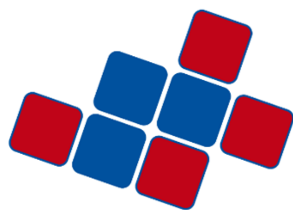


SYSTEMANALYSE UND
-OPTIMIERUNG
Carl von Ossietzky Universität Oldenburg

Modulare Softwarearchitektur in der Automobilindustrie: Eine mikroserviceorientierte Architektur zur Verbesserung der Entwicklung und Evaluation sicherheitsrelevanter Funktionen

Masterarbeit zur Erlangung des akademischen Grades
Master of Science
im Studiengang Informatik

Fakultät II - Department für Informatik
Abteilung Systemanalyse und -optimierung

Themensteller: Prof. Dr.-Ing. Axel Hahn
Zweitprüfer: Dr.-Ing. Arne Lamm

angefertigt von: B. Sc. Niklas Rahenbrock

Abgabetermin 24.02.2025

Zusammenfassung

Die zunehmende Softwarekomplexität in der Automobilindustrie stellt neue Herausforderungen an die Entwicklung und Validierung sicherheitskritischer Funktionen, insbesondere in Software-Definied Vehicles (SDVs). Daher untersucht diese Arbeit die Frage, wie eine Simulations- und Testumgebung die Entwicklung und frühzeitige Evaluation modularer Softwaresysteme in verschiedenen Entwicklungsstufen eines SDV unterstützen kann. Um diese Frage zu beantworten, wird eine Testumgebung entworfen, die eine schrittweise Annäherung von vollständig simulierten Tests bis hin zu Tests nahe am Endprodukt mit einer Hardware-in-the-Loop (HiL)-Umgebung und Over-The-Air (OTA)-Sensoren ermöglicht. Das Konzept integriert standardisierte Schnittstellen wie das Open-Simulation-Interface (OSI) sowie eine nachrichtenorientierte Middleware. Zentrale Bestandteile der Umgebung sind ein Adaptermodul zur nahtlosen Integration des zu testenden Systems und eine vollständige Automatisierung zur kontinuierlichen Integration und Validierung. Die Evaluation der Testumgebung erfolgt anhand verschiedener Use-Cases und den definierten Anforderungen. Die Ergebnisse zeigen, dass die Testumgebung die definierten Anforderungen erfüllt. Damit ermöglicht die entwickelte Testumgebung eine frühe Evaluation in verschiedenen Stufen, reduziert den Testaufwand und erleichtert die Integration modularer Softwaresysteme eines SDV.

Abstract

The increasing software complexity in the automotive industry poses new challenges for the development and validation of safety-critical functions, especially in Software-Definied Vehicles (SDVs). Therefore, this thesis examines the question of how a simulation and test environment can support the development and early evaluation of modular software systems in different development stages of a SDV. To answer this question, a test environment is designed that enables a stepwise approach from fully simulated tests to tests close to the final product with a Hardware-in-the-Loop (HiL) environment and Over-The-Air (OTA) sensors. The concept integrates standardized interfaces such as the Open-Simulation-Interface (OSI) and a message-oriented middleware. Central components of the environment are an adapter module for seamless integration of the system under test and complete automation for continuous integration and validation. The results show that the test environment fulfills the defined requirements. The developed test environment thus enables an early evaluation in various stages, reduces the testing effort and facilitates the integration of modular software systems of a SDV.

Inhaltsverzeichnis

Abkürzungen **iv**

Abbildungsverzeichnis **vii**

Tabellenverzeichnis **x**

1. Einleitung **1**
1.1. Problemstellung . . . 2
1.2. Ziele der Arbeit . . . 3
1.3. Aufbau der Arbeit . . . 3

2. Grundlagen **4**
2.1. Hochautomatisierte Fahrzeuge und Fahrsysteme . . . 4
2.1.1. Modell eines hochautomatisierten Fahrsystems . . . 5
2.1.2. Software-Defined Vehicle . . . 6
2.2. Software- und Systementwicklung . . . 7
2.2.1. Entwicklung autonomer Fahrzeugsysteme . . . 7
2.2.2. XiL-Ansatz . . . 8
2.3. Einsatz von Testumgebungen/Testfeldern . . . 11
2.3.1. Open-Loop-Tests und Closed-Loop-Tests . . . 12
2.3.2. Referenzarchitektur für Testfelder . . . 12
2.3.3. Sensorsimulation und Komplexitäten . . . 14
2.4. Architekturen von automatisierten Fahrzeugen . . . 15
2.4.1. AUTOSAR Adaptive Plattform . . . 16
2.4.2. Kommunikationsprotokolle, Frameworks und Standards . . . 17

3. Use-Cases **19**
3.1. Use-Case 1: Modulares Testen in der Test- und Simulationsumgebung . . . 19
3.2. Use-Case 2A: Migration und Test von Mikroservices auf realer Hardware . . . 20
3.3. Use-Case 2B: Automatische Integration und Tests neuer Versionen des SUT . 21

4. Anforderungen **23**
4.1. Anforderungen aus Use-Case 1 . . . 23
4.2. Anforderungen aus Use-Case 2A . . . 25
4.3. Anforderungen aus Use-Case 2B . . . 26

5. Verwandte Arbeiten **27**
5.1. Testfelder für Cyber-Physical-Systems . . . 27
5.2. Testumgebungen für das Testen von autonomen Fahrfunktionen . . . 30
5.3. Softwareentwicklung von Automated Driving Systems . . . 34
5.4. Handlungsbedarf . . . 38
5.5. Zusammenfassung der verwandten Arbeiten . . . 42

6. Konzept und Systemarchitektur der Testumgebung **44**
6.1. Konzeptübernahme aus verwandter Arbeit . . . 44
6.1.1. Änderungsbedarf und Abgrenzung . . . 45
6.1.2. Erweiterung . . . 48
6.2. Architektur der Testumgebung . . . 49
6.2.1. Einheitliches Datenformat OSI . . . 49
6.2.2. Nachrichtenorientierte Middleware . . . 52
6.3. Komponenten der virtuellen und physischen Testumgebung . . . 53
6.4. System under Test (SUT) . . . 56
6.5. Adaptermodul . . . 58
6.6. Test-, Verifikations- und Validierungs-Management . . . 59
6.6.1. Monitoring . . . 60
6.6.2. Szenarien . . . 61
6.6.3. Bereitstellungsorchestrator und Automatisierungsdienst . . . 61
6.7. Zusammenfassung der Systemarchitektur . . . 63

7. Implementierung **64**
7.1. Dockerprojekt . . . 64
7.2. Nachrichtenorientierte Middleware mit ROS . . . 67
7.3. Benutzerdefinierte OSI-Nachrichtenklasse . . . 68
7.4. Komponenten der virtuellen Testumgebung . . . 69
7.5. Adaptermodul . . . 71
7.5.1. Auswahl des Kommunikationsprotokolls . . . 71
7.5.2. Auswahl der Eingabedaten. . . . 73
7.5.3. Modellierung . . . 73
7.6. Monitoring-Schnittstelle . . . 74
7.7. Erstellung eines Basisimage für das SUT . . . 75
7.8. Bereitstellungsorchestrator . . . 76
7.9. Starten und Stoppen der Testumgebung . . . 77
7.10. Automatisierungsdienst . . . 77

8. Evaluation der Testumgebung **79**
8.1. Beschreibung des verwendeten SUT . . . 79
8.2. Closed-Loop-Test eines Lane-Followers – Use-Case 1 . . . 80
8.2.1. Durchführung . . . 80
8.2.2. Erfüllung der Anforderungen . . . 84
8.3. Migration und Tests auf realer Hardware – Use-Case 2A . . . 91
8.3.1. Durchführung . . . 92
8.3.2. Erfüllung der Anforderungen . . . 94
8.4. Automatisierte Testläufe über eine CI/CD – Use-Case 2B . . . 95
8.4.1. Durchführung . . . 96
8.4.2. Erfüllung der Anforderungen . . . 98
8.5. Validierung . . . 99
8.5.1. Vergleich der Laufzeit des Lane-Followers . . . 100
8.5.2. Lose Kopplung und Blackbox-Ansatz . . . 102
8.5.3. Durchführung von Open-Loop-Tests . . . 103
8.6. Zusammenfassung der Evaluierung und Zielabdeckung . . . 104

9. Zusammenfassung **106**
9.1. Ergebnisse und Fazit . . . 106
9.2. Ausblick . . . 108

Literaturverzeichnis **111**

A. Vollständige Architektur der Testumgebung und des zu testenden Systems **123**

B. Sequenzdiagramm für Use-Case 2A und 2B **124**

C. Verarbeitungskette der Testumgebung von Closed-Loop-Tests **126**

D. Anwendung der Referenzarchitektur auf einen Highway Piloten **127**

E. Definition der CI/CD-Pipeline über eine gitlab-ci.yml **128**

Abkürzungen

ADAS Advanced Driver Assistance System

ADS Automated Driving System

API Application Programming Interface

ASAM Association for Standardization of Automation and Measuring Systems

ACC Adaptive Cruise Control

AUTOSAR Automotive Open System Architecture

CI/CD Continuous Integration/Continuous Delivery

CAN Controller Area Network

CPS Cyber-Physical System

CARLA Car Learning to Act

CiL Camera-in-the-Loop

DiL Data-in-the-Loop

DDS Data Distribution Service

DSL Domain-Specific Language

ECU Electronic Control Unit

E/E Elektrische/Elektronische

FOV Field of View

FMI Functional Mock-up Interface

FMU Functional Mock-up Unit

GNSS Global Navigation Satellite Systems

HiL	Hardware-in-the-Loop
HLA	High Level Architecture
HTTP	Hypertext Transfer Protocol
IoU	Intersection over Union
MQTT	Message Queuing Telemetry Transport
MiL	Model-in-the-Loop
OSI	Open-Simulation-Interface
OEM	Original Equipment Manufacturer
OTA	Over-The-Air
PiL	Processor-in-the-Loop
PPC	Perception-Planning-Control
QoS	Quality of Service
ROS	Robot Operating System
REST	Representational State Transfer
RTOS	Real-Time Operating System
SAE	Society of Autonomous Engineers
SiL	Software-in-the-Loop
SDV	Software-Definied Vehicle
SUT	System Under Test
SSH	Secure Shell
SWOT	Strengths, Weaknesses, Opportunities and Threats
SOME/IP	Scalable Service-Oriented Middleware over IP
SSE	Server-Sent Events
TCP	Transmission Control Protocol
TSN	Time-Sensitive Networking

UML	Unified Modeling Language
UDP	User Datagram Protocol
URL	Uniform Resource Locator
ViL	Vehicle-in-the-Loop
V-ECU	Virtual Electronic Control Unit
V2X	Vehicle-to-Everything
V+V	Verifikation und Validierung
XiL	X-in-the-Loop
XML	Extensible Markup Language

Abbildungsverzeichnis

2.1. Blockdiagramm des Sense-Plan-Act-Schemas mit Verbindung zum Umfeld . . 6
2.2. V-Modell, nach Hagen (2020, S.33) 8
2.3. Blockdiagramm für eine allgemeine Methodik der Validierung von XiL-Ansätzen (Riedmaier et al. 2018) 10
2.4. V-Modell in Kombination mit XiL-Ansätzen für die Entwicklung von ADS (Magosi et al. 2022) 11
2.5. Blockschaltbild für Open-Loop-Test und Closed-Loop-Test, nach Baumann (2006) und Brinkmann (2018) 12
2.6. Referenzarchitektur eines generischen Testfeldes für automatisierte CPS (Nickovic et al. 2017) 13
2.7. Verschiedene Level der Sensorsimulation, die als Graph zwischen den Achsen Realitätstreue und Komplexität aufgetragen sind (Sievers et al. 2018) 14

5.1. Elemente eines Testfeldes für automatisierte Schiffsführungssysteme (Brinkmann 2018, S.66) 28
5.2. Systemarchitektur des physischen Testfeldes (Brinkmann 2018, S.69) 29
5.3. Gemischte ADS-HiL-Testumgebung mit verschiedenen Sensormodellen (Feilhauer et al. 2016) 31
5.4. Validierung eines Sensor-ECU über verschiedene Abstraktionsoptionen, nach Sievers et al. (2018) 32
5.5. Injizierung von rohen Kameradaten zur Umgehung der Kameralinse und des Kamerasensors (Sievers et al. 2018) 33
5.6. Co-Simulation zwischen SiL und HiL über eine Verbindung über Ethernet (Ruehl und Bronner 2024) 34
5.7. Überblick über die vorgestellte Tool-Suite der Mikroservices-Architektur zu Bereitstellung von ROS-Applikationen (Busch et al. 2024) 35
5.8. Architektur des Lane-Followers, nach Lotz et al. (2019) 36

6.1. Anpassung der Elemente einer Testumgebung für Autonomous Driving Systems, nach Brinkmann (2018, S.66) 46
6.2. Systemarchitektur der Testumgebung 50

6.3. Aufbau der Klasse osi3::BaseMoving (ASAM 2024c) 51
6.4. Mockup-Version einer Perzeptionskomponente 52
6.5. Veröffentlichen und Abonnieren von Daten in der Testumgebung, nach Brinkmann (2018, S.76) 53
6.6. Erweiterung der bereits gezeigten Systemarchitektur um die Komponenten der physischen und virtuellen Testumgebung 54
6.7. Softwarekomponenten des Systems under Test (SUT) und Aufteilung in die HiL- und SiL-Umgebung 57
6.8. Erweiterung der Nachrichtenkommunikationskette zwischen Datenquelle und ADS-Funktion durch das Adapatermodul 59
6.9. Trennung zwischen der Testumgebung und dem zu testenden System durch das Adaptermodul 60

7.1. Visualisierung des Docker-Stacks der Testumgebung mit Umgebungsvariablen, Ports und Volumen 65
7.2. UML Klassendiagramm des Simulationsadapters (Publisher) 69
7.3. UML Klassendiagramm des Adaptermoduls (Subscriber) 74

8.1. Aufbau der Testumgebung mit Integration des Lane-Followers als SUT 81
8.2. Visualisierung des Ego-Fahrzeugs in der Simulation 82
8.3. Visualisierung der beim SUT vorliegenden Eingabedaten 83
8.4. Hardware-Aufbau des OTA-Sensors (links) und die daraus resultierenden Eingabedaten (rechts) 83
8.5. Aufbau der Testumgebung im hybriden Ansatz, in dem virtuelle Radardaten mit Kameradaten aus einem realen Sensor verbunden werden 85
8.6. Notwendige Änderungen (Eingabe- und Ausgabedaten) des zu testenden Systems für die Integration in die Testumgebung 88
8.7. Visualisierung der berechneten Steuerbefehle des SUT 91
8.8. Konfiguration des zu testenden Systems und Portierung von der SiL- in die HiL-Umgebung 92
8.9. Visualisierung des Docker-Swarms und der zugehörigen Services 94
8.10. Ausschnitts des instanziierten Konzeptes für Use-Case 2B 96
8.11. Ablauf der automatisierten Testläufe über einen GitLab-Runner 97
8.12. Durchlauf der verschiedenen Jobs der Pipeline 97
8.13. Vergleich der Laufzeit des Lane-Followers zwischen monolithischer und mikroserviceorientierter Architektur 101

A.1. Vollständige Systemarchitektur der Testumgebung 123

B.1. Sequenzdiagramm für UC-2A 124
B.2. Sequenzdiagramm für UC-2B 125

C.1. Verarbeitungskette der Testumgebung von Closed-Loop-Tests, zwischen der Simulation, dem Simulationsadapter, dem Sensormodell, dem Adaptermodul und dem SUT 126

D.1. Anwendung der Referenzarchitektur auf einen Highway Piloten. Die schwarzen gestrichelten Linien zeigen, dass die verschiedene Komponenten der Testumgebung virtuell oder physisch existieren können (Nickovic et al. 2017) 127

Tabellenverzeichnis

2.1. SAE Automatisierungsstufen nach J3016, aus (SAE 2021) übersetzt 5

5.1. Kompatibilität der Testmethode in unterschiedlichen Testumgebungen, nach Johansson und Paulsson (2024) 37

5.2. Erfüllung der aus Use-Case 1, 2A und 2B abgeleitete Anforderungen durch verwandte Arbeiten 43

7.1. Übersicht der WebSocket-Verbindungen, ROS-Topics und verwendeten OSI-Nachrichten 72

7.2. Endpunkte der REST-API 74

Quellcodeverzeichnis

7.1. Verpacken einer OSI-Nachricht in eine ROS-Nachricht 68
7.2. Struktur der benutzerdefinierten OSI-Nachrichtenklasse Control-Command . . 68
7.3. Anwenden der Steuerbefehle auf das Ego-Fahrzeug in der Simulation über die CARLA-Python-API 70
7.4. Definition der Registry als Docker-Service und Hochladen des Images 76
7.5. Verwendung des Startskripts zum Hochfahren der Testumgebung und Starten eines Testlaufs 77

8.6. Verwendung des OSI-Formats als standardisiertes Format für Eingabedaten . 86
8.7. Auszug aus einem beispielhaften Bericht des LaneFollow-Szenarios 90
8.8. Kompilieren der Docker-Images für die Architekturen linux/amd64 und linux/arm64 93
8.9. Auszug aus dem Dockerfile des zu testenden Systems, in dem der Rollenname festgelegt wird 93
8.10. Anzeige der laufenden Docker-Prozesse auf dem Simulations-PC und dem NVIDIA-Jetson 95
8.11. Konfigurationsdatei für automatisierte Testläufe 99

1. Einleitung

Die Automobilindustrie befindet sich im Wandel, da die klassische Interaktion zwischen Fahrer und mechanischem Fahrzeug zunehmend einer vollständigen Automatisierung des Fahrvorgangs weicht (Garikapati und Shetiya 2024). Diese automatisierten Funktionen werden aufgrund des Zusammenspiels von Hardware, Software und Umweltauswirkungen auch als Cyber-Physical System (CPS) bezeichnet (Al-Jaroodi et al. 2016). Hochautomatisierte Fahrzeuge sind in der Lage, Aufgaben wie das Navigieren auf der Straße oder das Erkennen von Hindernissen ohne menschliches Eingreifen zu übernehmen, was potenziell die Verkehrssicherheit erhöht und den Fahrer entlastet (Isermann 2016, S.6f). Ein Beispiel hierfür ist der von Mercedes entwickelte Level 3 „Drive Pilot“. Dieser übernimmt bis zu Geschwindigkeiten von 95 km/h in bestimmten Szenarien alle Fahrfunktionen und ermöglicht dem Fahrer Freiheiten wie die Nutzung des Infotainmentsystems[2]. Um diese vom Kunden gewünschten Funktionen zu ermöglichen, reagieren die Fahrzeughersteller zunehmend mit softwaregesteuerten Systemen. Diese Entwicklung wird unter dem Begriff Software-Definied Vehicle (SDV) zusammengefasst. Ein SDV zeichnet sich durch einen starken Fokus auf die Software aus, ermöglicht kontinuierliche Funktionsupdates über Over-The-Air (OTA)-Mechanismen und verbessert die Wartung und Weiterentwicklung ohne physische Eingriffe (Hills 2020).

Um die Sicherheit und Funktionsweise zu gewährleisten, ist die Verifikation und Validierung (V+V) solcher Funktionen essenziell, da Fehler fatale Folgen haben können. Gleichzeitig gewinnen softwaregesteuerte Systeme zunehmend an Bedeutung, weshalb auch das V+V weiterentwickelt werden muss, um den neuen Herausforderungen gerecht zu werden (Siemens 2024). Insbesondere die Perzeption, d. h. die Wahrnehmung der Umgebung, ist häufig einer der schwierigsten Aspekte eines Automated Driving System (ADS) und erfordert eine große Aufmerksamkeit in der V+V (Gruyer et al. 2017). Die Möglichkeit, ein ADS in einer Testumgebung mit verschiedenen Realitätsstufen von Sensordaten (von perfekten Simulationsdaten bis hin zu einem realen Sensor) zu testen, bietet eine schrittweise Annäherung an reale Einsatzbedingungen und ermöglicht eine gezielte Optimierung der Sensordatenverarbeitung der Perzeption (dSPACE 2024).

Ein vielversprechender Ansatz, um diese Herausforderungen zu bewältigen, ist der Ein-

2. https://www.adac.de/rund-ums-fahrzeug/ausstattung-technik-zubehoer/autonomes-fahren/technik-vernetzung/autonomes-fahren-staupilot-s-klasse/ (Abgerufen am 10.02.2025)

satz virtueller Testmethoden, wie Software-in-the-Loop (SiL)- und Hardware-in-the-Loop (HiL)-Verfahren, die die Entwicklung von ADS gezielt unterstützen (Lou et al. 2022; Riedmaier et al. 2018). Die dafür notwendigen Testumgebungen müssen flexibel und standardisiert sein, um das Testen von virtuellen und physischen Komponenten zu ermöglichen. Dabei können Tests schrittweise von rein virtuellen Umgebungen (SiL) zu immer realistischeren Teststufen (HiL, Vehicle-in-the-Loop (ViL)) überführt werden, was eine inkrementelle Entwicklung von ADS ermöglicht (Riedmaier et al. 2018). Ein logischer nächster Schritt ist daher ein hybrider Ansatz, bei dem virtuelle und physische Komponenten mit virtuellen Umgebungen kombiniert werden, um die Entwicklungsqualität weiter zu steigern (dSPACE 2024). Um diesen Prozess insbesondere im Kontext von mikroserviceorientierten Architekturen weiter zu optimieren, kann eine gezielte Portierung einzelner Teilsysteme oder Services auf spezifische Hardware eingesetzt werden. Damit können einzelne Funktionen frühzeitig in Umgebungen mit realitätsnahen Ressourcen getestet werden.

1.1. Problemstellung

ADS sind hochkomplexe Systeme, die aus verschiedenen Soft- und Hardwarekomponenten bestehen. Das Zusammenspiel dieser Komponenten und insbesondere die Integration in die reale Umgebung erfordern eine umfangreiche Validierung und Verifikation (Schäuffele und Zurawka 2024, S.25f). Nach einer Studie von McKinsey können diese Entwicklungskosten für ein hochautomatisiertes Fahrzeug (SAE Level 4) zwischen 500 Mio. und 1 Mrd. US-Dollar liegen (McKinsey 2022). Diese hohen Entwicklungskosten lassen sich aus den erforderlichen Testkilometern ableiten, die für das Testen eines ADS notwendig sind. Bisherige Testmethoden und -umgebungen sind oft starr und nicht flexibel genug, um die verschiedenen Entwicklungsstufen von ADS zu unterstützen. Es fehlen standardisierte Ansätze, die eine nahtlose Integration von virtuellen und physischen Komponenten ermöglichen. Dies erschwert die iterative Entwicklung und das frühzeitige Testen einzelner Teilsysteme. Zudem ist eine enge Kopplung von virtuellen Simulationen und physischen Tests erforderlich, bei denen virtuelle Komponenten schrittweise durch physische ersetzt werden. Bisherige Ansätze sind jedoch oft isoliert und erlauben keine effiziente Kombination von HiL, SiL und ViL Tests. Dadurch entsteht eine Lücke zwischen Simulation und realer Welt, die die Entwicklung verlangsamt und die Qualität der Tests beeinträchtigt. Die Entwicklung von ADS erfordert eine frühzeitige Evaluierung von Teilsystemen, um Fehler zu erkennen und zu beheben. Bisherige Testumgebungen sind jedoch oft nicht in der Lage, modulare Softwaresysteme in frühen Entwicklungsphasen zu testen, was zu einer verzögerten Fehlererkennung und höheren Entwicklungskosten führt.

