten (in Form von ROS2-Nodes) so zu verwalten, dass zwischen den angebotenen Services neu hinzugefügter oder redundant vorhandener Komponenten wie ECUs oder intelligen-

ten Sensoren basierend auf einer vorgegebenen Metrik umgeschaltet werden kann und gleichzeitig sichergestellt ist, dass die Funktionalität des Fahrzeuges erhalten bleibt oder gegebenenfalls gezielt degradiert werden kann.

Dazu wird ein zusätzlicher Zustand für Nodes hinzugefügt, in dem verharrt wird, bis alle Abhängigkeiten erfüllt sind (*Abhängigkeitssuche*). Erst dann wird in den aktiven Zustand gewechselt, in dem die eigentlich Funktionalität des Nodes ausgeführt wird.

Ebenso neu hinzugefügt wurde der bereits angesprochene *Heartbeat*, um den Ausfall von Nodes erkennen zu können. Damit wird eine Art Watchdog-Funktionalität integriert, die eine Reaktion auf zu langes Ausbleiben von Lebenszeichen der Nodes ermöglicht.

Insgesamt ergeben sich also die folgenden fünf Zustände, in denen sich ein Node befinden kann und die im Folgenden genauer erläutert werden (s. auch Abbildung 4.6 auf Seite 115): *Unkonfiguriert*, *Inaktiv*, *Abhängigkeitssuche*, *Aktiv* und *Herunterfahren*.

Unkonfiguriert In diesem Zustand befindet sich ein Node direkt nach dem Start des Fahrzeuges oder im Fehlerfall. Er kann nun konfiguriert werden, dabei werden zum Betrieb notwendige Parameter gesetzt und außerdem die Kommunikation initialisiert. War die Konfiguration erfolgreich, wird in den Zustand *Inaktiv* gewechselt, andernfalls bleibt der Node im Zustand *Unkonfiguriert*. Durch einen Befehl zum Herunterfahren kann der Node außerdem in den Zustand *Herunterfahren* überführt werden.

Inaktiv Nach der Konfiguration ist der Node prinzipiell betriebsbereit. Da die Funktionalität noch nicht ausgeführt wird, können außerdem noch Parameter angepasst werden, ohne diese zur Laufzeit zu beeinflussen. Soll der Node gestartet werden und dazu die Verfügbarkeit der Abhängigkeiten abwarten beziehungsweise überprüfen, wird er in den Zustand *Abhängigkeitssuche* überführt. Dabei wird außerdem eine Suchanfrage mit den zum Start benötigten Fähigkeiten an das gesamte System versendet.

Falls der Node nicht gestartet werden soll, kann die Konfiguration gelöscht (resultiert in Zustand *Unkonfiguriert*) oder der Befehl zum Herunterfahren (resultiert in Zustand *Herunterfahren*) gegeben werden.

Abhängigkeitssuche Bei der Abhängigkeitssuche wird solange gewartet, bis alle notwendigen Abhängigkeiten verfügbar sind. Sind Abhängigkeiten zum Betrieb nicht essentiell, können sie als *weich* deklariert werden. Damit kann der Node seine Funktionalität (gegebenenfalls eingeschränkt beziehungsweise degradiert) bereits aufnehmen, ohne dass diese Abhängigkeit verfügbar ist, indem in den Zustand *Aktiv* gewechselt wird. Außer der sich dadurch ergebenden Fähigkeit zur Degradation wird so vermieden, dass Zirkelabhängigkeiten das gesamte System am Starten hindern.

Auch aus diesem Zustand heraus kann in den inaktiven Zustand zurückgewechselt oder das Herunterfahren des Nodes eingeleitet werden.

Aktiv Im aktiven Zustand startet der Node die Ausführung seiner Funktionalität und kündigt seine eigenen Fähigkeiten an. Damit kann der Node anderen als Abhängigkeit dienen, außerdem wird ein Heartbeat-Signal gesendet, das diesen Nodes die Lebendigkeit anzeigt. Ebenso wird zyklisch überprüft ob alle Abhängigkeiten noch verfügbar sind. Ist dies für eine notwendige Abhängigkeit nicht der Fall wird die Ausführung gestoppt und in den Zustand *Inaktiv* gewechselt, bei einer weichen Abhängigkeit wird der Funktionsumfang gegebenenfalls reduziert.

Wie jeder andere Zustand erlaubt auch der aktive Zustand den Übergang in den inaktiven Zustand oder das Herunterfahren des Nodes.

Herunterfahren Der Zustand *Herunterfahren* ist immer der letzte Zustand und wird eingenommen, wenn der Node beendet werden soll. Dies ist zum Beispiel beim Herunterfahren des Systems bei Abstellen des Fahrzeuges der Fall. Dabei werden alle Aufgaben durchgeführt, die zum ordnungsgemäßen Herunterfahren notwendig sind, zum Beispiel ein Schließen von offenen Dateien und Verbindungen, um Datenverlust vorzubeugen.

Verwaltung des Lebenszyklus der Nodes

Jeder Node enthält einen Manager für seinen Lebenszyklus, den *Lifecycle-Manager*. Dieser besteht aus einer Klasse, die auf den aktuellen Zustand als Zustandsobjekt verweist. Dabei gibt es für jeden möglichen Zustand eine

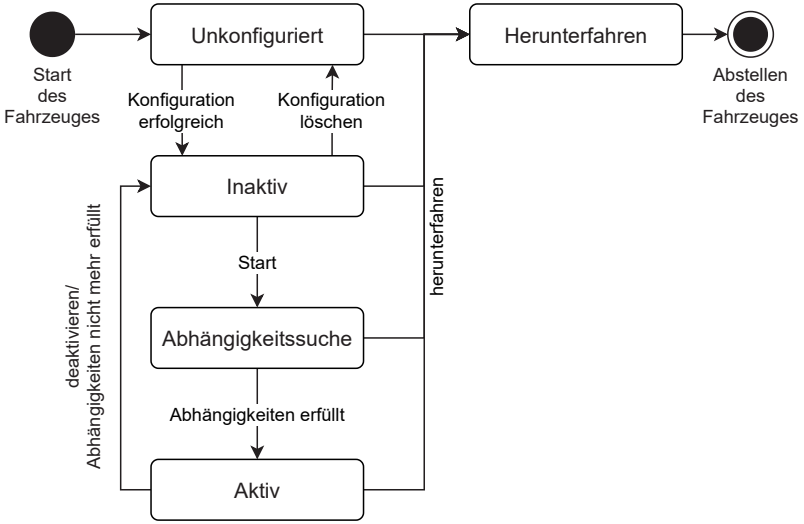


Abbildung 4.6: Zustände eines Managed Nodes [SGS+21]

eigene Klasse, die auf einem definierten Zustandsinterface basiert. So ist sichergestellt, dass neue Zustände falls erforderlich hinzugefügt werden können, die Schnittstellen aber weiterhin kompatibel bleiben.

Ein Timer sorgt dafür, dass die Zustandsmaschine aus den Zuständen der Managed Nodes (s. vorheriger Abschnitt) synchron und mit definierter Frequenz über eine Update-Methode ausgeführt wird, die den nächsten Zustandsübergang evaluiert und durchführt. Dabei ist diese Frequenz zum einen von der verfügbaren Rechenleistung, aber auch den Anforderungen des Nodes abhängig.

Das Heartbeat-Signal wird bei jedem Ausführen der Update-Methode des Lifecycle-Managers auf dem Heartbeat-Topic gesendet, solange der Node sich im aktiven Zustand befindet, entspricht daher auch der Zyklendauer der Zustandsmaschine. Dieses Topic hat die in Tabelle 4.1 auf der nächsten Seite dargestellten QoS-Einstellungen (vgl. Tabelle A.5 auf Seite 205), die nicht enthaltenen Parameter wurden auf ihrem Standardwert belassen. Dabei sollen die Heartbeats möglichst zeitnah ankommen und verlieren nach kurzer Zeit ihren Wert.

Bezeichnung	Wert	Bedeutung
History	Keep Last: 0	Nur die neueste Nachricht soll verfügbar sein, daher werden alte Nachrichten verworfen.
Reliability	**Best Effort**	Kurze Latenzen sind wichtiger als zuverlässige Ankunft, Nachrichten werden daher nur einmal versendet.
Durability	Volatile	Nachrichten werden nicht für spätere Abonnenten zwischengespeichert.

Tabelle 4.1: Die aktiven QoS-Policies des Heartbeat-Topics

Anbieten und Finden von Fähigkeiten

Fähigkeiten werden in einem eigenen Plug-and-Play-Topic angeboten. Dieses wird von jedem interessierten Konsumenten von Services abonniert, diese Services speichern anschließend selbst, welche Abhängigkeiten erfüllt werden können (notwendig für den Übergang in den Zustand *Aktiv*). In diesen Services kommt analog zum Lifecycle-Manager dazu ein *Plug-and-Play-Manager* zum Einsatz, der über die am besten passende, zu verwendende Fähigkeit entscheidet. Dabei wird auch überwacht, ob die entsprechenden die Fähigkeiten anbietenden Nodes noch lebendig sind, also Heartbeats senden. Die Nachrichten des Plug-and-Play-Topics müssen im Gegensatz zu den Heartbeats sicher ankommen, dabei werden auch Verzögerungen in Kauf genommen. Die QoS-Einstellungen gestalten sich daher wie in Tabelle 4.2 enthalten.

Bezeichnung	Wert	Bedeutung
History	Keep Last: 0	Nur die neueste Nachricht soll verfügbar sein, daher werden alte Nachrichten verworfen.
Reliability	**Reliable**	Jede Nachricht muss ankommen, ggfs. auch auf Kosten der Laufzeit wegen mehrfachen Sendens.
Durability	Volatile	Nachrichten werden nicht für spätere Abonnenten zwischengespeichert.

Tabelle 4.2: Die aktiven QoS-Policies des Plug-and-Play-Topics

Umschalten zwischen Fähigkeiten

Wichtigste Komponente des hier vorgestellten Plug-and-Play-Konzeptes ist das Umschalten zwischen Fähigkeiten. Zun einen soll immer die Fähigkeit mit den besten Gütekriterien genutzt werden, zum anderen muss aber gegebenenfalls umgeschaltet werden, falls ein Anbieter einer bestimmten Fähigkeit ausfällt. Diese Umschaltung ist Teil der Verwaltung der Lebenszyklen und findet im Zustand *Aktiv* des Service-Konsumenten statt.

In Abschnitt 6.2 auf Seite 179 wird das Umschalten zwischen Fähigkeiten anhand eines Versuchsaufbaus mit mehreren verfügbaren Kamerafähigkeiten unterschiedlicher Güte evaluiert.

4.2.8 Softwareupdates

Sobald eine neue Version einer Softwarekomponente im Rahmen der Software-Wartung verfügbar ist, wird sie vom Hersteller verteilt (s. Abbildung 4.7 auf der nächsten Seite). Dazu stellt der Hersteller die neue Version in seinem Software-Repository im Backend zur Verfügung. Dabei muss entweder für alle Hardwarearchitekturen, die potenziell in den ausgelieferten Fahrzeugen des Herstellers verbaut sind, das Softwareupdate entsprechend kompiliert und vorgehalten werden, oder die Auslieferung erfolgt im Quelltext mit Kompilierung im Zielfahrzeug. Nachteil hiervon ist der Energie- und Zeitbedarf beim Kunden, außerdem erhält dieser möglicherweise Zugriff auf den Sourcecode. Zusätzlich wird die Absicherung erschwert. Allerdings spart dieses Vorgehen Speicherplatz im Software-Repository des Herstellers und erhöht die Flexibilität.

Nachdem das Fahrzeug über die Verfügbarkeit des Updates benachrichtigt wurde, wird im ersten Fall die neue Software direkt in das Software-Repository des Fahrzeuges übertragen, im zweiten durchläuft sie die CI/CD-Pipeline und wird im Laufe dieser in das Software-Repository übertragen.

Um die Software im Fahrzeug auszurollen, wird sie anschließend auf die Ziel-ECUs verteilt und parallel zur alten Version gestartet. Erst wenn dies erfolgreich war, die neue Version also vollständig funktionsfähig ist, wird die alte Version beendet und aus dem Software-Repository des Fahrzeuges gelöscht.

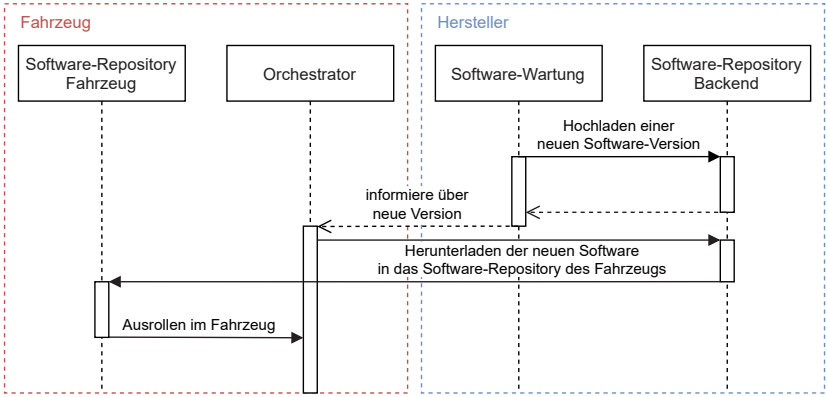


Abbildung 4.7: Vorgehen bei Softwareupdates

4.2.9 Wieso ROS2?

Basierend auf den Erkenntnissen des Vergleichs der SOA in Abschnitt 3.6.3 auf Seite 83 (vergleichende Tabelle A.7 auf Seite 207) wurde ROS2 als zu verwendende SOA und damit DDS als Middleware ausgewählt. Letztere stellt dabei sowohl das *Service Repository* (s. Abschnitt 3.5.3 auf Seite 64) sowie den *Service Bus* (s. Abschnitt 3.5.4 auf Seite 65) zur Verfügung.

ROS2 wurde dabei AUTOSAR Adaptive vorgezogen, da es einige im Rahmen dieser Dissertation relevante Vorteile besitzt. Die standardmäßig von ROS2 verwendete Middleware DDS ist in diesem Kontext umfangreicher und leistungsfähiger als SOME/IP von AUTOSAR Adaptive, aber vor allem flexibler. So erfolgt die Entdeckung von Services vollständig dezentral und dynamisch, außerdem ist keine Anpassung von Konsumenten an spezifische Service-Versionen notwendig.

Zusätzlich spielen andere Aspekte wie Verfügbarkeit und Kosten eine Rolle. Während AUTOSAR Adaptive an sich nur einen Standard darstellt, von welchem entweder kostenpflichtige Implementierungen eingekauft werden müssen oder alternativ eine vollständige Selbstentwicklung durchgeführt werden müsste, ist ROS2 und viele Zusatzkomponenten frei als bestehender Code verfügbar. Dabei kann es im Gegensatz zu AUTOSAR Adaptive auch ohne kostenpflichtige Lizenz in eigenen Produkten verwendet werden.

Nachteile von ROS2 wie die fehlenden Möglichkeiten zur Ressourcenbeschränkung werden durch den Orchestrator (s. Abschnitt 4.2.2 auf Seite 104) ausgeglichen.

4.3 Use-Cases

Um eine mögliche Lösung der Herausforderungen aus dem Motivationskapitel (s. Abschnitt 1.2 auf Seite 1) und die Erfüllung der Anforderungen an eine (re-) konfigurierbare Architektur (s. Abschnitt 1.6 auf Seite 8) zu zeigen, werden exemplarische Use-Cases untersucht und prototypisch umgesetzt:

UC1 Die Erweiterung einer bestehenden, signalbasierten Architektur (s. Abschnitt 4.4 auf Seite 121).

UC2 Der Austausch von Komponenten (s. Abschnitt 4.5 auf Seite 122).

UC3 Die Erweiterung der Funktionalität (s. auch Abschnitt 4.5 auf Seite 122).

UC4 Die Auslagerung von Services ins Backend (s. Abschnitt 4.6 auf Seite 123).

4.3.1 Verortung der Use-Cases im Lebenszyklus der Fahrzeuge

Die Sortierung der Use-Cases erfolgt dabei chronologisch, entsprechend der sich aus Tabelle A.3 auf Seite 204 ergebenden und in Abbildung 4.8 auf der nächsten Seite dargestellten Zuordnung der Use-Cases zu den Lebenszyklusphasen der Fahrzeuge. Die Use-Cases UC2 und UC3 stellen ein besonderes Alleinstellungsmerkmal dar und befinden sich daher im Fokus dieser Dissertation.

Dabei betrifft die Anpassung der Architektur beziehungsweise der Umstieg auf eine (re-)konfigurierbare Fahrzeugarchitektur (UC1) alle Fahrzeuge, da sie im Rahmen der Entwicklung einer neuen Baureihe stattfindet.

Der Austausch von Komponenten (UC2) oder das Hinzufügen von Funktionalität (UC3) können hingegen sowohl während der Produktion oder in der Phase Aftersales stattfinden. Ersteres, wenn aufgrund von mangelhafter Ver-

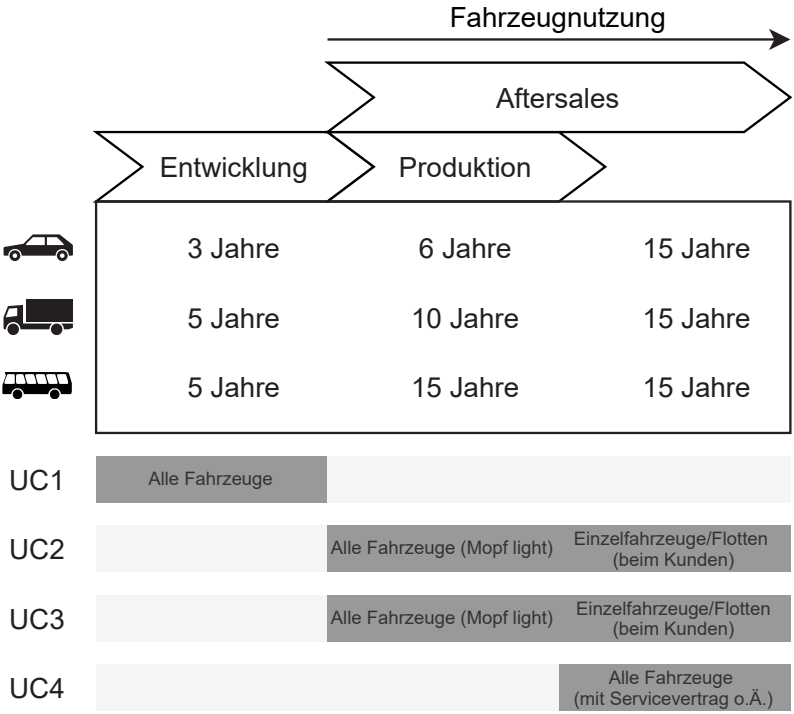


Abbildung 4.8: Verortung der Use-Cases in den Lebenszyklen der Fahrzeuge

fügbarkeit oder geänderter gesetzlicher Rahmenbedingungen eine spontane Anpassung der in Fertigung befindlichen Baureihe (eine Art „Modellpflege (Mopf) light“) vor Auslieferung erforderlich ist. Letzteres, wenn Defekte an Kundenfahrzeugen auftreten oder zusätzliche Funktionalitäten vom Kunden hinzugebucht werden. Daher betreffen Änderungen in der letzten Phase üblicherweise nur einzelne Fahrzeuge oder Flotten, allerdings ist auch die Umsetzung von Rückrufaktionen denkbar.

Die Auslagerung von Services ins Backend (UC4) oder die Nutzung von externen Diensten ist hauptsächlich im Rahmen von Serviceverträgen über eine entsprechende Dienstleistung relevant und betrifft daher alle Kundenfahrzeuge mit einem solchen.

4.4 Use-Case 1: Erweiterung einer bestehenden Architektur

Eine bestehende signalbasierte Architektur eines automatisiert fahrenden Fahrzeuges soll auf eine SOA mit ROS2 aktualisiert werden. Damit wird eine vor allem für bestehende Hersteller von Fahrzeugen wichtige Herausforderung dargestellt: Der Umstieg von einer bestehenden klassischen auf eine moderne serviceorientierte Architektur, ohne das gesamte Fahrzeug zeit- und kostenintensiv von Grund auf neu entwickeln zu müssen. Dieser Use-Case positioniert sich also im Entwicklungsprozess des Fahrzeuges und erleichtert die Wiederverwendung bereits vorhandener Prozessartefakte von Vorgängerbaureihen.

Die in ROS umgesetzte Grundfunktionalität des im Use-Case behandelten automatisiert fahrenden Fahrzeuges (basierend auf dem Basisfahrzeug für die Umsetzung der (re-)konfigurierbaren Fahrzeugarchitektur „Nora“, s. Abschnitt 5.1 auf Seite 125) umfasst die **Wahrnehmung** im Sinne von Objekterkennung, Kartenerstellung und Lokalisierung, sowie die **Fahrfunktion**, welche die Pfadplanung und deren Umsetzung über die Aktorik umfasst.

4.4.1 Migration auf die serviceorientierte Architektur

Nicht alle der verwendeten ROS-Nodes sind für ROS2 verfügbar. Eine direkte Migration ist nicht immer sinnvoll möglich, ohne die Softwarekomponenten neu für ROS2 zu erstellen.

Mit dem in Abschnitt 3.5.2 auf Seite 63 eingeführten *Technologie-Gateway* wird eine Lösung hierfür umgesetzt. Dieses dient dazu, nicht SOA-kompatible Komponenten in die SOA zu integrieren, indem diese gekapselt und nach außen hin mit einer serviceorientierten Schnittstelle versehen werden.

Im Rahmen dieser Dissertation wird es daher dazu genutzt, nicht für ROS2 verfügbare Komponenten ohne Neuentwicklung zu integrieren.

Dabei muss für jede einzelne Komponente überprüft werden, ob sie für ROS2 verfügbar beziehungsweise sinnvoll anpassbar ist. Ansonsten muss eine entsprechende Brücke, das Technologie-Gateway integriert werden. In den meis-

ten Fällen ist dies nicht automatisierbar, muss also manuell im Entwicklungsprozess erfolgen. Dennoch ist der Aufwand gegenüber einer vollständigen Neuentwicklung reduziert.

4.5 Use-Cases 2 und 3: Austausch von Komponenten und Erweiterung der Funktionalität

Wie in der Motivation (s. Abschnitt 1.2 auf Seite 1) besprochen, gibt es verschiedene Gründe, eine Komponente während des Lebenszyklusses eines Fahrzeuges zu tauschen beziehungsweise eine neue hinzuzufügen. Dies kann zum Beispiel eine Veränderung der Verfügbarkeit sein, aber auch der Ersatz der Komponente durch einen kostengünstigeren und/oder leistungsfähigeren Nachfolger, womit Kosten gesenkt und die Energieeffizienz gesteigert werden können. Ein anderer Grund ist die Erweiterung der Funktionalität des Fahrzeuges mit Features, die gegebenenfalls höhere Ressourcenanforderungen besitzen oder zusätzliche Hardwarekomponenten benötigen (s. Use-Case 3). Dies kann zum Beispiel eine Erweiterung der Fähigkeiten zum automatisierten Fahren (zum Beispiel von Stufe 3 auf 4/5), aber auch zusätzliche vom Kunden erworbene Funktionalität sein.

Ein solcher Tausch wird also vor allem spät in der Entwicklungsphase oder während der Produktion einer Baureihe geschehen, im Rahmen einer Art „Modellpflege light“. Hierdurch kann schneller auf geänderte rechtliche Rahmenbedingungen reagiert werden. Außerdem kann der Tausch in der Wartungsphase erforderlich sein, wenn identischer Ersatz für defekte Komponenten durch fehlende Verfügbarkeit nicht mehr beschaffbar ist oder Kunden zusätzliche Features dynamisch erwerben möchten anstatt bei Kauf des Fahrzeuges.

Der Austausch von Komponenten ist die Kombination von A) Entfernen und B) Hinzufügen von ECUs und zugehöriger Software, worunter als Spezialisierung auch (intelligente) Sensoren und Aktoren fallen. Soll der Austausch zur Laufzeit stattfinden (Anforderung AP4), muss die Koordination so erfolgen, dass die Funktion nicht unterbrochen wird.

Dies kann auf mehreren Wegen erfolgen: Findet B) vor A) statt, kann vor dem Entfernen auf die neue Komponente umgeschaltet werden und es entsteht keine Ausfallzeit. Wird allerdings zuerst die Komponente entfernt (A) findet

also vor B) statt), entspricht dies dem Ausfall dieser Komponente. Daher muss also auf eine Alternative wie kompatible Services einer redundanten oder ähnlichen Komponente zurückgegriffen werden. Ist dies nicht möglich, lässt sich eine Unterbrechung oder Degradation der Funktion nicht vermeiden.

Daher werden diese beiden Use-Cases im Folgenden gemeinsam behandelt, dazu wird als Beispiel ein intelligenter Sensor an das System angeschlossen, welcher sich entsprechend anmeldet und die zum Betrieb notwendige Software ausrollt. Anschließend kündigt er seine Fähigkeiten im Fahrzeug an. Außerdem wird ein Ausfall dieses neuen Sensors und das nahtlose Umschalten auf einen Ersatz dargestellt.

4.6 Use-Case 4: Auslagern von Services ins Backend

Das Auslagern von Services ins Backend kann aus verschiedenen Gründen wünschenswert sein. Zum Beispiel kann so die erforderliche Rechenleistung und Speicherkapazität im Fahrzeug verringert werden, indem rechenleistungsintensive Operationen dezentral ausgeführt werden. Dies lohnt sich besonders bei Berechnungen, die sonst jedes einzelne Fahrzeug selbst ausführen müsste, die aber insgesamt redundant sind. Als Betreiber einer Flotte können so die Betriebskosten, aber auch die einmaligen Kosten der Fahrzeuge gesenkt werden.

Über die finanziellen Aspekte hinaus können sich noch andere Vorteile ergeben: So können Sensoren in der Umgebung tote Winkel der Fahrzeuge beziehungsweise den Bereich hinter Hindernissen erfassen und Listen mit den erkannten Objekten an die Fahrzeuge übertragen.

Inhalt dieses Use-Cases ist es daher, die Anbindung an das Backend und die Möglichkeit zur Auslagerung von Softwarekomponenten darzustellen. Dazu wird eine sichere Verbindung zwischen Fahrzeug und Backend aufgebaut und somit das Backend mit der Infrastruktur des Fahrzeuges verknüpft. Anschließend funktioniert das Hinzufügen und Entfernen von Softwarekomponenten so, als wäre das Backend Teil des Fahrzeugnetzwerkes, dazu gehört auch die Installation von im Backend bereitgestellten Softwareupdates.

5 Prototypische Umsetzung der identifizierten Use-Cases

5.1 Das zugrundeliegende Fahrzeug: Nora

Das Basisfahrzeug für die Untersuchung des Konzepts der (re-)konfigurierbaren Architektur (s. Kapitel 4 auf Seite 99) und dem darauf aufbauenden Use-Case 1: *Erweiterung einer bestehenden Architektur*, bei dem eine bestehende Architektur eines automatisiert fahrenden Fahrzeuges auf eine (re-)konfigurierbare aktualisiert wird (s. Abschnitt 4.4 auf Seite 121), wurde basierend auf einem elektrisch angetriebenen Aufsitzrasenmäher der Firma ETESIA umgesetzt (s. Abbildung 5.1, dort schon mit dem 19"-Schrank für die hinzuzufügende Hardware). Da jedoch der Fokus auf der (re-)konfigurierbaren Architektur liegt, wurde das entstandene Fahrzeug auf den Namen „Nora" getauft, die Kurzform von „**no**t a **Ra**senmäher".

Abbildung 5.1: Der als Basis verwendete elektrisch angetriebene Aufsitzrasenmäher der Firma ETESIA

Die ursprüngliche E/E-Architektur enthält drei Steuergeräte, die untereinander per CAN-Bus verbunden sind (s. Abbildung 5.2): Eine Anzeigeeinheit, die Betriebsstunden, Ladestand und Fehlercodes anzeigen kann, sowie je eine ECU für die Regelung von Mähwerk und Antriebsmotor. Zu diesem Zweck werden jeweils Motortemperatur und -drehzahl an das zugehörige Steuergerät zurückgemeldet.

Eine Sitzbelegungserkennung verhindert, dass das Fahrzeug ohne Fahrer fährt oder mäht, dieser kann außerdem über Brems- und Gaspedal sowie einen Tempomattaster das Fahrzeug in Längsrichtung steuern, die Lenkung ist rein manuell ohne jegliche Lenkunterstützung.

Die Boardspannung beträgt durch vier in Reihe geschaltete 100 Ah-Blei-Säure-Batterien 48 V. Diese Kapazität reicht für bis zu 8 h Fahrbetrieb mit einer Höchstgeschwindigkeit von 8 km/h oder maximal 2 h Mähbetrieb mit auf 5 km/h gedrosselter Geschwindigkeit aus. Für den Betrieb weiterer Geräte stellt außerdem ein Inverter eine 230 V-Steckdose mit einer Maximalleistung von 700 W zur Verfügung.

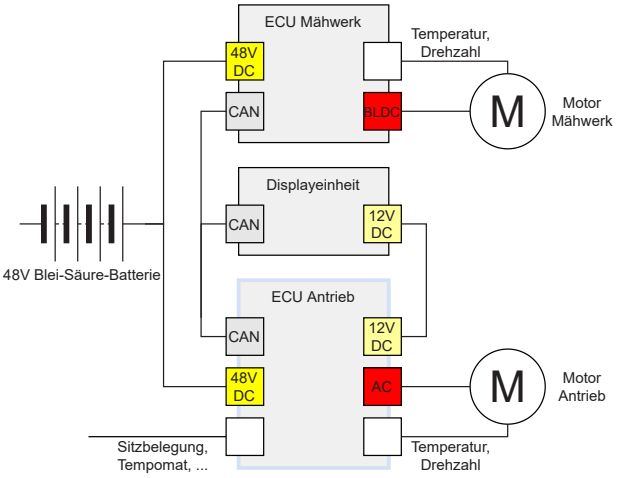


Abbildung 5.2: Die ursprüngliche E/E-Architektur von Nora

Regelung des Antriebsmotors im Ursprungszustand

Das Steuergerät für die Regelung des Antriebsmotors nimmt den Drehmomentwunsch des Fahrers an und setzt diesen bis zu einer fest definierten Maximalgeschwindigkeit um (s. Abbildung 5.3). Aus Sicherheitsgründen wird der Motor gestoppt, wenn der Fahrer den Fahrersitz verlässt, zu lange das Beschleunigungspedal nicht bedient (beziehungsweise dieses ausfällt) oder der Motor überhitzt. Um diese Aufgaben zu erfüllen, erhält die ECU verschiedene Eingangssignale, die nicht alle abgebildet sind: Die Drehzahl am Getriebe, die Temperatur des Motors, Batteriespannung, Beschleunigungsbeziehungsweise Bremspedalstellung sowie Sitzbelegung.

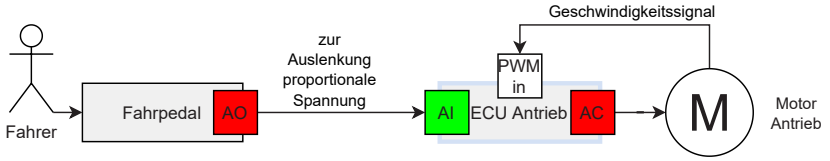


Abbildung 5.3: Geschwindigkeitsregelung im Ursprungszustand, AO: Analogausgang, AI: Analogeingang

5.1.1 Hinzugefügte Steuergeräte für die automatisierte Fahrfunktion

Die Fahrfunktion setzt die Fahraufgabe um, diese besteht bei einem automatisierten Fahrzeug darin, die von der Wahrnehmung gewonnenen Daten wie Karte, Position und Objekte zu nutzen, um die gewünschte Position durch Ansteuerung der Aktorik zu erreichen. Dabei müssen erkannte Hindernisse vermieden werden, entweder durch Abwarten oder Umsetzung einer Ausweichtrajektorie.

Bereits auf Nora vorhandene Steuergeräte für die Grundfunktionen, wie die Regelung des Antriebsmotor und das HMI[1] mit Batteriestands- und Laufzeitanzeige (s. Abbildung 5.2 auf der vorherigen Seite) wurden, bis auf das

[1] Human Machine Interface

deaktivierte Mähwerk, beibehalten. Dabei wurde der bereits vorhandene CAN-Bus weitergenutzt. Zusätzlich wurden ECUs zur Geschwindigkeitsregelung und Regelung des Lenkwinkels hinzugefügt, die die Grundlage zur Automatisierung der Fahrfunktion bilden, indem sie deren Befehle umsetzen. Im Falle der Regelung des Lenkwinkels wurden außerdem zusätzliche Sensoren und Aktoren integriert, da das Fahrzeug auf eine manuelle Lenkung ausgelegt war.

Die resultierende Struktur ist in Abbildung 5.4 dargestellt, dabei sind die neu hinzugekommenen Komponenten gestrichelt umrandet.

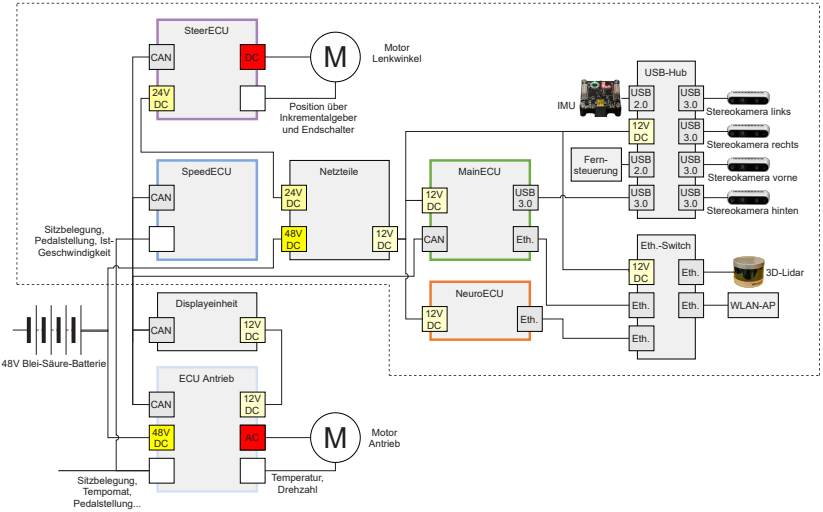


Abbildung 5.4: Die E/E-Architektur von Nora mit ROS1

Geschwindigkeitsregelung – SpeedECU

Die im Ursprungszustand des Fahrzeuges vorhandene manuelle Geschwindigkeitsregelung wurde in Abschnitt 5.1 auf der vorherigen Seite beschrieben. Diese wird so modifiziert, dass der Fahrer durch eine Fahrautomatisierung ersetzt werden kann.

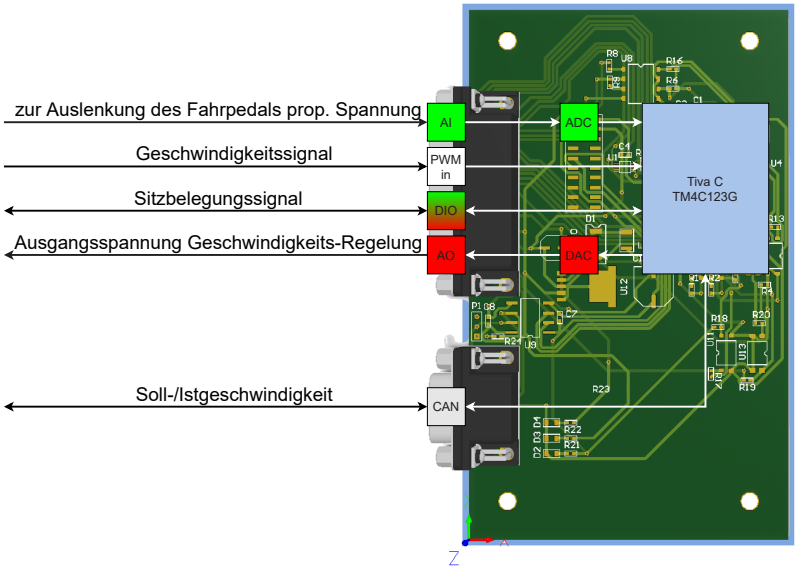


Abbildung 5.5: Blockschaltbild der Geschwindigkeitsregelung mit eigener ECU – *SpeedECU*

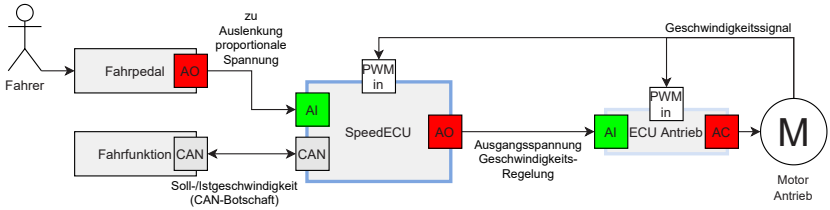


Abbildung 5.6: Modifizierte Geschwindigkeitsregelung, AO: Analogausgang, AI: Analogeingang

Ein neu zur Geschwindigkeitsregelung hinzugefügtes Steuergerät (*SpeedECU*, s. Abbildung 5.5), übernimmt, soweit gewünscht, die Rolle des Fahrers (für die Einbindung in das Gesamtsystem der Geschwindigkeitsregelung des Fahrzeuges, s. Abbildung 5.6). Dazu besitzt es die Fähigkeit, zwischen den Eingaben eines Fahrers und per CAN-Botschaft empfangenen Geschwindigkeitsanforderungen umzuschalten. Die Eingaben das Fahrers erfolgen dabei zum einen über eine in Beschleunigungs- und Bremspedal unterteilte Pedalerie, die eine

zur Pedalstellung proportionale Spannung zwischen 0 und 5V ausgibt und außerdem über zwei Digitalpins mitteilt, welches der Pedale betätigt ist. Eine gleichzeitige Betätigung wird durch eine mechanische Sperre verhindert. Zum anderen ist in den Sitz ein Sensor integriert, der bei Belegung eine Spannung ausgibt. Eine Möglichkeit zur automatischen Umschaltung ist die Erkennung der Sitzbelegung, eine weitere das dauerhafte Auslesen der Pedalstellung über den in der SpeedECU integrierten ADC und Überschreiben der per CAN empfangenen Anforderung. Diese Umschaltung kann in der Software konfiguriert werden und ist im Normalfall deaktiviert.

