

# Programmierung einer LED-Matrix für das Forschungsprojekt UNICARagil

Programming of a LED matrix for the research project UNICARagil

Semesterarbeit  
an der Fakultät für Maschinenwesen der Technischen Universität München.

**Betreut von** Univ.-Prof. Dr. phil. Klaus Bengler  
M. Sc. Johannes Schwiebacher  
M. Sc. Jonas Bender  
Lehrstuhl für Ergonomie

**Eingereicht von** Gia-Phong Tran  
Mühlgasse 9  
85748 Garching

**Eingereicht am** Garching, den 12.11.2021



# Erklärung

Ich versichere hiermit, dass ich die von mir eingereichte Abschlussarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Garching, 12.11.2021

Ort, Datum, Unterschrift

A handwritten signature in black ink, consisting of stylized cursive letters, positioned above a horizontal line.

# Vereinbarung zum Urheberrecht

Hiermit gestatte ich dem Lehrstuhl für Ergonomie diese Studienarbeit bzw. Teile davon nach eigenem Ermessen an Dritte weiterzugeben, zu veröffentlichen oder anderweitig zu nutzen. Mein persönliches Urheberrecht ist über diese Regelung hinaus nicht beeinträchtigt.

Eventuelle Geheimhaltungsvereinbarungen über den Inhalt der Arbeit zwischen mir bzw. dem Lehrstuhl für Ergonomie und Dritten bleiben von dieser Vereinbarung unberührt.

Garching, 12.11.2021

Ort, Datum, Unterschrift

A handwritten signature in black ink, consisting of stylized cursive letters, positioned above a horizontal line.

# Lizenzvereinbarung

Der Student Gia-Phong Tran, Matrikelnummer 03736578, stellt im Rahmen der Studienarbeit „Programmierung einer LED-Matrix für das Forschungsprojekt UNICARagil“ entstandenen Quellcode und dazugehörige Dokumentation („die Software“) unter die angekreuzte Lizenz:

☐ **Public Domain**

Eine Textdatei mit dem Lizenztext von [unlicense.org](http://unlicense.org) liegt dem Code bei.

☐ **MIT-Lizenz**

Eine Textdatei mit dem Lizenztext von <http://choosealicense.com/licenses/mit/> liegt der Software bei. (Empfohlen für die meisten Studienarbeiten, wenn keine kommerzielle Nutzung geplant ist.)

☒ **GPLv2**

Eine Textdatei mit dem Lizenztext von <http://choosealicense.com/licenses/gpl-2.0/> liegt der Software bei. (Zwingend für Studienarbeiten, die Bibliotheken nutzen, die unter GPL stehen.)

Der Lehrstuhl für Ergonomie und die Professur für Sportgeräte und -materialien der TU München erhalten die Erlaubnis, die Software uneingeschränkt zu benutzen, inklusive und ohne Ausnahme dem Recht, sie zu verwenden, kopieren, ändern, fusionieren, verlegen, verbreiten, unter-lizenzieren und/oder zu verkaufen, und Personen, die diese Software erhalten, diese Rechte zu geben.

Der Urheberrechtsvermerk „© Copyright Gia-Phong Tran“ und folgender Haftungsausschluss sind in allen Kopien oder Teilkopien der Software beizulegen.

Die Software wird ohne jede ausdrückliche oder implizierte Garantie bereitgestellt, einschließlich der Garantie zur Benutzung für den vorgesehenen oder einen bestimmten Zweck sowie jeglicher Rechtsverletzung, jedoch nicht darauf beschränkt. In keinem Fall sind die Autoren oder Copyrightinhaber für jeglichen Schaden oder sonstige Ansprüche haftbar zu machen, ob infolge der Erfüllung eines Vertrages, eines Deliktes oder anders im Zusammenhang mit der Software oder sonstiger Verwendung der Software entstanden.

Unterschrift des Studenten:  \_\_\_\_\_

# Erläuterung

## Public Domain

Unverbindliche Info: Der Urheber kann nicht haftbar gemacht werden. Jeder darf mit der Software machen was er will. Public Domain führt zu maximaler Nutzbarkeit durch alle denkbaren Parteien. Dies führt zu keinerlei Aufwand für den Lehrstuhl und empfiehlt sich für Urheber, denen jegliche Weiterverwendung ihres Werkes egal ist oder die maximale Verbreitung erreichen wollen. Allerdings verliert der Urheber auch jegliche Kontrolle und Wertschätzung für sein Werk. Ein Dritter könnte die Software für beliebige Zwecke einsetzen und auch unter eigenem Namen verkaufen.

## MIT-Lizenz

Unverbindliche Info: Der Urheber kann nicht haftbar gemacht werden. Jeder darf mit der Software machen was er will, solange er den Lizenztext, der den Name des Urhebers enthält, immer mit der Software verteilt. Die MIT-Lizenz macht die einfache Nutzung und maximale Verbreitung möglich und ist die bevorzugte Variante des Lehrstuhls. Der Name des Urhebers bleibt weiterhin mit seinem Werk verbunden und er bekommt so die verdiente Wertschätzung.

## GPL

Unverbindliche Info: Freies Verteilen, verändern und verkaufen möglich, keine Haftung. Jede unveränderte oder veränderte Fassung, sowie jedes Programm, das beliebige Teile dieser Software nutzt, muss ebenfalls unter der GPL stehen **und der Quellcode öffentlich verfügbar gemacht werden**. Die GPL erfordert vom Lehrstuhl bei jeder Änderung etwas Aufwand, weil sie veröffentlicht werden muss und ist deshalb nicht die erste Wahl, außer die gemeinschaftliche Weiterentwicklung der Software ist erklärtes Ziel.

## Individuelle Rechtevergabe

Die individuelle Rechtevergabe ermöglicht dem Lehrstuhl eine freie Nutzung ohne die Software allgemein zugänglich zu machen. Dies ist vor allem nützlich, wenn die Software im Rahmen eines Projektes des LfE/SpGM mit weiteren Partnern genutzt oder weiterentwickelt werden soll oder anderweitig kommerzielles Potential hat, das der Urheber aber nicht weiter verfolgen will.

Die Copyright-Notiz sollte mit dem Jahr und dem Namen des Urhebers ausgefüllt oder der Satz gestrichen werden, falls nicht gewünscht.

Falls der Urheber keine freie Verbreitung der Software wünscht (z.B. weil er sie kommerziell vertreiben möchte), der Lehrstuhl aber dennoch Rechte an der Software bekommen soll, so können nicht gewünschten Rechte aus dem obigen Text gestrichen werden.

Die vier Optionen schließen sich gegenseitig aus.

Weiter Informationen zu Lizenzen unter: <http://choosealicense.com/>

Anmerkung: Falls ein Student bei einer externen Studienarbeit einen Arbeitsvertrag mit einer Firma hat, hat er meistens (je nach Arbeitsvertrag) kein Recht die Software an Dritte zu lizenzieren.

**Der Student ist in keiner Weise verpflichtet die hier vorgeschlagenen oder andere Lizenzvereinbarungen einzugehen. Es steht ihm frei, auch andere Lizenzvereinbarungen als die hier beschriebenen zu wählen.**

Er verliert bei keiner der hier beschriebenen Lizenzen in irgendeiner Weise sein persönliches Urheberrecht oder Nutzungsrecht. Sie dienen lediglich dazu, Dritten (dem LfE/SpGM oder der Allgemeinheit) nicht-exklusive Nutzungsrechte einzuräumen, wenn dies im Sinne des Studenten ist.

Die Wahl einer beliebigen anderen Lizenz für neuere Versionen der Software ist dem Studenten als Urheber jederzeit möglich.

## Kurzfassung / Abstract

Das Forschungsprojekt UNICARagil liefert im Bereich automatisiertes Fahren einen innovativen Ansatz. Das Ziel ist die Entwicklung innovativer modularer Architekturen und Methoden für neue agile, automatisierte Fahrzeugkonzepte. Hierbei ist die Interaktion anderer Verkehrsteilnehmer mit den autonomen Fahrzeugen ein wichtiger Aspekt für die Verkehrseffizienz und die Sicherheit, die im Rahmen dieses Projekts über die externe Kommunikation realisiert wird. Die externe Kommunikation besteht hierbei aus je einer LED-Matrix in der Fahrzeugfront und im Heck. Über die LED-Matrizen sollen im Fahrbetrieb Bilder und Animationen angezeigt werden, die andere Verkehrsteilnehmer über den aktuellen Fahrzeugzustand informieren. Ziel der Arbeit ist die Implementierung einer Software zur Ansteuerung dieser LED-Matrizen. Dazu existiert bereits ein UML-Zustandsdiagramm, welches an einem Raspberry Pi umgesetzt werden soll. Die Programmierung erfolgt in der Programmiersprache C++ innerhalb der sog. ASOA-Architektur. Dabei handelt es sich um eine Dienste-basierte Softwarearchitektur, die im UNICARagil Projekt zum Einsatz kommt. Anschließend soll ein Testkonzept entwickelt und angewandt werden, um die Funktionalität der Software zu erproben. Ergebnis der Arbeit ist eine Software, die eine schnelle LED-Anzeige ermöglicht und die alle Zustände mitsamt Zustandsübergängen aus dem vorgegebenen UML-Diagramm umsetzt.

The UNICARagil research project provides an innovative approach in the field of automated driving. The aim is to develop innovative modular architectures and methods for new agile, automated vehicle concepts. Here, interaction of other traffic participants with the autonomous vehicles is an important aspect for traffic efficiency and safety, which is realised via external communication. The external communication consists of an LED matrix in the front and in the rear of the vehicle. Images and animations are to be displayed via the LED matrices during driving to inform other traffic participants about the current vehicle status. The aim of the work is to implement a software to control these LED matrices. A UML state diagram already exists for this purpose, which is to be implemented on a Raspberry Pi. The programming is done in the programming language C++ within the so-called ASOA architecture, a service-based software architecture that is used in the UNICARagil project. Subsequently, a test concept will be developed and applied to test the functionality of the software. The result of the work is a software that enables a fast LED display and that implements all states including state transitions from the given UML diagram.



# Inhaltsverzeichnis

1	Einleitung .....	1
1.1	Das Forschungsprojekt UNICARagil .....	1
1.2	Ziel der Semesterarbeit.....	2
1.3	Gliederung der Arbeit .....	2
2	Stand der Technik.....	2
2.1	Externe Anzeigeconzepte (eHMI) .....	4
2.2	UNICARagil eHMI: Aufbau und Erkennbarkeit .....	5
2.3	ASOA Software-Architektur .....	8
2.3.1	ASOA Dienste .....	9
2.3.2	ASOA Orchestrator .....	10
2.3.3	Zusammenwirken der ASOA Elemente.....	11
2.4	UML-Zustandsdiagramme .....	11
3	Problemstellung .....	13
4	Implementierung .....	16
4.1	Software-Aufbau und Einrichtung .....	16
4.1.1	Einrichtung Raspberry Pi .....	16
4.1.2	Einrichtung virtuelle Maschine und Orchestrator .....	17
4.1.3	Netzwerkkonfiguration .....	18
4.1.4	Build-Tools .....	20
4.1.5	Hzeller-Library .....	21
4.1.6	Gesamtkonzept LED-Ansteuerung.....	22
4.2	Programmierung.....	22
4.2.1	Vorüberlegungen .....	23
4.2.2	Einbinden der LED-Softwarebibliothek in die ASOA-Dienststruktur .....	25
4.2.3	IDL-Files mit Architekturtool.....	26
4.2.4	rgbController: LED Image Viewer.....	27
4.2.5	Front-Service und Back-Service .....	32
5	Softwaretest.....	37
5.1	Dummy-Service .....	37
5.2	Grafische Benutzeroberfläche .....	38
5.3	Server-Client-Kommunikation .....	41
5.4	Testen des Front- und Back-Service .....	44
5.5	Testmethodik und Durchführung .....	46
6	Testergebnisse und Diskussion .....	47
6.1	Ergebnisse des Softwaretests .....	47
6.2	Ladezeiten der Symbole und Animationen .....	49
6.3	Weiterführende Ansätze für die Programmierung .....	50
7	Fazit und Ausblick.....	53

Abbildungsverzeichnis .....	55
Abkürzungsverzeichnis .....	56
Tabellenverzeichnis .....	57
Literaturverzeichnis .....	58
A   IDL-Files und Nutzdaten (Front-Service) .....	62
B   Netzwerkkonfiguration YAML-Datei .....	63
C   CMakeLists.txt .....	64

# 1 Einleitung

Autonomes Fahren ist ein vieldiskutiertes technisches, wirtschaftliches und rechtliches Thema in der Gesellschaft und erlangt heutzutage hohe Aufmerksamkeit in den Medien. Die Forschung an vollautomatisierten Fahrzeugen wurde in der Vergangenheit und wird auch heute aus vielen Gründen betrieben. Die Forschung sieht u.a. Potenziale in der Senkung der Unfalltoten und in der Optimierung des Verkehrsflusses (Maurer, 2015, S. 4).

Infrastrukturen in Städten werden immer stärker belastet und gleichzeitig steigen Emissionswerte. Ein Treiber hierfür ist bspw. der sich immer weiter abzeichnende Trend des E-Commerce, der immer mehr Zustellungen erfordert (FZI Forschungszentrum Informatik, 2018). Dieser Trend strapaziert zusätzlich das Verkehrsaufkommen, besonders in städtischen Gebieten, sodass Fahrtzeiten auf Grund eines höheren Verkehrsaufkommens länger dauern und damit zusätzliche Kosten erzeugen (Heinemann, 2018, S. 121). Damit bedarf es intelligenter Fahrzeugkonzepte, die umweltfreundlich und effizient sind und die Sicherheit für alle Verkehrsteilnehmer erhöhen.

Das autonome Fahren könnte nicht nur die Verkehrsinfrastruktur in Städten entlasten, sondern auch Senioren neue Möglichkeiten zur Mobilität bieten. Dem Verband der Automobilindustrie (2021) zufolge wird aktuell ein fahrerloser und elektrisch betriebener Bus zu Testzwecken in der niederbayerischen Marktgemeinde Bad Birnbach eingesetzt. Der Einsatz dieses Busses soll einerseits einen umweltfreundlicheren Personentransport bewirken und andererseits Mobilität für diejenigen bieten, die selbst kein Fahrzeug mehr bedienen können oder wollen. Das autonome Fahren hat somit ein großes Potential für ältere Menschen und damit auch einen sozialen Aspekt (Verband der Automobilindustrie e.V., 2021).

## 1.1 Das Forschungsprojekt UNICARagil

Das Forschungsprojekt UNICARagil liefert im Bereich automatisiertes Fahren einen innovativen Ansatz. Das vom Bundesministerium für Bildung und Forschung geförderte Projekt vereint Kompetenzen aus führenden deutschen Universitäten im Bereich des autonomen Fahrens und wurde 2018 gestartet. Das Ziel ist die Entwicklung innovativer modularer Architekturen und Methoden für neue agile, automatisierte Fahrzeugkonzepte (Buchholz, 2020, S. 1532).

Hierbei ist die Interaktion mit anderen Verkehrsteilnehmern mit den autonomen Fahrzeugen ein wichtiger Aspekt für die Verkehrseffizienz und die Sicherheit, die im Rahmen dieses Projekts über die externe Kommunikation realisiert wird. Die externe Kommunikation kann sowohl die Verkehrseffizienz als auch die Verkehrssicherheit erhöhen und kann zudem die Kooperation im Verkehr ermöglichen (UNICARagil news, 2021, S. 69).

## 1.2 Ziel der Semesterarbeit

Die externe Kommunikation wird durch die Vermittlung von Informationen mittels LED-Matrizen realisiert. Am UNICARagil Fahrzeug befindet sich in der Front und im Heck jeweils eine LED-Matrix, die im Fahrbetrieb bestimmte Symboliken anzeigen soll. Je nach Fahrzeugzustand soll eine dazu korrespondierende Animation oder ein Symbolbild auf beiden LED-Matrizen angezeigt werden. Ein Symbolkatalog ist gegeben und die Zustände der LED-Matrizen sind in einem UML-Zustandsdiagramm definiert. Ziel der Arbeit ist die Ansteuerung der LED-Matrizen mit Hilfe eines Raspberry Pi. Dazu soll eine Software für den Raspberry Pi entwickelt werden, die das Zustandsdiagramm umsetzt und damit die Darstellung der Symboliken auf den LED-Matrizen ermöglicht.

## 1.3 Gliederung der Arbeit

Nach dieser kurzen Einführung in die Semesterarbeit werden in Kapitel 2 notwendige Grundlagen vorgestellt. Dabei werden kurz drei Themenschwerpunkte behandelt: externe Anzeigekonzepte an autonomen Fahrzeugen, der Hardware-Aufbau des in UNICARagil verwendeten Anzeigekonzepts und die für das Projekt spezifische Softwarearchitektur ASOA. Zusätzlich sollen einige Grundlagen zu UML-Zustandsdiagrammen vorgestellt werden.

Basierend auf diesen Grundlagen wird die Programmieraufgabe in Kapitel 3, der Problemstellung, beschrieben.

In Kapitel 4 wird schließlich die Implementierung dargestellt. Dabei wird konkret auf die einzelnen Bestandteile der Software eingegangen und mit Hilfe von UML-Klassendiagrammen verdeutlicht. Das Kapitel behandelt zunächst den Software-Aufbau, d.h. die benötigten Betriebssysteme, Programmierwerkzeuge und die für die LED-Matrix genutzte C++ Bibliothek. Danach wird ein Lösungsvorschlag für die Ansteuerung der LED-Matrix in der Front und im Heck vorgestellt.

Kapitel 5 befasst sich mit dem Softwaretest. Das verwendete Testkonzept, bestehend aus dem Dummy-Service, der grafischen Oberfläche und der Server-Client-Anbindung wird erklärt. Die konkrete Testdurchführung wird erläutert, die Diskussion der Testergebnisse und die Fehlerbehebung wird im darauffolgenden Kapitel erklärt.

Kapitel 6 diskutiert die Ergebnisse aus der Implementierung und dem Test. Zudem werden umgesetzte Maßnahmen zur Optimierung der Software vorgestellt und weiterführende Ansätze und Ideen für die Programmierung vorgeschlagen.

Das Fazit in Kapitel 7 fasst alle Kernpunkte dieser Semesterarbeit zusammen. Abschließend folgt im selben Kapitel ein Ausblick auf weitere Forschungsaktivitäten.

## 2 Stand der Technik

In diesem Kapitel werden die notwendigen Grundlagen für die Implementierung des eHMIs vorgestellt. Zunächst wird die Thematik externer Anzeigekonzepte an Fahrzeugen erklärt. Dann sollen vorangegangene Forschungsarbeiten rund um das für das in UNICARagil verwendete eHMI-Konzept vorgestellt werden, diese beinhalten den Aufbau eines eHMI-Prototyps und die Erkennbarkeit der Anzeige. Aus diesen Betrachtungen soll die grundlegende Funktionsweise der LED-Matrix erschlossen werden. Anschließend soll die für die Programmierung zu Grunde liegende Software-Architektur „ASOA“ eingeführt werden. Zuletzt werden die Grundlagen von UML-Zustandsdiagrammen vermittelt, da diese den Ausgangspunkt für die Programmierung in Kapitel 4 darstellt.

### 2.1 Externe Anzeigekonzepte (eHMI)

Die Interaktion von Verkehrsteilnehmern mit autonomen Fahrzeugen im Verkehrsbetrieb ist ein wichtiger Aspekt hinsichtlich Sicherheit und Effizienz. Dies wird im Rahmen dieses Forschungsprojekts über die externe Kommunikation realisiert. Die externe Kommunikation kann sowohl die Verkehrseffizienz als auch die Verkehrssicherheit erhöhen und kann zudem die Kooperation im Verkehr ermöglichen. Eine Form der externen Kommunikation ist das sog. external Human-Machine-Interface (kurz: eHMI). Im Allgemeinen ist ein eHMI ein Interface, die an der Außenfläche eines Fahrzeugs installiert ist oder aus dieser heraus Informationen projiziert (Bengler, Rettenmaier, Fritz & Feierle, 2020, S. 8). Informationen können hierbei sowohl optisch (z.B. über LEDs) als auch auditiv (z.B. über einen Lautsprecher) vermittelt werden, wobei Ersteres das Hauptthema dieser Arbeit darstellt.

Schwiebacher zufolge wird das eHMI genutzt, um den Status, das Verhalten und auch die Intention des Fahrzeuges zu kommunizieren. So können bspw. ungewöhnliche Fahrmanöver oder die Halteintention an einem Fußgängerüberweg kommuniziert werden. Die Sicherheit eines Fußgängers und der Komfort beim Überqueren einer Straße wird im Allgemeinen durch die Kommunikation über ein eHMI erhöht (Bengler et al., 2020, S. 8).

Nach Bengler et al. gibt es trotz der eben genannten Vorteile und Potentiale dennoch Herausforderungen und offene Fragen, so gibt es aktuell noch keine Standards oder

Minimalanforderungen für eHMIs. Welches Medium ein eHMI nutzen soll, welche Farben verwendet werden sollen oder auch wo eHMIs am Fahrzeug angebracht werden sollen noch bleibt ungeklärt (Bengler et al., 2020, S. 8).

Zum Zwecke der Vollständigkeit wird an dieser Stelle kurz auf einen zweiten Ansatz zur Kommunikation hingewiesen, dem dynamicHMI (kurz: dHMI). Nach Bengler et al. handelt es sich hierbei um eine implizite Art der Kommunikation, bei der über die Fahrzeugdynamik Informationen beabsichtigt oder auch unbeabsichtigt vermittelt werden (Bengler et al., 2020, S. 8–9). Im Rahmen dieser Semesterarbeit soll nicht weiter auf das dHMI eingegangen werden, der Fokus soll stattdessen auf dem eHMI und dessen Programmierung liegen.

## 2.2 UNICARagil eHMI: Aufbau und Erkennbarkeit

Für das visuelle eHMI beim UNICARagil-Fahrzeug existiert bereits ein finales Konzept. Für das Fahrzeug soll jeweils eine LED-Matrix in der Front und im Heck verwendet werden. Für die Erprobung des eHMIs des Fahrzeugs wurde im Rahmen einer Masterarbeit von Robin Storm am Lehrstuhl für Ergonomie (LfE) ein Prototyp für das eHMI konzipiert und umgesetzt. Dieser Prototyp dient auch für Erprobungen in zukünftigen Forschungsarbeiten (Storm, 2019, S. 5).

Ausgehend von den Ergebnissen von Storm wurde der Prototyp im Rahmen der Masterarbeit von Jonas Schulze (LfE) weiterentwickelt (Schulze, 2020, S. 1–2). Das Resultat ist eine Displayanzeige in Form einer LED-Matrix. Da sich diese Semesterarbeit mit der Programmierung der LED-Matrix beschäftigt soll im Folgenden auf den Prototypenaufbau - insbesondere auf die Hardware – eingegangen werden.

Der Prototyp besteht aus zwölf Coreman P4 Outdoor RGB-LED-Modulen (Shenzhen Technology Co., Limited), wobei ein Modul aus 64 x 32 Pixel besteht und einen 4 mm Pixelabstand aufweist (Schulze, 2020, S. 28). Die zwölf LED-Module sind in einem 4 x 3 Matrixlayout angeordnet, dies wird in Abbildung 2-1 verdeutlicht. Präziser kann die Anordnung wie folgt beschrieben werden: „Über einen Dateneingang werden sechs in Reihe geschaltete LED-Module angesteuert, die wiederum in U-Form angeordnet sind“ (Schulze, 2020, S. 29). Der Prototyp besitzt zwei Dateneingänge, also zwei LED-Ketten, die mit Flachbandkabeln parallel geschaltet sind. In Abbildung 2-1 wird zudem der Energiefluss über die Stromversorgung verdeutlicht.

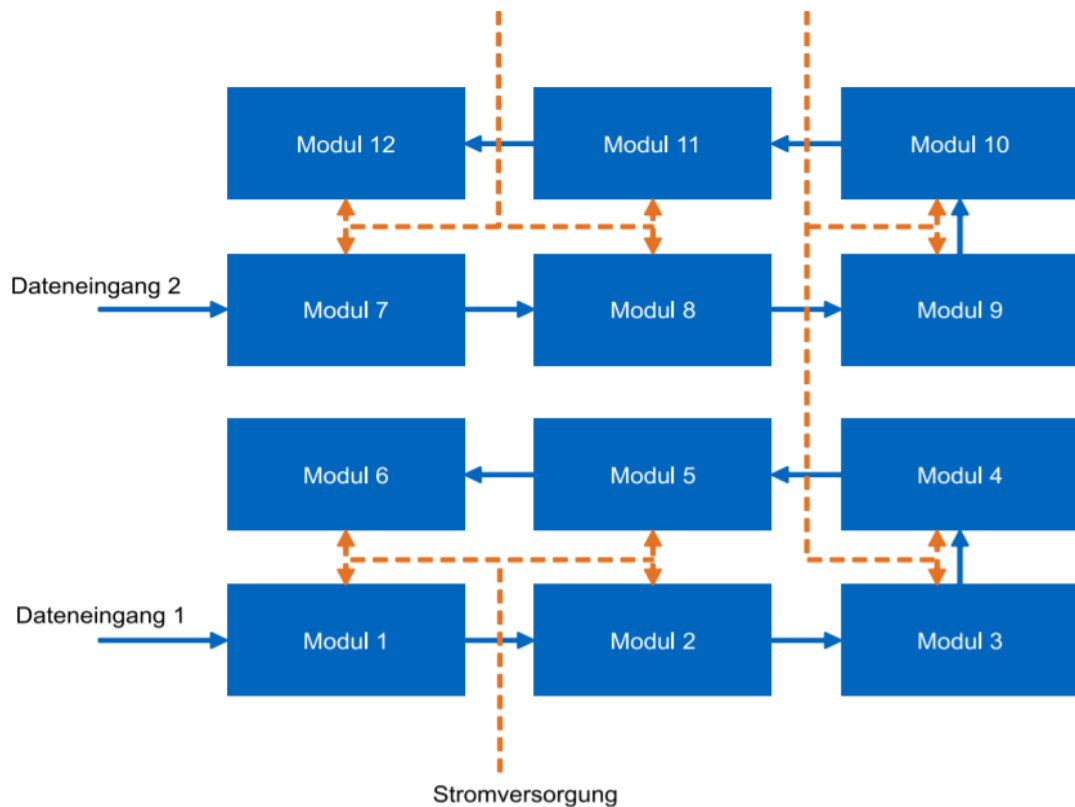


Abbildung 2-1: Aufbau der LED-Module (Schulze, 2020, S. 29)

Die Ansteuerung der LED-Module erfolgt über einen Raspberry Pi 4 Model B. Das Datenausgangssignal wird über das GPIO (general purpose input/output) gesendet. Zur einfachen Handhabung wird die „RGB-MATRIXCTRL“ Erweiterungsplatine genutzt. Diese dient sowohl als Pegelwandler als auch dazu, das Datensignal auf drei HUB75 Stecker zu verteilen (Schulze, 2020, S. 28–29). Somit können bis zu drei parallel geschaltete LED-Modulketten einfach betrieben werden. Der Pegelwandler ist notwendig, um den 3,3V Ausgangspegel des GPIO auf die für die LED-Module benötigten 5V zu wandeln (Schulze, 2020, S. 28). Das Softwaregerüst, das für die Ansteuerung benötigt wird, wird in Kapitel 4 „Implementierung“ ausführlich beschrieben. Der finale Prototyp nach Schulze wird in der nachfolgenden Abbildung dargestellt.