1.2. Ziele der Arbeit

Anhand der beschriebenen Problemstellung entsteht der Bedarf nach einer Simulations- und Testumgebung, die die Entwicklung eines SDVs verbessern soll. Es ergibt sich folgende Forschungsfrage, welche in dieser Arbeit beantwortet wird:

> *Wie kann eine Simulations- und Testumgebung die Entwicklung und frühzeitige Evaluation modularer Softwaresysteme in unterschiedlichen Entwicklungsstufen eines Software-Defined Vehicles ermöglichen?*

Um diese Fragestellung beantworten zu können, soll zunächst auf der Basis von Anwendungsfällen, Anforderungen und verwandter Arbeiten ein Konzept erstellt werden. Dieses Konzept beschreibt die Systemarchitektur der Simulations- und Testumgebung, die eine frühzeitige Evaluation modularer ADS ermöglichen soll. Dabei wird eine Testumgebung angestrebt, die durch ihre Flexibilität und Erweiterbarkeit sowohl virtuelle als auch physische Testkomponenten integriert und somit den Einsatz verschiedener Testmethoden wie HiL und SiL unterstützt. Das erarbeitete Konzept wird anschließend durch eine prototypische Implementierung realisiert. Zur Evaluierung des Konzepts kommen Teile des Aufbaus sowie der Lane-Follower als ADS aus Modrakowski et al. (2024) zum Einsatz.

1.3. Aufbau der Arbeit

In Kapitel 2 werden zunächst Grundlagen zu hochautomatisierten Fahrzeugen, Softwareentwicklung, Testumgebungen und Fahrzeugarchitekturen behandelt. Aufbauend auf der Problemstellung und den Grundlagen werden in Kapitel 3 drei verschiedene Anwendungsfälle (engl. „Use-Cases“) beschrieben. Aus diesen Use-Cases leiten sich die in Kapitel 4 definierten Anforderungen ab. Aufbauend auf den Grundlagen, den Use-Cases und den Anforderungen folgen in Kapitel 5 die verwandten Arbeiten, aus denen zusammenfassend ein Handlungsbedarf abgeleitet wird. Darauf folgt in Kapitel 6 die Beschreibung des Konzepts und der Systemarchitektur der Testumgebung, die in Kapitel 7 prototypisch umgesetzt wird. Abschließend erfolgt eine Evaluation der Testumgebung (Kapitel 8), in der die definierten Anforderungen überprüft, die Forschungsfrage beantwortet und in Kapitel 9 ein Fazit gezogen wird.

2. Grundlagen

Wie bereits in der Einleitung definiert, ist das Ziel dieser Arbeit die Entwicklung einer Simulations- und Testumgebung zur Verbesserung der Entwicklung eines SDV. Im folgenden Kapitel werden Grundlagen vorgestellt, die für die Einordnung und das Verständnis der weiteren Arbeit relevant sind. In diesem Kapitel werden Technologien und Ansätze betrachtet, die dem Stand der Technik entsprechen. Zunächst erfolgt eine allgemeine Beschreibung hochautomatisierter Fahrzeuge und Fahrsysteme (Abschnitt 2.1) und der zugehörigen Entwicklung dieser Systeme (Abschnitt 2.2). Die für diese Entwicklung notwendigen Testumgebungen und Testfelder werden in Abschnitt 2.3 vorgestellt. Abschließend werden bestehende Architekturen von ADS mit einigen Standardbegriffen vorgestellt (siehe Abschnitt 2.4).

2.1. Hochautomatisierte Fahrzeuge und Fahrsysteme

Unter dem Oberbegriffe des autonomen Fahrens versteht man die Fähigkeit eines Fahrzeugs, Fahraufgaben selbstständig und ohne menschliches Eingreifen zu übernehmen (Maurer et al. 2015, S.2ff). Solche Systeme basieren auf der Integration von Sensorik, Datenverarbeitung und Aktorik, wodurch sie der Kategorie der CPSs zugeordnet werden können. Ein CPS definiert ein System, in dem physische Komponenten mit virtuellen Komponenten verschmelzen (Acatech 2012, S.5). Diese Produkte sind Teil der vernetzten Welt und laufen auf eingebetteter Hardware. Sensoren verarbeiten Daten aus der physischen Welt, Berechnungen werden von Softwarekomponenten durchgeführt und Aktoren wirken auf die physische Welt ein (Acatech 2012, S.5f).

Zur Klassifizierung und einer Begriffsabgrenzung des autonomen Fahrens definiert der Standard J3016 der Society of Autonomous Engineers (SAE) sechs Automatisierungsstufen (Level 0 bis 5), die von keiner bis zur vollständigen Automatisierung der Fahrzeugführung reichen (SAE 2021). Tabelle 2.1 stellt diese Stufen des J3016-Standards für die Klassifizierung von Automatisierungssystemen in Straßenfahrzeugen dar.

Tabelle 2.1.: SAE Automatisierungsstufen nach J3016, aus (SAE 2021) übersetzt

Stufe	Automatisierung	Beschreibung
0	Keine	Der Fahrer übernimmt die Fahraufgabe, unterstützt durch Systeme wie ABS oder ESP.
1	Assistierter Modus	Das System übernimmt Lenkung oder Beschleunigung/-Verzögerung; der Fahrer bleibt verantwortlich.
2	Teilautomatisierung	Das System übernimmt Lenkung und Beschleunigung/-Verzögerung; der Fahrer bleibt verantwortlich.
3	Bedingte Automatisierung	Das System übernimmt alle Fahraufgaben; der Fahrer muss bei Bedarf eingreifen.
4	Hochautomatisierung	Das System übernimmt alle Fahraufgaben; kein Eingreifen des Fahrers nötig.
5	Vollautomatisierung	Das System übernimmt alle Fahraufgaben unter allen Bedingungen.

Eine weitere Konkretisierung des CPS-Begriffs findet sich in den Konzepten von ADS und Advanced Driver Assistance System (ADAS) wieder. Eine einheitliche Definition und Abgrenzung dieser beiden Begriffe ist in der Literatur jedoch nicht eindeutig festgelegt. Antony und Whenish (2021) definieren ADAS als „Gruppe von Fahrzeugtechnologien, die den Fahrer rechtzeitig vor riskanten oder gefährlichen Situationen warnen, um Unfälle zu vermeiden“ (übersetzt aus Antony und Whenish 2021, S.165). ADAS-Technologien gelten als Vorläufer von ADSs und werden den SAE-Levels 1 und 2 zugeordnet, während Systeme ab Level 3 allgemein als ADS gelten (Antony und Whenish 2021; Ding et al. 2024). Ein Beispiel für ein aktuelles ADS ist der „Drive Pilot“ von Mercedes. Zum Zeitpunkt dieser Arbeit ist der „Drive Pilot“ das einzige zugelassene SAE Level 3 System[1]. Die Implementierung solcher Systeme basiert häufig auf einer modularen Struktur, die wesentliche Komponenten wie Perzeption, Planung und Steuerung umfasst. Eine Erläuterung und Details dieses Schemas folgen im nächsten Abschnitt.

2.1.1. Modell eines hochautomatisierten Fahrsystems

Die Unterteilung eines ADS kann mithilfe des Schemas `Sense-Plan-Act` (Perception-Planning-Control (PPC)) erfolgen. Dieses Schema basiert auf den Aufgaben des Fahrers im klassischen Fahrzeug, der die Brücke zwischen der Beobachtung der Umgebung und der Steuerung von Lenkrad und Pedalen darstellt (Jüstel und Stöckmann 2018). Abbildung 2.1 zeigt das Schema

1. https://www.kfz-betrieb.vogel.de/level-3-fahrzeuge-duerfen-tempo-130-fahren-a-c5bc18087a6de7d2e0b5351f461de6b5/ (Abgerufen am 13.10.2024)

als Blockdiagramm. Es beschreibt den grundsätzlichen Ablauf, wie ein ADS auf die Umwelt

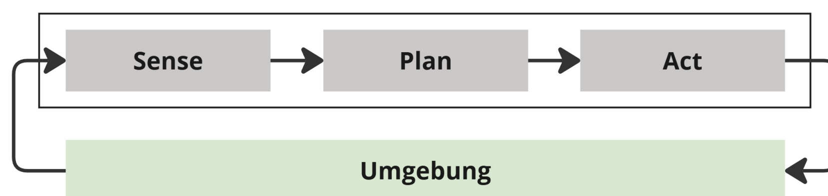


Abbildung 2.1.: Blockdiagramm des Sense-Plan-Act-Schemas mit Verbindung zum Umfeld

reagiert. In der `Sense`-Phase erfasst das System die Umgebungsbedingungen mithilfe von Sensoren wie Kameras, Lidar oder Radar. Diese Informationen umfassen unter anderem die Position anderer Objekte, Fahrzeuge oder Verkehrsschilder. Auf Basis dieser Daten erfolgt in der `Plan`-Phase die Auswertung und Entscheidungsfindung. Das ADS analysiert die Situation und entwickelt eine Handlungsempfehlung, beispielsweise ein Lenkwinkel zum Halten der Spur oder ein bestimmtes Bremsverhalten. In der `Act`-Phase wird der geplante Entscheidungsprozess in physische Aktionen umgesetzt. Die Steuerbefehle für Lenkung, Beschleunigung und Bremsen werden an die entsprechenden Aktuatoren des Fahrzeugs übermittelt, um die Handlung durchzuführen (Srivastava 2019; Jüstel und Stöckmann 2018). Durch die im Abbildung 2.1 dargestellte Rückkopplungsschleife zwischen Umgebung und Sense/Act ist eine kontinuierliche Anpassung der Berechnungen möglich. Die Wahrnehmung der Umgebung umfasst neben den verschiedenen Sensoren wie Kamera, Radar oder LiDAR auch die Kommunikation mit anderen Verkehrsteilnehmern, z. B. über Vehicle-to-Everything (V2X). Mit diesen Informationen ist es möglich, die „Sichtweite“ von ADS zu erweitern, indem ein Datenaustausch zwischen den Fahrzeugen stattfindet (Stellwagen 2024, S.27).

Allerdings können nicht alle Teile und Funktionen des ADS im PPC-Schema abgebildet werden. Der Trend zu maschinellem Lernen und künstlicher Intelligenz hat dazu geführt, dass das Schema in einigen Arbeiten um eine vierte Phase `Learn` erweitert wurde (Oddi et al. 2020). Diese Komponente führt einen kontinuierlichen Rückkopplungsprozess ein, der sowohl die Wahrnehmung nach der Sense-Phase als auch die Handlungsbewertung nach der Act-Phase verbessert.

2.1.2. Software-Defined Vehicle

Der Begriff des SDVs beschreibt die Entwicklung von einem hardware- bzw. mechanisch gesteuerten Fahrzeug hin zu einem elektronisch gesteuerten und softwareabhängigen Fahrzeug. Dabei wird in der Literatur eine Analogie zwischen einem SDV und einem Smartphone (auf Rädern) gesehen (Slama et al. 2023, S.2) (Lu und Shi 2024, S.1f). Dies bezieht sich häufig auf den Einsatz großer interaktive Bildschirme im Fahrzeug, eine reibungslose Konnektivität und

häufige Software-Updates. Gleichzeitig wird jedoch betont, dass die Benutzerfreundlichkeit und Funktionalität moderner Fahrzeuge oft hinter den Erwartungen der Verbraucher zurückbleibt (Slama et al. 2023, S.1). Diese Abweichung führt zu Fragen wie: „Warum kann mein 50.000-Dollar-Auto nicht die gleichen Aufgaben erfüllen wie mein 300-Dollar-Smartphone?“ (übersetzt aus Slama et al. 2023, S.1) Ein SDV adressiert diese Herausforderung, indem es ein vollständig programmierbares Fahrzeug darstellt, dessen Funktionen innerhalb weniger Monate entwickelt und über drahtlose Updates bereitgestellt werden können.

Einer der Kernaspekte des SDV ist die Möglichkeit diese drahtlosen Updates, auch OTA-Updates genannt, durchführen zu können. Damit können Updates, beispielsweise für das Infotainment, aber zukünftig auch für Funktionen des ADS, durchgeführt werden, ohne dass das Fahrzeug zur Werkstatt gebracht werden muss. Durch die bereits beschriebene Ablösung von hardwarezentrierten Entwicklungsparadigmen ermöglicht diese Architektur die Entwicklung und Integration neuer Funktionen unabhängig von der physischen Hardware. Dadurch wird nicht nur die Entwicklungszeit verkürzt, sondern auch die Innovationsgeschwindigkeit des gesamten Entwicklungszyklus erhöht (Slama et al. 2023, S.42f). Zusätzlich kann auf Technologien und Frameworks aus der klassischen Softwareentwicklung zugegriffen werden, wie einer kontinuierlichen Integration und Auslieferung (engl. Continuous Integration/Continuous Delivery (CI/CD)). Der folgende Abschnitt erläutert diese Konzepte und wie sie die Entwicklung von ADS unterstützen.

2.2. Software- und Systementwicklung

Wie breits in der Problemstellung erläutert, stellt die Software- und Systementwicklung von ADSs eine große Herausforderung dar. Im folgenden Abschnitt wird die Entwicklung dieser Systeme anhand des V-Modells in Kombination mit virtuellen Testmethoden und dem Einsatz von DevOps vorgestellt.

2.2.1. Entwicklung autonomer Fahrzeugsysteme

Die Entwicklung von ADSs stellt aufgrund der Sicherheitsaspekte eines CPS eine besondere Herausforderung dar. Um diesen Herausforderungen bei der Erstellung und Wartung dieser Systeme gerecht zu werden, erfolgt die Entwicklung anhand strukturierter Vorgehens- bzw. Phasenmodelle (Eigner et al. 2014, S.42f). Ein zentraler Aspekt dieser Entwicklung ist die V+V dieser sicherheitskritischen Systeme. Die Verifikation stellt sicher, dass die Funktionen gemäß den Anforderungen korrekt implementiert sind. Die Validierung untersucht, ob die richtige Lösung ausgewählt wurde und die Bedürfnisse des Nutzers erfüllt werden (Lunkeit und Zimmer 2021, S.304f). Wie bereits erläutert, werden einige dieser Herausforderungen durch Ansätze der SDV adressiert. Es gilt, Konzepte weiterzuentwickeln, die sich bereits in

der klassischen Softwareentwicklung bewährt haben. Ein klassisches Vorgehensmodell, das auch in der Automobilindustrie eingesetzt wird, ist das V-Modell als modifizierte Version des Wasserfallmodells. Es eignet sich aufgrund der definierten Testschritte besonders für Systeme mit hohen Zuverlässigkeits- und Sicherheitsanforderungen (Eigner et al. 2014, S.88f). Das V-Modell hat zusätzlich an Bedeutung gewonnen, da es zur Grundlage der ISO 26262 (ISO 2018), der internationalen Norm für die funktionale Sicherheit von Straßenfahrzeugen, erhoben wurde (Koopman und Wagner 2016). Die im Jahr 2021 vorgestellte Verein Deutscher Ingenieure (VDI) (2021) Richtlinie zur „Entwicklung mechatronischer und cyber-physischer Systeme" basiert ebenfalls auf dem V-Modell.

Das V-Modell unterteilt die Entwicklung eines solchen Systems in verschiedene Phasen (Phasenmodell), wobei ein neuer Prozessschritt erst begonnen werden kann, wenn der vorhergehende abgeschlossen ist. Dabei erfolgt auf dem linken absteigenden Ast des V-Modells die Zerlegung und Spezifikation des zu entwickelnden Systems. Auf dem aufsteigenden rechten Ast werden Integrations- und Testschritte zugeordnet (Hagen 2020, S.32f). Abbildung 2.2 zeigt eine Darstellung des Modells, in der auch das namensgebende „V" zu erkennen ist. Das V-Modell lässt sich, wie in Abbildung 2.2 zu erkennen, unterteilen in Systementwicklung

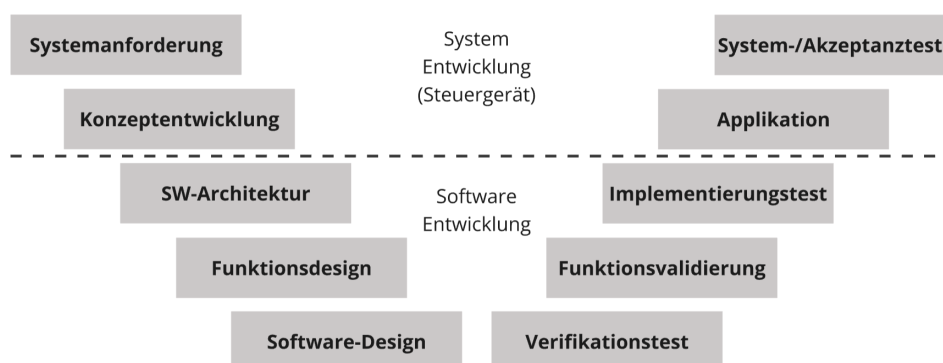


Abbildung 2.2.: V-Modell, nach Hagen (2020, S.33)

(engl. „systems engineering") und Softwareentwicklung (engl. „software engineering").

2.2.2. XiL-Ansatz

Wie bereits erwähnt, ist die Entwicklung und Validierung sicherheitsrelevanter Systeme (CPS) eng mit dem Einsatz von Simulation in verschiedenen Ausprägungen verbunden. Häufig stehen Prototypen oder die endgültige Hardware für frühzeitige Tests nicht zur Verfügung. Durch die Simulation und Emulation verschiedener Software-/Hardwaresysteme ist es möglich, unterschiedliche Systemarchitekturen zu testen und die Anzahl der benötigten Prototypen zu reduzieren (Magosi et al. 2022). In der Literatur wird hierzu häufig der Ansatz des X-in-the-Loop (XiL) verwendet, der das dynamische Testen in verschiedenen virtuellen

Testumgebungen beschreibt. Dabei wird das X durch die Komponente ersetzt, die sich im Regelkreis befindet (Riedmaier et al. 2018). Im Folgenden werden die gängigsten Ansätze kurz beschrieben:

Model-in-the-Loop (MiL) wird in frühen Entwicklungsstadien, wie der Konzeptentwicklung, genutzt. Ein Modell kann in diesem Kontext eine abstrahierte Repräsentation des Systems sein, die beispielsweise dessen Funktionen, Verhalten oder Schnittstellen beschreibt. Dabei wird das Modell des zu entwickelnden Systems in einer simulationsbasierten Umgebung getestet, um erste Validierungen durchzuführen und potenzielle Schwächen im Design frühzeitig zu erkennen (Riedmaier et al. 2018).

Software-in-the-Loop (SiL) ermöglicht die Integration und das Testen der Software in einer simulationsbasierten Umgebung. Die Software wird dabei auf einem virtuellen System ausgeführt, um deren Funktionalität unabhängig von der Hardware zu überprüfen (Jaikamal 2009).

Processor-in-the-Loop (PiL) ist ein Testansatz, bei dem die Software auf einem realen Prozessor ausgeführt wird, der in die Simulationsumgebung eingebunden ist. Dies ermöglicht eine Überprüfung der Software in einer nahezu realen Umgebung unter Berücksichtigung von Timing- und Ressourcenbeschränkungen (Mina et al. 2016).

Hardware-in-the-Loop (HiL) verbindet die zu testende Software mit realer Hardware, simulierten Umgebungsmodellen und erweitert damit den PiL-Ansatz. Die reale Hardware, auch als Electronic Control Unit (ECU) bezeichnet, empfängt Signale auf dem im Fahrzeug verwendeten Bussystem, die verarbeitet werden. Basierend auf den Algorithmen und der Software werden die berechneten Signale an die Simulation zurückgesendet. Dies ermöglicht die Validierung der Hard- und Software in Echtzeit (Jaikamal 2009).

Vehicle-in-the-Loop (ViL) erweitert den HiL-Ansatz, indem ein reales Fahrzeug in die Testumgebung integriert wird. Dies ermöglicht die Validierung und Verifizierung von Systemen unter realitätsnahen Bedingungen, während gleichzeitig die Sicherheit durch simulierte Umgebungsbedingungen gewährleistet bleibt (Riedmaier et al. 2018).

Mit jeder weiteren Stufe dieser Testansätze steigt die Realitätsnähe der Tests, was jedoch die Komplexität der Simulation/Emulation und den Testaufwand erhöht. Um diese XiL-Ansätze auf die Domäne Automotive zu übertragen, beschreibt Riedmaier et al. (2018) die Klassifizierung der Komponenten: Fahrzeug, Funktion, Sensorik und Szenarien als virtuell oder real.

Diese Klassifizierung orientiert sich an der Verarbeitungskette eines realen Tests und ist in Abbildung 2.3 zu erkennen. Es ist zu erkennen, dass bei einem MiL- oder SiL-Ansatz jede

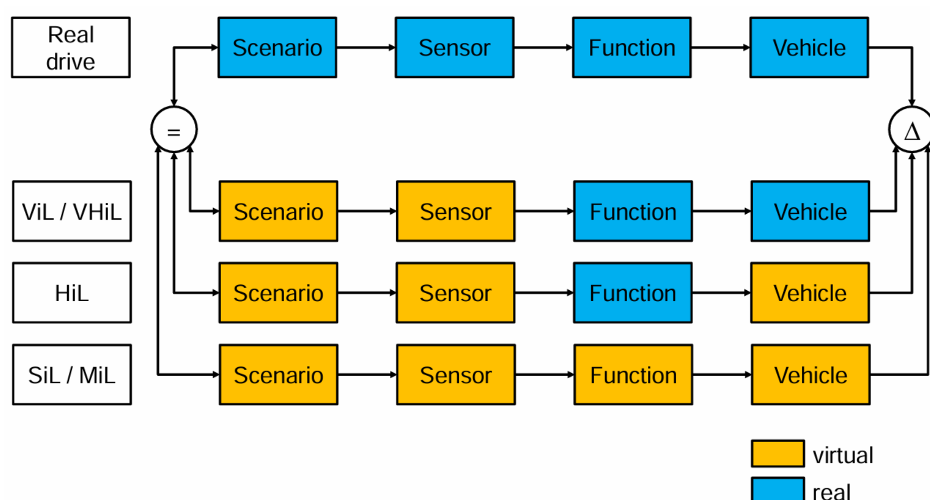


Abbildung 2.3.: Blockdiagramm für eine allgemeine Methodik der Validierung von XiL-Ansätzen (Riedmaier et al. 2018)

dieser Komponenten rein virtuell existiert. In einem HiL-Ansatz wird die hochautomatisierte Fahrfunktion (beschrieben als `Function`) auf realer Hardware ausgeführt. In einem ViL-Ansatz folgt zusätzlich der Wechsel des Fahrzeugs von virtuell zu real. Um diese Ansätze mit realen Tests vergleichen zu können, benötigen alle Testinstanzen die gleichen Eingabedaten (=). Je nach Genauigkeit der Simulationsmodelle ergeben sich Unterschiede in den Ausgabedaten (Δ) (Riedmaier et al. 2018). Die genannten Testmethoden des XiL-Ansatzes lassen sich mit dem V-Modell kombinieren. Magosi et al. (2022) zeigen in ihrer Arbeit, in welcher Phase des V-Modells, welcher Ansatz genutzt werden kann. Zusätzlich wird das Modell auf die Entwicklung von ADS spezifiziert (siehe Abbildung 2.4).