Dadurch, dass die Pedalerie vom ursprünglich vorhandenen Steuergerät abgetrennt wurde, kann das Steuergerät zur Geschwindigkeitsregelung das Verhalten bei manueller Fahrt beeinflussen, zum Beispiel durch einen Proportionalitätsfaktor oder eine einstellbare Begrenzung der Maximalgeschwindigkeit.

Unabhängig von der Quelle der Geschwindigkeitsanforderung wird am Schluss mittels ebenfalls in die SpeedECU integriertem DAC[2] eine „virtuelle“ Pedalstellung an das Motorsteuergerät zurückgegeben, entsprechend auch ein Sitzbelegungssignal.

Kern der SpeedECU ist das *EK-TM4C123GXL Tiva C Launchpad*, ein Evaluationsboard von Texas Instruments mit ARM Cortex-M4F CPU, auf dem ein PI-Regler zur Regelung der Geschwindigkeit und die CAN-Kommunikation läuft. Letztere schließt dabei auch eine Ausgabe der über den im Getriebe befindlichen Hall-Effekt-Drehzahlsensor (s. *Magnetische Abtastung* in Abschnitt 2.2.2 auf Seite 24) gemessenen Ist-Geschwindigkeit als CAN-Botschaft ein.

Diese wird anschließend von der Haupt-ECU als ROS-Message weitergeleitet. Damit steht die Ist-Geschwindigkeit auch anderen ECUs zur Verfügung und kann zum Beispiel als Eingangssignal für die Lokalisierung genutzt werden.

[2] Digital-Analog-Wandler

Regelung des Lenkwinkels – SteerECU

Anders als für der Geschwindigkeitsregelung ist für die Regelung des Lenkwinkels im Werkszustand keinerlei Aktorik oder Sensorik vorhanden. Diese musste daher hinzugefügt werden.

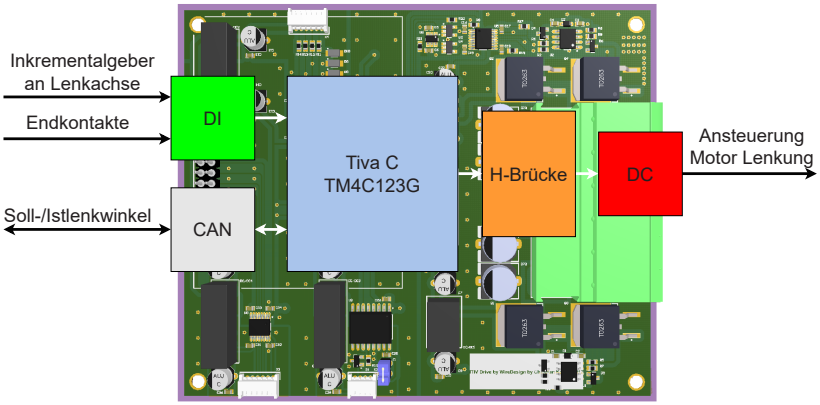


Abbildung 5.7: Blockschaltbild der Regelung des Lenkwinkels – *SteerECU*

Das neu integrierte Steuergerät ist in Abbildung 5.7 schematisch abgebildet. Beim Start des Systems werden die Endpositionen angefahren, dazu treibt der DC-Motor an der Lenkachse diese solange an, bis ein Endanschlag erreicht ist und der entsprechende Endschalter dies registriert. Entsprechend wird für den anderen Endanschlag verfahren. Dabei werden die von einem an der Lenkachse montierten Inkrementalgeber (s. *Optische Abtastung* in Abschnitt 2.2.2 auf Seite 23) ausgegebenen Pulse gezählt, über den maximalen Lenkachsenwinkel kann somit der aktuelle Zählerstand dem Ist-Lenkachsenwinkel zugeordnet werden. Da eine Art Ackermann-Lenkung zum Einsatz kommt, ist die tatsächliche Stellung der Vorderräder unterschiedlich; das kurveninnere Rad wird stärker eingeschlagen. Über einen Proportionalitätsfaktor wird der tatsächliche Lenkwinkel des Fahrzeuges, der aus dieser Geometrie resultiert, berechnet. Ein PI-Regler regelt nun über ein über die H-Brücke ausgegebenes pulsweitenmoduliertes Signal die Ist-Position auf den per CAN-Botschaft empfangenen Sollwert. Zum Beispiel zur Verbesserung der Lokalisierung über die Odometrie wird zusätzlich zur Ist-Geschwindigkeit auch der Ist-Lenkwinkel als CAN-Botschaft an das System weitergegeben.

Steuergeräte für ROS

Die Intelligenz für das *automatisierte* Fahren wie Wahrnehmung und Navigation wurde auf Basis des Robot Operating Systems *ROS* (s. Abschnitt 3.3.3 auf Seite 56) umgesetzt. Dieses wird auf Standardhardware aus dem PC-Bereich ausgeführt (Struktur s. Abbildung 5.4 auf Seite 128). Im Gegensatz zur direkten Ansteuerung der Sensorik und Aktorik zur Regelung von Geschwindigkeit und Lenkwinkel im Ursprungszustand ergeben sich durch den Umbau mit den hinzugefügten Features wesentlich höhere Anforderungen an die Rechenleistung. Konkret ist das als Haupt-ECU, *MainECU* genannt, ein Mini-PC von Intel, der NUC7i7DNHE, der das Grundsystem und die meisten weiteren Funktionen ausführt, und als sogenannte *NeuroECU* ein Nvidia Jetson TX2, das durch die höhere Grafikleistung für die Berechnung neuronaler Netze zum Einsatz kommt, wie zum Beispiel bei der Objekterkennung im Rahmen der Wahrnehmung (s. Abschnitt 5.1.3 auf Seite 136).

Haupt-ECU – MainECU Ausgestattet mit einem Intel Core i7-8650U und 16GB RAM erreicht der NUC eine Performanz im Bereich eines Mittelklasse-PCs, ist dabei jedoch deutlich kompakter und sparsamer. Zusätzlich zu einer 256GB NVMe-SSD, auf der ROS Melodic Morenia auf Ubuntu 18.04 installiert ist, dient eine 2TB HDD zur Aufzeichnung von Daten, zum Beispiel in Form von rosbags. Zur Kommunikation mit den anderen ECUs kommt zum einen ein opto-entkoppelter PEAK PCAN-USB-Adapter zum Einsatz, der mit dem Fahrzeug-CAN-Bus verbunden ist und außerdem Ethernet im Falle der NeuroECU.

NeuroECU Das Jetson TX2 ist mit einem sogenannten Tegra „Parker“ ausgestattet. Dieser besteht aus insgesamt sechs CPU-Kernen, bestehend aus einem ARM Cortex A57 Quadcore und zwei NVIDIA Denver2 Kernen. Ausgestattet mit 8GB RAM und 32GB Flash läuft auf diesem Board ebenso Ubuntu 16.04 mit ROS Kinetic Kame. Vorhandene ROS-Nodes von der Haupt-ECU laufen auch hier, müssen jedoch wegen der unterschiedlichen Prozessorarchitekturen neu kompiliert werden. Während das Jetson trotz der höheren Anzahl an Prozessorkernen bei der CPU-Leistung unterlegen ist, ist die Grafikleistung durch die integrierte 256-Kern-Pascal-GPU deutlich höher, damit auch die Leistung bei der Ausführung neuronaler Netze.

5.1.2 Allgemeine Funktionalität

Insgesamt kommen ca. 30 ROS-Nodes zum Einsatz. Einige stellen dabei Informationen zur Verfügung, die von anderen genutzt werden, zum Beispiel werden Sensor-Positionen und Fahrzeugabmessungen beim Start des Fahrzeuges aus dem Fahrzeugmodell eingelesen und zentral zur Verfügung gestellt.

Da dieses Fahrzeugmodell auch Massen und andere Eigenschaften der Komponenten wie Geometrie der Lenkung enthält, kann es auch zur Simulation des Fahrzeuges genutzt werden.

Ein RC-Node empfängt und sendet Daten über Funk („Fernsteuerung“ in Abbildung 5.4 auf Seite 128) und setzt diese in ROS-Messages um. Damit kann das Fahrzeug ferngesteuert und zum Beispiel das Datenlogging gestartet, aber auch Informationen auf einer Fernsteuerung angezeigt werden. Eine alternative Möglichkeit hierfür ist das integrierte Web-Interface.

5.1.3 Wahrnehmung

Die Wahrnehmung umfasst Erfassung und Interpretation der Umgebung des Fahrzeuges und besteht aus mehreren Teilen. Zum einen sind Sensoren erforderlich, die Daten über die Umgebung liefern, zum anderen sind diese Daten ohne weitere Verarbeitung nutzlos. Teil dieser Verarbeitung sind die Objekterkennung, aber auch die Lokalisierung des Fahrzeuges und die Kartenerstellung.

Sensorik

Zur Erfassung seiner Umwelt besitzt das Fahrzeug eine Reihe verschiedener Sensortypen. Bereits besprochen (s. Abschnitt 5.1.1 auf Seite 127, *SpeedECU* und *SteerECU*) wurden die über CAN angebundenen Sensoren für z.B. Geschwindigkeit und Lenkwinkel, weitere sind zum Beispiel optische Sensoren, die eine Punktwolke der Umgebung in 3D (Lidar, 3D-Kamera) oder ein einfaches 2D-Bild (auch 3D-Kamera) liefern, s. Abbildung 5.8. Die einzelnen Typen werden im Detail in Abschnitt 2.2 auf Seite 20 beschrieben. Für die konkrete absolute Position gibt es mehrere Quellen, zur Unterscheidung werden hier

die Begriffe „UWB“ für eine Möglichkeit zur Indoor-Lokalisierung und „GNSS“ für eine Outdoorlokalisierung ohne Karte genutzt.

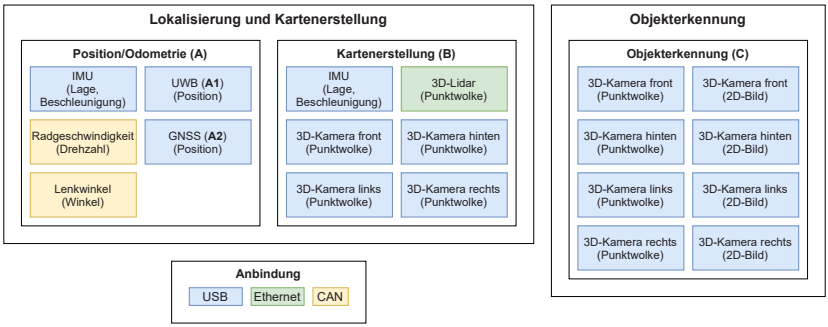


Abbildung 5.8: Quellen der Sensordaten, die Formate und ihre Anbindung an das System

GNSS-unabhängige Lokalisierung mittels UWB (A1)

Für Einsatzgebiete, in denen kein GNSS-Empfang verfügbar ist, zum Beispiel innerhalb von Gebäuden oder auf abgeschatteten Betriebshöfen, kommt eine alternative Lokalisierungslösung zum Einsatz. Eine Beschreibung im Detail sprengt den Rahmen dieser Dissertation und wurde in [SZHS17] veröffentlicht.

Um die Position zu schätzen, besitzt das Fahrzeug einen UWB-Empfänger, der über eine Laufzeitmessung des Signals von in der Umgebung positionierten UWB-Ankern die Entfernung zu diesen bestimmt (s. Abbildung 5.9 auf der nächsten Seite). Zusätzlich zur Triangulation der Position aus den Entfernungen werden die Geschwindigkeit des Fahrzeuges und seine Drehrate genutzt. Dazu kommt ein Extended Kalman-Filter zum Einsatz, das basierend auf den Eingangssignalen und einem geeigneten Fahrzeugmodell die tatsächliche Position schätzt.

Drei Anker genügen bereits, um die Position zuverlässig bestimmen zu können, eine Erhöhung der Anzahl erhöht die Genauigkeit geringfügig weiter. Werden sieben Anker eingesetzt, beträgt der Median des Fehlers 17 cm, der Mittelwert 23 cm und die Standardabweichung 18 cm. Insgesamt liegt der Fehler immer unter 1 m und die Position ist damit genauer bestimmbar als über GPS ohne

RTK oder DGPS. Mit diesen Korrekturdaten lässt sich jedoch, Empfang eines Satellitensignals vorausgesetzt, mit GNSS die Position wesentlich genauer bestimmen (s. nächster Abschnitt).

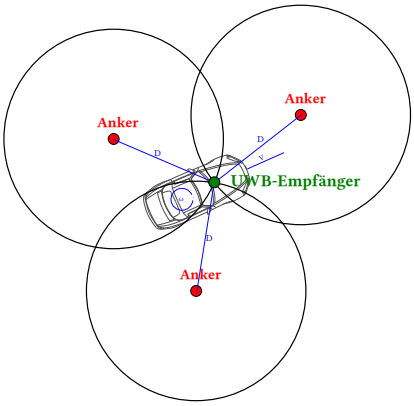


Abbildung 5.9: Aufbau der Lokalisierung mittels UWB

Lokalisierung über GNSS (A2)

Für die Lokalisierung über GNSS kommt ein System zum Einsatz, das gleichzeitig DGPS und GPS RTK, sowie alle weiteren gängigen GNSS-Verfahren wie Galileo, GLONASS und BeiDou unterstützt. Dazu ist eine Basisstation (*base*) fest auf dem Dach eines Gebäudes montiert und übermittelt Korrekturdaten per Funk an ein Modul (*Rover*) auf dem Fahrzeug. Hierdurch wird im Außenbereich eine Lokalisierungsgenauigkeit im niedrigen einstelligen Zentimeterbereich erreicht, womit eine exaktere Position als mit dem UWB-System (s. vorheriger Abschnitt) bestimmt werden kann.

Ein ROS-Node übernimmt die Ansteuerung und Konfiguration des GNSS-Moduls, ein weiterer gibt die Positionsdaten an das Fahrzeug weiter.

SLAM: Kartenerstellung mit gleichzeitiger Lokalisierung (B)

Zur Navigation wird außer der Position auch eine Karte benötigt, der Google Cartographer [HKRA] erstellt die 3D-Karte. Dazu werden die Punktwolken von 3D-Kameras und Lidar fusioniert und die z-Achse über die Lage der von der IMU gemessenen Erdbeschleunigung bestimmt. Da der SLAM-Algorithmus nur die Lage in der selbst erstellten Karte schätzen kann, werden GNSS-Daten genutzt, um die absolute Position zu bestimmen.

Aus der vom Cartographer-Node gelieferten 3D-Karte erstellt ein weiterer Node eine 2D-Karte, die als Basis für die Navigation und Hindernisvermeidung dient, indem die Objekterkennung (s. nächster Abschnitt) in diese die erkannten Hindernisse einfügt.

Objekterkennung (C)

Um die Hindernisse in der Umgebung des Fahrzeuges zu erkennen, wurde eine Objekterkennungs-Node in ROS erstellt.

Details wie den Vergleich der verschiedenen zur Auswahl stehenden Netze und die Kalibrierung der Sensoren wurden in [SS21] veröffentlicht.

Der Objekterkennungs-Node übernimmt die Objekterkennung über 2D-Bildinformationen und die Übertragung in die Karte mithilfe weiterer 3D-Informationen von Kamera und Lidar. Außerdem wird eine Segmentierung vorgenommen, sprich die Unterteilung der Umgebung in verschiedene Bereiche, zum Beispiel befahrbar und nicht befahrbar. Für Objekterkennung und Segmentierung werden künstliche neuronale Netze eingesetzt (s. Abschnitt 2.3 auf Seite 30).

In Abbildung 5.10 auf der nächsten Seite sind die Ergebnisse beispielhaft dargestellt. Die Struktur entspricht dabei Abbildung 5.11 auf der nächsten Seite: Aufgrund der hohen Anforderungen an die Grafikleistung wird die eigentliche Objekterkennung durch das neuronale Netz an die NeuroECU ausgelagert, welche hier leistungsfähiger als die MainECU ist. Auf dieser wird anschließend die Segmentierung in 3D durchgeführt und die Objekte als zusätzliche Hindernisebene in die Karte der Navigation eingefügt. Hierdurch werden

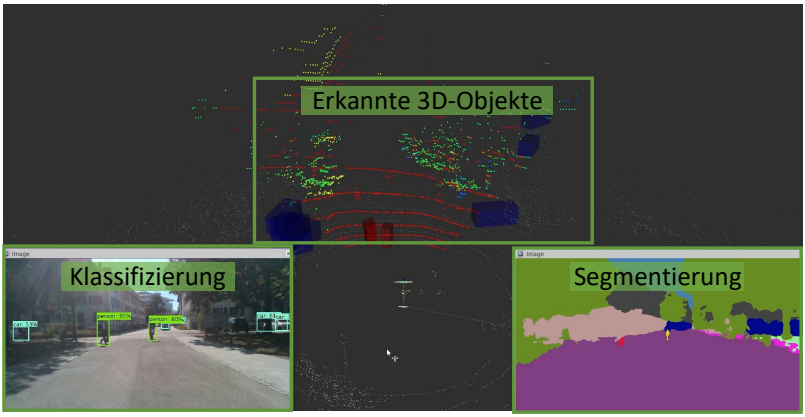


Abbildung 5.10: Vor dem Institut aufgenommenes Beispielbild zur Begriffserklärung

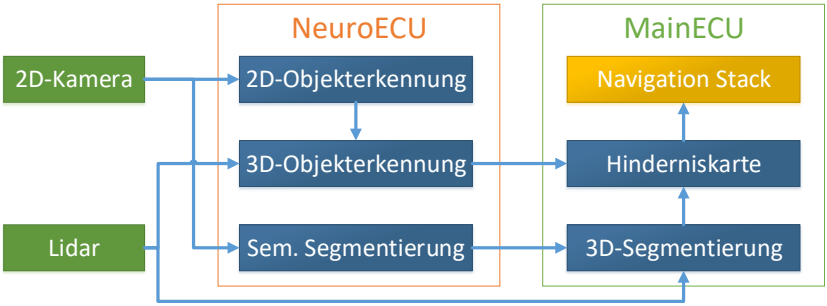


Abbildung 5.11: Übersicht über die Objekterkennung

erkannte Hindernisse vermieden und befahrbare Bereiche zur Routenplanung genutzt.

Erkennung und Segmentierung Zur Erkennung von Objekten und zur Segmentierung der Bilddaten wurde ein Objekterkennungs-Node in ROS erstellt (s. Abbildung 5.12 auf der nächsten Seite). Dieser führt die entsprechenden neuronalen Netze mittels TensorFlow (s. Abschnitt 2.3.1 auf Seite 32) aus. Dabei ist der Node so konfigurierbar, dass zum Beispiel eine Segmentierung

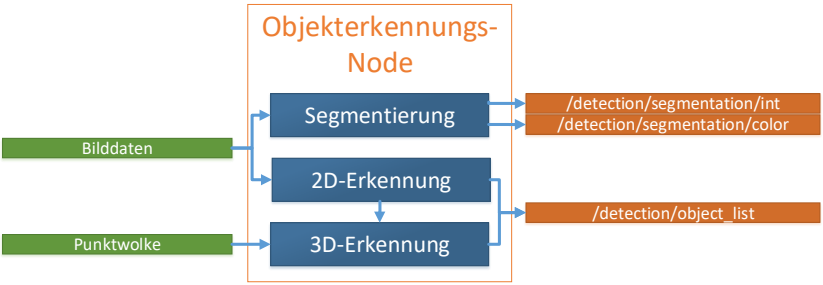


Abbildung 5.12: Objekterkennungs-Node in ROS

ausgeführt und die Erstellung der Bounding Boxes deaktiviert werden kann. Einzige Einschränkung ist, dass für die Objekterkennung in 3D auch die 2D-Objekterkennung erforderlich ist.

Zur Untersuchung von 2D- und 3D-Objekterkennung wurde der KITTI-Datensatz (für eine Übersicht über verfügbare Datensätze, s. Anhang A.2.1 auf Seite 191) eingesetzt, entsprechend wurde die Benchmarkliste von KITTI ([GLU12], [APCR18]) zur Auswahl der passenden neuronalen Netze verwendet. Die für die **3D-Objekterkennung** zu diesem Zeitpunkt aussichtsreichsten Kandidaten (unter Berücksichtigung von Parametern wie benötigte Rechenleistung, Verfügbarkeit und Genauigkeit) waren Aggregate View Object Detection (AVOD) [KML+17] und Frustum-PointNets [QLW+17]. Ausgewählt wurde Frustum-PointNets, da sich das Netz zur 2D-Erkennung gegen ein anderers austauschen lässt, was aufgrund der geringen Performanz der NeuroECU notwendig ist.

Für die **2D-Objekterkennung** ist TensorFlow (s. Abschnitt 2.3.1 auf Seite 32) als Framework vorgesetzt, da bereits der verwendete 3D-Detektor von Frustum-PointNets darauf setzt. Über die dokumentierte API von TensorFlow ist der Einsatz und Austauschbarkeit einer Vielzahl verschiedener Netzarchitekturen möglich. Aufgrund der zuvor angesprochenen vergleichsweise geringen Rechenleistung der NeuroECU, ist die einzige hier praktikable Meta-Architektur ein Single-Shot-Detektor (SSD).

Die Auswahl des Netzes für die **Segmentierung** wurde mit dem Cityscapes-Datensatz durchgeführt, da KITTI keine vorsegmentierten Daten enthält.

ICNet wurde aufgrund des guten Verhältnisses von Genauigkeit zu Leistungsanforderungen ausgewählt.

Kommunikation des Knotens mit ROS Unabhängig vom restlichen Knoten läuft in einem eigenen Thread das Empfangen der Punktwolken: über die ROS-Funktion *ApproximateTimeSynchronizer* wird sichergestellt, dass Bilddaten von den 2D-Kameras sowie Punktwolken von 3D-Kamera und Lidar zeitsynchronisiert vorliegen. Dies ist wichtig für eine korrekte Funktion der Objekterkennung. Für die Objektliste wurde eine neue ROS-Nachricht erstellt, die jeweils eine Liste der 2D- und 3D-Objekte enthält (s. Listing 2). Die erste

```
1 std_msgs/Header header
2 Object2D[] object_2d_list
3 Object3D[] object_3d_list
```

Listing 2: ObjectList.msg

Zeile enthält dabei den standardmäßigen Nachrichtenkopf von ROS [ros19], jede Nachricht erhält dabei eine um eins erhöhte ID, einen Zeitstempel und das Frame der Nachricht (zum Beispiel die Position des sendenden Sensors).

Ein 2D-Objekt *Object2D* sieht wie in Listing 3 dargestellt aus und wird aus den von TensorFlow gelieferten Daten erstellt. Über x- und y-Werte werden

```
1 float32 xmin
2 float32 xmax
3 float32 ymin
4 float32 ymax
5 float32 type
6 float32 score
```

Listing 3: Object2D.msg

hierbei die Koordinaten der Bounding-Box des einzelnen Objekts angegeben. Zu beachten ist, dass die Positionen auf die Bildgröße normiert sind, daher im Bereich von 0 bis 1 liegen. Die Objektklasse wird als Zahl in *type* gespeichert. Die Zuordnung ist abhängig von trainiertem Modell und Datensatz und wird entsprechend von TensorFlow bereitgestellt. Die Wahrscheinlichkeit des Objektes wird durch *score* angegeben. Damit kann eine Schwelle vorgegeben

werden, wodurch nur Objekte ab einer bestimmten Wahrscheinlichkeit der Liste hinzugefügt werden.

Wie angesprochen wurde für die 3D-Objekterkennung Frustum-PointNets genutzt, hierzu muss zu jeder Bounding Box aus der 2D-Objektliste mit einer Wahrscheinlichkeit von über 50% ein entsprechender Pyramidenstumpf aus der Punktwolke ausgeschnitten werden, was recht CPU-intensiv ist. Mit der Ausgabe des Netzwerkes hierzu lässt sich die Liste mit 3D-Objekten erstellen:

Die bereits vorhandene 2D-Grundfläche der Bounding-Box wird erhalten, zusätzlich hinzugefügt werden Höhe h, Breite w und Länge l, sowie Mittelpunkt $\vec{t} = \begin{bmatrix} t_x & t_y & t_z \end{bmatrix}^T$ und Ausrichtungswinkel r_y. Hier ist nur der Gierwinkel relevant, da Roll- und Nickwinkel meist 0 sind. Außerdem wird die Objektklasse direkt als String eingetragen. Das Resultat ist in Listing 4 dargestellt.

```
float32 xmin
float32 xmax
float32 ymin
float32 ymax
string type
float32 score

float32 h
float32 w
float32 l
float32 tx
float32 ty
float32 tz
float32 ry
```

Listing 4: Object3D.msg

Als letzte Aufgabe übernimmt der Objekterkennungsnode die Segmentierung der Bilddaten mittels ICNet. Dabei wird ein Bildarray mit Abmessungen, die ein Vielfaches von 32 Pixeln sind, in das Netz gespeist. Heraus kommt ein Bildarray, das als zusätzlichen Kanal die Objektklassen enthält, außerdem wird bereits ein Bildarray erstellt, in dem die Originalfarben durch die erkannten Objektklassen ersetzt werden. Die weitere Verarbeitung wie 3D-Segmentierung übernimmt der Visualisierungs-Node.

Visualisierung der Ergebnisse Der Visualisierungs-Node (s. Abbildung 5.13) wird aufgrund seiner hohen Anforderungen an die CPU-Leistung anders als der Objekterkennungs-Node auf der MainECU ausgeführt. Zusätzlich zu den Objekten, die er vom Objekterkennungs-Node erhält, empfängt der Knoten Nachrichten im Topic */tf*. Dieses enthält Informationen zur Lage der Datenquellen untereinander. Außerdem werden die Rohdaten von Kamera und Lidar verwendet. Auch hier werden *ApproximateTimeSynchronizer* zur Synchronisierung der Daten eingesetzt. Besonders wichtig ist dies, da der Objekterkennungsnode eine gewisse Zeit zur Verarbeitung der Daten und Erstellung der Objektliste benötigt. Für die Darstellung werden in jedes einzelne Bild die erkannten Bounding Boxes (mit Wahrscheinlichkeit über 50%) eingezeichnet, jede einzelne mit einem Text-Label mit ihrer Klassenbezeichnung sowie Wahrscheinlichkeit versehen. Diese Darstellung ist rein für die Visualisierung für einen Benutzer gedacht und nicht für die Funktion notwendig. Notwendig hingegen ist die Transformation der 3D Bounding Boxes in die Lidar-Koordinaten, weg von relativen Angaben in Pixeln hin zu absoluten Angaben in Metern, hierbei helfen die Informationen des Transformations-Topics. Außerdem für die Darstellung in *Rviz* gedacht ist das Einfügen von

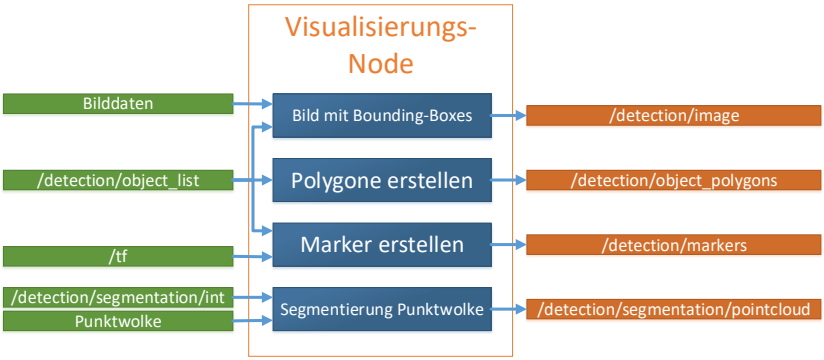


Abbildung 5.13: Visualisierungs-Node in ROS

sogenannten Markern und Polygonen. Marker können zum Beispiel einen Quader darstellen und in Rviz angezeigt werden. Die benötigten Informationen sind in der 3D-Objektliste bereits enthalten, müssen jedoch noch konvertiert werden. Zum Beispiel muss zwischen der Darstellung der Pose als Quaternion und in Eulerwinkeln umgerechnet werden. Außerdem müssen obsolete Mar-

ker nach einer gewissen Zeit gelöscht werden, bei einem bewegten Objekt würden sonst die Marker mit jedem Bild hinzugefügt, ohne dass Marker an den alten Positionen entfernt würden. Jedoch kann es sein, dass ein Objekt durch die Objekterkennung in einzelnen Bildern übersehen wird. Um ein Umherspringen zwischen dargestellt und nicht dargestellt zu vermeiden, wurde daher definiert, dass jeder Marker 5 s gültig ist. Eine weitere notwendige Transformation ist die der Positionen in das Weltkoordinatensystem, da sich bei einer Bewegung des Referenzkoordinatensystems des Fahrzeuges sonst Fehldarstellungen ergeben würden. Mit einem Timestamp versehen werden die Marker schließlich gebündelt als Marker-Array an Rviz gesendet.

Zusätzlich werden die 3D-Objekte noch in Polygone umgewandelt, dies kann später für die Erstellung der Hinderniskarte genutzt werden. Hierfür wird die Grundfläche der Projektion der 3D Bounding Box auf die Grundebene genutzt. Um nicht nur einzelne Polygone ohne Timestamp versenden zu können (ein Timestamp ist für die Kartenerstellung zwingend notwendig), wurde hier eine neue ROS-Message-Definition für eine Polygon-Liste mit Timestamp erstellt (s. Listing 5).

```
1 std_msgs/Header header
2 geometry_msgs/Polygon[] polygons
```

Listing 5: ObjectListPolygons.msg

Die letzte Aufgabe des Visualisierungs-Nodes ist die **3D-Segmentierung**: Hier wird anstatt der Bereiche eines 2D-Bildes die Punktwolke „eingefärbt“. Hierzu werden die Daten der Punktwolke in die Koordinaten der 2D-Kamera umgerechnet und alle Punkte entfernt, die nicht im Erfassungsbereich der Kamera enthalten sind. Nun wird über den Vergleich mit bereits segmentierten 2D-Bilddaten geprüft, welcher Punkt der Punktwolke zu welcher Objektklasse gehört und diese entsprechend den Punktkoordinaten in der Wolke hinzugefügt. Anschließend wird diese neue Punktwolke bereitgestellt. Auch diese kann zur Erstellung der Hinderniskarte genutzt werden. Zur Veranschaulichung sei hier erneut auf die Abbildung 5.10 auf Seite 137 verwiesen.

Einfügen der Objekte in die Karte Da der im ROS Navigation Stack verwendete Wegfindungsalgorithmus eine sogenannte Costmap zur Routenplanung

nutzt, muss diese entsprechend angepasst werden, um die durch die Objekterkennung erkannten Hindernisse hinzuzufügen. Generell sind jedem Punkt auf der Karte Kosten zugeordnet: Umso höher die Kosten, umso stärker muss versucht werden, diesen Bereich zu vermeiden. Dabei werden zum Beispiel Hindernissen hohe Kosten zugewiesen. Allerdings lässt sich dieses Verhalten auch nutzen, um zum Beispiel bestimmte Bodenarten (Straße vs. Gehweg oder Rasen) vorzuziehen, sobald diese durch die semantische Segmentierung erkannt sind. Hierzu wird eine neue Schicht eingefügt, die parallel zu eventuell bereits vorhandenen Schichten genutzt werden kann. Außerdem ist eine Anpassung des Kostensystems notwendig, die Kosten von *FREE_SPACE* wurden von 0 auf 50 erhöht, sodass unbekannte Gebiete nicht mehr unbedingt den besten Weg darstellen (eventuell befinden sich dort Hindernisse, die noch nicht erkannt wurden). Neu eingefügte Konstanten sind *OBJECT_COST* (252, erkannte Objekte werden auf jeden Fall vermieden), *TERRAIN_COST* (10, kann befahren werden und wird unbekannten Gebieten vorgezogen) und *ROAD_COST* (0, Straßen sollten bevorzugt befahren werden).

Um diese Informationen effektiv einspeisen zu können, wurde ein Plugin für das *costmap_2d*-Paket von ROS erstellt. Die neue Schicht muss dazu zwei neue Funktionen implementieren: *updateBounds* und *updateCosts*. Erstere gibt an, welche Bereiche aktualisiert werden müssen, somit muss nicht für jedes neu erkannte Objekt die Costmap rechenaufwändig neu erstellt werden, letztere führt diese Aktualisierung durch. Empfangene Polygone des Objekterkennungs-Nodes werden auf die Koordinaten der Costmap gemappt, die Kostenklasse wird entsprechend der Objektklassen gesetzt, zum Beispiel werden die verschiedenen Objekte, die als Hindernis erkannt werden sollen (Fußgänger, Bäume…) der Klasse *OBJECT_COST* zugeordnet. Analog wird hierbei bei der segmentierten 3D-Punktwolke vorgegangen. Anschließend werden die einzelnen Schichten zusammengefasst, dabei wird entsprechend der Konfiguration von oben nach unten durchgegangen (s. Listing 6 auf der nächsten Seite).

Ohne weitere Änderung des Vorgehens werden Schichten durch nachfolgende Schichten überschrieben: eine vorher bekannte Karte, als *static_layer* eingebunden, wird von der in dieser Arbeit integrierten Objekterkennungsschicht mit Hindernissen versehen. Schließlich dienen die bereits vorhandenen Schichten als eine Art Rückfallebene: Durch die Objekterkennung nicht erkannte Hindernisse werden durch die einfachere *obstacle_layer*-Schicht ergänzt. So-

```
plugins:
 - {name: static_layer,           type:
   ↪  "costmap_2d::StaticLayer"}
 - {name: object_detection_layer, type:
   ↪  "object_detection_namespace::ObjectDetectionLayer"}
 - {name: obstacle_layer,         type:
   ↪  "costmap_2d::ObstacleLayer"}
 - {name: inflation_layer,        type:
   ↪  "costmap_2d::InflationLayer"}
```

Listing 6: Ausschnitt aus der Konfigurationsdatei der Costmap

weit ist das Verhalten gewünscht, es kann jedoch auch der umgekehrte Fall auftreten: Ein erkanntes Objekt wird von der *obstacle_layer*-Schicht als freier Raum erkannt. Das Hindernis hiermit zu überschreiben wäre fatal. Daher wurde der Algorithmus der Aktualisierung angepasst, freier Raum kann nur gesetzt werden, wenn die Schicht darüber an dieser Stelle kein Hindernis erkannt hatte, also die Kosten *NO_INFORMATION* zugewiesen sind.

5.1.4 Wegfindung und Navigation

Wegfindung und Navigation basieren auf dem *navigation stack* von ROS [ros20a]. Dieser nutzt die vom Cartographer erstellte und von der Objekterkennung mit der Position der Hindernisse angereicherte Karte, um das Fahrzeug zum vorgegebenen Ziel zu bewegen.

Die Wegfindung wird dabei zweistufig ausgeführt, um Ressourcen wie RAM zu schonen: Zuerst wird auf einer globalen Karte, die die gesamte bekannte Umgebung umfasst und relativ grob aufgelöst ist, die Route geplant. Auf einer sogenannten lokalen Karte findet anschließend die Trajektorienplanung statt. Diese Karte ist ein kleiner Ausschnitt aus der globalen Karte an der aktuellen Position des Fahrzeuges, der feiner aufgelöst ist. Ziel dieser Trajektorienplanung ist, der Route zu folgen und dabei Hindernisse zu umfahren.

Dazu wird eine Geschwindigkeits- und Winkelgeschwindigkeitsanforderung ausgegeben, die durch einen ROS-Node basierend auf den Informationen aus dem Fahrzeugmodell wie Wendekreis und Maximaleinschlag in ein für

das Fahrzeug interpretierbares Format, bestehend aus Geschwindigkeit und Lenkwinkel übersetzt wird.

Diese wird dann über einen CAN-ROS-Node per CAN an das Fahrzeug gesendet. Außer dem Senden von Sollwerten für Lenkwinkel und Geschwindigkeit dient dieser Node auch dem Empfangen von Istwerten und dem Publizieren dieser im ROS-System.

5.2 Beispielhafte Fahrzeugarchitektur zur Umsetzung der Use-Cases

Basierend auf dem allgemeinen Konzept für die (re-)konfigurierbare Fahrzeugarchitektur (s. Abbildung 4.1 auf Seite 102), wird eine Demonstrations- und Evaluationsplattform erstellt (s. Abbildung 5.14 auf der nächsten Seite). Diese dient dazu, die Use-Cases prototypisch umsetzen und damit das Konzept evaluieren zu können.

Dabei soll ein Netzwerk von Steuergeräten dargestellt werden, dessen Hardware automatisiert „getauscht“ werden kann. Kubernetes als Orchestrator für dieses Netzwerk wurde dabei Docker Swarm (s. auch Abschnitt 3.7.1 auf Seite 90) vorgezogen, da es im Gegensatz zu diesem bereits leistungsfähige Tools für Logging und Monitoring sowie ein Dashboard zur Verwaltung enthält.

Die einzelnen ECUs des Fahrzeuges stellen dabei die Nodes des Kubernetes-Clusters dar, mit dem Orchestrator als *Master-Node* (s. Abschnitt 5.5.3 auf Seite 153).

Dabei ermöglicht die Virtualisierung (s. Abschnitt 3.7.1 auf Seite 86) von Steuergeräten eine reproduzierbare und veränderbare Umgebung, was wertvoll für die spätere Evaluation ist. Zum Beispiel lassen sich so Hardwareressourcen wie CPU-Kerne oder RAM-Ausstattung dynamisch anpassen, ebenso wie Netzwerkverbindungen beziehungsweise Unterbrechungen dieser.

So wird die bereits in Abschnitt 4.2.2 auf Seite 104 vorgestellte Orchestrator-ECU hier virtualisiert.