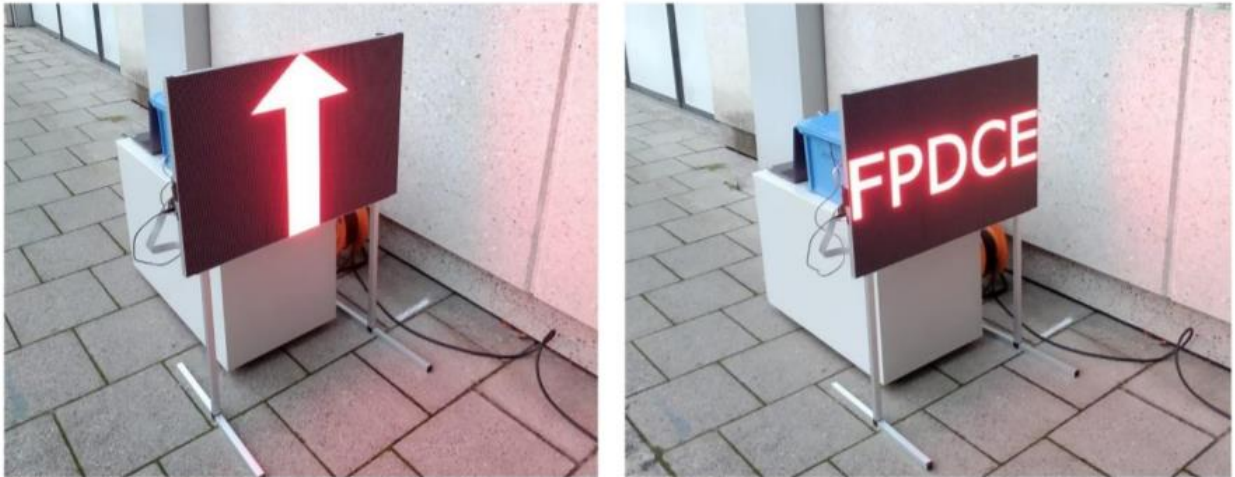


Abbildung 2-3: LED-Matrix Prototyp (Schulze, 2020, S.32)

Die zwölf verschalteten LED-Module ergeben die LED-Matrix. Diese ist auf einem Aluminiumträger befestigt, somit kann der gesamte Aufbau sicher und stabil stehen. Auf der Rückseite der LED-Matrix ist der Raspberry Pi angebracht mitsamt der Verkabelung für die Datenübertragung und Stromversorgung (siehe Abbildung 2-2).

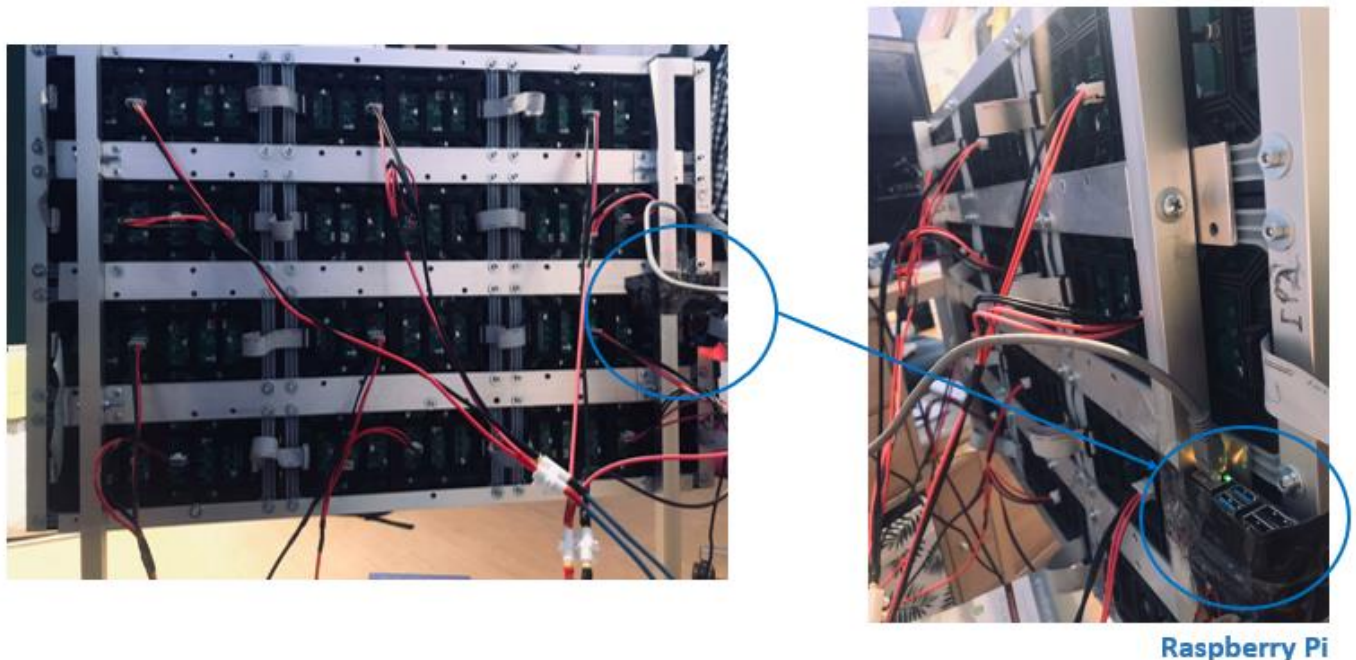


Abbildung 2-2: Rückseite LED-Matrix und Verkabelung mit Raspberry Pi

Neben der Weiterentwicklung des Prototyps beschäftigt sich die Masterarbeit von Storm auch mit der Erprobung der Erkennbarkeit der LED-Anzeige mit Hilfe einer Feldstudie (Schulze, 2020, S. 1–2). Die Studie ergab u.a. erste Anhaltspunkte für eine Lesbarkeitsdistanz. Die Lesbarkeitsdistanz ist ein wichtiger Faktor für die Kommunikation im Verkehr und spielt bei der Auswahl der anzuzeigenden Inhalte auf der LED-Matrix eine wesentliche Rolle (Schwiebacher, 2021, S. 4).

Die anzuzeigenden Inhalte wurden basierend auf ausgewählten Szenarien, bei denen die Notwendigkeit der externen Kommunikation bestand, entworfen. In einem Experten-gespräch am LfE wurden grundsätzliche Kriterien dieser Szenarien erarbeitet und basierend darauf wurden elf Szenarien skizziert, in denen sich die externe Kommunikation als sinnvoll erweist (Schwiebacher, 2021, S. 5). Für diese Szenarien wurde dann in Zusammenarbeit mit der FH Salzburg ein Symbolkatalog entwickelt. In einer anschließenden Onlinestudie, in dem Experten des UNICARagil Projekts befragt wurden, wurden die Symbole evaluiert. Die Erkennbarkeit, die Verständlichkeit und die Interpretierbarkeit waren grundlegende Anforderungen der Symbolik (Schwiebacher, 2021, S. 6). Die Symboliken wurden laut Meilensteinbericht in einem Lastenheft dokumentiert.

Weiterhin muss die Platzierung der LED-Matrizen am Fahrzeug festgelegt werden. Dabei ist zu beachten, dass die Erkennbarkeit der Symbole auf den Matrizen gewährleistet ist. In der am Lehrstuhl durchgeführten Studienarbeit von Rettenmaier et al. wurde die Erkennbarkeit von Symbolen untersucht. Dabei wurde untersucht, inwieweit Probanden bestimmte Symbole bei einer Höhe von 17cm erkennen konnten (Schwiebacher, 2021, S. 3). Die Untersuchung ergab eine ausreichende Lesbarkeitsdistanz, die für die Verwendbarkeit dieses eHMI-Konzepts spricht (Rettenmaier, Schulze & Bengler, 2020, S. 11–15).

## 2.3 ASOA Software-Architektur

Zur Inbetriebnahme der LED-Matrix am Fahrzeug ist neben der Hardware (LED-Module, Verkabelung, Energieversorgung) und der Display-Inhalte (Symboliken) auch ein Softwaregerüst notwendig. Im UNICARagil Projekt kommt eine Service-orientierte (auch „Dienste-orientiert“ genannt) Softwarearchitektur – die sog. ASOA Architektur (Automotive Service-Oriented Architecture) - zum Einsatz. Im Folgenden wird dieses Konzept ausführlich beschrieben und erläutert, da dieses Konzept die Grundlage für die in Kapitel 4 beschriebene Implementierung der LED-Matrix bildet.

Das Konzeptpapier zur ASOA Software des UNICARagil-Projekts beschreibt ASOA folgendermaßen: „Die ASOA Software dient der Umsetzung der funktionalen Architektur“ (UNICARagil, S. 1). Heutige Software-Architekturen sind oftmals rigide und werden zur Entwicklungszeit integriert, somit fehlt ihnen die Flexibilität mit den Herausforderungen der kurzen Lebenszyklen von Technologien und auch neuen Geschäftsmodellen umzugehen (Kampmann et al., 2019, S. 2102). Weiterhin beschreibt Kampmann et al., dass service-orientierte Architekturen auf zur Laufzeit integrierten Diensten basiert und flexiblere Software-Architekturen erlaubt.

Um eine klare Vorstellung über die ASOA Architektur zu bekommen werden nun die wesentlichen Merkmale von ASOA aufgezählt. Es handelt sich hierbei um eine Zusammenfassung aus dem Konzeptpapier (UNICARagil, S. 1):

- Software-Funktionen werden als Dienste mit vereinheitlichem Schnittstellenkonzept implementiert
- Die Integration der Dienste erfolgt zur Laufzeit und nicht zur Design- bzw. Entwicklungszeit
- Durch die Laufzeit-Integration und den wohldefinierten Schnittstellen wird die Software-Architektur modular
- Die wohldefinierten Schnittstellen ermöglichen eine automatische Analyse und Inferenz von möglichen Systeminstantiierungen
- Dienste sollen transparent über die Grenzen höchst unterschiedlicher Rechnerarchitekturen hinweg kommunizieren können

### **2.3.1 ASOA Dienste**

Konkrete Fahraufgaben, wie bspw. das Einschalten des Bremslichts oder die Anzeige von Inhalten auf einer LED-Matrix, werden somit durch Services realisiert. „Service“ ist in diesem Kontext mit dem Begriff „Dienst“ gleichzusetzen.

Laut Konzeptpapier (UNICARagil, S. 1) bestehen Dienste konzeptuell betrachtet aus vier Elementen: Eingänge, Ausgänge, Quellcode und die Dienstspezifikation.

Die obige Abbildung verdeutlicht das Zusammenwirken der vier Elemente. Ein Dienst beinhaltet eine Beschreibung der Ein- und Ausgänge auf Basis der Schnittstellendatenbank. In der Schnittstellendatenbank sind alle Ein- und Ausgänge eines Dienstes beschrieben.

Ein- und Ausgänge beinhalten hierbei wesentliche Informationen (Gestalt von Nutzdaten, Parameterdaten und Qualitätsdaten), die für die Funktionserfüllung des Dienstes notwendig sind. Dienste besitzen einen Quellcode, mit der Funktionen der UNICARagil Fahrzeuge erfüllt werden können. Bspw. kann die Umfeldwahrnehmung, die Verhaltensplanung oder das Stellen von Kräften über Dienste realisiert werden (vgl. UNICARagil, S. 1).

Für den Quellcode existiert ein sog. Architektur-Tool, mit dem automatisiert ein lauffähiges Code-Gerüst erzeugt wird. In dieses Gerüst kann der Entwickler dann seine Funktionen integrieren (UNICARagil, S. 7).

Zu beachten ist, dass es im Fahrzeug verschiedene Rechnerarchitekturen gibt, die über Ethernet-Verbindungen kommunizieren. Im Konzeptpapier wird festgehalten, dass die ASOA so implementiert ist, dass Dienste unabhängig von der Rechnerarchitektur transparent über Ethernet miteinander kommunizieren können (UNICARagil, S. 6).

Im UNICARagil Projekt können Dienste mit der Programmiersprache C++ implementiert werden. In C++ hat ein ASOA Dienst im Allgemeinen eine einheitliche Dateistruktur. Auch können zur Kommunikation zwischen Diensten Konzepte wie Tasks, Guarantees, Requirements oder auch Remote Procedure Calls angewandt werden. Die genaue Implementierung wird in Kapitel 4 beschrieben.

### **2.3.2 ASOA Orchestrator**

Der ASOA Orchestrator steuert die Dienstkomposition, d.h. dass verschiedene Zusammenschaltungen der Dienste in Abhängigkeit des Betriebsmodus gestellt werden. Dabei sind die zur Laufzeit möglichen Verknüpfungen zwischen Diensten bereits im Vorfeld bekannt, da die stellbaren Dienstkompositionen durch das Architektur-Tool offline vordefiniert werden (UNICARagil, S. 5). Ferner beschreibt Kampmann et al. (2019, S. 2105) den Orchestrator als ein „Zustandsübergangssystem, das auf Ereignisse mit bewachten Übergängen und Aktionen reagiert“.

### 2.3.3 Zusammenwirken der ASOA Elemente

Folgende Abbildung fasst die grundlegende Funktionalität der ASOA Architektur zusammen:

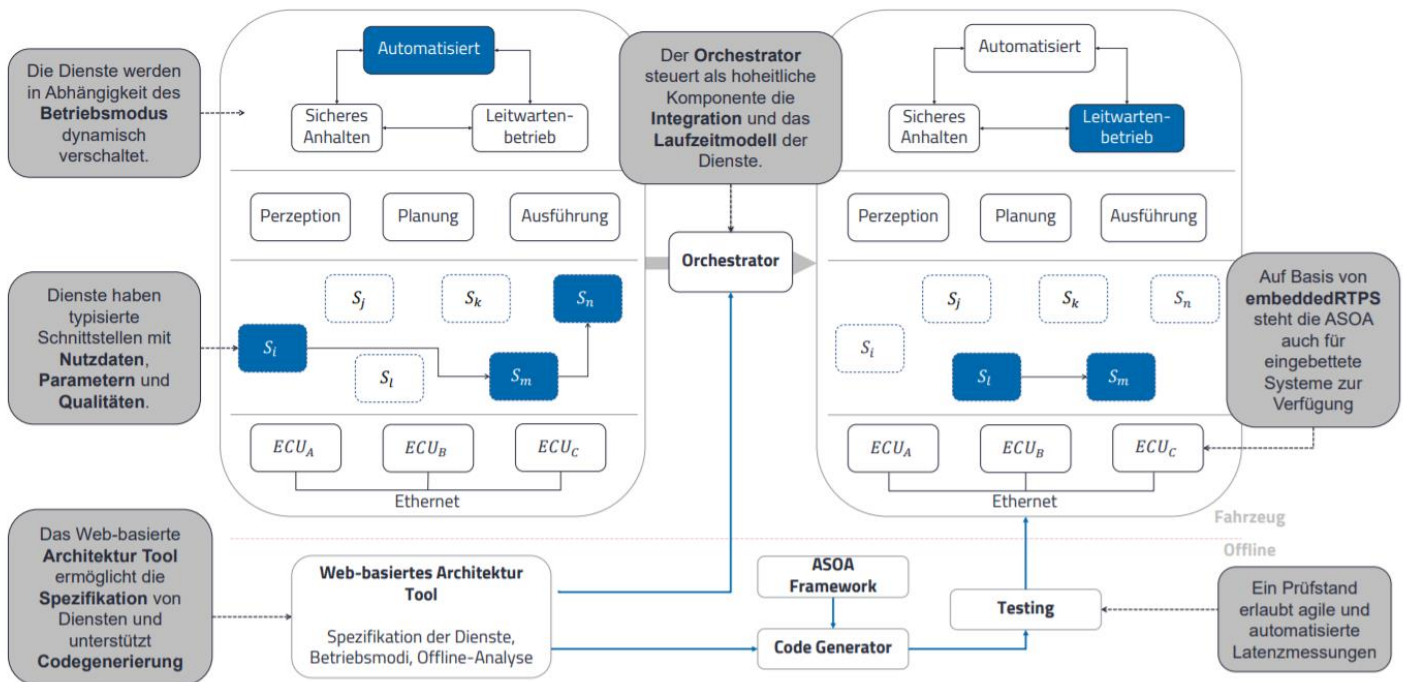


Abbildung 2-4: ASOA Funktionalität (UNICARagil news, 2021, S. 75)

## 2.4 UML-Zustandsdiagramme

Da die Programmierung der LED-Matrix anhand eines vorgegebenen UML-Zustandsdiagramms erfolgt wird hier ein allgemeiner theoretischer Überblick gegeben.

Die Unified Modeling Language (kurz: UML) „dient zur Modellierung, Dokumentation, Spezifizierung und Visualisierung komplexer Systeme, unabhängig von deren Fach- und Realisierungsgebiet“ (Rupp & Queins, 2012, S. 25). Dabei besitzt UML unterschiedliche Diagrammart, wobei in dieser Arbeit nur Zustandsdiagramme vertieft werden sollen.

Der Inhalt des folgenden Abschnitts beruht im Wesentlichen auf Kleuker (2018, S. 158–167), der in seinem Werk einen Überblick über UML-Zustandsdiagramme liefert.

Zustandsdiagramme (auch: Zustandsautomaten) beschreiben das Verhalten von Zuständen mit ihren Zustandsübergängen (Transitionen). Jedes Zustandsdiagramm enthält einen Startzustand, erkennbar als ausgefüllter Kreis, und kann mehrere Endzustände, erkennbar als umkreister ausgefüllter Kreis, enthalten. Zustände sind durch Transitionen, also durch Zustandsübergänge, die durch gerichtete Pfeile dargestellt

werden, verbunden und zudem neben dem Namen auch eine Zustandsbeschreibung enthalten. Eine Zustandsbeschreibung kann bis zu drei Teile beinhalten: den Entry-Teil, den Do-Teil und den Exit-Teil. Der Entry-Teil gibt an, was gemacht werden soll, wenn der Zustand über eine Transition neu betreten wird. Der Do-Teil beschreibt Aktionen, die ausgeführt werden sollen. Der Exit-Teil beschreibt, was passieren soll, wenn der Zustand verlassen wird.

Abbildung 2-5 zeigt einen Ausschnitt aus dem vom UNICARagil Projekt vorgegebenen UML-Zustandsdiagramm für die LED-Anzeige in der Front und im Heck. Dieser beinhaltet die eben genannten Elemente und zusätzlich auch Boolesche Bedingungen, die auf den gerichteten Pfeilen stehen (sog. Guards). „Eine Transition wird dann durchlaufen, wenn das Ereignis eintritt und die Boolesche Bedingung in eckigen Klammern erfüllt ist“, so Kleuker (2018, S. 159). Für den Ausschnitt aus Abbildung 2-5 bedeutet dies also, dass der Zustandsübergang von „Stillstand“ in „Fahrzeug wird geladen“ dann eintritt, wenn die Boolesche Bedingung „currently\_loading=true“ erfüllt, also wahr, ist. Ist die gegenläufige Bedingung „currently\_loading=false“ wahr, so findet der Zustandsübergang in die entgegengesetzte Richtung statt.

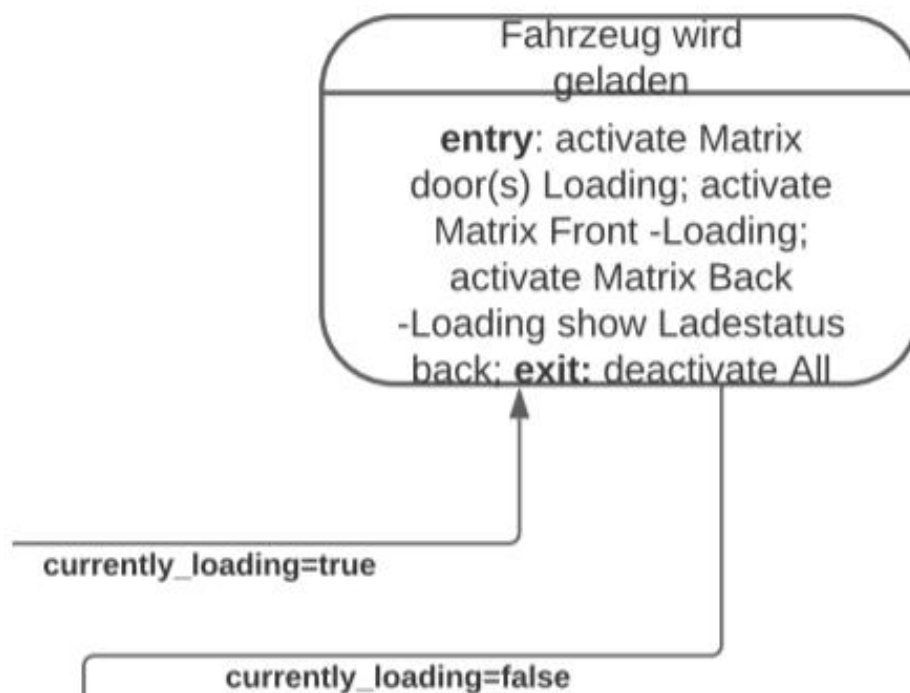


Abbildung 2-5: Ausschnitt aus dem UML-Zustandsdiagramm

Weiterhin führt Kleuker (2018, S. 160) an, dass Zustandsdiagramme visuell eingängig sind und sich deshalb sehr gut zur Diskussion in Gruppen eignet.

### 3 Problemstellung

Wie in Kapitel 1 und 2 beschrieben ergeben sich aus dem Einsatz eines eHMI an autonomen Fahrzeugen klare Vorteile, wie erhöhte Verkehrseffizienz, erhöhte Sicherheit und stärkere Kooperation im Verkehr. Im UNICARagil Fahrzeug soll die Kommunikation optisch, also über ein Display bzw. über eine LED-Matrix erfolgen. Wie erwähnt existiert bereits ein finales Interaktionskonzept mit einem fertigen Symbolkatalog. Auch existiert ein UML-Zustandsdiagramm, das implementiert werden soll. Im Folgenden wird die genaue Problemstellung und Aufgabe dieser Semesterarbeit besprochen.

Konkret soll für das eHMI der Fahrzeuge jeweils eine LED-Matrix in der Front und im Heck verwendet werden. Ziel dieser Arbeit ist es, die LED-Matrizen mit dem Microcontroller Raspberry Pi anzusteuern. Die für den reibungslosen Verkehrsablauf relevanten Informationen, also die im Projekt festgelegten Symboliken, sollen auf den LED-Matrizen dargestellt werden. Diese Symboliken sind entweder Farbbilder im PNG Format oder Animationen im GIF Format.

Die Funktionen und Zustände der LED-Matrizen werden durch ein bereits existierendes UML-Zustandsdiagramm beschrieben. Dieses Diagramm beschreibt dabei die Zustände des eHMI für definierte Verkehrssituationen, in denen die externe Kommunikation notwendig ist. Der Kern der Arbeit liegt in der Implementierung dieses Zustandsdiagrammes am Raspberry Pi, unter Beachtung der zu Grunde liegenden ASOA Software-Architektur und unter Ausnutzung einer C++ Softwarebibliothek (Zeller, 2021) zur Steuerung der LEDs. Abbildung 3-1 auf nachfolgender Seite verdeutlicht die Fahrzeugkonstellation mit den LED-Matrizen und den Zuständen aus dem UML-Diagramm.

Anschließend soll für den Lösungsvorschlag ein Testkonzept entworfen werden und anschließend umgesetzt werden. Mit Hilfe dessen soll das Programmverhalten untersucht werden, d.h. es wird erprobt, wie sich die LED-Matrizen in der Front und im Heck während des Tests verhalten. Im Test auftretende Fehler sollen so behoben werden, dass die vorgesehene Funktion der LED-Anzeige erfüllt wird. Dabei ist die Funktionserfüllung der primäre Leitgedanke.

Zudem soll auch das Laufzeitverhalten betrachtet werden. Dabei steht die Betrachtung der Ladezeiten der Symboliken im Fokus. Die Implementierung soll eine schnelle Anzeige ermöglichen, d.h. die Ladezeit der Symboliken soll möglichst gering sein.