Eine weitere Methode aus der klassischen Softwareentwicklung ist DevOps. Dabei werden Entwicklung (engl. „Development (Dev)“) und der Betrieb (engl. „Operations (Ops)“) miteinander verknüpft. DevOps nutzt und erweitert einige Aspekte der agilen Softwareentwicklung, wird aber nicht als eigenständiges Vorgehensmodell betrachtet (Halstenberg et al. 2020, S.14). Für die Anwendung dieser Ansätze im Automobilbereich sind aufgrund der sicherheitskritischen Funktionen einige Anpassungen erforderlich. Kallweit et al. (2020) zeigt einige Ansätze, wie das V-Modell in Kombination mit DevOps auch im Automobilbereich eingesetzt werden kann. Dabei werden auch die verschiedenen Testmethoden (SiL, HiL, etc.) auf der rechten Seite des V-Modells dargestellt. Zentraler Aspekt dieser Modellkombination ist die Verwendung einer zentralen Datenbank, in der verschiedene Szenarien, Modelle, Sensoren und Standards gespeichert sind. Ebenso werden über ein „Continuous Monitoring“ alle

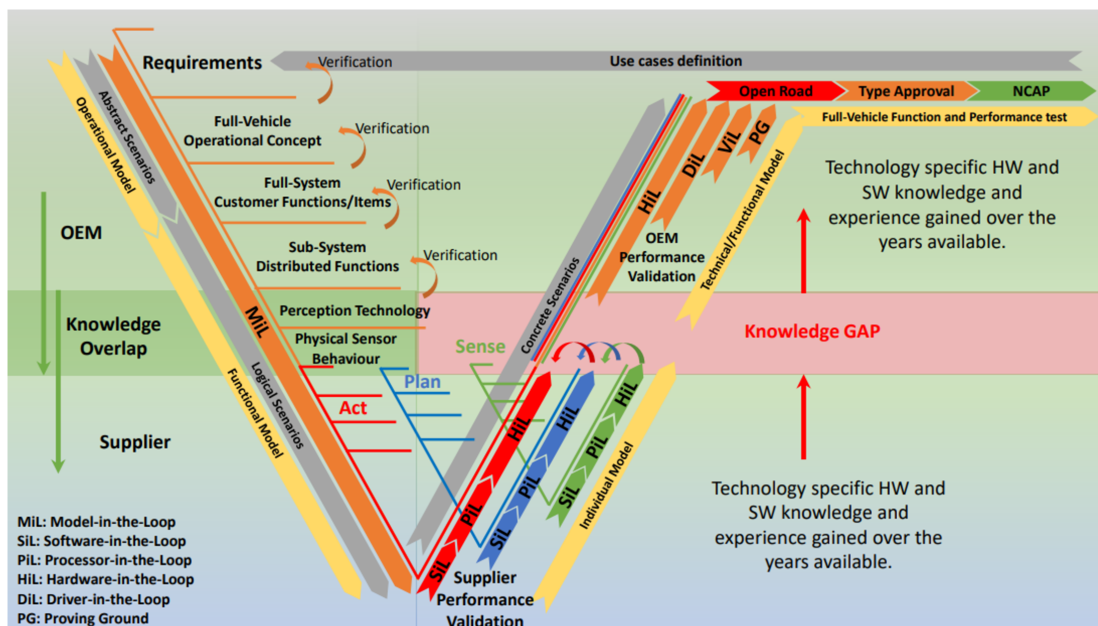


Abbildung 2.4.: V-Modell in Kombination mit XiL-Ansätzen für die Entwicklung von ADS (Magosi et al. 2022)

Design- und Testphasen überwacht und ebenfalls in der Datenbank gespeichert. Diese Daten ermöglichen den Einsatz von Automatisierungswerkzeugen, um Prozesse zu beschleunigen. Ein klassisches Werkzeug in DevOps ist der Einsatz einer CI/CD-Pipeline. Diese ermöglicht es, die Entwicklung, das Testen und die Bereitstellung neuer Versionen zu automatisieren (Nouri et al. 2022). Eine Verbindung zu einem Versionskontrollsystem könnte beispielsweise automatische Testläufe ermöglichen, um neue Versionen eines ADSs in den verschiedenen Testmethoden zu testen und zu analysieren.

2.3. Einsatz von Testumgebungen/Testfeldern

Um die im V-Modell definierten Tests während der Entwicklung eines ADS durchführen zu können, werden spezielle Testumgebungen/Testfelder (engl. „testbed“) benötigt. Der Standard IEEE 610 beschreibt den Begriff als „eine Umgebung, die Hardware, Instrumente, Simulatoren, Software-Tools und andere unterstützende Elemente, die für die Durchführung eines Tests benötigt werden“ (übersetzt aus IEEE 1990, S.74). Die Begriffe Testumgebung und Testfeld werden im Deutschen oft synonym verwendet, unterscheiden sich aber in einigen Aspekten. Eine Testumgebung bezeichnet eine technische Infrastruktur, in der Tests durchgeführt werden können (Hardware, Software, Simulatoren, Schnittstellen, Netzwerk). Dies ermöglicht die Verifikation einzelner Komponenten oder Funktionen unter kontrollier-

ten Bedingungen. Ein Testfeld hingegen umfasst physische oder virtuelle Orte, an denen das System unter realistischen Bedingungen getestet werden kann, um die Validierung des Gesamtsystems zu ermöglichen (Maurer et al. 2015, S.460f). In einer Testumgebung und in Testfeldern werden Prototypen in frühen Entwicklungsstadien simulativ getestet. Erreicht die Entwicklung des zu testenden Systems einen gewissen Reifegrad, werden die virtuellen Modelle, Simulationen oder Emulationen durch physische Komponenten ersetzt. Dabei ermöglicht eine klar definierte Testinfrastruktur, insbesondere im Rahmen einer Testumgebung, die nahtlose Integration und Verifikation des zu testenden Systems (Brinkmann 2018, S.33f).

2.3.1. Open-Loop-Tests und Closed-Loop-Tests

Hochautomatisierte Fahrfunktionen (ADS) lassen sich allgemein als Systeme beschreiben, die aus definierten Eingangssignalen bestimmte Ausgangssignale berechnen. Da die Ausgangssignale direkt auf die Umwelt wirken und damit auch die Eingangssignale beeinflussen, kann dieses Verhalten als Regelkreis (engl. „loop“) beschrieben werden. Beim Testen dieser Systeme wird zwischen Open-Loop-Tests und Closed-Loop-Tests unterschieden. Abbildung 2.5 stellt die Unterscheidung dieser beiden Verfahren als Blockdiagramm dar. Bei Open-Loop-Tests werden die Eingangsdaten direkt in das zu testende System (System Under Test (SUT)), in diesem Fall das ADS, eingespeist. Die Ausgangssignale werden mit der Testreferenz verglichen. Bei Closed-Loop-Tests wird ein Modell der Regelstrecke hinzugefügt, das die Eingangssignale empfängt und als Rückkopplung in das SUT zurückführt. Der Vergleich erfolgt zwischen dem Ausgang des Regelkreises, dem ADS, und der Referenz (Baumann 2006).

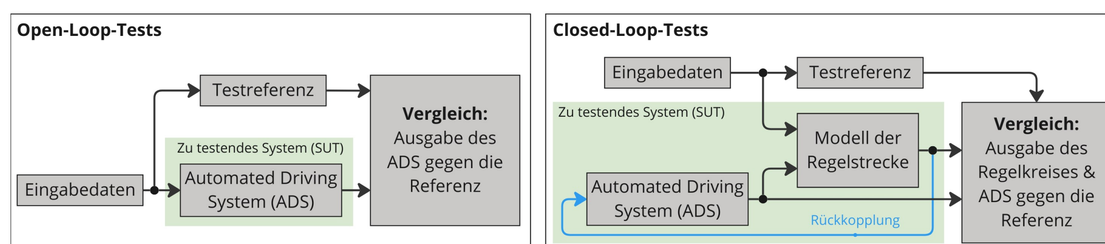


Abbildung 2.5.: Blockschaltbild für Open-Loop-Test und Closed-Loop-Test, nach Baumann (2006) und Brinkmann (2018)

2.3.2. Referenzarchitektur für Testfelder

Das Forschungsprojekt `ENABLE-S3`[1] stellt in seiner Ergebniszusammenfassung (Nickovic et al. 2017) eine generische Testfeldarchitektur für automatisierte CPS vor. Diese Architektur

1. https://www.offis.de/offis/projekt/enable-s3.html (Abgerufen am 20.10.2024)

ermöglicht das bereit erwähnte Austauschen von virtuellen durch physische Komponenten. Eine vollständige Erläuterung der Referenzarchitektur übersteigt den Umfang dieser Arbeit. Diese ist in (Nickovic et al. 2017) und Abbildung 2.6 finden.

Nickovic et al. (2017) unterteilen die Architektur in drei Teilsysteme: das V+V-Management (auch Testdatenmanagement genannt), das Testmanagement und die Testplattform. Das V+V-Management umfasst alle Komponenten, die die Verifikation und Validierung unterstützen. Dazu gehören Simulations-/Messergebnisse, Szenariengenerierung sowie Sicherheitsanweisungen und Testberichte. Das Testmanagement beinhaltet Komponenten zur Durchführung einzelner Testläufe, wie die Durchführung von Messungen und die Testinitialisierung/-automatisierung. Die Testplattform stellt Elemente dar, die zur Durchführung eines Tests benötigt werden. Im Zentrum der Testplattform steht das SUT, in diesem Fall das automatisierte CPS. Die dort ebenfalls vorhandene Umgebung und Infrastruktur übergibt Eingabesignale an das SUT, das auf dieser Basis Befehle über die Systemdynamik ausführt. Die in Abbildung 2.6 erkennbare Rückkopplung zwischen Systemdynamik und Umgebung zeigt, dass die Referenzarchitektur Closed-Loop-Tests unterstützt (Nickovic et al. 2017). Ziel des Projektes `ENABLE-S3` ist Verknüpfung der generischen Referenzarchitektur

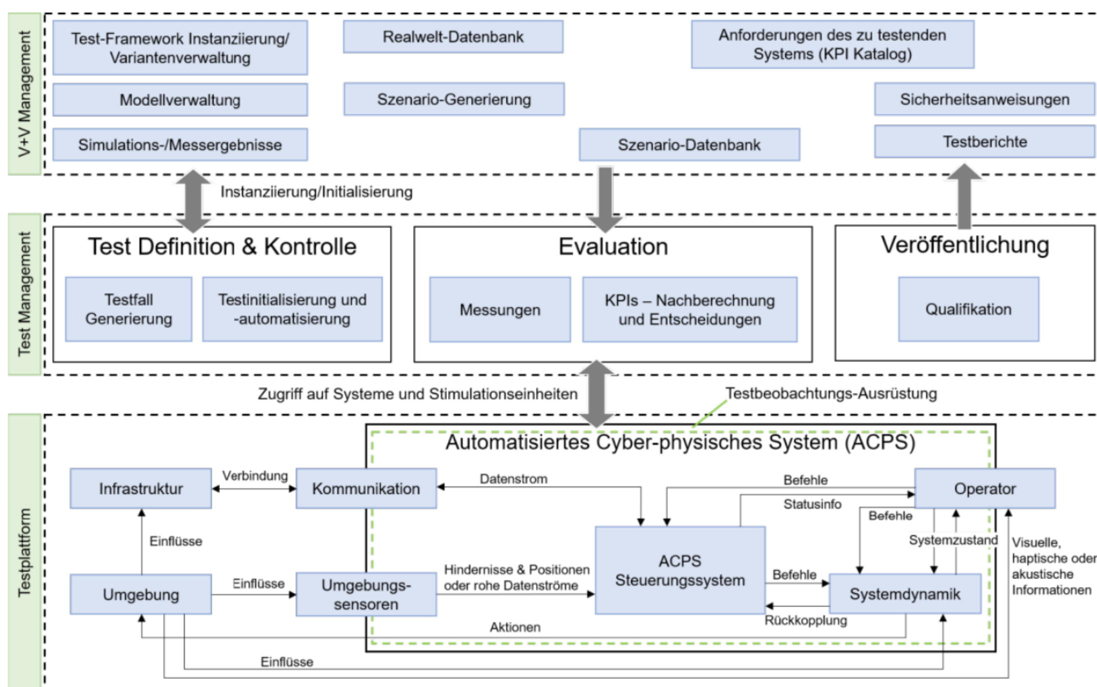


Abbildung 2.6.: Referenzarchitektur eines generischen Testfeldes für automatisierte CPS (Nickovic et al. 2017)

mit wiederverwendbaren Technologien, um eine anwendungsspezifische Instanziierung der Architektur zu ermöglichen. Im weiteren Verlauf des Berichts wird der Use-Case einer „Highway

Pilot Function“ unter Verwendung der Referenzarchitektur gezeigt. Anhang D stellt diesen Anwendungsfall und die Architektur dar. Wie bereits beim Begriff Testfeld erläutert, beschreiben die Autoren, dass der Ersatz von virtuellen durch physische Komponenten in der Architektur unterstützt wird. Im Use-Case des Highway-Piloten besteht die Möglichkeit, die Fahrzeugdynamik virtuell oder physisch über einen Rollenstand abzubilden (Nickovic et al. 2017).

2.3.3. Sensorsimulation und Komplexitäten

Unter Verwendung des erläuterten PPC-Schemas (siehe Unterabschnitt 2.1.1) gehört die Entwicklung der Perzeption (Sense) zu den anspruchsvollsten Bereichen eines ADS. Dies liegt an der Vielfalt und Dynamik der Umgebung sowie an den hohen Anforderungen an die Genauigkeit und Robustheit der Wahrnehmung (Reway et al. 2018). Um eine modulare Entwicklung zu ermöglichen, kann es von Vorteil sein, verschiedene Sensormodelle und Implementierungen zu verwenden. Sievers et al. (2018) stellen dabei eine Verbindung zwischen Realitätsnähe und Sensorkomplexität her (Sensorsimulation). Diese geht von einem idealen Sensor aus, der auf Grundwahrheiten (engl. „Ground-Truth“) basiert. Dieser gibt z. B. direkt eine Objektliste zurück. Eine Objektliste enthält dabei die Ground-Truth-Daten aller Objekte, wie z. B. Position und Orientierung (Bounding Box), Beschleunigung und Abstand zum Fahrzeug. Darauf aufbauend können probabilistische Sensormodelle verwendet

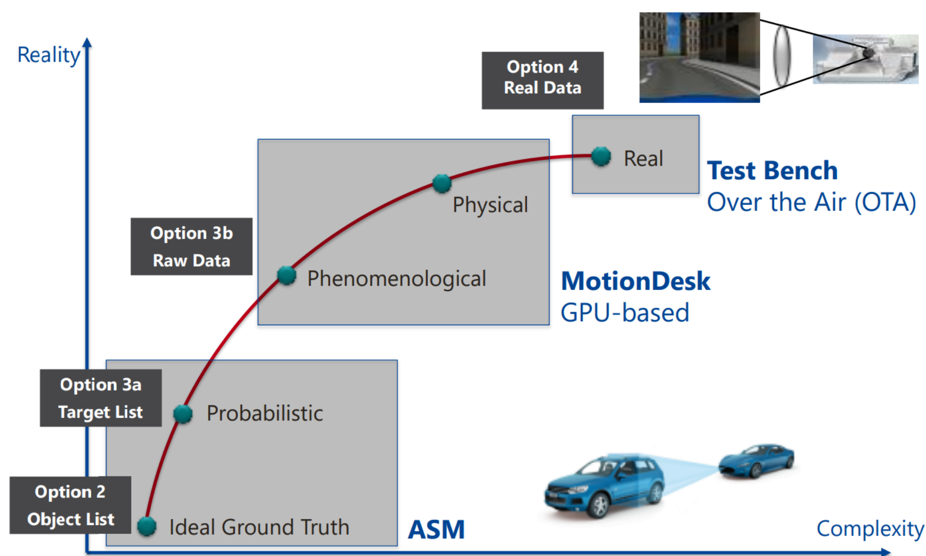


Abbildung 2.7.: Verschiedene Level der Sensorsimulation, die als Graph zwischen den Achsen Realitätstreue und Komplexität aufgetragen sind (Sievers et al. 2018)

werden, die die gesamte Objektliste auf bestimmte Ziele reduzieren. Beispielsweise nur für

Objekte, die sich im Sichtfeld des Sensors befinden und nicht durch andere Objekte verdeckt werden. Zur weiteren Erhöhung der Realitätsnähe können Rohdaten des simulierten Sensors verwendet werden, die von 3D-Simulatoren erzeugt werden. Im letzten Schritt wird die Verwendung eines Testfeldes und eines OTA-Sensors beschrieben. Dabei wird die 3D-Simulation der Sensordaten auf einem Monitor visualisiert, der den realen Kamerasensor stimuliert. Diese Methode ermöglicht eine sehr hohe Realitätsnähe, erfordert aber auch eine hohe Komplexität. Diese verschiedenen Schritte können bei der Entwicklung eines ADS genutzt werden, um eine inkrementelle Entwicklung zu ermöglichen oder die Perzeptionskomponente durch Mockups zu ersetzen (Sievers et al. 2018; Feilhauer et al. 2016).

2.4. Architekturen von automatisierten Fahrzeugen

Eine Fahrzeugarchitektur bildet die Grundlage, auf der ADS entwickelt, implementiert und betrieben werden. Der Begriff Architektur kann je nach Kontext unterschiedliche Dinge widerspiegeln. Im Fahrzeugbereich beschreibt eine Architektur die Gesamtheit der Hard- und Softwarekomponenten sowie deren Zusammenspiel (Behere und Törngren 2015). Dabei gibt sie die Rahmenbedingungen für die Anordnung der physischen und virtuellen Komponenten im Fahrzeug vor. Die in der Literatur häufig verwendete Elektrische/Elektronische (E/E)-Architektur beschreibt die Struktur der Fahrzeugelektrik und -elektronik (Streichert und Traub 2012, S.15f). Diese muss den hohen Anforderungen des CPS an Sicherheit, Skalierbarkeit und Effizienz gerecht werden. Sie muss die elektronischen Komponenten, wie Sensoren, Aktoren, Steuergeräte über verschiedene Kommunikationsnetzwerke miteinander verbinden und die Kommunikation sicherstellen (Frigerio et al. 2021). Dabei wird zwischen zentralisierten und dezentralisierten Ansätzen unterschieden. Bei einer dezentralen E/E-Architektur werden Funktionen auf verschiedene Steuergeräte verteilt, die über Bussysteme miteinander kommunizieren. Bei einer zentralen Architektur werden Berechnungen und Entscheidungen auf einem leistungsfähigen Steuergerät durchgeführt. Frigerio et al. (2021) stellen verschiedene Architekturtopologien (domänenbasiert, zonenbasiert, etc.) und Kombinationen vor.

Um Innovationen und steigende Anforderungen bei gleichzeitiger Reduktion der Komplexität dieser Systeme zu ermöglichen, ist eine Zentralisierung notwendig (Benckendorff et al. 2019, S.1191). Benckendorff et al. (2019) beschreiben eine Roadmap für E/E-Architekturen, die sich von modular über domänenübergreifend hin zu fahrzeugzentral entwickeln. Ein Beispiel für einen solchen Wandel ist die Verwendung spezifischer Sensor-ECU. In einer dezentralen Architektur ist der Sensor (z. B. Kamerasensor) mit einem Steuergerät ausgestattet, das die Berechnung der Kamerabilder (Objekterkennung) durchführt. In einer zentralen Architektur werden die Kameradaten vom Sensor über ein Netzwerk zum zentralen Steuergerät gesendet, wo die Berechnungen durchgeführt werden. Es lässt sich erkennen, dass beide

Ansätze unterschiedliche Anforderungen voraussetzen (Baic et al. 2018; Benckendorff et al. 2019).

Wie die E/E-Architektur kann auch die Software-Architektur mit unterschiedlichen Ansätzen entwickelt werden. Ein Beispiel aus der klassischen Softwareentwicklung ist eine monolithische oder mikroserviceorientierte Architektur. „Microservices sind unabhängig deploybare Services, die rund um eine Businessdomäne modelliert wurden. Sie kommunizieren untereinander über das Netzwerk und bieten als Architektur viele Möglichkeiten, Probleme zu lösen [...]"(Newman und Demmig 2020, S.1). Bei einer monolithischen Architektur wird die Software als vollständige, in sich geschlossene Einheit erstellt (Harris 2017). Beide Ansätze haben Vor- und Nachteile hinsichtlich der Bereitstellung, Wartung und Testbarkeit der Software. Beide Ansätze werden auch in der Automobilindustrie und der damit verbundenen Forschung eingesetzt. Einige dieser Ansätze werden in den verwandten Arbeiten unter Kapitel 5 vorgestellt.

2.4.1. AUTOSAR Adaptive Plattform

Der Begriff Automotive Open System Architecture (AUTOSAR) beschreibt ein Konsortium aus Automobilherstellern, Zulieferern, Toolherstellern, eine standardisierte Softwarearchitektur und Schnittstellen für die Entwicklung (Gliwa 2021, S.317). Dabei verfolgt AUTOSAR das Ziel, eine offene standardisierte Softwarearchitektur zu etablieren. Zu unterscheiden sind die Plattformen Classic und Adaptive, die beide auf der AUTOSAR-Foundation basieren. Die Classic-Plattform[1] wurde 2002 entwickelt, um tief eingebettete dezentrale Systeme und Steuergeräte über das Bussystem Controller Area Network (CAN) zu verbinden. AUTOSAR Adaptive[2] hingegen ist eine Plattform, die auf die Anforderungen moderner ADS ausgelegt ist (Fürst und Bechter 2016). Sie basiert auf einer serviceorientierten Kommunikation via Ethernet und ermöglicht damit große Datenpakete, wie Sensordaten, zu übertragen. Ziel ist die Entwicklung und Bereitstellung von rechenintensiven Funktionalitäten mit hohen Echtzeitanforderungen (Hong und Moon 2024). Als Protokoll der Middleware wird SOME/IP verwendet, das im Gegensatz zu CAN den gezielten Austausch von Datenpaketen zwischen zwei Funktionen ermöglicht, anstatt kleinere Pakete über das gesamte Netzwerk zu übertragen. Die Funktionen nutzen dabei einen Broker-Dienst, um mitzuteilen, welche Datenpakete sie benötigen. Diese Flexibilität in der Kommunikation ermöglicht das Hinzufügen von Anwendungen auch nach der Entwurfsphase (Schäuffele und Zurawka 2024, S.41).

1. https://www.autosar.org/standards/classic-platform (Abgerufen am 20.10.2024)
2. https://www.autosar.org/standards/adaptive-platform (Abgerufen am 20.10.2024)

2.4.2. Kommunikationsprotokolle, Frameworks und Standards

In diesem Kapitel wurden bereits einige Kommunikationsprotokolle, Frameworks und Standards vorgestellt. In diesem Abschnitt folgt eine kurze Erläuterung der in dieser Arbeit eingesetzten Protokolle.

Automotive Ethernet. Automotive Ethernet ist eine spezielle Weiterentwicklung von Ethernet, die auf dem Standard IEEE 802.3 basiert und für den Einsatz in Fahrzeugen optimiert wurde. Unterstützt werden verschiedene Datenübertragungsraten wie 100 MBit/s (100BASE-T1), 1 GBit/s (1000BASE-T1) oder Multi-Gigabit Automotive Ethernet (z. B. 2.5GBASE-T1, 5GBASE-T1, 10GBASE-T1)(IEEE 2020). Diese Standards wurden speziell für den Einsatz in der rauen Umgebung eines Fahrzeugs entwickelt, in der elektromagnetische Störungen eine große Rolle spielen. Um Material und Kosten zu sparen, verwendet Automotive Ethernet nur eine verdrillte Zweidrahtleitung (Single Twisted Pair) für die Datenübertragung. Um Ethernet um Echtzeitfunktionen zu erweitern, kann ein Time-Sensitive Networking (TSN) eingesetzt werden, das den Standard IEEE 802.1 erweitert. TSN bietet Mechanismen für eine deterministische Kommunikation, bei der garantierte Latenzzeiten und eine Synchronisation möglich sind (Farkas et al. 2018).