Zwei ebenfalls virtualisierte ECUs stellen die Flexibilität im Hinblick auf unterschiedliche Hardwareausstattung und Prozessorarchitekturen dar: Eine

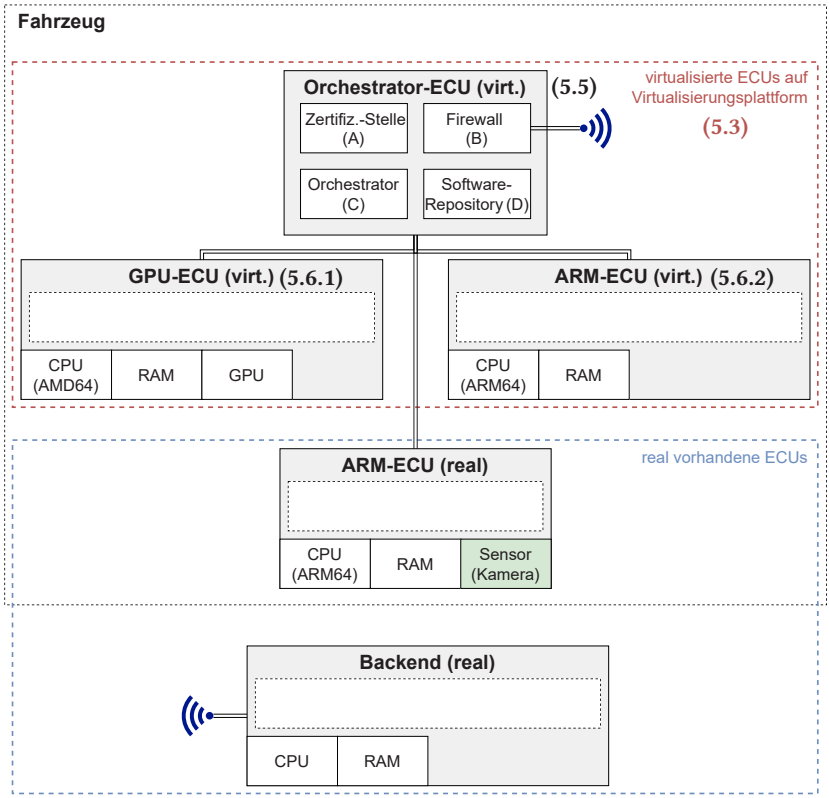


Abbildung 5.14: Struktur der (re-)konfigurierbaren Beispiels-Fahrzeugarchitektur.

enthält eine GPU zur Beschleunigung von zum Beispiel der Berechnung neuronaler Netze, die andere basiert auf einer ARM64-Prozessorarchitektur, wie sie bereits heutzutage in Fahrzeugen eingesetzt wird.

Letztere muss dabei emuliert werden (s. Abschnitt 3.7.1 auf Seite 92), um die ARM64-Prozessorarchitektur auf einer AMD64-Plattform darstellen zu können.

Abgesehen von der Demonstration der Lauffähigkeit muss auch der Einfluss der Emulation auf die Performanz evaluiert und mit der Ausführung auf realer

Hardware verglichen werden. Dabei wird der Fokus auf die Emulation von ARM64 gelegt, da diese Architektur im eingebetteten Bereich weit verbreitet ist.

Ein weiteres Steuergerät auf ARM64-Basis ist real vorhanden und zeigt so, dass das Vorgehen auch auf echte Hardware übertragbar ist. Zusätzlich kann es durch eine verbundene Kamera als intelligenter Sensor eingesetzt werden. Um die Kommunikation mit dem Backend zu untersuchen, ist dieses als reale Komponente vorhanden und vom restlichen Aufbau räumlich getrennt.

Für die Virtualisierungsplattform bietet sich dabei QEMU in Verbindung mit KVM an, da ersteres für die Emulation alternativlos ist und von den in Abschnitt 3.7.1 auf Seite 86 genannten alternativen Lösungen für die Virtualisierung nur KVM mit QEMU kompatibel ist. KVM wird hierbei benötigt, da es die Ausführung von binärkompatiblem Code mit QEMU maßgeblich beschleunigt.

Verteilung von Softwarekomponenten auf die ECUs

Virtualisierung wird außerdem genutzt, um mehrere Services oder Softwarekomponente gleichzeitig ohne Wechselwirkungen auf einem Steuergerät auszuführen. Damit wird von der zugrundeliegenden Hardware abstrahiert, sodass ein Verschieben zwischen verschiedenen Steuergeräten (Hosts) möglich ist.

Dazu wird das Konzept der Containerisierung (s. Abschnitt 3.7.1 auf Seite 88) genutzt, um die einzelnen Komponenten als Bündel zu verpacken und mittels Registry zu verteilen (gemäß AP5). Zusätzlich wird überflüssiger Overhead im Sinne von Rechenleistung und Arbeitsspeicherauslastung vermieden, indem Teile des Betriebssystems geteilt werden und somit nicht mehrfach ausgeführt werden müssen.

Aufgrund der Sicherheitsmängel von chroot, den Einschränkungen von LXC und LXD (keine Entsprechung zu Dockerfiles vorhanden, keine Registry für Images) und der noch geringen Verbreitung von Podman wird hier auf Docker gesetzt.

Eine CI/CD-Pipeline (s. Abschnitt 3.7.2 auf Seite 93) automatisiert dabei die Schritte von der Erstellung der Container bis zu ihrer Verteilung auf den

ECUs des Fahrzeuges. Gitlab (s. Abschnitt 3.7.2 auf Seite 94) eignet sich hierfür besonders, da es die gesamte benötigte Funktionalität inklusive der Integration in Kubernetes (s. Abschnitt 3.7.1 auf Seite 90) abdeckt.

5.3 Einrichtung der Virtualisierungsplattform

Da sowohl KVM als auch QEMU unterstützt werden müssen und zu Demonstrationszwecken eine grafische Benutzeroberfläche vorhanden sein sollte, wird *Proxmox VE* [Pro20] als Basis für die Virtualisierung eingesetzt, welches diese Anforderungen erfüllt. Im Rahmen dieser Dissertation ist von Vorteil, dass es sich per Webbrowser bedienen lässt und dort auch die grafische Ausgabe der virtuellen Maschinen erfolgt. Es ist daher nicht erforderlich, einen Monitor an das Fahrzeug anzuschließen.

Da der von Proxmox VE bereitgestellte Installer für eine direkte Installation die verwendete GPU noch nicht unterstützt, muss die Linux-Basis zuerst getrennt installiert werden. Hier ist dies *Debian 10 „Buster"*, das eine modernere Installationsroutine mitliefert. Anschließend kann *Proxmox VE* wie gewohnt über den Paketmanager installiert werden.

5.3.1 Einrichtung des GPU-Passthrough

Eine wichtige Vorraussetzung für das effektive Ausführen von Algorithmen zur Objekterkennung für das automatisierte Fahren ist die Verfügbarkeit einer GPU in der virtuellen Maschine. Dabei kann jeweils eine reale GPU an eine virtuelle Maschine durchgereicht werden, das sogenannte *GPU-Passthrough*. Damit dies reibungslos funktioniert, muss dem Host zuerst der Zugriff auf die GPU entzogen werden. Anschließend kann diese an virtuelle Maschinen durchgereicht werden, ohne dass diese Zugriff auf die restliche Hardware des Hosts erlangen.

5.3.2 Netzwerkkonfiguration

Damit die Komponenten untereinander kommunizieren können und außerdem eine durch eine Firewall geschützte Kommunikation nach außen, zum Beispiel mit dem Backend, möglich ist, existieren zwei getrennte Netzwerke (s. Abbildung 5.15).

Externes Netzwerk (1) In diesem Netzwerk befindet sich das WAN-Interface der Firewall (2, s. Abschnitt 5.5.2 auf Seite 152). Es ist mit der Außenwelt verbunden, zum Beispiel über einen WLAN-Access-Point oder ein Netzwerkkabel mit Internetverbindung. In Abbildung 5.14 auf Seite 146 wird hierüber zum Beispiel die sichere Verbindung zum Backend über ein VPN realisiert.

Internes Netzwerk (5) Hier sind alle anderen Komponenten des Fahrzeugs miteinander verbunden und durch die Firewall vor der Außenwelt und unbefugten Zugriffen geschützt (4). Dabei ermöglicht die Netzwerkbrücke zu einer getrennten Netzwerkkarte (3) das Einbinden weiterer, realer Komponenten in das Fahrzeugnetzwerk (zum Beispiel die zusätzliche reale ARM-ECU aus Abschnitt 5.2 auf Seite 145). In Abbildung 5.14 auf Seite 146 handelt es sich dabei um das die ECUs verbindende Netzwerk.

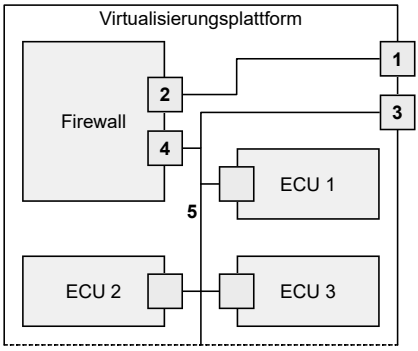


Abbildung 5.15: Struktur Fahrzeugnetzwerk auf dem Virtualisierungshost

5.4 Grundkonfiguration der Steuergeräte-Nodes

Wie in Abschnitt 5.2 auf Seite 145 beschrieben, werden die einzelnen Steuergeräte des Fahrzeuges bis auf wenige Ausnahmen auf der im vorherigen Absatz eingerichteten Virtualisierungsplattform virtualisiert.

Die Grundkonfiguration dieser virtuellen Maschinen ist bis auf einige Ausnahmen identisch, so wird jeder virtuellen Maschine bei der Erstellung eine Anzahl an CPU-Kernen und Ressourcen wie Arbeitsspeicher (RAM) oder Festplattenspeicher zugewiesen. Damit lässt sich genau festlegen, welche der virtuellen Maschinen welchen Anteil der Ressourcen maximal nutzen kann. Indem vermieden wird, dass den virtuellen Maschinen in Summe mehr Ressourcen als auf dem Host verfügbar zugewiesen werden, ist automatisch ausgeschlossen, dass eine fehlerhafte Komponente beziehungsweise virtuelle Maschine die restlichen beeinträchtigt.

Wie auf dem Host (s. vorherigen Abschnitt), kommt sowohl in den virtuellen Maschinen als auch auf den realen Steuergeräten die Linux-Distribution *Debian 10 „Buster“* beziehungsweise ihr Äquivalent für den Raspberry Pi *Raspberry Pi OS* zum Einsatz. Da das dort standardmäßig zur Filterung von Netzwerkpaketen verwendete *nftables* nicht mit der Netzwerkkonfiguration von Kubernetes kompatibel ist, muss auf das Paket *iptables-legacy* umgestellt werden. Ohne diese Änderung könnten Kubernetes-Pods auf verschiedenen ECUs nicht miteinander kommunizieren.

Als Basis für die einzelnen das Fahrzeugnetzwerk darstellenden Kubernetes-Nodes kommt die ressourcenschonende Kubernetes-Variante *K3s* zum Einsatz. Diese wird so konfiguriert, dass Docker für die Ausführung der Container genutzt wird. Dadurch ist ein Zugriff auf das Software-Repository in Form einer von Gitlab bereitgestellten Docker-Registry (s. Abschnitt 5.5.4 auf Seite 154) sowie die Nutzung der GPU in Containern möglich ist. Da für den Demonstrator eine NVIDIA-GPU genutzt wird, ist hierfür das *NVIDIA GPU Device Plugin* notwendig.

5.5 Die virtualisierte Orchestrator-ECU

Wie in Abbildung 5.14 auf Seite 146 dargestellt, führt die Orchestrator-ECU die Komponenten *Zentrale Zertifizierungsstelle* (A, Abschnitt 5.5.1), *Firewall* (B, Abschnitt 5.5.2 auf der nächsten Seite), *Orchestrator* (C, Abschnitt 5.5.3 auf Seite 153) und *Software-Repository* (D, Abschnitt 5.5.4 auf Seite 154) aus.

5.5.1 Zentrale Zertifizierungsstelle (A)

Die zentrale Zertifizierungsstelle dient dazu, dass Zertifikate ausgestellt und widerrufen werden können, die zur Authentifizierung von Komponenten und der Verschlüsselung der Kommunikation im Fahrzeug dienen. Da normalerweise das Fahrzeug von außen nicht über eine öffentliche Adresse erreichbar sein soll, wird keine kostenpflichtige externe Domain benötigt. Daher kann eine eigene, vom Hersteller definierte Adresse (für die prototypische Umsetzung im Rahmen dieser Dissertation *nora.lan*) gewählt werden.

Dadurch lassen sich allerdings verfügbare Dienste wie *Letsencrypt* nicht nutzen, um Zertifikate zu erstellen. Die Konsequenz ist, dass eine eigene Zertifizierungsstelle erstellt werden muss, die die zur Domain gehörenden Zertifikate signiert. Dies bedeutet auch, dass nahezu alle Clients dieser Zertifizierungsstelle standardmäßig nicht vertrauen und diese entsprechend auf diesen hinzugefügt werden muss. Das Hinzufügen kann entweder manuell oder automatisch über Skripte (s. Abschnitt 5.8.1 auf Seite 165) geschehen. Anschließend wird allen von dieser Zertifizierungsstelle signierten Zertifikaten vertraut, ohne dass diese explizit bekannt sein müssten.

Besonders wichtig ist in diesem Kontext, dass die privaten Zertifikate der Zertifizierungsstelle, mit denen neue Zertifikate signiert werden könnten, vor einem unbefugten Auslesen geschützt werden. Ansonsten könnten beliebig Zertifikate erstellt werden, denen automatisch vertraut würde.

Ein Detail, das in diesem Zusammenhang zu beachten ist, ist, dass der sogenannte *Common Name* der Zertifikate nicht mit dem Namen der Zertifizierungsstelle übereinstimmen darf, da alle *OpenSSL*-basierten Implementierungen diese ansonsten gültigen Zertifikate nicht akzeptieren.

5.5.2 Firewall (B)

Als Firewall kommt *pfSense* [pfS20] zum Einsatz. Diese ist weit verbreitet und damit erprobt, prinzipiell ließen sich jedoch ebenso andere Lösungen einsetzen, die auch einen VPN-Server beinhalten.

Die Installation erfolgt dabei über das auf der Produktwebseite erhältliche CD-Image, bei der Installation wird das Netzwerkinterface *WAN*[3] (2 in Abbildung 5.15 auf Seite 149) dem externen und das LAN-Interface (4) dem internen Netzwerk zugewiesen.

In Verbindung mit der Virtualisierung muss anschließend das *hardware checksum offloading* deaktiviert werden, ansonsten ist keine Kommunikation mit oder über die Firewall möglich. Zusätzlich muss das Blockieren von Paketen aus privaten Adressbereichen auf dem WAN-Interface deaktiviert werden. Dieses Verhalten ist eigentlich ein Sicherheitsmechanismus, um Attacken über das Internet zu erschweren. Im Rahmen dieser Dissertation liegt jedoch auch das WAN-Interface in einem privaten Adressbereich.

Da die Firewall auch für die Namensauflösung zuständig ist, sprich der Zuordnung von IP-Adressen zu Hostnamen, müssen für Komponenten wie Gitlab oder Kubernetes-Master feste IP-Adressen und definierte Namen vergeben werden. Die Domain entspricht dabei der der zentralen Zertifizierungsstelle. So ist sichergestellt, dass neue Komponenten sich am System anmelden können, ohne dass Details zur Netzwerkkonfiguration vorab bekannt sind.

Verbindung des Fahrzeuges mit dem Backend

Damit mit dem Backend kommuniziert werden kann und auch entfernte Steuergeräte oder Sensoren hinzugefügt werden können, wird eine sichere Verbindung zu diesen benötigt, besonders bei einer Anbindung über Mobilfunk oder WLAN (in Abbildung 5.14 auf Seite 146 als blaue Antennen dargestellt). Wie auf Seite 153 beschrieben, müssen Paket-Multicasts, also Pakete mit mehreren Empfängern, sowie weitere Ethernet-Features unterstützt werden. Dazu reicht ein standardmäßiges, geroutetes VPN auf dem sogenannten *Layer*

[3] Wide Area Network

3 nicht aus, da Features des darunterliegenden *Layer 2* erforderlich sind (vgl. OSI-Layer, Abschnitt 3.2.1 auf Seite 51). Daher muss eine Netzwerkbrücke erstellt werden, die die Kommunikationsteilnehmer auch auf dieser Ebene miteinander kommunizieren lässt.

Die Umsetzung findet als Teil von Use-Case 4: „Auslagern von Services ins Backend“ statt (s. Abschnitt 5.9 auf Seite 167).

5.5.3 Orchestrator (C)

Der Orchestrator stellt den Kubernetes-Master dar. Bei der Einrichtung des Kubernetes-Masters wird automatisch ein Token erstellt, das zum Hinzufügen weiterer Nodes in Form der anderen Steuergeräte des Fahrzeuges zum Cluster genutzt werden kann. Um die Konfiguration und den Status möglichst komfortabel bearbeiten beziehungsweise überwachen zu können, wird das grafische Webinterface *Kubernetes Dashboard* installiert und auch hier ein Token für den Zugriff generiert.

Kommunikation im Kubernetes-Cluster

Standardmäßig können Container mit anderen Containern im selben Pod kommunizieren, jedoch nicht über Pods oder gar Cluster-Nodes beziehungsweise Steuergeräte hinweg. Um dies zu ermöglichen, gibt es mehrere Möglichkeiten: Zum einen können vor den Pods sitzende Loadbalancer Anfragen empfangen und auf die Pods verteilen. Die Bezeichnung *Loadbalancer* rührt daher, dass sie vor allem dazu genutzt werden, die Anfragelast gleichzeitig auf mehrere Serviceanbieter zu verteilen. Nachteil ist, dass diese Loadbalancer selbst eine mögliche Fehlerquelle darstellen und üblicherweise nur bestimmte Netzwerkprotokolle unterstützen. Eine Brücke der Pod-Interfaces an die Netzwerkschnittstelle des Hosts ist hingegen einfacher einzurichten und flexibler, da jeglicher Netzwerkverkehr weitergeleitet wird. Dieses Verhalten ähnelt damit einer direkten physischen Verbindung der Container untereinander. Notwendig ist das beispielsweise für eine effiziente Unterstützung von Multicasts. Dabei werden Pakete, ähnlich wie bei einem Bus einer signalbasierten Architektur, an mehrere Netzwerkteilnehmer auf einmal verteilt, was den dazu benötigten Overhead stark verringert.

Um diese Netzwerkbrücke zu aktivieren, muss in Kubernetes für jeden hinzuzufügenden Pod in dessen Konfiguration `hostNetwork: true` gesetzt werden.

5.5.4 Software-Repository (D)

Als Software-Repository wird die Registry von *Gitlab* (s. Abschnitt 3.7.2 auf Seite 94) eingesetzt.

Gitlab wird dabei wie ein gewöhnliches Paket entsprechend der Installationsleitung installiert, mit einer Anpassung:

```
1 EXTERNAL_URL="http://gitlab.nora.lan" apt-get install
  ↪ gitlab-ce
```

Dies bewirkt, dass bei der Installation HTTPS deaktiviert wird, da Gitlab ansonsten eine Konfiguration der Zertifikate mittels Letsencrypt versuchen und damit die gesamte Installation scheitern lassen würde. Da *nora.lan* keine gültige vergebene Domäne ist und nur intern genutzt wird, muss auch hier ein Zertifikat erstellt werden, das von der zentralen Zertifizierungsstelle des Fahrzeuges signiert wird. Anschließend wird HTTPS wieder aktiviert und dieses neue Zertifikat zur Verschlüsselung von Verbindungen und zur Authentifizierung genutzt.

Gitlab Runner

Der *Gitlab Runner* dient der Ausführung der CI/CD-Pipeline (s. übernächster Abschnitt auf Seite 156). Dazu führt er in Docker eine Build-Umgebung aus, in denen die Dockerfiles der Softwarekomponenten gebaut, in das Software-Repository übertragen und anschließend im Kubernetes-Cluster verteilt werden. Diese Build-Umgebung in Docker erfordert besondere Rechte, daher ist es erforderlich, die Option `privileged=true` in der Docker-Konfiguration zu setzen. Da die damit ausgeführten Container besondere Befugnisse haben, die auch den Host betreffen können, ist besonders darauf zu achten, dass dieser Runner nicht genutzt wird, um nicht vertrauenswürdige Softwarekomponenten zu bauen.

Verknüpfung von Gitlab und Kubernetes

Die Verknüpfung von Gitlab und Kubernetes muss bidirektional erfolgen. Das heißt, beide Komponenten müssen auf die jeweils andere zugreifen können, um dort administrative Aufgaben durchführen und Vorgänge wie das Ausrollen von Softwarekomponenten starten zu können.

Erster Schritt ist dabei, um von Gitlab aus auf den Kubernetes-Cluster zugreifen zu können, dass auf dem Kubernetes-Master ein sogenanntes *Service-Token* erstellt wird. Dieses ermöglicht es Gitlab in Verbindung mit der ebenfalls zu konfigurierenden Adresse des Clusters, auf die Ressourcen des Kubernetes-Clusters zuzugreifen und diesen zu konfigurieren.

Um die speziell das Fahrzeug betreffenden Artefakte von denen trennen zu können, die von Kubernetes standardmäßig angelegt werden, wurde der Namespace „Nora“ angelegt. Damit lassen sich leichter Änderungen an der Fahrzeugkonfiguration nachverfolgen und rückgängig machen.

So wird auch für diesen Namespace ein getrenntes Token angelegt, das das Starten von Containern erlaubt und somit von der CI/CD-Pipeline (s. nächster Abschnitt) genutzt werden kann, um neu gebaute Softwarekomponenten im Fahrzeug zu verteilen und zu starten. Dieses Token kann in Gitlab an die Pipeline über sogenannte *Variablen* übergeben werden, sodass ein Auslesen durch Unbefugte behindert wird. Damit sind keine unautorisierten Zugriffe möglich oder zumindest stark erschwert.

Um auch einen Zugriff in die andere Richtung zu ermöglichen, also dem Kubernetes-Cluster den Zugriff auf Gitlab zu erlauben, muss auch dort ein Token erstellt werden. Dieses erlaubt es Kubernetes, auf das Software Repository zuzugreifen. Dies ist erforderlich, um neu gebaute Softwarekomponenten in Form von Container-Images von dort beziehen zu können. Zusätzlich zu diesem Token muss außerdem in Gitlab explizit der Zugriff vom Kubernetes-Master aus freigeschaltet werden, da dies standardmäßig aus Sicherheitsgründen aus dem gesamten Netzwerk blockiert wird. Ohne diese Freischaltung ist keine Kommunikation der Gitlab-Pipeline mit dem Kubernetes-Cluster möglich.

CI/CD-Pipeline

Die CI/CD-Pipeline dient dazu, neue Softwarekomponenten zu bauen und im Fahrzeug zu verteilen. Dazu werden mit der Dockerpipeline (s. nächster Abschnitt) aus sogenannten *Dockerfiles* Docker-Images erstellt. GIT-Repositories in Gitlab dienen als Basis, um die von Gitlab angebotenen Möglichkeiten der Continuous Integration (s. Abschnitt 3.7.2 auf Seite 93) zu nutzen, um Pakete zu bauen und anschließend über das Deployment in Kubernetes (s. übernächster Abschnitt) auf die Steuergeräte zu verteilen.

Die Docker-Pipeline

Die Docker-Pipeline wird basierend auf dem alle Softwarekomponenten des Fahrzeuges enthaltenden Repository ausgeführt, jedoch werden nur die Komponenten neu gebaut, die entweder verändert wurden oder die Abhängigkeiten zu veränderten Komponenten besitzen (s. Abbildung 5.16 auf der nächsten Seite). Um die Anforderung, dass es möglich sein soll, Softwarekomponenten auf unterschiedliche ECUs zu verschieben (AP2), zu erfüllen, müssen die Container entweder für jede im Fahrzeug enthaltene Prozessorarchitektur gebaut oder dort emuliert werden. Ersteres Vorgehen kann erfordern, dass auch bestehende Software neu gebaut werden muss, wenn neue ECUs hinzugefügt werden. Letzteres Vorgehen hat einen starken Einfluss auf die Performanz durch die aufwendige Emulation. Daher müssen beide Alternativen auf ihre Auswirkungen untersucht werden (s. Abschnitt 6.1.4 auf Seite 177).

Das Deployment in Kubernetes

Die Verteilung auf die verschiedenen ECUs erfolgt über das Deployment in Kubernetes.

Dazu wird basierend auf den Anforderungen des Containers an die Hardware und freie Ressourcen im Cluster entschieden. Dabei umfassen die Hardwareanforderungen nicht nur, wie viel RAM oder CPU-Leistung benötigt werden, sondern auch erforderliche Spezialhardware wie FPGAs oder GPUs.

Für Ressourcen wie CPU und RAM sind bereits Mechanismen in Kubernetes vorhanden: Einem Container kann in seiner Konfiguration per *request*

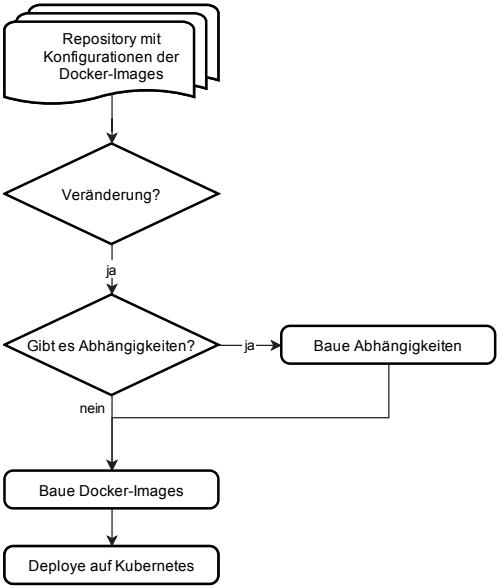


Abbildung 5.16: Ablauf der Docker-Pipeline

eine Mindestmenge an Ressourcen zugewiesen werden, die ein Cluster-Node verfügbar haben muss, damit der Container auf diesem ausgeführt werden kann. Ebenso ist es möglich, per *limit* die maximale Menge der Ressourcen zuzuweisen, die der Container nutzen kann.

Um auch Spezialhardware, andere Komponenten oder zum Beispiel auch eine bestimmte Hardwarearchitektur gezielt zuweisen zu können, werden die *Labels* und *Selectors* von Kubernetes verwendet. Dabei lassen sich erstere mehreren ECUs beziehungsweise Cluster-Nodes zuweisen und geben damit Auskunft über deren Eigenschaften. Zum Beispiel kann ein Label *GPU* bedeuten, dass eine schnelle GPU verbaut ist und ein weiteres Label *AMD64*, dass die Prozessorarchitektur *AMD64* genutzt wird.

Bei der Verteilung der Softwarekomponenten werden dann, basierend auf den Anforderungen, über einen *Selector* passende Cluster-Nodes beziehungsweise Steuergeräte für die Ausführung ausgewählt. Dies ist in Abbildung 5.17 auf Seite 159 dargestellt: Da nur eine ECU mit GPU (hellblaues Label) zur Ver-

fügung steht, muss ein Container, der eine GPU benötigt, immer auf dieser ECU ausgeführt werden. Sobald zusätzlich zum Beispiel die Prozessorarchitektur *ARM64* benötigt würde, könnte in diesem Beispiel der Container nicht ausgeführt werden (die Kombination der *Selectors GPU* und *ARM64* ergäbe eine leere Menge). Dafür kann eine Softwarekomponente, die lediglich von der AMD64-Architektur (hellgrünes Label) abhängt, auf beiden ausgeführt werden.

Ähnlich ist auch eine gleichzeitige Ausführung derselben Softwarekomponente auf mehreren Steuergeräten möglich. Das kann zum einen für die Redundanz entsprechend der Safetyanforderung AS1 oder zum anderen für eine Skalierung der Softwarekomponenten geschehen. Durch die Skalierung können dabei die zur Verfügung stehenden Ressourcen effizienter ausgenutzt werden beziehungsweise mehr Ressourcen zur Verfügung gestellt werden, insbesondere bei an sich wenig effizienten beziehungsweise auf weniger leistungsfähige ECUs ausgelegten Softwarekomponenten (s. Abschnitt 6.1.2 auf Seite 174).

In Kubernetes existieren dazu die sogenannten *ReplicaSets*. Damit kann angegeben werden, wie oft ein identischer Pod (der selbst jeweils mehrere Softwarekomponenten in Form von Containern enthalten kann) gleichzeitig ausgeführt werden soll. Abgestürzte Pods werden automatisch neu gestartet, außerdem kann die Redundanz zur Laufzeit verringert oder erhöht werden, es werden dann automatisch Pods beendet beziehungsweise gestartet.

5.6 Sonstige virtualisierte Steuergeräte

5.6.1 GPU-Node – Steuergerät mit Spezialhardware

Der GPU-Node dient dazu, die Einbindung von Steuergeräten mit Spezialhardware darzustellen. Dazu wird diesem Node die in Abschnitt 5.3.1 auf Seite 148 zum Durchreichen vorbereitete GPU zugewiesen, wodurch sie exklusiv genutzt werden kann.

Bei der Virtualisierung gibt es einige Eigenheiten zu beachten, die gegenüber der Einrichtung einer normalen virtuellen Maschine abgeändert werden müssen. So muss das BIOS auf die modernere Alternative *UEFI* umgestellt werden

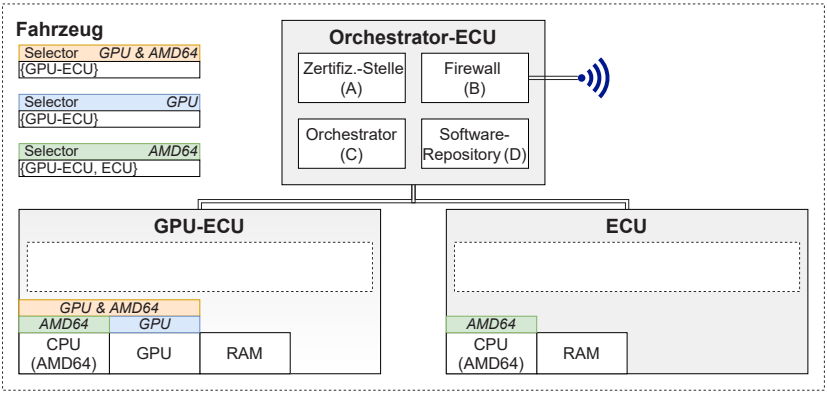


Abbildung 5.17: Deployment in Kubernetes

und ebenso eine dazu kompatible GPU verbaut sein. Zusätzlich muss der Chipsatz der virtuellen Maschine auf *Q35* geändert werden. Dabei handelt es sich um real existierende Chipsätze, die für die virtuelle Maschine nachgebildet werden, damit diese mit Standardtreibern angesprochen werden können. Der Chipsatz Q35 hat den Vorteil, dass er einen virtuellen PCIe-Bus besitzt, der für das Durchreichen einer Grafikkarte mit PCIe-Interface notwendig ist. Sind diese Konfigurationsoptionen gesetzt, kann die durchzureichende Grafikkarte ausgewählt werden. Orientierung bei der Auswahl bietet die in Abschnitt 5.3.1 auf Seite 148 bereits verwendete Hardware-ID.

Ist das Betriebssystem installiert (auch hier *Debian 10 „Buster“*), müssen die Grafiktreiber und Docker mit der *nvidia-container-runtime* installiert werden, damit die Grafikkarte auch in Docker-Containern genutzt werden kann.

5.6.2 ARM-Node – Steuergerät mit anderer Prozessorarchitektur

Um zu zeigen, dass der Ansatz auch funktioniert, wenn ein Fahrzeugnetzwerk aus ECUs mit heterogenen Prozessorarchitekturen besteht, wird eine virtuelle Maschine erstellt, die ein Steuergerät mit ARM64-kompatibler Prozessorarchitektur emuliert.

Auch für den ARM-Node muss das BIOS auf UEFI beziehungsweise *ovmf* umgestellt werden. Da keine der von Proxmox VE zur Verfügung gestellten virtuellen Grafikkarten mit der ARM-Plattform kompatibel ist, muss die Ausgabe über eine vorher hinzuzufügende serielle Schnittstelle erfolgen. Außerdem muss das CD-Laufwerk, von dem das Betriebssystem installiert wird, als SCSI-Laufwerk (anstatt IDE) hinzugefügt werden, damit davon gebootet werden kann.

Da die Nutzung anderer Prozessorarchitekturen von Proxmox VE ausdrücklich nicht unterstützt wird, muss die Architektur auf der Bash explizit mit `qm set ID -arch aarch64` gesetzt werden.

5.7 Use-Case 1: Erweiterung einer bestehenden Architektur

Das Basisfahrzeug für die Umsetzung der (re-)konfigurierbaren Fahrzeugarchitektur „Nora“ (s. Abschnitt 5.1 auf Seite 125) dient dazu, die Erweiterung einer bestehenden signalbasierten Architektur darstellen zu können. Basierend auf einem elektrisch angetriebenen Aufsitzrasenmäher wurde die Grundfunktionalität für das automatisierte Fahren auf diesem *nicht* serviceorientiert in ROS implementiert.

5.7.1 Integration der bestehenden ROS1-Nodes in eine serviceorientierte Architektur in ROS2

In Abbildung 5.18 auf der nächsten Seite ist das Vorgehen bei der Integration der bestehenden ROS-Nodes in die neue serviceorientierte Architektur mit ROS2 dargestellt. Dabei muss für jeden einzelnen Node überprüft werden, ob seine Pakete auch für ROS2 verfügbar sind.

Ist dies der Fall, können auf ROS2 basierende Dockerfiles erstellt, mit der Dockerpipeline gebaut werden (s. Abschnitt 5.5.4 auf Seite 156) und die entstandenen Docker-Images anschließend in das Software-Repository übertragen werden.

Ist dies jedoch nicht der Fall, muss überprüft werden, ob sich die Pakete zu ROS2 migrieren lassen. Wenn ja, zum Beispiel wenn es sich um sehr simple Pakete handelt, wird die Migration durchgeführt und wie gehabt mit der Docker-Pipeline fortgefahren.

Wenn nein, muss zuvor eine Brücke zwischen ROS und ROS2 integriert werden, ein sogenanntes *Technologie-Gateway* (s. Abschnitt 3.5.2 auf Seite 63). Dabei werden weiterhin die alten ROS-Nodes eingesetzt.

Als Technologie-Gateway kommt das Paket *ros1_bridge* [ros20b] zum Einsatz (s. nächster Abschnitt).

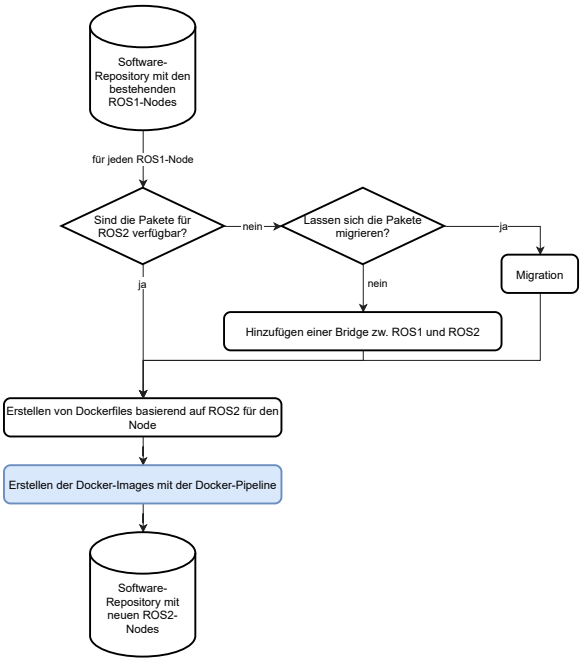


Abbildung 5.18: Vorgehen zum Erstellen neuer Softwarekomponenten oder zur Migration existierender Komponenten

Das Technologie-Gateway *ros1_bridge*

Dieses Paket stellt eine Brücke zwischen ROS- und ROS2-Nachrichten im Netzwerk dar. Die Unterstützung von Nachrichtentypen ist auf solche beschränkt, die bereits zur Kompilierzeit der Bridge existierten zum Beispiel die Standardnachrichtentypen von ROS und ROS2. Im Rahmen dieser Dissertation stellt dies kein Problem dar, da neu hinzukommende Nodes auf ROS2 basieren und hiervon daher nicht betroffen sind. Eine mögliche Lösung für eine dynamische Unterstützung von Nachrichtentypen wäre die Umsetzung der Bridge in *Python* statt *C++*.

Zusätzliche Nachrichtentypen werden beim Kompilieren der Bridge automatisch erkannt und hinzugefügt, falls einige Bedingungen erfüllt sind: So müssen die Nachrichtentypdefinitionen (s. Erläuterung 3.6.1 auf Seite 68) in korrekt benamten Ordnern liegen (auf *_msg* endend), selbst identische Namen besitzen und außerdem müssen sie identisch aufgebaut sein, inklusive exakt übereinstimmender Datentypen.

In der Realität werden diese Bedingungen jedoch in vielen Fällen nicht erfüllbar sein. Ein Grund ist, dass die Konventionen zur Benennung von Feldern in den Nachrichtentypdefinitionen in ROS2 deutlich verschärft wurden. So ist die Verwendung der verbreiteten *Camel-Case*-Bezeichnungen nicht mehr erlaubt: *messageType* muss nun zum Beispiel durch *message_type* ersetzt werden.

Lösung ist hier die Erstellung einer Datei namens *mapping_rules.yaml* (s. Listing 7). In dieser kann das Mapping der Felder der Nachrichtentypdefinitionen von ROS und ROS2 definiert werden, damit eine Übersetzung möglich ist.

```
- ros1_package_name: "beispiel_package"
  ros1_message_name: "beispielMessage"
  ros2_package_name: "beispiel_package"
  ros2_message_name: "beispiel_message"
  fields_1_to_2:
  messageType: "message_type"
```

Listing 7: Beispiel für eine mapping_rules.yaml

Im Beispiel sind die Nachrichtentypdefinitionen sowohl in der ROS- als auch in der ROS2-Version im Paket *beispiel_package* enthalten, jedoch musste der

Name der Nachricht (*beispiel_message*) angepasst werden, ebenso wie der Datentyp *message_type*.

Ist die Brücke erstellt, kann die ehemalige ROS-Komponente mit dem ROS2-System kommunizieren und wie eine native Komponente genutzt werden.