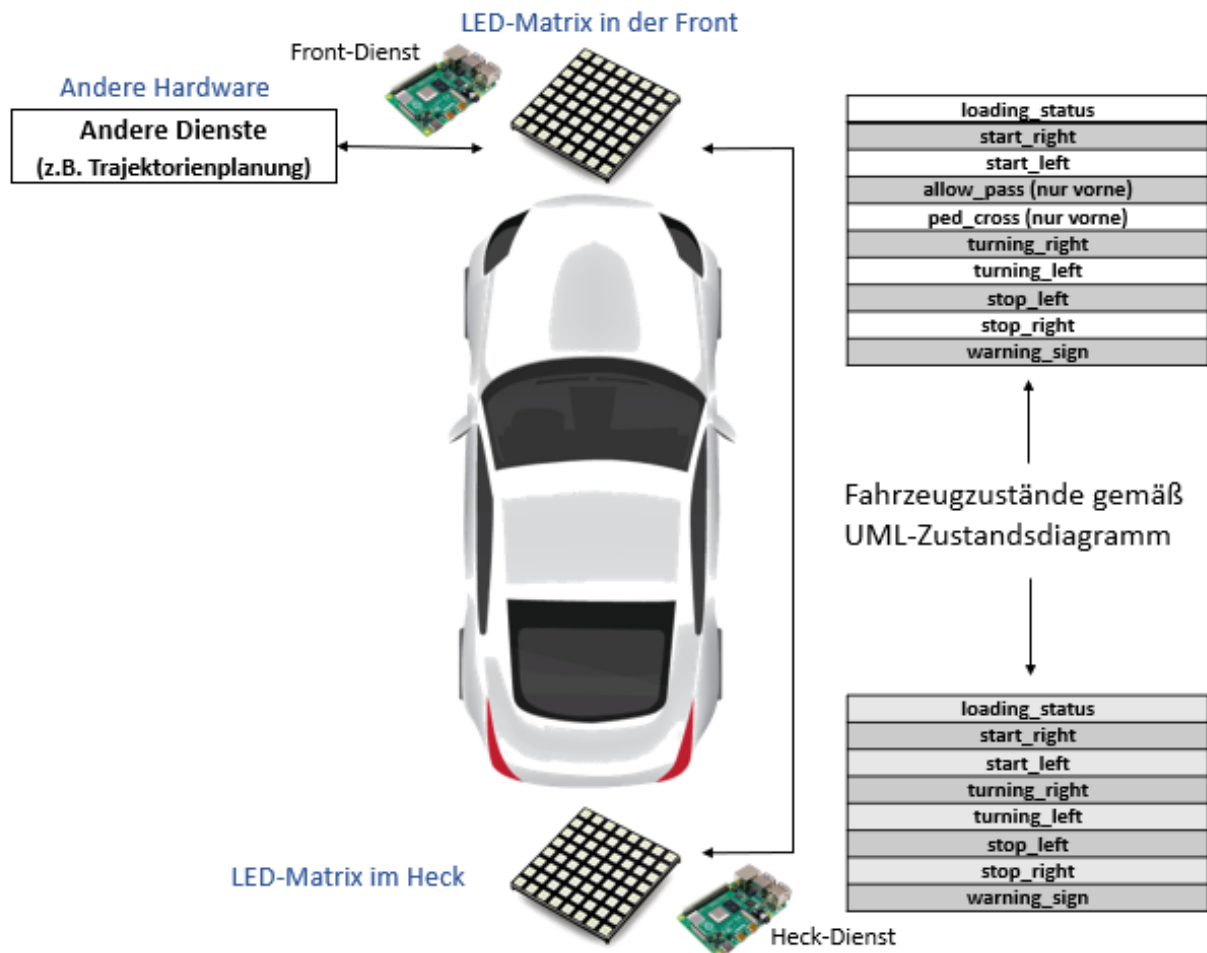


Abbildung 3-1: Fahrzeugkonstellation mit LED-Matrizen und Zuständen

Folgender Ablaufplan stellt die Vorgehensweise bei der Bearbeitung dieser Problemstellung dar:



## Programmierung LED-Matrix mit Raspberry Pi

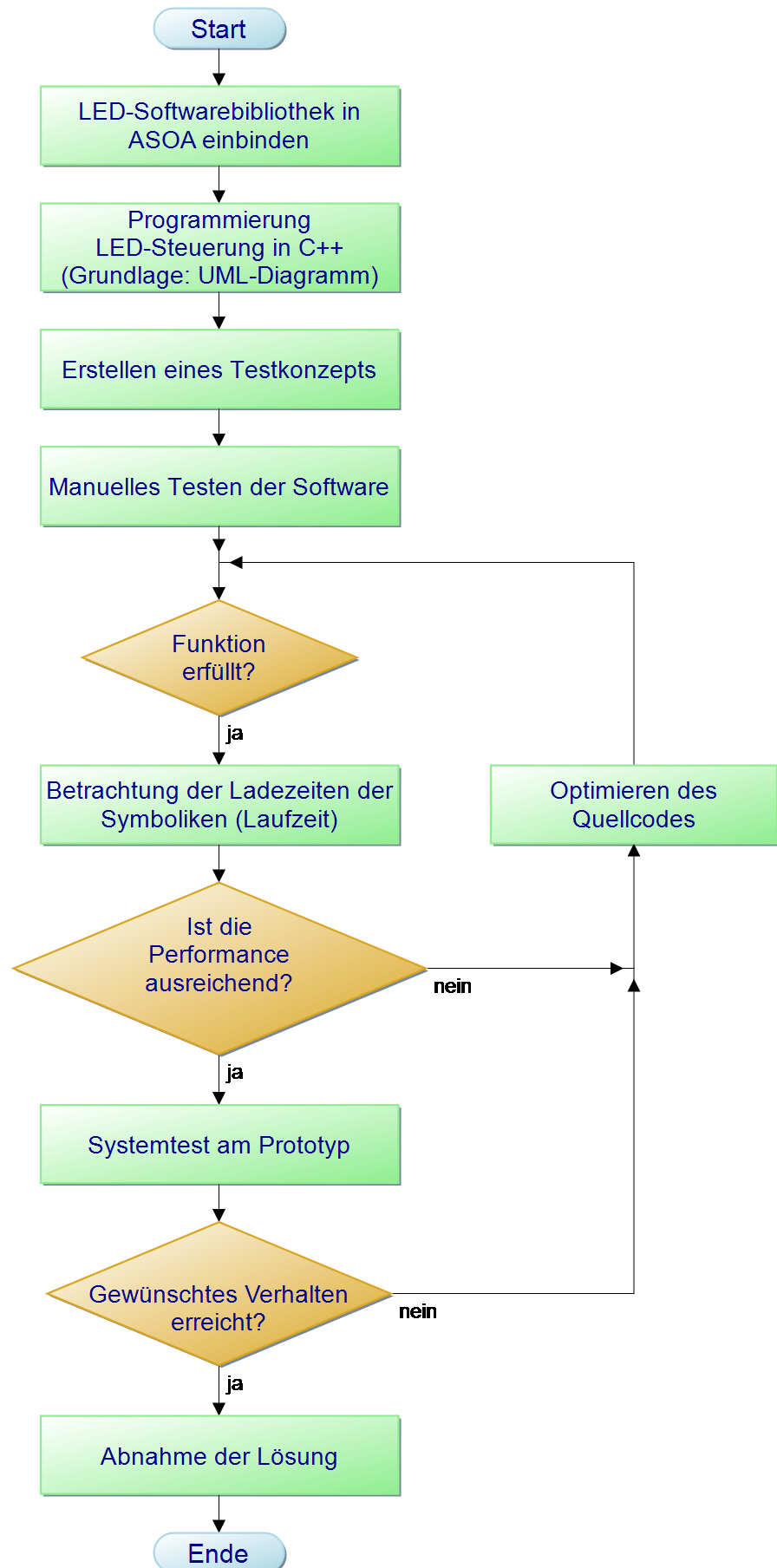


Abbildung 3-2: Ablaufplan Implementierung

## 4 Implementierung

Dieses Kapitel behandelt die konkrete Programmierung der LED-Matrix. Zunächst wird die Anbindung vom Raspberry Pi zur LED Hardware beschrieben. Danach wird die für die Anzeige benutzte Softwarebibliothek von Henner Zeller (2021) eingeführt. Daraufhin wird die Integration der Bibliothek mit der ASOA Architektur erläutert und schließlich die Implementierung der Funktionalität in C++ erklärt. Das resultierende Programm soll hierbei als ein möglicher Lösungsvorschlag betrachtet werden. Die Implementierung basiert auf dem vorliegenden UML-Zustandsdiagramm, die Zustände und die dazugehörigen Transitionen aus dem Diagramm werden dabei sukzessive implementiert. Für die Programmierung werden grundlegende Konzepte der C++ Sprache angewandt, wie bpsw. Variablen, Datentypen oder auch objektorientierte Konzepte. Diese werden in diesem Kapitel weitestgehend als bekannt vorausgesetzt und sollen daher nicht neu eingeführt und erläutert werden. Weiterführende Ansätze wie z.B. Multithreading oder das Server-Client-Modell werden bei der Anwendung und im darauffolgenden Testkonzept dann ausführlicher erklärt. UML-Klassendiagramme sollen hierbei die Funktionsweise der Software verdeutlichen, woraus ein Gesamtverständnis für die Lösung abgeleitet werden kann. Kapitel 5 und 6 befassen sich dann mit dem Softwaretest und der Diskussion der Lösung.

### 4.1 Software-Aufbau und Einrichtung

Im Folgenden wird die für die Programmierung verwendete Softwareumgebung und die Anbindung zur Hardware dargestellt.

#### 4.1.1 Einrichtung Raspberry Pi

Die Ansteuerung der LED-Matrix erfolgt über einen Mikrocontroller, wobei im UNICARagil Projekt – wie in Kapitel 2.2 erwähnt - ein Raspberry Pi 4 Computer Model B zum Einsatz kommt. Für den Betrieb mit dem Raspberry Pi muss im Vorfeld ein für die Anwendung geeignetes Betriebssystem installiert werden. In dieser Semesterarbeit wird die Linux Distribution „Ubuntu 18.04.6 LTS Server“ gewählt. Neben der Server-Version gibt es auch eine Desktop-Version, bei der eine benutzerfreundliche grafische Oberfläche existiert. Da der Raspberry Pi im Fahrzeug nur eine LED-Matrix betreiben soll, d.h. nur seinen dafür notwendigen Dienst ausführen soll genügt die Server-Version ohne grafische Oberfläche. Die dafür benötigten Ressourcen würden den Raspberry Pi unnötig belasten. Ausschlaggebend für dieses gewählte Betriebssystem ist zudem die Kompatibilität mit der ASOA Architektur und der C++ Softwarebibliothek von Henner

Zeller. Sowohl ASOA als auch die Bibliothek sind mit Ubuntu 18 kompatibel. Die Bibliothek wird in Kapitel 4.1.5 näher beschrieben.

Nachdem Ubuntu 18 auf dem Raspberry Pi installiert wurde muss die ASOA Architektur installiert werden. Dazu wird die vom LfE herausgegebene aktuellste Version des ASOA Core (Version 0.3.0) installiert, die den Betrieb von Diensten ermöglicht. Zu beachten ist, dass es für den ASOA Core zwei verschiedene Kompilierungen gibt: eine AMD-Version und eine ARM-Version. Zweiteres ist der für den Raspberry Pi kompatible ASOA Core. Die ARM-Architektur ist dabei der Auslegungstyp des Prozessors im Raspberry Pi, wobei Prozessoren in Mikrocontrollern meistens als ARM-Architektur ausgelegt sind (Follmann, 2018, S. 12). Entsprechend ist die ASOA Core AMD-Version kompatibel mit der Mikroprozessor-Architektur des Unternehmens AMD. Neben ASOA müssen zudem die Werkzeuge „build-essential“ und „libssl-dev“ (OpenSSL Foundation, Inc., 2021) installiert werden, die grundlegende Werkzeuge zur Softwareentwicklung liefern. Diese lassen sich sehr einfach über die Paketverwaltung „Advanced Packaging Tool“ bzw. dem Kommandozeilen-Programm „apt“ installieren. Auf die detaillierte Dokumentation der Kommandozeilen-Befehle für die Installation von ASOA und den Kompilier-Tools wird hier zur besseren Übersichtlichkeit verzichtet. Für den Betrieb von ASOA Diensten auf dem Raspberry Pi sind weitere Build-Tools notwendig, diese werden in Kapitel 4.1.4 beschrieben.

Eine komfortable Möglichkeit den Raspberry Pi zu bedienen ist die Steuerung über eine Secure Shell Fernverbindung (kurz: SSH). Dazu wird der SSH-Zugang aktiviert und der Raspberry Pi per Ethernet mit einem Rechner verbunden. Das Programm „PuTTY“ ermöglicht dann den Fernzugriff per SSH (*putty.org*, 2018). Damit eine Verbindung erfolgreich hergestellt werden kann muss zudem ein Netzwerk konfiguriert werden. Dies wird näher in Kapitel 4.1.3 behandelt.

#### **4.1.2 Einrichtung virtuelle Maschine und Orchestrator**

Damit der Dienst auf dem Raspberry Pi mit anderen Diensten kommunizieren kann wird der ASOA Orchestrator benötigt, der - wie in Kapitel 2.3.2 beschrieben - Dienste zusammenschalten kann. Das in dieser Semesterarbeit zu entwickelnde Programm für die LED-Steuerung soll anschließend mit Hilfe von einem sog. „Dummy-Dienst“ (siehe Kapitel 5.1) getestet werden, daher wird der Orchestrator benötigt. Der Orchestrator ist – ähnlich zum ASOA Core - linux-basiert und ist mit Ubuntu 18 kompatibel. Da im Projekt keine ARM-kompatible Version des Orchestrators existiert und diese auch i.d.R. nicht notwendig ist, wird eine „virtuelle Maschine“ (kurz: VM) aufgesetzt, auf dem der Orchestrator installiert wird. Eine virtuelle Maschine ermöglicht den Betrieb weiterer Betriebssysteme in einer virtuellen Umgebung auf dem übergeordneten Computer, dem sog.

„Host-PC“. Es sei erwähnt, dass die Installation auch auf einem nativen Linux-System funktioniert und lauffähig ist. Mit Hilfe der Software „VirtualBox“ des Unternehmens Oracle lässt sich so eine virtuelle Maschine bspw. auf einem Windows PC betreiben. Voraussetzung für die Installation des Orchestrators ist eine erfolgreiche Installation des ASOA Cores. Sobald der Raspberry Pi mitsamt Tools und die VM mit Orchestrator eingerichtet sind, können beide Systeme kommunizieren und der Orchestrator kann Dienste auf dem Raspberry Pi und der virtuellen Maschine verschalten, sofern ein Netzwerk eingerichtet ist. Das folgende Kapitel beschäftigt sich mit der Netzwerkkonfiguration.

#### **4.1.3 Netzwerkkonfiguration**

Dieses Unterkapitel beschäftigt sich mit der Konfiguration eines Netzwerkes, welches die Kommunikation zwischen dem Raspberry Pi, der virtuellen Maschine und dem Host-PC ermöglicht. Ausgewählte Grundlagen zu Computer-Netzwerken werden hier sehr kompakt vorgestellt, sollen aber nicht im Detail behandelt werden. Stattdessen liegt der Fokus darauf, einen kurzen Überblick auf die konkrete Netzwerkkonfiguration dieses Projekts zu geben und zu zeigen, wie die Kommunikation zwischen den Geräten ermöglicht werden kann.

Nach Zisler (2015, S. 75) gelingt die Adressierung in einem Netzwerk über Hardware- und Internetprotokoll-Adressen (IP-Adressen). In diesem Entwicklungsprozess eignet sich ein privates lokales Netzwerk, um die Dienste zukünftig auf der VM und dem Raspberry Pi laufen zu lassen und zu testen. Dafür müssen die IP-Adressen selbst vergeben werden (Zisler, 2015, S. 83). Dabei ist zu beachten dass die IP-Adressen innerhalb eines Netzwerkes einzigartig sein müssen. Neben der IP-Adresse ist auch die Netz- bzw. Subnetzmaske essentiell. Zisler (2015, S. 86) führt an, dass eine Netzmaske alle möglichen Adressen eines Netzes umfasst und eine Subnetzmaske den Adressraum unterteilt. Diese müssen bei der Netzwerkkonfiguration beachtet werden.

Zunächst wird die Konfiguration des Entwicklungsrechners – hier handelt es sich um einen Windows PC, auf dem die Programmierung statt findet - vorgenommen. Dazu wird die Netzwerkkarte des Rechners eingestellt. Die IP-Adresse (IP-Version IPv4) wird manuell gesetzt. Alternativ kann die Option „IP-Adresse automatisch beziehen“ gewählt werden, dann holt sich der Rechner die Adresse von einem sog. DHCP-Server (Dynamic Host Configuration Protocol) ab, d.h. die Zuweisung der Netzwerkkonfiguration findet durch einen Server statt (Zisler, 2015, S. 115). Da hier ein privates lokales Netzwerk verwendet wird, soll die IP-Adresse stattdessen manuell (statisch) festgelegt werden. Abbildung 4-1 zeigt die gewählte Konfiguration für den Windows Rechner. Hier wird 192.168.1.42 als IP-Adresse frei gewählt. Nach Zisler bilden die ersten drei Bytes den Netzwerkanteil, das letzte Byte bildet den Hostanteil der Adresse. Alle IP-Adressen bei

denen der Netzanteil identisch ist, befinden sich im gleichen Netzwerk. Daraus wird deutlich, dass andere Geräte im Netzwerk erst kommunizieren können, wenn sie eine statische IP-Adresse besitzen mit gleichem Netzwerkanteil, also mit 192.168.1 beginnen. Zusätzlich muss die Subnetzmaske für jeden Teilnehmer des Netzwerks korrekt eingestellt werden. Hier wurde die Subnetzmaske 255.255.255.0 gewählt. Dieses trennt die IP-Adresse nach dem dritten Byte und ermöglicht im letzten Byte die Adressierung von mehreren Hosts. Verwendet jedes Gerät im Netzwerk also die Subnetzmaske 255.255.255.0 und eine IP-Adresse mit dem Netzwerkanteil 192.168.1.X und unterschiedlichem Hostanteil X („42“ für den Windows-PC), so können die Geräte über dieses Netzwerk Daten austauschen. Wichtig hierbei ist, dass zu konfigurierende Geräte im Netzwerk angeschlossen sind (Zisler, 2015, S. 114).

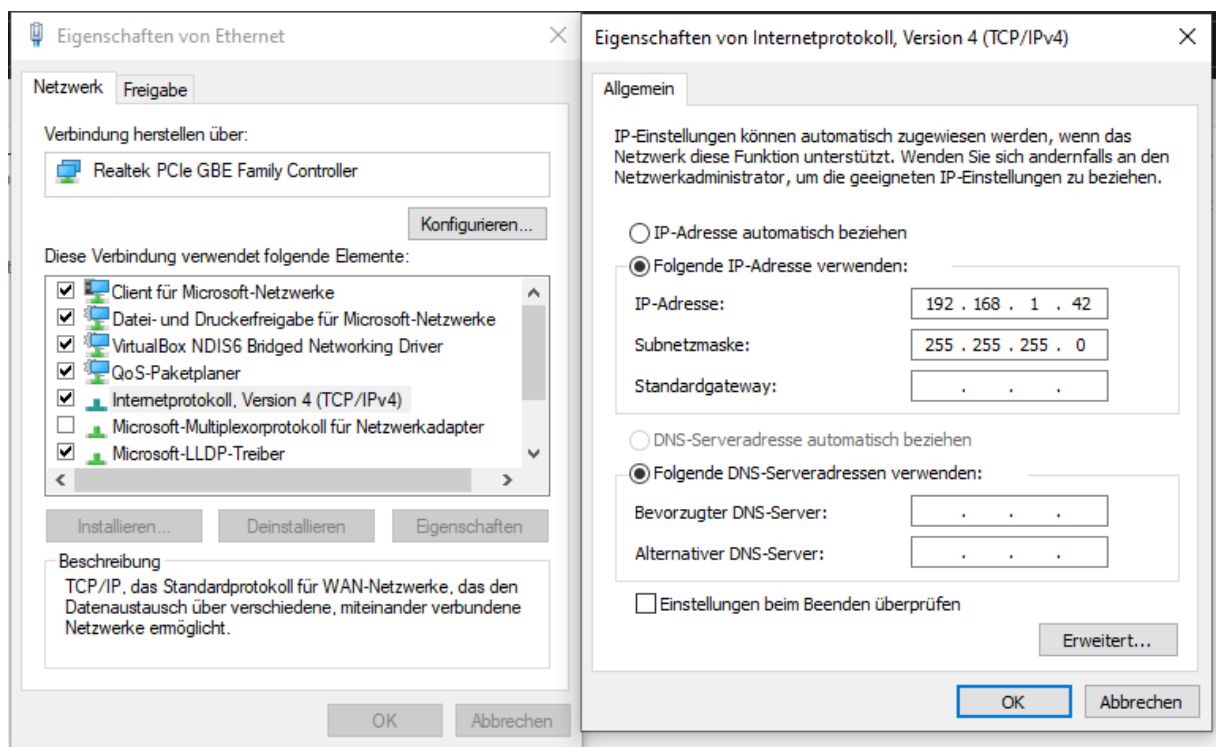


Abbildung 4-1: Netzwerkkonfiguration Windows Rechner

Nachdem die Netzwerkkarte des Windows-PC eingestellt wurde, wird der Raspberry Pi konfiguriert. In Ubuntu 18 gibt es „Netplan“, ein Netzwerk-Tool mit Hilfe dessen sich ein Netzwerk auf einem Linux-System einfach konfigurieren lässt (*Netplan / Backend-agnostic network configuration in YAML*, 2021d). Der Webseite des Entwicklers (2021d) zufolge genügt die Erstellung einer YAML-Datei, in der die IP-Zuweisung und Subnetzmaske spezifiziert wird. Dabei liest Netplan die erstellte YAML-Datei aus einem vorgegebenen Pfad im System und generiert die Konfiguration automatisch. YAML ist eine sog. „markup language“ und wird oftmals für Konfigurationsdateien verwendet. Über Netplan wird dem Raspberry Pi nun die IP-Adresse 192.168.1.2 mit Subnetzmaske

255.255.255.0 zugewiesen, sodass dieser im selben Netzwerksegment ist wie der Windows PC. Die dafür erstellte YAML-Datei befindet sich im Anhang in Kapitel B

In der grafischen Oberfläche der VM mit Ubuntu 18 Desktop wird analog die IP-Adresse und Subnetzmaske des Netzwerkadapters gesetzt. Damit ist das Netzwerk eingerichtet und die Geräte können darüber kommunizieren. Auch der Orchestrator, der auf der VM installiert ist, kann nun laufende Dienste im Netzwerk erkennen. Folgende Abbildung zeigt den schematischen Aufbau des Netzwerkes.

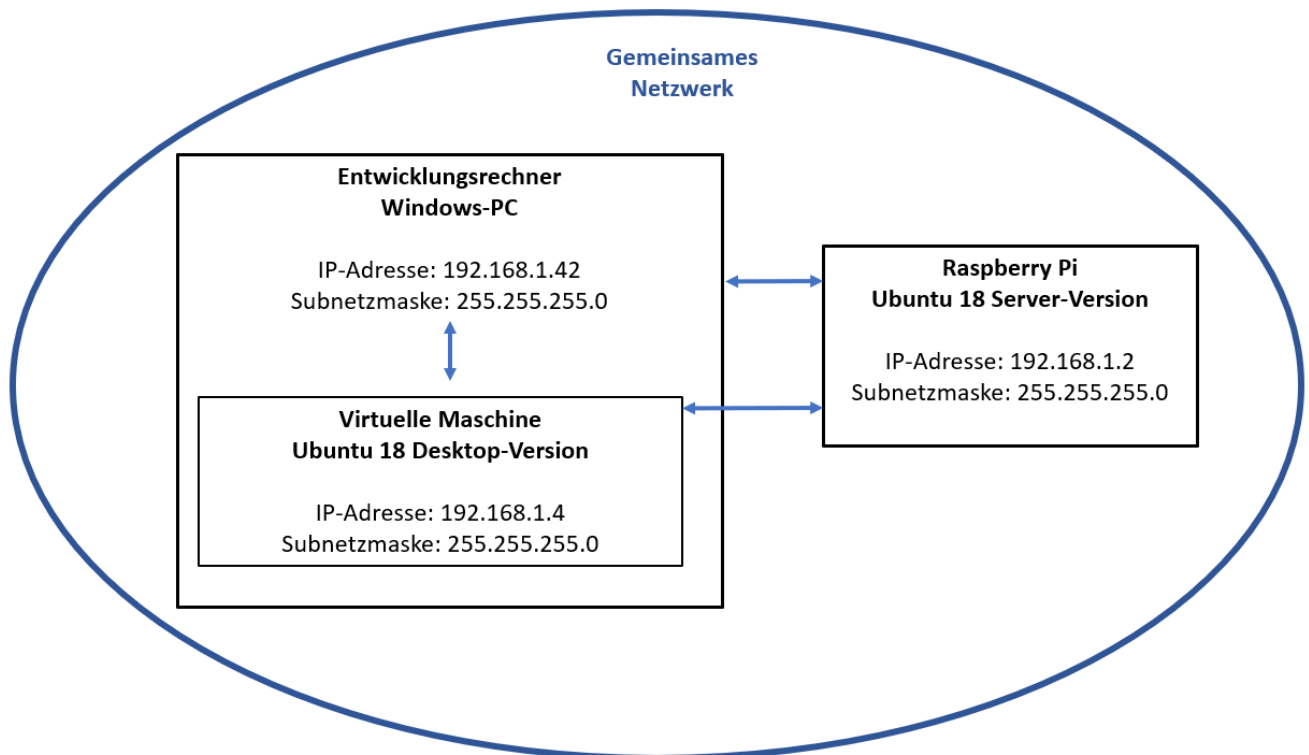


Abbildung 4-2: Prinzipbild des gemeinsamen Netzwerkes

#### 4.1.4 Build-Tools

Nachdem der Raspberry Pi und die virtuelle Maschine eingerichtet sind und die Kommunikation über das im vorherigen Kapitel beschriebene Netzwerk funktioniert können erste ASOA Dienste programmiert werden.

Für den Kompiliervorgang in C++ werden im Vorfeld einige Werkzeuge, sog. Build-Tools, benötigt. So werden die Werkzeuge „make“ und „cmake“ von Kitware (2021) benötigt. Make wird bei der Installation des Softwarepakets build-essential mitgeliefert. Build-Tools sind nicht notwendig aber hilfreich, da jedes Softwareprojekt nach einer gewissen Konfiguration gebaut werden muss, d.h. der Compiler und Linker müssen nach einem genauen Schema zusammenarbeiten, um das ausführbare Programm (.exe) zu bauen. Da dieser Prozess sehr oft im Entwicklungsprozess eines Softwareprojekts statt

findet ist es sinnvoll einfache Tools zu haben mit denen man diesen Prozess reproduzieren bzw. automatisieren kann (Clemencic & Mato, 2012, S. 1). Clemencic und Mato (2012, S. 1) merken weiterhin an, dass es gerade für Open Source Projekte noch wichtiger ist einfache Build-Tools zu haben, da die veröffentlichten Projektversionen eventuell von Endnutzern benutzt bzw. Kompiliert werden könnten, die wahrscheinlich keine professionellen Entwickler sind. Insgesamt ist cmake einfach, mächtig und effizient (Clemencic & Mato, 2012, S. 2).