ROS und ROS2. Das Robot Operating System (ROS)[1] ist ein Open-Source-Framework für die Entwicklung von Robotik-Software. Es stellt Werkzeuge und Bibliotheken zur Verfügung, die die Kommunikation zwischen Prozessen und die einfache Integration von Sensoren und Aktoren ermöglichen. Durch die einfache Entwicklung wird dieses Framework häufig für die Implementierung von Prototypen, auch im Automobilbereich, eingesetzt. Unterstützt wird dies durch den modularen Aufbau und die Kommunikation über ein Publish-Subscribe-Muster. Die Weiterentwicklung ROS 2 basiert auf einer Kommunikation über Data Distribution Service (DDS). Dies ermöglicht Echtzeitanforderungen und die Quality of Service (QoS) Funktionen (Macenski et al. 2022).

OSI. Das Open-Simulation-Interface (OSI)[2] ist ein standardisiertes Schnittstellenformat für die Kommunikation zwischen Simulationstools und Softwarekomponenten im Kontext von ADSs. Dieser von Association for Standardization of Automation and Measuring Systems (ASAM) in 2017 vorgestellte öffentliche Standard ermöglicht eine Integration von automatisierten Fahrfunktionen und einer Vielzahl von Fahrsimulatoren. OSI organisiert die Datenübertragung in einer hierarchischen Struktur aus verschiedenen Nachrichtenklassen. Diese reichen von Basisklassen zur Beschreibung eines sich bewegenden Objekts (`osi3::BaseMoving`),

1. https://www.ros.org/ (Abgerufen am 20.10.2024)
2. https://www.asam.net/standards/detail/osi/ (Abgerufen am 05.12.2024)

bis hin zu komplexen Klassen für Sensordaten (`osi3::CameraSensorView`) oder Ground-Truth-Daten (`osi3::DetectedMovingObject`). In diesen Nachrichtenklassen können zusätzliche Meta-Informationen angegeben werden. OSI basiert auf dem vom Google entwickelten „Protocol Buffer“[1], auch `protobuf` genannt. Der in dieser Arbeit verwendete OSI-Standard befindet sich in der Version 3.7.0 „Jolly Jones“, welche am 03.04.2024 vorgestellt wurde (ASAM 2024c). Der Standard ist als Open-Source-Projekt auf GitHub[2] zu finden (ASAM 2024c).

OpenSCENARIO und OpenDRIVE OpenSCENARIO[3] wurde ebenfalls von ASAM entwickelt und ist ein offener Standard für die Definition und Beschreibung von Szenarien im Bereich der Fahrzeugsimulation. Er erlaubt die detaillierte Spezifikation von Verkehrssituationen (Manövern), Fahrzeugverhalten und Umgebungsparametern. Die Manöver enthalten dabei eine Beschreibung der Aktionen des Ego-Fahrzeugs (z. B. ein Spurwechsel) oder eine konkrete Trajektorie. Zusätzlich können Umgebungsparameter wie Wetterbedingungen oder Ampelverhalten vorgegeben werden. ASAM spezifiziert OpenSCENARIO in den Versionen Extensible Markup Language (XML) und Domain-Specific Language (DSL), die parallel entwickelt werden. Das XML-Schema enthält eine sehr konkrete Definition des Szenarios und ist für eine einfache maschinelle Verarbeitung optimiert. Das DSL-Schema ermöglicht eine Beschreibung des Szenarios auf einer höheren Ebene in einer für Entwickler optimierten Programmiersprache (ASAM 2024b).

OpenDRIVE[4] ist ein ASAM-Standard, um die Beschreibung von Straßeninfrastrukturen in der Fahrzeugsimulation zu standardisieren. Dieser verwendet ein XML-basiertes Format, um die Geometrie von Straßen, Fahrbahnen, Fahrspuren, Straßenmarkierungen und Verkehrszeichen zu beschreiben. Darüber hinaus ermöglicht es die Darstellung von Merkmalen entlang der Straße, wie etwa Ampelanlagen oder Baustellen. Diese Daten werden in einer OpenDRIVE-Datei gespeichert. Durch die Verwendung dieses offenen Formats können unterschiedliche Simulatoren miteinander kommunizieren und die Daten austauschen, was eine vereinfachte Integration und Interoperabilität von Simulationssystemen ermöglicht (ASAM 2024a).

1. https://protobuf.dev/ (Abgerufen am 24.01.2025)
2. https://github.com/OpenSimulationInterface/open-simulation-interface (Abgerufen am 07.01.2025)
3. https://www.asam.net/standards/detail/openscenario-xml/ (Abgerufen am 05.12.2024)
4. https://www.asam.net/standards/detail/opendrive/ (Abgerufen am 05.12.2024)

3. Use-Cases

Die folgenden Use-Cases geben einen Überblick über die Interaktionen zwischen verschiedenen Akteuren und dem in dieser Arbeit entwickelten System. Sie zeigen, wie eine modulare Fahrzeugarchitektur, bestehend aus verschiedenen Mikroservices, in unterschiedlichen Umgebungen (HiL, SiL) eingesetzt werden kann. Gleichzeitig zeigen die Use-Cases die Möglichkeiten der Simulations- und Testumgebung, Tests mit virtuellen und physischen Komponenten durchzuführen. Die Use-Cases sind aus der Sicht des Entwicklers formuliert, der die bereitgestellte Test- und Simulationsumgebung nutzt, um den Entwicklungsprozess von ADS (hier am Beispiel eines Lane-Followers als Subkomponente) zu verbessern. Jeder Use-Case enthält eine detaillierte Beschreibung der Systemakteure, der Auslöser, der Vorbedingungen, der Standardabläufe und der Nachbedingungen, um die Interaktionen klar zu definieren und die Grundlage für die Evaluation der entwickelten Systeme zu schaffen. Das Format der Use-Cases orientiert sich an Kulak und Guiney (2012) und Gomaa (2011).

3.1. Use-Case 1: Modulares Testen in der Test- und Simulationsumgebung

Beschreibung: Ein Entwicklerteam arbeitet an der Entwicklung eines Lane-Followers, der aus mehreren Mikroservices zur Fahrspurerkennung (mithilfe eines Kamerasensors) und zur Berechnung des notwendigen Lenkwinkels besteht. Die Entwickler möchten ihr System nicht erst nach Abschluss aller Entwicklungsarbeiten im realen Fahrzeug testen, sondern bereits während des Entwicklungsprozesses in der vorliegenden Testumgebung (Pham et al. 2024). Mithilfe der Testumgebung und der vordefinierten standardisierten Schnittstellen zum SUT kann der Entwickler sein System testen. Um den Entwicklungsprozess zu optimieren, können Eingabedaten aus einem breiten Spektrum verwendet werden: von präzisen Ground-Truth-Daten aus der Simulation bis hin zur OTA-Stimulation über eine reale Sensorik (Sievers et al. 2018; Schlager et al. 2020). Da die Genauigkeit der Wahrnehmung stark von der Qualität der Eingabedaten abhängt, können die Entwickler ihre Fahrfunktion mit immer realistischeren Eingabedaten testen. Über eine spezielle Softwarekomponente (im Folgenden Adaptermodul genannt) werden diese Eingabedaten in einem standardisierten Format übertragen, sodass das SUT für die jeweiligen Eingabedaten nicht angepasst werden muss. Zur Evaluierung des

Gesamtsystems verwenden die Entwickler Schnittstellen für Systemmonitore, die Metriken wie z. B. das Einhalten von Fahrspuren überprüfen. Diese Tests werden in vordefinierten Szenarien innerhalb der Simulation durchgeführt und die Ergebnisse über die vorhandenen Systemmonitore überwacht. Die vorhandenen Szenarien können von den Entwicklern durch eigene Szenarien im OpenSCENARIO-Format (ASAM 2024b) erweitert werden. Der konkrete Ablauf und Inhalt der Testumgebung wird in UC-2A beschrieben.
Systemakteure: Entwickler, Lane-Follower (SUT), Testumgebung
Auslöser: Bedarf des frühzeitigen Testens des Lane-Followers
Vorbedingung:

- SUT nutzt die definierten Schnittstellen als Ein- und Ausgabedaten
- Testumgebung und Systemmonitore sind funktionsbereit

Standardablauf:

1. Die Entwickler beginnen mit der Entwicklungsphase.
2. Die Entwickler können ihr SUT in Szenario A mit Ground-Truth-Eingabedaten testen.
3. Die Entwickler können ihr SUT in Szenario A mit realistischen OTA-Daten testen.
4. Die Entwickler können ihr SUT in Szenario B mit realistischen OTA-Daten testen.

Nachbedingung:

- Der Lane-Follower wurde mit unterschiedlichen Eingabedaten getestet.
- Die Ergebnisse der Systemmonitore (z. B. das Einhalten der Fahrspur) sind für den Entwickler sichtbar und dokumentiert.

3.2. Use-Case 2A: Migration und Test von Mikroservices auf realer Hardware

Beschreibung: Das Entwicklerteam plant, einzelne Mikroservices auf realer Hardware zu testen. Durch die Verwendung einer Ethernet-basierten Kommunikationsschnittstelle untereinander können einzelne Mikroservices des SUT von einer SiL-Umgebung in eine HiL-Umgebung portiert werden. Die Möglichkeit zur Evaluation des Gesamtsystems anhand von vordefinierten Szenarien und die Wahl der verschiedenen Eingabedaten bleibt dabei erhalten. Die Portierung einzelner Services in die jeweiligen Umgebungen wird von dem Entwickler über ein Bereitstellungsorchestrator manuell durchgeführt. Die HiL-Umgebung ermöglicht es den Entwicklern, ihr SUT auf der in der Realität verwendeten Fahrzeughardware zu testen.

Zusätzlich können die Entwickler über die Systemmonitoren überprüfen, ob die begrenzten Ressourcen der HiL-Umgebung die Funktion der Mikroservices beeinflussen.

Systemakteure: Entwickler, SUT im HiL, SUT im SiL, Bereitstellungsorchestrator, Adaptermodul, Simulation, Systemmonitore

Auslöser: Einzelne Mikroservices sollen auf realer Hardware getestet werden.

Vorbedingung:

- Das Gesamtsystem befindet sich im **SiL-Betriebszustand**, d. h. das SUT ist im SiL funktionsfähig und wird dort aktuell getestet.
- Die Kommunikation der Mikroservices erfolgt über ein standardisiertes Protokoll.
- Die HiL-Umgebung ist bereit und mit der Testumgebung verbunden.

Standardablauf: In Anhang B ist ein Sequenzdiagramm von einem beispielhaften Ablauf zu finden.

Nachbedingung:

- Der ausgewählte Mikroservice wurde erfolgreich auf die reale Hardware portiert.
- Das Gesamtsystem wurde mit der Kombination von HiL- und SiL-Komponenten getestet.

3.3. Use-Case 2B: Automatische Integration und Tests neuer Versionen des SUT

Beschreibung: Das Entwicklerteam entwickelt eine neue Version eines Mikroservices innerhalb des Lane-Followers. Um den Entwicklungsprozess zu verbessern und den Entwicklern direktes Feedback zu geben, sollen neue Versionen des SUT automatisch integriert und getestet werden können. Der vom Entwickler manuell konfigurierte Bereitstellungsorchestrator aus UC-2A wird in diesem Use-Case durch ein Versionskontrollsystem und einem Automatisierungsdienst ersetzt. Sobald eine neue Version eines Mikroservice bereitsteht, kann der Entwickler über das Versionskontrollsystem einen automatisierten Testdurchlauf starten. Der Automatisierungsdienst kompiliert das System und stellt die Services automatisch im HiL oder SiL bereit. Genau wie in UC-2A werden die Tests in den ausgewählten Szenarien durchlaufen. Bei einer fehlerhaften Version, etwa bei einem Fehlschlag wichtiger Tests, kann ein Rollback auf eine funktionsfähige Version durchgeführt werden.

Systemakteure: Entwickler, Versionskontrollsystem, Automatisierungsdienst, SiL, Adaptermodul, Simulation, Systemmonitor

Auslöser: Eine neue Version eines Mikroservices ist für Integration und Tests bereit.

Vorbedingung:

- Das Gesamtsystem befindet sich im **manuellen Betriebszustand**, d. h. der Lane-Follower kann manuell in der Testumgebung vom Entwickler getestet werden.
- Das Versionskontrollsystem und der Automatisierungsdienst sind eingerichtet.

Standardablauf: In Anhang B ist ein Sequenzdiagramm von einem beispielhaften Ablauf zu finden.
Nachbedingung:

- Die neue Version des Mikroservices wurde erfolgreich integriert.
- Die automatisierten Tests wurden durchgeführt und die Ergebnisse dem Entwickler zur Verfügung gestellt.

4. Anforderungen

In diesem Kapitel werden die Anforderungen dargestellt, die sich aus der definierten Problemstellung, der Forschungsfrage sowie den spezifischen Use-Cases ableiten. Diese Anforderungen ergeben sich sowohl aus dem zu testenden System als auch aus dem Aufbau der Simulations- und Testumgebung. Sie dienen im weiteren Verlauf der Arbeit als Grundlage für die verwandten Arbeiten, die verwendeten Konzepte und die Evaluation. Die Definition der Anforderungen orientiert sich an den Prinzipien des Standards 830 der IEEE (1998) und den Ansätzen von Burge et al. (2008). Die Anforderungen werden durch Erläuterungen ergänzt, um den Kontext und die Motivation zu verdeutlichen.

4.1. Anforderungen aus Use-Case 1

Die nachfolgenden Anforderungen wurden aus dem Use-Case 1 abgeleitet. Dieser Use-Case befasst sich mit dem Testen der Fahrfunktion und zielt darauf ab, den Entwicklungsprozess durch frühzeitige und effiziente Tests zu optimieren.

A1 – Bereitstellung von Sensordaten: Die Testumgebung muss in der Lage sein, die vom zu testenden System benötigten Eingangsdaten (Sensordaten) bereitzustellen. Je nach Konfiguration der Testumgebung stammen diese Daten von simulierten Sensoren in der Simulation oder von realen Sensoren über die HiL-Umgebung. Damit ist ein hybrider Ansatz möglich, bei dem beispielsweise Kameradaten von realen Sensoren mit Radardaten aus der Simulation kombiniert werden (Feilhauer et al. 2016). Auch Daten aus Kommunikationsinfrastrukturen wie beispielsweise V2X oder Global Navigation Satellite Systems (GNSS) werden als Sensordaten betrachtet und an das SUT gesendet (Kang et al. 2022).

A1.1 – Sensordaten aus verschiedenen Quellen: Die zu versendenden Sensordaten aus A1 stammen aus der virtuellen und physischen Komponenten der Testumgebung (wie in UC-1 beschrieben). Die folgenden Sensordaten müssen bereitgestellt werden: Eingabedaten direkt aus dem Simulationssensor, Eingabedaten aus einer realitätsnäheren Variante des Simulationssensors (Carlson et al. 2018) und Eingabedaten aus einem OTA-Sensor (Sievers et al. 2018; Reway et al. 2018; Johansson und Paulsson 2024).

A1.2 – Bereitstellung im standardisiertem Format: Die Testumgebung muss die genannten Sensordaten über eine definierte Schnittstelle, in einem standardisierten Format an das SUT übertragen.

A2 – Bereitstellung von Eigendaten: Zusätzlich zu den Sensordaten müssen dem zu testenden System auch Eigendaten des Fahrzeugs zur Verfügung gestellt werden. Dabei werden Daten wie der aktuelle Lenkwinkel, Motorstatus, Quer- und Längsbeschleunigung etc. in einem standardisierten Format über eine Schnittstelle übertragen. Diese Daten werden häufig in Ansätzen der Sensordatenfusion verwendet (Yeong et al. 2021).

A3 – Bereitstellung von Ground-Truth-Daten: Die Testumgebung muss neben den Sensor- und Eigendaten auch Ground-Truth-Daten an das zu testende System übertragen. Diese enthalten direkte Informationen über andere Verkehrsteilnehmer, Objekte, Fahrlinien, etc. Wie in Unterabschnitt 2.1.1 beschrieben, kann eine gängige ADAS-Funktion in die Teile Perzeption, Planung und Kontrolle unterteilt werden. Mithilfe der Ground-Truth-Daten ist es möglich ein Attrappe (engl. „mockup“) der Perzeption zu erstellen, die oft den anspruchsvollsten Aspekt einer solchen Funktion darstellt (Ruthardt und Michalke 2022).

A4 – Empfang der Befehle des zu testenden Systems: Um in der Testumgebung Closed-Loop-Tests zu ermöglichen, müssen die vom SUT berechnet Steuerbefehle empfangen und verarbeitet werden. Diese Steuerbefehle beinhalten unter anderem den gewünschten Lenkwinkel, Beschleunigung, Bremskraft/-druck oder Licht- und Blinkersteuerung. Diese Daten werden in einem standardisierten Format empfangen und von der jeweiligen Aktuatorik in der physischen oder virtuellen Testumgebung umgesetzt (Lotz et al. 2019).

A5 – Lose Kopplung zwischen Testumgebung und zu testendem System: Das SUT und die Testumgebung müssen lose gekoppelt voneinander arbeiten. Dabei wird das zu testende System aus Sicht der Testumgebung als Blackbox betrachtet, die Eingabedaten (Umwelt- und Eigendaten) empfängt (A1, A2) und Steuerbefehle sendet (A4). Die konkrete Architektur, die Kommunikation und der Quellcode des zu testenden Systems sind damit unabhängig von der Testumgebung. Lediglich die Verwendung einer Ethernet-basierten Kommunikation und der Einsatz von Containerisierungstechnologien werden vorgeschrieben (Brinkmann 2018).

A6 – Interoperabilität der Testumgebung: Die physische und die virtuelle Testumgebung mit ihren unterschiedlichen Sensoren und Aktoren arbeiten interoperabel miteinander. Das standardisierte Datenformat (A1.2) zum und vom zu testenden System muss von einer

Schnittstelle entgegengenommen werden und in die für die jeweilige Umgebung spezifische Datenrepräsentation umgewandelt werden. Dadurch ist es möglich, die Testumgebung ohne großen Anpassungsaufwand zu erweitern und beispielsweise neue Sensoren hinzuzufügen (Oruganti et al. 2019).

A7 – Dezentralisierung der Testumgebung: Um eine flexible Nutzung der Testumgebung zu gewährleisten, muss die Integration der verteilten Komponenten lose gekoppelt sein (Hahn 2015). Dadurch können die einzelnen Komponenten der virtuellen Testumgebung, der physischen Testumgebung, des Test- und V+V-Managements sowie des Adaptermoduls physisch getrennt auf verschiedenen Geräten ausgeführt werden. Dies ermöglicht es beispielsweise, Simulationen auch auf externen Servern im Netzwerk der Testumgebung durchzuführen.

A8 – Schnittstelle für Systemmonitore: Die Testumgebung muss eine generische Schnittstelle bereitstellen, die eine flexible Anbindung von Systemmonitoren ermöglicht. Über diese Schnittstelle werden sowohl relevante Daten des SUTs (z. B. Ergebnisdaten der Fahrfunktion) als auch Ressourcenkennzahlen wie die CPU-Auslastung in einem standardisierten Format bereitgestellt.

A9 – Auswahl und Durchführung von Szenarien: Das zu testende System muss mithilfe von Szenarien innerhalb der Simulation getestet werden können, welche sich am ASAM (2024b) OpenSCENARIO-Standard orientieren. Hierbei ist es notwendig, aus einer Liste vordefinierter Szenarien auszuwählen und eigene Szenarien zu definieren, da verschiedene Fahrfunktionen mit unterschiedlichen Szenarien getestet werden (Nguyen et al. 2024). Aus den verfügbaren Informationen wird nach Abschluss des Szenarios eine Dokumentation generiert und dem Entwickler zur Verfügung gestellt (Fremont et al. 2020).

A10 – Beobachtbarkeit: Für eine verbesserte Entwicklung und Evaluation ist eine Beobachtbarkeit der Testläufe notwendig. Die Testumgebung muss verschiedene Visualisierungen zur Verfügung stellen, mit denen die Ein-/Ausgabedaten des SUT und das Verhalten des Fahrzeugs in der Simulation beobachtet werden können (Brinkmann 2018).

4.2. Anforderungen aus Use-Case 2A

Die folgenden Anforderungen wurden aus dem Use-Case 2A abgeleitet und ergänzen die zuvor definierten Anforderungen. Dieser Use-Case beschreibt die Portierung einzelner Mikroservices von der SiL- in die HiL-Umgebung, während das System weiterhin als Gesamtheit nutzbar bleibt.

A11 – Portierbarkeit der Mikroservices zwischen SiL- und HiL-Umgebungen: Für einen verbesserten Entwicklungsprozess muss es möglich sein, die einzelnen Komponenten des zu testende Systems in der HiL- oder in der SiL-Umgebung bereitzustellen. Auf diese Weise können Komponenten gezielt isoliert und in der HiL-Umgebung getestet werden. Die Portierung der Mikroservices zwischen den Umgebungen erfolgt vollständig ohne Anpassungen am Quellcode (Ruehl und Bronner 2024; Sievers et al. 2018; Lotz et al. 2019).

A12 – Bereitstellungsorchestrator für das zu testende System: Die Verteilung der einzelnen Services des SUT in den Zielumgebungen übernimmt ein Bereitstellungsorchestrator (Villari et al. 2017). Der Entwickler muss dabei die Auswahl treffen können, welche Services in welcher Umgebung bereitgestellt werden sollen (Abhishek et al. 2022).

4.3. Anforderungen aus Use-Case 2B

Use-Case 2B stellt eine Erweiterung aus dem vorherigen Use-Case dar. In diesem wird ein Automatisierungsdienst genutzt, um neue Versionen von einzelnen Mikroservices automatisiert testen zu können. Dafür werden folgende Anforderungen definiert.

A13 – Automatisierung der Testumgebung: Um den Entwicklungs- und Testprozess des zu testenden Systems zu verbessern, werden automatisierte Tests durchgeführt. Hierfür steht ein Automatisierungsdienst zur Verfügung, der mit einem Versionskontrollsystem integriert ist. Der Automatisierungsdienst muss automatisierte Testläufe von neuen Versionen des zu testenden Systems durchführen können (Johansson und Paulsson 2024). Das Ergebnisprotokoll des Testlaufs wird dem Entwickler über das Versionskontrollsystem zur Verfügung gestellt (Busch et al. 2024; Mustyala 2022).

A13.1 – Konfiguration der Testläufe: Für die automatisierten Testläufe muss eine Konfiguration der Testumgebung durch den Entwickler möglich sein. Dabei trifft er eine Auswahl für: die Art der Eingabedaten (A1.1), das verwendete Szenario (A9) und die Zielumgebung der einzelnen Mikroservices (A12).

5. Verwandte Arbeiten

Diese Kapitel beschreibt die verwandten Arbeiten zu dieser Arbeit in einzelnen Themenbereichen: Testfelder für Cyber-Physical-Systems (Abschnitt 5.1), Testumgebungen für das Testen von Fahrfunktionen (Abschnitt 5.2) und die Softwareentwicklung von Automated Driving Systems (Abschnitt 5.3). Die verwandten Arbeiten leiten sich aus den Anforderungen (Kapitel 4) ab und werden in die drei genannten Abschnitte eingeteilt. Anschließend werden in Abschnitt 5.4 die ausgewählten Arbeiten kritisch betrachtet und Konzepte für diese Arbeit abgeleitet.

5.1. Testfelder für Cyber-Physical-Systems

In diesem Abschnitt werden zwei verwandte Arbeiten vorgestellt, die sich mit Testfeldern für CPS aus anderen Domänen beschäftigen. Testfelder und Umgebungen im Bereich der Automobilindustrie werden in Abschnitt 5.2 erläutert.