In Abbildung 5.19 ist die aus der Integration der bestehenden Fahrzeugarchitektur mittels Technologie-Gateway in die (re-)konfigurierbare Fahrzeugarchitektur resultierende Gesamtarchitektur abgebildet. Dazu wird in diesem Fall die Brücke (in rot dargestellt) auf der *NeuroECU* ausgeführt und übersetzt zwischen ROS und ROS2, wodurch die bestehenden Softwarekomponenten weiterverwendet werden können. Insgesamt besteht das neue Gesamtsystem also aus den beiden in den Abbildungen 5.4 und 5.14 auf Seite 128 und auf Seite 146 dargestellten Systemen, die über eine Netzwerkverbindung miteinander verbunden sind.

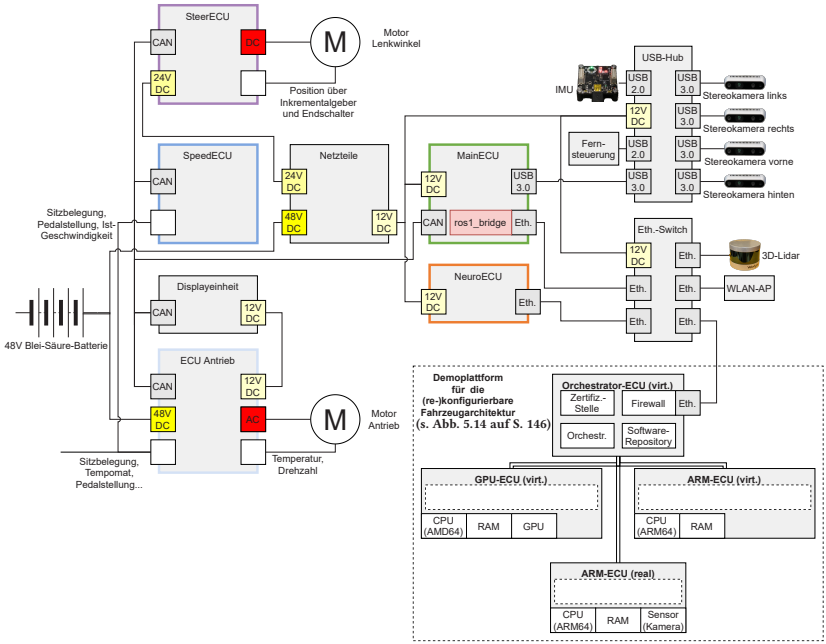


Abbildung 5.19: Gesamtarchitektur nach Integration der bestehenden Fahrzeugarchitektur von Nora in die neue (re-)konfigurierbare Architektur

5.8 Use-Cases 2 und 3: Austausch von Komponenten und Erweiterung der Funktionalität

Das Hinzufügen und der Austausch von Komponenten lässt sich in zwei Abschnitte unterteilen: Zum einen das Hinzufügen und Entfernen der ECUs im Fahrzeug, inklusive Anmeldung im System und Verteilen von gegebenenfalls benötigten Softwarekomponenten, zum anderen die Bekanntmachung von Services und das Umschalten auf diese.

Letzteres kann auch erforderlich werden, sobald eine Komponente zum Beispiel durch einen mechanischen Schaden nach einem Unfall ausfällt und dynamisch auf Services einer anderen Komponente umgestiegen werden muss, um die Funktionalität zu erhalten.

Zur prototypischen Untersuchung des Konzeptes dient die reale ARM-ECU aus Abbildung 5.14 auf Seite 146: Diese besitzt eine Kamera mit der Fähigkeit, ein schwarz-weißes Bild darzustellen.

Als neuer intelligenter Sensor wird eine Farbkamera hinzugefügt. Die Farbsättigung ist in den Metadaten enthalten und dient hier zur Auswahl der „besseren“ Kamera. So wird das Farbbild (hohe Sättigung) dem Schwarzweißbild vorgezogen (niedrige Sättigung).

Zusätzlich zum Umschalten auf die Farbkamera wird der Ausfall dieser Kamera und der Fallback auf die Schwarzweißkamera abgebildet.

5.8.1 Hinzufügen und Entfernen von ECUs

Da der Kubernetes-Master immer unter einem festen Namen erreichbar ist und die IP-Adresse vom DNS der Firewall aufgelöst wird, können sich neue Komponenten (egal ob intelligente Sensoren beziehungsweise Aktoren oder Steuergeräte) immer an diesem anmelden, ohne Kenntnisse über die Netzwerktopologie und verwendete Adressbereiche zu benötigen.

Authentifizierung

Zunächst muss die neue Komponente authentifiziert werden, das heißt ihre „Echtheit“ als vertrauenswürdige Einheit beweisen. Dazu besitzt jede ECU oder sonstige Komponente einen privaten Schlüssel, der nur ihr selbst bekannt ist. Dieser Schlüssel muss von einer vertrauenswürdigen Zertifizierungsstelle signiert sein, also beispielsweise der *zentralen Zertifizierungsstelle* des Fahrzeuges (s. Abschnitt 5.5.1 auf Seite 151) oder einer Zertifizierungsstelle des Herstellers.

Übertragung der Konfiguration und Anmeldung im Fahrzeug

Ist das hinzuzufügende Steuergerät authentifiziert, muss es konfiguriert und im Fahrzeug angemeldet werden. Zur Konfiguration und Anmeldung erhält es vom Kubernetes-Master, dem Orchestrator, Skripte und Konfigurationsdateien. Erstere laden zusätzlich benötigte Softwarepakete (wie zum Beispiel Docker oder Kubernetes) nach, letztere enthalten Information zu beispielsweise der Anmeldung am System.

Im Laufe dieses Vorgangs wird der neuen Komponente auch automatisch die Zertifizierungsstelle des Fahrzeuges hinzugefügt, sodass von dieser signierten Zertifikaten vertraut wird und damit auch das Software-Repository des Fahrzeuges genutzt werden kann. Ein generelles Akzeptieren aller Zertifikate wäre ein Sicherheitsrisiko. So wären zwar die Daten im Fahrzeug immer noch transportverschlüsselt, es könnten aber auch unautorisierte Teilnehmer als Datenquelle oder -senke fungieren (Verletzung von Anforderungen AS3 und AS4), eine sogenannte *Man-in-the-middle-Attacke*.

Hinzufügen von Software-Komponenten zum Fahrzeug

Die Anmeldung am Kubernetes-Cluster des Fahrzeugs über einen ebenfalls in der Konfiguration enthaltenen Schlüssel erlaubt es, dass Softwarekomponenten auf die neue ECU verteilt und dort ausgeführt werden können. Dazu kann das neue Steuergerät eigene Softwarekomponenten in das Software-Repository hochladen, mitsamt dazugehöriger Manifeste, die Hardwareanforderungen und sonstige Informationen enthalten. Dies kann entweder in Form

von Docker-Images über die in Gitlab integrierte Docker-Registry als Software-Repository (s. Abschnitt 5.5.4 auf Seite 154) erfolgen oder in Form von Quellcode und Dockerfiles über die CI/CD-Pipeline von Gitlab (s. Abschnitt 5.5.4 auf Seite 156). Letzteres Vorgehen hat den Vorteil, dass die Softwarekomponente auf der Zielplattform kompiliert werden kann und daher die Software nicht für jede mögliche Hardwareplattform vorkompiliert mitgeliefert werden muss.

Sobald die Softwarekomponenten in das Software-Repository übertragen wurden, werden sie vom Orchestrator gemäß ihrer Anforderungen im Fahrzeug verteilt (s. Abschnitt 5.5.4 auf Seite 156 *Das Deployment in Kubernetes*).

Bekanntmachung von Services und Umschalten auf diese

In dem in dieser Dissertation gezeigten Beispiel liefert jede Kamera ein Bild mit 5 bis 15 Hz, dabei ist die Frequenz der Update-Methode an diese gekoppelt. Hier liegt der Faktor bei fünf, ist aber je nach Bedarf konfigurierbar.

Die Fähigkeiten selbst werden über Metadaten in Form von JSON[4]-Objekten beschrieben, ein Beispiel ist in Listing 8 dargestellt. Darin sind die Position und die Sättigung einer Kamera enthalten, dabei handelt es sich hier um die Farbkamera.

```
1 {
2   "location" : ["0.5f", "0.5f", "0.5f"],
3   "saturation" : 100
4 }
```

Listing 8: Beispiel der Metadaten einer Kamerafähigkeit

Die in Abschnitt 4.2.7 auf Seite 112 beschriebenen erweiterten *Managed Nodes* werden genutzt, um die Umschaltung zwischen den Services basierend auf den Fähigkeiten umzusetzen.

[4] JavaScript Object Notation

Abmeldung vom Fahrzeug

Eine separate Abmeldung von Komponenten vom Fahrzeug ist nicht notwendig. Diese erfolgt nach Ausbleiben von Lebenszeichen nach einer definierten Zeit automatisch. Dabei wird zuerst der entsprechende Kubernetes-Node der ECU so deaktivert, dass ihm keine Softwarekomponenten zur Ausführung zugewiesen werden und nach einer weiteren Wartezeit wird er vollständig aus dem Cluster entfernt. Auch die vormals angebotenen Services und Fähigkeiten (s. Abschnitt 4.2.7 auf Seite 112) werden automatisch als nicht mehr vorhanden erkannt.

Hintergrund dieser Lösung ist, dass im Falle eines Defektes oder bei einem Ausbau bei ausgeschaltetem Fahrzeug keine Abmeldung durch diese Komponente erfolgen kann. Durch den zweistufigen automatischen Entfernprozess wird sichergestellt, dass keine Komponente irrtümlich aus dem System abgemeldet wird.

5.9 Use-Case 4: Auslagern von Services ins Backend

Um Services in das Backend auslagern zu können, muss das Fahrzeug zuerst mit diesem verbunden werden. Ein VPN zwischen Backend, weiteren externen Komponenten und dem Fahrzeug stellt eine sichere verschlüsselte Verbindung her. Dabei kann entweder der VPN-Server im Backend ausgeführt werden und das Fahrzeug stellt eine Verbindung her oder umgekehrt. Letzterer Fall wird hier im Folgenden behandelt, ersterer lässt sich analog umsetzen.

5.9.1 Erstellen einer Verbindung zwischen Fahrzeug und Backend

Damit es keine Probleme mit nicht empfangenen Datenpaketen gibt, muss die Firewall für die virtuellen Maschinen auf der Orchestrator-ECU von *Proxmox VE* deaktiviert werden. Das stellt kein Sicherheitsrisiko dar, da das Netzwerk nur fahrzeugintern ist und weiterhin nach außen durch die Firewall (s. Abschnitt 5.5.2 auf Seite 152) geschützt ist.

Für den Prototypen wurde diese Verbindung per *OpenVPN* auf *pfSense* umgesetzt, das Vorgehen lässt sich jedoch auch auf andere Lösungen übertragen.

Ein von der zentralen Zertifizierungsstelle des Fahrzeuges bereitgestelltes Zertifikat erlaubt es den über das VPN verbundenen Komponenten wie dem Backend, die Echtheit des VPN-Servers des Fahrzeuges zu verifizieren. Ebenso können sich diese über eigene Zertifikate ausweisen, die entweder auch von der zentralen Zertifizierungsstelle des Fahrzeuges oder von einer vertrauenswürdigen Zertifizierungsstelle zum Beispiel des Herstellers signiert wurden. Dieses Vorgehen ist sicherer als der Austausch von Passwörtern. Da eine

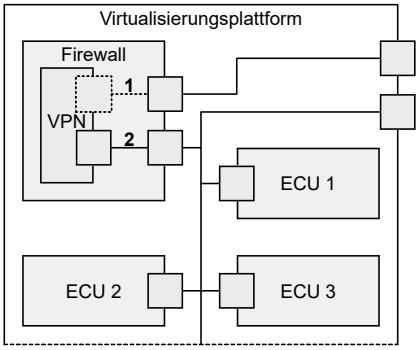


Abbildung 5.20: Integration des VPN-Servers in das Fahrzeugnetzwerk

Netzwerkbrücke auf Layer 2 benötigt wird (s. Abschnitt 5.5.2 auf Seite 152), muss ein *TAP*-Interface angelegt werden, die entstehende Struktur ist in Abbildung 5.20 dargestellt: (1) ist die Verbindung des VPN-Servers zur Außenwelt, (2) die angesprochene Netzwerkbrücke zum Fahrzeugnetzwerk. Anschließend können über das VPN mit dem Fahrzeug verbundene Komponenten so mit diesem kommunizieren, als wären sie lokal mit dem Fahrzeugnetzwerk verbunden. Damit können sie in den Kubernetes-Cluster des Fahrzeuges integriert werden. Dies erfolgt wie in Abschnitt 5.8.1 auf Seite 165 beschrieben.

Verteilen und Nutzen von Services

Ist die Verbindung mit dem Backend oder einer anderen externen Komponente hergestellt, kann darauf zugegriffen werden als befinde sich die Komponente

oder das Backend direkt im Fahrzeugnetzwerk. Daher verläuft das Verteilen und Nutzen von Services vom beziehungsweise ins Backend analog zu dem Verschieben auf andere ECUs wie in den beiden vorherigen Use-Cases (s. Abschnitt 5.8 auf Seite 164).

6 Performanzevaluation kritischer Punkte

6.1 Bewertung der Performanz von Containerisierung, Emulation und Neukompilierung

Um eine Bewertung der Machbarkeit und Sinnhaftigkeit des Konzepts und seiner Detailentscheidungen durchführen zu können, genügt es nicht, nur die theoretische Umsetzbarkeit in der Praxis zu untersuchen. Es muss auch sichergestellt werden, dass die Performanz in der Praxis nutzbare Werte erreicht und keine gravierenden Nachteile gegenüber bestehenden Lösungen entstehen.

Die Containerisierung wird dabei allgemein verwendet, um Softwarekomponenten gebündelt verteilen und zwischen ECUs verschieben zu können und ist daher eine Basisvoraussetzung für das Konzept der (re-)konfigurierbaren Fahrzeugarchitektur. Emulation und Neukompilierung sind darüber hinaus zwei Möglichkeiten, die Plattformunabhängigkeit umzusetzen und kommen in verschiedenen Situationen zum Einsatz (s. Abschnitt 4.2.1 auf Seite 100).

6.1.1 Plattformen für die Bewertung der Performanz

Für die Bewertung der Performanz kommen verschiedene exemplarische Hardwareplattformen zum Einsatz. Jede einzelne entspricht dabei einer einzelnen ECU in Abbildung 5.14 auf Seite 146. Der Raspberry Pi 4 ist die real vorhandene ARM-ECU, die emulierte ARM64-Plattform stellt die virtualisierte ARM-ECU dar.

Bezeichnung	CPU-Kerne	CPU-Architektur	GPU/Leistungsklasse	Größe des RAMs	emuliert
12C/AMD64/64GB	12	AMD64	keine GPU	64 GB	nein
12C/AMD64/64GB/lGPU	12	AMD64	RTX2060, Mittelklasse	64 GB	nein
12C/AMD64/64GB/sGPU	12	AMD64	RTX2080, Oberklasse	64 GB	nein
12C/ARM64/64GB	12	**ARM**64	keine GPU	64 GB	**ja**
4C/ARM64/4GB	4	ARM64	keine GPU	4 GB	nein
NeuroECU/oGPU	6	AMD64	keine GPU	8 GB	nein
NeuroECU/GPU	6	AMD64	Pascal, Einstiegsklasse	8 GB	nein

Tabelle 6.1: Hardwareplattformen für die Benchmarks

In Tabelle 6.1 sind die so untersuchten Hardwareplattformen dargestellt. Dabei sind diese auf zwei Prozessorarchitekturen aufgeteilt: *ARM64* und *AMD64*. Erstere (beziehungsweise ihre 32 Bit-Varianten) wird häufiger im Embedded-Bereich wie in Fahrzeugen eingesetzt, da entsprechende CPUs häufig günstiger und mit niedrigerem absolutem Energieverbrauch verfügbar sind. Letztere wird im IT-Bereich, zum Beispiel in Desktopsystemen und Servern, eingesetzt und hat üblicherweise eine höhere Performanz und bei entsprechender Auslastung eine bessere Energieeffizienz.

Daher sind die AMD64-Plattformen in diesem Vergleich auch mit wesentlich mehr RAM und CPU-Kernen ausgestattet. Bei den ARM64-Plattformen kommen zwei besondere Exemplare zum Einsatz: Zum einen die *NeuroECU* (s. Abschnitt 5.1.1 auf Seite 132) zum anderen ein *Raspberry Pi 4* mit 4 GB RAM, der die reale ARM-ECU in Abbildung 5.14 auf Seite 146 und damit den intelligenten Sensor zur Evaluation von Use-Case 2: *Austausch von Komponenten* darstellt (s. Abschnitt 6.2 auf Seite 179).

Diese Aufteilung entspricht dem Trend von verteilten Fahrzeugarchitekturen hin zu zentralisierten (s. Abschnitt 3.1.3 auf Seite 46): Dabei werden nicht mehr einzelne Funktionen auf ihren eigenen, dafür vergleichsweise leistungs-

schwachen Steuergeräten ausgeführt, sondern mehrere Softwarekomponenten laufen gemeinsam auf wenigen, dafür leistungsstarken ECUs.

Um einen Vergleich der Leistungsfähigkeit der Emulation unternehmen zu können, wurde außerdem eine ARM64-Plattform auf der AMD64-Basis emuliert, daher ist diese mit wesentlich mehr RAM und CPU-Kernen als die realen ARM64-Plattformen ausgestattet.

Zur Evaluation der Leistung in für das automatisierte Fahren relevanten Bereichen auf den unterschiedlichen Plattformen und in der Emulation wird MLPerf (siehe Abschnitt 2.3.3 auf Seite 35) eingesetzt. Dabei wird die Leistungsfähigkeit in der Objekterkennung untersucht, zusätzlich wird das leichte offline-Szenario ausgeführt.

Für die Bewertung wurde ein Docker-Image mit einer modifizierten Version von MLPerf erstellt, jeweils für ARM64 und AMD64 getrennt. So kann jedes einzelne Image sowohl nativ als auch emuliert auf den verschiedenen Plattformen ausgeführt werden, sowohl mit als auch ohne GPU-Beschleunigung. Damit lassen sich Aussagen über den Einfluss der Emulation auf die Performanz und damit ihre Anwendbarkeit treffen.

Anpassungen sind vor allem für die NeuroECU notwendig, da sie die in MLPerf verwendete Version (1.14) von TensorFlow (s. Abschnitt 2.3.1 auf Seite 32) nicht unterstützt. Daher wird die ähnlichste kompatible Version (1.15) eingesetzt. Aufgrund der limitierten Ressourcen (vor allem RAM) des Jetson TX2 muss außerdem die maximale Anzahl der gleichzeitig verarbeiteten Bilder (Batchsize) bei *Resnet50* auf 16 statt der standardmäßigen 128 gesetzt werden. Dabei bleiben die Ergebnisse vergleichbar; leistungsfähigere Hardware würde aber gegebenenfalls durch zusätzlichen Overhead nicht optimal ausgelastet (s. auch Abschnitt 6.1.2 auf der nächsten Seite). Während der Laufzeit des Benchmarks wurden alle aus Energiespargründen standardmäßig deaktivierten Kerne aktiviert und auf den höchstmöglichen Takt gesetzt (`nvpmodel -m0`).

Für die AMD64-Plattform muss eingestellt werden, dass nicht der gesamte zur Ausführung der neuronalen Netze benötigte Speicher der GPU schon zu Beginn allokiert wird (`export TF_FORCE_GPU_ALLOW_GROWTH=true`), damit der Benchmark fehlerfrei durchläuft.

6.1.2 Effizienz

Bereits ohne den Vergleich der verschiedenen Architekturen und ohne dass der Einfluss von Emulation oder Containerisierung berücksichtigt wird, zeigen sich einige Auffälligkeiten.

In Tabelle 6.2 sind die beiden Varianten *leicht* und *schwer* des MLPerf-Benchmarks gegenübergestellt. Erstere bildet die Ausführung auf leistungsschwacher Hardware wie zum Beispiel Smartphones nach, letztere eine Berechnung auf leistungsstarker Hardware wie Workstations.

	leichte Szenarien	schwere Szenarien
Neuronales Netz	Single-Shot-Detektor (SSD) + MobileNet-v1-1.0	SSD + ResNet34
Datensatz	COCO, 300 ×300 Pixel	COCO, 1200 ×1200 Pixel
Zielintervall Multistream	50 ms, 20 Bilder/s	66,67 ms, ca. 15 Bilder/s
Gleichzeitige Bilder Multistream	4	4

Tabelle 6.2: Verwendete Einstellungen und Varianten der MLPerf-Szenarien

Betrachtet man die Ausführung der *leichten* Versionen der Szenarien (Tabelle A.10 auf Seite 210, Tabelle A.12 auf Seite 212 und Tabelle A.14 auf Seite 214), zeigt sich, dass die Ausführung auf der AMD64-Plattform *ohne* GPU-Beschleunigung im leichten Szenario durchgehend schneller ist, während sie bei der *NeuroECU* wie erwartet langsamer ist. Zusätzlich unterscheiden sich die Ergebnisse der verschiedenen leistungsfähigen GPUs auf der AMD64-Plattform nur geringfügig.

In den schweren Szenarien (Tabelle A.11 auf Seite 211 und Tabelle A.13 auf Seite 213) sind die Ergebnisse hingegen weniger überraschend: Das Nutzen einer GPU beschleunigt die Ausführung stark, ebenso werden die unterschiedlichen Leistungsklassen der GPUs sichtbar. Für die AMD-Plattformen sind diese Ergebnisse noch einmal in Abbildung 6.1 dargestellt.

Wie schon in Abschnitt 6.1.1 auf Seite 171 angesprochen, deutet das auf eine geringe Effizienz der leichten Szenarien hin. Das in den leichten Varianten verwendete neuronale Netz (*MobileNet-v1-1.0*) ist auf die Ausführung auf leistungsschwachen Geräten wie Smartphones optimiert. Diese besitzen aus Energieeffizienzgründen üblicherweise wenige CPU-Kerne und sparsame GPUs mit vergleichsweise wenigen Ausführungseinheiten. Daher kann hier nicht von der hohen Anzahl an CPU-Kernen oder den leistungsfähigen GPUs profitiert werden. Diese These wird dadurch untermauert, dass CPU- und GPU-Auslastung in den schweren Szenarien sehr viel höher ist (nicht in den Tabellen abgebildet), die Hardware wird nahezu vollständig ausgelastet.

Dies ist jedoch für das in dieser Dissertation vorgestellte Konzept einer (re-) konfigurierbaren Architektur kein Ausschlusskriterium. Im Gegenteil kann die hier mögliche parallele Ausführung in mehreren getrennten Containern die Ausnutzung der verfügbaren Ressourcen sogar stark erhöhen.

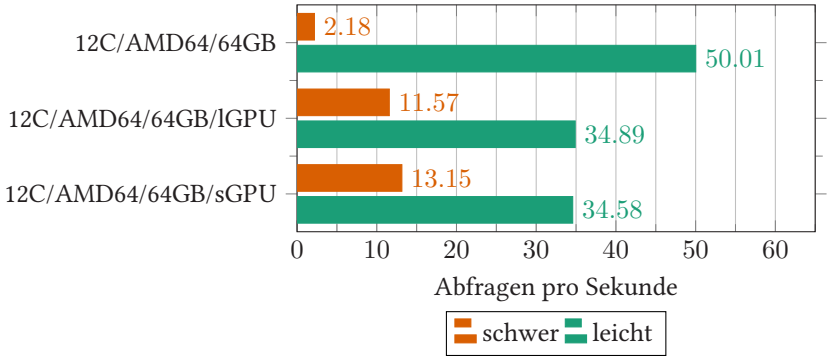


Abbildung 6.1: Vergleich der Ergebnisse der AMD64-Plattformen für die Singlestream-Szenarien

6.1.3 Auswirkungen der Containerisierung

Auch beim Vergleich der Ausführung in Docker oder direkt auf dem System wiederholt sich das Bild der geringen Effizienz der leichten Szenarien. Hier ist der Zuwachs an Geschwindigkeit bei direkter Ausführung deutlich höher (+14,1% Abfragen pro Sekunde *Singlestream leicht*, -11,73% mittlere Latenz *Multistream leicht*) als in den schweren Szenarien (+2,77% Abfragen pro Se-

kunde *Singlestream schwer*, -3,67% mittlere Latenz *Multistream schwer*). In Abbildung 6.2 ist das für Singlestream leicht dargestellt, in Abbildung 6.3 für Singlestream schwer.

Insgesamt ist also der durch Docker hinzukommende Overhead zu vernachlässigen, da er nur bei schlecht optimierter Software einen nennenswerten Anteil erreicht.

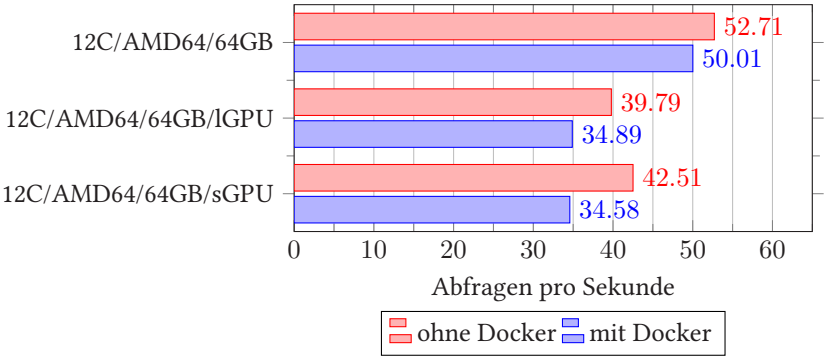


Abbildung 6.2: Vergleich der Auswirkungen der Containerisierung auf Singlestream leicht (AMD64)

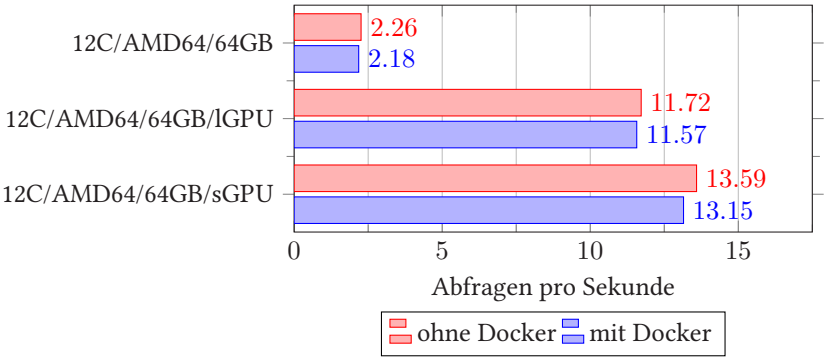


Abbildung 6.3: Vergleich der Auswirkungen der Containerisierung auf Singlestream schwer (AMD64)

6.1.4 Auswirkungen der Emulation

Weiterhin zeigt sich, dass die Emulation zur Laufzeit nur in Fällen, in denen es entweder essentiell ist, dass die absolut identische Software ausgeführt wird, oder in solchen, bei denen eher einfache Algorithmen ausgeführt werden sollen, eine nutzbare Performanz erzielt. Dies ergibt sich auch daher, dass der RAM-Verbrauch durch die Emulation extrem zunimmt. Im besten Fall (in den *Singlestream*-Szenarien und im Szenario *Offline leicht*) ist die Ausführung auf der langsamsten nicht-emulierten ARM64-Plattform, dem Raspberry Pi 4 (4C/ARM64/4GB), „nur“ einige Hundert Prozent schneller, sonst durch den explodierenden Speicherverbrauch (*Multistream leicht*) oder eine extrem lange Laufzeit (*Multistream schwer*) gar nicht erst möglich. Verglichen mit der nativen Ausführung beträgt der Faktor sogar mehrere Hundert bis Tausend.

Eigenschaft	Vorteile	Nachteile
Emulator als Zwischenebene	Abstraktion von der Hardware	Performanzverlust Zusätzliche Fehlerquelle ggfs. werden nicht alle Komponenten der Hardwareplattform durch den Emulator unterstützt
Exakt derselbe Binärcode wird ausgeführt	Code ist ggfs. bereits abgesichert Code muss nicht neu kompiliert werden	Keine Optimierungen auf die Zielarchitektur
Zielarchitekturen identisch	auch neue, zukünftige Hardwareplattformen können unterstützt werden	von neuen Hardwarefeatures kann nur bedingt profitiert werden

Tabelle 6.3: Vor- und Nachteile der Emulation

Tabelle 6.3 zeigt einige der wichtigsten Vor- und Nachteile der Emulation auf. So sorgt der Emulator als Zwischenebene für eine zusätzliche Abstraktion von der Hardware, es ist also nur die Anpassung auf exakt eine Zielplattform notwendig. Dadurch, dass die Zielarchitekturen identisch sind, können auch zukünftige auf den Markt gebrachte Hardwareplattformen unterstützt werden.

Jedoch ist diese Emulatorebene immer auch eine zusätzliche Fehlerquelle (zum Beispiel das Speicherleck beim leichten Multistream-Szenario) und außerdem beeinflusst sie die Performanz sehr stark. Zusätzlich muss so auch eine bestimmte Grundhardwareplattform als Zielplattform für die Kompilierung von Code definiert werden, wodurch Fortschritte in der Entwicklung wie neue Hardwarefeatures (zum Beispiel die Beschleunigung von Verschlüsselungsalgorithmen in Hardware) nur begrenzt genutzt werden können.

Da überall exakt derselbe Binärcode ausgeführt wird, kann eventuell ausgenutzt werden, dass dieser schon abgesichert ist, ohne eine vollständige Neuabsicherung für die neue Hardwareplattform durchführen zu müssen. Das Vermeiden der Neukompilierung kann zwar einmalig einige Zeit einsparen, allerdings wird diese eingesparte Zeit meist durch schlechtere Performanz durch die Emulation oder auch das Fehlen von Optimierungen auf die Zielprozessorarchitektur zunichte gemacht.

6.1.5 Auswirkungen Neukompilierung

Die Neukompilierung hat den Vorteil, dass durch den Compiler Optimierungen für die Zielplattform durchgeführt werden können. Allerdings ist der Code in den seltensten Fällen für verschiedene Architekturen ohne Anpassungen direkt kompilierbar. Daher müssen diese Anpassungen in den Zeit- und Ressourcenbedarf eingerechnet werden. Die eigentliche Kompilierung verlängert sich üblicherweise proportional zur Anzahl der unterstützten Zielplattformen und fällt insgesamt wenig ins Gewicht. Jedoch spielt dabei eine Rolle, wo die Neukompilierung durchgeführt wird: für einen Flottenbetreiber kann es durchaus relevant sein, wenn dies auf jedem einzelnen Fahrzeug geschieht anstatt beim Hersteller. Zusätzlicher Vorteil ist, dass Hardwaretreiber wie für GPUs integriert werden können. Insgesamt zeigen auch die Ergebnisse, dass in den meisten Fällen eine Neukompilierung notwendig ist.

6.2 Evaluation Performanz Use-Case 2: Austausch von Komponenten

Ein weiterer kritischer Punkt für die Anwendbarkeit des Konzeptes der (re-) konfigurierbaren Fahrzeugarchitektur ist die tatsächliche Rekonfiguration. Ein zeitkritischer Teil davon ist das Plug-and-Play-System aus Use-Case 2: Austausch von Komponenten. Eine detailliertere Evaluation befindet sich in [SGS+21].

Um die Funktion des Systems zu zeigen und zu bewerten, wurde ein Versuchsaufbau aus zwei *Raspberry Pi 4* mit jeweils 4 GB RAM (s. Abschnitt 6.1.1 auf Seite 171, Plattform *4C/ARM64/4GB*) und identischen Kameras erstellt (s. Abbildung 6.4 auf der nächsten Seite). Der einzige Unterschied ist hier, dass einer dieser Sensor-Nodes ein Schwarzweißbild liefert (Sättigung 0), der andere ein farbiges (Sättigung 100). Damit entsprechen diese ECUs jeweils der realen ARM-ECU aus Abbildung 5.14 auf Seite 146, die gegen eine aktuellere Version ausgetauscht wird beziehungsweise ausfällt.

Da das Umschalten der Fähigkeiten der Kamera-Nodes wie gewünscht funktioniert, wird für die weitere Evaluation der Umschaltezeiten die Übertragung des Kamerabildes eingestellt und stattdessen Timestamps übertragen, mittels derer die genauen Laufzeiten bestimmt werden können.

Dazu läuft der Node mit der geringeren Sättigung, also dem Schwarzweißbild, durchgehend und dient als Fallback für den Farbkamera-Node. Letzterer wird anschließend mehrfach neu gestartet und sendet dabei die Timestamps. Ein Consumer-Node schaltet entsprechend auf den Node mit höherer Sättigung sobald verfügbar und zurück, sobald kein Heartbeat von diesem mehr empfangen wird. Am Ende werden die gemessenen Zeiten gemittelt und die Standardabweichung berechnet.

Dies wird für verschiedene „Bildraten“ durchgeführt, die Timestamps also mit unterschiedlicher Frequenz gesendet. Außer den beiden Fällen 5 Hz und 15 Hz wird untersucht, wie sich ein Senden mit doppelter Sendefrequenz im Vergleich zur Empfangsfrequenz auswirkt. Hiermit werden Service-Konsumenten nachgebildet, die nicht die volle Nachrichtenfrequenz des Service-Anbieters benötigen. Die Zyklendauer der Zustandsmaschine der Nodes liegt dabei bei einem Fünftel der Nachrichtenperiode.

Abbildung 6.4: Versuchsaufbau zur Evaluation des Plug-and-Play-Konzeptes

6.2.1 Wechsel auf einen besseren Sensor

Sobald ein neuer Sensor integriert wird, soll auf diesen umgeschaltet werden, sobald er eine Fähigkeit „besser“ erfüllt als der bereits verwendete. Diese Entscheidung basiert auf zuvor definierten Metadaten der Fähigkeit. Im untersuchten Fall ist das die Farbsättigung. Dabei gibt es zwei Zeitspannen, die von Bedeutung sind: Zum einen die Dauer, bis der Sensor-Node bereit ist, sich also im Zustand *Aktiv* befindet (Zustände s. auch Abschnitt 4.2.7 auf Seite 112, Zeit in den einzelnen Zuständen in Tabelle A.15 auf Seite 215). *Ohne Zustand* ist dabei die zur Starten benötigte Zeit des Sensor-Nodes, bis sich dieser im Zustand *Unkonfiguriert* befindet. Die zweite Zeitspanne ist die Dauer bis der Konsument auf diesen neuen Node umgeschaltet hat, gemessen ab Erreichen des Zustands *Aktiv* des Sensor-Nodes (Mittelwert und Standardabweichung σ in Tabelle A.15 auf Seite 215, in Klammern ist jeweils die Anzahl von Bildperioden angegeben, der die Zeitspanne vor der Klammer entspricht.).

Dabei fällt auf, dass die zum Starten und in den unterschiedlichen Zuständen benötigte Zeit bei Erhöhung der „Bildrate“ kürzer wird. Das liegt daran, dass die Zyklendauer der Zustandsmaschine an diese gekoppelt ist, in diesem

Fall um den (konfigurierbaren) Faktor fünf schneller ist. Außerdem nimmt entsprechend auch die Zeit ab, die der Konsument benötigt, bis zum einen der neue Sensor-Node entdeckt wurde und zum anderen auf diesen umgeschaltet wurde. Im Schnitt ist die Zeitspanne zum Wechsel auf den besseren Sensor kurz genug, dass kein Bild verloren geht.

6.2.2 Wechsel bei Ausfall eines Sensors

Wichtiger als der Wechsel auf einen besseren Sensor ist für die Zuverlässigkeit eines Fahrzeuges der Wechsel im Falle des Ausfalls eines Sensors. Da der Konsument zuerst fünf verlorene Heartbeats abwartet (und die Zyklendauer und damit auch $f_{Heartbeat}$ auch hier von der Bildrate abhängt), nimmt die benötigte Zeit mit wachsender Bildrate ab. Dieses Verhalten ist in Tabelle A.16 auf Seite 215 dargestellt. In Klammern ist jeweils die Anzahl von Bildperioden angegeben, der die Zeitspanne vor der Klammer entspricht. Diese Umschaltezeit ließe sich mit dem Reduzieren der tolerierten ausgefallenen Heartbeats senken, allerdings steigt damit auch die Rate der fälschlicherweise als ausgefallen angenommenen Sensor-Nodes, sobald Heartbeats ihr Ziel nicht oder zu spät erreichen. Insgesamt geht im Schnitt ein Bild beim Wechsel auf einen anderen Sensor verloren, was jedoch in den meisten Fällen tolerierbar ist.

In beiden Fällen liegen die Wechselzeiten ab $f_{Heartbeat} = 15\,\text{Hz}$ durchgehend unter den in Abschnitt 3.7.3 auf Seite 95 geforderten Rekonfigurationszeiten bei einem Ausfall von Lenkung (130 ms) oder Bremse (160 ms), liegen also auf jeden Fall im akzeptablen Rahmen. Dies ersetzt allerdings keine, für den Praxiseinsatz benötigte, umfassendere Untersuchung der Echtzeiteigenschaften, die nicht im Fokus dieser Dissertation stand.

6.3 Diskussion der Ergebnisse und Praxisrelevanz des Ansatzes

Die Abdeckung der in Abschnitt 1.6 auf Seite 8 aus den Herausforderungen (s. Abschnitt 1.2 auf Seite 1) hervorgegangenen Anforderungen ist in den Tabellen A.17 (Adaptierbarkeitsanforderungen, Seite 216), A.18 (Portabilitätsanforderungen, Seite 217) und A.19 (Sicherheitsanforderungen, Seite 218)

dargestellt. Dabei zeigt sich, dass alle Anforderungen durch den in dieser Dissertation vorgestellten Ansatz abgedeckt werden.

Auch wenn die Ergebnisse der Evaluation nicht allgemeingültig sind und die Akzeptanz der Fahrzeughersteller sowie die Einbindung in (bestehende) Entwicklungsprozesse noch getrennt untersucht werden müssen, wird gezeigt, dass der Ansatz der (re-)konfigurierbaren Architektur prinzipiell funktioniert.