Mit cmake können aus Skriptdateien (genauer: CMakeLists.txt) Projekte bzw. sog. Makefiles erstellt werden. So kann bspw. aus vielen verschiedenen C++ Dateien eines Projekts (z.B. Header-Files und cpp-Files) mit Hilfe von cmake und einem selbst geschriebenen cmake-Skript ein Linux-kompatibles C++ Makefile oder auch ein Visual Studio Projekt erzeugt werden.

Hierbei ist Make ein „Werkzeug, das die Kompilierung großer Gruppen von Quelldateien verwaltet“ (Seaman, 2013, S. 3). Die für den Kompilierungsvorgang benötigten Regeln und Abhängigkeiten werden in dem oben erwähnten Makefile vom Entwickler definiert. Ein Makefile ist eine Textdatei, die „Makefile“ heißt und prinzipiell eine Anleitung für den Compiler und dem Linker beinhaltet. Für die Entwicklung des ASOA Dienstes zur Steuerung der LED-Matrix werden cmake und make als unterstützende Build-Tools herangezogen.

#### **4.1.5 Hzeller-Library**

Um die LED-Matrix mit C++ ansteuern zu können wird die Open-Source Softwarebibliothek „rpi-rgb-led-matrix“ des Entwicklers Henner Zeller (kurz: hzeller) genutzt (2021). Die Bibliothek ermöglicht die Ansteuerung von bis zu drei Ketten von 64x64, 32x32, 16x32 Pixel oder ähnlichen RGB LED Displays mit Hilfe des Raspberry Pi GPIO. Die Bibliothek steht unter der GNU General Public License v2.0. Mit Hilfe der Bibliothek kann man einzelne Pixel durch die Vorgabe eines Farbschemas im RGB-Farbraum ansteuern, man kann zudem Dateien in verschiedenen Bildformaten (.jpg, .png oder auch .gif) und auch Videoformate abspielen. Die meisten Funktionalitäten sind in den Header-Dateien, z.B. in include/led-matrix.h enthalten und können für eigene Softwareprojekte eingebunden werden. Für den Einstieg gibt es mitgelieferte Programmbeispiele. Im Ordner „examples-api-use“ findet man sog. Demos, also vorgefertigte C++ Programme, die verschiedene Bilder bzw. Animationen anzeigen. Diese kann man über das Terminal ausführen. Der Ordner „utils“ beinhaltet den sog. Image-Viewer und den Video-Viewer. Dies sind ebenfalls Programme, mit Hilfe dessen man über das Terminal (eigene) Bilder, Animationen und Videos abspielen kann.

Da nicht jede LED-Matrix bzw. jedes LED-Panel gleich aufgebaut ist gibt es Einstellungen, die man für jeden spezifischen Aufbau verändern kann. Das Setzen bestimmter Einstellungen erfolgt über sog. Flags, also über zusätzliche Terminal-Angaben. So kann man bspw. mit den Flags „—led-cols=64 —led-rows=32“ die fehlerlose Anzeige auf einem 64x32 Display ermöglichen. Dabei gibt es eine Vielzahl von Flags mit der man den Panel-Typ, das Multiplexing, die GPIO-Geschwindigkeit und vieles mehr einstellen kann. Für eine fehlerfreie Anzeige auf der in UNICARagil benutzten LED-Matrix ist eine korrekte Einstellung aller Flags essentiell.

#### 4.1.6 Gesamtkonzept LED-Ansteuerung

Folgende Abbildung verdeutlicht das Gesamtkonzept der Hardware-Software-Anbindung zur Programmierung des Raspberry Pi.

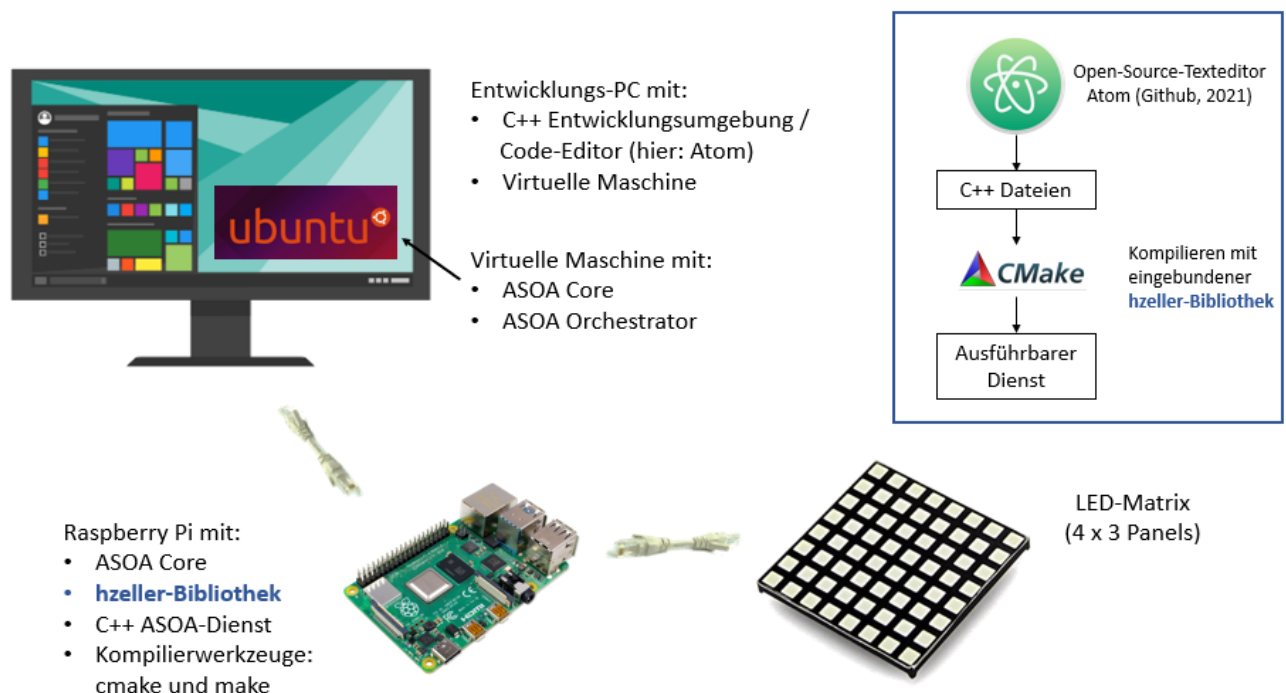


Abbildung 4-3: Gesamtkonzept Hardware-Software-Anbindung

## 4.2 Programmierung

Dieses Unterkapitel beschäftigt sich mit der konkreten Programmierung des Raspberry Pi zur Steuerung der LED-Matrix in der Front und im Heck. Zunächst werden Überlegungen getroffen, aus denen heraus ein Plan für den Programmierablauf entstehen soll. Anschließend wird das Einbinden der hzeller-Bibliothek in die ASOA-Dienststruktur erläutert. Die für die Funktionalität relevanten Ein- und Ausgangsdaten werden dann mit Hilfe des Architekturtools in sog. IDL-Files (Interface Definition Language; oder auch: Interface-Dateien) definiert und innerhalb der ASOA-Dienste verwendet. Ein IDL-File



definiert, welche Daten zwischen Diensten ausgetauscht werden. Die eigentliche Funktionalität wird dann in der rgbController Klasse und in den Front- und Back-Service Klassen implementiert. Der rgbController ist eine Header-Datei, die wesentliche Funktionen für die LED-Anzeige mit Hilfe der hzeller-Bibliothek beinhaltet. Diese Klassen orientieren sich an dem vorgegebenen UML-Zustandsdiagramm und erfüllen die darin dargestellte Funktionalität. Die Umsetzung der Zustände und deren Übergänge soll konzeptionell präsentiert werden. Dabei soll auch auf einige in dieser Lösung verwendeten programmiertechnischen Mittel konkret eingegangen werden.

#### **4.2.1 Vorüberlegungen**

Die Programmierung des Raspberry Pi soll im Wesentlichen die Anzeige von Bildern und Animationen auf der LED-Matrix in der Front und auf der LED-Matrix im Heck des UNICARagil Fahrzeuges ermöglichen. Was dabei angezeigt werden soll hängt vom Zustand ab, in dem sich das Fahrzeug gerade befindet. Alle möglichen Zustände sind im UML-Zustandsdiagramm enthalten.

Aus dem Diagramm gehen zehn mögliche Anzeigezustände hervor, wobei im Heck nur acht angezeigt werden sollen. Folgende Tabelle zeigt die Zustände mit Angabe, ob sie in der Front bzw. im Heck angezeigt werden.

Gemäß obiger Tabelle werden die Zustände „Engstelle beidseitig – Vorfahrt wird gewährt“ und „Fußgänger an Zebrastreifen“ nur in der Front-Matrix angezeigt. Dies ist sinnvoll, da bei Ersterem das Gewähren der Vorfahrt nur den Verkehrsteilnehmer betrifft, der sich unmittelbar vor dem autonomen Fahrzeug im Gegenverkehr befindet. Bei Zweiterem betrifft die Anzeige nur den Fußgänger, der sich ebenfalls unmittelbar vor dem Fahrzeug befindet. Die Anzeige dieser beiden Zustände in der Heck-Matrix ist damit überflüssig.

Es bietet sich an, einen Dienst für die Front (Front\_Service.hpp) und einen separaten Dienst für das Heck (Back\_Service.hpp) zu erstellen, die jeweils alle relevanten Zustände mit Übergängen enthalten. Diese Dienste sind aus mehreren Gründen unterschiedlich. Wie oben erläutert soll die hintere LED-Matrix nur acht Zustände anzeigen, die vordere zehn. Ein weiterer Grund ist, dass in den acht Zuständen der hinteren Matrix gleichzeitig auch die vordere Matrix aktiv sein soll und dabei das Gleiche anzeigen soll. Zur Vereinfachung kann man nun die Logik hinter den Zuständen – damit sind die Bedingungen gemeint, die für den Übergang in einen Zustand nötig sind – in der Front-Matrix implementieren. Die Front-Matrix kann dann seinen Zustand an die Heck-Matrix weitergeben und die Heck-Matrix initiiert dann das Verhalten der Front-Matrix. Durch

dieses Prinzip muss nicht die komplette Logik jedes einzelnen Zustandes im Back-Service implementiert werden, vielmehr folgt der Back-Service dem Verhalten des Front-Service. Dadurch verringert sich die Anzahl der Quellcodezeilen im Back-Service und das Programm wird kompakter, geradliniger in der Struktur und in der Logik und insgesamt leichter verständlich. Ein weiterer Grund ist, dass sich andere Dienste, die das eHMI nutzen, nur mit dem Front-Service verschalten müssen und nicht auch noch zusätzlich mit den Back-Service. Das bedeutet, dass sich der befehlende Dienst nur mit dem Front-Service verbinden muss und damit nicht nur die Front-Matrix steuern kann, sondern auch indirekt die Heck-Matrix. Das eHMI, hier bestehend aus Front- und Heck-Matrix, kann demnach gesteuert werden, alleine durch das Übermitteln der Zustände an den Front-Service. Folgende Abbildung verdeutlicht den Zusammenhang.

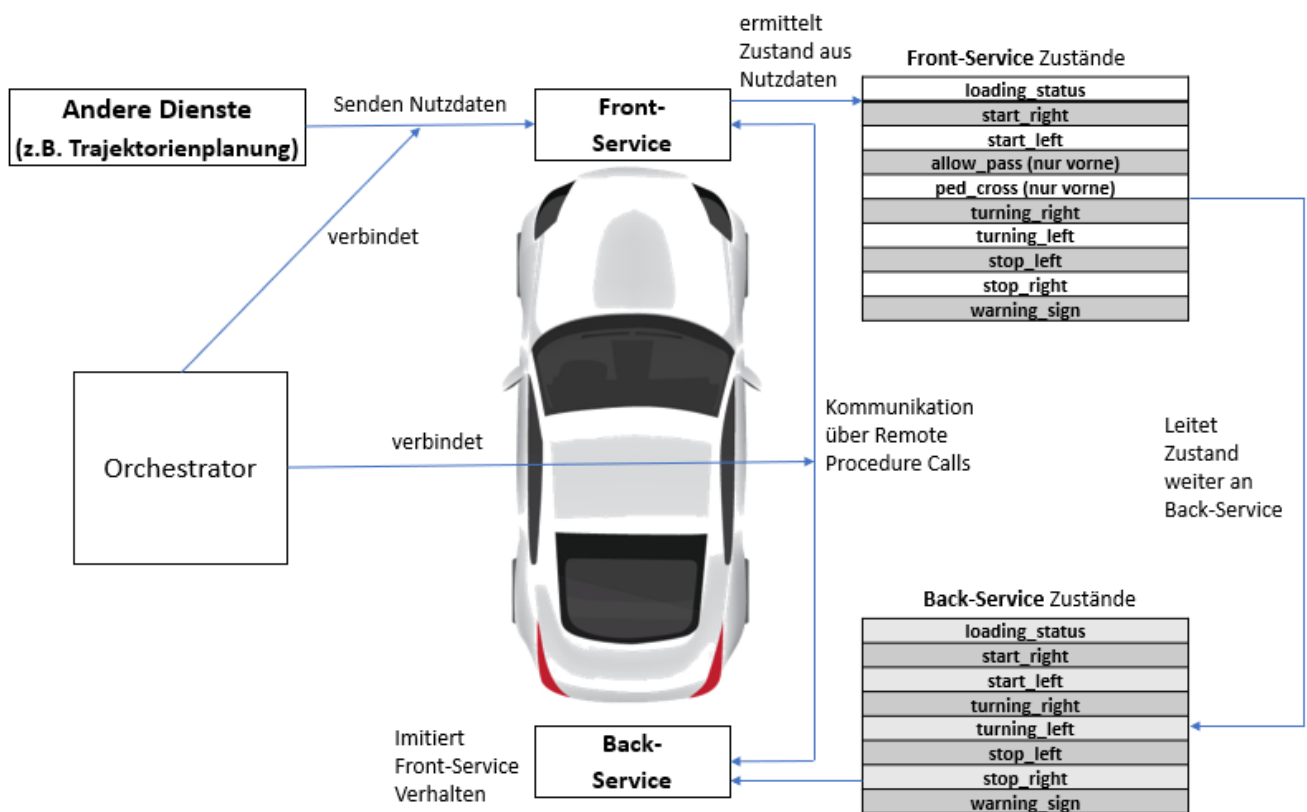


Abbildung 4-4: Vernetzung von Front- und Back-Service

Um die Zustände in den beiden Diensten zu implementieren muss auf grundlegende Elemente der ASOA Architektur zurückgegriffen werden. Wie in Kapitel 2.3.1 beschrieben handelt es sich dabei um Elemente wie Tasks, Guarantees, Requirements oder auch Remote Procedure Calls (RPCs). Welche Elemente für die einzelnen Dienste verwendet wird, wird im entsprechenden Unterkapitel Kap. 4.2.5 behandelt.

Weiterhin ist die Aufteilung thematisch ähnlicher Funktionen in verschiedene Dateien sinnvoll. So liegt es nahe alle LED-Funktionen wie bspw. das Anzeigen von Bildern oder Animationen in einer eigenen Header-Datei (RGB\_Matrix\_Controller.hpp oder kurz:

rgbController) zu speichern. Diese Header-Datei kann man dann im Front- und Back-Service einbinden. Sollten in Zukunft weitere Dienste hinzukommen, so können diese ebenfalls die Header-Datei inkludieren und dessen Funktionen nutzen. Die Dienste enthalten dann lediglich die Logik der Zustände und deren Übergänge, d.h. wenn bspw. der Front-Service den Zustand „Fußgänger am Zebrastreifen“ als aktiv sieht, dann wird über eine Funktion aus der Header-Datei (rgbController) die Bildanzeige ermöglicht – die Funktion selbst befindet sich also nicht in der Datei für den Front-Service. Dadurch wird eine Modularisierung des Programms erreicht und die Wiederverwendbarkeit der LED-Funktionalität wird wesentlich erleichtert, es genügt das Inkludieren des rgbControllers um deren Funktionen zu nutzen und die Anzeige in der Front und im Heck sind in einzelnen Dateien implementiert. Genaues zum rgbController wird in Kapitel 4.2.4 beschrieben. Es folgen nun weitere Unterkapitel, die sich näher mit der Umsetzung im Programm beschäftigen und auf den hier genannten Vorüberlegungen basieren.

#### **4.2.2 Einbinden der LED-Softwarebibliothek in die ASOA-Dienststruktur**

ASOA Dienste werden nicht manuell kompiliert, sondern mit Hilfe des Build-Tools cmake, um die Reproduzierbarkeit und die Automatisierung zu gewährleisten (Vgl. Kapitel 4.1.4). Dazu muss eine cmake-Textdatei (CMakeLists.txt) existieren, die Anweisungen in der cmake-Sprache enthält. Das UNICARagil Projekt bietet eine Vorlage-Textdatei, die für einen allgemeinen ASOA Dienst ohne zusätzliche Bibliotheken angelegt wurde.

In diese Vorlage sollen Befehle ergänzt werden, die die hzeller-Bibliothek im Kompilierungsvorgang, d.h. bei der Erstellung der resultierenden Makefiles, berücksichtigt. Dabei muss der Bibliothekspfad angegeben werden und für jede zu kompilierende C++ Datei (.cpp-Datei, diese werden in cmake „target“ genannt) eben diese Bibliothek verbunden werden. In diesem Fall handelt es sich bei dem Bibliothekspfad um den /lib Ordner, der Quellcode für grundlegende Funktionen für die Steuerung der LED-Matrix enthält.

An dieser Stelle ist der Image-Viewer noch nicht berücksichtigt. Der Image-Viewer verwendet nämlich die sog. Magick++ Bibliothek, eine Open-Source Bildverarbeitungsbibliothek (*Magick++ API for GraphicsMagick*, 2021c). Um auch diese Bibliothek mit Hilfe von cmake korrekt einzubinden müssen in der cmake-Textdatei zusätzliche Compiler- und Linker-Flags angegeben werden. Sind die Flags korrekt angegeben, so wird dann auch der Image-Viewer, der im Dienst-Quellcode benutzt wird, berücksichtigt bzw. kompiliert. Die cmake-Dokumentation befindet sich auf der Website des Entwicklers (*CMake Reference Documentation — CMake 3.21.3 Documentation*, 2021a), eine Benutzeran-

leitung für die Magick++ Bibliothek existiert ebenfalls auf der entsprechenden Entwickler-Website von GraphicsMagick (2021c). Die finale cmake-Textdatei befindet sich im Anhang in Kapitel C.

#### 4.2.3 IDL-Files mit Architekturtool

Es ist zu beachten, dass für die Kommunikation zwischen ASOA Diensten im Vorfeld auszutauschende Daten definiert sein müssen, da der Orchestrator diese Informationen benötigt. Ferner müssen die auszutauschenden Daten im Vorfeld durch eine Orchestrator-Textdatei definiert werden. Um diese Daten zu definieren, werden IDL-Files (oder auch: Interfaces) benötigt. Der nachfolgende Absatz orientiert sich an der projekt-internen Anleitung für die Erstellung eines ASOA Dienstes (UNICARagil, 2021).

Ein Interface besitzt im Allgemeinen drei Strukturen: Data, Quality und Parameter. Die Data Struktur enthält die Nutzdaten, die von einer Garantie dieses Types gesendet werden oder einem verbundenen Requirement empfangen werden. Dabei dienen Garanties und Requirements der Kommunikation zwischen Diensten. Requirements sind Anforderungen nach Daten, die ein Dienst braucht um seine Funktionen korrekt auszuführen und Garanties stellen anderen Diensten Daten zur Verfügung. Um also Daten zwischen Diensten auszutauschen ist die Definition der Nutzdaten im Vorfeld zwingend notwendig. Die Quality Struktur definiert den Datentyp, der die Qualität der gesendeten Nutzdaten wiedergibt. Die Parameter Struktur erlaubt einem Requirement zu entscheiden, ob eine Garantie verwendbare Informationen bereitstellt. Alle drei Strukturen sind ähnlich aufgebaut, sie bestehen aus einer Definition der Struktur und beinhalten dazu Funktionen, welche die Daten, die in der Struktur gespeichert sind entweder in einen Puffer serialisieren oder aus einem Puffer in die Struktur deserialisieren.

Diese Funktionen müssen hier jedoch nicht selbst geschrieben werden, die Erstellung eines Interface mitsamt dieser Funktionen erfolgt über das in Kapitel 2.3.1 bereits vorgestellte ASOA Architekturtool (*UNICARagil Architektur Tool*, 2021f). Abbildung 4-5 zeigt eine Interface-Definition im Architekturtool anhand des Beispiels „Fußgänger an Zebrastreifen“. In der Nutzdaten Typdefinition werden die Daten, die zwischen Diensten ausgetauscht werden sollen, in einem struct definiert. Welche Daten für welchen Zustand ausgetauscht werden sollen werden dem UML-Zustandsdiagramm entnommen. Diese ergeben sich aus den Bedingungen für die Transitionen (Guards).

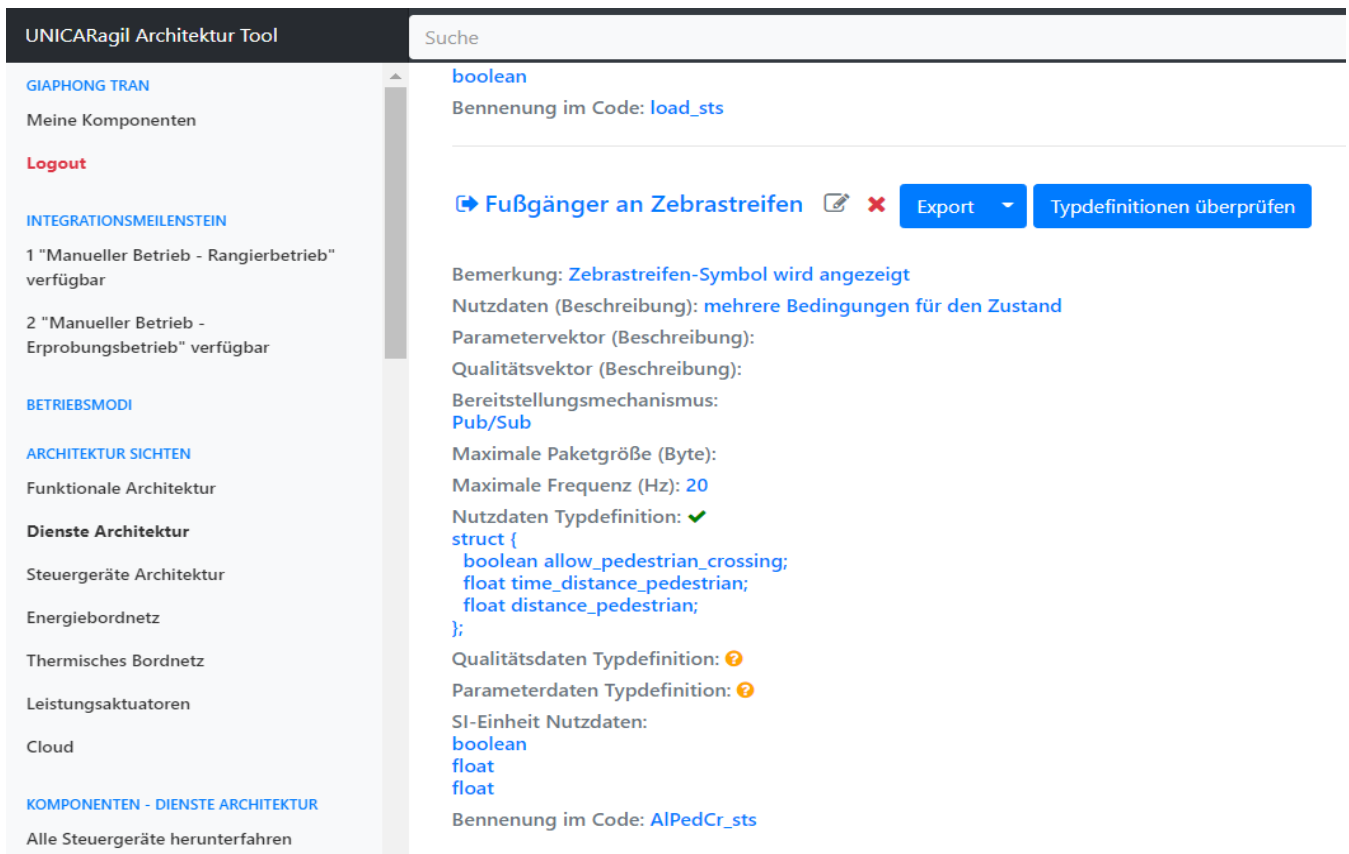


Abbildung 4-5: UNICARagil Architektur Tool

#### 4.2.4 rgbController: LED Image Viewer

In den Vorüberlegungen aus Kapitel 4.2.1 wurde festgelegt, dass die Funktionalität der Bildanzeige in einer separaten Header-Datei (rgbController) implementiert bzw. gekapselt wird. Motivation dafür ist das Erreichen einer Modularisierung und die Erhöhung der Wiederverwendbarkeit durch simples Inkludieren der Header-Datei in andere C++ Dateien.