Brinkmann (2018) stellt in seiner Dissertation eine physikalische Testfeld-Architektur zur Unterstützung der Entwicklung automatisierter Schiffsführungssysteme vor. Der Autor geht der Frage nach, wie solch ein Testfeld gestaltet werden muss, um die V+V automatisierter Schiffsführungssysteme effizient zu unterstützen. Zur Beantwortung dieser Frage entwickelt er eine systematische und wiederverwendbare Testfeld-Architektur, die auf den Prinzipien des Systems-Engineering basiert und sich über den gesamten Entwicklungszyklus erstreckt. Abbildung 5.1 zeigt einen Überblick über das Testfeld und die strikte Trennung des SUT und dem eigentlichen Testfeld. Im Folgenden wird statt von „physikalischer“ von „physischer“ Architektur gesprochen, da eine Verwechslung der Begriffe durch den Autor vermutet wird. Der Autor beschreibt ebenfalls, dass entsprechend dem jeweiligen Testszenario, verschiedene Instanzen (der zu sehenden Elemente in Abbildung 5.1) entweder virtuell oder physisch existieren.

Im Falle einer SiL oder MiL Erprobung, findet die Bereitstellung über eine Simulation statt, während die Elemente zukünftig sukzessiv durch physische ausgetauscht werden. Zusätzlich schreibt der Autor über die konkrete Testfeld-Architektur, in der die nötigen Schnittstellen und Kommunikationen dargestellt werden. Abbildung 5.2 zeigt diese Systemarchitek-

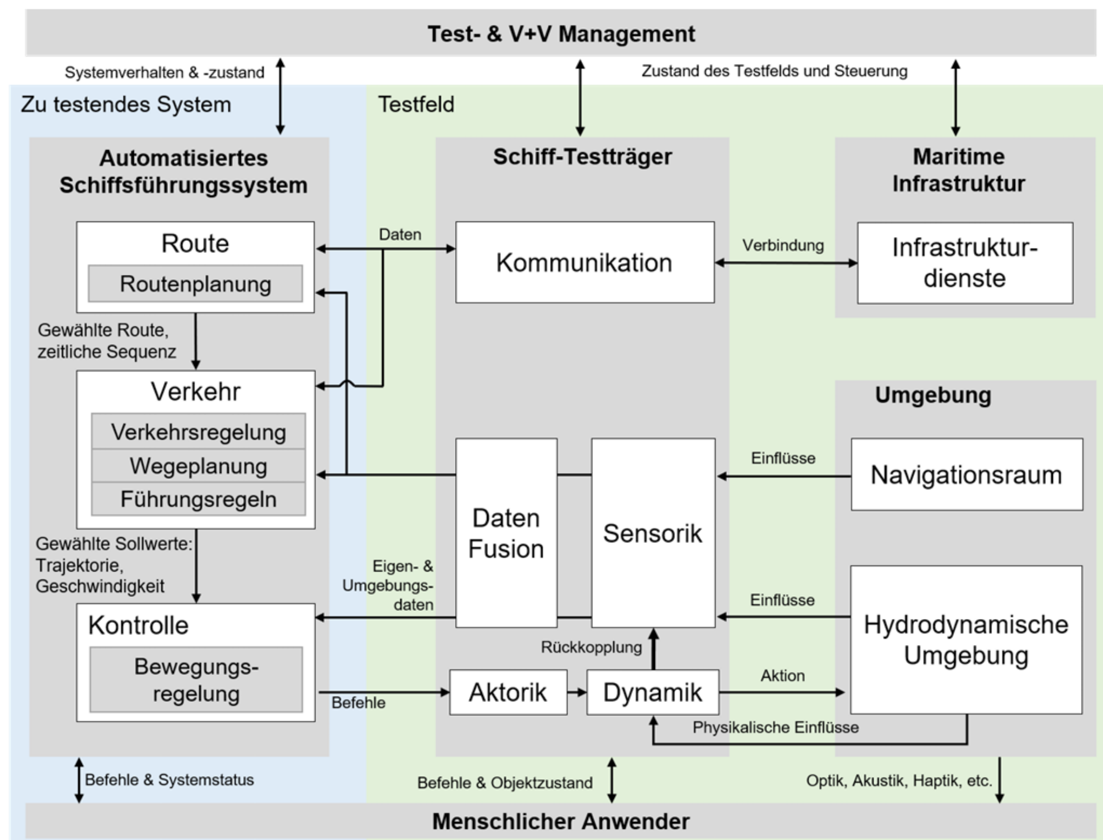


Abbildung 5.1.: Elemente eines Testfeldes für automatisierte Schiffsführungssysteme (Brinkmann 2018, S.66)

tur. In dieser sind die drei Hauptkomponenten: die Kommunikationsinfrastruktur, das Test- & V+V Management und die Testfeld-Komponenten zu erkennen. Diese Aufteilung basiert auf der von Nickovic et al. (2017) vorgestellten generischen Testarchitektur. Brinkmann (2018) legt besonderen Fokus auf die polymorphe Schnittstelle. Diese Schnittstelle ermöglicht eine flexible Integration verschiedenartiger SUT in das Testfeld, unabhängig von deren spezifischen Kommunikationsprotokollen oder Datenformaten. Sie dient als Vermittler zwischen den Testkomponenten und gewährleistet so die Interoperabilität der Testumgebung, indem sie Datenströme transformiert und semantisch anpasst. Als übergeordnete Datenstruktur verwendet der Autor das S-100 Framework, welches in der maritimen Domäne etabliert und vereinheitlicht ist (Ward et al. 2008). Für die konkrete Implementierung verwendet Brinkmann das Testfeld LABSKAUS (Labor für sicherheitskritische Analysen auf See), welches die Integration und Prüfung automatisierter Schiffsführungssysteme unterstützt (Hahn und Noack 2016). Mithilfe der polymorphen Schnittstelle konnte der Autor eine wiederverwendbare Infrastruktur zeigen, die es ermöglicht, verschiedene SUT in die Testumgebung zu integrie-

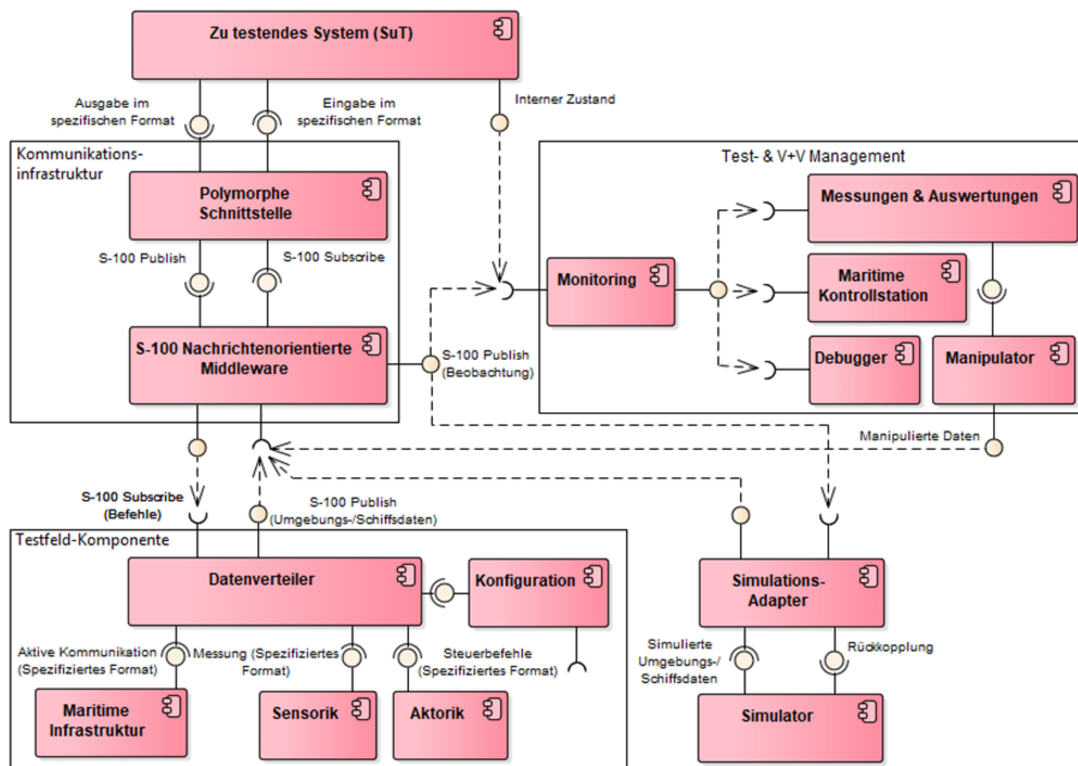


Abbildung 5.2.: Systemarchitektur des physischen Testfeldes (Brinkmann 2018, S.69)

ren und zu testen. Diese konnte in virtuellen und physischen Szenarien geprüft werden. Die Arbeit zeigt, wie eine flexible Schnittstelle den Entwicklungs- und Testaufwand senken kann, und erstellt eine Basis für zukünftige Testfelder, die auch andere maritime Automatisierungstechnologien unterstützen.

Hahn et al. (2015) stellen in ihrer Arbeit ein virtuelles Testfeld (HAGGIS) für die Bewertung der Sicherheit im Seeverkehr vor. Die Forschungsfrage der Autoren bezieht sich auf die Notwendigkeit von neuen Sicherheitsbewertungsmethoden in Bezug auf die zunehmende Automatisierung im Schiffsverkehr. Sie identifizieren das Fehlen einer flexiblen Testumgebung für SiL-/HiL-Tests und Simulation von präzisen Sensordaten als Forschungslücke. Das vorgestellte Framework der Autoren, umfasst Sensor- und Verkehrssimulationen sowie automatisierte Testverfahren, die Fehlerszenarien und Risikobewertungen untersuchen kann. Die generelle Architektur des HAGGIS Frameworks zeigt, dass die Komponenten zur Laufzeit mit der `High Level Architecture` über ein `S-100 Data Model` interagieren. Über dieses Datenmodell kann eine Integration zwischen HAGGIS, dem virtuellen Testfeld, und LABSKAUS, einem physischen Testfeld stattfinden. Das Datenmodell und die physische Testumgebung (LABSKAUS) werden in der bereits vorgestellten Arbeit von Brinkmann (2018) erläutert.

Die Ergebnisse zeigen, dass HAGGIS und das ergänzende physische Testfeld LABSKAUS eine nahtlose Entwicklung und Evaluierung neuer e-Navigationstechnologien ermöglichen, beginnend mit virtuellen Simulationen bis hin zu physischen Demonstrationen.

5.2. Testumgebungen für das Testen von autonomen Fahrfunktionen

Dieser Abschnitt enthält verwandte Arbeiten, die sich mit Ansätzen für Testumgebungen für Fahrzeugarchitekturen befassen. Der Schwerpunkt liegt auf dem XiL-Ansatz, der in den Arbeiten vorgestellt wird.

Feilhauer et al. (2016) präsentieren in ihrer Arbeit aktuelle Ansätze zur Verifikation und Validierung von ADAS mithilfe von HiL-Testständen. Sie beschreiben Ansätze, wie XiL mit dem V-Modell zusammengefasst werden können. Für den Aufbau von HiL-Testumgebungen identifizieren die Autoren die größte Herausforderung: „Wie können die Stimuli in den ADAS ECU eingespeist werden?“ (Feilhauer et al. 2016, S.65). Dies wird insbesondere in Testumgebungen mit mehreren realen Sensoren und deren Kommunikation zu einem komplexen Problem. Die Autoren adressieren diese Herausforderung, indem sie ein Klassifikationsschema für die Einspeisung von Daten vorschlagen. Dieses Klassifikationsschema, bestehend aus physischer Signalebene, Rohdatenschicht und verarbeiteter Datenschicht, stellt eine Struktur bereit, die den Datenfluss von der physischen Signalverarbeitung bis zur fertigen Informationsschicht organisiert. Abbildung 5.3 zeigt eine Testumgebung, in der ein Kamerasensor über die physische Signalebene und ein Radarsensor über die verarbeitete Datenschicht gleichzeitig genutzt werden können. Die Autoren betonen, dass die Testbarkeit autonomer Systeme durch die Möglichkeit der Dateneinspeisung auf relevanten Ebenen gewährleistet werden muss. Die Standardisierung von Softwareschnittstellen und Rohdatenformaten würde den Austausch und die Wiederverwendbarkeit von Softwareanwendungen erleichtern und den Testaufwand reduzieren. Die Autoren haben mit der oben genannten Klassifizierung von Eingangsdaten eine Grundlage geschaffen, wie solche Testumgebungen strukturiert werden können. Gleichzeitig konnte an ihrem Beispiel gezeigt werden, wie mehrere Sensoren unterschiedlicher Klassen in einer Testumgebung eingesetzt werden können.

Sievers et al. (2018) diskutieren in ihrer Arbeit den Einsatz verschiedener Testumgebungen und XiL-Systeme. Konkret schreiben sie über eine gemeinsame Toolchain für HiL- und SiL-Tests unter Verwendung des „offline simulators dSPACE VEOS“ [1] (Sievers et al. 2018,

1. https://www.dspace.com/de/gmb/home/products/sw/simulation_software/veos.cfm (Abgerufen am 02.10.2024)

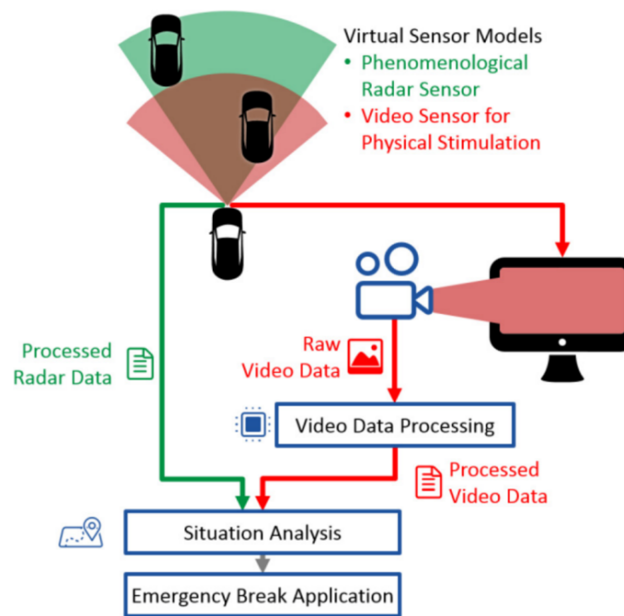


Abbildung 5.3.: Gemischte ADS-HiL-Testumgebung mit verschiedenen Sensormodellen (Feilhauer et al. 2016)

S.129). Wie bereits beschrieben, setzen auch sie die Umgebungswahrnehmung (Perzeption) von ADAS in den Fokus eines hochautomatisierten Fahrzeugs. Als konkretes Beispiel wählen die Autoren eine kamerabasierte Fahrfunktion, die in einer eigenständigen ECU ausgeführt wird. Die Daten für diese Funktion können aus verschiedenen Quellen kommen und können in unterschiedliche Stufen der Funktion eingespielt werden. In Abbildung 5.4 sind die verschiedenen Möglichkeiten zu erkennen. Auf der rechten Seite ist die Sensor-ECU zu erkennen. Diese enthält die verschiedenen Schritte für die Fahrfunktion, von der Vorverarbeitung bis zu Trajektorienplanung. Ebenfalls ist zu erkennen, wo die Eingabendaten von der Simulationsumgebung (links) in die Funktion übertragen werden, z. B. die rohen Sensordaten, die direkt in die Detektionsfunktion gesendet werden. Die Autoren beschreiben verschiedene Optionen, wie Daten in einen sensorbasierten ECU injiziert werden können:

- **OTA-Sensordaten**: Die Umgebungssimulation wird mithilfe einer 3D-Engine auf einem Monitor visualisiert, der als Eingangssignal für den Kamerasensor dient, welcher vor dem Monitor platziert ist.
- **Rohe Kameradaten**: Sensordaten werden direkt aus der Simulation in Form von Rohdaten in das Steuergerät eingespeist, wobei die Linse, der Bildsensor und andere optische Komponenten der Kamera umgangen werden.
- **Zielliste**: Eine Liste erkannter Ziele, die direkt vor dem Tracking-Algorithmus der Ka-

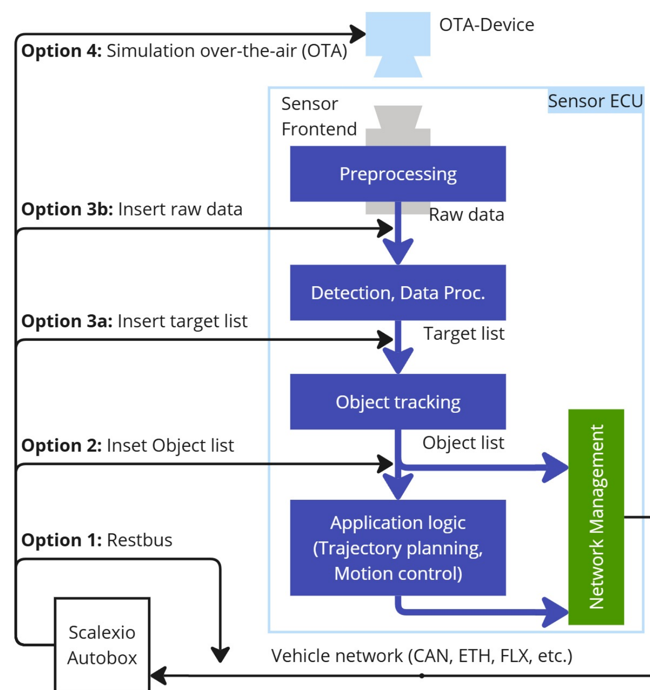


Abbildung 5.4.: Validierung eines Sensor-ECU über verschiedene Abstraktionsoptionen, nach Sievers et al. (2018)

mera in das Steuergerät eingespeist wird.

- **Sensorunabhängige Objektliste**: Eine objektbasierte Ground-Truth-Liste aus der Simulation, die in die Anwendungslogik des Steuergeräts injiziert wird, unabhängig von spezifischen Sensoren.

Den Schritt von den rohen Kameradaten zum OTA-Sensordaten beschreiben die Autoren als Systemwechsel vom SiL zum HiL. Durch den Einsatz der Kameralinse und Objektiv wird die echte Fahrzeughardware in die Loop integriert (HiL). In Abbildung 5.5 lässt sich erkennen, wie die Autoren die rohen Kameradaten über ein Interface in die Bildverarbeitung senden und damit die Linse und Kamerasensor des OTA-Sensors umgehen. Die Kamerarohdaten werden über das `Environment Sensor Interface Unit` und den `ESI-POD` an das in rot dargestellte Interface übertragen. Von dort gelangen die Daten direkt zur Bildbearbeitungseinheit. Die Autoren konnten mit ihrer Arbeit eine gemeinsame Toolchain für das Testen von ADAS in SiL- und HiL-Umgebungen zeigen und wie verschiedene Arten von Sensordaten verwendet werden können. Damit kann die Entwicklung und Integration von neuen Fahrfunktionen verbessert werden.

Reway et al. (2018) entwickeln eine Testmethodik für kamerabasierte ADAS mit einem realen Camera-in-the-Loop (CiL)-Ansatz. Ziel ist dabei die Prüfung von verschiedenen Al-

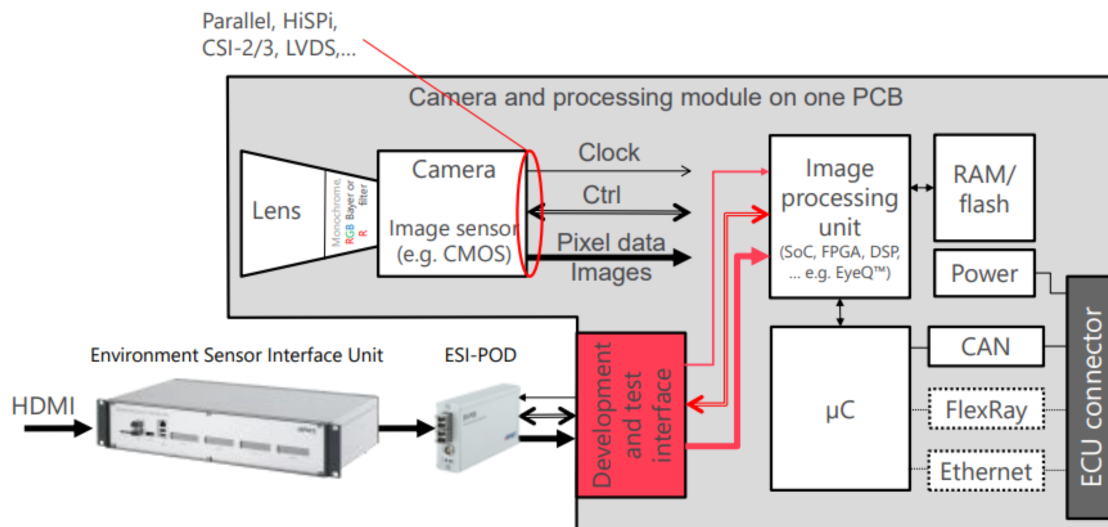


Abbildung 5.5.: Injizierung von rohen Kameradaten zur Umgehung der Kameralinse und des Kamerasensors (Sievers et al. 2018)

gorithmen zur Objekterkennung in einer kontrollierten virtuellen Umgebung. Dabei können Herausforderungen wie schlechte Wetterbedingungen simuliert und die Grenzen des Systems getestet werden. Der Testaufbau der Autoren besteht aus einem Closed-Loop-System, bei der die Fahrfunktion auf die erkannte Objekte reagiert und Entscheidungen an die Simulation weiterleitet. Die Teststrategie der Autoren beginnt mit dem Laden des Szenarios und dem Start des ersten Testfalls. Anschließend wird die Umgebung in der Simulation verändert (Wetter-/Straßenbedingungen), verschiedene Kameraeffekte (Rauschen, Verzerrungen, etc.) hinzugefügt und die Testfälle wiederholt. Mit dieser Strategie konnten die Autoren die Grenzen ihrer Objekterkennung gezielt identifizieren und planen, das System zukünftig als Benchmarking-Plattform für verschiedene Hard- und Softwarelösungen für autonome Fahrfunktionen zu nutzen.

Ruehl und Bronner (2024) stellen in ihrer Arbeit den Einsatz von Virtual Electronic Control Units (V-ECUs) in SiL- und HiL-Umgebungen vor. Sie schreiben, dass die virtuelle Entwicklung und Verifikation vor allem im Bereich des SDVs immer wichtiger wird. Als Forschungslücke beschreiben die Autoren die Migration zwischen der SiL- und der HiL-Umgebung. Eine einfache Migration hat den Vorteil, dass Artefakte, wie Modell, Testfälle und Umgebungen, wiederverwendbar werden. Die Autoren erläutern zusätzlich die Vorteile einer hybriden Testumgebung. Beispielsweise wenn einige Komponenten eines SUT als reale ECU, jedoch eine Komponente bisher nur als virtuelles Artefakt existiert. In einem hybriden Testsystem könnte bereits Integrationstests ausgeführt werden. Die Autoren beschreiben

ebenfalls eine Co-Simulation zwischen HiL und SiL. Dabei werden reale ECUs im HiL zusammen mit virtuellen im SiL verwendet. Abbildung 5.6 zeigt diese Co-Simulation und den Datenaustausch über Ethernet.