So lassen sich Komponenten dynamisch hinzufügen und entfernen, das Hinzufügen neuer Funktionalität in Form von Softwarekomponenten eingeschlossen. Die Verwendung von standardkonformer COTS-Hardware, um zentralisiert die benötigte Rechenleistung skalierbar zur Verfügung zu stellen, erlaubt in Verbindung mit über ein Netzwerk angebundenen intelligenten Sensoren und Aktoren eine Anpassbarkeit der Architektur über den gesamten Lebenszyklus des Fahrzeuges hinweg.

Damit ist die Anwendbarkeit nicht auf die in ihrem Umfang stark beschränkte und auf dem Basisfahrzeug *Nora* aufbauende Demonstrations- und Evalutionsplattform aus dieser Dissertation begrenzt. Der Orchestrator *Kubernetes* ermöglicht eine Skalierung bis hin zu mehreren Tausend ECUs in einem Fahrzeug, was auch in Zukunft für jegliche Fahrzeugarchitektur genügen sollte, besonders da die Entwicklung hin zu zentralisierten Plattformen die Anzahl an Steuergeräten in einem Fahrzeug eher stark reduzieren wird.

Als Beispiel für eine Anwendung der (re-)konfigurierbaren Fahrzeugarchitektur in Bussen sei die Cloud-basierte Klimatisierung der Fahrzeuge (s. Abbildung 6.5 auf der nächsten Seite) genannt. In [BLS+20] stellen Böhme et. al. hierzu einen auf maschinellem Lernen basierenden Ansatz vor, mit dem besonders bei elektrisch angetriebenen Fahrzeugen durch auf Vorhersagen basierende effizientere Klimatisierungsstrategien die Reichweite optimiert werden kann. Eine Vorstellung und Evaluation weiterer Regleralgorithmen erfolgt in [SJRS21]. Dazu wird bei der Cloud-basierten Klimatisierung die Solltemperatur der Fahrzeuge einer Flotte zentral im Backend aus Daten wie Wetter, Anzahl und Art der Fahrgäste im Fahrzeug oder Fahrtstrecke berechnet. Je nach Sonnenstand, Jahreszeiten oder auch wetterbedinger Bekleidung unterscheiden sich die Wohlfühltemperaturen der Passagiere; Schüler haben gegebenenfalls eine andere Wunschtemperatur als ältere Personen. Dabei kann eine Anpassung sogar feingranular erfolgen und das Fahrzeug in unterschiedliche „Klimazonen“ unterteilt werden. Hierfür muss jedes Fahrzeug

mit dem Backend kommunizieren können: Das Fahrzeug sendet die Passagieranzahl und bekommt im Gegenzug die individuelle Solltemperatur zurück. Diese und weitere Informationen können dem Fahrer auf einem Display im Kombiinstrument angezeigt werden.

Insgesamt zeigt dieses Beispiel damit mehrere Aspekte des Ansatzes dieser Dissertation auf. So können bestehende Busse (über die gesamte in Abbildung A.1 auf Seite 189 gezeigte Variantenvielfalt hinweg) durch zusätzliche Sensorik (Kameras zur Fahrgastzählung), Steuergeräte (zur Ausführung der Algorithmen zur Fahrgastzählung) und Aktorik (Informationsdisplay für den Fahrer) für die Cloud-basierte Klimatisierung bereit gemacht werden. Da günstige COTS-Komponenten aus der IT wie zum Beispiel Smartphonedisplays oder Standard-Industrie-PCs genutzt werden können, kann sich ein solches Vorgehen auch bei relativ geringen Stückzahlen lohnen, da keine teuren Spezialkomponenten mit besonders langer Lieferbarkeit benötigt werden. Das ist entsprechend auch ein Vorteil bei einer Fahrzeugneuentwicklung. Gleichzeitig können die Services, die Bestandteil der Cloud-basierten Klimatisierung sind, als Basis für weitere auf Kundenwunsch hinzuzufügende Funktionalitäten dienen. So können die Kameras zur Fahrgastzählung durch intelligente Algorithmen, die auf einem geeigneten Steuergerät im Fahrzeug ausgeführt werden, auch der selbstständigen Erkennung von Vandalismus dienen.

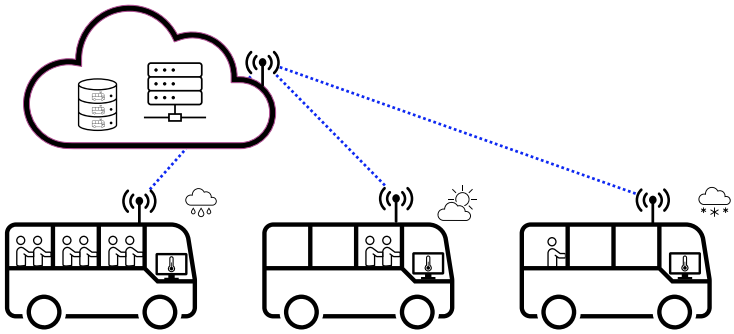


Abbildung 6.5: Cloud-basierte Klimatisierung

7 Zusammenfassung und Ausblick

In dieser Dissertation wurden Herausforderungen identifiziert, denen sich Fahrzeughersteller aufgrund unterschiedlicher Garantiezeiträume und Lebenszyklen der Fahrzeuge sowie zunehmend kürzerer Innovationszyklen der Technologie stellen müssen. Dies ist zum Beispiel das Nutzen von Standardkomponenten, um Kosten zu sparen, aber auch eine lange Verfügbarkeit sicherzustellen. Letztere ist durch kurze Lebenszyklen in der Informationstechnik eine zunehmende Herausforderung, daher müssen während des Lebenszyklus eines Fahrzeuges auch Anpassungen aufgrund nicht mehr verfügbarer Komponenten durchgeführt werden können. Zusätzlich lassen sich so auch Erweiterungen der Funktionalität durchführen, sei es auf Kundenwunsch oder aufgrund geänderter gesetzlicher Randbedingungen.

Herkömmliche und eher statische signalbasierte Architekturen stoßen hier an ihre Grenzen, flexibel konfigurierbare serviceorientierte Architekturen schaffen hier Abhilfe. Dabei wurden einige mögliche Realisierungen miteinander verglichen und Vor- und Nachteile gegenübergestellt.

Der Stand der Technik an verfügbaren Fahrzeugarchitekturen wurde bezüglich der Abdeckung dieser Herausforderungen untersucht. Dabei zeigte sich die Automobilindustrie traditionell verschlossen, einzig Volvo wagte den Versuch einer öffentlichen standardisierten Fahrzeugarchitektur. Verschiedene Forschungsprojekte wurden vorgestellt, von denen jedoch keines alle identifizierten Herausforderungen abdeckt. Dies liegt auch an divergenten Zielsetzungen dieser Projekte. Daher ist der Ansatz der (re-)konfigurierbaren Fahrzeugarchitektur, wie er in dieser Dissertation vorgestellt wurde, einzigartig.

Verschiedene grundsätzliche Anforderungen an eine solche Fahrzeugarchitektur wurden aufgestellt, eingeteilt in die Kategorien Anforderungen an die Adaptierbarkeit, an die Portabilität und an die Sicherheit (sowohl Safety als auch Security). Anschließend wurde ein Konzept entwickelt, das zum

einen die Herausforderungen als auch diese Anforderungen erfüllen kann. Dieses basiert auf dem Orchestrator Kubernetes, mit einer serviceorientierten Architektur auf Basis von ROS2 und anderen frei verfügbaren Produkten. Komponenten und Fähigkeiten können dynamisch hinzugefügt, entfernt und in das Backend verlagert werden. Dabei spielt es keine Rolle, welche Prozessorarchitektur die zugrundeliegende Hardwareplattform besitzt.

Damit werden die Fortschritte der Informationstechnik ausgenutzt, indem leistungsfähigere und zentrale Steuergeräte zum Einsatz kommen können, die voneinander abgekapselte Softwarekomponenten ausführen und damit dynamisch Redundanzen oder Leistungsreserven durch eine parallele Ausführung schaffen können. Außerdem können so vormals der Serverwelt vorbehaltene Technologien im Fahrzeug genutzt werden, wie in dieser Dissertation gezeigt wurde.

Dazu wurden Use-Cases identifiziert, die durch eine (re-)konfigurierbare Fahrzeugarchitektur erst ermöglicht beziehungsweise zumindest stark vereinfacht werden. Als Basisfahrzeug *Nora* für die Umsetzung und Evaluation der (re-)konfigurierbaren Fahrzeugarchitektur wurde ein elektrisch angetriebener Aufsitzrasenmäher mittels neu hinzugefügter Sensorik und Aktorik bereit für automatisiertes Fahren gemacht. Dabei wurde die Grundfunktionalität in ROS umgesetzt. Die (re-)konfigurierbare Fahrzeugarchitektur wurde auf einem leistungsfähigen Virtualisierungshost umgesetzt, wobei die einzelnen ECUs teilweise virtualisiert wurden, um den Einfluss der Prozessorarchitektur auf die Performanz und die Unabhängigkeit des Ansatzes davon untersuchen zu können.

Bei der anschließenden Evaluation der (re-)konfigurierbaren Fahrzeugarchitektur zeigte sich, dass eine solche lauffähig ist und eine nutzbare Performanz liefert. Durch die dynamische Verteilung von Softwarekomponenten konnte die Auslastung von Ressourcen durch an sich wenig effiziente Algorithmen optimiert werden, außerdem wurde gezeigt, dass ein Umschalten auf Fähigkeiten, wie zum Beispiel das Liefern eines Bildes der Umgebung, mit praxistauglichen Umschaltzeiten möglich ist. Besonders wichtig ist dies, wenn der Ausfall eines Sensors kompensiert werden muss, indem auf einen anderen Sensor umgeschaltet wird.

Die Auslagerung von Services in das Backend wurde prototypisch dargestellt und eine mögliche Umsetzung der Kommunikation über VPN untersucht.

Zur Darstellung der Anwendbarkeit auf bereits bestehende Fahrzeuge beziehungsweise die Möglichkeit, bestehende Komponenten und Architekturen weiterverwenden zu können, wurde eine bereits bestehende signalbasierte Architektur auf eine serviceorientierte umgestellt. Dabei wurde ein Prozess gezeigt, mit dem dies mithilfe eines sogenannten Technologie-Gateways erfolgen kann.

In Zukunft könnte über diese Dissertation hinaus die Umsetzbarkeit bezüglich der Freigabe detaillierter untersucht werden. Teil eines Übergangs in Richtung Freigabe eines fertigen Produktes wäre die Untersuchung der Echtzeiteigenschaften der (re-)konfigurierbaren Fahrzeugarchitektur. Erster Schritt wäre dabei der Umstieg auf ein echtzeitfähiges Betriebssystem anstatt Linux, da nur mit einem solchen die Erfüllung von harten Echtzeitanforderungen garantiert werden kann. Außerdem müsste gezeigt werden, dass auch alle Safety-relevanten Komponenten der (re-)konfigurierbaren Fahrzeugarchitektur wie zum Beispiel der Orchestrator und damit das Umschalten auf redundante Services nicht nur unter Laborbedingungen sondern auch im Realbetrieb unter allen Umständen die Echtzeitanforderungen erfüllen. Auch die garantierte Sicherstellung der Datenkonsistenz zwischen einzelnen Services bietet Stoff für weitere Untersuchungen. Zusätzliche Herausforderungen bei der Freigabe ergeben sich dadurch, dass wesentlich mehr Freiheitsgrade existieren als zuvor und eine Determiniertheit nicht mehr zwingend gegeben ist.

Im Rahmen der Entwicklung eines marktreifen Produktes würde auch der entsprechende Entwicklungsprozess an die durch eine (Re-)Konfigurierbarkeit entstehenden neuen Möglichkeiten und Herausforderungen angepasst werden, was eine entsprechende Akzeptanz bei den Herstellern vorraussetzt. Um diese zu erreichen, muss die Übertragbarkeit des Ansatzes auf ein industrielles Produkt in Form eines PKW oder Busses gezeigt werden, worunter auch die Umsetzung und das Management der Fahrzeugvarianten des Produktportfolios fallen. Teil dieser Akzeptanz und Übertragbarkeit ist auch eine Untersuchung der Wirtschaftlichkeit. Es ist nicht von der Hand zu weisen, dass der in dieser Dissertation gezeigte Ansatz gewisse Anforderungen an die Rechenleistung stellt (s. Abschnitt 6.1.2 auf Seite 174 und folgende), die sich in entsprechend höheren Hardwarekosten niederschlagen können. Dies kann zum einen durch die Verwendung von günstigeren COTS-Komponenten und zum anderen durch gesenkte Entwicklungskosten abgefangen werden, muss aber gegebenenfalls individuell untersucht werden. Zu dieser Untersu-

chung gehört entsprechend auch die Bestimmung der tatsächlich benötigten Hardwareressourcen und die Beurteilung der Realisierbarkeit als (re-)konfigurierbare Fahrzeugarchitektur. Ebenso könnten weitere Untersuchungen mögliches Potenzial bezüglich weiterer Optimierung der Ressourcennutzung aufzeigen.

Erst durch die Verbreitung von mit der (re-)konfigurierbaren Fahrzeugarchitektur ausgestatteten Fahrzeugen würde sich außerdem das Potenzial zeigen, indem Daten aus der Flotte für Cloud-basierte und maschinelles Lernen verwendende Dienste zur Verfügung stünden. Dies schließt auch ein, dass eine entsprechende Infrastruktur aufgebaut werden muss. Zu diesem Zweck müsste auch die Skalierung des Konzeptes im Backend und auf größere Netzwerke untersucht werden.

A Anhang

A.1 Grafiken

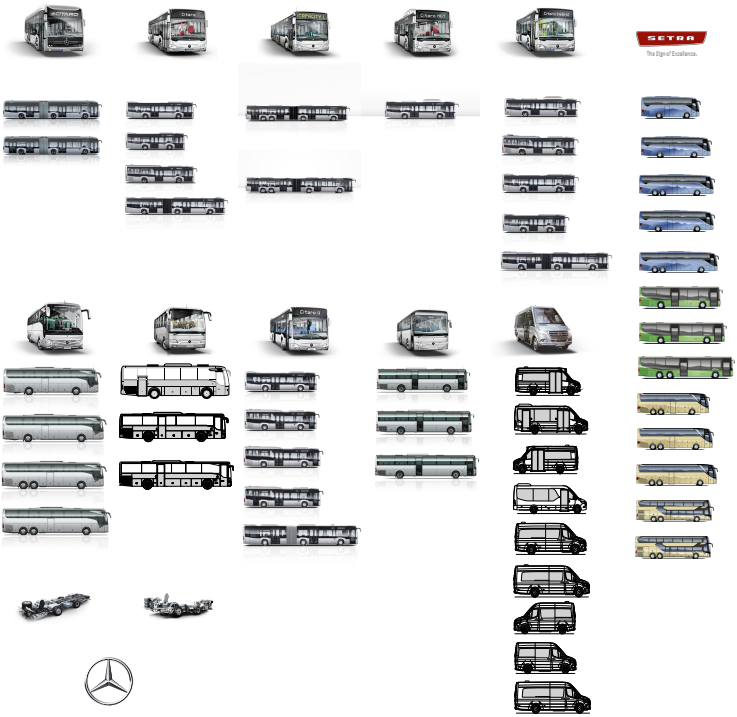


Abbildung A.1: Variantenüberblick der EvoBus-Marken Mercedes-Benz (links) und Setra (rechte Spalte) [Mer20] [Set20]

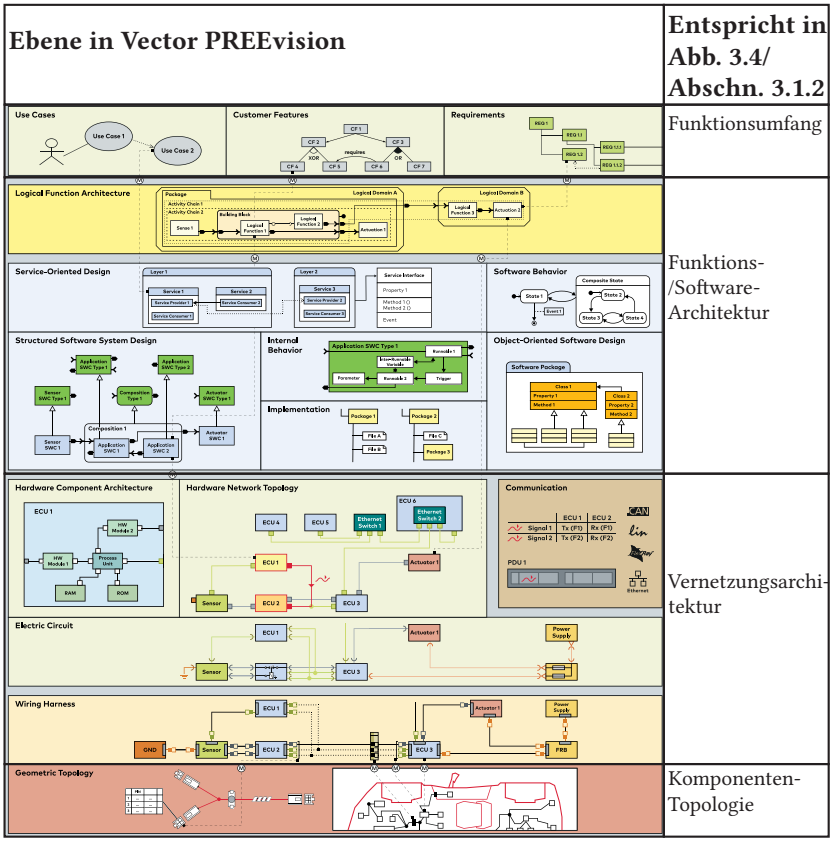


Abbildung A.2: Die Ebenen der E/E-Architektur in Vector PREEvision [Vec19]

A.2 Weitere Grundlagen und Stand der Technik

A.2.1 Quellen für Daten zum Training von Algorithmen zur Objekterkennung

Um die verwendeten Algorithmen zu untersuchen und neuronale Netze zu trainieren oder andere Verfahren des maschinellen Lernens anzuwenden, sind große Mengen von (idealerweise schon gelabelten, das heißt vorklassifizierten und mit einer für den Algorithmus verständlichen Bedeutung versehenen) Daten notwendig. Diese ausschließlich selbst zu erzeugen und aufzuzeichnen ist zeitintensiv und durch bereits vorhandene Quellen nicht notwendig.

Es existiert ein breites Angebot an Datensätzen, die neben Kamera- und Lidar-Daten und weiteren Sensordaten auch eine Ground Truth enthalten, die zur Bewertung der Ergebnisse eigener Algorithmen genutzt werden kann.

Die wichtigsten Datensätze werden im Folgenden in chronologischer Reihenfolge ihrer Veröffentlichung vorgestellt und im Anschluss kurz verglichen. Konkret sind das: ImageNet ([DDS+09], 2009), KITTI ([GLSU13], 2013), Daimler Urban Segmentation ([SEFR13], 2013), Microsoft COCO ([LMB+14], 2014) und Cityscapes ([COR+16], 2016).

ImageNet

ImageNet [DDS+09] ist mit rund 14,2 Mio. annotierten Bildern eine der größten frei verfügbaren Bilddatenbanken [Ima10]. Davon sind ca. 1 Mio. Bilder mit Bounding Boxes (Abbildung 2.5 auf Seite 31) versehen, die die Position der erkannten Objekte angeben, anstatt nur Auskunft darüber zu geben, dass ein Objekt auf dem Bild vorhanden ist. Diese Fotos sind in ca. 22k sogenannter **synsets** unterteilt, eine verbreitete Abkürzung von engl. „synonym set“. Dies sind Wortfelder, die Synonyme zusammenfassen. Dabei ist jedoch zu beachten, dass Fotos aus beliebigen Kategorien enthalten sind, die meisten sich daher nicht für den Straßenverkehr nutzen lassen (zum Beispiel, zumindest in den meisten Situationen, Tiere oder Obst). Besonders wird mit einer hohen Qualität der Labels geworben und die Precision mit im Schnitt 99,7 % angegeben.

	#Seq.	Länge	Anmerkungen
Straße	12	600 s	hauptsächlich Landstraßen und Autobahn
Stadt	28	840 s	
Wohngebiet	21	2820 s	
Campus	10	120 s	Campus des KIT
Personen	80	420 s	wenige verschiedene Personen in immer derselben Umgebung, typische Fußgängerbewegungen
Gesamt	151	4800 s	

Tabelle A.1: Kategorien der KITTI-Sequenzen

KITTI

Der KITTI-Datensatz legt den Fokus auf Daten für das automatisierte Fahren. Hierbei wurden, anders als bei ImageNet nicht nur 2D-Bilddaten aufgezeichnet, sondern auch Bilder von 3D-Kameras, Lidar-Daten sowie Positionsdaten mit hoher Genauigkeit. Dabei wurden möglichst diverse Alltagsfahrsituationen aufgezeichnet, von Fahrten auf der Autobahn über Landstraßen bis hin zum Stadtverkehr. [GLSU13]

Da diese Daten kalibriert, synchronisiert und mit Zeitstempeln versehen sind, können sie daher direkt zur Untersuchung verschiedener Algorithmen, die über das Klassifizieren einzelner Bilder hinausgehen, genutzt und so mit selbst aufgezeichneten Daten verglichen werden. Ein Beispiel hierfür sind Algorithmen zur Kartenerstellung. Dazu wurde das Tool „kitti2bag“ veröffentlicht, welches die Daten automatisiert in das von ROS und ROS2 unterstützte Format umwandeln kann. [Tom17]

Der Datensatz ist in lediglich fünf, dafür aber für das automatisierte Fahren relevante Kategorien eingeteilt, s. Tabelle A.1. Insgesamt ergeben sich also 48000 Bilder bei 10 Hz. Die Daten sind in Sequenzen unterteilt, jede Sequenz enthält Rohdaten und Daten zur Kalibrierung, sowie fertig nachbearbeitete Daten, bei denen zum Beispiel die Kalibrierung und Synchronisation schon durchgeführt wurden. Außerdem sind Annotationen dynamischer Objekte in Form von 3D Bounding Boxes enthalten, die wiederum in verschiedene Klassen unterteilt sind: „Auto“, „Kleinbus“, „LKW“, „Fußgänger“, „sitzende

Person“, „Fahrradfahrer“, „Tram“ sowie eine Mischklasse für andere Verkehrsteilnehmer, wie Anhänger oder Segways. Vorteil der 3D-Annotierung ist, dass die genaue Position des Objekts im Raum und die Abmessungen bekannt sind. Diese Ground Truth ermöglicht es, die Ergebnisse der Algorithmen zur 3D-Objekterkennung des Basisfahrzeuges für die Umsetzung der (re-)konfigurierbaren Fahrzeugarchitektur „Nora“ (s. Abschnitt 5.1 auf Seite 125) zu evaluieren (s. Abschnitt 5.1.3 auf Seite 136).

Für einen möglichst objektiven Vergleich der Ergebnisse verschiedener Algorithmen, stellt die KITTI-Seite basierend auf dem KITTI-Datensatz außerdem eine Benchmark-Suite zur Verfügung ([GLU12], [APCR18]). Diese enthält eine definierte Teilmenge der im Datensatz enthaltenen Daten mit möglichst hoher Diversität. Diese Teilmenge ist wiederum in Trainings- und Testdatensatz unterteilt. Ersterer dient dazu, den verwendeten Algorithmus zu trainieren, letzterer dient dazu, ihn zu evaluieren. Indem eine gemeinsame Datenbasis verwendet wird, wird eine optimale Vergleichbarkeit sichergestellt. Die Güte der Algorithmen zur 2D-Objekterkennung wird über die mAP (s. Abschnitt 2.3.2 auf Seite 33) bewertet, in 3D um eine Ähnlichkeitsbewertung der vorhergesagten Lage im Raum ergänzt.

Daimler Urban Segmentation, DUS

Dieser Datensatz ist auf den Verkehr in Städten spezialisiert. Im Jahre 2014 wurde dieser Datensatz stark erweitert auf 5000 Stereo-Bildpaare und entsprechende Tiefenbilder, von denen jedes zehnte semantisch annotiert ist (erhältlich unter [USC18]). [SEFR13] Damit ist dieser Datensatz relativ klein und wurde mittlerweile vom Cityscapes-Datensatz abgelöst.

Im DUS-Datensatz werden 3D-Bilddaten genutzt, um eine semantische Segmentierung der Szene durchzuführen. Dazu werden die Eingangsdaten in die sogenannte „Stixel-Welt“ überführt. [BFP09]

Ein Beispiel für die Stixel-Darstellung ist in Abbildung A.3 auf der nächsten Seite dargestellt: Links ist das sogenannte „Dense Disparity Image“ dargestellt, das auf Disparitätsinformationen der Stereo-Kamera basiert. Die Stixel-Welt wird dann in Regionen unterteilt und diese klassifiziert. Dabei wird die Entfernung des Bildinhaltes zur Kamera farblich kodiert, eine rote Einfärbung bedeutet eine niedrige Entfernung, grün gefärbte Bereiche sind weiter ent-

fernt. Die Stixel-Darstellung auf der rechten Seite orientiert sich mehr am erwarteten Fahrkorridor des Fahrzeuges (in blau eingezeichnet). Die sogenannten „Stixel“ stellen senkrecht auf dem Boden stehende, durch die zur Kamera relative 3D-Position definierte, Quader dar, die zusammen die Grenzen der erkannten Objekte annähern. Dabei stellen die Farben die Entfernung zum Fahrkorridor dar.

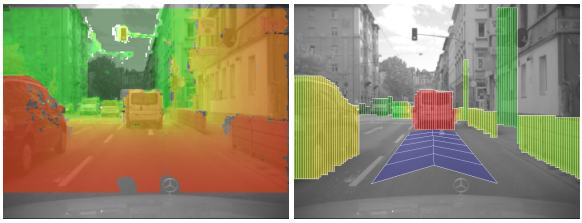


Abbildung A.3: Vergleich Dense Disparity Image mit Stixel-Repräsentation [BFP09]

Microsoft COCO

Dieser Datensatz enthält gewöhnliche Objekte aus dem alltäglichen Leben (COCO = Common Objects in Context). Dabei wird jedes einzelne Vorkommen von insgesamt 91 Objektklassen in insgesamt 328000 Fotos gelabelt, was zu ca. 2,5 Mio. gelabelten Objekten im Datensatz führt. [LMB+14] Im Vergleich zu ImageNet sind das zwar weniger Kategorien, aber dafür mehr Objekte pro Kategorie, was positiv für das Training neuronaler Netze ist. Besonderes Augenmerk wurde darauf gelegt, dass auch Objekte gelabelt werden, die sich nicht nur im Vordergrund des Bildes befinden und daher von auf ImageNet-Daten trainierten Algorithmen leicht übersehen werden. Anders als bei den älteren genannten Datensätzen (außer DUS) sind die Objekte pixelgenau und nicht nur über Bounding Boxes gelabelt. Zusätzlich sind die Szenen semantisch gelabelt, was theoretisch auch die Kategorisierung ansonsten schwer zähl- oder einordenbarer Objekte ermöglicht, wie zum Beispiel Straßen oder Rasen. Diese sind jedoch bisher nicht in COCO enthalten, da nicht dem Fokus des Datensatzes entsprechend.

Cityscapes

Der Cityscapes-Datensatz ist wie KITTI auf das automatisierte Fahren spezialisiert und enthält daher eine Vielzahl an annotierten Daten aus dem Straßenverkehr. [COR+16] Dabei stellt Cityscapes den Nachfolger zum DUS-Datensatz dar. Wie KITTI vereint Cityscapes Datensatz und dazugehörige Benchmark-Suite. Dazu wurden Stereo-Videosequenzen in 50 verschiedenen Städten aufgezeichnet, unter anderem Frankfurt und Berlin, wobei 5000 Bilder pixelgenau semantisch annotiert wurden und weitere 20000 immerhin noch grober. Diese grob annotierten Daten wurden als Trainingsdaten verwendet, die 5000 fein annotieren Fotos in Trainings-/Validierungs- und Testdaten unterteilt (ca. 70 % zu 30 %). Außerdem sind (wie in KITTI) auch Tiefeninformationen in Form von Stereobildern enthalten. [COR+16] Aufgrund der Aufnahme der Daten in verschiedenen Städten anstatt nur in einer (Karlsruhe bei KITTI) ist nicht nur die Menge der Daten, sondern auch deren Diversität höher. Ein Beispiel hierfür ist dichter Innenstadtverkehr, der in KITTI nicht enthalten ist, aber besonders hohe Anforderungen an automatisiertes Fahren stellt und daher entsprechend wichtig ist.

Vergleich der Quellen

In Tabelle A.2 auf der nächsten Seite werden die Quellen verglichen und ihre Eignung für diese Arbeit dargestellt. Hierbei wird der Fokus auf den Anwendungsbereich dieser Arbeit gelegt, sprich die Eignung für das Training von Algorithmen für das automatisierte Fahren (zum Beispiel durch Tiefendaten oder 3D-Annotationen). Dadurch fallen Datensätze wie ImageNet und MS COCO trotz ihres immensen Umfangs aus, da sie auf Alltagsgegenstände spezialisiert sind, deren Anteil im Straßenverkehr relativ gering ist, und keine Tiefendaten wie Lidar oder Stereokamera enthalten. Auch wenn Cityscapes umfangreicher und vielseitiger ist, wird in dieser Arbeit hauptsächlich der KITTI-Datensatz verwendet, da dort unter anderem auch wie beim Demonstratorfahrzeug Lidar-Daten enthalten sind.

	ImageNet	KITTI	DUS	COCO	Cityscapes
Markierungen	2D BB	3D BB	sem. Seg.	sem. Seg.	sem. Seg.
Personen	ja	ja	ja	ja	ja
Fahrzeuge	ja	ja	ja	ja	ja
Tiefe	nein	**Lidar**, Stereo	Stereo	nein	Stereo

Tabelle A.2: Vergleich der Datensätze, BB = Bounding Boxes

A.2.2 Middleware-Lösungen

Während ROS2 auf DDS in Verbindung mit RTPS festgelegt ist, unterstützt AUTOSAR Adaptive zusätzlich SOME/IP und weitere in diesem Kontext nicht betrachtete, da zum Beispiel auf signalbasierte Kommunikation ausgelegte (*Signal PDU*), Middleware-Lösungen.

DDS

Ursprünglich für den Einsatz im Umfeld von Industrie 4.0 entworfen, eignet sich DDS auch für den Einsatz im Fahrzeug und wird zum Beispiel von ROS2 genutzt. Dabei handelt es sich um einen offenen Standard für die Kommunikation in verteilten Echtzeitsystemen. Für die Übertragung von Daten werden unterschiedliche QoS-Policies unterstützt (s. Abschnitt 3.6.1 auf Seite 72). Jedoch wird von DDS nicht das tatsächliche Protokoll vorgegeben, mit dem Implementierungen schlussendlich über verschiedene Netzwerkprotokolle Daten austauschen. Dieses wird auch als *wire protocol* bezeichnet (entsprechend OSI-Schicht 4 – *Transport*, s. Abschnitt 3.2.1 auf Seite 51). [OMG15]

RTPS

Da ohne ein *wire protocol* verschiedene DDS-Implementierungen nicht miteinander kommunizieren könnten, wurde RTPS zum DDS-Standard hinzugefügt. Auch dieses entstammt ursprünglich der Industrieautomatisierung und ist Teil der *Real-Time Industrial Ethernet Suite*. Damit muss es Echtzeitanforderungen erfüllen und auch die QoS-Prinzipien von DDS unterstützen. Fehlertoleranz-

konzepte wie der *GDS* ermöglichen den Aufbau von Netzwerken ohne *single point of failure* und damit eine zuverlässige *publish-subscribe*-Kommunikation. Der GDS wird durch die beteiligten Subscriber und Publisher aufgespannt, indem alle notwendigen Informationen dezentral auf diese verteilt werden. Dadurch ist kein zentrales Service-Repository (s. Abschnitt 3.5.3 auf Seite 64) notwendig [CC12]. Neue Applikationen und Services im Netzwerk werden automatisch dynamisch entdeckt, außerdem können diese dem Netzwerk beitreten oder dieses verlassen, ohne dass eine Rekonfiguration notwendig wird. [OMG18b]

SOME/IP

SOME/IP ist eine auf die Automotive Domäne spezialisierte, damit im Vergleich zu DDS vergleichsweise einfach aufgebaute, Middleware und Teil des AUTOSAR-Standards, jedoch auch außerhalb von AUTOSAR einsetzbar. Im Vergleich zu DDS werden zum Beispiel weniger QoS-Policies unterstützt, nämlich nur die *reliability*-Policy (s. Tabelle A.5 auf Seite 205). Für den Einsatz in serviceorientierten Architekturen wird SOME/IP durch das SOME/IP-SD[1] ergänzt, welches die Entdeckung und Registrierung von Services im Fahrzeugnetzwerk nachrüstet. [AUT14][AUT17b]

Vergleich der Middleware-Lösungen

Wie bereits beim Vergleich der SOA-Lösungen (s. Abschnitt 3.6.3 auf Seite 83) angesprochen, hat die verwendete Middleware einen großen Einfluss auf die Funktionalität der SOA. Die Eigenschaften vorgestellten Lösungen werden daher hier gegenübergestellt, ein tabellarischer Vergleich befindet sich in Tabelle A.8 auf Seite 208.

Während das aus der Automobilbranche stammende SOME/IP hauptsächlich für den Einsatz in AUTOSAR Classic und AUTOSAR Adaptive entwickelt wurde, lässt sich das in der Industrieautomatisierung verbreitete DDS sowohl in Verbindung mit AUTOSAR Adaptive als auch ROS2 einsetzen. So wird die Standard-API von SOME/IP nicht direkt spezifiziert, sondern im Rahmen von

[1] SOME/IP Service Discovery Protocol

AUTOSAR Classic definiert. DDS gibt dagegen gleich mehrere APIs für verschiedene Programmiersprachen vor, im Rahmen von herstellerspezifischen DDS-Implementierungen existieren noch weitere.

Aufgrund seiner weniger spezialisierten Herkunft stellt DDS (in Verbindung mit RTPS) die deutlich flexiblere und umfangreichere, aber damit auch mit höherem Konfigurationsaufwand verbundene, Middleware dar. So stellt DDS im Security-Unterstandard Security-Mechanismen zusätzlich zur Transportverschlüsselung zur Verfügung, während sich SOME/IP auf letztere beschränkt. Dazu nutzt DDS RTPS, das von den zugrundeliegenden Netwerkprotokollen abstrahiert. Dadurch lässt sich DDS im Gegensatz zu SOME/IP auch mit zusätzlichen Transportprotokollen ergänzen. Features wie QoS lassen sich durch die Abstraktion auch mit diesen weiternutzen. Entsprechend ist auch die Unterstützung von QoS-Policies in DDS wesentlich umfangreicher (s. Tabelle A.5 auf Seite 205), während SOME/IP nur eine einzige, nämlich die von TCP indirekt zur Verfügung gestellte *reliability*-Policy, beherrscht.

Eine höhere Flexibilität bietet DDS auch bei der Entdeckung von Services: Verfügbare Services können zur Laufzeit bekannt gemacht werden, Konsumenten können direkt ohne Vermittler auf diese zugreifen (*peer to peer*). Dabei ist keine Anpassung der Konsumenten auf spezifische Service-Informationen notwendig, da alle für den Zugriff benötigten Metadaten übermittelt werden. Bei SOME/IP hingegen ist die Kommunikation objektorientiert. Konsumenten müssen auf vordefinierte Objektklassen zugreifen, diese müssen daher schon zur Entwicklungszeit definiert und bekannt sein.

A.2.3 Verfügbare Fahrzeugarchitekturen

Traditionell existieren in der Automobilindustrie keine veröffentlichten standardisierten Fahrzeugarchitekturen, bestehend aus E/E-Architektur und SW-Architektur. Jedoch gibt es einige Bemühungen, dies zu ändern. Einen aktuellen Vertreter stellt Volvos NGEA[2] dar, die in [PKH+17] vorgestellt wird. Dieses Paper liefert auch einen Überblick über den Stand der Technik auf dem die NGEA aufsetzt, ein erster Ansatz ist dabei das AAF[3], das das gesamte

[2] Next Generation Electrical Architecture
[3] Automotive Architecture Framework

Fahrzeug über alle funktionalen und anderen Ebenen hinweg konform zur ISO/IEC/IEEE 42010:2011 [Int11] beschreibt [BGM+09]. Das ADF[4] ist eine verwandte Implementierung von Renault mit einigen Anpassungen und Detaillierungen [GGT13]. Schließlich stellt AFAS[5] eine Weiterentwicklung der ersten beiden unter Berücksichtigung vorhandener ADLs wie EAST-ADL mit zusätzlicher Betrachtung der Softwareimplementierung dar [Daj19]. Abseits der Automobilindustrie gibt es einige weitere Projekte, die sich mit Architekturen für neuartige, über das herkömmliche Automobil hinausreichende, Fahrzeugkonzepte befassen.

RACE–Reliable Control and Automation Environment

Das RACE-Projekt ist der Ansatz, eine Fahrzeugarchitektur aus der informationstechnischen Perspektive mit Standardkomponenten aufzubauen. Dazu gehören neben einer Datenzentrierung, Daten werden jeweils nur an einer Stelle im System erzeugt und gespeichert, Plug-and-Play-Ansätze, um Features während des gesamten Lebenszyklus des Fahrzeuges nachrüsten zu können. Diese neuen Features sollen dabei als Software-Updates, auch von Drittanbietern, im Feld den Weg in das Fahrzeug finden, aber auch neue Hardware wie Sensoren oder Aktoren soll hinzugefügt werden können. [KBKS19]

Um diese Konzepte umzusetzen, wurde eine eigene Laufzeitumgebung, *RACE RTE*, mit der ebenfalls eigenen Middleware *CHROMOSOME* entwickelt. Diese läuft auf General-Purpose-Steuergeräten, wodurch flexibel zusätzliche Softwarekomponenten hinzugefügt werden können.