Die Hauptfunktion des rgbControllers ist die Anzeige beliebiger Bilder und Animationen mittels Angabe eines relativen Dateipfades, ausgehend vom Ordner in dem sich der rgbController befindet. Zulässige Dateiformate sind u.a. .png-Dateien, .jpg-Dateien, .gif-Dateien und auch .stream-Dateien. Stream-Dateien (kurz: Streams) sind spezielle Dateien, bei denen die Ladedauer eines Bildes bzw. einer Animation im Gegensatz zu anderen Dateiformaten deutlich kürzer ist. Nach Zeller (2021) kann das Laden von Bild- oder Animationsdateien viel Zeit in Anspruch nehmen, so können diese vorher vorverarbeitet werden und in eine Stream-Datei geschrieben werden, welches sehr schnell geladen werden kann. Streams können mit dem Image-Viewer aus dem util-Ordner der hzeller-Bibliothek (das Erstellen von Streams ist auch mit dem rgbController möglich) werden. Die oben genannten Formate können als Input in den Image-Viewer gegeben

werden und mit dem Terminal-Flag „-O bildname.stream“ zu einem Stream verarbeitet und gespeichert werden. Zudem kann auch die Ladedauer berechnet werden, die Berechnungsvorschrift ist im Image-Viewer enthalten. Front- und Back-Service nutzen diese Streams. Eine Analyse der Ladedauer und die dafür in Kauf genommene Speicherbelastung folgt ausführlich in der Diskussion in Kapitel 6.

Der `rgbController` verfolgt einen objektorientierten Ansatz und besitzt die „`RGBMatrixController`“-Klasse. Zweck dieses Ansatzes ist, dass im Front- und Back-Service ein Objekt bzw. Instanz des Typs `RGBMatrixController` erzeugt wird, das die LED-Funktionen beinhaltet. Instanzfunktionen und Attribute dieses Objekts können dann aufgerufen werden um auf eine kompakte und übersichtliche Weise die Zustände des UML-Zustandsdiagramms zu implementieren. In den beiden Diensten wird dann lediglich der Dateipfad zu den anzuzeigenden Streams übergeben und der Zustand übergeben, der angibt ob die LED-Matrix aktiv sein soll oder nicht (Aktivitätsstatus). Die Anzeige des Streams, also letztendlich die anzuzeigenden Inhalte, wird dann vom `rgbController` übernommen. Dadurch erhält man zudem eine Trennung der Funktionen: die Dienste stellen die Logik der Zustandstransitionen bereit und geben an, welcher Zustand gerade aktiv ist und was die LED-Matrix anzeigen soll; der `rgbController` erhält diese Information und zeigt die entsprechenden Inhalte an. Folgende Abbildung zeigt das UML-Klassendiagramm der Klasse „`RGBMatrixController`“.

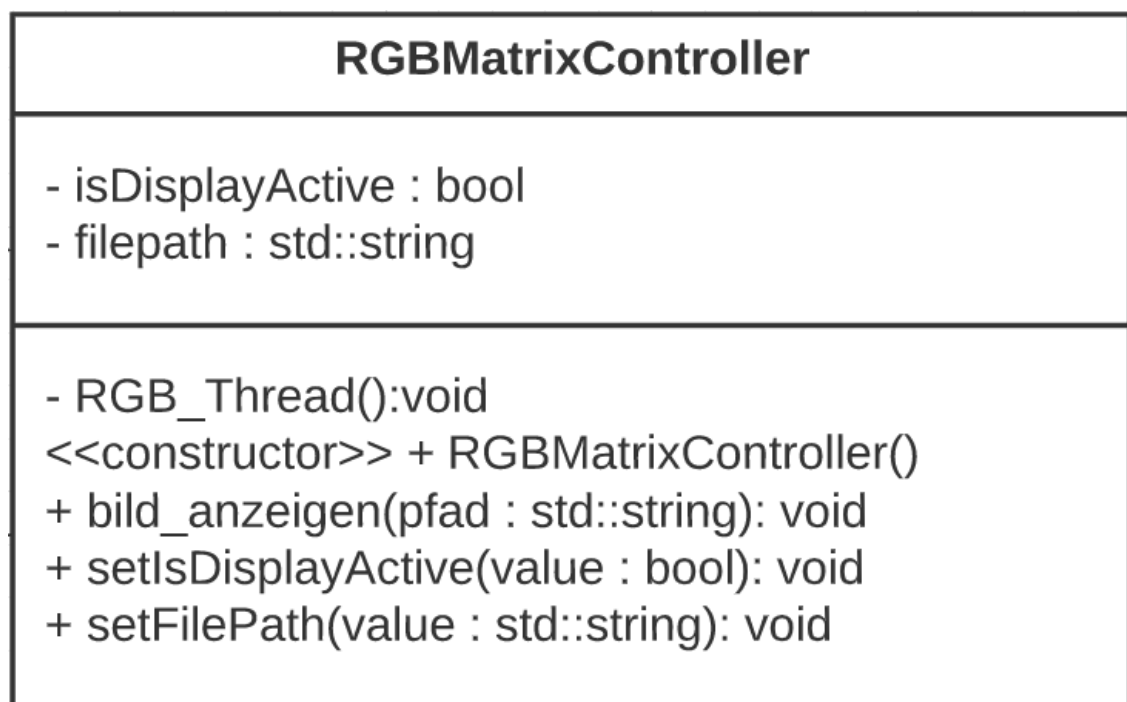


Abbildung 4-6: UML-Klassendiagramm `RGBMatrixController`

Die „RGBMatrixController“-Klasse besteht aus folgenden Elementen:

- Zwei Attribute (Klassenvariablen)
- Ein Thread mit Dauerschleife
- Ein Konstruktor
- Die „bild\_anzeigen“-Funktion
- Zwei Setter-Funktionen für die Klassenvariablen

Die beiden Attribute sind genau diejenigen, die von den Diensten festgelegt wird: der Dateipfad und der Aktivitätsstatus. Diese werden in den Diensten über zwei Setter-Funktionen verändert, d.h. wenn der bspw. Dienst in den „Fußgänger an Zebrastreifen“-Zustand wechselt, dann werden über die Setter-Funktionen des Objekts die entsprechenden Werte (Stream-Datei des Fußgänger-Bildes und Aktivitätsstatus auf „True“) gesetzt.

Die Anzeige auf der LED-Matrix wird durch die „bild\_anzeigen“-Funktion realisiert. Diese basiert zum Großteil auf dem Image-Viewer der hzeller-Bibliothek. Folgende Abbildung zeigt einen Programmablaufplan der Funktion bild\_anzeigen.

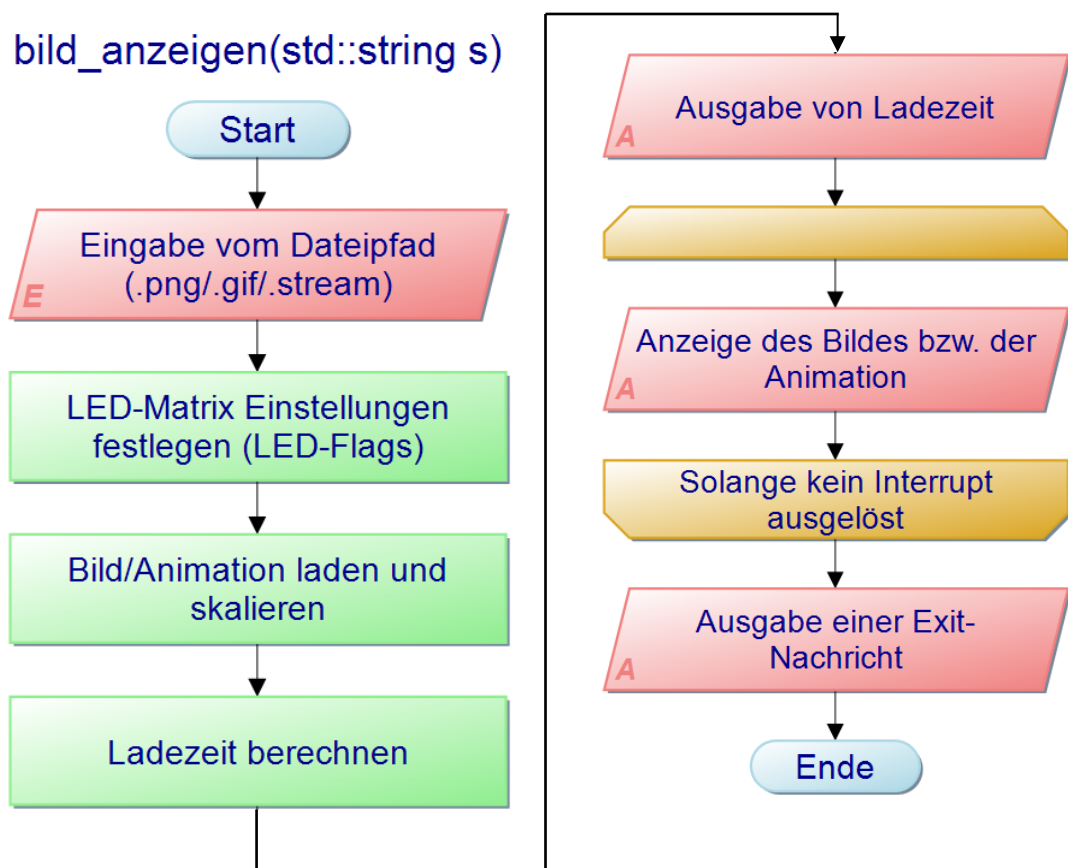


Abbildung 4-7: Programmablaufplan bild\_anzeigen

Bei jedem Aufruf der Funktion wird ein Objekt vom Typ `RGBMatrix` instanziiert. Dieser Objekttyp basiert auf einer bibliotheksinternen Klasse, die mit Hilfe von vorher definierten Anzeige-Einstellungen die LED-Anzeige vorbereitet. Das bedeutet, dass zum einen diese Einstellungen vorher im Quellcode definiert werden (siehe dazu Kapitel 4.1.5) müssen und zum anderen, dass die `Magick++` Bibliothek bei jedem Aufruf initialisiert werden muss. Der Image-Viewer basiert auf `Magick++` und kann ohne die Initialisierung keine Inhalte anzeigen.

Für die Anzeigedauer auf der LED-Matrix gibt es zwei Möglichkeiten. Entweder wird eine Dauer im Programm definiert oder die Anzeige läuft so lange bis eine Unterbrechung eintritt. Bei den möglichen Zuständen des Fahrzeuges kann die benötigte Anzeigedauer im Voraus nicht sinnvoll bestimmt werden, da die Zustände von der aktuellen Verkehrssituation abhängen und oftmals nicht-deterministisch sind. Z.B. ist es dem Fahrzeug nicht möglich exakt vorherzusagen wie lange ein Fahrzeug aus dem Gegenverkehr benötigt um eine Engstelle zu passieren, da das Fahrverhalten des Verkehrsteilnehmers i.d.R. nicht vorhersehbar ist. Die Anzeigedauer muss demnach dynamisch bestimmt werden und wird in den Diensten nicht statisch im Voraus festgelegt. Daher wird die Anzeige in der `bild_anzeigen` Funktionen so lange bewerkstelligt, bis eine Unterbrechung eintritt. Die Unterbrechung erfolgt dann wenn sich der Zustand des Fahrzeuges ändert, also wenn eine Transition durchlaufen wird. Sobald ein Zustandsübergang statt findet muss also ein Unterbrechungssignal gesendet werden, sodass die Anzeige gestoppt wird bzw. sich ändert.

Hierfür wird auf die „`signal.h`“-Datei der C-Standardbibliothek zurückgegriffen. Klingebiel (2021) führt dazu Folgendes aus: „Die Definitionsdatei `<signal.h>` stellt Hilfsmittel zur Behandlung von Ausnahmebedingungen zur Verfügung, die während der Ausführung eines Programms eintreten können, wie zum Beispiel ein „Interrupt-Signal“ von einer externen Quelle oder ein Fehler während der Ausführung.“ Der `rgbController` beinhaltet einen sog. `InterruptHandler`, dieser behandelt auftretende Signale während der Programmausführung. Für die Implementierung der Zustände des Fahrzeuges bedeutet dies, dass die `bild_anzeigen` Funktion des `rgbControllers` so lange die geforderten Inhalte auf der LED-Matrix anzeigt, bis durch einen Zustandsübergang eine Unterbrechung der aktuellen Anzeige ausgelöst wird, was im Quellcode mittels Interrupt-Signale realisiert wird. Ein Interrupt-Signal wird dann ausgelöst, wenn ein Zustandsübergang stattfindet, z.B. wenn der Zebrastreifen-Zustand in den „Fahrt auf Fahrspurmitte“-Zustand übergeht.



An dieser Stelle können die eHMI-Dienste Dateipfade übergeben, die Aktivitätsstatus setzen und die Anzeige – je nach aktuellem Fahrzeugzustand – unterbrechen bzw. stoppen. Jedoch muss der rgbController auch periodisch überprüfen welcher Aktivitätsstatus aktuell vorliegt und welcher Dateipfad vorgegeben wird, um im Fahrbetrieb immer die richtigen Inhalte zur vorgesehenen Zeit anzuzeigen. Dafür enthält der rgbController einen Thread mit Dauerschleife. Abbildung 4-8 zeigt einen Programmablaufplan dieses Threads.

#### Dauerschleife: RGB\_Thread()

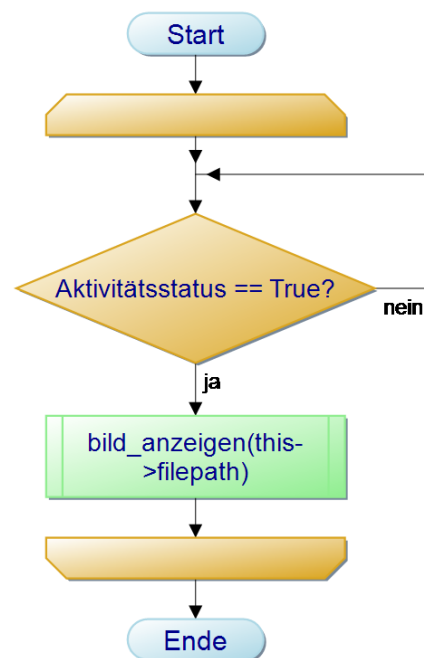


Abbildung 4-8: Programmablaufplan RGB\_Thread

Die Überprüfung des Dateipfades und des Aktivitätsstatus soll dauerhaft erfolgen, daher wird eine Dauerschleife eingesetzt. Damit der Dienst nicht in der Dauerschleife „festsitzt“ wird diese Dauerschleife in einem Thread ausgeführt. Threads sind leichtgewichtige Prozesse, wobei ein Prozess eine Abstraktion eines Programmes ist, das ausgeführt wird (Knoll & Lenz, 2020, S. 7–12).

Ferner erklärt Williams (2012, S. 5) das Konzept der Threads folgendermaßen: Jedes C++ Programm besitzt mindestens einen Thread, die durch die C++ Laufzeit gestartet wird, nämlich der Thread, der die Funktion main() ausführt. Das Programm kann dann weitere Threads starten, die andere Funktionen als Eintrittspunkte haben. Diese Threads laufen dann parallel zueinander mit dem initialen Thread. Threads beenden, wenn die angegebene Eintrittsfunktion zurückkehrt bzw. seine Ausführung beendet hat.

Um einen Thread zu starten wird zunächst ein Objekt des Typs `std::thread` erstellt, das eine vorher definierte Funktion als Parameter erfordert. Diese Funktion wird dann innerhalb dieses Threads ausgeführt. Die Header-Datei `<thread>`, das Teil der Standard-Bibliothek in C++ ist, muss dabei vorher im Programm inkludiert werden.

Zuletzt besitzt der `rgbController` einen Konstruktor für die Klasse `RGBMatrixController`. Dieser ist so implementiert, dass beim Instanzieren die beiden Attribute mit je einem Startwert initialisiert wird. Der Aktivitätsstatus wird auf „inaktiv“ (`false`) gesetzt und ein leerer Dateipfad wird gesetzt. Diese werden dann verändert, sobald neue Fahrzeug-Zustände eintreten. Zusätzlich wird im Konstruktor ein Thread erstellt, der die Funktion zur ständigen Abfrage (Dauerschleife) des Zustandes ausführt.

Durch einfaches Inkludieren dieser `rgbController` Header-Datei und durch Instanziierung eines Objekts diesen Typs können nun in den Diensten auf die LED-Anzeigefunktionen dieses Objekts zurückgegriffen werden. Das nachfolgende Unterkapitel befasst sich mit der Implementierung des Front-Service und des Back-Service.

#### **4.2.5 Front-Service und Back-Service**

Der Front-Service erhält Nutzdaten aus anderen Diensten und ermittelt daraus den aktuellen Zustand des Fahrzeuges. Wie erwähnt sind alle möglichen Zustände mitsamt Transitionen im vorgegebenen UML-Zustandsdiagramm enthalten. Abbildung 4-9 am Ende dieses Kapitels zeigt einen Programmablaufplan des Front-Service.

Die Funktionsweise des Front-Service soll nun am Beispiel des Zustandes „Fußgänger an Zebrastreifen“ erläutert werden. Aus einem anderen Dienst, z.B. dem Dienst für das Fahrzeugumfeldmodell, werden nun folgende Informationen gesendet:

- „Fußgängerüberweg wurde erkannt“
- „Zeit-Distanz zum Fußgänger beträgt weniger als fünf Sekunden“

Diese Informationen werden mit Hilfe der in den IDL-Files festgelegten Datentypen gesendet, die textliche Beschreibung oben dient nur zur Veranschaulichung. Der Front-Service erhält diese Nutzdaten und erkennt, dass der Zustand „Fußgänger an Zebrastreifen“ aktiv ist. Daraufhin werden die Attribute der `rgbController`-Instanz gesetzt:

- Der Dateipfad wird gesetzt: „ZebrastreifenStream.stream“ (Stream der Bilddatei)
- Der Aktivitätsstatus wird auf „aktiv“ (`True`) gesetzt

Der nebenläufige Thread des `rgbControllers` erkennt diese Änderung und führt die Funktion `bild_anzeigen` aus. Die Funktion zeigt dann die entsprechenden Inhalte auf der LED-Matrix an bis ein anderer Zustand durch den Front-Service erkannt wurde. Dann

wird ein Interrupt ausgelöst, der die Anzeige beendet bzw. daraufhin andere Inhalte anzeigt. Der genaue Ablauf wurde bereits im vorherigen Unterkapitel 4.2.4 beschrieben.

Da der Front-Service Nutzdaten empfangen soll, werden Requirements benötigt. Ein Requirement ist eine Anforderung nach Daten, die ein Dienst braucht um seine Funktionen korrekt auszuführen (UNICARagil, S. 5). Die empfangenen Daten kommen von anderen Diensten, die ihre Daten über Guarantees senden. Diese benötigt der Front-Service um festzustellen in welchem Zustand sich das Fahrzeug gerade befindet. Folgende Tabelle zeigt, welche IDL-Files für welches Requirement notwendig ist und um welchen Zustand es sich handelt. Diese Tabelle stellt einen Ausschnitt dar, die vollständige Tabelle befindet sich im Anhang in Kapitel A.

**Tabelle 4-1: IDL-Files und Nutzdaten für die Requirements des Front-Service (Ausschnitt)**

<b>Zustand</b>	<b>IDL-File für das benötigte Requirement</b>	<b>Nutzdaten</b>
Akku Laden Loading_status	Load_sts	bool currently_loading
Ausparken rechts (90°) Start_Right	LRPR_sts	bool leaving_parking_rectangular_right
Engstelle beidseitig Allow_pass	APass_sts	bool allow_passing float distance_passing_vehicle float distance_narrowing bool allow_pedestrian_crossing float time_distance_pedestrian
Fußgänger an Zebrastreifen Ped_cross	AlPedCr_sts	bool allow_pedestrian_crossing float time_distance_pedestrian float distance_pedestrian

Damit der Front-Service die Nutzdaten prüfen kann und dann einem entsprechenden Zustand zuweisen kann werden Conditional Tasks benötigt.

Laut Konzeptpapier (UNICARagil, S. 7) sind Tasks „Startpunkt für die Berechnungen in einem Dienst“. Conditional Tasks sind dabei Tasks, die ihre onWork()-Methode nur ausführen, wenn bestimmte Bedingungen gegeben sind. Einzige Bedingung ist das Vorhandensein von neuen Daten bei allen, für den Task registrierten, Requirements.

Wenn bspw. neue Nutzdaten für den Zustand „Fußgänger an Zebrastreifen“ vorliegen, so führt die entsprechende Conditional Task ihre onWork()-Methode aus. In dieser onWork()-Methode werden dann die Bedingungen geprüft, die eine Transition in den Zebrastreifen-Zustand ermöglichen. Sind die Bedingungen erfüllt, so werden Aktivitätsstatus und Dateipfad für den anzuzeigenden Inhalt gesetzt. Dieses Prinzip spiegelt sich in allen zehn Zuständen, für die die LED-Matrix in der Front aktiv sein soll, wider. Ein besonderer Zustand ist der Zustand „Sicheres Anhalten“ (Safe Stop bzw. Notfallmodus), da die LED ein Warnzeichen anzeigen soll, sobald dieser Zustand aktiv ist. Die Anzeige für diesen Zustand soll aktiv sein, unabhängig davon ob und welcher andere Zustand gerade aktiv ist. Damit erhält dieser Notfall-Zustand eine höhere Priorität als alle anderen Zustände. Dies wird im Programm insofern berücksichtigt, dass bei jedem neuen Eingang von Nutzdaten überprüft wird, ob der Notfall-Zustand aktiv ist oder nicht. Ist der Notfall-Zustand aktiv wird die aktuelle Anzeige unterbrochen und ein Warnzeichen wird angezeigt.

Die Nutzdaten von anderen Diensten empfängt der Front-Service periodisch, denn diese werden über sog. Periodic Tasks gesendet. „Diese Tasks werden einfach mit festgelegter Frequenz ausgeführt ab einem gewissen Startzeitpunkt“, laut Konzeptpapier (UNICARagil, S. 9). So empfängt der Front-Service Nutzdaten mit einer gewissen vorher festgelegten Frequenz und führt mit eben dieser Frequenz die onWork()-Methode, also die Überprüfung und Ermittlung des aktuellen Zustandes aus. So kann man Periodic Tasks, die sehr wichtig sind bzw. bei denen sich die Daten schnell ändern, eine hohe Frequenz zuweisen.

Implementiert man den Front-Service so wie bisher beschrieben, wird eine funktionsfähige Anzeige in der LED-Matrix der Fahrzeugfront ermöglicht, aber die LED-Matrix im Heck ist noch inaktiv. Wie aber bereits in den Vorüberlegungen aus Unterkapitel 4.2.1 hervorgeht, soll der Heck-Dienst (Back-Service) das Verhalten des Front-Service imitieren. Wenn die Front-Matrix eine Animation oder ein Bild anzeigt, soll die Heck-Matrix selbiges anzeigen außer bei den beiden Zuständen „Engstelle beidseitig“ und „Fußgänger an Zebrastreifen“. Damit die Heck-Matrix die Front-Matrix imitieren kann, müssen Front- und Back-Service kommunizieren, d.h. die aktuelle Anzeige und das Ende der Anzeige muss vom Front-Service übermittelt werden.

Damit diese Information vom Front-Service zum Back-Service gelangt, werden Remote Procedure Calls (kurz: RPC) verwendet. „Remote Procedure Calls ermöglichen Funktionsaufrufe über Rechnergrenzen hinweg. Folglich kann ein Dienst eine Funktion bei einem anderen Dienst aufrufen und die Ergebnisse dieses Funktionsaufrufs erhalten“ (UNICARagil, S. 6). Hier bedeutet das, dass der Front-Service für die acht Zustände der

Heck-Matrix genau acht RPC-Requirements besitzt und der Back-Service acht RPC-Guarantees besitzt. Ein RPC-Requirement kann mit der call()-Funktion aufgerufen werden, dann führt der verbundene Dienst die Programmzeilen seiner RPC-Guarantee auf. Dabei funktionieren die acht RPC-Guarantees des Back-Service nach dem gleichen Prinzip: wird ein RPC über den Front-Service aufgerufen, so wird über die rgbController-Instanz des Back-Service der Aktivitätsstatus und der Dateipfad gesetzt. Das Prinzip der LED-Anzeige über den rgbController ist demnach genauso wie beim Front-Service.

Der Hauptunterschied zur Funktionsweise des Front-Service ist, dass die Anzeige auf der Heck-Matrix nicht nach Überprüfung der Nutzdaten eines anderen Dienstes (z.B. Fahrzeugumfeldmodell) statt findet, sondern nach Aufruf durch den Front-Service. Demnach erhält der Back-Service nicht periodisch mit festgelegter Frequenz Nutzdaten und muss durch seine Logik ermitteln welcher Zustand vorliegt, sondern zeigt das an, was durch den Front-Service vorgegeben wird. Dadurch erübrigt sich die Kommunikation zwischen Back-Service und anderen Diensten wie bspw. das Fahrzeugumfeldmodell. Die auszutauschende Datenmenge reduziert sich auf die Menge zwischen Front-Service und Back-Service und das Netzwerk entlastet sich dadurch.

Für alle Guarantees und Requirements gilt, dass sie durch eine Orchestrator-Textdatei verbunden werden muss, sonst ist der Austausch der Daten nicht möglich. Somit gibt es die Dateien „orchestrator.txt“ und „orchestrator\_front\_heck.txt“. Erstere verbindet den Front-Service mit einem sog. Dummy-Service, der im nachfolgenden Kapitel vorgestellt wird. Zweiteres verbindet Front- und Back-Service.