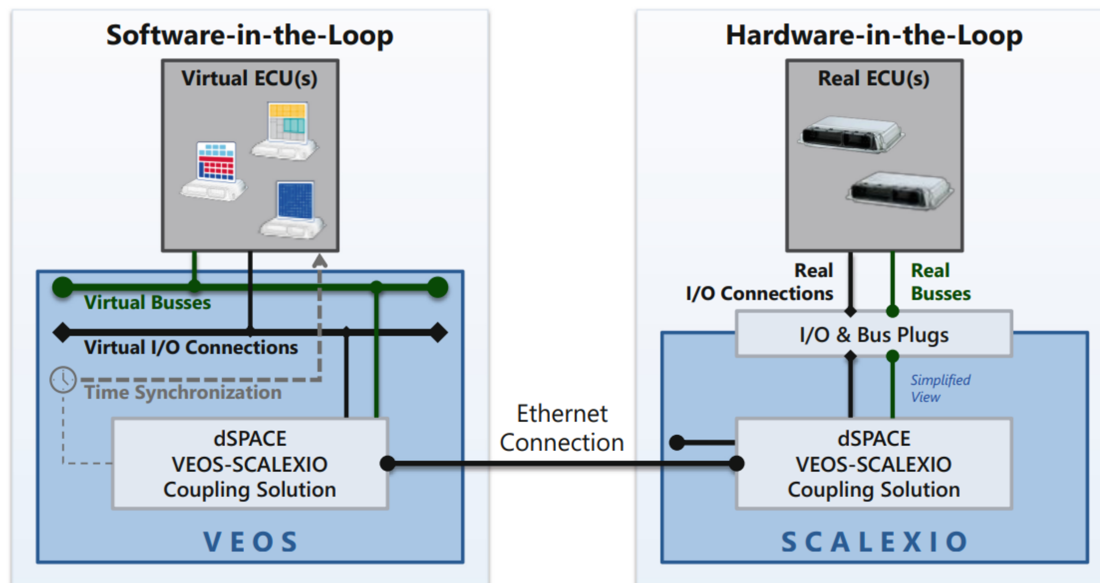


Abbildung 5.6.: Co-Simulation zwischen SiL und HiL über eine Verbindung über Ethernet (Ruehl und Bronner 2024)

Diese Methode hat den Vorteil, dass ECUs nach und nach mit einem fließenden Übergang von SiL auf HiL übertragen werden können. Als beispielhafte Implementierung haben die Autoren in Abbildung 5.6 eine CAN-Kommunikation nachgebildet, die über Ethernet getunnelt wird. Der Fokus lag dabei auf der Latenz zwischen der Signalgenerierung im HiL und der Antwort im SiL, die zwischen 4 und 20 ms liegt. Dies ist auf das Nicht-Echtzeitverhalten des VEOS-Simulators zurückzuführen. Mithilfe der Co-Simulation konnten die Autoren eine hybride Testumgebung demonstrieren, in der reale ECUs im HiL mit virtuellen im SiL kommunizieren können. Dadurch kann der Entwicklungsprozess von ECUs beschleunigt und Artefakte von einer Testumgebung in die andere wiederverwendet werden. Sie geben ebenfalls an, dass der gezeigte Ansatz mit dem VEOS-Simulator für zeitkritische Anwendungen mit Kommunikationszyklen unterhalb der genannten Latenz nicht geeignet sein könnten.

5.3. Softwareentwicklung von Automated Driving Systems

In diesem Abschnitt werden verwandte Arbeiten vorgestellt, die sich mit der Softwareentwicklung von ADS befassen. Der Schwerpunkt liegt auf der Verbesserung der Entwicklung mithilfe von Docker-Containern, automatisierten CI/CD-Pipelines und der Verwendung von

Microservices.

Busch et al. (2024) schreiben in ihrer Arbeit über die Verbesserung der Entwicklung von robotischen Applikationen mithilfe von Mikroservices-Architekturen und automatisierte Containerisierung. Die Autoren beschreiben, dass automatisierter DevOps-Zyklen und Mikroservices-Architekturen bereits große Erfolge in großen Webservices (z. B. Netflix) erzielen konnte. Sie sehen die Nutzung dieser Technologien im Bereich der ROS-Applikationen als Forschungslücke und stellen eine Tool-Suite als Lösung vor. Diese Suite umfasst unter anderem: ein generisches Dockerfile zur Automatisierung des Erstellungsprozesses von Container-Images, eine Sammlung von Machine-Learning-unterstützten Basis-Container-Images und ein Command-Line-Tool zur Interaktion mit Containern während der Entwicklungsphase. Abbildung 5.7 zeigt einen Überblick über die von den Autoren vorgestellte Tool-Suite und deren Integration.

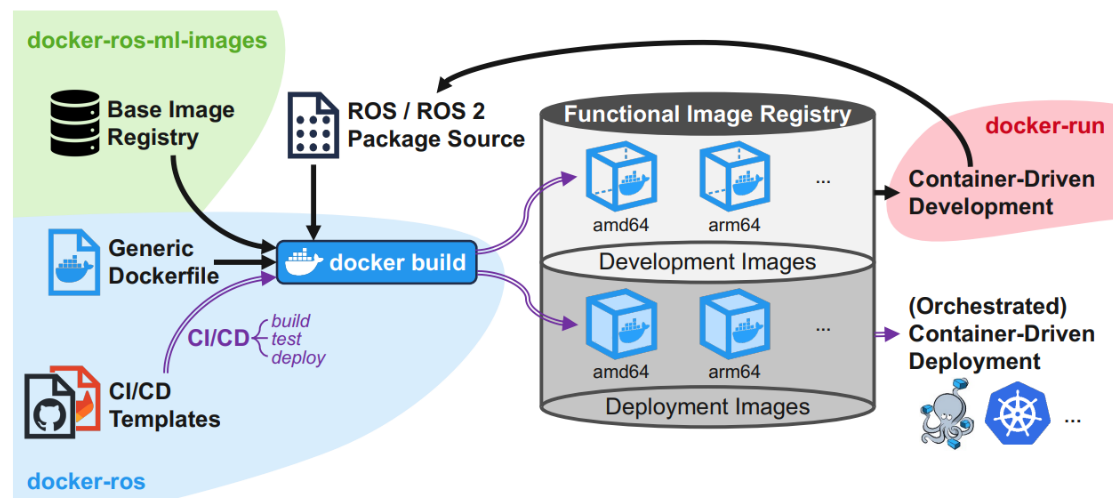


Abbildung 5.7.: Überblick über die vorgestellte Tool-Suite der Mikroservices-Architektur zu Bereitstellung von ROS-Applikationen (Busch et al. 2024)

Über `docker-ros` wird ein generisches Dockerfile erstellt, in bereits Abhängigkeiten und CI/CD-Templates enthalten sind (siehe Abbildung 5.7). `docker-ros-ml-image` enthält bereits nötige Frameworks, die für Maschine-Learning benötigt werden. Diese Images werden automatisiert in ein Entwicklungs- und ein Bereitstellungs-Docker-Image gebaut. Mithilfe von `docker-run` kann der containerisierte Entwicklungsprozess beschleunigt werden. Im letzten Schritt werden die Bereitstellungs-Images über Framework wie Kubernetes oder Docker-Swarm automatisiert orchestriert. Mit der vorgestellten Tool-Suite konnten die Autoren die Erfolge von Mikroservices-Architekturen und DevOps-Zyklen auf den Bereich der

robotischen Applikationen übertragen. Mit den beschriebenen Open-Source-Software[1] wie `docker-ros`, `docker-ros-ml-images` und `docker-run` kann die Entwicklung von solchen containerbasierten Applikationen verbessert und die Bereitstellung automatisiert werden.

Lotz et al. (2019) untersuchen in ihrer Fallstudie die Effekte der Umstellung einer komplexen ADAS-Funktion auf eine Mikroservice-Architektur. Ihre Forschungsfrage konzentriert sich auf die Herausforderungen, die Eignung, die Vor- und Nachteile einer solchen Umstellung. Für ihre Studie entwickeln die Autoren ein Lane-Following-System, das mithilfe von OpenDLV und verschiedenen Mikroservices realisiert wird, die über Ethernet miteinander kommunizieren. Die verschiedenen Services und deren Kommunikation sind in Abbildung 5.8 zu erkennen. Die Autoren verwenden das OpenDLV-Framework[2], das als ganzes Ökosystem

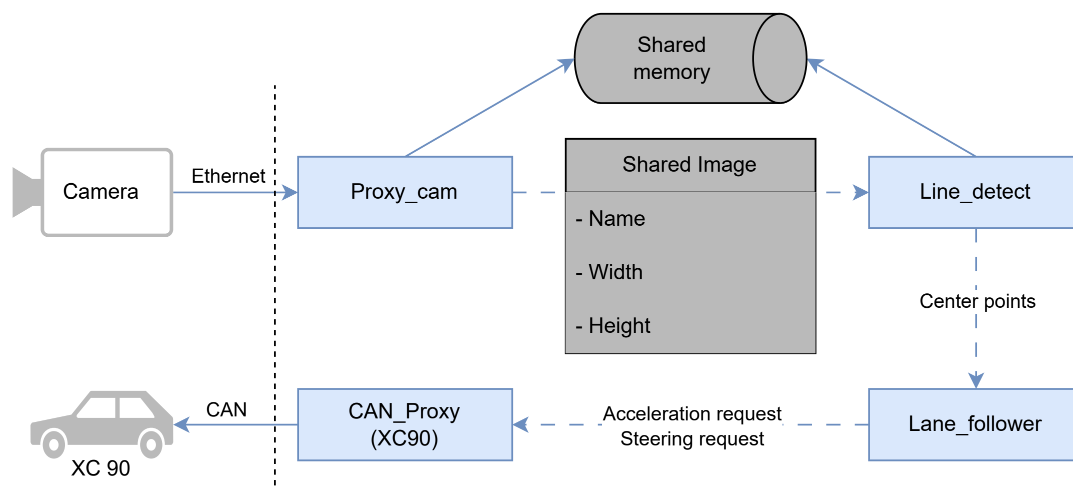


Abbildung 5.8.: Architektur des Lane-Followers, nach Lotz et al. (2019)

für mikroservice-basierte Fahrsysteme entwickelt wurde. Es basiert auf libcluon, das der Mitautor C. Berger in seiner Arbeit (Giaimo und Berger 2017) vorstellt. Die Kommunikation der Sensordaten mit der Fahrfunktion erfolgt über Ethernet. Für die Bereitstellung der einzelnen Mikroservices verwenden die Autoren Docker zusammen mit Docker-Compose. Ein besonderer Fokus liegt dabei auf der Verwendung von „Alpine Linux und den Linux Kernel Patch PREEMPT_RT“ (Lotz et al. 2019, S. 3). Damit können Echtzeitanforderungen wie das jederzeitige Unterbrechen von Prozessen und das Sperren bestimmter Ressourcen von sensiblen Prozessen realisiert werden. Ihre Ergebnisse basieren auf einer Strengths, Weaknesses, Opportunities and Threats (SWOT)-Analyse von Mikroservices-Architekturen. Zu den Vorteilen, zählen die Autoren eine hohe Modularität, Wiederverwendbarkeit und eine verein-

1. https://github.com/ika-rwth-aachen (Abgerufen am 29.10.2024)
2. https://github.com/chalmers-revere/opendlv (Abgerufen am 23.10.2024)

fachte Skalierung. Sie stellen jedoch fest, dass die Komplexität der Integration des Systems und die nötigen Beschreibungen der Schnittstellen zwischen den Services die größten Herausforderungen darstellen.

Johansson und Paulsson (2024) stellen in ihrer Masterarbeit ein mikroservicebasiertes Framework vor, in dem ein CPS mithilfe von Simulation, Datenwiedergabe und Hardware getestet werden kann. Die Autoren formulieren die Forschungsfragen, ob eine Mikroservice-Architektur Multi-Level-Tests unterstützen kann, ob sie für HiL-Tests geeignet ist und welche Herausforderungen bei der Integration in eine CI/CD-Pipeline bestehen. Für die Beantwortung der Fragen verwenden die Autoren eine SiL-, Data-in-the-Loop (DiL)- und HiL-Testumgebung, Docker für die Bereitstellung und die GitLab-API als CI/CD-Integration. Als beispielhafte Fahrzeugplattform verwenden die Autoren ein bereits entwickeltes Miniaturfahrzeug, welches auf dem OpenDLV-Framework[1] basiert. Dieses kann mithilfe einer Kamera Pylonen in der Umgebung erkennen und der vorgegebenen Spur folgen. Die Autoren kategorisieren ihre Tests in die reine Perzeption, Open-Loop (Datenwiedergabe) und Closed-Loop (Feedback wird zurückgespielt). Tabelle 5.1 zeigt eine Tabelle, welche Tests in welcher Testumgebung durchgeführt werden können.

Tabelle 5.1.: Kompatibilität der Testmethode in unterschiedlichen Testumgebungen, nach Johansson und Paulsson (2024)

	Perzeption	**Open-Loop**	**Closed-Loop**
SiL	Ja	Ja	Ja
DiL	Ja	Ja	Nein
HiL	Ja	Ja	Ja

Um die Perzeption bewerten zu können, verwenden die Autoren die Intersection over Union (IoU)-Metrik, um die erkannten Pylone zu vergleichen. In den Closed-Loop-Tests, in denen das Feedback in das System zurückgespielt wird, beschreiben die Autoren die Herausforderung einer solchen Bewertung, da kleine Veränderungen in der Mitte des Testlaufs große Veränderungen in der Zukunft bedeuten können. In ihren Testläufen wird deshalb nur der letzte Systemzustand evaluiert (in diesem Beispiel das Erreichen des Ziels und das Vermeiden des Fahrens außerhalb der Spur). Für das automatisierte Testen verwenden die Autoren den CI/CD-Runner von GitLab. Sobald Codeänderungen in einem bestimmten Branch auftreten, erstellt der Runner die notwendigen Docker-Images und lädt sie in die Registry hoch. Eine zusätzlich von den Autoren entwickelte Anwendung (DoDo-Test-Runner) startet dann die

1. https://opendlv.org/

Tests und stellt die Ergebnisse in GitLab zur Verfügung.

Die Autoren beschreiben in ihrer Diskussion, dass die Verwendung von Docker-Containern und einer standardisierten Kommunikation (OpenDLV), eine einfache Transition zwischen den Testumgebungen erfolgen konnte. Durch die verschiedenen Testfälle wurde ein Multi-Level-Testing ermöglicht, das ebenfalls in der HiL-Umgebung getestet werden konnte.

5.4. Handlungsbedarf

In den vorangegangenen Abschnitten wurden bereits einige Arbeiten vorgestellt, die sich mit Testumgebungen in verschiedenen sicherheitskritischen Bereichen beschäftigen. Auch eine verbesserte Entwicklung von CPS durch den Einsatz von XiL-Technologien und verschiedenen Abstraktionen von Sensordaten konnte in verschiedenen verwandten Arbeiten beobachtet werden. Die Erkenntnisse aus diesen Arbeiten werden zusammengefasst, kritisch betrachtet und der sich daraus ergebende Handlungsbedarf erläutert. Dabei wird auch auf Anforderungen aus Kapitel 4 verwiesen.

Brinkmann (2018) schreibt in seiner Dissertation über eine physische Testfeld-Architektur LABSKAUS für die Entwicklung von automatisierten Schiffsführungssystemen. Diese Architektur ist an die von Nickovic et al. (2017) vorgestellte generische Testarchitektur angelehnt und spezifisch für die maritime Domäne konzipiert. Anpassungen auf die Automobilindustrie mit ihrer eigenen Kommunikationsinfrastruktur, Sensordaten und Anforderungen sind notwendig. Dennoch werden in der vorgestellten Testfeld-Architektur Umgebungs- und Eigendaten an das zu testenden System gesendet (A1, A2) und gleichzeitig Befehle entgegengenommen (A4). Die dafür verwendeten Konzepte einer nachrichtenorientierten Middleware und der generellen Aufteilung des Testfeldes können für diese Arbeit eingesetzt werden. Ebenso konzentriert sich die Arbeit von Brinkmann (2018) auf eine physische Testfeld-Architektur. Es gibt zwar Ansätze, wie eine virtuelle Testumgebung fehlende Daten aus der physischen ersetzten kann, sie ist aber nicht der Fokus der Arbeit. Die virtuelle Testumgebung HAGGIS, vorgestellt von Hahn et al. (2015), steht im Zusammenhang mit der Arbeit von Brinkmann (2018). Die in den Arbeiten gezeigten Standards der maritimen Domäne müssen auf den Automobilbereich angepasst werden. Ebenso beschreiben die Autoren Hahn et al. (2015) und Brinkmann (2018) die Kommunikation zwischen Infrastrukturen und dem zu testenden System über das S-100-Datenmodell, welches ebenfalls nicht für das hochautomatisierte Fahren konzipiert wurde.

Die vorgestellten Ansätze von Brinkmann (2018) ermöglichen es, verschiedene SUT in die Testumgebung über ein standardisiertes Format zu integrieren. Die verwendete Testumgebung bleibt dabei gleich. In dieser Arbeit wird der Fokus auf die Erweiterbarkeit

der Testumgebung gelegt. Die Daten an und vom SUT über ein standardisiertes Format werden dabei als Rahmenbedingung an den Entwickler der Fahrfunktion übergeben (A1.2). Die Anforderung einer polymorphen Schnittstelle werden in dieser Arbeit auf einen Adapter reduziert. Dadurch kann die virtuelle und physische Testumgebung erweitert werden. Die Ansätze der Testumgebung von Brinkmann (2018) dienen in dieser Arbeit als Grundlage für das verwendete Konzept und werden in Kapitel 6 näher erläutert.

Sievers et al. (2018) zeigen in ihrer Arbeit einige Ansätze zur Validierung von Sensor-ECUs über verschiedene Abstraktionsstufen der Sensordaten. Ebenso stellen die Autoren eine gemeinsame Toolchain vor, um Tests in HiL- und SiL-Umgebungen durchzuführen. Der Einsatz einer solchen dezentralen Fahrzeugarchitektur über dedizierten Steuergeräten, wird in kommenden Fahrzeugtrends, wie dem SDV, immer mehr durch eine zentralisierte Architektur ersetzt (Slama et al. 2023, S. 22). Funktionen wie die Bildverarbeitung, Perzeption oder eine Trajektorienplanung werden häufig in leistungsstarken Rechenknoten ausgeführt, die von neuronalen Netzen unterstützt werden. In solch einer Architektur sind dezentrale Sensor-ECUs nicht mehr vorzufinden.

Die von Sievers et al. (2018) gezeigte Abstraktion der Sensordaten und die Einspeisung in die verschiedenen Stufen des ECUs ermöglicht, laut den Autoren, einen verbesserten Entwicklungsprozess. Entwickler können schrittweise Daten einspeisen und Komponenten, wie z. B. die Objekterkennung, einzeln testen. Einen ähnlichen Ansatz verfolgt auch Feilhauer et al. (2016) in seiner Arbeit über die Verifikation und Validierung von ADAS. Auch sie schreiben über verschiedene Klassen von Sensordaten und deren Abstraktion. Diese Herangehensweisen von Sievers et al. (2018) und Feilhauer et al. (2016) baut auf der Annahme auf, dass die Konfiguration des Sensor-ECUs zum Aufbau der Testumgebung schon bekannt ist. Ebenso setzten die vorgestellten Ansätze eine feste Definition der Sensorschnittstellen voraus, bevor Fahrfunktionen entwickelt oder getestet werden kann. Damit ist die Testumgebung und die Entwicklung der CPS eng miteinander verbunden. Bei dem Testen einer neuen Fahrfunktion, über die vorgestellte Testumgebung, müsste diese grundlegend überarbeitet und auf den neuen Anwendungsfall angepasst werden. Dies steht im Widerspruch zu modernen Entwicklungsmethoden, bei denen die Flexibilität und eine Entkopplung von Sensoren und Funktionen eine große Rolle spielen (Lou et al. 2022).

Die in dieser Arbeit angestrebte lose Kopplung der beiden System (A5) ist mit diesem Vorgehen nicht möglich. In dieser Arbeit wird die Sensorhardware von dem zu testenden System getrennt. Die Fahrfunktion hat die Möglichkeit, auf die Sensordaten zuzugreifen, sie ist sich dem Ursprung dieser Daten nicht bewusst. Die von Sievers et al. (2018) vorgestellte Injizierung von rohen Kameradaten zur Umgehung der Kameralinse und des Objektivs wird in dieser Arbeit über die verschiedenen Eingabedaten abgebildet (A1.1). Der Ansatz von

Sievers et al. (2018) ermöglicht es zusätzlich, eine Zielliste und eine Sensorunabhängige Objektliste abzugreifen. Diese beiden Optionen sind durch die beschriebene Entkoppelung der Systeme nicht möglich. Darüber hinaus spricht Feilhauer et al. (2016) von einem hybriden Ansatz, bei dem virtuelle und physische Sensordaten in Kombination genutzt werden (A1). Die Herausforderungen bei der Synchronisation der Sensordaten in einem solchen hybriden Ansatz werden in der Arbeit jedoch nicht thematisiert. Dennoch ermöglicht solch ein Ansatz einen sukzessiven Austausch von virtuellen Testkomponenten durch physischen und kann damit den Entwicklungsprozess von CPSs verbessern.

Sievers et al. (2018) speisen die Kamerarohdaten über proprietäre Hardware des Unternehmens dSpace[1] und HDMI-Schnittstellen ein. Dieser Ansatz hängt damit stark mit der Verwendung von Original Equipment Manufacturer (OEM) spezifischen Werkzeugen zusammen und ermöglicht keinen einfachen Nachbau der gezeigten Testumgebung, da die Datenformate nicht bekannt sind. Durch die Verwendung eines standardisierten Formats für Sensordaten (A1.2) und Open-Source-Projekte in dieser Arbeit soll dies möglich sein. Abschließend beschreibt Sievers et al. (2018) den Wechsel zwischen OTA-Sensordaten und Kamerarohdaten aus der Simulation, als Systemwechsel zwischen einer HiL- und SiL-Testumgebung. Der vorgestellte Ansatz in dieser Arbeit soll es ermöglichen, das zu testende Sytem, welches sich im HiL oder SiL befinden kann, mit allen möglichen Sensordaten testen zu können. Diese Trennung des SUT und den Sensordaten wird im Ansatz von Sievers et al. (2018) nicht betrachtet.

Reway et al. (2018) entwickeln in ihrer Arbeit eine Testmethodik für kamerabasierte ADAS über CiL-Ansätze. Die von den Autoren vorgestellte iterative Teststrategie bietet einige interessanten Möglichkeiten. Das vorgestellte CPS, in diesem Beispiel eine Objekterkennung, wird mit verschiedenen Szenarien unter der Verwendung eines OTA-Sensors getestet. Diese enthalten unterschiedliche Straßen- als auch Wetterbedingungen. Es ist möglich, die Objekterkennung unter gleichen Straßen- und Verkehrsverhältnisse, mit verschiedenen Wetterbedingungen zu testen (A9). Damit ist es möglich, konkret die Schwachstellen der Erkennung zu ermitteln. Besonders interessant, ist der Ansatz, die Kameraeffekte (Rauschen, Bildverzerrung), auf dem Monitor darzustellen, der die Kamera stimuliert und nicht erst danach. Dies hat den Vorteil, dass Testläufe reproduzierbar sind, und Effekte wie Schmutz auf der Kamera simuliert werden können, ohne die Linse oder das Objektiv der Kamera zu beschädigen. Auch das Field of View (FOV) der simulierten Kamera, die die Sensordaten auf dem Monitor visualisiert, wird in dem vorgestellten Ansatz von der realen Kamera übernommen. Über eine korrekte Positionierung der realen Kamera kann damit ein realitätsnahes Bild für die Fahrfunktion zur Verfügung gestellt werden. Reway et al. (2018) verwenden für

1. https://www.dspace.com/de/gmb/home.cfm (Aberufen am 28.11.2014)

ihren physischen Testaufbau, drei Kameras: vorne, hinten und links, die jeweils durch einen angepassten CiL-Aufbau umgesetzt werden. Jedoch sprechen die Autoren nicht über eine mögliche Synchronisation dieser Daten. Die Autoren schließen ihre Arbeit damit ab, dass eine Objekterkennung über eine Kamera sehr anfällig für Wetter- und Straßenbedingungen ist und dass für bessere Testergebnisse weitere Sensoren wie LiDAR oder RADAR nötig sind. Jedoch fehlt, wie solche Daten über einen OTA-Sensor generiert werden könnten.