Sich selbst vergleicht das Projekt eher mit AUTOSAR Classic anstatt mit AUTOSAR Adaptive. Zum Beispiel ist es keine echte SOA, die Daten werden über themenbezogene (zum Beispiel Geschwindigkeit) Kanäle publiziert, ähnlich wie in ROS (s. Abschnitt 3.3.3 auf Seite 56). Dabei sind in CHROMOSOME die erlaubten Nachrichtentypen und ihre Datenformate statisch in einem sogenannten *dictionary* vordefiniert [BGG+14] und damit nicht aufwärtskompatibel zu potenziell neu hinzukommenden Nachrichtentypen.

[4] Architectural Design Framework
[5] Architecture Framework for Automotive Systems

UNICARagil

UNICARagil ist der Versuch, eine modulare Architektur für agile, vollautomatisierte fahrerlose Fahrzeugkonzepte zu entwickeln [KNR+19]. Der Fokus liegt dabei auf „urban mobility concepts“ [WEK+18]. Dies umfasst eine modulare und skalierbare Fahrzeugplattform, die durch sogenannte „Addons“ mit in Länge und Höhe skalierbaren Transportmodulen versehen werden kann, und so in vier verschiedenen Demonstrationsfahrzeugen zum Einsatz kommt [KNR+19]:

AUTOtaxi entspricht einem Taxi ohne Fahrer: Das Fahrzeug kann per Smartphone an einen beliebigen Ort bestellt werden und bringt die Passagiere an einen Ort ihrer Wahl.

AUTOelfe stellt das klassische Individualfahrzeug dar. So ist es zum Beispiel für den wöchentlichen Großeinkauf gedacht.

AUTOliefer holt Pakete selbstständig ab und stellt sie ebenso selbstständig zu. Damit automatisiert es Lieferdienste.

AUTOshuttle soll den öffentlichen Personennahverkehr ergänzen beziehungsweise in abgelegenen Gegenden ersetzen. Dabei fährt das Shuttle ähnlich wie ein Personenzug eine feste Linie mit definierten Haltestellen und festem Fahrplan ab, gegebenenfalls sogar auf speziellen Fahrspuren.

Dabei sind die Hauptkomponenten wie Sensorik und Aktorik sowie das analog zur Struktur des menschlichen Nervensystems aufgebaute und bezeichnete Netzwerk der Steuergeräte identisch (s. Abbildung A.4 auf der nächsten Seite). So besteht die Sensorik aus fest definierten Sensormodulen, die jeweils Lidar, Radar sowie Mono- und Stereokamera beinhalten. Zusätzlich enthält jedes Sensormodul eine Sensordatenverarbeitungseinheit, die für die Verarbeitung der Rohdaten zum Beispiel zu Objektlisten verantwortlich ist. Die „Großhirn“ beziehungsweise „Cerebrum“ genannte zentrale Verarbeitungseinheit fusioniert die so gewonnenen Daten, eine Überlappung der Bildbereiche soll dabei die Redundanz sicherstellen und Ausfälle einzelner Sensoren kompensieren

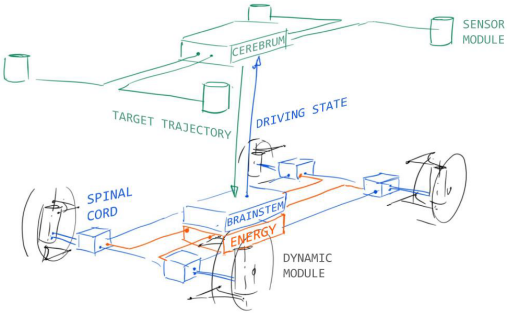


Abbildung A.4: Komponentenarchitektur von UNICARagil [KNR+19]

können. Außerdem können so fehlerhafte oder inkonsistente Daten detektiert werden. Basierend hierauf führt das Cerebrum die Trajektorienplanung aus. Diese kann entweder auf der normalen Fahrfunktion (dynamische Fahraufgabe, vergleiche Abschnitt 2.1.1 auf Seite 13) basieren oder das Einnehmen eines sicheren Zustands darstellen. Dieser kann sich je nach Fahrsituation unterschiedlich gestalten, üblicherweise muss an den Fahrbahnrand gefahren und dort angehalten werden. Da wie ebendort beschrieben beim vollautomatisierten Fahren nach Stufe 5 kein menschlicher Fahrer für diese Aufgabe verfügbar ist, muss eine Fallbackebene vorhanden sein. Diese besteht hier aus dem sogenannten „Stammhirn (brainstem)“, welches über eine eigene, vom Cerebrum unabhängige, Sensorik verfügt und damit bei einem Ausfall dessen davon unabhängig den risikominimalen Zustand einnehmen kann. Um die Ausfallwahrscheinlichkeit des Stammhirns zu senken, wird auf eine Duo-Duplex-Architektur gesetzt. Dies ist ein in der Luftfahrt verbreitetes Redundanzkonzept, bei dem zwei redundante Systeme eingesetzt werden, von denen jedes einzelne ebenfalls zweifach redundant ist. [Flü12] Hierbei können zur Vermeidung systematischer Fehler zum Beispiel von verschiedenen Teams entwickelte Software-Komponenten auf verschiedenen CPU-Architekturen ausgeführt werden (Softwareredundanz). Im Vergleich zur ebenfalls verbreiteten Triple Modular Redundancy (s. Abschnitt 4.2.5 auf Seite 109) entfällt der Voter, dafür sind jedoch vier statt drei Ausführungseinheiten notwendig. Als Konsequenz ist keine so aufwändige Redundanz der kritischen externen Ressourcen wie Spannungs- oder Taktversorgung notwendig, wofür eine leicht

höhere Ausfallrate in Kauf genommen wird. Jedes Subsystem ist hierbei *fail-silent* (s. Definition 1.6.3 auf Seite 12) ausgeführt, wodurch die Ausgangsdaten korrekt oder gar nicht vorhanden sind. Damit ist das Gesamtsystem immer noch *fail-operational* (s. auch Definition 1.6.3 auf Seite 12), da die redundante Komponente weiterhin korrekte Daten liefert. [KNR+19]

Im Konkreten besteht das Stammhirn aus zwei mit Gigabit-, Automotive-Ethernet und CAN-FD ausgestatteten Boards, auf welchen jeweils ein MPSoC[6] arbeitet, der mehrere Prozessorkerne, einen Mikrocontroller und ein FPGA besitzt, womit das oben angesprochene Redundanzkonzept vollständig umgesetzt werden könnte. [KNR+19]

Als Aktorik kommen immer vier identische und einzeln steuerbare Dynamikmodule zum Einsatz. Diese bestehen jeweils aus einem 48V-Radnabenmotor und einem Lenkmotor, der einen Lenkwinkel von 90° ermöglicht, sowie einer ECU, „Rückenmark (spinal chord)“ genannt. Diese ECU besteht aus einem Infineon AURIX [Inf19], auf dem FreeRTOS ausgeführt wird, ein Echtzeitbetriebssystem für eingebettete Systeme [Ama19]. Zur Kommunikation verfügt jedes der Dynamikmodule über Automotive Ethernet, CAN und FlexRay. [KNR+19]

Basis für das Zusammenspiel der Komponenten ist eine eigens entwickelte SOA mit RTPS (s. Anhang A.2.2 auf Seite 196) als Middleware [KAK+19]. Da es sich bei RTPS um das verwendete Netzwerkprotokoll von DDS (s. Anhang A.2.2 auf Seite 196) handelt, soll die UNICARagil-SOA zu anderen SOA kompatibel sein, die ebenfalls DDS einsetzen. Dies sind zum Beispiel ROS2 (s. Abschnitt 3.6.1 auf Seite 65) und AUTOSAR Adaptive (s. Abschnitt 3.6.2 auf Seite 74).

Dabei ist diese SOA vollständig auf das automatisierte Fahren ausgelegt. Die Integration der Services findet zwar nicht mehr vollständig in der Entwurfsphase sondern zur Laufzeit statt, jedoch ist die Verschaltung der Services weiterhin statisch festgelegt. Ein sogenannter *Orchestrator* (nicht zu verwechseln mit dem leistungsfähigeren Orchestrator aus dieser Dissertation, s. Abschnitt 4.2.2 auf Seite 104) wählt Services basierend auf einem zur Entwurfszeit definierten Regelwerk aus und bestimmt deren Verknüpfungen untereinander, um die automatisierte Fahrfunktion deterministisch auszuführen. [KAK+19] Damit lassen sich die Implementierungen der Services austauschen, die Gesamt-

[6] Multiprocessor System-on-Chip

struktur jedoch nicht dynamisch ändern. Dies ist, von der Spezialisierung auf das automatisierte Fahren abgesehen, der Hauptunterschied zu dem in dieser Dissertation vorgestellten Ansatz, der basierend auf *Fähigkeiten* auch zur Laufzeit dynamisch (re-)konfigurierbar ist, zum Beispiel um Komponenten hinzuzufügen oder auszutauschen.

AKIT

Einen anderen Ansatz verfolgt hier dagegen *AKIT – Autonomie-KIT für serienahe Arbeitsfahrzeuge zur vernetzten und assistierten Bergung von Gefahrenquellen*. Hier wird keine von Grund auf neue Architektur entwickelt, ebenso wird nicht der Individualverkehr adressiert. Stattdessen sollen bereits verfügbare Bau- und Arbeitsmaschinen in unbemannt operierendes Bergegerät umgerüstet werden. Damit sollen Einsatzkräfte in Katastrophenfällen besser geschützt und zum Beispiel durch Strahlung bisher schwer zugängliche Orte (Beispiel: Atomunfall in Fukushima, bei dem spontan Rettungsroboter benötigt wurden [NKO+13]) erreicht werden. Dazu werden die Maschinen mit Sensorik wie Kameras, GNSS-Empfängern und Lidar, sowie mit einer Rechnereinheit zur Ansteuerung von Sensorik und Aktorik sowie für weitere Aufgaben wie die Trajektorienplanung und Kartierung des Geländes ausgestattet. Kommunikationskomponenten stellen dabei die Möglichkeit der Fernsteuerung und der Koordination aus der Ferne sicher. [Com19]

A.3 Tabellen

A.3.1 Zahlen, Daten, Fakten zu Fahrzeugen

Typ	Entwicklung	Produktion	Aftersales
PKW	3	6	15
LKW	5	10	15
Busse	5	15	15

Tabelle A.3: Lebenszyklen verschiedener Fahrzeugtypen in Jahren [Sax18]

Typ	Anzahl	Alter in Jahren	Anteil in %
PKW	47 715 977	9,6	89,46
LKW	3 276 093	8,1	6,14
Busse	81 364	8,5	0,15
Zugmaschinen gesamt	2 265 585	29,9	4,25
- Land-/forstwirt. Z.	(1 514 564)	(28,7)	
- Sattelzugmaschinen	(219 149)	(4,4)	
- sonstige Zugmaschinen	(531 872)	(43,6)	
Durchschnittsalter		10,4	

Tabelle A.4: ausgewählte zugelassene Fahrzeuge (ohne Krafträder oder sonstige Fahrzeuge) in Deutschland [KBA20a] und deren durchschnittliches Alter [KBA20b] und Anteil an der Gesamtzahl der Fahrzeuge (Stand Januar 2020)

A.3.2 Stand der Technik

Bezeichnung	Beschreibung
History	**Keep last:** maximal N Werte werden gespeichert, über die *Depth*-Option konfigurierbar. **Keep all:** alle empfangenen Werte bis zum Limit der zugrundeliegenden Middleware werden gespeichert.
Depth	**Depth:** Wenn die *History*-Policy auf *keep last* gestellt wurde, wird hierbei die Anzahl zwischenzuspeichender Werte angegeben.
Reliability	**Best effort:** Es wird versucht, die Werte auszuliefern, jedoch können Werte bei unzuverlässiger Netzwerkverbindung verloren gehen. **Reliable:** Es wird garantiert, dass alle Werte ausgeliefert werden, notwendigerweise wird eine Übertragung mehrfach versucht.
Durability	**Transient local:** Der Publisher ist dafür verantwortlich, Werte für später beitretende Konsumenten vorzuhalten. **Volatile:** Es werden keine Werte für noch nicht verbundene Subscriber zwischengespeichert.
Deadline	**Duration:** Die erwartete maximale Zeitspanne zwischen zwei aufeinanderfolgenden Nachrichten desselben Topics.
Lifespan	**Duration:** Die maximale Zeitspanne zwischen dem Senden einer Nachricht und deren Empfang, bevor diese als ungültig betrachtet und entsprechend verworfen wird.
Liveliness	**Automatic:** Es wird vom System angenommen, dass alle Publisher eines Nodes lebendig sind, sobald ein beliebiger Publisher eine Nachricht versandt hat. **Manual by topic:** Der Publisher muss dem System aktiv mitteilen, dass er noch lebendig ist.
Lease Duration	**Duration:** Die maximale Zeitdauer, die das System wartet bevor es einen Publisher als tot betrachtet, was häufig ein Zeichen für einen Ausfall darstellt.

Tabelle A.5: Die unterstützten Policies des Basis-QoS-Profils in ROS2 [ros20c]

	AUTOSAR Classic Platform	AUTOSAR Adaptive Platform
Veröffentlichung	2005	2017
Betriebssystem	OSEK	POSIX
Programmierparadigma	Prozedural (C)	Objektorientiert (C++)
Adressierung	Gleicher Adressraum für alle Anwendungen, MPU-Unterstützung für Safety	Eigener virtueller Adressraum für jede Anwendung durch MMU
Kommunikation	Signalbasiert (CAN, FlexRay)	Serviceorientiert (Ethernet)
Taskkonfiguration	Statische Anzahl an Tasks, kooperatives Multitasking	Dynamische Anzahl an Tasks, präemptives Multitasking

Tabelle A.6: Unterschiede AUTOSAR Adaptive und Classic [AUT18] [Vec18]

Eigenschaft	ROS2	AUTOSAR Adaptive
Anpassbarkeit	Open Source, daher sind alle Komponenten frei modifizierbar. Quellcode und Binärpakete sind frei verfügbar. Apache 2.0 License: Software darf frei verwendet, modifiziert und verteilt werden. Eigene Software und Änderungen müssen nicht öffentlich gemacht werden und können unter beliebige Lizenz gestellt werden	Nur ein Standard, daher von verwendetem Stack abhängig (z.B. Vector Adaptive MICROSAR vs. selbst implementiert). Standard frei verfügbar, Verwendung in Produkten erfordert kostenpflichtige Registrierung als Partner, ab ca. 15 000 € pro Jahr bzw. 6000 € und einem Beitrag im Umfang einer halben Vollzeitstelle pro Jahr. [AUT17a]
Betriebssystembasis	POSIX (MacOS, Linux) oder Windows als Basis verwendbar	POSIX-kompatibles Betriebssystem als Basis erforderlich
Service-Discovery	dynamisch zur Laufzeit	Konfigurierbar: komplett statische Konfiguration, keine Entdeckung durch Applikationen, dynamische Entdeckung zur Laufzeit.
Deployment	*launch*-Files mit Abhängigkeiten und zu startenden Komponenten, keine Ressourcenzuweisung/-beschränkung vorgesehen	*Execution Manifest* mit Abhängigkeiten und zu startenden Komponenten, sowie zuzuweisenden Ressourcen
HW-Konfiguration	nicht im Fokus	im *Machine Manifest* definiert
Security	Keine integrierten Maßnahmen, Nutzung des DDS-Security-Standards möglich, sowie Transportverschlüsselung	Module zur Authentifizierung und Authorisierung (*Identity and Access Management*) und Verschlüsselung (*Cryptography*). Transportverschlüsselung etc. von der Middleware abhängig (s. Tabelle A.8 auf der nächsten Seite)

Tabelle A.7: Vergleich ROS2 und AUTOSAR Adaptive

Eigenschaft	DDS und RTPS	SOME/IP
Ursprung	gesamte Industrie	Automobilbranche
Kommunikation	Entdeckungsmechanismus erlaubt einzelnen Teilnehmern verfügbare Services zur Laufzeit zu identifizieren. Konsumenten können direkt ohne Vermittler auf Daten und Services zugreifen (*peer to peer*). Keine Anpassung auf spezifische Service-Version notwendig	Objektorientierte Kommunikation, Konsumenten müssen auf vordefinierte Serviceklassen zugreifen und daher zur Entwicklungszeit an diese angepasst werden.
API	APIs für verschiedene Programmiersprachen standardisiert, beziehungsweise Teil von herstellerspezifischen DDS-Implementierungen	Standard-API wird nicht vorgegeben, wird stattdessen in AUTOSAR Classic definiert.
Netzwerk	Nutzt RTPS, dieses abstrahiert von TCP und UDP, aber auch weiteren wie shared memory, bei dem auf einen gemeinsamen Speicherbereich zugegriffen wird. Zusätzlich lassen sich dadurch weitere Transportprotokolle hinzufügen und dennoch Features wie QoS weiternutzen.	Netzwerkprotokolle TCP (verbindungsorientiert, zuverlässiger) und UDP (verbindungslos, weniger Overhead) werden unterstützt. Keine Erweiterung um eigene Transportprotokolle.
Security	Kann die Transportverschlüsselung des zugrundeliegenden Transportprotokolls nutzen, bietet aber selbst von diesem unabhängige Security-Mechanismen an. [OMG18a]	Transportverschlüsselung (zum Beispiel Transport Layer Security bei TCP) wird genutzt. Eigene Security-Mechanismen existieren nicht. [Wol]
QoS	Mehrere verschiedene QoS-Policies werden unterstützt, s. Tabelle A.5 auf Seite 205	Nur eine *reliability*-Policy wird unterstützt, basierend auf TCP.
Verwendung in	AUTOSAR Adaptive, ROS2	AUTOSAR Classic (über Transformer), AUTOSAR Adaptive

Tabelle A.8: Vergleich DDS und SOME/IP, Erweiterung von [Jam18]

Herausforderung	NGEA	RACE	UNICARagil	AKIT	(re-)konfigurierbare Fahrzeugarchitektur
HF1: Nutzen von Standardkomponenten für Verfügbarkeit und günstige Einkaufspreise	nein	ja	ja[3]	ja[4]	ja
HF2: Anpassungen aufgrund nicht mehr verfügbarer Komponenten wie Sensoren oder Aktoren, aber auch Bauteilen von ECUs wie CPUs	nein	ja[2]	nein	nein	ja
HF3: Anpassungen aufgrund von Aktualisierungen oder Erweiterungen der Funktionalität	nein	ja	nur manuell und offline	nein[4]	automatisch zur Laufzeit
HF4: Verwaltung der möglichen Varianten der Komponenten, besonders unter Betrachtung von Aktualisierungen	ja[1]	ja	nein	nein[4]	ja
HF5: Universelle Anwendbarkeit der Architekturkonzepts (auf PKW, LKW, Busse, ...), um Synergien zu nutzen	nein	nicht im Fokus, prinzipiell möglich	nein	ja[4]	ja

[1] keine Aktualisierung von Komponenten
[2] keine Berücksichtigung der Aufwärtskompatibilität
[3] keine Berücksichtigung von Langzeitverfügbarkeit
[4] nur für ein Einzelfahrzeug

Tabelle A.9: Abdeckung der Herausforderungen durch den Stand der Technik

A.3.3 Ergebnisse MLPerf

	Abfragen pro Sekunde	Min. Latenz (ms)	Max. Latenz (ms)	Mittl. Latenz (ms)	0,5-Quantil (ms)	0,9-Quantil (ms)	0,95-Quantil (ms)	0,97-Quantil (ms)	0,99-Quantil (ms)	0,999-Quantil (ms)
12C/AMD64/64GB	50,01	18,61	27,18	20,00	19,95	20,45	20,67	20,90	21,50	25,15
12C/AMD64/64GB, ohne Docker	52,71	17,69	24,38	18,97	18,95	19,34	19,51	19,68	20,21	22,81
12C/AMD64/64GB/lGPU	34,89	25,76	32,14	28,66	28,66	29,69	30,03	30,25	30,56	31,25
12C/AMD64/64GB/lGPU, ohne Docker	39,79	22,40	27,81	25,13	25,14	26,00	26,23	26,48	26,83	27,72
12C/AMD64/64GB/sGPU	34,58	26,32	31,85	28,92	28,88	29,81	30,21	30,51	30,84	31,59
12C/AMD64/64GB/sGPU, ohne Docker	42,51	21,67	25,58	23,52	23,51	24,24	24,47	24,63	24,90	25,47
12C/ARM64/64GB	1,21	800,49	856,75	829,78	829,67	841,65	845,17	847,97	850,78	855,00
4C/ARM64/4GB	3,68	256,58	299,84	271,66	271,46	278,05	280,07	282,04	284,58	289,73
NeuroECU/oGPU	5,28	183,02	204,52	189,53	189,33	192,69	193,94	195,11	197,33	202,87
NeuroECU/GPU	8,22	104,71	149,65	121,72	121,85	128,80	131,24	132,55	134,58	139,55

Tabelle A.10: MLPerf-Ergebnisse Singlestream leicht

	Abfragen pro Sekunde	Min. Latenz (ms)	Max. Latenz (ms)	Mittl. Latenz (ms)	0,5-Quantil (ms)	0,9-Quantil (ms)	0,95-Quantil (ms)	0,97-Quantil (ms)	0,99-Quantil (ms)	0,999-Quantil (ms)
12C/AMD64/64GB	2,18	439,70	474,73	459,04	459,06	463,46	465,21	466,54	468,59	470,77
12C/AMD64/64GB, ohne Docker	2,26	432,89	458,84	443,10	442,64	449,00	450,56	451,72	454,01	458,74
12C/AMD64/64GB/lGPU	11,57	82,47	93,04	86,41	86,23	88,24	88,96	89,35	90,20	91,48
12C/AMD64/64GB/lGPU, ohne Docker	11,72	81,70	91,00	85,33	85,25	87,07	87,77	88,28	89,22	90,23
12C/AMD64/64GB/sGPU	13,15	73,06	81,76	75,94	75,78	77,50	78,06	78,57	79,28	81,61
12C/AMD64/64GB/sGPU, ohne Docker	13,59	70,67	81,71	73,57	73,33	75,04	76,07	77,72	79,63	81,60
12C/ARM64/64GB	0,01	$9{,}66 \cdot 10^4$	$9{,}96 \cdot 10^4$	$9{,}87 \cdot 10^4$	$9{,}88 \cdot 10^4$	$9{,}91 \cdot 10^4$	$9{,}92 \cdot 10^4$	$9{,}93 \cdot 10^4$	$9{,}95 \cdot 10^4$	$9{,}96 \cdot 10^4$
4C/ARM64/4GB	0,05	$1{,}82 \cdot 10^4$	$1{,}99 \cdot 10^4$	$1{,}89 \cdot 10^4$	$1{,}89 \cdot 10^4$	$1{,}93 \cdot 10^4$	$1{,}94 \cdot 10^4$	$1{,}94 \cdot 10^4$	$1{,}96 \cdot 10^4$	$1{,}98 \cdot 10^4$
NeuroECU/oGPU	0,06	$1{,}53 \cdot 10^4$	$1{,}59 \cdot 10^4$	$1{,}55 \cdot 10^4$	$1{,}55 \cdot 10^4$	$1{,}56 \cdot 10^4$	$1{,}56 \cdot 10^4$	$1{,}56 \cdot 10^4$	$1{,}56 \cdot 10^4$	$1{,}57 \cdot 10^4$
NeuroECU/GPU[1]	–	–	–	–	–	–	–	–	–	–

[1] Aufgrund der Aufteilung des Arbeitsspeichers zwischen GPU und CPU nicht genügend Ressourcen für die Ausführung, daher Abbruch.

Tabelle A.11: MLPerf-Ergebnisse Singlestream schwer

	Min. Latenz (ms)	Max. Latenz (ms)	Mittl. Latenz (ms)	0,5-Quantil (ms)	0,9-Quantil (ms)	0,95-Quantil (ms)	0,97-Quantil (ms)	0,99-Quantil (ms)	0,999-Quantil (ms)
12C/AMD64/64GB	52,48	76,40	55,40	55,36	56,29	56,62	56,91	57,95	59,97
12C/AMD64/64GB, ohne Docker	50,28	63,33	53,18	53,06	54,27	54,80	55,25	56,41	58,25
12C/AMD64/64GB/lGPU	69,64	438,12	79,54	80,08	82,69	83,30	83,67	84,34	85,46
12C/AMD64/64GB/lGPU, ohne Docker	61,10	370,73	69,91	70,21	71,80	72,21	72,48	72,99	73,98
12C/AMD64/64GB/sGPU	73,28	400,94	82,58	82,63	84,54	85,11	85,46	86,12	87,22
12C/AMD64/64GB/sGPU, ohne Docker	58,25	349,49	66,82	66,85	68,21	68,60	68,85	69,33	70,21
12C/ARM64/64GB[1]	–	–	–	–	–	–	–	–	–
4C/ARM64/4GB	923,41	1.965,22	972,19	970,13	993,75	1.000,65	1.004,98	1.013,62	1.046,75
NeuroECU/oGPU	583,60	911,32	611,99	611,97	620,56	623,15	624,97	628,74	638,97
NeuroECU/GPU	410,48	8.139,58	461,74	461,44	473,99	477,74	480,23	485,02	495,47

[1] Speicherleck, sodass selbst die 64GB RAM mit der Emulation nicht ausreichen, um den Benchmark erfolgreich abzuschließen.

Tabelle A.12: MLPerf-Ergebnisse Multistream leicht

	Min. Latenz (ms)	Max. Latenz (ms)	Mittl. Latenz (ms)	0,5-Quantil (ms)	0,9-Quantil (ms)	0,95-Quantil (ms)	0,97-Quantil (ms)	0,99-Quantil (ms)	0,999-Quantil (ms)
12C/AMD64/64GB	1.840,83	1.995,89	1.909,63	1.910,67	1.927,48	1.931,55	1.934,06	1.938,50	1.946,14
12C/AMD64/64GB, ohne Docker	1.799,86	1.980,50	1.854,24	1.854,71	1.866,27	1.869,44	1.871,44	1.875,46	1.882,68
12C/AMD64/64GB/lGPU	344,23	1.573,11	362,04	362,08	366,72	368,06	368,94	370,68	374,12
12C/AMD64/64GB/lGPU, ohne Docker	335,96	1.548,41	353,16	353,25	357,38	358,55	359,31	360,76	363,65
12C/AMD64/64GB/sGPU	301,02	1.254,14	316,91	316,93	321,41	322,75	323,62	325,34	328,84
12C/AMD64/64GB/sGPU, ohne Docker	284,33	1.239,60	298,98	299,11	302,78	303,76	304,42	305,68	308,07
12C/ARM64/64GB[1]	–	–	–	–	–	–	–	–	–
4C/ARM64/4GB[1]	–	–	–	–	–	–	–	–	–
NeuroECU/oGPU[1]	–	–	–	–	–	–	–	–	–
NeuroECU/GPU	3.539,24	12.330,08	3.781,60	3.782,69	3.855,20	3.875,52	3.888,75	3.914,80	3.967,54

[1] Derart schlechte Performanz, dass nach einer Laufzeit von mehreren Wochen abgebrochen wurde.

Tabelle A.13: MLPerf-Ergebnisse Multistream schwer

	Bilder pro Sekunde	Min. Latenz (s)	Max. Latenz (s)	Mittl. Latenz (s)	0,5-Quantil (s)	0,9-Quantil (s)	0,95-Quantil (s)	0,97-Quantil (s)	0,99-Quantil (s)	0,999-Quantil (s)
12C/AMD64/64GB	105,73	7,15	232,43	119,89	123,35	210,86	225,19	232,30	232,41	232,43
12C/AMD64/64GB, ohne Docker	107,11	7,10	229,44	118,41	121,91	208,02	222,26	229,36	229,42	229,44
12C/AMD64/64GB/lGPU	57,41	15,58	428,12	225,50	225,71	395,16	417,16	425,04	428,11	428,12
12C/AMD64/64GB/lGPU, ohne Docker	67,40	14,32	364,61	192,55	193,08	337,05	354,94	362,06	364,49	364,61
12C/AMD64/64GB/sGPU	56,11	16,74	438,02	230,69	230,98	406,05	427,09	435,35	437,85	438,02
12C/AMD64/64GB/sGPU, ohne Docker	71,27	12,74	344,81	181,59	182,19	317,99	337,09	342,39	344,76	344,81
12C/ARM64/64GB	1,55	478,38	$1{,}58 \cdot 10^4$	8.172,58	8.143,48	$1{,}45 \cdot 10^4$	$1{,}53 \cdot 10^4$	$1{,}55 \cdot 10^4$	$1{,}58 \cdot 10^4$	$1{,}58 \cdot 10^4$
4C/ARM64/4GB	4,69	28,83	5.243,73	2.638,67	2.641,47	4.740,17	5.001,15	5.088,98	5.195,96	5.243,73
NeuroECU/oGPU	7,60	25,45	3.233,86	1.633,26	1.634,90	2.924,86	3.085,79	3.148,56	3.221,04	3.233,86
NeuroECU/GPU	7,74	122,25	3.176,81	1.655,43	1.658,76	2.889,38	3.036,20	3.101,97	3.167,15	3.176,81

Tabelle A.14: MLPerf-Ergebnisse Offline leicht

A.3.4 Ergebnisse Austausch von Komponenten

$f_{Heartbeat}$	Mittelwert[1]	σ[1]	Ohne Zustand[2]	Unkonfiguriert[2]	Inaktiv[2]	Aktiv[2]
5 Hz	138,47 ms (0,69)	78,55 ms (0,39)	41,18 ms	51,58 ms	29,69 ms	32,18 ms
15 Hz	57,76 ms (0,87)	30,00 ms (0,45)	13,87 ms	43,93 ms	6,77 ms	24,21 ms
30 Hz[3]	28,35 ms (0,43)	25,13 ms (0,38)	6,81 ms	44,77 ms	3,98 ms	13,00 ms

[1] Dauer des Umschaltens aus Sicht des Konsumenten, Sensor-Node im Zustand *Aktiv*
[2] Verweildauer in den Zuständen des Sensor-Nodes bis bereit
[3] Erwartete Bildrate auf Empfängerseite nur 15 Hz

Tabelle A.15: Zeit für den Wechsel auf einen besseren Sensor-Node. In Klammern: Dauer in Bildperioden [SGS+21]

$f_{Heartbeat}$	Mittelwert	σ
5 Hz	384,50 ms (1,92)	67,00 ms (0,34)
15 Hz	102,93 ms (1,54)	32,76 ms (0,49)
30 Hz[1]	83,33 ms (1,25)	14,71 ms (0,22)

[1] Erwartete Bildrate auf Empfängerseite nur 15 Hz

Tabelle A.16: Zeit für den Wechsel bei Ausfall eines Sensor-Nodes [SGS+21]

A.3.5 Abdeckung der Anforderungen durch diese Dissertation

	Beschreibung	Abdeckung	Abschnitt(e)
AA1	Anpassung an veränderte HW-/SW-Konfigurationen muss automatisiert erfolgen.	✓Abstraktion der Software von der Hardware durch Containerisierung	3.7.1 (S. 88)
AA2	Sensoren, Aktoren sowie neue ECUs müssen sich selbst am System registrieren und damit Services bekanntmachen können.	✓definierter Anmeldeprozess	4.2.4 (S. 107)
AA3	Neue Komponenten dürfen keine Vorkonfiguration des Systems erfordern.	✓Übertragung von Metadaten bei der Anmeldung	4.2.4 (S. 107)
AA4	Parameter neuer Komponenten müssen an das System übermittelt werden können.	✓Übertragung von Metadaten bei der Anmeldung	4.2.4 (S. 107)

Tabelle A.17: Abdeckung der aufgestellten Adaptierbarkeitsanforderungen

	Beschreibung	Abdeckung	Abschnitt(e)
AP1	Softwarekomponenten müssen auch auf aktualisierten Steuergeräten ausführbar sein. D.h. die Ausführung darf nicht von neueren Treiber- oder Betriebssystemversionen beeinträchtigt werden.	✓Containerisierung und Übertragung der Konfiguration bei der Anmeldung	3.7.1 (S. 88), 4.2.4 (S. 107)
AP2	Softwarekomponenten müssen plattformunabhängig, d.h. z.B. auf verschiedenen Prozessorarchitekturen lauffähig sein.	✓Containerisierung	3.7.1 (S. 88)
AP3	Softwarekomponenten und Services müssen sich zur Laufzeit zwischen Steuergeräten verschieben lassen.	✓Orchestrierung	4.2.2 (S. 104)
AP4	Rekonfiguration und Aktualisierung von HW und SW müssen ausfallfrei erfolgen, dürfen also keine Unterbrechungen zur Laufzeit verursachen.	✓erweiterte „Managed Nodes“ zur Unterbrechungsfreien Umschaltung	4.2.7 (S. 112)
AP5	Softwarekomponenten sollen als Einheit mit der Beschreibung ihrer Anforderungen verteilt werden können.	✓Software-Repository	4.2.2 (S. 104)

Tabelle A.18: Abdeckung der aufgestellten Portabilitätsanforderungen

	Beschreibung	Abdeckung	Abschnitt(e)
AS1	Wichtige Services müssen redundant ausgeführt werden und ausgefallene Services ersetzen können.	✓Redundanz durch Orchestrierung	4.2.5 (S. 109)
AS2	Die Services müssen fail-silent sein und dürfen im Fehlerfall keine fehlerhaften Daten senden.	✓Geeignete Konzipierung der Services	
AS3	Daten im Fahrzeugnetzwerk oder Backend dürfen nicht von unautorisierten Teilnehmern gelesen werden können.	✓Transportverschlüsselung mit eigenen Zertifikaten, nur authentifizierte und autorisierte Komponenten können sich im Fahrzeug anmelden	4.2.2 (S. 104), 4.2.4 (S. 107)
AS4	Daten im Fahrzeugnetzwerk oder Backend dürfen nicht von unautorisierten Teilnehmern gesendet werden können.	✓Transportverschlüsselung mit eigenen Zertifikaten, nur authentifizierte und autorisierte Komponenten können sich im Fahrzeug anmelden	4.2.2 (S. 104), 4.2.4 (S. 107)

Tabelle A.19: Abdeckung der aufgestellten Sicherheitsanforderungen

Glossar

A

Adaptive Application

User-Applications in AUTOSAR Adaptive, bauen auf den Komponenten der ARA auf. 231,

Adaptive Cruise Control

System, das die Fahrzeuggeschwindigkeit nach Wunsch so regelt, dass ein sinnvoller Abstand zum vorausfahrenden Fahrzeug eingehalten wird. 15, 231,

Analog-Digital-Wandler

Wandelt ein analoges in ein digitales Signal. 231,

Application Frontend

Schnittstelle der Architektur zur Außenwelt. 61, 65

Application Programming Interface

Programmierschnittstelle, die Funktionalitäten der zugrundeliegenden Implementierung abstrahiert und anderen Anwendungen zur Verfügung stellt 231,

Architectural Design Framework

Framework zur Beschreibung der Fahrzeugarchitektur. 199, 231,

Architecture Description Language

Dient der Beschreibung und Darstellung von Systemarchitekturen. 231,

Architecture Framework for Automotive Systems

Framework zur Beschreibung der Fahrzeugarchitektur. 199, 231,

Automotive Architecture Framework

Framework zur Beschreibung der Fahrzeugarchitektur. 198, 231,

AUTomotive Open System ARchitecture

Offener Standard für eine Automotive Software-Architektur. 42, 231,

AUTOSAR Adaptive Platform

Serviceorientierte Softwarearchitektur, spezialisiert auf umfangreichere ECUs, zum Beispiel für Fahrerassistenzfunktionen. 54, 74, 231,

AUTOSAR Basic Software

Unterste Ebene von AUTOSAR Classic. Stellt die Abstraktionsschicht von der zugrundeliegenden Hardware dar. 55, 232,

AUTOSAR Classic Platform

Signalbasierte Softwarearchitektur mit Fokus auf einfachen, echtzeitfähigen ECUs. 54, 74, 232,

AUTOSAR Runtime for Adaptive Applications

Laufzeitumgebung für Ausführung der FCs von AUTOSAR Adaptive 78, 231,

Average Precision

Metrik zur Bewertung einer Objekterkennung, Fläche unter der Precision/Recall-Kurve. 35, 231,

B

Bundesanstalt für Straßenwesen

Forschungsinstitut des Bundesministeriums für Verkehr und digitale Infrastruktur. 15, 232,

C

Central Processing Unit

Zentrale digitale Schaltung, die Berechnungen von Algorithmen durchführt. 232,

Commercial off-the-shelf

Serienprodukte, die identisch aufgebaut in hohen Stückzahlen gefertigt werden und daher zu vergleichsweise geringen Stückkosten verfügbar sind. 100, 232,

Continuous Integration, Continuous Delivery/Deployment

Automatisierung von Bauen, Testen und Deployment einer Softwarekomponente 81, 232,

Contract

Formale Definition der Schnittstelle eines Services. 62, 63

Controller Area Network

Signalbasierter Fahrzeugbus. Dabei werden feste Nachrichtentypen versandt, die ihrerseits die Arbitrierung bestimmen. 41, 232,

D

Data Distribution Service

Middleware zur Kommunikation zwischen Teilnehmern einer serviceorientierten Architektur. 62, 232,

Diagnostic Trouble Code

Standardisierter Fehlercode eines Fahrzeuges zur Diagnose. 232

Differential Global Positioning System

Erweiterung zu GPS, das Korrekturdaten von Bodenstationen zur Verbesserung der Genauigkeit nutzt. 26, 232,

Digital-Analog-Wandler

Wandelt ein digitales in ein analoges Signal. 130, 232,

Domain Control Unit

Eine DCU stellt die Hauptsoftwarefunktionalität für eine bestimmte Domäne im Fahrzeug zur Verfügung, Basisfunktionen wie die Regelung von Aktoren werden an sogenannte intelligente Aktoren delegiert. 232,

Domain Name System

Dezentrales System, um Namen von mit dem Netzwerk verbundenen Teilnehmern auf ihre IP-Adressen aufzulösen. 105, 232,

E

Electronic Control Unit

Auch Steuergerät genannt. Es handelt sich hierbei um ein beliebiges eingebettetes System im Fahrzeug, das Steuer- und Regelungsaufgaben basierend auf Eingabewerten von Sensoren übernimmt. Ein Beispiel dafür ist das Motorsteuergerät. 233,