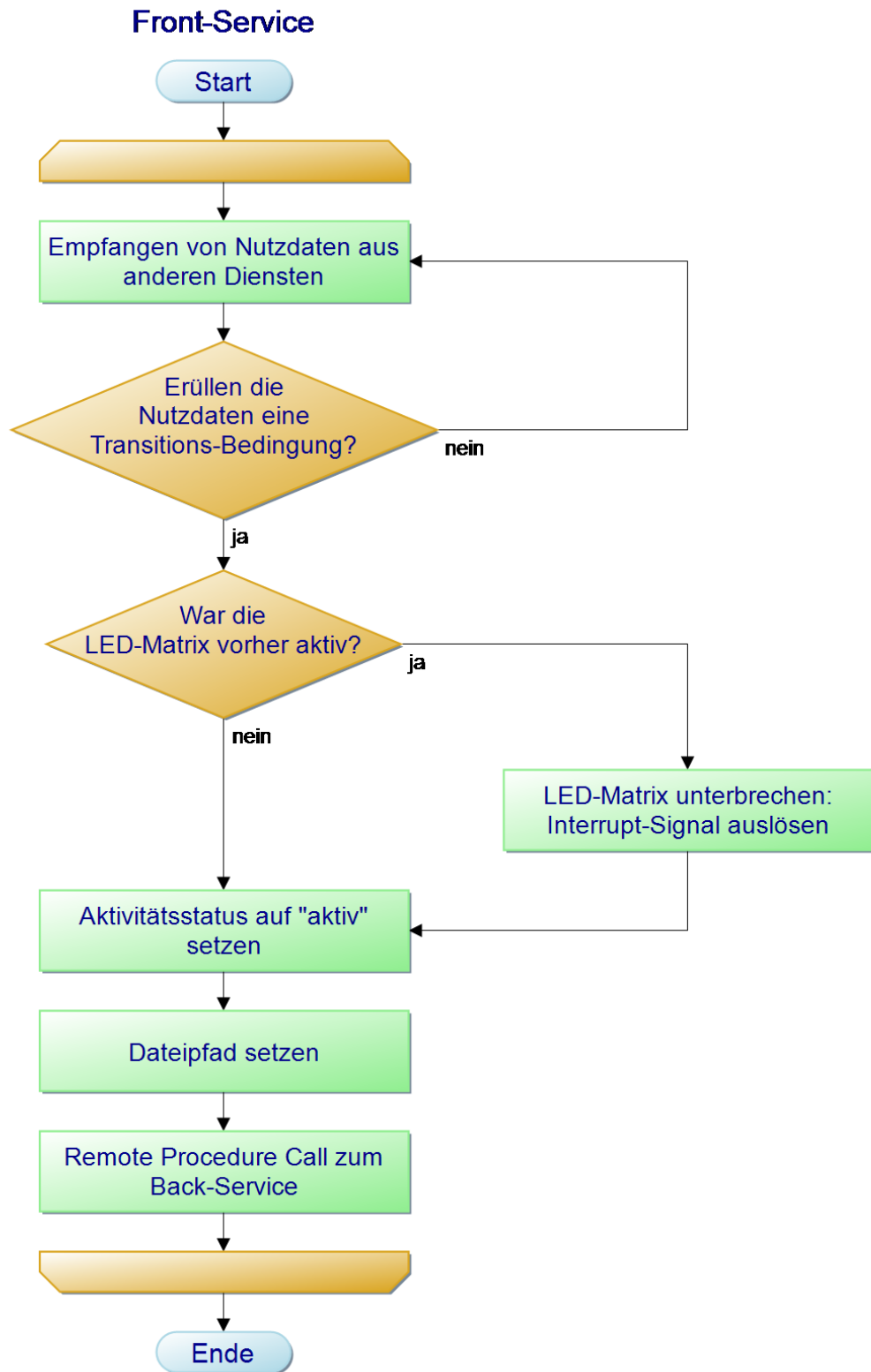


Abbildung 4-9: Programmablaufplan Front-Service

## 5 Softwaretest

In diesem Kapitel wird das Programm zur Ansteuerung der LED-Matrix getestet und auf ihre Funktionalität überprüft. Unterschiedliche Testkonzepte werden hierfür vorgestellt. Das finale Testkonzept beinhaltet einen Dummy-Service (Dummy-Dienst), der mit dem Front- und Back-Service kommuniziert. Dabei ist der Dummy-Service über eine Server-Client-Kommunikation verbunden mit einer grafischen Benutzeroberfläche (GUI), mit der man bestimmte Zustände des autonomen Fahrzeuges simulieren kann. Anhand dieser simulierten Zustände soll dann überprüft werden, ob sich die LED-Anzeige korrekt verhält, d.h. in den Zuständen jeweils die entsprechenden Inhalte anzeigt. Dieses Testkonzept wird in den folgenden Unterkapiteln vorgestellt, dabei werden zunächst die einzelnen Komponenten dieses Konzepts präsentiert und am Ende dann das gesamte Testkonzept. Die Ergebnisse des Tests werden im nachfolgenden Kapitel besprochen.

### 5.1 Dummy-Service

Der Dummy-Service soll diejenigen Dienste simulieren, die im realen Fahrbetrieb mit dem Front- bzw. Back-Service verbunden sind, z.B. der Dienst für die Trajektorienplanung oder das Fahrzeugumfeldmodell. Typische Verkehrssituationen wie bspw. das Einparken am Seitenstreifen oder das Anhalten vor einem Fußgängerüberweg sollen dabei simuliert werden. Dabei soll der Dummy-Service Nutzdaten über Periodic Tasks senden, sodass der Front- und Back-Service die dazu korrespondierenden Zustände interpretiert und die entsprechenden Inhalte auf den LED-Matrizen anzeigt.

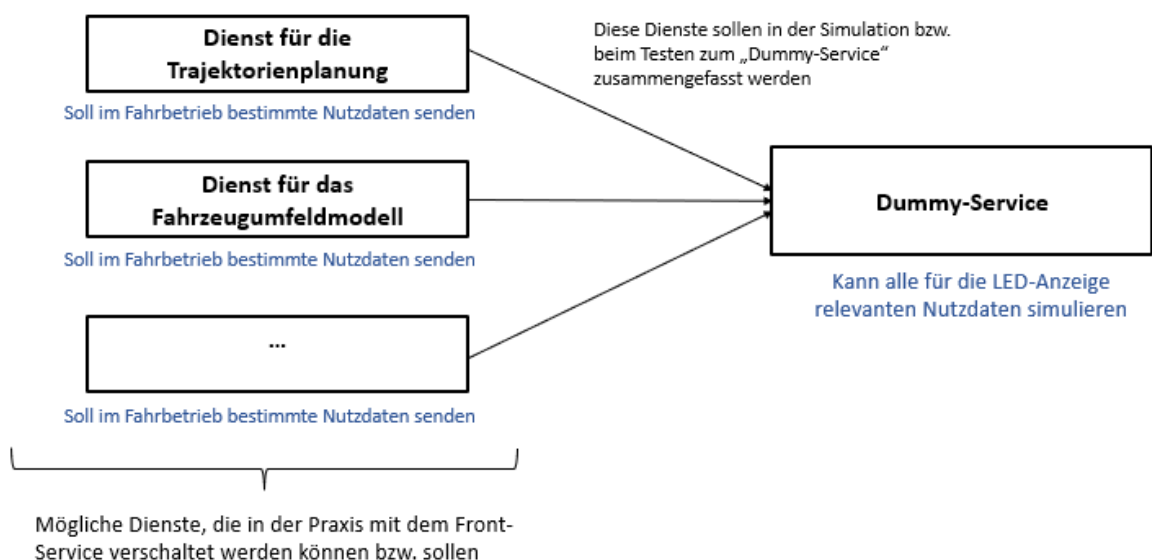


Abbildung 5-1: Vereinigung der Dienstekommunikation im Dummy-Service

Wie in Kapitel 4.2.5 beschrieben empfängt der Front-Service Nutzdaten und berechnet daraus den aktuellen Zustand, während der Back-Service das Verhalten des Front-Service imitiert. Da also nur der Front-Service Nutzdaten von anderen Diensten erhalten soll, genügt es, dass der Dummy-Service nur mit dem Front-Service kommuniziert.

Welche Daten der Dummy-Service sendet, soll dynamisch zur Laufzeit der Dienste simuliert werden. Dies bedeutet, dass die Simulation der Verkehrssituation vorher nicht festgelegt ist, d.h. die Nutzdaten werden nicht nach einem festen Schema (statisch) vom Dummy-Service gesendet. Eine statische Lösung könnte bspw. dadurch erzielt werden, dass eine nach Zeitplan festgelegte Reihenfolge von Verkehrssituationen durchlaufen wird. Dies wäre bspw. durch die Einführung einer Zählervariable und die Verknüpfung der Transitionen an diese Zählervariable realisierbar.

Durch den dynamischen Ansatz sind die Zustände zur Laufzeit frei wählbar, d.h. die Dauer eines simulierten Zustandes kann frei gewählt werden und Transitionen in andere Zustände können jeder Zeit erfolgen. Ein intuitives und benutzerfreundliches Mittel für die Umsetzung des dynamischen Ansatzes ist die Verwendung einer GUI (engl. graphical user interface). Mit Hilfe einer GUI kann bspw. per Knopfdruck der Zustand „Fahrzeug wird geladen“ gesetzt werden. Die GUI müsste dann die Periodic Tasks des Dummy-Service beeinflussen. Im Gegensatz zur statischen Lösung hat dieser Ansatz einen höheren Programmieraufwand, dafür lassen sich Front- und Back-Service jedoch flexibler und eleganter testen, der Aufwand wird dafür in Kauf genommen. Im folgenden Kapitel sollen zwei mögliche GUI-Konzepte vorgestellt werden.

## 5.2 Grafische Benutzeroberfläche

Aufgabe der GUI ist es die Nutzdaten, die vom Dummy-Service gesendet werden, zur Laufzeit der Dienste zu beeinflussen.

Die GUI soll dabei folgende Anforderungen erfüllen:

- Die Benutzung der GUI soll im Allgemeinen intuitiv und benutzerfreundlich sein
- Alle zehn Zustände (bzw. acht für die Heck-Matrix) aus dem UML-Zustandsdiagramm sollen simuliert werden können
- Welche Zustände gerade aktiv sind soll die GUI rückmelden bzw. visualisieren
- Einige Transitionen basieren auf dem Vergleich von Zahlenwerten, z.B. muss eine Fahrzeug-Distanz von 70 Metern zum Gegenverkehr unterschritten werden, damit der Zustand „Engstelle beidseitig“ eintritt. Die dafür relevanten Zahlenwerte sollen dann in der GUI angezeigt werden.



Es gibt viele Möglichkeiten eine solche GUI zu implementieren. In dieser Semesterarbeit sollen zwei Ansätze vorgestellt werden.

Eine Möglichkeit besteht darin die GUI direkt im Dummy-Dienst mit C++ zu schreiben. Das Unternehmen Qt Group stellt das Qt GUI Modul zur Verfügung, die Werkzeuge bereitstellt um auf einfache Weise in C++ eine GUI zu erstellen (*Qt GUI 5.15.6*, 2021e). Mit der Qt Creator Entwicklungsumgebung kann die GUI auf visuelle Weise erstellt werden (The Qt Company, 2021). Eine Schwierigkeit dabei ist das Einbinden in den Kompilierprozess zusammen mit der ASOA Architektur in cmake. Ist diese Schwierigkeit überwunden, so können in der GUI die zu sendenden Nutzdaten des Dummy-Service direkt beeinflusst werden.

Alternativ kann die GUI auch in einer anderen beliebigen Programmiersprache als eigenständiges Programm geschrieben werden und dann mit Hilfe eines Server-Client-Systems mit dem Dummy-Service verbunden werden.

Die Alternative ist zunächst aufwändiger, da zusätzlich die Kommunikation zwischen den beiden Programmen (GUI und Dummy-Service) ermöglicht werden muss. Der (Abkürzungsverzeichnis) LfE-Hilfswissenschaftler und Mitwirkende des UNICARagil Projekts Michael Sammereier hat jedoch bereits für seinen Dienst eine GUI in Java programmiert und dazu ein Server-Client-System aufgebaut (Sammereier, 2021). Da diese Programme bereits zum Entwicklungszeitpunkt des Front- und Back-Service existierten, können diese durch geringfügige Modifikation auch für den Dummy-Service dieser Semesterarbeit verwendet werden. Daher wird für die GUI dieses Dummy-Service die vorangegangene Arbeit von Michael Sammereier als Grundlage verwendet und so abgeändert, dass diese dann die oben genannten Anforderungen der GUI erfüllt. Dieser Ansatz stellt den kleinstmöglichen Programmieraufwand dar und wird in dieser Arbeit umgesetzt.

Die GUI von Sammereier wurde mit Hilfe der Java Swing Bibliothek erstellt (*javax.swing (Java Platform SE 8)*, 2021b). Diese besteht im wesentlichen aus Buttons, Grafiken und Funktionen für die Verbindung zum sog. Server Socket. Das verwendete Server-Client-System wird im nachfolgenden Unterkapitel ausführlich erläutert.

Die modifizierte Version der GUI, die für den Dummy-Service verwendet wird, wird in nachfolgender Abbildung veranschaulicht.

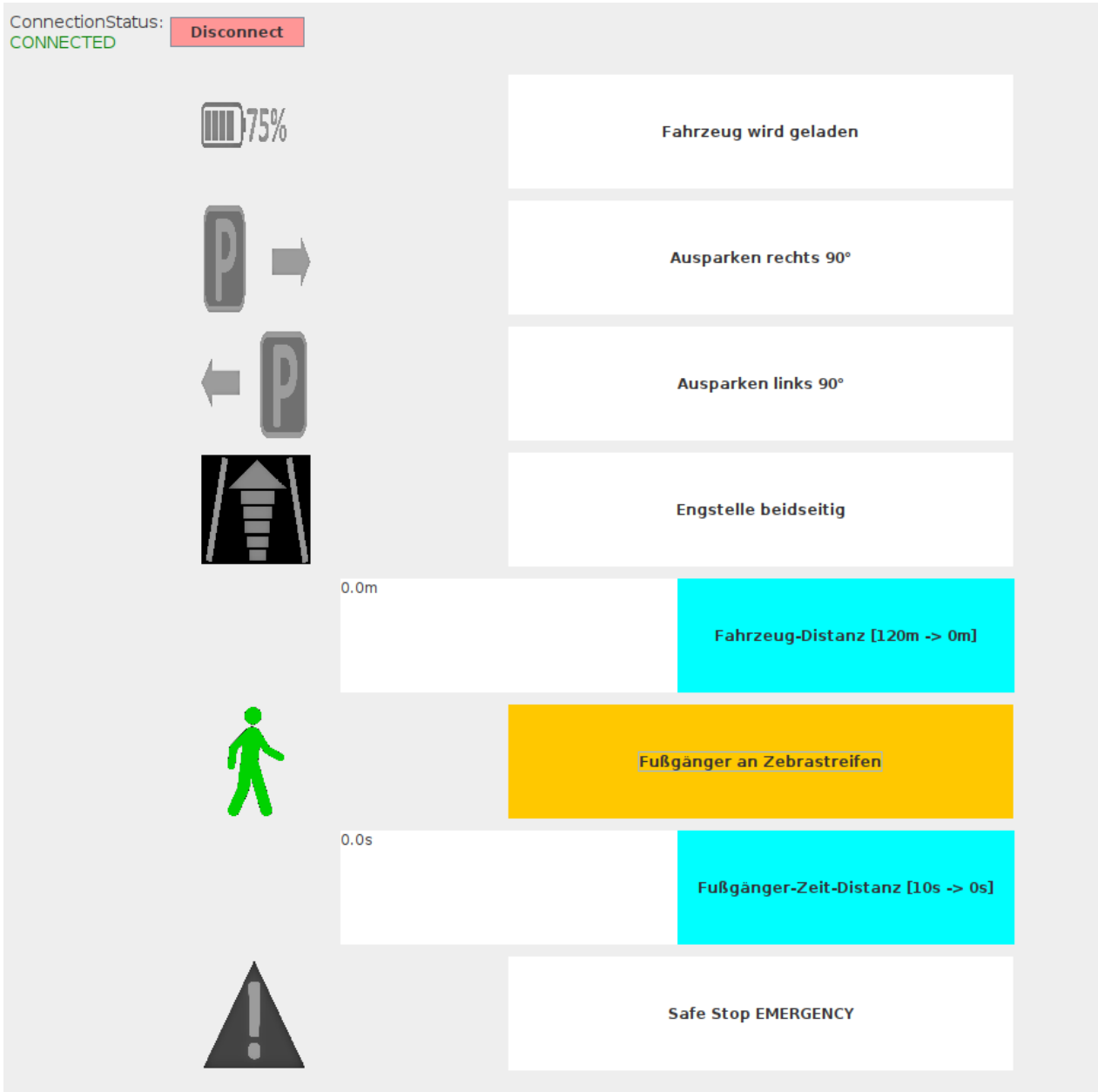


Abbildung 5-2: Ausschnitt modifizierte GUI

Nach Anklicken der Buttons wird die Buttonfläche farblich hervorgehoben um den aktuellen Zustand dieses Buttons („gedrückt“ / „nicht gedrückt“) anzuzeigen. Sollte ein Zustand aktiv sein, so wird ein dazu passendes Symbol in Farbe angezeigt. Ist der Zustand inaktiv, so wird dieses Symbol in schwarz-weiß angezeigt. Somit hat der Benutzer immer die Übersicht welche Zustände aktiv bzw. inaktiv sind. Bei den Zuständen „Engstelle beidseitig“, „Fußgänger an Zebrastreifen“ und „Wendemanöver auf Punkt links/rechts“ gibt es jeweils neben dem Button eine Textbox, die bestimmte Nutzdaten anzeigt. Z.B. wird für den Zustand „Engstelle beidseitig“ die Fahrzeug-Distanz zum Gegenverkehr in Metern ausgegeben. Damit kann die Bedingung überprüft werden, dass die Transition in diesen Zustand nur statt findet, wenn die Fahrzeug-Distanz einen bestimmten Wert unterschritten hat.

### 5.3 Server-Client-Kommunikation

Aus dem vorherigen Unterkapitel geht hervor, dass die GUI mit dem Dummy-Service verbunden sein muss, um die Funktionalität des Front- und Back-Service per Knopfdruck zu testen. Dazu wird ein Server-Client-System verwendet.

Thienen (1999, S. 5) definiert ein solches System folgendermaßen: „Ein System wird als Client/Server-System bezeichnet, wenn es aus mehreren Komponenten besteht, diese miteinander kommunizieren und eine Aufgabenverteilung zwischen ihnen stattfindet in der Form, daß eine Komponente (Client) Dienste von einer anderen Komponente (Server) anfordert, die diese Dienste erbringt“.

Hier stehen zwei Programme in einer Server-Client-Beziehung: die Java GUI in der Rolle des Clients und der Dummy-Dienst in der Rolle als Server. Dabei kommunizieren die Programme, indem sie von sog. Sockets lesen und auf diese schreiben. „Sockets stellen die Endpunkte einer Kommunikationsbeziehung zwischen Prozessen dar. Sie basieren auf einer IP-Socket-Adresse, bestehend aus einer IP-Adresse und einer Portnummer“, beschreibt Abts (2019, S. 20).

Folgender Absatz basiert größtenteils auf dem Beitrag von Kalita zur „Socket-Programmierung“:

Der Begriff Socket, auf Deutsch übersetzt „Steckdose“ bzw. „Buchse“, stammt aus der Telefonbuchsen-Metapher, bei der Steckdosen als Schnittstellen fungieren, die über ein Netzwerk miteinander verbunden werden (Kalita, S. 4802). Kalita (Kalita, S. 4804) zufolge ist ein Port ein Software-Konstrukt, das als Kommunikationsendpunkt im Host-Betriebssystem eines Computersystems dient. Dabei ist der Zweck von Ports die eindeutige Identifizierung verschiedener Anwendungen oder Prozesse, die auf einem einzigen Computer laufen. Beim Einrichten einer Server-Client-Kommunikation muss also ein Port gewählt werden, an dem sich Server und Client „treffen“. Es handelt sich dabei um einen logischen Anschluss, der durch eine 16bit-Ganzzahl (0 bis 65535) angegeben wird. Da einige Ports zwischen Nummer 0 und 1024 für bestimmte Zwecke (sog. „well-known services“) reserviert sind, benutzen entwicklerspezifische Prozesse i.d.R. Ports mit Nummern über 1024. Die nachfolgende Abbildung veranschaulicht beispielhaft die Kommunikationsverbindung zweier Programme mit Hilfe von Sockets.

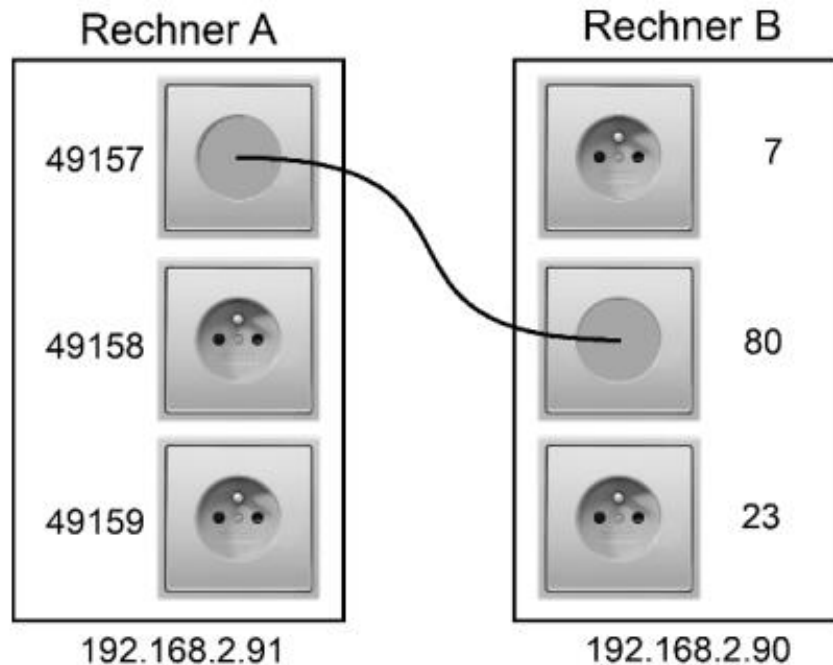


Abbildung 5-3: Prinzipbild: Beispielhafte Kommunikationsverbindung zweier Programme mit Hilfe von Sockets (Abts, 2019, S. 20)

Für diese GUI wird die IP-Adresse der Domäne „localhost“ genutzt, da sich die Programme auf demselben Rechner (lokal) befinden. „Der Hostname localhost identifiziert den eigenen Rechner und entspricht der IP-Adresse 127.0.0.1“, erklärt Abts (2019, S. 18). Die Portnummer wird mit 13337 gewählt, wobei dieser Port ein sog. „user port“ ist und das TCP-Protokoll (Transmission Control Protocol) nutzt. User ports sind Ports mit einer Nummer im Bereich 1024 bis 49151. Der verbleibende Bereich 49152 bis 65535 steht zur freien Verfügung (Abts, 2019, S. 18). Damit die GUI also mit dem Dummy-Service kommunizieren kann wird zur gegenseitigen Identifizierung die IP-Adresse der Domäne localhost genutzt, zusammen mit der Portnummer 13337 für die Datenübertragung mittels TCP-Protokoll. Auf die Funktionsweise des TCP-Protokolls soll hier nicht genauer eingegangen werden. Die IP-Adresse und die Portnummer können über Textfelder im oberen Bereich der GUI eingestellt werden (siehe Abbildung 5-2). Zudem kann oben links in der GUI die Verbindung zwischen den beiden Programmen hergestellt bzw. wieder unterbrochen werden.

## JAVA

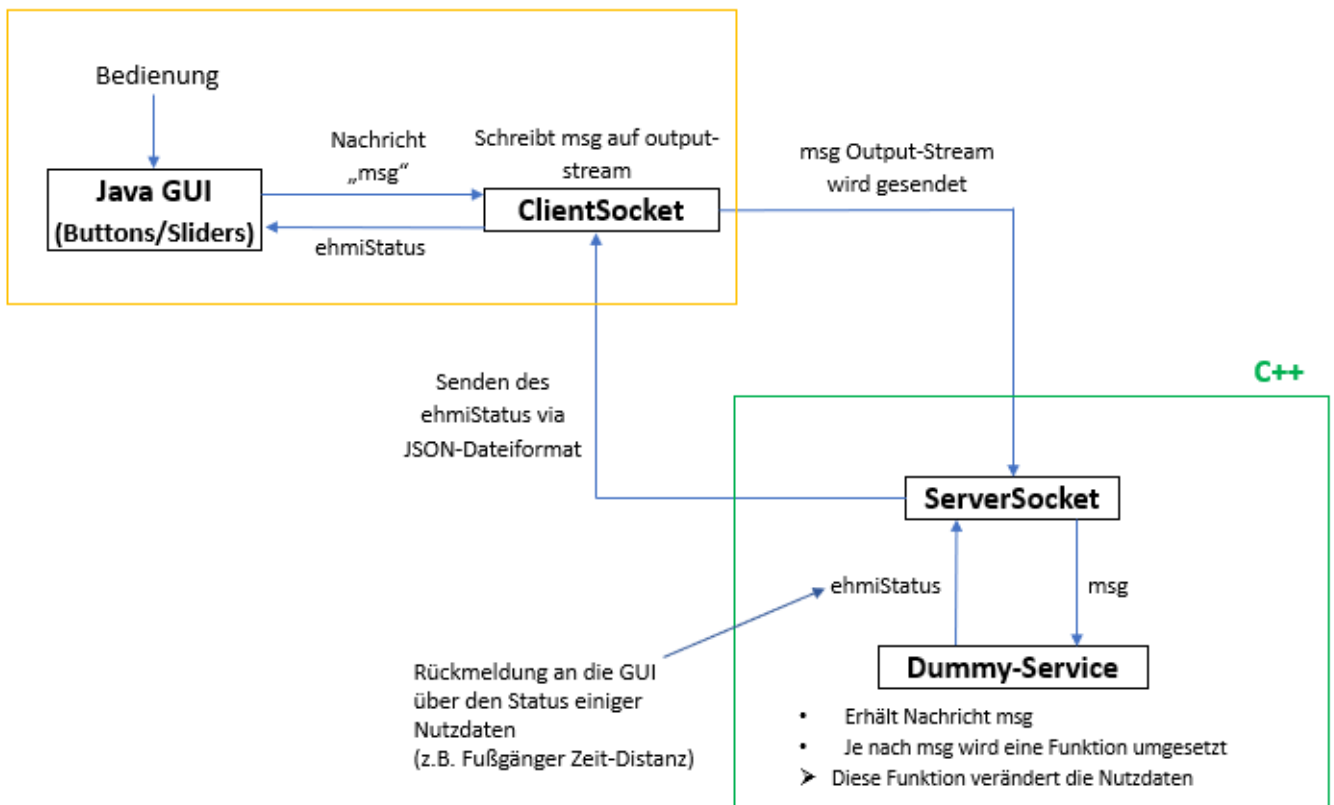


Abbildung 5-4: Prinzipbild Server-Client-Kommunikation

Obige Abbildung zeigt die prinzipielle Funktionsweise der Kommunikation über Server- und Client-Sockets. Klicks in der GUI lösen vordefinierte Methoden aus, die String-Nachrichten zum ServerSocket senden. Diese String-Nachrichten werden auch „Streams“ genannt. „Ein Stream ist eine Sequenz von Zeichen, die in oder aus einem Prozess fließen“, so Kalita (Kalita, S. 4805). Je nach einfließendem Stream wird eine Funktion umgesetzt. Fließt bspw. die Nachricht „ZEBRASTREIFEN\n“ ein, so setzt der Dummy-Service die korrespondierenden Nutzdaten auf einen bestimmten Wert. In diesem Beispiel wird dann der Wert der bool’schen Variable „allow\_pedestrian\_crossing“ invertiert, d.h. von „True“ auf „False“ gesetzt bzw. andersherum. Diese Variable wird zusammen mit anderen Nutzdaten über Periodic Tasks an den Front-Service gesendet, der daraufhin den entsprechenden Zustand interpretiert und die entsprechenden Inhalte auf der LED-Matrix anzeigt.