Ruehl und Bronner (2024) schreiben in ihrer Arbeit über Ansätze, wie virtuelle ECUs in SiL- und HiL-Umgebungen zu testen, um die Entwicklung von im Bereich des Software-Definied Vehicles zu verbessern. Besonders der gezeigte hybride Ansatz, bei dem virtuelle ECUs im SiL und physische ECUs im HiL gemeinsam verwendet werden, ist für diese Arbeit relevant (A1). In dem gezeigten Test werden CAN-Daten aus dem HiL über eine Ethernet-Schnittstelle ins SiL versendet. Genau wie Sievers et al. (2018) verwenden die Autoren ebenfalls proprietäre Hard- und Software von dSpace. Das konkrete Kommunikationsprotokoll oder der Aufbau der Ethernet-Nachrichten wird nicht beschrieben. Ruehl und Bronner (2024) messen in ihrer Evaluation die Latenz von sehr kleinen CAN-Nachrichten, von der Signalgenerierung bis zum Empfang. Sie geben die Latenz mit 4 bis 20 ms an. Interessant wäre hier ein Test mit größeren Daten, die z. B. bei dem Versenden von Sensordaten aufkommen. Die Autoren erklären die auftretende Latenz durch die Verwendung des dSpace VEOS-Simulators, der keine Echtzeitfähigkeiten besitzt. Um diese Aussage zu beweisen, müsste zusätzlich die reine Latenz der Ethernet-Verbindung gemessen werden. Dies wird in der Arbeit nicht betrachtet.

Lotz et al. (2019) zeigen in ihrer Arbeit eine Umstellung einer Fahrfunktion auf eine mikroserviceorientierte Architektur. Als ADAS-Funktion untersuchen die Autoren einen Lane-Follower, der aus verschiedenen Mikroservices besteht. Sie verwenden als Kommunikationsprotokoll das Framework OpenDLV[1], welches von Giaimo und Berger (2017) vorgestellt, und von Johansson und Paulsson (2024) ebenfalls verwendet wird. Dieses Framework könnte in dieser Arbeit für die Bereitstellung von Sensordaten (A1) und Entgegennahme von Steuerbefehlen (A4) verwendet werden. Eine Verfolgung dieses Ansatzes, würde jedoch den Entwickler des zu testenden Systems zwingen, ebenfalls dieses Framework zu nutzen. Dies spricht gegen die definierte Anforderung A5. Die Autoren schreiben in ihrer Evaluation und Fazit auch über die großen Vorteile einer mikroserviceorientierten Architektur. Beispielsweise die hohe Modularität und eine mögliche Skalierung sprechen für diesen Ansatz. Als größten Nachteil stellen die Autoren die nötige Beschreibung der Schnittstellen dar. Durch die geforderte lose Kopplung des SUT und der Testumgebung (A5) reduzieren sich diese Schnittstellen deutlich.

1. https://github.com/chalmers-revere/opendlv (Abgerufen am 10.12.2024)

Busch et al. (2024) schreiben in ihrer Arbeit über die Verbesserung der Entwicklung von robotischen Applikationen mithilfe von mikroserviceorientierten Architekturen, zusammen mit einem automatisierten DevOps-Zyklus. Besonders die Ansätze der automatisierten Containerisierung über vorgefertigte Images und die gezeigte CI/CD könnte in dieser Arbeit ebenfalls zum Einsatz kommen (A13). Leider bezieht sich die komplette Automatisierung und die gezeigten Images lediglich auf die Verwendung von ROS, welches auf generelle Systeme angepasst werden müsste. Außerdem schreiben die Autoren, dass die Ansätze auch in der Domäne des automatisierten Fahrens in einem Fahrzeug-Testfeld getestet wurden. Die Autoren beschreiben nicht, welches SUT getestet wurde, sondern lediglich, dass Erfolge erzielt wurden.

5.5. Zusammenfassung der verwandten Arbeiten

Die Analyse der verwandten Arbeiten zeigt, dass viele Ansätze bereits Teilanforderungen erfüllen, jedoch keiner alle Anforderungen gleichzeitig adressiert. In der Tabelle 5.2 werden die referenzierten Anforderungen aus Kapitel 4 den jeweiligen Arbeiten zugeordnet, um den Erfüllungsgrad zu verdeutlichen. Es lässt sich erkennen, dass keine der gezeigten Arbeiten alle Anforderungen umsetzt, die sich aus Use-Case 1 ableiten (Anforderung 1 bis 10). Dennoch hebt sich die Arbeit von Brinkmann (2018) von den anderen ab, indem alle bis auf vier Anforderungen erfüllt werden. Die vorgestellten Ansätze können nicht direkt übernommen werden, sondern erfordern eine spezifische Anpassung von der maritimen Domäne zur Automobilindustrie. Die dort vorgestellte physische Testumgebung und die Ergänzung einer virtuellen Testumgebung von Hahn et al. (2015) können als Grundlage für das in dieser Arbeit verwendete Konzept verwendet werden. Die von Sievers et al. (2018) und Feilhauer et al. (2016) vorgestellten Ansätze, der Stimulation verschiedene Sensordaten (Kamerarohdaten oder Daten aus einem OTA-Sensor) und der hybriden Testumgebung (virtuell und physisch) finden ebenfalls Verwendung in dieser Arbeit. Die Erfüllung der aus Use-Case 2A und Use-Case 2B resultierenden Anforderungen, wie der Portierbarkeit des zu testenden Systems in eine HiL-Umgebung und die Automatisierung der Testumgebung, werden ebenfalls in Tabelle 5.2 gezeigt (Anforderung 11 bis 13.1). Auch hier lässt sich erkennen, dass keine der Arbeiten sich mit allen Anforderungen beschäftigt. Die von Reway et al. (2018) vorgestellte Teststrategie über die Wahl verschiedener Szenarien und Wetterbedingungen deckt sich sehr gut mit den geforderten Konfigurationen einzelner Testläufe in A13.1. Johansson und Paulsson (2024) setzen in ihrer Masterarbeit eine vollständige Toolchain für das Testen einer autonomen Fahrfunktion um. Besonders die Ansätze der Bereitstellung und Automatisierung dieser Toolchain über eine CI/CD in GitLab können in dieser Arbeit verwendet werden.

Tabelle 5.2.: Erfüllung der aus Use-Case 1, Use-Case 2A und Use-Case 2B abgeleitete Anforderungen durch verwandte Arbeiten. Ein Kreuz symbolisiert, dass die verwandte Arbeit keine Konzepte oder Implementierung verwendet, die für diese Anforderung genutzt werden kann. Ein Haken symbolisiert, dass Konzepte oder Implementierung genutzt werden können.

Arbeit	A1	A1.1	A1.2	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A13.1
Brinkmann (2018)	✓	✗	✓	✓	✗	✓	✓	✓	✓	✗	✗	✓	✗	✗	✗	✗
Hahn et al. (2015)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Sievers et al. (2018)	✓	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Feilhauer et al. (2016)	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Reway et al. (2018)	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓
Ruehl und Bronner (2024)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
Lotz et al. (2019)	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Busch et al. (2024)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
Johansson und Paulsson (2024)	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗

6. Konzept und Systemarchitektur der Testumgebung

Dieses Kapitel beschreibt die Konzepte und die Systemarchitektur der Testumgebung, die für den Test und die Evaluierung vom SUT entwickelt werden. Die Ansätze basieren auf bestehender gezeigter verwandter Arbeiten (Kapitel 5) und der dargestellte aktuelle Stand der Technik (Kapitel 2). In diesem Kapitel werden die definierten Anforderungen (Kapitel 4) referenziert, die als Grundlage des Lösungsansatzes dienen.

Ein Konzept oder konkreter eine Systemarchitektur legt die Werkzeuge der Gestaltung und dem Management verteilter Informationssysteme und ihre gegenseitigen Abhängigkeiten fest (Ness 2013, S.24). Dabei können zwischen verschiedenen Ebenen, wie der logischen oder technischen Systemarchitektur unterschieden werden (Wolf 2018, S.43). Es folgt zunächst die Beschreibung der Konzeptübernahme, Erweiterung und Anpassung aus der verwandten Arbeit von Brinkmann (2018) (Abschnitt 6.1). Anschließend wird die Systemarchitektur der Testumgebung (Abschnitt 6.2) und der einzelnen Teilsysteme (Abschnitt 6.4, Abschnitt 6.5 und Abschnitt 6.6) beschrieben.

6.1. Konzeptübernahme aus verwandter Arbeit

Wie bereits im Handlungsbedarf (Abschnitt 5.4) beschrieben, basiert das Konzept dieser Arbeit auf der Testfeld-Architektur von Brinkmann (2018). Diese Architektur basiert auf einer generischen Referenzarchitektur, die im Projekt ENABLE-S3 von Nickovic et al. (2017) vorgestellt wurde (siehe Unterabschnitt 2.3.2). Dabei wird das zu testende System strikt vom Testfeld getrennt. Brinkmann (2018) verfolgt in seiner Architektur das Ziel, ein möglichst generisches Testfeld zu schaffen, das für viele SUT verwendet werden kann. Dabei stellt er die Kommunikation zwischen dem Testfeld und dem zu testenden System, die sogenannte polymorphe Schnittstelle, in den Mittelpunkt seiner Arbeit. Diese Schnittstelle arbeitet als dynamischer Adapter, mit dem auch sich ändernde SUT zur Laufzeit in das Testfeld integriert und getestet werden können. Um dies zu ermöglichen, wird das Framework S-100 als einheitliches Datenmodell verwendet. Für die Kommunikation innerhalb des Testfelds wird eine nachrichten-orientierte Middleware verwendet. Die Kommunikation zwischen den Test-

feldkomponenten und der polymorphen Schnittstelle erfolgt über einen Datenverteiler, der als Sammelstelle einer Testfeldkomponente dient. Der Datenverteiler stellt alle Daten, wie z. B. Sensordaten, über die Middleware zur Verfügung und nimmt Steuerbefehle für die Aktoren entgegen. Zur Anreicherung des physischen Testfeldes mit virtuellen Simulationsdaten wird ein Simulationsadapter eingesetzt. Auch dieser verwendet die Middleware, um Daten zu versenden oder für eine Rückkopplung in die Simulation zu empfangen (Brinkmann 2018, S.72).

Das Konzept und die Systemarchitektur der Testumgebung in dieser Arbeit orientieren sich an der von Brinkmann (2018) vorgestellten Testfeldarchitektur, wobei zentrale Elemente übernommen und spezifische Elemente an die Anforderungen dieser Arbeit angepasst werden. Ein Kernelement dieser Übernahme ist die Aufteilung der gesamten Testumgebung und die strikte Trennung des SUT. Die Testumgebung wird in drei zentrale Hauptkomponenten unterteilt: das zu testende System, die Komponenten der Testumgebung und das Test- & V+V Management. Das SUT umfasst dabei die Hard- und Softwarekomponenten der Rechenplattform des ADSs, die für die Ausführung der automatisierten Fahrfunktionen verantwortlich sind. Physische Fahrzeugmodelle oder externe Sensorik sind explizit nicht Bestandteil des SUT, sondern werden durch Simulations- oder Testumgebungskomponenten repräsentiert. Die Testumgebung stellt dem ADS die Sensordaten zur Verfügung und empfängt die berechneten Steuerbefehle. Somit können auch Closed-Loop-Tests durchgeführt werden, die für die Entwicklung eines solchen Systems notwendig sind. Das Test- & V+V Management fasst die beiden Komponenten (V+V Management und Test Management) aus der Referenzarchitektur (ENABLE-S3) eines generischen Testfeldes für automatisierte CPS zusammen (Nickovic et al. 2017). In diesen Komponenten finden Messungen, Szenariensteuerung, Testberichtserstellung, Automatisierung etc. statt. Diese strikte Trennung ermöglicht einen modularen Aufbau der Testumgebung, bei dem die Komponenten der Testumgebung unabhängig vom SUT gestaltet werden können. Analog zu Brinkmann (2018) sind diese Komponenten und Methoden der V+V nicht Gegenstand dieser Arbeit.

Ebenso wird die Verwendung einer nachrichtenorientierten Middleware über ein Publish-Subscribe-Kommunikationsmuster aus dem Konzept von Brinkmann (2018) übernommen. Für das Format der versendeten Nachrichten wird ein einheitliches Datenformat aus der maritimen Domäne verwendet (Ward et al. 2008). Auch in dieser Arbeit wird ein einheitliches Datenmodell für die Nachrichtenkommunikation innerhalb der Testumgebung verwendet, jedoch wird ein Standard aus der Automobilindustrie verwendet.

6.1.1. Änderungsbedarf und Abgrenzung

Das von Brinkmann (2018) vorgestellte Konzept des physischen Testfeldes wurde für den maritimen Bereich entwickelt. Für den in dieser Arbeit vorgestellten Bereich der Automo-

bilindustrie und der autonomen Fahrfunktionen müssen einige Modifikationen vorgenommen werden. Einige Elemente der Testumgebung werden durch Elemente aus dem Automobilbereich ersetzt. Abbildung 6.1 zeigt eine Anpassung dieser Elemente der Testumgebung für ADS, die auf dem Testfeld von Brinkmann (2018) basiert. Die von Brinkmann vorgestellten Elemente des maritimen Testfeldes sind in Abbildung 5.1 zu finden. Die maritime Infrastruk-

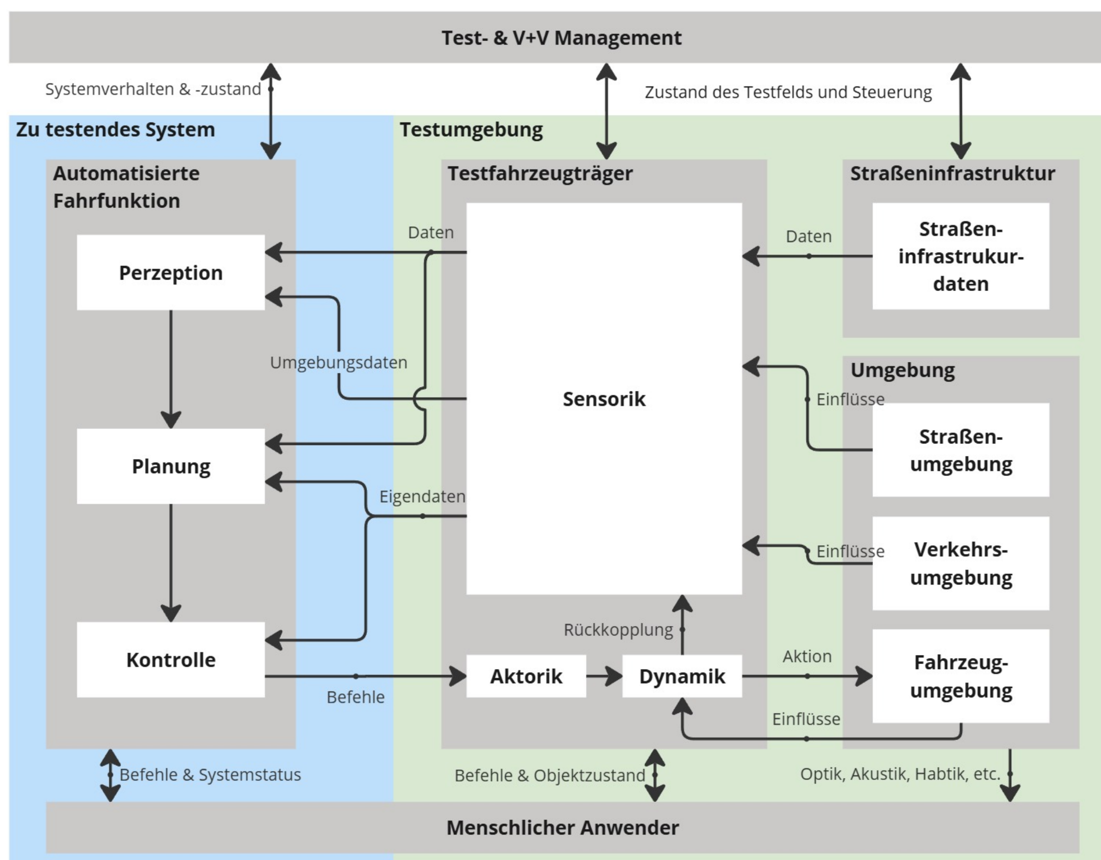


Abbildung 6.1.: Anpassung der Elemente einer Testumgebung für Autonomous Driving Systems, nach Brinkmann (2018, S.66)

tur und die maritime Umgebung ändert sich zur Straßeninfrastruktur und Umgebung. Die Daten der Infrastrukturdienste (z. B. V2X oder OpenDRIVE), welche im Konzept von Brinkmann (2018) über eine eigene Kommunikation an das zu testende System geleitet werden, werden in dieser Arbeit als Sensordaten über die Sensorik versendet (siehe A1). Ebenfalls ist die Datefusion nicht Teil des Testfahrzeugträgers. Die Fusion von Sensordaten ist häufig ein kritischer Teil der automatisierten Fahrfunktion und wird in der Perzeption durchgeführt (siehe Unterabschnitt 2.1.1).

Brinkmann (2018) konzentriert sich in seiner Arbeit auf das physische Testfeld und beschreibt nur kurz, wie ein virtuelles Testfeld integriert werden kann. Der Fokus dieser Arbeit

liegt auf der virtuellen Testumgebung, die durch Komponenten in der physischen Umgebung ergänzt wird. Die genannte Aufteilung in ein virtuelles (HAGGIS vorgestellt von Hahn et al. (2015)) und ein physisches (LABSKAUS vorgestellt von Brinkmann (2018)) Testfeld wird in der Arbeit von Rüssmeier et al. (2019) weitergeführt. Durch die nachrichtenorientierte Middleware ist es möglich, Komponenten in einer der beiden Umgebungen auszutauschen, ohne große Anpassungen in der Testumgebung vornehmen zu müssen. Die von Rüssmeier et al. (2019), Hahn (2015) und Brinkmann (2018) verwendete High Level Architecture (HLA) für das virtuelle Testfeld ist ein internationaler Standard nach IEEE 1516-2010 für verteilte Simulationen (IEEE 2010). In dieser Arbeit wird keine formale Implementierung einer HLA nach dem IEEE-Standard vorgenommen, jedoch werden einzelne Eigenschaften einer HLA in diesem Konzept berücksichtigt und in Abschnitt 6.2 näher erläutert.

Einer der Hauptaspekte der Arbeit von Brinkmann (2018) ist die bereits erläuterte polymorphe Schnittstelle. Ziel dieser Schnittstelle ist es, eine hohe Interoperabilität auf der Seite des zu testenden Systems zu erreichen. Sie ermöglicht es, das spezifische Format des SUT für die Ein- und Ausgabedaten in ein standardisiertes Format (S-100-Datenformat) umzuwandeln. Da sich das zu testende System ändern kann, wird ein dynamischer Adapter benötigt. Der Autor beschreibt ihn als „[...] Kopplungselement, das auf der Basis von Rahmeninformationen auf unterschiedliche maritime Standards wandelbar ist [...]“ (Brinkmann 2018, S.91). Für das Konzept dieser Arbeit wird die polymorphe Schnittstelle durch ein Adaptermodul ersetzt. Die Kommunikation zwischen der Testumgebung und dem zu testenden System erfolgt nicht über ein beliebiges spezifisches Format, welches (polymorph) transformiert werden muss, sondern über ein definiertes standardisiertes Datenformat. Das Adaptermodul hat somit die Aufgabe, die verschiedenen Sensordaten von den physischen und virtuellen Komponenten der Testumgebung über die Middleware zu empfangen, zu selektieren und an das SUT weiterzuleiten. Dabei werden die Daten bereits im richtigen Nachrichtenformat empfangen, müssen aber über ein spezifisches Kommunikationsprotokoll übertragen werden. Dies ist notwendig, um die geforderte lose Kopplung (A5) zu ermöglichen und den Entwickler des SUT nicht auf die Implementierung der gleichen Middleware festzulegen. Die Übertragung der Daten zum und vom SUT soll über ein einfaches Transmission Control Protocol erfolgen, das in jeder gängigen Programmiersprache implementiert werden kann. Hintergrund des gewählten Ansatzes ist die Interoperabilität der virtuellen und physischen Testumgebung. Dabei steht die Erweiterbarkeit durch das Hinzufügen z. B. neuer Sensoren im Vordergrund, die über ein einheitliches Nachrichtenformat und Protokoll an das SUT übertragen werden müssen. Gleichzeitig soll es möglich sein, einen anderen Simulator als Basis der virtuellen Testumgebung zu verwenden. So kann die Testumgebung z. B. um einen realistischeren Kamerasensor erweitert werden, ohne dass das SUT angepasst werden muss.

Zusätzlich ist eine Modifikation des Simulationsadapters erforderlich. Durch die bereits

erläuterte Aufteilung der Testumgebung in einen virtuellen und einen physischen Teil ist der Simulationsadapter und nicht der Datenverteiler der Hauptakteur. Er übergibt die Sensordaten über die Middleware an das Adaptermodul und empfängt die Steuerbefehle, die in die Simulation eingespeist werden. Der Datenverteiler in der physischen Testumgebung kann eigene Sensordaten von einem realen Sensor senden oder auch Steuerbefehle für einen Rollenstand empfangen. Da die Simulation die Grundlage der Testumgebung bildet, ist eine Rückkopplung der Steuerbefehle an das Ego-Fahrzeug in der Simulation notwendig, um den Regelkreis zu schließen. Die realen Sensoren werden mit Daten aus der Simulation stimuliert und die realen Aktoren werden gemessen und in die Simulation übertragen. Im Gegensatz zu Brinkmann (2018) steht hier die virtuelle Testumgebung im Vordergrund.

Brinkmann (2018) stellt in seiner Testfeld-Architektur eine Fehlerinjektion durch Manipulation vor. Diese spielen „[...] neben dem Testen der Erfüllung von Anforderungen und Systemeigenschaften eine wesentliche Rolle zur Überprüfung insbesondere von Randszenarien [...]“ Brinkmann (2018, S.83). Konkret werden Eingangssignale aus der Testfeldinfrastruktur von einem Manipulator an das SUT abgefangen und modifiziert weitergeleitet. Zusätzlich ist es möglich, neue Datenobjekte zu erzeugen und an das SUT zu senden. Da diese Funktionalitäten von den Use-Cases und den daraus abgeleiteten Anforderungen nicht gefordert werden, wird die Komponente im Konzept dieser Arbeit nicht berücksichtigt.