Elektrisch/Elektronische Architektur

Umfasst die Elektrik und Elektronik im Fahrzeug von der Lage der Komponenten und dem Kabelbaum bis hinauf zum kundenerlebbaren Funktionsumfang und der dazugehörigen Software (s. auch Abbildung 3.4 auf Seite 43). 19, 39, 232,

End of Life

Übergang von Serienproduktion zur Abkündigung. Nur noch kleine Stückzahlen werden zu hohen Preisen angeboten. 2, 233,

Extended Kalman-Filter

Nichtlineare Version des Kalman-Filters. 29, 134, 233,

F

Failure Tolerance Time Interval

Zeit zwischen Auftreten eines Defekts und dem Auftreten eines daraus resultierenden gefährlichen Ereignisses. 96, 233,

Fault Reaction Time

Summe der Zeitspanne von Auftreten bis Erkennung eines Defekts und der Zeitspanne von Erkennung bis Behebung. 96, 233,

Field Programmable Gate Array

Ein FPGA ist ein integrierter Schaltkreis, der nach seiner Herstellung beim Kunden mittels einer Hardware Description Language (HDL) programmiert werden kann. Daher die Bezeichnung „field programmable". 233,

Functional Cluster

Stellen Anwendungsschnittstellen zu den von AUTOSAR Adaptive angebotenen Services her. 233,

G

Global Data Space

Dezentrale Verteilung von Informationen über Services auf alle Kommunikationsteilnehmer. 64, 233,

Global Navigation Satellite System

System zur Positionsbestimmung basierend auf Signalen von Navigationssatelliten. Dabei stellt GNSS einen Sammelbegriff für Systeme wie GPS (USA), GLONASS (Russland), Galileo (EU) und Beidou (VR China) dar. 26, 233,

Graphics Processing Unit

Elektronische Schaltung, die auf Grafikberechnungen spezialisiert ist. Durch die hohe Anzahl an parallelen Recheneinheiten können auch z.B. neuronale Netze damit effektiv berechnet werden. 32, 233,

H

Human Machine Interface

Schnittstelle der Interaktion zwischen Mensch und Maschine, zum Beispiel das Kombiinstrument eines Fahrzeuges. 127, 233,

I

Implementierung

Technische Realisierung der Funktionalität eines Services. 63

Inertial Measurement Unit

Räumliche Kombination mehrerer Inertialsensoren zur Messung von Beschleunigungen, Drehraten und gegebenenfalls weiteren wie beispielsweise dem Magnetfeld. 25, 233,

Inter-Process Communication

Austausch von Daten zwischen unterschiedlichen auf einem gemeinsamen Betriebssystem laufenden Prozessen. 81, 233,

Interface Definition Language

Sprache, um die Schnittstellen eines Programms zu beschreiben. 62, 233,

Intersection over Union

Metrik zur Bewertung der Accuracy einer Objekterkennung auf einem bestimmten Datensatz. Überlappung der Bounding Box des erkannten Objekts mit der Ground Truth. 34, 233,

J

JavaScript Object Notation

Menschenlesbares Datenformat zur Serialisierung von verschiedenen Datentypen. 166, 234,

K

Kernel-based Virtual Machine

Virtualisierungsmodul im Linuxkernel. 87, 234,

L

Light Detection and Ranging

Ursprünglich ein Kofferwort aus Laser und Radar, stellt es eine dem Radar ähnliche Methode zur optischen Abstands- und Geschwindigkeitsmessung dar. Dazu wird die Laufzeit ausgesendeter Laserpulse bis zur Detektion ihrer Reflektion gemessen. 17, 234,

Local Area Network

Ein Kommunikationsnetzwerk, das sich nur über eine begrenzte Entfernung erstreckt, zum Beispiel ein Gebäude oder ein Fahrzeug. 27, 234,

Local Interconnect Network

Günstigere Alternative zum CAN-Bus, bei der ein Master bis zu 15 angeschlossene Slaves abfragt. 75, 234,

M

mean Average Precision

Durchschnitt der AP über alle Klassen. 35, 234,

Micro-Electro-Mechanical Systems

Bauteile im Mikrometerbereich, die elektrische/elektronische mit mechanischen Baugruppen zu z.B. Sensoren oder Aktoren kombinieren. 25, 234,

Multiprocessor System-on-Chip

Häufig in eingebetten Systemen eingesetzt, enthält es mehrere (häufig heterogene) Verarbeitungseinheiten, beispielsweise mehrere CPU-Kerne. Im Gegensatz zu einer Multicore-CPU sind auch Komponenten

wie RAM, ROM oder Peripherie und Kommunikationsschnittstellen wie CAN oder Ethernet enthalten. 202, 234,

N

National Highway Traffic Safety Administration

US-Bundesbehörde für Straßen- und Fahrzeugsicherheit. 15, 234,

Network Time Protocol

Protokoll zur Zeitsynchronisierung von Teilnehmern eines Netzwerkes. 234

Next Generation Electrical Architecture

Framework zur Beschreibung der Fahrzeugarchitektur. 198, 234,

O

Object and Event Detection and Response

Beobachtung der Fahrzeugumgebung und Reaktion auf diese. 16, 234,

Offene Systeme und deren Schnittstellen für die Elektronik im KFZ

Gegründet im Jahre 1993 mit dem Ziel, eine Standardsoftwarearchitektur für ECUs in einem Fahrzeug zu definieren. Teilweise in ISO 17356 standardisiert. 53, 235,

Onboard-Diagnose

Selbstdiagnose eines Fahrzeuges, hauptsächlich abgas- und motorrelevante Signale. 234

Open Systems Interconnection Model

Schichtenmodell der verschiedenen Abstraktionsebenen der Netzwerkkommunikation. 51, 235,

Operational Design Domain

Die Umgebung einer automatisierten Fahrfunktion, in der diese fehlerfrei arbeiten soll. 16, 234,

Original Equipment Manufacturer

In diesem Kontext der Fahrzeughersteller, der dieses in Verkehr bringt. 81, 234,

over-the-air

Verteilung von Updates von Software über kabellose Medien. 9, 235,

P

Portable Operating System Interface

Familie von Standards für definierte und kompatible Schnittstellen zwischen Betriebssystemen. 66, 235,

Q

Quality of Service

Bei Netzwerken Maßnahmen zur Priorisierung von Paketen und Sicherstellung ihrer pünktlichen Zustellung. 52, 235,

R

Random Access Memory

Speicher eines Computer, der in beliebiger Reihenfolge gelesen und geschrieben werden kann. 45, 235,

Read-only Memory

Speicher, der nach der Herstellung nicht mehr (ohne gesteigerten Aufwand) beschrieben werden kann. 45, 235,

Real Time Kinematic

Erweiterung zu GPS, die Messungen der Phase von Satellitensignalen durch eine Referenzstation zur Verbesserung der Genauigkeit nutzt. 26, 235,

Real-time Publish-Subscribe Protocol

Definiert das DDS zugrundeliegende Kommunikationsprotokoll. 67, 235,

Remote Procedure Call

Eine entfernte (z.B. auf einem anderen PC im Netzwerk) Prozedur wird wie eine lokale aufgerufen. 235,

Robot Operating System

Eine Plattform zur Entwicklung von Software für Roboter. 44, 65, 132, 235,

Robot Operating System 2

Eine Plattform zur Entwicklung von Software für Roboter und Fahrzeuge, serviceorientiert. 52, 235,

S

Scalable service-Oriented MiddlewarE over IP

Middleware zur Kommunikation zwischen Teilnehmern einer serviceorientierten Architektur. 75, 236,

Schnittstelle

Abstraktion der Funktionalität eines Services nach außen. 62, 221, 230

Service

Eigenständige Blackbox mit definierten Schnittstellen und Funktionalitäten. 5, 10, 52, 60, 62, 63, 90, 97, 100, 103, 109–112, 123, 166, 183, 187, 197, 198, 202, 217, 221, 224, 228

Service Level Agreement

Vereinbarung zwischen Service-Anbieter und Service-Konsument über Eigenschaften des Services wie Qualität und Verfügbarkeit. 64, 235,

Service-Bus

Verbindet die Teilnehmer einer serviceorientierten Architektur miteinander. 65

Service-Repository

Stellt Informationen zur Verfügung, die es ermöglichen, Services zu entdecken und zu nutzen. 64

serviceorientierte Architektur

Eine Softwarearchitektur, die auf Services basiert. Diese stellen eigenständige Blackboxes mit definierten Schnittstellen und Funktionalitäten dar. 49, 236,

Simultaneous Localization and Mapping

Gleichzeitige Erstellung einer Karte der Umgebung und Lokalisierung in dieser. 28, 235,

Softwarearchitektur

Umfasst die zugrundeliegende Struktur eines Softwaresystems. 39, 236,

Softwarekomponente

Funktional und logisch zusammengehörige Softwareartefakte, die nach außen hin von einer gemeinsamen Schnittstelle gekapselt werden. 5, 44, 48, 50, 52, 56, 60, 75, 83, 86, 90, 92–98, 100–103, 105–112, 117, 121, 123, 147, 154–158, 163–167, 173, 182, 186, 199, 221

SOME/IP Service Discovery Protocol

Erweiterung von SOME/IP zur Bekanntmachung und Auffindung von Services. 197, 236,

T

Technologie-Gateway

Kapselt nicht serviceorientierten Code und versieht ihn nach außen hin mit serviceorientierten Schnittstellen 63, 121, 161, 163, 187

Triple Modular Redundancy

Redundanzkonzept, bei dem Berechnungen dreifach ausgeführt werden und das gültige Ergebnis durch einen Voter per Mehrheitsentscheid bestimmt wird. 109, 110, 201, 236,

U

Ultra High Frequency

Frequenzen elektromagnetischer Wellen im Bereich 300 bis 3000 MHz. 27, 236,

Ultra-wideband

Übertragung von Informationen über ein sehr breites Frequenzspektrum von mindestens 500 MHz. 27, 236,

V

Virtual Private Network

Spannt ein privates Netzwerk über ein öffentliches Netzwerk (wie das Internet) auf. Üblicherweise verschlüsselt. 104, 236,

W

Web Service Description Language

Sprache, die dazu dient, die von Web-Services angebotene Funktionalität zu beschreiben. 62, 236,

Wide Area Network

Ein Kommunikationsnetzwerk, das sich über weite Entfernungen erstreckt. Beispiel: das Internet. 152, 236,

Abkürzungsverzeichnis

A

AA Adaptive Application 79, *Glossar:* Adaptive Application

AAF Automotive Architecture Framework 198, *Glossar:* Automotive Architecture Framework

ACC Adaptive Cruise Control 15, *Glossar:* Adaptive Cruise Control

ADC Analog-Digital-Wandler 21, 130, *Glossar:* Analog-Digital-Wandler

ADF Architectural Design Framework 199, *Glossar:* Architectural Design Framework

ADL Architecture Description Language 42, 199, *Glossar:* Architecture Description Language

AFAS Architecture Framework for Automotive Systems 199, *Glossar:* Architecture Framework for Automotive Systems

AP Average Precision 35, 225, *Glossar:* Average Precision

API Application Programming Interface 68, 197, 198, 208, *Glossar:* Application Programming Interface

ARA AUTOSAR Runtime for Adaptive Applications 78, 219, *Glossar:* AUTOSAR Runtime for Adaptive Applications

AUTOSAR AUTomotive Open System ARchitecture 42, 44, *Glossar:* AUTomotive Open System ARchitecture

AUTOSAR Adaptive AUTOSAR Adaptive Platform 54, 59, 62–64, 73–81, 84, 85, 109, 118, 196, 197, 199, 202, 207, 208, 219, 220, 223, 238, 241,

Glossar: AUTOSAR Adaptive Platform

AUTOSAR Classic AUTOSAR Classic Platform 54, 55, 59, 74–76, 197–199, 208, 220, 238, *Glossar:* AUTOSAR Classic Platform

B

BASt Bundesanstalt für Straßenwesen 15, *Glossar:* Bundesanstalt für Straßenwesen

BSW AUTOSAR Basic Software 55, *Glossar:* AUTOSAR Basic Software

C

CAN Controller Area Network 41, 45, 46, 55, 59, 75, 126, 128–130, 145, 202, 225, 226, *Glossar:* Controller Area Network

CI/CD Continuous Integration, Continuous Delivery/Deployment 81, 93, 106, 117, 147, 166, *Glossar:* Continuous Integration, Continuous Delivery/Deployment

COTS Commercial off-the-shelf 100, 182, 183, 187, *Glossar:* Commercial off-the-shelf

CPU Central Processing Unit 2, 3, 44, 45, 55, 77, 93, 109, 130, 145, 150, 156, 172, 173, 175, 209, *Glossar:* Central Processing Unit

D

DAC Digital-Analog-Wandler 130, *Glossar:* Digital-Analog-Wandler

DCU Domain Control Unit 40, *Glossar:* Domain Control Unit

DDS Data Distribution Service 62, 64, 66, 67, 72, 84, 85, 118, 196–198, 202, 207, 208, 228, 241, *Glossar:* Data Distribution Service

DGPS Differential Global Positioning System 26, 135, *Glossar:* Differential Global Positioning System

DNS Domain Name System 105, 164, *Glossar:* Domain Name System

DTC Diagnostic Trouble Code *Glossar:* Diagnostic Trouble Code

E

E/E-Architektur Elektrisch/Elektronische Architektur 19, 39, 41, 42, 46, 49, 95, 126, 198,

Glossar: Elektrisch/Elektronische Architektur

ECU Electronic Control Unit 2, 3, 5, 8, 9, 11, 40, 41, 44, 46–50, 53–56, 58, 59, 64, 74, 77, 78, 80, 93, 95, 96, 101, 102, 104, 105, 107–109, 111, 112, 117, 122, 126–130, 132, 145, 148–151, 156–159, 164, 165, 167, 169, 171–173, 179, 182, 186, 202, 209, 216, 220, 226, 239, *Glossar:* Electronic Control Unit

EKF Extended Kalman-Filter 29, 134, *Glossar:* Extended Kalman-Filter

EoL End of Life 2, 4, *Glossar:* End of Life

F

FC Functional Cluster 78, 220, *Glossar:* Functional Cluster

FPGA Field Programmable Gate Array 10, 44, 74, 95, 96, 101, 156, 202, *Glossar:* Field Programmable Gate Array

FRT Fault Reaction Time 96, *Glossar:* Fault Reaction Time

FTTI Failure Tolerance Time Interval 96, *Glossar:* Failure Tolerance Time Interval

G

GDS Global Data Space 64, 197, *Glossar:* Global Data Space

GNSS Global Navigation Satellite System 26–28, 134–136, 203, *Glossar:* Global Navigation Satellite System

GPU Graphics Processing Unit 32, 44, 74, 89, 90, 101, 109, 146, 148, 150, 156–159, 172–175, 178, *Glossar:* Graphics Processing Unit

H

HMI Human Machine Interface 127, *Glossar:* Human Machine Interface

I

IDL Interface Definition Language 62, *Glossar:* Interface Definition Language

IMU Inertial Measurement Unit 25, 28, 136, *Glossar:* Inertial Measurement Unit

IoU Intersection over Union 34, 35, *Glossar:* Intersection over Union

IPC Inter-Process Communication 81, *Glossar:* Inter-Process

Communication

J

JSON JavaScript Object Notation 166, *Glossar:* JavaScript Object Notation

K

KVM Kernel-based Virtual Machine 87, *Glossar:* Kernel-based Virtual Machine

L

LAN Local Area Network 27, 152, *Glossar:* Local Area Network

Lidar Light Detection and Ranging 17, 21, 28, 195, 203, *Glossar:* Light Detection and Ranging

LIN Local Interconnect Network 75, *Glossar:* Local Interconnect Network

M

mAP mean Average Precision 35, 37, 193, *Glossar:* mean Average Precision

MEMS Micro-Electro-Mechanical Systems 25, *Glossar:* Micro-Electro-Mechanical Systems

MPSoC Multiprocessor System-on-Chip 202, *Glossar:* Multiprocessor System-on-Chip

N

NGEA Next Generation Electrical Architecture 198, 209, *Glossar:* Next Generation Electrical Architecture

NHTSA National Highway Traffic Safety Administration 15, *Glossar:* National Highway Traffic Safety Administration

NTP Network Time Protocol *Glossar:* Network Time Protocol

O

OBD Onboard-Diagnose *Glossar:* Onboard-Diagnose

ODD Operational Design Domain 16, *Glossar:* Operational Design Domain

OEDR Object and Event Detection and Response 16, *Glossar:* Object and Event Detection and Response

OEM Original Equipment Manufacturer 81, *Glossar:* Original Equipment Manufacturer

OSEK Offene Systeme und deren Schnittstel-

len für die Elektronik im KFZ 53, 54, 59, 206, *Glossar:* Offene Systeme und deren Schnittstellen für die Elektronik im KFZ

OSI Open Systems Interconnection Model 51, 54, *Glossar:* Open Systems Interconnection Model

OTA over-the-air 9, 74, 81, 104, *Glossar:* over-the-air

P

POSIX Portable Operating System Interface 66, 78, 85, 206, 207, *Glossar:* Portable Operating System Interface

Q

QoS Quality of Service 52, 59, 72, 73, 115, 116, 196–198, 208, *Glossar:* Quality of Service

R

RAM Random Access Memory 45, 77, 132, 144, 145, 150, 156, 172, 173, 177, 179, *Glossar:* Random Access Memory

ROM Read-only Memory 45, *Glossar:* Read-only Memory

ROS Robot Operating System 44, 56–58, 60, 63, 65–68, 71, 73, 75, 95, 121, 130, 132, 133, 135, 136, 139, 142–145, 160–163, 186, 192, 199, *Glossar:* Robot Operating System

ROS2 Robot Operating System 2 52, 56, 59, 60, 62–64, 66–68, 71–73, 75, 79, 84, 85, 109, 112, 118, 119, 121, 160–163, 186, 192, 196, 197, 202, 207, 208, 238, 241, *Glossar:* Robot Operating System 2

RPC Remote Procedure Call 57, *Glossar:* Remote Procedure Call

RTK Real Time Kinematic 26, 135, *Glossar:* Real Time Kinematic

RTPS Real-time Publish-Subscribe Protocol 67, 196, 198, 202, 208, *Glossar:* Real-time Publish-Subscribe Protocol

S

SLA Service Level Agreement 64, *Glossar:* Service Level Agreement

SLAM Simultaneous Localization and Mapping 28, 136, *Glossar:* Simultaneous

Localization and Mapping

SOA serviceorientierte Architektur 49, 55, 59–61, 63, 65, 74, 83, 100, 118, 121, 197, 199, 202, *Glossar:* serviceorientierte Architektur

SOME/IP Scalable service-Oriented MiddlewarE over IP 75, 81, 118, 196–198, 208, 241, *Glossar:* Scalable service-Oriented MiddlewarE over IP

SOME/IP-SD SOME/IP Service Discovery Protocol 197, *Glossar:* SOME/IP Service Discovery Protocol

SW-Architektur Softwarearchitektur 39, 48–51, 59, 60, 198, *Glossar:* Softwarearchitektur

T

TMR Triple Modular Redundancy 109, 110, 201, *Glossar:* Triple Modular Redundancy

U

UHF Ultra High Frequency 27, *Glossar:* Ultra High Frequency

UWB Ultra-wideband 27, 134, 135, *Glossar:* Ultra-wideband

V

VPN Virtual Private Network 104, 149, 152, 167, 168, 186, *Glossar:* Virtual Private Network

W

WAN Wide Area Network 152, *Glossar:* Wide Area Network

WSDL Web Service Description Language 62, *Glossar:* Web Service Description Language

Abbildungsverzeichnis

Tabellenverzeichnis

Verwendete Quellen

[AAB+16] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu und X. Zheng, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems, November 2-4, 2016, Savannah, GA, USA*, eng. Berkeley, CA: USENIX Association, 2016, 786 S., Version 2 updates only the metadata, to correct the formatting of Martín Abadi's name, ISBN: 9781931971331. Adresse: https://www.usenix.org/conference/osdi16/program.

[Abe02] H. Abel. „GPS: Global Positioning System - Funktionsweise und mathematische Grundlagen", Hochschule Esslingen. (14.11.2002), Adresse: http://www2.hs-esslingen.de/~abel/gps/Abel-GPS.htm (besucht am 30. 11. 2020).

[Ama19] Amazon Web Services. „FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions". (8.11.2019), Adresse: https://www.freertos.org/ (besucht am 09. 11. 2019).

[APCR18] Andreas Geiger, Philip Lenz, Christoph Stiller und Raquel Urtasun. „The KITTI Vision Benchmark Suite". (2018), Adresse: http://www.cvlibs.net/datasets/kitti/.

[AUT14] AUTOSAR, *Example for a Serialization Protocol (SOME/IP)*, AUTOSAR, Hrsg., 2014.

[AUT17a] ——, „How to join AUTOSAR". en. Copyright: AUTOSAR development cooperation, AUTOSAR. (2017), Adresse: https://www.autosar.org/how-to-join/ (besucht am 23. 09. 2020).

[AUT17b] ——, *SOME/IP Service Discovery Protocol Specification*, AUTOSAR, Hrsg., 2017.

[AUT18] AUTOSAR Web Team, *AUTOSAR Introduction*, 2018.

[AUT19a] AUTOSAR, *AUTOSAR Layered Software Architecture*, AUTOSAR, Hrsg., AUTOSAR, 2019. Adresse: https://www.autosar.org/fileadmin/user_upload/standards/classic/19-11/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf (besucht am 13. 05. 2020).

[AUT19b] ——, *Explanation of Adaptive Platform Design*, AUTOSAR, Hrsg., AUTOSAR, 2019. Adresse: https://www.autosar.org/fileadmin/user_upload/standards/adaptive/19-11/AUTOSAR_EXP_PlatformDesign.pdf (besucht am 13. 05. 2020).

[BBK14] B. Beck, R. Baxley und J. Kim, „Real-time, anchor-free node tracking using ultrawideband range and odometry data“, in *2014 IEEE International Conference on Ultra-WideBand (ICUWB)*, Sept, 2014, S. 286–291. DOI: 10.1109/ICUWB.2014.6958994.

[BCK13] L. Bass, P. Clements und R. Kazman, *Software architecture in practice*, eng, 3. ed., Ser. Always learning. Upper Saddle River, NJ: Addison-Wesley/Pearson, 2013, 589 S., ISBN: 9780321815736.

[BF20] G. Biggs und T. Foote. „Managed nodes“. (17.11.2020), Adresse: https://design.ros2.org/articles/node_lifecycle.html (besucht am 02. 12. 2020).

[BFP09] H. Badino, U. Franke und D. Pfeiffer, „The Stixel World - A Compact Medium Level Representation of the 3D-World“, in *Pattern Recognition*, Ser. Lecture Notes in Computer Science, J. Denzler, G. Notni und H. Süße, Hrsg., Bd. 5748, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, S. 51–60, ISBN: 978-3-642-03797-9. DOI: 10.1007/978-3-642-03798-6_6.

[BGG+14] C. Buckl, M. Geisinger, D. Gulati, F. J. Ruiz-Bertol und A. Knoll, „CHROMOSOME“, *ACM SIGBED Review*, Jg. 11, Nr. 3, S. 36–39, 2014, ISSN: 1551-3688. DOI: 10.1145/2692385.2692391.

[BGM+09] M. Broy, M. Gleirscher, S. Merenda, D. Wild, P. Kluge und W. Krenzer, „Toward a Holistic and Standardized Automotive Architecture Description“, *Computer*, Jg. 42, Nr. 12, S. 98–101, 2009, ISSN: 0018-9162. DOI: 10.1109/MC.2009.413.

[BHR+06] R. Balani, C.-C. Han, R. K. Rengaswamy, I. Tsigkogiannis und M. Srivastava, „Multi-level software reconfiguration for sensor networks“, in *Proceedings of the 6th ACM & IEEE International conference on Embedded software - EMSOFT '06*, (Seoul, Korea), S. L. Min und W. Yi, Hrsg., New York, New York, USA: ACM Press, 2006, S. 112, ISBN: 1595935428. DOI: 10.1145/1176887.1176904.

[BKPS07] M. Broy, I. H. Kruger, A. Pretschner und C. Salzmann, „Engineering Automotive Software“, *Proceedings of the IEEE*, Jg. 95, Nr. 2, S. 356–373, 2007, ISSN: 0018-9219. DOI: 10.1109/JPROC.2006.888386.

[BLS+20] M. Böhme, A. Lauber, M. Stang, L. Pan und E. Sax, „Using Machine Learning to Optimize Energy Consumption of HVAC Systems in Vehicles“, in *Human Interaction and Emerging Technologies, Proceedings of the 1st International Conference on Human Interaction and Emerging Technologies (IHIET 2019), August 22-24, 2019, Nice, France*, Ser. Advances in Intelligent Systems and Computing volume 1018, R. Taiar, Hrsg., Bd. 1018, Cham: Springer International Publishing, 2020, S. 706–712, ISBN: 978-3-030-25628-9. DOI: 10.1007/978-3-030-25629-6_110.

[BP20] A. Bucaioni und P. Pelliccione, „Technical Architectures for Automotive Systems“, in *2020 IEEE International Conference on Software Architecture (ICSA)*, (Salvador, Brazil), IEEE, 16.03.2020 - 20.03.2020, S. 46–57, ISBN: 978-1-7281-4659-1. DOI: 10.1109/ICSA47634.2020.00013.

[Bri20] Brian Gerkey. „Why ROS 2?“ (30.07.2020), Adresse: https://design.ros2.org/articles/why_ros2.html (besucht am 08. 08. 2020).

[BRKW17] S. Brunner, J. Roder, M. Kucera und T. Waas, „Automotive E/E-architecture enhancements by usage of ethernet TSN“, in *2017 13th Workshop on Intelligent Solutions in Embedded Systems (WISES)*, (Hamburg, Germany), IEEE, 2017, S. 9–13, ISBN: 978-1-5386-1157-9. DOI: 10.1109/WISES.2017.7986925.

[Bun18] Bundesnetzagentur, *Allgemeinzuteilung von Frequenzen für die Nutzung durch Ultrabreitband-Anwendungen (UWB), Vfg. 73 /2018*, Version Vfg. 73 /2018, 2018.

[CC12] A. Corsaro und D. C., „The Data Distribution Service – The Communication Middleware Fabric for Scalable and Extensible Systems-of-Systems“, in *System of Systems*, A. V. Gheorghe, Hrsg., InTech, 2012, ISBN: 978-953-51-0101-7. DOI: 10.5772/30322.

[Com19] T. Com. „AKIT: Autonomie-KIT für seriennahe Arbeitsfahrzeuge zur vernetzten und assistierten Bergung von Gefahrenquellen“. (16.05.2019), Adresse: http://www.a-kit.de/ (besucht am 15. 11. 2019).

[COR+16] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth und B. Schiele, „The Cityscapes Dataset for Semantic Urban Scene Understanding“, S. 3213–3223, 2016. DOI: 10.1109/CVPR.2016.350.

[CPL14] D. Chen, Z. Peng und X. Ling, „A low-cost localization system based on artificial landmarks with two degree of freedom platform camera“, in *2014 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, 2014, S. 625–630. DOI: 10.1109/ROBIO.2014.7090400.

[Daj19] Y. Dajsuren, „Defining Architecture Framework for Automotive Systems“, in *Automotive systems and software engineering, State of the art and future trends*, Y. Dajsuren und M. van den Brand, Hrsg., Cham: Springer, 2019, S. 141–168, ISBN: 978-3-030-12156-3. DOI: 10.1007/978-3-030-12157-0_7.

[DB16] M. Dikmen und C. M. Burns, „Autonomous Driving in the Real World“, in *Proceedings of the 8th International Conference on Automotive User Interfaces and Interactive Vehicular Applications - Automotive'UI 16*, (Ann Arbor, MI, USA), P. Green, S. Boll, G. Burnett, J. Gabbard und S. Osswald, Hrsg., New York, New York, USA: ACM Press, 2016, S. 225–228, ISBN: 9781450345330. DOI: 10.1145/3003715.3005465.

[DDS+09] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li und L. Fei-Fei, „ImageNet: A large-scale hierarchical image database“, in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, (Miami, FL), IEEE, 20.06.2009 - 25.06.2009, S. 248–255, ISBN: 978-1-4244-3992-8. DOI: 10.1109/CVPR.2009.5206848.

[deb20] debiman. „chroot(2) — manpages-de-dev — Debian buster — Debian Manpages“. (8.08.2020), Adresse: https://manpages.debian.org/buster/manpages-de-dev/chroot.2.de.html (besucht am 15. 08. 2020).

[DHS15] J. Dai, K. He und J. Sun, „BoxSup: Exploiting Bounding Boxes to Supervise Convolutional Networks for Semantic Segmentation“, in *2015 IEEE International Conference on Computer Vision, 11-18 December 2015, Santiago, Chile : proceedings*, (Santiago, Chile), hrsg. von R. Bajcsy, G. Hager und Y. Ma, IEEE International Conference on Computer Vision, Institute of Electrical and Electronics Engineers und ICCV, Piscataway, NJ: IEEE, 2015, S. 1635–1643, ISBN: 978-1-4673-8391-2. DOI: 10.1109/ICCV.2015.191.

[Doc20] Docker Documentation. „Swarm mode overview“. (24.08.2020), Adresse: https://docs.docker.com/engine/swarm/ (besucht am 25. 08. 2020).

[EAS19] EAST-ADL Association. „EAST-ADL Specification“. (17.10.2019), Adresse: https://www.east-adl.info/Specification.html (besucht am 23. 11. 2019).

[EvW+10] M. Everingham, L. van Gool, C. K. I. Williams, J. Winn und A. Zisserman, „The Pascal Visual Object Classes (VOC) Challenge“, *International Journal of Computer Vision*, Jg. 88, Nr. 2, S. 303–338, 2010, PII: 275, ISSN: 0920-5691. DOI: 10.1007/s11263-009-0275-4.

[FK16] M. H. Farzaneh und A. Knoll, „An ontology-based Plug-and-Play approach for in-vehicle Time-Sensitive Networking (TSN)“, in *The 7th IEEE Annual Information Technology, Electronics & Mobile Communication Conference, IEEE IEMCON - 2016 : 13-15 October, 2016, University of British Columbia, Vancouver, Canada*, (Vancouver, BC, Canada), S. Chakrabati, Hrsg., IEEE Annual Information Technology, Electronics & Mobile Communication Conference, Institute of Electrical and Electronics Engineers und IEEE IEMCON, Piscataway, NJ: IEEE, 2016, S. 1–8, ISBN: 978-1-5090-0996-1. DOI: 10.1109/IEMCON.2016.7746299.

[Flü12] H. Flühr, „Flugzeugsysteme“, in *Avionik und Flugsicherungstechnik*, H. Flühr, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 307–329, ISBN: 978-3-642-33575-4. DOI: 10.1007/978-3-642-33576-1_11.

[GGT13] H. G. C. Góngora, T. Gaudré und S. Tucci-Piergiovanni, „Towards an Architectural Design Framework for Automotive Systems Development“, in *Complex Systems Design & Management, Proceedings of the Third International Conference on Complex Systems Design & Management CSD&M 2012*, M. Aiguier, Y. Caseau, D. Krob und A. Rauzy, Hrsg., Bd. 14, Berlin und Heidelberg: Springer, 2013, S. 241–258, ISBN: 978-3-642-34403-9. DOI: 10.1007/978-3-642-34404-6_16.

[GH04] H. Gomaa und M. Hussein, „Dynamic Software Reconfiguration in Software Product Families“, in *Software product-family engineering, 5th international workshop, PFE 2003, Siena, Italy, November 4 - 6, 2003 ; revised papers*, Ser. Lecture Notes in Computer Science 3014, F. van der Linden, Hrsg., Bd. 3014, Berlin: Springer, 2004, S. 435–444, ISBN: 978-3-540-21941-5. DOI: 10.1007/978-3-540-24667-1_33.

[Gil12] D. L. Gilblom, „Cameras“, in *Machine Vision Handbook*, B. G. Batchelor, Hrsg., London: Springer London, 2012, S. 355–476, ISBN: 978-1-84996-168-4. DOI: 10.1007/978-1-84996-169-1_10.

[Git20] Gitlab. „The first single application for the entire DevOps lifecycle - GitLab“. (25.08.2020), Adresse: https://about.gitlab.com/ (besucht am 25. 08. 2020).

[GKS+16] D. Gangadharan, J. H. Kim, O. Sokolsky, B. Kim, C.-W. Lin, S. Shiraishi und I. Lee, „Platform-Based Plug and Play of Automotive Safety Features: Challenges and Directions (Invited Paper)“, in *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2016 : 17-19 August 2016, Daegu, South Korea : proceedings*, (Daegu, South Korea), hrsg. von S. H. Son, IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Institute of Electrical and Electronics Engineers und RTCSA, Piscataway, NJ: IEEE, 2016, S. 76–84, ISBN: 978-1-5090-2479-7. DOI: 10.1109/RTCSA.2016.18.

[GLSU13] A. Geiger, P. Lenz, C. Stiller und R. Urtasun, „Vision meets robotics: The KITTI dataset“, *The International Journal of Robotics Research*, Jg. 32, Nr. 11, S. 1231–1237, 2013, ISSN: 0278-3649. DOI: 10.1177/0278364913491297.

[GLU12] A. Geiger, P. Lenz und R. Urtasun, „Are we ready for autonomous driving? The KITTI vision benchmark suite“, S. 3354–3361, 2012. DOI: 10.1109/CVPR.2012.6248074.

[Goo20a] Google Trends. „Google Trends, Interesse im zeitlichen Verlauf“, Vergleich Docker, LXC, LXD, Podman. (15.08.2020), Adresse: https://trends.google.de/trends/explore?date=2015-08-15%202020-08-15&q=Docker,LXC,LXD,Podman (besucht am 15. 08. 2020).

[Goo20b] ——, „Google Trends, Interesse im zeitlichen Verlauf“, Vergleich LXC, LXD, Podman. (15.08.2020), Adresse: https://trends.google.de/trends/explore?date=2015-08-15%202020-08-15&q=LXC,LXD,Podman,Docker (besucht am 15. 08. 2020).

[GRB+17] T. Gaspar, B. Ridge, R. Bevec, M. Bem, I. Kovac, A. Ude und Z. Gosar, „Rapid hardware and software reconfiguration in a robotic workcell“, in *Proceedings of the 2017 18th International Conference on Advanced Robotics (ICAR), Hong Kong, China, July 2017*, (Hong Kong, China), hrsg. von M. Q.-H. Meng, International Conference on Advanced Robotics und ICAR, Piscataway, NJ: IEEE, 2017, S. 229–236, ISBN: 978-1-5386-3157-7. DOI: 10.1109/ICAR.2017.8023523.

[Hir08] H. Hirschmüller, „Stereo processing by semiglobal matching and mutual information“, eng, *IEEE transactions on pattern analysis and machine intelligence*, Jg. 30, Nr. 2, S. 328–341, 2008, Journal Article, ISSN: 0162-8828. DOI: 10.1109/TPAMI.2007.1166. eprint: 18084062.

[HKRA] W. Hess, D. Kohler, H. Rapp und D. Andor, „Real-time loop closure in 2D LIDAR SLAM“, in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, (Stockholm, Sweden), S. 1271–1278. DOI: 10.1109/ICRA.2016.7487258.

[HZC+17] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto und H. Adam, *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*, 17.04.2017. Adresse: https://arxiv.org/pdf/1704.04861.

[HZRS16] K. He, X. Zhang, S. Ren und J. Sun, „Deep Residual Learning for Image Recognition“, in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (Las Vegas, NV, USA), IEEE, 27.06.2016 - 30.06.2016, S. 770–778, ISBN: 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.90.

[IDC20] IDC. „Absatz von Smartphones weltweit in den Jahren 2009 bis 2019 (in Millionen Stück)“, zitiert nach de.statista.com. (2020), Adresse: https://de.statista.com/statistik/daten/studie/173049/umfrage/weltweiter-absatz-von-smartphones-seit-2009/ (besucht am 10. 08. 2020).

[IEE17] IEEE, Hrsg., *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) Base Specifications, Issue 7*, IEEE, Version 1, Piscataway, NJ, USA: IEEE, 2017. DOI: 10.1109/IEEESTD.2008.4694976.

[Ima10] ImageNet. „Summary and Statistics“. (30.04.2010), Adresse: http://image-net.org/about-stats (besucht am 18. 12. 2018).

[Inf19] Infineon Technologies AG. „32-bit AURIX™ Microcontroller based on TriCore™ - Infineon Technologies“. Copyright: Copyright Infineon Technologies AG - all rights reserved, Infineon Technologies AG. (8.11.2019), Adresse: https://www.infineon.com/cms/de/product/microcontroller/32-bit-tricore-microcontroller/ (besucht am 09. 11. 2019).

[INK12] Z. Irahhauten, H. Nikookar und M. Klepper, „2D UWB localization in indoor multipath environment using a joint ToA/DoA technique“, in *2012 IEEE Wireless Communications and Networking Conference (WCNC)*, April, 2012, S. 2253–2257. DOI: 10.1109/WCNC.2012.6214168.

[Int08] International Organization for Standardization, Hrsg., *Systems and software engineering — Requirements for designers and developers of user documentation*, ISO/IEC, 2008.

[Int11] International Organization for Standardization, Hrsg., *Systems and software engineering - Architecture description*, ISO/IEC/IEEE, 2011.

[Int15] International Organization for Standardization, Hrsg., *Software and systems engineering — Reference model for product line engineering and management*, ISO/IEC, 2015.

[Int17a] International Organization for Standardization, Hrsg., *Qualitätsmanagement - Leitfaden für Konfigurationsmanagement*, ISO, Berlin: Beuth Verlag GmbH, 2017. DOI: 10.31030/3044849.

[Int17b] International Organization for Standardization, Hrsg., *Systems and software engineering - Vocabulary*, ISO/IEC/IEEE, 2017.

[Int18] International Organization for Standardization, Hrsg., *Road vehicles - Functional safety*, ISO, Version 2, 1. Dez. 2018.

[Int19] International Organization for Standardization, Hrsg., *ISO/PAS 21448:2019*, ISO/PAS, Version 1, 1. Jan. 2019.

[Int20a] Intel Corporation. „IntelRealSense/librealsense". (27.08.2020), Adresse: https://github.com/IntelRealSense/librealsense (besucht am 27. 08. 2020).