Die GUI soll jedoch nicht nur die Nutzdaten verändern, sondern auch den aktuellen Status der LED-Anzeige wiedergeben. Dazu wird eine Rückmeldung des Servers benötigt, die angibt wie der aktuelle Status ist. Dazu werden die Nutzdaten in der dafür vorgesehenen Klasse „RGBStatus“ gespeichert und dann an den Client Socket gesendet. Dieser Datenaustausch wird mit Hilfe von JSON (JavaScript Object Notation) realisiert. Abts (2019, S. 23) erklärt, dass JSON ein platzsparendes Datenaustauschformat ist, das für Menschen einfach zu lesen und zu schreiben ist und für das es viele Tools gibt, um

Datenstrukturen im JSON-Format zu analysieren bzw. zu generieren. Für die Übermittlung der Nutzdaten an den Client Socket eignet sich dieses Format besonders, da die Vielzahl der Nutzdaten praktischerweise als Name/Wert-Paare angegeben werden können. JSON-Dokumente beruhen im Allgemeinen auf einer ungeordneten Menge von Name/Wert-Paaren (Objekte) und einer geordneten Liste von Werten (Arrays) (Abts, 2019, S. 23). Somit lassen sich die Nutzdaten sehr übersichtlich mit Hilfe von JSON senden. Die JSON-Dokumente die beim Client Socket eingehen, können über Java-Funktionen einfach interpretiert werden und anschließend dazu genutzt werden, um bspw. den aktuellen Wert der Variable für die Fahrzeug-Distanz in der GUI anzuzeigen. Auf diese Art und Weise können per Knopfdruck in der GUI bestimmte Zustände ausgelöst werden, die zu einer Anzeige auf den LEDs führen.

## 5.4 Testen des Front- und Back-Service

Über die GUI lassen sich Anzeigen auf der LED-Matrix, die durch den Front-Service hervorgerufen werden, dynamisch zur Laufzeit testen. Da im UNICARagil Projekt jedoch nur ein Prototyp für die LED-Matrix vorhanden ist kann der Front- und Back-Service nicht gleichzeitig getestet werden. Daher wird ein Dienst mit der LED-Matrix getestet und zeitgleich wird der andere Dienst mit Hilfe von Textausgaben auf einer Konsole getestet.

Der Front-Service wird mit Hilfe des Prototyps der LED-Matrix getestet, der Back-Service wird mit Hilfe von Textausgaben auf der Konsole getestet. Zu erwarten ist, dass die Fahrzeugzustände gemäß UML-Zustandsdiagramm von den Diensten anhand der Nutzdaten erkannt wird. Nach Identifikation eines Zustandes soll ein entsprechendes Bild bzw. Animation auf der LED-Matrix angezeigt werden und zeitgleich soll durch ein print-Befehl eine Textausgabe durch den Back-Service statt finden. Die Textausgabe sollte die Information über den Aktivitätszustand der LED-Anzeige haben, also ob die Anzeige aktiv ist oder nicht. Darüber hinaus soll ein Dateipfad angegeben werden zum anzuzeigenden Inhalt. Anhand dieser Informationen erkennt man, ob der Back-Service die richtigen Inhalte anzeigt bzw. das gleiche anzeigt wie der Front-Service.



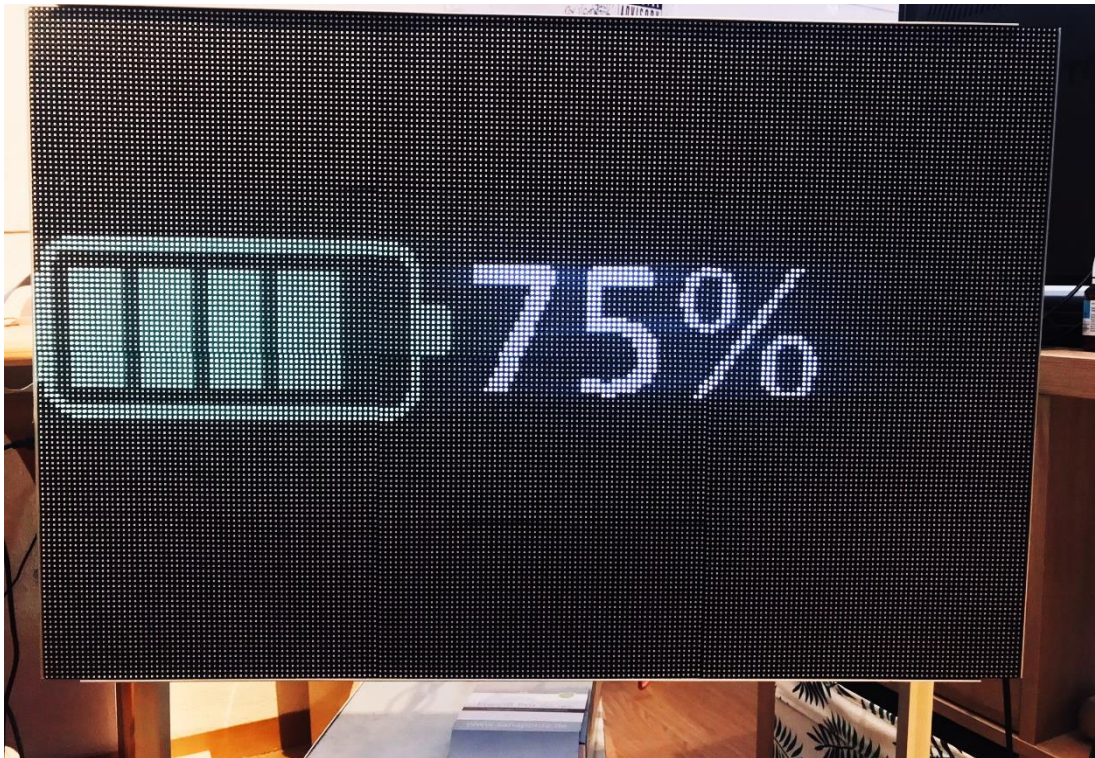


Abbildung 5-6: Bildausgabe auf dem Prototyp der LED-Matrix

Damit der Back-Service Text auf der Konsole ausgibt wird eine Dummy-Version des rgbControllers aus Kapitel 4.2.4 (RGB\_Matrix\_Controller\_Dummy.hpp) erstellt. Der Dummy-rgbController ist prinzipiell gleich aufgebaut, lediglich die „bild\_anzeigen“-Funktion ist unterschiedlich. Diese Funktion enthält statt der Bildausgabe-Funktionalität zwei print-Befehle. Die Ausgabe könnte folgendermaßen aussehen:

Konsolenausgabe  
Dummy-Service  
(GUI-Input Ausgabe)

```
phong@phong-VirtualBox: ~/Desktop/FINAL_eHMI_RGBMATRIX/RGB_eHMI_Dummy_Services/build
File Edit View Search Terminal Help
.....Received input from 127.0.0.1:45320: FAHRZEUG_WIRD_GELADEN
Received input from 127.0.0.1:45320: FAHRZEUG_WIRD_GELADEN
Received input from 127.0.0.1:45320: FAHRZEUG_WIRD_GELADEN
Received input from 127.0.0.1:45320: FAHRZEUG_WIRD_GELADEN
Received input from 127.0.0.1:45320: FAHRZEUG_WIRD_GELADEN
Received input from 127.0.0.1:45320: FAHRZEUG_WIRD_GELADEN
Received input from 127.0.0.1:45320: FAHRZEUG_WIRD_GELADEN
Received input from 127.0.0.1:45320: FAHRZEUG_WIRD_GELADEN
Received input from 127.0.0.1:45320: FAHRZEUG_WIRD_GELADEN
Received input from 127.0.0.1:45320: FAHRZEUG_WIRD_GELADEN
Received input from 127.0.0.1:45320: FAHRZEUG_WIRD_GELADEN
Received input from 127.0.0.1:45320: FAHRZEUG_WIRD_GELADEN
Received input from 127.0.0.1:45320: FAHRZEUG_WIRD_GELADEN
Received input from 127.0.0.1:45320: FAHRZEUG_WIRD_GELADEN
Received input from 127.0.0.1:45320: FAHRZEUG_WIRD_GELADEN
```

Konsolenausgabe  
Back-Service  
(Testmodus:  
keine LED-Ausgabe, nur  
Textausgabe)

```
phong@phong-VirtualBox: ~/Desktop/11_2_eHMI_Back_Service/build
File Edit View Search Terminal Help
Dummy_Thread erkennt isDisplayActive: 1
Showing animation:
../eHMI_symbols/eHMI_streams/AkkuLadenStream.stream
Dummy_Thread erkennt isDisplayActive: 1
Showing animation:
../eHMI_symbols/eHMI_streams/AkkuLadenStream.stream
Dummy_Thread erkennt isDisplayActive: 1
Showing animation:
../eHMI_symbols/eHMI_streams/AkkuLadenStream.stream
Dummy_Thread erkennt isDisplayActive: 1
Showing animation:
../eHMI_symbols/eHMI_streams/AkkuLadenStream.stream
```

Abbildung 5-5: Ausschnitt Konsolenausgabe Dummy-Service und Back-Service

## 5.5 Testmethodik und Durchführung

Der Softwaretest hat den Zweck, fehlerhaftes Verhalten der Dienste aufzudecken. Diese Fehler sollen nach und nach behoben werden, d.h. der Test mittels GUI wird mehrmals durchgeführt. Das Programm soll mehrere Testiterationen durchlaufen und bei jeder Iteration verbessert werden. Wesentlich ist dabei, dass die Funktionalität des UML-Zustandsdiagramms erfüllt wird. Der Front- und Back-Service soll demnach alle Zustände mitsamt der Transitionen enthalten, d.h. die LED-Ausgabe soll in jedem Zustand den Inhalt anzeigen, der im UML-Diagramm angegeben ist. Das Hauptbestreben besteht darin, auftretende Fehler so zu beheben, dass die Funktion erfüllt wird. Erst wenn die Funktion erfüllt wird, werden Maßnahmen getroffen, die die Performance des Programms verbessern.

Beim Testen werden alle Fahrzeug-Zustände geprüft, indem die Buttons der GUI geklickt werden und dann beobachtet wird was auf der LED-Matrix angezeigt wird und welcher Text auf der Konsole ausgegeben wird. Bei einem Klick sollte unmittelbar danach eine Transition stattfinden, d.h. nach kurzer Zeit sollte die LED-Ausgabe und die Textausgabe statt finden bzw. abgeschaltet werden.

Wie in Kapitel 5.2 genannt, basieren vier Zustände auf dem Vergleich von Zahlenwerten. Bei diesen Zuständen wird in der GUI der aktuelle Wert in einer Textbox angezeigt. Beim Testen sollte überprüft werden, ob die Transitionen in diese Zustände nur durchlaufen werden, wenn der Zahlenwert die Bedingung im UML-Diagramm erfüllt. Z.B. sollte die Animation für den Zustand „Wendemanöver auf Punkt rechts“ nur angezeigt werden, wenn der geplante Zeitpunkt für die Wende fünf Sekunden unterschreitet. Die entsprechende Variable wird in der dafür vorgesehenen Textbox in der GUI angezeigt. Dadurch kann überprüft werden ob die Anzeige sich korrekt verhält bei Erfüllen derjenigen Bedingungen, die den Vergleich von Zahlenwerten beinhalten.

Zu beachten ist zudem, dass der Zustand „Sicheres Anhalten“ als Notfall-Zustand eine höhere Priorität besitzt. Wenn der GUI-Button für diesen Zustand gedrückt ist, muss ein Warnzeichen auf der LED-Matrix ausgegeben werden bzw. die entsprechende Textausgabe auf der Konsole stattfinden – unabhängig davon in welchem Zustand sich das Fahrzeug vorher befand.

Es ist zu erwähnen, dass diese Testmethodik nicht die Funktionalität unter realen Bedingungen im Straßenverkehr prüft. Das Testen am Fahrzeug ist unerlässlich und sollte in zukünftigen Forschungsarbeiten vorgenommen werden.



## 6 Testergebnisse und Diskussion

Dieses Kapitel befasst sich mit Erkenntnissen, die sich aus dem Softwaretest nach der Methodik aus Kapitel 5 ergaben. Es soll diskutiert werden, ob und wie gut die implementierten Dienste die geforderte Funktionalität aus dem UML-Zustandsdiagramm erfüllen. Zudem soll auf die Effizienz beim Laden der LED-Inhalte eingegangen werden, dabei wird auf die im Unterkapitel 4.2.4 erwähnten Streams näher eingegangen. Im Unterkapitel 6.3 sollen weitere Konzepte und Ideen betrachtet werden, die das Programm erweitern und ggf. verbessern würden.

### 6.1 Ergebnisse des Softwaretests

Beim anfänglichen Testen wurde eine Vielzahl verschiedener Fehler im Programmcode deutlich. Zwei funktionsrelevante Fehler wurden in der anfänglichen Testphase bei folgenden Aktionen festgestellt:

- **Drücken eines Buttons in der GUI**

Erwartung: Zu erwarten ist, dass sich dadurch Nutzdaten, die vom Dummy-Service gesendet werden, verändern und es eine entsprechende Rückmeldung in der GUI gibt.

Fehler-Beobachtung: Es wurde beobachtet, dass sich Nutzdaten bei einigen Button-Klicks nicht verändert haben, auch gab es keine Rückmeldung in der GUI.

Auswirkung: Dadurch konnte der Front- bzw. Back-Service keine Transition durchlaufen und die LED-Matrix zeigt nicht die erwartete Ausgabe. Die erwartete Ausgabe ist der Inhalt, der laut UML-Zustandsdiagramm zu dem simulierten Zustand korrespondiert.

Ursache: Die String-Nachrichten, die von der GUI gesendet wurden, kamen beim ServerSocket an, jedoch hat die notify-Funktion aus dem Dummy-Service, die diese Nachrichten behandelt, den Nutzdaten keine Werte zugewiesen. Deshalb hat ein GUI-Klick keine Veränderung in den zu sendenden Nutzdaten hervorgerufen. Hier lag der Fehler nicht im Front- oder Back-Service, sondern beim Dummy-Service.

Lösung: Die Notify-Funktion wird so überarbeitet, dass je nach eingehender Nachricht die entsprechende(n) Nutzvariable(n) einen Wert zugewiesen bekommen. Dies geschieht durch Wertezuweisung von statischen Variablen (Hilfsvariablen) des Dummy-Service durch die notify-Funktion. Die Werte dieser statischen Variablen werden dann in den Periodic Tasks des Dummy-Service den zu sendenden Nutzdaten zugewiesen. Dadurch verändern sich die Nutzdaten bei jedem Klick eines GUI-Buttons.

- Die Zustandstransition vom Zustand „Engstelle beidseitig“ in den Zustand „Fußgänger an Zebrastreifen“ funktioniert nicht

Erwartung: Die Transition soll laut untenstehendem UML-Diagrammausschnitt stattfinden, wenn die Bedingung dafür erfüllt ist (rote Markierung in Abbildung 6-1).

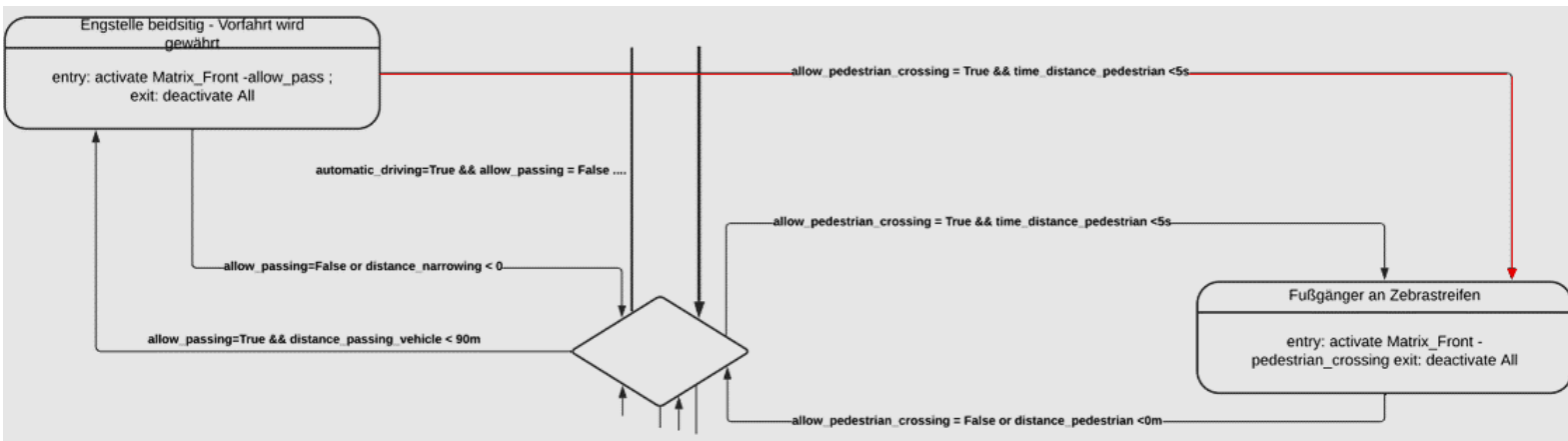


Abbildung 6-1: Ausschnitt Transition im UML-Zustandsdiagramm

Fehler-Beobachtung: Trotz Erfüllung der Transitionsbedingung durch entsprechende Nutzdaten wird auf der LED-Matrix nicht das Zebrastreifen-Symbol angezeigt. Stattdessen wird weiterhin das Engstelle-Symbol angezeigt.

Auswirkung: Im realen Fahrbetrieb hätte dieser Fehler folgende Auswirkung: Das autonome Fahrzeug nähert sich einem Zebrastreifen und soll die entsprechende Symbolik anzeigen um die überquerende Fußgängerin bzw. Fußgänger zu vermitteln, dass das Fahrzeug anhält. Stattdessen wird immernoch das Engstelle-Symbol angezeigt, was eine Verwirrung bei der Fußgängerin bzw. Fußgänger auslösen könnte. Das eHMI hätte somit seinen Zweck nicht erfüllt, da es unpassende Informationen kommuniziert.

Ursache: Ursache war hier ein Softwarefehler im Front-Service. Eine LED-Anzeige wird sobald sie aktiv ist nur beendet, wenn ein Interrupt-Signal ausgelöst wird. Das Interrupt-Signal aus dem Zustand „Engstelle beidseitig“ wird jedoch erst ausgelöst, wenn dieser Zustand verlassen wird. Es wurde jedoch ignoriert, dass ein Zebrastreifen unmittelbar vor der Engstelle sein könnte, also zeitgleich ein anderer Zustand aktiv sein könnte. In diesem Fall kann dieser Zustand nicht die LED-Anzeige für sich beanspruchen, da sie noch im anderen Zustand verharrt und nicht unterbrochen wird. Hier lag der Fehler im Front-Service.

Lösung: Dem Conditional Task für den Zustand „Engstelle beidseitig“ wird eine Abfrage hinzugefügt, die periodisch überprüft, ob der Zustand „Fußgänger an Zebrastreifen“ auch aktiv ist. Ist dies der Fall, so wird die Anzeige sofort unterbrochen und

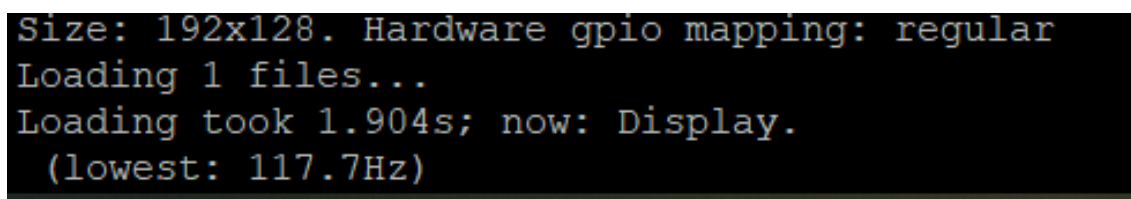
die Transition in den Zebrastreifen-Zustand findet statt. Wird der Zebrastreifen-Zustand verlassen, bevor der Engstelle-Zustand verlassen wird, so wird unmittelbar danach die Symbolik für die Engstelle angezeigt, bis dieser Zustand verlassen wird. Dadurch wird der Zebrastreifen-Zustand mit höherer Priorität als der Engstelle-Zustand behandelt. Die Kommunikation mit der Fußgängerin bzw. dem Fußgänger wird somit der Kommunikation mit dem Gegenverkehr der Engstelle übergeordnet.

Neben diesen beiden funktionsrelevanten Fehlern wurden noch weitere Fehler aufgedeckt, die jedoch keine großen Auswirkungen auf die Funktion hatten und hier nicht weiter vertieft werden sollen. Nach Behebung dieser beiden Fehler konnte festgestellt werden, dass die Dienste die Funktionalität des UML-Zustandsdiagramms erfüllen.

Nachdem bei den ersten Testdurchläufen die Funktionalität sichergestellt wurde, wurde die Performance des Programmes analysiert. Eine relevante Anforderung dabei ist, dass die Bilder und Animationen schnell auf der LED-Matrix angezeigt werden. Ideal wäre eine sofortige Anzeige auf der LED-Matrix, sobald ein Zustand aktiv wird. Ausschlaggebend für die Dauer ist die Ladezeit der Bilder und Animationen. Das nachfolgende Unterkapitel beschäftigt sich mit der Analyse und Optimierung dieser Ladezeiten.

## 6.2 Ladezeiten der Symbole und Animationen

Zeller (2021) gibt an, dass die Anzeige bzw. das Laden von Bildern oder Animationen viel Zeit in Anspruch nehmen kann. Die Bibliothek enthält eine Funktion, die die Ladezeit berechnen kann. Diese ist auch im rgbController vorhanden und gibt die Ladezeit auf der Konsole aus nach jedem Ladevorgang.



```
Size: 192x128. Hardware gpio mapping: regular
Loading 1 files...
Loading took 1.904s; now: Display.
(lowest: 117.7Hz)
```

Abbildung 6-2: Ladezeit für eine GIF-Animation wird mit Hilfe einer Funktion der hzeller-Bibliothek auf der Raspberry Pi Konsole ausgegeben (Ausschnitt)

Beim Testen wurde die Ladezeit für ein Bild (Akku\_Laden.png) und für eine Animation (Wendemanöver\_gegen\_UZS.gif) dokumentiert. Die Ladezeit für das Bild betrug ca. 0,2 Sekunden, für die Animation ca. 1,2 Sekunden. Die Ladezeit für die Animation ist mit einem Wert von über einer Sekunde hoch. Diese hohe Ladezeit könnte die Verkehrseffizienz mindern und sollte daher – auch im Hinblick auf zeitkritische Situationen - reduziert werden.

Eine praktische Möglichkeit um diese Ladezeit zu reduzieren ist die Verwendung von Streams (siehe Kapitel 4.2.4). Der Image-Viewer kann Bild- und Animationsformate einlesen und diese zu einer Stream-Datei verarbeiten. Diese kann anschließend sehr schnell geladen werden und verbraucht weniger Arbeitsspeicher. Diese Methodik eignet sich für große LED-Panels und für Animationen mit vielen Frames. Nachteil ist, dass die resultierenden Stream-Dateien viel Speicherplatz einnehmen können, da diese nicht komprimiert sind (Zeller, 2021). Der benötigte Speicherplatz für die hier verwendeten Streams für dieses Projekt beträgt ca. 14,4 Megabyte. Die Speicherbelastung ist also sehr gering, was auf die geringe Anzahl der anzuzeigenden Symbole und Animationen zurückzuführen ist.

Berechnet man die Ladezeiten für die obigen Dateien, die davor in Stream-Dateien verarbeitet wurden, so ergeben sich drastisch geringere Ladezeiten. Die Ladezeit für das Symbol (AkkuLadenStream.stream) beträgt ca. 0,001 Sekunden, die für die Animation (WendemanöverGegenUZSSStream.stream) beträgt ca. 0,002 Sekunden. Folgende Tabelle fasst die Ergebnisse zusammen und verdeutlicht die Verbesserung der Ladezeit.

**Tabelle 6-1: Verbesserung der Ladezeiten**

	<b>Ladezeit ohne Stream</b>	<b>Ladezeit mit Stream</b>	<b>Verbesserung der Ladezeit</b>
Akku Laden (Bild)	0,2 Sekunden (.png)	0,001 Sekunden	200 mal schneller im .stream-Format
Wendemanöver (Animation)	1,2 Sekunden (.gif)	0,002 Sekunden	600 mal schneller im .stream-Format

Für den realen Betrieb sollten also Stream-Dateien verwendet werden um eine schnelles Laden der anzuzeigenden Inhalte zu bewerkstelligen.

### 6.3 Weiterführende Ansätze für die Programmierung

Dieses Kapitel behandelt weiterführende Ansätze und Ideen für die Software, die in dieser Arbeit nicht verfolgt wurden, aber für Weiterentwicklungen der LED-Software interessant sein könnten.

#### Automatisiertes Testen

In dieser Arbeit wurde ein eigenes Testkonzept entwickelt, das die LED-Anzeige optisch kontrolliert und das manuell durchgeführt wird. Die für das Testen benötigte Zeit kann durch automatisiertes Testen reduziert werden. Dustin et al. (2001, S. 4) merkt an, dass

sich die Automatisierung von Testaktivitäten während der Entwicklungs- und Integrationsphasen besonders auszahlen, wenn wiederverwendbare Testskripts bzw. Testprogramme sehr häufig ausgeführt werden können. (Dustin et al., 2001) Testanforderungen werden dabei durch eine Reihe von Testfällen festgelegt, diese Testfälle legen damit die Testqualität fest. Automatisiertes Testen kann effizient ausgeführt werden und lässt sich leichter wiederholen als beim manuellen Testen (Dustin et al., 2001, S. 5).

Obwohl automatisiertes Testen kostensparend, effizient und insgesamt nützlich sein kann und zu einer Qualitätssicherung der Software beitragen kann, wird in dieser Arbeit darauf verzichtet. Die geringe Anzahl der zu testenden Fälle - also der anzuzeigenden Zustände – rechtfertigt nicht den Aufwand, der für die Entwicklung eines automatisierten Testsystems nötig wäre. Zudem ist das manuelle Testkonzept mittels GUI, nach Auswertung der Testergebnisse, als effektiv und einfach zu bewerten.

### Wettlaufsituation

Als Wettlaufsituation (engl. „race condition“) in Programmen bezeichnet man die Situation, in denen es bei parallelen Programmen mit gemeinsamen Speicher zu zeitabhängigen Fehlern kommt (Netzer & Miller, 1992, S. 1). Wettlaufsituationen treten auf, wenn verschiedene Programme gemeinsamen Speicher nutzen (z.B. eine gemeinsame Variable), ohne dass diese durch Synchronisationsmechanismen (z.B. Semaphore oder Mutex) verwaltet wird. Wettlaufsituationen können beim Programmablauf zu unerwartetem bzw. unerwünschten Verhalten führen.

Bei den Diensten zur Anzeige auf der LED-Matrix könnte es zu einer solchen Situation kommen, wenn mehrere Zustände gleichzeitig aktiv sind und es dafür keine entsprechenden behandelnden Mechanismus gibt. Für die Fahrzeugzustände „Engstelle beidseitig“, „Fußgänger am Zebrastreifen“ oder beim Notfall-Zustand „Sicheres Anhalten“ werden Wettlaufsituation im Quellcode bereits abgefangen bzw. behandelt, indem ein Zustand dem anderen höher priorisiert wird und Vorrang bei der Anzeige erhält. Wenn aber bspw. der Zustand „Wendemanöver auf Punkt links“ und „Wendemanöver auf Punkt rechts“ gleichzeitig aktiv ist, so ist unklar welcher Zustand auf den LEDs angezeigt werden soll. Die Conditional Tasks, die zu diesen Zuständen korrespondieren, nutzen nämlich gleichzeitig die Bildpfad-Variable des rgbControllers – eine Wettlaufsituation tritt ein. Betrachtet man den Fall, dass diese Zustände gleichzeitig aktiv sind, stellt man fest, dass diese Fahrzeugsituation im Hinblick auf die Anwendung im Straßenverkehr sehr untypisch ist. Das Fahrzeug sollte in der Praxis nicht gleichzeitig links- und rechtsherum wenden. Die Implementierung würde aber eine solche Konstellation theoretisch zulassen.

sen. Bspw. könnte man beim Testen mittels GUI eben diese Zustände gleichzeitig aktivieren. Der Front- und Back-Service hat beim aktuellen Entwicklungsstand dafür keine Einschränkung.

In der Praxis wird erwartet, dass beim Fahrzeug nicht gleichzeitig mehrere Zustände aktiv sind, bei denen es keine entsprechende Behandlung gibt, wie es beim Notfall-Zustand bspw. der Fall ist. Wettlaufsituationen können bei der Software demnach in der Theorie auftreten, jedoch nicht in der Praxis, deshalb erübrigen sich Maßnahmen, diese im Programm durch Mechanismen auszuschließen. Daher wurde in dieser Arbeit auf die Behandlung möglicher Wettlaufsituationen bei der Bildpfad-Variable verzichtet. Möchte man Wettlaufsituationen trotzdem ausschließen, so kann man einen sog. mutex (engl. für mutual exclusion) benutzen. Downey (2008, S. 16) beschreibt, dass durch ein mutex garantiert wird, dass nur ein Thread gleichzeitig Zugriff auf eine gemeinsame Ressource (gemeinsame Variable) hat. Weiterhin führt er aus, dass ein mutex wie eine Marke (Token) ist, die von einem Thread zum anderen gereicht wird. Nur dasjenige Thread mit dem mutex kann auf eine gemeinsame Ressource zugreifen. In C++ bietet sich hierfür die Klasse mutex an, die in der Standardbibliothek unter `std::mutex` zu finden ist (cppreference.com, 2021).

## 7 Fazit und Ausblick

In dieser Arbeit wurde eine Software entwickelt, die die externe Kommunikation eines UNICARagil Fahrzeuges realisiert. Für die externe Kommunikation kommt ein eHMI-Konzept zum Einsatz, das aus zwei LED-Matrizen besteht, eines in der Front und eines im Heck des Fahrzeuges. Die zu vermittelnden Informationen über das eHMI sind in einem vorgegebenen Symbolkatalog festgelegt und bestehen aus Bildern und Animationen. Die Symboliken korrespondieren dabei zu bestimmten Fahrzeugzuständen, die in einem UML-Zustandsdiagramm definiert sind. Ziel der Arbeit war, einen Raspberry Pi zu programmieren, der das Zustandsdiagramm umsetzt und damit die externe Kommunikation realisiert. Die Software sollte dabei in jedem Zustand die dazu korrespondierenden Inhalte möglichst ohne zeitliche Verzögerung auf den LED-Matrizen ausgeben. Die Funktionalität des Programms sollte zuletzt mit Hilfe eines eigenständig entwickelten Softwaretests überprüft werden.

Ergebnis der Implementierung sind zwei ASOA-Dienste: Front- und Back-Service. Diese setzen die LED-Anzeige auf den entsprechenden Matrizen in der Front und im Heck um. Diese sind untereinander vernetzt, sodass lediglich der Front-Service mit einem anderen Dienst verbunden werden muss, der dann Fahrzeug-Nutzdaten übermittelt. Aus den Nutzdaten ermittelt der Front-Service dann den aktuellen Fahrzeugzustand und teilt dem Back-Service diesen Zustand mit. Beide Dienste bewerkstelligen dann die Ausgabe der entsprechenden Symbolik auf den Matrizen.

Um diese Funktionalität zu erproben wurde ein Testkonzept entwickelt, das aus einem Dummy-Dienst besteht, der Nutzdaten senden kann, d.h. beliebige Fahrzeugzustände beim Front-Service auslösen kann. Um die Nutzdaten zur Laufzeit einzustellen gibt es eine grafische Benutzeroberfläche, mit der man alle Zustände per Mausklick simulieren kann. Da die grafische Benutzeroberfläche mit der Programmiersprache Java geschrieben ist, wird ein Server-Client-System verwendet, welches die Benutzeroberfläche mit dem in C++ geschriebenen Dummy-Dienst verbindet. So kann die Ausgabe auf der LED-Matrix erprobt werden.

Auffallend bei der Erprobung war, dass die Ladezeit der Symboliken viel Zeit in Anspruch nahm. Gerade bei GIF-Animationen ergaben sich Ladezeiten von über einer Sekunde. Diese Verzögerung machte sich beim Testen stark bemerkbar. Um die Ladezeiten der Symboliken zu verringern gibt es bei der hzeller-Bibliothek die Möglichkeit Dateien im .stream-Format zu laden. Nach Umwandlung in stream-Formate verbesserte sich die Ladezeit bei den Bildern in etwa um das 200-fache und bei den Animationen im Schnitt um das 600-fache. Dadurch konnte eine drastische Reduzierung

der Ladezeiten realisiert werden. Nach dieser Optimierung und nach der Behebung einiger aufgedeckter Softwarefehler ergab sich nach einigen Testdurchläufen ein Programm, das die geforderte Funktionalität aus dem UML-Diagramm vollständig erfüllt.

In dieser Arbeit verhindert die Implementierung nicht das Auftreten von Wettlaufsituationen durch gleichzeitiges Eintreten mehrerer Zustände. Stattdessen wurden gleichzeitig aktive Zustände durch eine höhere Priorisierung eines Zustandes behandelt, wie bspw. beim Notfall-Zustand. Dadurch gibt es für alle in der Praxis auftretenden gleichzeitigen Zustände eine Behandlung. Somit werden Wettlaufsituationen in der Praxis nicht auftreten bzw. werden durch eine Behandlung abgefangen. Sollten in Zukunft weitere Zustände hinzukommen, weil das Fahrzeug bspw. dann mehr Funktionen besitzt, so könnte es zu Wettlaufsituationen führen, die unerwünschtes Verhalten des eHMI zur Folge haben. Diese Problematik könnte in zukünftigen Forschungsarbeiten ggf. behandelt werden.

Zudem stand in dieser Arbeit nur ein Prototyp für die LED-Matrix zur Verfügung. Somit konnten Front- und Back-Service nicht gleichzeitig an LED-Matrizen erprobt werden. Stattdessen wurde der Front-Service am Prototyp getestet und der Back-Service mit Hilfe von Textausgaben auf der Konsole. Die Software sollte daher in zukünftigen Arbeiten mit Hilfe von zwei LED-Matrizen getestet werden, um das Zusammenwirken der beiden Dienste im Betrieb zu testen.

Zuletzt muss die Software unter realen Bedingungen am Fahrzeug und im Straßenverkehr getestet werden.



# Abbildungsverzeichnis

Abbildung 2-1: Aufbau der LED-Module (Schulze, 2020, S. 29) .....	6
Abbildung 2-2: Rückseite LED-Matrix und Verkabelung mit Raspberry Pi.....	7
Abbildung 2-3: LED-Matrix Prototyp (Schulze, 2020, S.32).....	7
Abbildung 2-4: ASOA Funktionalität (UNICARagil news, 2021, S. 75) .....	11
Abbildung 2-5: Ausschnitt aus dem UML-Zustandsdiagramm.....	12
Abbildung 3-1: Fahrzeugkonstellation mit LED-Matrizen und Zuständen .....	14
Abbildung 3-2: Ablaufplan Implementierung .....	15
Abbildung 4-1: Netzwerkkonfiguration Windows Rechner .....	19
Abbildung 4-2: Prinzipbild des gemeinsamen Netzwerkes .....	20
Abbildung 4-3: Gesamtkonzept Hardware-Software-Anbindung .....	22
Abbildung 4-4: Vernetzung von Front- und Back-Service.....	24
Abbildung 4-5: UNICARagil Architektur Tool.....	27
Abbildung 4-6: UML-Klassendiagramm RGBMatrixController .....	28
Abbildung 4-7: Programmablaufplan bild_anzeigen .....	29
Abbildung 4-8: Programmablaufplan RGB_Thread .....	31
Abbildung 4-9: Programmablaufplan Front-Service .....	36
Abbildung 5-1: Vereinigung der Dienstekommunikation im Dummy-Service .....	37
Abbildung 5-2: Ausschnitt modifizierte GUI .....	40
Abbildung 5-3: Prinzipbild: Beispielhafte Kommunikationsverbindung zweier Programme mit Hilfe von Sockets .....	42
Abbildung 5-4: Prinzipbild Server-Client-Kommunikation .....	43
Abbildung 5-5: Ausschnitt Konsolenausgabe Dummy-Service und Back-Service .....	45
Abbildung 5-6: Bildausgabe auf dem Prototyp der LED-Matrix .....	45
Abbildung 6-1: Ausschnitt Transition im UML-Zustandsdiagramm.....	48
Abbildung 6-2: Ladezeit für eine GIF-Animation wird mit Hilfe einer Funktion der hzeller- Bibliothek auf der Raspberry Pi Konsole ausgegeben (Ausschnitt) .....	49

# Abkürzungsverzeichnis

## A

ASOA *Automotive Service-Oriented Architecture*

## D

DHCP *Dynamic Host Configuration Protocol*

dHMI *dynamic Human-Machine-Interface*

## E

eHMI *external Human-Machine-Interface*

## G

GIF *Graphics Interchange Format*

GPIO *General Purpose Input Output*

## H

hzeller-Bibliothek *Henner Zeller Softwarebibliothek*

## I

IDL *Interface Definition Language*

## J

JSON *JavaScript Object Notation*

## L

LfE *Lehrstuhl für Ergonomie*

## P

PNG *Portable Network Graphics*

## R

RGB *Grundfarben: Rot, Grün, Blau*

RPC *Remote Procedure Call*

## U

UML *Unified Modeling Language*

## V

VM *Virtuelle Maschine*

## Y

YAML *Yet Another Markup Language*

# Tabellenverzeichnis

Tabelle 4-1: IDL-Files und Nutzdaten für die Requirements des Front-Service (Ausschnitt) .....	33
Tabelle 6-1: Verbesserung der Ladezeiten .....	50

# Literaturverzeichnis

- Abts, D. (2019). *Masterkurs Client/Server-Programmierung mit Java. Anwendungen entwickeln mit Standard-Technologien* (Lehrbuch, 5. Auflage). Wiesbaden: Springer Vieweg. <https://doi.org/10.1007/978-3-658-09921-3>
- Bengler, K., Rettenmaier, M., Fritz, N. & Feierle, A. (2020). *From HMI to HMIs: Towards an HMI Framework for Automated Driving* (Bd. 11). <https://doi.org/10.3390/info11020061>
- Buchholz, M. (Ed.). (2020). *Automation of UNICARagil Vehicles*. Aachen: Institute for Automotive Engineering RWTH Aachen University; Institute for Combustion Engines RWTH Aachen University. Accessed 28.10.2021. Retrieved from <https://www.aachener-kolloquium.de/en/conference-documents/delayed-manuscripts/2020.html>
- Clemencic, M. & Mato, P. (2012). A CMake-based build and configuration framework. *Journal of Physics: Conference Series*, 396(5). <https://doi.org/10.1088/1742-6596/396/5/052021>
- CMake Reference Documentation — CMake 3.21.3 Documentation. (2021, 20. September). Zugriff am 12.10.2021. Verfügbar unter: <https://cmake.org/cmake/help/latest/index.html#>
- Cppreference.com. (2021). *std::mutex*. Zugriff am 27.10.2021. Verfügbar unter: <https://en.cppreference.com/w/cpp/thread/mutex>
- Downey, A. (2008). *The Little Book of Semaphores*. Green Tea Press. Retrieved from <https://lib.hpu.edu.vn/handle/123456789/21469>
- Dustin, E., Rashka, J. & Paul, J. (2001). *Software automatisch testen. Verfahren, Handhabung und Leistung ; mit 100 Tabellen* (Xpert.press). Berlin, Heidelberg: Springer. Verfügbar unter: <http://swbplus.bsz-bw.de/bsz086628690cov.htm>
- Follmann, R. (2018). *Das Raspberry Pi Kompendium* (2. Aufl. 2018). Berlin, Heidelberg: Springer Berlin Heidelberg. <https://doi.org/10.1007/978-3-662-58144-5>
- FZI Forschungszentrum Informatik. (2018). *LieferBot-E. Nachhaltige, autonome Ver- und Entsorgung rund um die Uhr*. Zugriff am 28.10.2021. Verfügbar unter: <https://www.lieferbot-e.de/>
- Heinemann, G. (2018). *Der neue Online-Handel. Geschäftsmodelle, Geschäftssysteme und Benchmarks im E-Commerce* (9., vollständig überarbeitete Auflage). Wiesbaden: Springer Gabler. <https://doi.org/10.1007/978-3-658-20354-2>

- Javax.swing (Java Platform SE 8 )*. (2021, 20. Oktoberb). Zugriff am 20.10.2021. Verfügbar unter: <https://docs.oracle.com/javase/8/docs/api/index.html?javax/swing/package-summary.html>
- Kalita, L. Socket programming. In *International Journal of Computer Science (IJCSIT)* (Bd. 5, S. 4802–4807). Zugriff am 20.10.2021. Verfügbar unter: <https://doc.presentica.com/11477009/5ebad98adffc8.pdf>
- Kampmann, A., Alrifae, B., Kohout, M., Wustenberg, A., Woopen, T., Nolte, M. et al. (Hrsg.). (2019). *A Dynamic Service-Oriented Software Architecture for Highly Automated Vehicles*. Zugriff am 30.09.2021.
- Kitware Inc. (2021, 20. September). *Home - Kitware Inc.* Zugriff am 07.10.2021. Verfügbar unter: <https://www.kitware.com/>
- Kleuker, S. (2018). *Grundkurs Software-Engineering mit UML. Der pragmatische Weg zu erfolgreichen Softwareprojekten* (4. Auflage). Wiesbaden: Springer Fachmedien Wiesbaden. <https://doi.org/10.1007/978-3-658-19969-2>
- Klingebiel, P. (2021). *C Standard-Bibliothek: signal.h*, Hochschule Fulda. Zugriff am 13.10.2021. Verfügbar unter: <https://www2.hs-fulda.de/~klingebiel/c-stdlib/signal.htm>
- Knoll, A. & Lenz, A. (2020). *Real-Time Systems. Part 8: Concurrency, Threads, Processes and Resource Access Protocols*. Vorlesungsmanuskript. Technische Universität München, Garching. Zugriff am 14.10.2021.
- Magick++ API for GraphicsMagick*. (2021, 1. Januar). Zugriff am 12.10.2021. Verfügbar unter: <http://www.graphicsmagick.org/Magick++/>
- Maurer, M. (Hrsg.). (2015). *Autonomes Fahren. Technische, rechtliche und gesellschaftliche Aspekte* (Springer Open). Berlin: Springer Vieweg. <https://doi.org/10.1007/978-3-662-45854-9>
- Netplan | Backend-agnostic network configuration in YAML*. (2021, 19. Augustd). Zugriff am 06.10.2021. Verfügbar unter: <https://netplan.io/>
- Netzer, R. H. B. & Miller, B. P. (1992). What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1), 1–12. <https://doi.org/10.1145/130616.130623>
- OpenSSL Foundation, Inc. (2021, 7. Oktober). */source/index.html*. Zugriff am 07.10.2021. Verfügbar unter: <https://www.openssl.org/source/>
- Putty.org*. (2018, 18. April). Zugriff am 04.10.2021. Verfügbar unter: <https://www.putty.org/>
- The Qt Company. (2021, 20. Oktober). *Qt Creator*. Zugriff am 20.10.2021. Verfügbar unter: <https://www.qt.io/company>

- Qt GUI 5.15.6. (2021, 8. Septembere). Zugriff am 20.10.2021. Verfügbar unter: <https://doc.qt.io/qt-5/qtgui-index.html>
- Rettenmaier, M., Schulze, J. & Bengler, K. (2020). How Much Space Is Required? Effect of Distance, Content, and Color on External Human–Machine Interface Size. *Information*, 11(7), 346. <https://doi.org/10.3390/info11070346>
- Rupp, C. & Queins, S. (2012). *UML 2 glasklar. Praxiswissen für die UML-Modellierung* (4., aktualisierte und erw. Aufl.). München: Hanser.
- Sammereier, M. (2021, 20. Oktober). *GitLab HiWi Repo*. Zugriff am 20.10.2021. Verfügbar unter: <https://gitlab.lrz.de/ga87wop/unicaragil-ehmi-hiwi-repo>
- Schulze, J. (2020, 12. Februar). *Weiterentwicklung und Evaluation eines externen Anzeigekonzepts für automatisierte Fahrzeuge - Auswirkung von Distanzen und Lichtverhältnissen auf die Erkennbarkeit*. Masterarbeit. Technische Universität München, Garching.
- Schwiebacher, J. (2021). *Meilensteinbericht MVII-13*. Technische Universität München, Garching. Zugriff am 04.05.2021.
- Seaman, J. (2013). *C++ Programming on Linux*. Verfügbar unter: <https://user-web.cs.txstate.edu/~js236/201912/cs2308/multi-filedev.pdf>
- Storm, R. (2019, 25. Februar). *Konzeptionierung und Implementierung eines Prototyps für externe Anzeigen autonomer Fahrzeuge*. Masterarbeit. Technische Universität München, Garching. Zugriff am 29.09.2021.
- Thienen, W. (1999). *Client/Server. Moderne Informationstechnologien im Unternehmen* (Springer eBook Collection Computer Science and Engineering, 2., überarbeitete Auflage). Wiesbaden: Vieweg+Teubner Verlag. <https://doi.org/10.1007/978-3-322-86864-0>
- UNICARagil. *ASOA Basiskonzept. unveröffentlichtes Konzeptpapier*.
- UNICARagil. (2021). *Erstellung eines ASOA Dienstes*. Zugriff am 12.10.2021.
- UNICARagil Architektur Tool. (2021, 12. Oktoberf). Zugriff am 12.10.2021. Verfügbar unter: [https://architektur.unicaragil.embedded.rwth-aachen.de/index.php?action=show\\_view&view\\_id=1](https://architektur.unicaragil.embedded.rwth-aachen.de/index.php?action=show_view&view_id=1)
- UNICARagil news. (2021). *unicaragil: Digitales Halbzeitevent*. Zugriff am 04.05.2021. Verfügbar unter: <https://www.unicaragil.de/de/digitales-halbzeitevent.html>
- Verband der Automobilindustrie e.V. (2021, 28. Oktober). *(Keine) Zukunftsmusik: So gibt autonomes Fahren Senioren ihre Mobilität zurück*. Zugriff am 28.10.2021. Verfügbar unter: <https://www.iaa.de/de/mobility/newsroom/mobility-stories/digital/keine-zukunftsmusik-so-gibt-autonomes-fahren-senioren-ihre-mobilitaet-zurueck>
- Williams, A. (2012). *C++ concurrency in action. Practical multithreading*. Shelter Island, NY: Manning.

- Zeller, H. (2021, 11. Oktober). *rpi-rgb-led-matrix. Controlling up to three chains of 64x64, 32x32, 16x32 or similar RGB LED displays using Raspberry Pi GPIO*. Zugriff am 11.10.2021. Verfügbar unter: <https://github.com/hzeller/rpi-rgb-led-matrix>
- Zisler, H. (2015). *Computer-Netzwerke. Grundlagen, Funktionsweise, Anwendung ; [für Studium, Ausbildung, Beruf ; Theorie und Praxis: von der MAC-Adresse bis zum Router ; TCP/IP, IPv4, IPv6, (W)LAN, VPN, VLAN u.v.m. ; Konfiguration, Planung, Aufbau und sicherer Betrieb von Netzwerken; inkl. Open-WRT* (Galileo Press Galileo Computing, Bd. 3479, 3., aktualisierte und erw. Aufl.). Bonn: Galileo Press.

## A IDL-Files und Nutzdaten (Front-Service)

Zustand	IDL-File für das benötigte Requirement	Nutzdaten
Akku Laden Loading_status	Load_sts	bool currently_loading
Ausparken rechts (90°) Start_Right	LRPR_sts	bool leaving_parking_rectangular_right
Ausparken links (90°) Start_left	LPRL_sts	bool leaving_parking_rectangular_left
Engstelle beidseitig Allow_pass	APass_sts	bool allow_passing float distance_passing_vehicle float distance_narrowing bool allow_pedestrian_crossing float time_distance_pedestrian
Fußgänger an Zebrastreifen Ped_cross	AlPedCr_sts	bool allow_pedestrian_crossing float time_distance_pedestrian float distance_pedestrian
Wendemanöver rechts Turning_right	TurnRig_sts	float turning_right_dynamic_module_planned_t bool turning_finished char dynamic_module_position[255]
Wendemanöver links Turning_left	TurLeft_sts	float turning_left_dynamic_module_planned_t bool turning_finished char dynamic_module_position[255]
Einparken links (90°) Stop_left	RPL_sts	bool rectangular_parking_left bool parking_aborted bool parking
Einparken rechts (90°) Stop_right	RPR_sts	bool rectangular_parking_right bool parking_aborted bool parking
Sicheres Anhalten Warning_sign	SafStop_sts	bool set_safe_stop_status



## B Netzwerkkonfiguration YAML-Datei

```
1 network:
2   version: 2
3   renderer: networkd
4   ethernets:
5     eth0:
6       addresses: [192.168.1.2/24, ]
7       dhcp4: false
8       gateway4: "192.168.1.1"
9       nameservers:
10         addresses:
11           - "8.8.8.8"
12           - "8.8.4.4"
```

## C CMakeLists.txt

```
1  cmake_minimum_required(VERSION 3.5)
2  project(asa_template)
3
4  set(CMAKE_BUILD_TYPE Release)
5  set(CMAKE_CXX_STANDARD 17)
6
7  cmake_policy(SET CMP0054 NEW)
8  cmake_policy(SET CMP0042 NEW)
9  cmake_policy(SET CMP0079 NEW)
10 find_package(asa_core 0.2.0 REQUIRED)
11
12 #RGB Library einbinden
13 set(RGBSERVICE "librgbmatrix.a")
14 find_library(RGBSERVICE librgbmatrix.a PATHS /lib)
15 link_libraries(${target} ${RGBSERVICE})
16 set(RGBSERVICE_INCLUDE_DIRS /include)
17 include_directories(${target} ${RGBSERVICE_INCLUDE_DIRS})
18 include_directories("include/")
19 link_directories("lib/")
20
21
22 file(GLOB EXECES "src/*.cpp")
23 foreach(src_ ${EXECES})
24     get_filename_component(target ${src_} NAME)
25     string(REPLACE ".cpp" "" target ${target})
26     string(REPLACE "/" "_" target ${target})
27     add_executable(${target} ${src_})
28
29     #ASOA Core einbinden
30     target_link_libraries(${target} asa_core)
31
32     #RGB Matrix Library Target
33     target_link_libraries(${target} ${RGBSERVICE})
34
35     #Magick Library target
36     target_link_libraries(${target} -L/usr/lib -L/usr/lib/X11
37 -lGraphicsMagick++ -lGraphicsMagick -ljbig -lwebp -lwebpmux
38 -llcms2 -ltiff -lfreetype -ljpeg -lpng16 -lwmflite -lXext
39 -lSM -lICE -lX11 -llzma -lbz2 -lxml2 -lz -lm -lgomp -lpthread)
40
41     message("-- Found target:")
42     message("-- " ${target})
43 endforeach()
44
```