[Int20b] Intel® RealSense™ Depth and Tracking Cameras. „Depth Camera D435 – Intel® RealSense™ Depth and Tracking Cameras". (26.08.2020), Adresse: https://www.intelrealsense.com/depth-camera-d435/ (besucht am 27. 08. 2020).

[Int20c] International Organization for Standardization, Hrsg., *ISO/SAE DIS 21434*, ISO/SAE, Version 1, 2020.

[ITU94] ITU-T, Hrsg., *Information technology - Open Systems Interconnection*, X.200, 1. Juli 1994. Adresse: http://handle.itu.int/11.1002/1000/2820 (besucht am 11. 08. 2020).

[Jam18] Jams.Liu. „What's the difference between DDS and SOME/IP?" (2018), Adresse: https://stackoverflow.com/questions/51182471/whats-the-difference-between-dds-and-some-ip (besucht am 03. 11. 2020).

[JAX19] JAXenter. „Was ist Kubernetes und wie verhält es sich zu Docker?" (2019), Adresse: https://jaxenter.de/kubernetes/kubernetes - docker - container - payara - 84679 (besucht am 25. 08. 2020).

[Jen20] Jenkins. „Jenkins". (25.08.2020), Adresse: https://www.jenkins.io/ (besucht am 25. 08. 2020).

[JK20] S. Jacobs und O. Kracht, „Continuous Deployment: Kubernetes bei der Deutschen Bahn", *iX Magazin*, Jg. 2020, Nr. 9, S. 92–97, 2. Sep. 2020.

[KAK+19] A. Kampmann, B. Alrifaee, M. Kohout, A. Wustenberg, T. Woopen, M. Nolte, L. Eckstein und S. Kowalewski, „A Dynamic Service-Oriented Software Architecture for Highly Automated Vehicles“, in *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, (Auckland, New Zealand), IEEE, 27.10.2019 - 30.10.2019, S. 2101–2108, ISBN: 978-1-5386-7024-8. DOI: 10.1109/ITSC.2019.8916841.

[KB91] Y. Koren und J. Borenstein, „Potential field methods and their inherent limitations for mobile robot navigation“, in *Proceedings / 1991 IEEE International Conference on Robotics and Automation, April 9 - 11, 1991, Sacramento, California*, (Sacramento, CA, USA), International Conference on Robotics and Automation, Institute of Electrical and Electronics Engineers und IEEE International Conference on Robotics and Automation, Los Alamitos, Calif.: IEEE Computer Soc. Press, 1991, S. 1398–1404, ISBN: 0-8186-2163-X. DOI: 10.1109/ROBOT.1991.131810.

[KBA19] KBA. „Produkte der Statistik - Neuzulassungen nach Herstellern und Handelsnamen (FZ 4)“. (2019), Adresse: https://www.kba.de/DE/Statistik/Produktkatalog/produkte/Fahrzeuge/fz4_n_uebersicht.html (besucht am 31. 12. 2020).

[KBA20a] ——, „Bestand an Kraftfahrzeugen und Kraftfahrzeuganhängern nach ausgewählten Merkmalen (Bundesländern und Fahrzeugklassen), vierteljährlich (FZ 27)“. (2020), Adresse: https://www.kba.de/DE/Statistik/Produktkatalog/produkte/Fahrzeuge/fz27_b_uebersicht.html?nn=1146130 (besucht am 10. 08. 2020).

[KBA20b] ——, „Fahrzeugzulassungen (FZ), Bestand an Kraftfahrzeugen und Kraftfahrzeuganhängern nach Fahrzeugalter, 1. Januar 2020, FZ 15“. (2020).

[KBKS19] A. Knoll, C. Buckl, K.-J. Kuhn und G. Spiegelberg, „The RACE Project: An Informatics-Driven Greenfield Approach to Future E/E Architectures for Cars“, in *Automotive Systems and Software Engineering: State of the Art and Future Trends*, Y. Dajsuren und M. van den Brand, Hrsg., Cham: Springer International Publishing, 2019, S. 171–195, ISBN: 978-3-030-12157-0. DOI: 10.1007/978-3-030-12157-0_8.

[KBS09] D. Krafzig, K. Banke und D. Slama, *Enterprise SOA, Service-oriented architecture best practices*, eng, 8. print, Ser. The Coad series. Upper Saddle River, NJ: Prentice-Hall, 2009, 382 S., ISBN: 0131465759.

[KHO14] A. E. Kouche, H. S. Hassanein und K. Obaia, „WSN platform Plug-and-Play (PnP) customization“, in *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP 2014), Singapore, 21 - 24 April 2014*, (Singapore), Institute of Electrical and Electronics Engineers, IEEE International Conference on Intelligent Sensors, Sensor Networks and Information Processing und IEEE ISSNIP, Piscataway, NJ: IEEE, 2014, S. 1–6, ISBN: 978-1-4799-2843-9. DOI: 10.1109/ISSNIP.2014.6827642.

[KHS15] M. Kok, J. D. Hol und T. B. Schön, „Indoor Positioning Using Ultrawideband and Inertial Measurements“, *IEEE Transactions on Vehicular Technology*, Jg. 64, Nr. 4, S. 1293–1303, 2015, April, ISSN: 0018-9545. DOI: 10.1109/TVT.2015.2396640.

[KHS18] S. Kugele, D. Hettler und S. Shafaei, „Elastic Service Provision for Intelligent Vehicle Functions“, in *2018 IEEE Intelligent Transportation Systems Conference, November 4-7, Maui, Hawaii*, (Maui, HI), IEEE International Conference on Intelligent Transportation Systems u. a., Piscataway, NJ: IEEE, 2018, S. 3183–3190, ISBN: 978-1-7281-0321-1. DOI: 10.1109/ITSC.2018.8569374.

[KJ] U. Kiencke und D. John, „On the Way to An International Standard for Automotive Applications-Osek/Vdx“, in *SAE (Hg.) 1998 – International Congress on Transportation Electronics*.

[KML+17] J. Ku, M. Mozifian, J. Lee, A. Harakeh und S. Waslander, *Joint 3D Proposal Generation and Object Detection from View Aggregation*, For any inquiries contact aharakeh(at)uwaterloo(dot)ca, 6.12.2017. Adresse: http://arxiv.org/pdf/1712.02294v4.

[KNR+19] D. Keilhoff, D. Niedballa, H.-C. Reuss, M. Buchholz, F. Gies, K. Dietmayer, M. Lauer, C. Stiller, S. Ackermann, H. Winner, A. Kampmann, B. Alrifaee, S. Kowalewski, F. Klein, M. Struth, T. Woopen und L. Eckstein, „UNICARagil – New architectures for disruptive vehicle concepts“, in *19. Internationales Stuttgarter Symposium*, Ser. Proceedings, M. Bargende, H.-C. Reuss, A. Wag-

ner und J. Wiedemann, Hrsg., Bd. 95, Wiesbaden: Springer Fachmedien Wiesbaden, 2019, S. 830–842, ISBN: 978-3-658-25938-9. DOI: 10.1007/978-3-658-25939-6_65.

[KOB+17] S. Kugele, P. Obergfell, M. Broy, O. Creighton, M. Traub und W. Hopfensitz, „On Service-Orientation for Automotive Software“, in *2017 IEEE International Conference on Software Architecture (ICSA)*, (Gothenburg, Sweden), IEEE, 3.04.2017 - 07.04.2017, S. 193–202, ISBN: 978-1-5090-5729-0. DOI: 10.1109/ICSA.2017.20.

[KSGW07] S. Krishnan, P. Sharma, Z. Guoping und O. H. Woon, „A UWB based Localization System for Indoor Robot Navigation“, in *2007 IEEE International Conference on Ultra-Wideband*, 2007, S. 77–82. DOI: 10.1109/ICUWB.2007.4380919.

[Kub20a] Kubernetes. „Considerations for large clusters“. (8.10.2020), Adresse: https://kubernetes.io/docs/setup/best-practices/cluster-large/ (besucht am 08. 01. 2021).

[Kub20b] ——, „Was ist Kubernetes?“ (30.05.2020), Adresse: https : / / kubernetes.io/de/docs/concepts/overview/what-is-kubernetes/ (besucht am 25. 08. 2020).

[LAE+16] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu und A. C. Berg, „SSD: Single Shot MultiBox Detector“, in *Computer Vision – ECCV 2016*, (Cham, 2016), B. Leibe, J. Matas, N. Sebe und M. Welling, Hrsg., Cham: Springer International Publishing, 2016, S. 21–37, ISBN: 978-3-319-46448-0.

[LG 20] LG München I, Hrsg., *Irreführende Werbeaussagen über Fahrassistenzsysteme eines Pkw*, 33 O 14041/19, Urteil, 14. Juli 2020. Adresse: https://www.gesetze-bayern.de/Content/Document/Y-300-Z-BECKRS-B-2020-N-22849 (besucht am 08. 12. 2020).

[Lin20] Linux Containers. „Linux Containers, Infrastructure for container projects.“ (14.08.2020), Adresse: https://linuxcontainers.org/ (besucht am 15. 08. 2020).

[LKS20] C.-W. Lin, B. Kim und S. Shiraishi, „Hardware Virtualization and Task Allocation for Plug-and-Play Automotive Systems“, *IEEE Design & Test*, S. 1, 2020, ISSN: 2168-2356. DOI: 10.1109/MDAT.2019.2932936.

[LMB+14] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár und C. L. Zitnick, „Microsoft COCO: Common Objects in Context“, in *Computer Vision – ECCV 2014*, Ser. Lecture Notes in Computer Science, D. Fleet, T. Pajdla, B. Schiele und T. Tuytelaars, Hrsg., Bd. 8693, Cham: Springer International Publishing, 2014, S. 740–755, ISBN: 978-3-319-10601-4. DOI: 10.1007/978-3-319-10602-1_48.

[mag20] magicmaps. „Präzise GPS-Messungen mit Hilfe von DGPS und RTK“. (27.08.2020), Adresse: https://www.magicmaps.de/gnss-wissen/praezise-gps-messungen-mit-hilfe-von-dgps-und-rtk/?L=0 (besucht am 27. 08. 2020).

[Mat10a] J. Matevska, „Rekonfiguration komponentenbasierter Softwaresysteme“, in *Rekonfiguration komponentenbasierter Softwaresysteme zur Laufzeit*, J. Matevska, Hrsg., Wiesbaden: Vieweg+Teubner, 2010, S. 73–81, ISBN: 978-3-8348-1001-4. DOI: 10.1007/978-3-8348-9780-0_5.

[Mat10b] J. Matheis, *Abstraktionsebenenübergreifende Darstellung von Elektrik/Elektronik-Architekturen in Kraftfahrzeugen zur Ableitung von Sicherheitszielen nach ISO 26262*, Zugl.: Karlsruhe, Institut für Technologie, Diss., 2009, ger, 1. Aufl., Ser. Berichte aus der Informationstechnik. Aachen: Shaker, 2010, 217 S, ISBN: 9783832289683.

[Max20] Max-Planck-Gesellschaft. „ITER-Bolometer“, Max-Planck-Institut für Plasmaphysik. (26.08.2020), Adresse: https://www.ipp.mpg.de/3668759/iterbolo (besucht am 26. 08. 2020).

[MBB18] M. Maul, G. Becker und U. Bernhard, „Serviceorientierte EE-Zonenarchitektur Schlüsselelement für neue Marktsegmente“, *ATZelektronik*, Jg. 13, Nr. 1, S. 36–41, 2018, PII: 105, ISSN: 1862-1791. DOI: 10.1007/s35658-017-0105-3.

[McK19a] McKinsey. „Umsatzprognose der weltweiten Automobilindustrie im Bereich Software sowie elektrische und elektronische Bauteile nach Segmenten in den Jahren 2020 bis 2030 (in Milliarden US-Dollar)“, Zitiert nach de.statista.com. (2019), Adresse: https://de.statista.com/statistik/daten/studie/1047098/umfrage/umsatzprognose-im-bereich-automotive-software-und-elektronik/ (besucht am 23. 12. 2019).

[McK19b] ——, „Umsatzprognose der weltweiten Automobilindustrie in den Jahren 2020 bis 2030 (in Milliarden US-Dollar)“, Zitiert nach de.statista.com. (2019), Adresse: https://de.statista.com/statistik/daten/studie/1047089/umfrage/prognostizierter-umsatz-der-weltweiten-automobilindustrie/ (besucht am 23. 12. 2019).

[Mer20] Mercedes-Benz. „Mercedes-Benz Buses: Home“. (18.08.2020), Adresse: https://www.mercedes-benz-bus.com/de_DE/home.html (besucht am 18. 08. 2020).

[MKA16] Y. Maruyama, S. Kato und T. Azumi, „Exploring the performance of ROS2“, in *2016 proceedings of the International Conference on Embedded Software (EMSOFT), October 2-7, 2016 Pittsburgh Marriott City Center, Pittsburgh, PA*, (Pittsburgh Pennsylvania), International Conference on Embedded Software u. a., Piscataway, NJ: IEEE, 2016, S. 1–10, ISBN: 9781450344852. DOI: 10.1145/2968478.2968502.

[MM15] A. Malkis und D. Marmsoler, „A Model of Service-Oriented Architectures“, in *2015 IX Brazilian Symposium on Components, Architectures and Reuse Software (SBCARS 2015), Belo Horizonte, Minas Gerais, Brazil, 21 - 22 September 2015 : [member event of the Sixth Brazilian Conference on Software: Theory and Practice (CBSoft 2015)*, (Belo Horizonte), Sociedade Brasileira de Computação u. a., Piscataway, NJ: IEEE, 2015, S. 110–119, ISBN: 978-1-4673-9630-1. DOI: 10.1109/SBCARS.2015.22.

[MP43] W. S. McCulloch und W. Pitts, „A logical calculus of the ideas immanent in nervous activity“, in *British Enterprise in Nigeria*, A. N. Cook, Hrsg., Bd. 5, PII: BF02478259, Philadelphia: University of Pennsylvania Press, 1943, S. 115–133, ISBN: 9781512815306. DOI: 10.1007/BF02478259.

[MPA08] L. Mottola, G. P. Picco und A. Amjad Sheikh, „FiGaRo: Fine-Grained Software Reconfiguration for Wireless Sensor Networks“, in *Wireless sensor networks, 5th European conference, EWSN 2008, Bologna, Italy, January 30-February 1, 2008 ; proceedings*, Ser. Lecture Notes in Computer Science 4913, R. Verdone, Hrsg., Bd. 4913, Berlin: Springer, 2008, S. 286–304, ISBN: 978-3-540-77689-5. DOI: 10.1007/978-3-540-77690-1_18.

[NAPG15] H. Nurminen, T. Ardeshiri, R. Piche und F. Gustafsson, „A NLOS-robust TOA positioning filter based on a skew-t measurement noise model“, in *2015 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, Oct, 2015, S. 1–7. DOI: 10.1109/IPIN.2015.7346786.

[NAS] NASA Infrared Telescope Facility, *Encoder Primer*, NASA Infrared Telescope Facility, Hrsg., Institute for Astronomy, University of Hawaii. Adresse: http://irtfweb.ifa.hawaii.edu/~tcs3/tcs3/0306_conceptual_design/Docs/05_Encoders/encoder_primer.pdf (besucht am 14. 10. 2020).

[NKO+13] K. Nagatani, S. Kiribayashi, Y. Okada, K. Otake, K. Yoshida, S. Tadokoro, T. Nishimura, T. Yoshida, E. Koyanagi, M. Fukushima und S. Kawatsuma, „Emergency response to the nuclear accident at the Fukushima Daiichi Nuclear Power Plants using mobile rescue robots“, *Journal of Field Robotics*, Jg. 30, Nr. 1, S. 44–63, 2013, ISSN: 15564959. DOI: 10.1002/rob.21439.

[OBOT18] F. Oszwald, J. Becker, P. Obergfell und M. Traub, „Dynamic Reconfiguration for Real-Time Automotive Embedded Systems in Fail-Operational Context“, in *2018 IEEE 32nd International Parallel and Distributed Processing Symposium workshops, IPDPSW 2018 : proceedings : 21-25 May 2018, Vancouver, British Columbia, Canada*, (Vancouver, BC), Heterogeneity in Computing Workshop u. a., Piscataway, NJ: IEEE, 2018, S. 206–209, ISBN: 978-1-5386-5555-9. DOI: 10.1109/IPDPSW.2018.00039.

[OMG15] OMG, *Data Distribution Service (DDS)*, OMG, Hrsg., Version 1.4, 2015.

[OMG18a] ——, *Secure DDS*, OMG, Hrsg., Version 1.1, 2018.

[OMG18b] ——, *The Real-time Publish-Subscribe Protocol (RTPS) DDS Interoperability Wire Protocol Specification*, OMG, Hrsg., Version 2.3, 2018.

[OOTB19] F. Oszwald, P. Obergfell, M. Traub und J. Becker, „Reliable Fail-Operational Automotive E/E-Architectures by Dynamic Redundancy and Reconfiguration“, in *32nd IEEE International System-on-Chip Conference (SOCC), September 03-06, 2019, Singapore : proceedings*, (Singapore), D. Zhao, Hrsg., Piscataway, NJ: IEEE, 2019,

S. 203–208, ISBN: 978-1-7281-3483-3. DOI: 10.1109/SOCC46988.2019.1570547977.

[Ope20a] Open Source Robotics Foundation, Inc, Hrsg. „Core Stack Developer Overview — ros_core foxy documentation“. (23.07.2020), Adresse: http://docs.ros2.org/foxy/developer_overview.html#ros-middleware-implementations (besucht am 08. 08. 2020).

[Ope20b] OpenStack. „Open Source Cloud Computing Infrastructure - OpenStack“. (21.08.2020), Adresse: https://www.openstack.org/ (besucht am 25. 08. 2020).

[pfS20] pfSense. „pfSense® - World's Most Trusted Open Source Firewall“. (24.09.2020), Adresse: https://www.pfsense.org/ (besucht am 29. 10. 2020).

[PG74] G. J. Popek und R. P. Goldberg, „Formal requirements for virtualizable third generation architectures“, *Communications of the ACM*, Jg. 17, Nr. 7, S. 412–421, 1974, ISSN: 00010782. DOI: 10.1145/361011.361073.

[PKH+17] P. Pelliccione, E. Knauss, R. Heldal, S. Magnus Ågren, P. Mallozzi, A. Alminger und D. Borgentun, „Automotive Architecture Framework: The experience of Volvo Cars“, *Journal of Systems Architecture*, Jg. 77, S. 83–100, 2017, PII: S1383762117300954, ISSN: 13837621. DOI: 10.1016/j.sysarc.2017.02.005.

[Pod20] Podman. „What is Podman? — Podman documentation“. (24.08.2020), Adresse: https://podman.readthedocs.io/en/latest/index.html (besucht am 25. 08. 2020).

[Pro20] Proxmox. „Proxmox VE Virtualisierungsplattform“. (28.10.2020), Adresse: https://www.proxmox.com/de/proxmox-ve (besucht am 28. 10. 2020).

[QEM20] QEMU. „QEMU, the FAST! processor emulator“. (12.08.2020), Adresse: https://www.qemu.org/ (besucht am 25. 08. 2020).

[QLW+17] C. R. Qi, W. Liu, C. Wu, H. Su und L. J. Guibas, *Frustum PointNets for 3D Object Detection from RGB-D Data*, 15 pages, 12 figures, 14 tables, 22.11.2017. Adresse: http://arxiv.org/pdf/1711.08488v2.

[RCK+20] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. St. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang und Y. Zhou, „MLPerf Inference Benchmark“, in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, (Valencia, Spain), IEEE, 30.05.2020 - 03.06.2020, S. 446–459, ISBN: 978-1-7281-4661-4. DOI: 10.1109/ISCA45697.2020.00045.

[Red20] Redhat. „Was ist CI/CD?“ (23.08.2020), Adresse: https://www.redhat.com/de/topics/devops/what-is-ci-cd (besucht am 25. 08. 2020).

[Rop20] W. Roper. „Will Roper auf LinkedIn: #InnovativeAF #DevStar #DevSecOps“. (16.08.2020), Adresse: https://www.linkedin.com/feed/update/urn:li:activity:6672104843798216704/ (besucht am 16. 08. 2020).

[ros18] ros.org. „Powering the world’s robots“. (12.06.2018), Adresse: http://www.ros.org/ (besucht am 12. 06. 2018).

[ros19] ——, „std_msgs/Header Documentation“. (7.03.2019), Adresse: http://docs.ros.org/melodic/api/std_msgs/html/msg/Header.html (besucht am 02. 05. 2019).

[ros20a] ——, „navigation“. (7.10.2020), Adresse: http://wiki.ros.org/navigation (besucht am 07. 10. 2020).

[ros20b] ——, „ros2/ros1_bridge“. (4.12.2020), Adresse: https://github.com/ros2/ros1_bridge (besucht am 04. 12. 2020).

[ros20c] rosindex. „About Quality of Service settings“. (7.08.2020), Adresse: https://index.ros.org/doc/ros2/Concepts/About-Quality-of-Service-Settings/ (besucht am 09. 08. 2020).

[Ros58] F. Rosenblatt, „The perceptron: A probabilistic model for information storage and organization in the brain“, *Psychological Review*, Jg. 65, Nr. 6, S. 386–408, 1958, ISSN: 0033-295X. DOI: 10.1037/h0042519.

[RP05] A. Rasche und A. Polze, „Dynamic Reconfiguration of Component-based Real-time Software“, in *Proceedings / 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, WORDS 2005, 2 - 4 February 2005, Sedona, Arizona*, (Sedona, AZ, USA), International Workshop on Object Oriented Real Time Dependable Systems u. a., Los Alamitos, Calif.: IEEE Computer Society, 2005, S. 347–354, ISBN: 0-7695-2347-1. DOI: 10.1109/WORDS.2005.31.

[SAE16] SAE, Hrsg., *SURFACE VEHICLE RECOMMENDED PRACTICE*, SAE, 1. Sep. 2016.

[Sax08] E. Sax, Hrsg., *Automatisiertes Testen eingebetteter Systeme in der Automobilindustrie*, ger, München: Hanser, 2008, 219 S., ISBN: 978-3-446-41635-2. DOI: 10.3139/9783446419018. Adresse: http://www.hanser-elibrary.com/action/showBook?doi=10.3139/9783446419018.

[Sax18] E. Sax, *Wagen, fahr schon mal den Harry vor*, FKFS Forschungsinstitut für Kraftfahrwesen und Fahrzeugmotoren Stuttgart, Mitarb., Stuttgart: AutoTest, 27. Sep. 2018.

[Sco20] Scotiabank. „Neuzulassungen von Pkw weltweit bis 2020 Veröffentlicht von Andreas Ahlswede, 30.07.2020 Die Statistik zeigt die Anzahl der weltweiten Neuzulassungen von Personenkraftwagen (Pkw) von 2013 bis 2019 sowie eine Prognose für das Jahr 2020. Die Statistik enthält die Regionen Nordamerika, Westeuropa, Osteuropa, Asien und Südamerika. Im Jahr 2019 gab es schätzungsweise rund 75 Millionen Neuzulassungen von Pkw. Im Jahr 2020 sollen die Neuzulassungen u.a. aufgrund der Corona-Krise stark rückläufig sein. Anzahl der weltweiten Neuzulassungen von Pkw in den Jahren 2013 bis 2020“. Scotiabank, Hrsg., Scotiabank. (2020), Adresse: https://de.statista.com/statistik/daten/studie/247129/umfrage/weltweite-neuzulassungen-von-pkw/ (besucht am 16. 08. 2020).

[SEFR13] T. Scharwächter, M. Enzweiler, U. Franke und S. Roth, „Efficient Multi-cue Scene Segmentation“, in *Pattern Recognition*, Ser. Lecture Notes in Computer Science, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzo-

poulos, D. Tygar, M. Y. Vardi, G. Weikum, J. Weickert, M. Hein und B. Schiele, Hrsg., Bd. 8142, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, S. 435–445, ISBN: 978-3-642-40601-0. DOI: 10.1007/978-3-642-40602-7_46.

[Ser20] Sergey Dorodnicov. „Depth from Stereo". (27.08.2020), Adresse: https://github.com/IntelRealSense/librealsense/blob/master/doc/depth-from-stereo.md (besucht am 27. 08. 2020).

[Set20] Setra. „Omnibusse | The Sign of Excellence". (18.08.2020), Adresse: https : / / www . setra . de / setrastartseite . html (besucht am 18. 08. 2020).

[SGSM11] J. Schmid, T. Gädeke, W. Stork und K. D. Müller-Glaser, „On the fusion of inertial data for signal strength localization", in *2011 8th Workshop on Positioning, Navigation and Communication*, April, 2011, S. 7–12. DOI: 10.1109/WPNC.2011.5961006.

[SJRS21] M. Sommer, C. Junk, T. Rösch und E. Sax, „Intelligent control of HVAC systems in battery electric city buses", in *IHIET - AI: Focusing on Artificial Intelligence*, (Strasbourg), 2021.

[SPA+14] B. Silva, Z. Pang, J. Akerberg, J. Neander und G. Hancke, „Experimental study of UWB-based high precision localization for industrial applications", in *2014 IEEE International Conference on Ultra-WideBand (ICUWB)*, Sept, 2014, S. 280–285. DOI: 10.1109/ICUWB.2014.6958993.

[SSWH19] T. Schmid, S. Schraufstetter, S. Wagner und D. Hellhake, „A Safety Argumentation for Fail-Operational Automotive Systems in Compliance with ISO 26262", in *2019 4th International Conference on System Reliability and Safety (ICSRS)*, (Rome, Italy), IEEE, 20.11.2019 - 22.11.2019, S. 484–493, ISBN: 978-1-7281-4781-9. DOI: 10.1109/ICSRS48664.2019.8987656.

[ST12] T. Streichert und M. Traub, *Elektrik/Elektronik-Architekturen im Kraftfahrzeug*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, 294 S., ISBN: 978-3-642-25477-2. DOI: 10.1007/978-3-642-25478-9.

[Sta17] M. Staron, *Automotive Software Architectures*. Cham: Springer International Publishing, 2017, 248 S., ISBN: 978-3-319-58609-0. DOI: 10.1007/978-3-319-58610-6.

[Tal21] Talend Real-Time Open Source Data Integration Software. „Microservices vs. SOA: Die Unterschiede im Überblick | Talend“. (25.02.2021), Adresse: https://www.talend.com/de/resources/microservices-vs-soa/ (besucht am 07. 04. 2021).

[Tes20] Tesla. „Autopilot“. Tesla, Hrsg. (14.12.2020), Adresse: https://www.tesla.com/de_DE/autopilot (besucht am 14. 12. 2020).

[The21] The Open Group. „SOA Reference Architecture“. (7.01.2021), Adresse: https://www.opengroup.org/soa/source-book/soa_refarch/index.htm (besucht am 07. 01. 2021).

[TL08] S. Thrun und J. J. Leonard, „Simultaneous Localization and Mapping“, in *Springer handbook of robotics, With … 84 tables*, B. Siciliano und O. Khatib, Hrsg., Berlin: Springer, 2008, S. 871–889, ISBN: 978-3-540-23957-4. DOI: 10.1007/978-3-540-30301-5_38.

[Tom17] Tomas Krejci. „GitHub: kitti2bag“. (2017), Adresse: https://github.com/tomas789/kitti2bag (besucht am 19. 12. 2018).

[UHGB04] M. Ullmann, M. Hübner, B. Grimm und J. Becker, „An FPGA run-time system for dynamical on-demand reconfiguration“, in *Proceedings / 18th International Parallel and Distributed Processing Symposium, Santa Fe, New Mexico, April 26 - 30, 2004 ; [abstracts and CD-ROM*, (Santa Fe, NM, USA), IPDPS u. a., Los Alamitos, Calif.: IEEE Computer Society, 2004, S. 135–142, ISBN: 0-7695-2132-0. DOI: 10.1109/IPDPS.2004.1303106.

[USC18] Uwe Franke, Stefan Gehrig und Clemens Rabe. „Scene Labeling - 6D-Vision“. (13.12.2018), Adresse: http://www.6d-vision.com/scene-labeling/ (besucht am 22. 12. 2018).

[Vec18] Vector Informatik GmbH, Hrsg., *From Signal to Service, Challenges for the Development of AUTOSAR Adaptive Applications*, 2018.

[Vec19] Vector Informatik GmbH. „PREEvision | Die E/E-Engineering-Lösung | Vector“. (7.12.2019), Adresse: https://www.vector.com/de/de/produkte/produkte-a-z/software/preevision/ (besucht am 07. 12. 2019).

[Vel20] Velodyne Lidar. „Puck Lidar Sensor, High-Value Surround Lidar | Velodyne Lidar“. (27.08.2020), Adresse: https://velodynelidar.com/products/puck/ (besucht am 27. 08. 2020).

[VWE+07] C. Vieider, S. Wissmar, P. Ericsson, U. Halldin, F. Niklaus, G. Stemme, J.-E. Källhammer, H. Pettersson, D. Eriksson, H. Jakobsen, T. Kvisterøy, J. Franks, J. VanNylen, H. Vercammen und A. VanHulsel, „Low-cost far infrared bolometer camera for automotive use“, in *Infrared Technology and Applications XXXIII*, (Orlando, Florida, USA), B. F. Andresen, G. F. Fulop und P. R. Norton, Hrsg., Ser. SPIE Proceedings, SPIE, 2007, S. 65421L. DOI: 10.1117/12.721272.

[WEK+18] T. Woopen, L. Eckstein, S. Kowalewski, D. Moormann, M. Maurer, R. Ernst, H. Winner, S. Katzenbeisser, M. Becker, C. Stiller, K. Furmans, K. Bengler, M. Lienkamp, H.-C. Reuss, K. Dietmayer, H. Lategahn, N. Siepenkötter, M. Elbs, E. v. Hinüber, M. Dupuis und C. Hecker, „UNICARagil - Disruptive modulare Architektur für agile, automatisierte Fahrzeugkonzepte“, en, 2018. DOI: 10.18154/RWTH-2018-229909.

[Wik21] Wikipedia, Hrsg. „OSI-Modell“. de. Creative Commons Attribution-ShareAlike License Page Version ID: 213601999. (2021), Adresse: https://de.wikipedia.org/w/index.php?title=OSI-Modell&oldid=213601999 (besucht am 30. 07. 2021).

[Wol] J. Wolf, *Ethernet-Security im Automotive-Umfeld am Beispiel SOME/IP*, Wolfsburg.

Betreute studentische Arbeiten

[Ara18] L. Araújo, „Maturity level for system under test applied to Autonomous Driving“, Institut für Technik der Informationsverarbeitung, Masterarbeit, Karlsruher Institut für Technologie, Karlsruhe, 2018.

[Böh18] M. Böhme, „Konzept zur Elektrifizierung der Panoramabahn“, Institut für Technik der Informationsverarbeitung, Masterarbeit, Karlsruher Institut für Technologie, Karlsruhe, 2018.

[Bro20] M. Brodatzki, „Entwicklung und Evaluation einer serviceorientierten Architektur auf Basis von ROS2 für NORA“, Institut für Technik der Informationsverarbeitung, Bachelorarbeit, Karlsruher Institut für Technologie, Karlsruhe, 2020.

[Etk17] T. Etkin, „Development of an Electronic Control Unit for the Motor Control of a Ride-On Lawnmower“, Institut für Technik der Informationsverarbeitung, Bachelorarbeit, Karlsruher Institut für Technologie, Karlsruhe, 2017.

[Far17] M. Farhat, „Entwicklung einer Lokalisierungs- und Kartografierungslösung für einen autonomen Rasenmäher“, Institut für Technik der Informationsverarbeitung, Masterarbeit, Karlsruher Institut für Technologie, Karlsruhe, 2017.

[Hau17] M. Haußecker, „Integration einer Verkehrssimulation in ein Hardware-in-the-Loop Testsystem zur Systemabsicherung von Fahrerassistenzsystemen“, Institut für Technik der Informationsverarbeitung, Masterarbeit, Karlsruher Institut für Technologie, Karlsruhe, 2017.

[Hoo18] B. Hooi Choo, „Enhanced Interference Tool Integration for John Deere GNSS Receiver“, Institut für Technik der Informationsverarbeitung, Masterarbeit, Karlsruher Institut für Technologie, Karlsruhe, 2018.

[Käf17] A. Käfer, „Simulation dynamischer Positionsdaten eines Operationstisches mittels Hardware-in-the-Loop“, Institut für Technik der Informationsverarbeitung, Masterarbeit, Karlsruher Institut für Technologie, Karlsruhe, 2017.

[Koc19] E. Koch, „Integration von ROS-Kommunikationsschnittstellen in ein modellbasiertes Werkzeug zur Beschreibung AUTOSAR-konformer Elektrik/Elektronik-Architekturen im Fahrzeugbau“, Institut für Technik der Informationsverarbeitung, Masterarbeit, Karlsruher Institut für Technologie, Karlsruhe, 2019.

[Kra21] F. Krauter, „Safetykonzept für eine neue neuartige Fahrzeugarchitektur“, Institut für Technik der Informationsverarbeitung, Masterarbeit, Karlsruher Institut für Technologie, Karlsruhe, 2021.

[Lef18] F. Leferenz, „Entwicklung einer Objekterkennungs- und -klassifizierungslösung für einen autonomen Rasenmäher“, Institut für Technik der Nachrichtenverarbeitung, Masterarbeit, Karlsruher Institut für Technologie, Karlsruhe, 2018.

[Rau16] J. Raudonat, „Entwicklung eines Sensorkonzepts für einen autonomen Rasenmäher“, Institut für Technik der Informationsverarbeitung, Masterarbeit, Karlsruher Institut für Technologie, Karlsruhe, 2016.

[Rös19] F. Rösel, „Konzeption und Umsetzung einer funktionalen Sicherheitsebene für ein automatisiertes Kleinfahrzeug“, Institut für Technik der Informationsverarbeitung, Masterarbeit, Karlsruher Institut für Technologie, Karlsruhe, 2019.

[Sch21a] L. Schneider, „Dynamische Modellierung einer rekonfigurierbaren serviceorientierten Architektur in PREEvision“, Institut für Technik der Informationsverarbeitung, Bachelorarbeit, Karlsruher Institut für Technologie, Karlsruhe, 2021.

[Sch21b] D. Schuckmann, „Dynamisches Verteilen von Services in Serviceorientierten Architekturen“, Institut für Technik der Informationsverarbeitung, Masterarbeit, Karlsruher Institut für Technologie, Karlsruhe, 2021.

[Uhl17] N. Uhlemann, „Konzeptionierung und Implementierung eines Multi-Objekt-Reglerenwurfs für die Längsregelung von Nutzfahrzeugen“, Institut für Technik der Informationsverarbeitung, Masterarbeit, Karlsruher Institut für Technologie, Karlsruhe, 2017.

[Wol17] C. Wolff, „Modellbildung und Integration autonomer Fahrfunktionen am Beispiel eines Aufsitzrasenmähers“, Institut für Technik der Informationsverarbeitung, Masterarbeit, Karlsruher Institut für Technologie, Karlsruhe, 2017.

[Zim16] P. Zimmer, „Entwicklung und Evaluation einer Lokalisierungslösung für autonome Fahrzeuge“, Institut für Technik der Informationsverarbeitung, Masterarbeit, Karlsruher Institut für Technologie, Karlsruhe, 2016.

Eigene Veröffentlichungen

[BSS+16] F. K. Bapp, O. Sander, T. Sandmann, H. Stoll und J. Becker, „Programmable Logic as Device Virtualization Layer in Heterogeneous Multicore Architectures“, in *Applied Reconfigurable Computing, 12th International Symposium, ARC 2016 Mangaratiba, RJ, Brazil, March 22-24, 2016 Proceedings*, Ser. Lecture Notes in Computer Science 9625, V. Bonato, C. Bouganis und M. Gorgon, Hrsg., 1st ed. 2016, Bd. 9625, Cham und s.l.: Springer International Publishing, 2016, S. 273–286, ISBN: 978-3-319-30480-9. DOI: 10.1007/978-3-319-30481-6_22.

[GHSS] H. Guissouma, C. P. Hohl, H. Stoll und E. Sax, „Variability-Aware Process Extension for Updating Cyber Physical Systems Over the Air“, *2020 9th Mediterranean Conference on Embedded Computing (MECO)*, S. 1–8, DOI: 10.1109/MECO49872.2020.9134339.

[SGS+21] H. Stoll, D. Grimm, M. Schindewolf, M. Brodatzki und E. Sax, „Dynamic Reconfiguration of Automotive Architectures Using a Novel Plug-and-Play Approach“, im Druck, in *IV21 32nd IEEE Intelligent Vehicles Symposium, 2nd Workshop on Intelligent Transportation Systems, Intelligent Vehicles and Advanced Driver Assistant Systems for Unstructured Environments (ITSIVUE2021)*, (Nagoya), IEEE, 2021.

[SKS20] H. Stoll, E. Koch und E. Sax, „Integration of ROS communication interfaces in a model-based tool for the description of AUTOSAR-compliant electrical/electronic architectures (E/E-A) in vehicle development“, in *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, (Rhodes), IEEE, 9/20/2020 - 9/23/2020, S. 1–6, ISBN: 978-1-7281-4149-7. DOI: 10.1109/ITSC45102.2020.9294319.

[SS21] H. Stoll und E. Sax, „Object Detection and Semantic Segmentation for a Small Low-cost Vehicle“, in *Commercial Vehicle Technology 2020/2021*, (Wiesbaden, 2021), K. Berns, K. Dressler, R. Kalmar, N. Stephan, R. Teutsch und M. Thul, Hrsg., Wiesbaden: Springer Fachmedien Wiesbaden, 2021, S. 485–498, ISBN: 978-3-658-29717-6.

[SZHS17] H. Stoll, P. Zimmer, F. Hartmann und E. Sax, „GPS-independent localization for off-road vehicles using ultra-wideband (UWB)“, in *IEEE ITSC 2017, 20th International Conference on Intelligent Transportation Systems : Mielparque Yokohama in Yokohama, Kanagawa, Japan, October 16-19, 2017*, (Yokohama), IEEE Intelligent Transportation Systems Conference u. a., Piscataway, NJ: IEEE, 2017, S. 1–6, ISBN: 978-1-5386-1526-3. DOI: 10.1109/ITSC.2017.8317763.