6.1.2. Erweiterung

Neben der Abgrenzung des in Brinkmann (2018) vorgestellten Konzepts sind zwei weitere Erweiterungen notwendig, um das in dieser Arbeit angestrebte Ziel zu erreichen und die Anforderungen zu erfüllen. Im Use-Case 2A und den dafür definierten Anforderungen wird eine Bereitstellung der verschiedenen Komponenten des SUT in einer SiL- und HiL-Umgebung gefordert. Dafür muss nach Anforderung A12 ein Bereitstellungsorchestrator verwendet werden. Da dieser aktiv zum Test-Management beiträgt, erfolgt eine Platzierung in der Überkomponente `Test & V+V Management`, die aus der Referenzarchitektur aus dem Projekt ENABLE-S3 stammt (Nickovic et al. 2017). Für die Bereitstellung der einzelnen Komponenten ist eine direkte Verbindung zum SUT in der HiL- und SiL-Umgebung notwendig. Dementsprechend ist das SUT auch Teil des Konzepts der Testumgebung und wird in Abschnitt 6.4 näher erläutert. Die zweite Erweiterung des Konzepts wird durch den Use-Case 2B und die dafür definierten Anforderungen gefordert. Die Anforderung A13 beschreibt einen Automatisierungsdienst, über den automatisierte Testläufe durchgeführt werden können. Dieser Dienst befindet sich ebenfalls in der Überkomponente `Test & V+V Management`. Da der Automatisierungsdienst sowohl das Szenario in der Test-/Simulationsumgebung als auch das zu testende System startet, wird eine Kommunikation zum Bereitstellungsorchestrator und zum Szenario benötigt. Der Automatisierungsdienst muss gemäß den Anforderungen mit

einem Versionskontrollsystem zusammenarbeiten. Der Automatisierungsdienst ist daher auf externe Kommunikation angewiesen. Mit diesen beiden Erweiterungen des Konzepts nach Brinkmann (2018) können alle Anforderungen und Use-Cases abgebildet werden.

6.2. Architektur der Testumgebung

Dieser Abschnitt gibt einen Überblick über die Systemarchitektur der Testumgebung, die zwischen der technischen und der logischen Sichtweise angesiedelt ist. Wie im vorherigen Abschnitt erläutert, basiert diese auf der von Brinkmann (2018) vorgestellten Testfeld-Architektur. Diese Übernahme, Erweiterungen und Abgrenzungen sowie die Funktionen der einzelnen Komponenten werden im folgenden Abschnitt beschrieben. Abbildung 6.2 zeigt die Systemarchitektur der Testumgebung zusammen mit dem zu testenden System. Zu erkennen sind die fünf Teilsysteme:

1. die Komponenten der virtuellen und physischen Testumgebung
2. das zu testende System (SUT)
3. das Adaptermodul
4. das Test- & V+V Management

Nach diesem Abschnitt folgt für jedes Teilsystem ein eigener Abschnitt, welches die Architektur und die gezeigte Abbildung schrittweise erweitert. Eine vollständige Darstellung der Architektur mit allen nachfolgend erläuterten Komponenten befindet sich in Anhang A. Das SUT ist über eine Schnittstelle und eine Middleware (Adaptermodul) in die Testumgebung integriert. Die Kernfunktionalität ist die Übertragung von Daten über die nachrichtenorientierte Middleware im einheitlichen Datenformat OSI. Dieses Datenformat wird in der gesamten Testumgebung als Nachrichtenformat verwendet. Das Management-System realisiert Funktionalitäten wie Monitoring, Auswertung, Automatisierung und die verschiedenen Szenarien, mit denen das SUT getestet werden kann. Die virtuellen Komponenten der Testumgebung beinhalten einen Simulationsadapter, der direkt mit der Simulation verbunden ist. Dieser generiert die simulierten Sensordaten und empfängt die Steuerbefehle als Rückkopplung. In der physischen Testumgebung können zusätzlich reale Sensoren und Aktoren eingesetzt werden. Diese werden wie bereits erwähnt von der Simulation stimuliert oder in diese gekoppelt.

6.2.1. Einheitliches Datenformat OSI

Angelehnt an die Architektur von Brinkmann (2018), wird auch in dieser Systemarchitektur ein einheitliches Datenformat verwendet (A1.2). Dies ermöglicht eine „wiederverwendebare

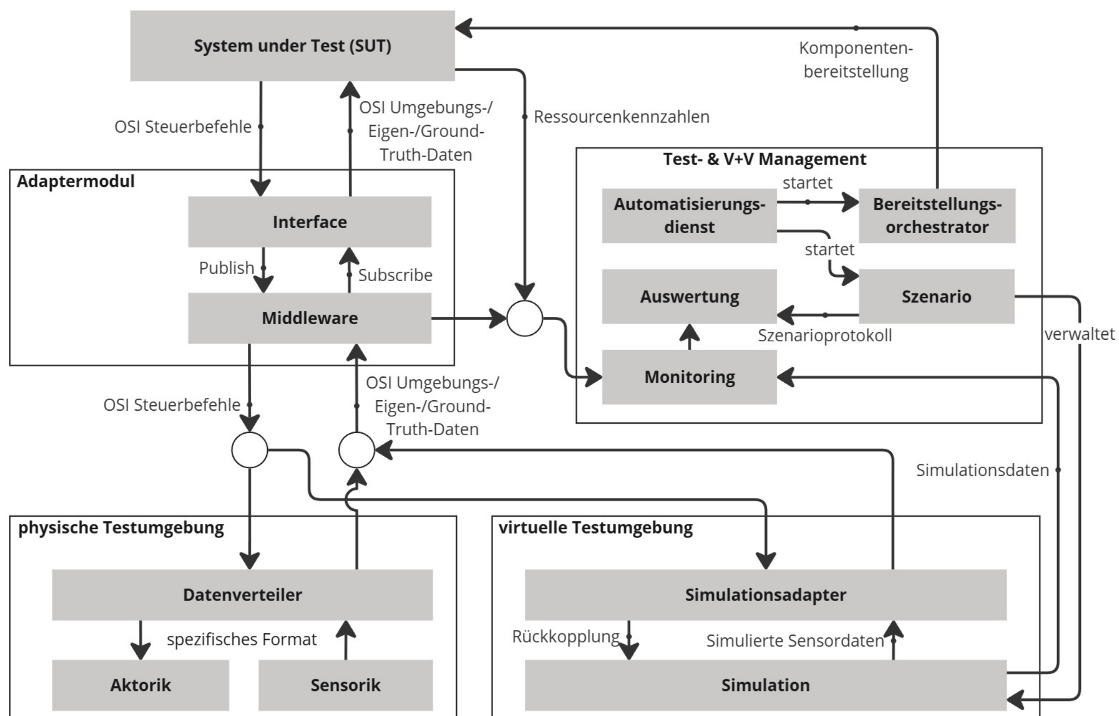


Abbildung 6.2.: Systemarchitektur der Testumgebung

und erweiterbare Testfeld-Architektur“ (Brinkmann 2018, S.69). Der im maritimen Bereich verwendete S-100-Standard muss durch einen Standard der Automobilindustrie ersetzt werden. In diesem Bereich sind die von ASAM e. V. entwickelten technischen Standards von großer Bedeutung. Der von ihnen in 2017 vorgestellte öffentliche Standard Open-Simulation-Interface (OSI) ermöglicht die Integration von automatisierten Fahrfunktionen und einer Vielzahl von Fahrsimulatoren, wie in Unterabschnitt 2.4.2 beschrieben. Beispielsweise kann für die Daten eines Kamerasensors die Klasse `osi3::CameraSensorView` [1] verwendet werden, die neben den Pixeldaten auch die Konfiguration des Sensors (eine ID, die Anzahl der Pixel, die Position des Sensors am Fahrzeug usw.) enthält (ASAM 2024c). Abbildung 6.3 zeigt die Struktur der Klasse `osi3::BaseMoving`, die Teil der oben genannten Klasse ist. Es ist zu erkennen, wie die Nachrichtenklasse strukturiert ist und welche Daten sie enthält.

Alternativ sind Nachrichtenstandards wie Scalable Service-Oriented Middleware over IP (SOME/IP) oder ROS möglich. SOME/IP wird als Middleware in Adaptive AUTOSAR verwendet (siehe Unterabschnitt 2.4.1). Ein großer Vorteil von OSI ist die spezifische Ausrichtung auf die Modellierung und Simulation automatisierter Fahrfunktionen. Während SOME/IP primär auf die Kommunikation im Fahrzeug zwischen Steuergeräten ausgelegt

1. https://opensimulationinterface.github.io/osi-antora-generator/asamosi/latest/gen/structosi3_1_1CameraSensorView.html (Abgerufen am 06.01.2025)

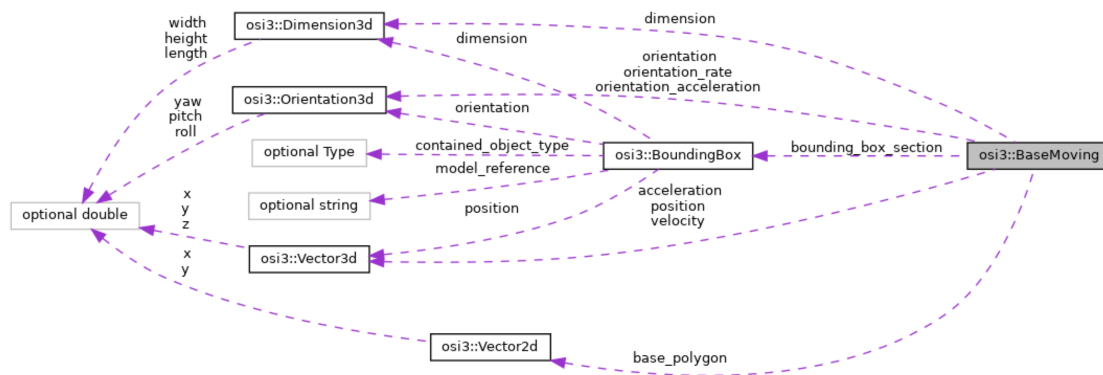


Abbildung 6.3.: Aufbau der Klasse osi3::BaseMoving (ASAM 2024c)

ist, fokussiert sich OSI auf die Beschreibung von Umweltwahrnehmung, Szenarien und Fahrzeugzuständen. ROS ist vor allem in der Robotik und Forschung weit verbreitet, wird aber aufgrund seiner Einfachheit zunehmend auch in Proof of Concepts für automatisierte Fahrfunktionen eingesetzt (Betz et al. 2024; Hong und Moon 2024; Paul et al. 2024). Die Verwendung von ROS würde den Entwickler des SUT zwingen, die Funktion in ROS zu implementieren, da ROS nicht nur ein Nachrichtenformat, sondern ein vollständiges Framework ist. Dies widerspricht der geforderten losen Kopplung und Unabhängigkeit der Architektur des SUT (A5). Aufgrund des geeigneten Kontexts und der weiten Verbreitung des Standards wird in dieser Arbeit OSI als einheitliches Datenmodell verwendet.

Der Einsatz von Closed-Loop-Tests erfordert nicht nur die Übertragung von Sensordaten an das zu testende System, sondern auch den Empfang von Steuerbefehlen (A4). OSI enthält keine vordefinierten Nachrichtenformate für Signale wie Beschleunigung, Bremsen oder Lenkbefehle, bietet aber Funktionen zur Erweiterung des Standards. Für die Lösung dieses Problems können zwei Herangehensweisen betrachtet werden. Entweder werden benutzerdefinierte OSI-Nachrichtenklassen verwendet oder es wird ein zweiter Standard für die Rückrichtung verwendet. Eine Erweiterung des OSI-Formats hat den Vorteil, dass die Entwickler der Fahrfunktion auch eigene Nachrichtenformate definieren können, ohne die bestehenden OSI-Konventionen zu verletzen. Die Einführung eines weiteren Standards für die Rückrichtung würde die Komplexität der Testumgebung ohne Mehrwert erhöhen. Die benutzerdefinierte OSI-Klasse für die Steuerbefehle (genannt `osi3::ControlCommand`) enthält Daten über: Beschleunigung, Bremskraft, Lenkbefehle, Licht- und Blinkersteuerung.

Neben der bereits erwähnten Klasse für Kamerasensordaten gibt es nicht nur weitere Klassen für gängige Sensoren (Radar, LiDAR, etc.), sondern auch Klassen für Ground-Truth-Daten. Diese Daten, wie Informationen über Objekte, Fahrzeuge oder Fahrlinien, können direkt aus der Simulation über die Klassen `osi::GroundTruth` oder `osi::DetectedMovingObject` übertragen werden. Wie in Unterabschnitt 2.1.1 beschrieben, kann eine

ADS-Funktion in die Teile Perzeption, Planung und Kontrolle unterteilt werden. Mithilfe der Ground-Truth-Daten ist es möglich ein Attrappe (engl. „mockup") der Perzeption zu erstellen, die oft den anspruchsvollsten Aspekt einer solchen Funktion darstellt (Ruthardt und Michalke 2022). Dieses Mockup empfängt die Ground-Truth-Daten über die OSI-Klasse und übersetzt sie in das vom Entwickler definierte Format der Planungskomponente. Abbildung 6.4 zeigt eine Mockup-Version einer Perzeptionskomponente mit den genannten Nachrichtenklassen. Wenn in einem späteren Entwicklungsstadium die erste Version einer

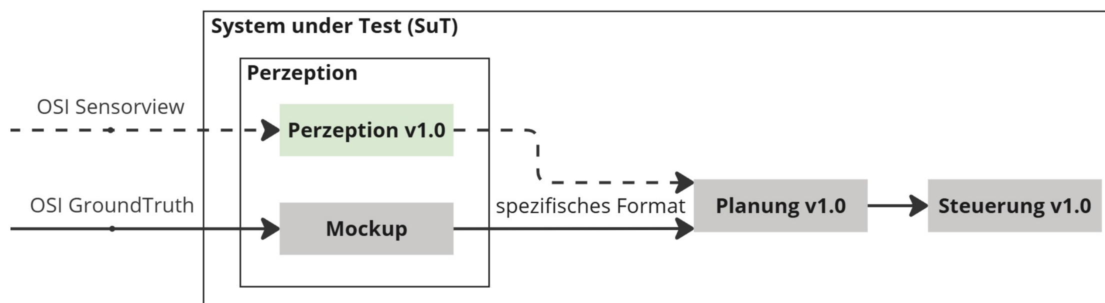


Abbildung 6.4.: Mockup-Version einer Perzeptionskomponente

Perzeptions-Komponente (`Perzeption v1.0`) implementiert wird, kann das Perzeptions-Mockup einfach ersetzt werden, ohne Änderungen in der Planungskomponente vornehmen zu müssen. Dadurch wird die Entwicklung des gesamten ADS verbessert.

6.2.2. Nachrichtenorientierte Middleware

Die Kommunikation zwischen dem Adaptermodul, der virtuellen und physischen Testumgebung sowie dem Test & V+V Management ist eine der Kernaspekte des in dieser Arbeit vorgestellten Konzepts. Um die Daten zwischen den einzelnen Teilen nicht über jeweils unterschiedliche Formate und Protokolle zu übertragen, ist die Verwendung einer gemeinsamen Middleware naheliegend. Eine Middleware ist „[...] eine Softwareschicht, welche auf Basis standardisierter Schnittstellen und Protokolle Dienste für eine transparente Kommunikation verteilter Anwedungen bereitstellt" (Österle et al. 2013, S.28). Der Einsatz einer Middleware unterstützt dabei die geforderte Interoperabilität (A6), die lose Kopplung und die Dezentralisierung (A7) der Testumgebung. Brinkmann (2018) stellt in seiner Testfeld-Architektur einen Vergleich zwischen synchroner Kommunikation, asynchroner Kommunikation und einem Publish-Subscriber-Kommunkationsmuster auf. Es wird gezeigt, dass eine „[...] nachrichtenorientierte Middleware mit einem Publish-Subscribe-Kommunikationsmuster [...] eine hinreichende Entkopplung bezüglich der Aspekte der Zeit, Raum und Synchronisation der Komponenten [...]" (Brinkmann 2018, S. 75) darstellt. Diese Aspekte treffen auch auf die Anforderungen dieser Arbeit zu. In einer nachrichtenorientierten Middleware mit einem Publish-

Subscribe-Kommunikationsmuster können verschiedene Komponenten in der Testumgebung separat miteinander kommunizieren, ohne die konkrete Implementierung der gegenüberliegenden Komponente zu kennen. Abbildung 6.5 zeigt die notwendigen Funktionen, um Daten im OSI-Format über die Middleware zu transportieren. Zunächst werden die Daten von der

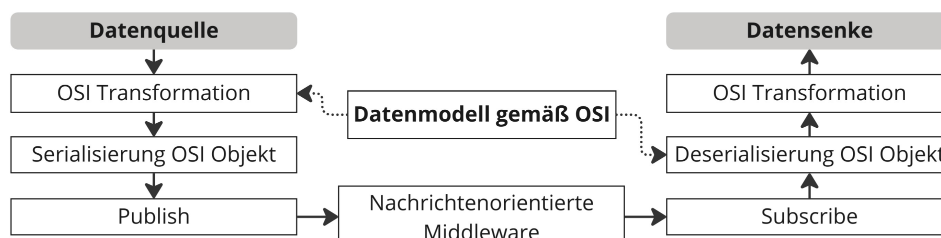


Abbildung 6.5.: Veröffentlichen und Abonnieren von Daten in der Testumgebung, nach Brinkmann (2018, S.76)

Datenquelle (beispielsweise einem Kamerasensor in der Simulation) in das einheitliche Datenformat OSI übertragen. Dabei können neben den eigentlichen Daten auch zusätzliche Informationen in das OSI-Objekt übertragen werden (Kameraauflösung, Kameraposition, etc.). Um die Leistung bei der Übertragung der Daten zu erhöhen, werden die OSI-Objekte in ein sequentielles Format serialisiert. Analog zu dieser Verarbeitungskette existiert auf der Empfangsseite eine Subscribe-Komponente, die die Informationen über die nachrichtenorientierte Middleware empfängt. Nach der Deserialisierung in ein OSI-Objekt werden die Nachrichten in das spezifische Format der Datensenke transformiert.

6.3. Komponenten der virtuellen und physischen Testumgebung

Wie bereits in der Einleitung zu diesem Kapitel beschrieben, ist die Testumgebung in physische und virtuelle Komponenten unterteilt. Das Konzept verfolgt dabei die Idee, die virtuelle Umgebung mit Daten aus der physischen Umgebung zu ergänzen, um realistischere Tests zu ermöglichen. Abbildung 6.6 erweitert die bereits in Abbildung 6.2 vorgestellte physische und virtuelle Testumgebung. Der Simulationsadapter als virtuelle Komponente und der Datenverteiler als physische Komponente können jeweils als Sammelstelle für Daten und als Kommunikator mit der nachrichtenorientierten Middleware betrachtet werden. Wie bereits beschrieben, stellt die Simulation die Basis der gesamten Testumgebung dar. Daher ist es notwendig, auch die physischen Komponenten der Sensorik und Aktorik mit ihr zu verbinden. Es folgt eine Erläuterung der einzelnen Komponenten.

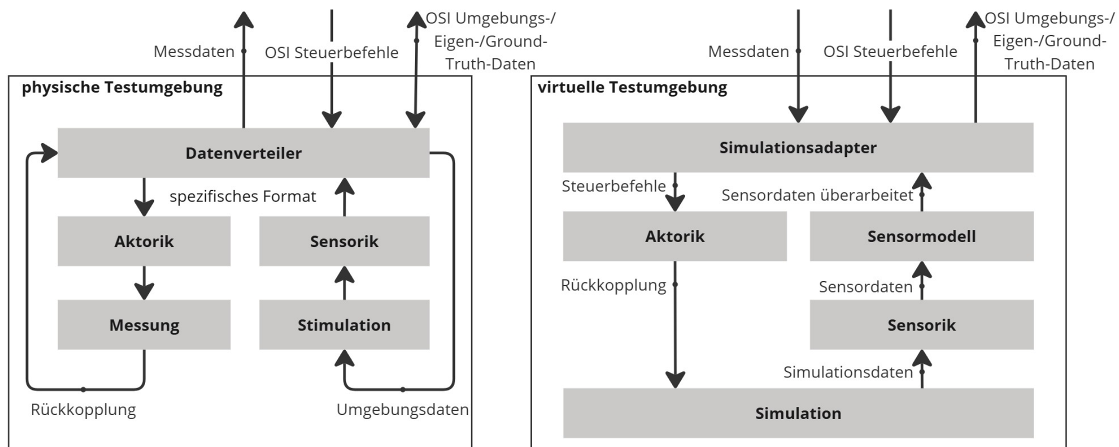


Abbildung 6.6.: Erweiterung der bereits gezeigten Systemarchitektur um die Komponenten der physischen und virtuellen Testumgebung

Simulationsadapter und Datenverteiler Der Simulationsadapter empfängt die Steuerbefehle (A4) vom SUT und die Messdaten aus der physischen Testumgebung über die Middleware. Gleichzeitig versendet er die Umgebungs-/Ground-Truth- und Eigendaten (A1, A2, A3) im OSI-Format an die Middleware. Je nachdem, ob eine Messung in der physischen Aktorik stattfindet, werden diese Messdaten oder die direkten Steuerbefehle des SUT als Rückmeldung an die Aktorik in der Simulation übertragen. Dazu ist eine Konvertierung des benutzerdefinierten OSI-Formats in das spezifische Format des Simulators notwendig. Die gleiche Funktion übernimmt der Datenverteiler als Bestandteil der physischen Testumgebung. Er bindet reale Sensorik und Aktorik in die Testumgebung ein. Damit ist es möglich, Daten von ViL-Testständen, z. B. Messungen von einem Rollenprüfstand, in dieser Testumgebung zu verwenden. Der bereits in den Anforderungen erwähnte OTA-Sensor (A1.1) ist ein realer Kamerasensor, der als physische Komponente eingesetzt wird. Da der Simulator die Basis der Tests darstellt, muss für den Einsatz von Closed-Loop-Tests die physische Sensorik durch den Simulator stimuliert und die Aktorik durch eine Messung rückgekoppelt werden. Dies geschieht ebenfalls über den Datenverteiler, der die Umgebungsdaten über die Middleware an die Stimulation sendet, die Messdaten empfängt und über die Middleware zur Verfügung stellt. Im Bereich der kamerabasierten Sensoren ist dies über eine Visualisierung der Daten auf einem Monitor vor dem Sensor möglich (Sievers et al. 2018; Reway et al. 2018).

Sensorik, Sensormodelle und Stimulation Die Sensorik als Komponente der virtuellen Testumgebung beschreibt die konkrete Implementierung eines Sensors in der Simulation. Die simulierten Umgebungsdaten werden über ein spezifisches Format aus dem Simulator bezo-

gen. Für eine verbesserte Entwicklung von ADS werden in der Anforderung A1 neben den perfekten Sensordaten aus der Simulation auch realitätsnähere Sensordaten gefordert. Diese werden durch verschiedene Sensormodelle in der Architektur berücksichtigt. Dabei wird wiederum auf die Funktionen von OSI zurückgegriffen. Neben einer Modifizierung von Sensordaten werden in der Literatur Sensormodelle beschrieben, die Ground-Truth-Informationen in detektierte Objektlisten im OSI-Format konvertieren (Marko et al. 2019). Diese Möglichkeit ist in dieser Arbeit aufgrund der losen Kopplung und der Betrachtung des SUT als Blackbox nicht möglich, wird aber in der Diskussion wieder aufgegriffen. In dieser Architektur beschränkt sich das Sensormodell auf die Modifikation der Sensordaten, um einen realitätsnäheren Sensor zu beschreiben. Um das Hinzufügen zukünftiger Sensormodelle zu ermöglichen, wird für die Modelle eine Schnittstelle mit den konkreten Datentypen und Parametern definiert. Als OSI-Format wird die allgemeine Klasse `osi3::SensorView` verwendet, die Sensordaten und Sensorkonfigurationen von Kamera-, Radar-, LiDAR-, Ultraschall- und generischen Sensoren enthalten kann. Dadurch ist es möglich, die Testumgebung mit weiteren Sensoren und Sensormodellen auszustatten, ohne die Schnittstelle anpassen zu müssen.

FMI und FMU Im Rahmen der Konzeptentwicklung wurde auch der Functional Mock-up Interface (FMI)-Standard[1] als mögliche Grundlage für die Architektur in Betracht gezogen. FMI ist ein offener Standard, der veröffentlicht wurde, um den Integrationsaufwand zwischen verschiedenen Simulationsmodellen zu reduzieren. Damit ist es möglich, Simulationswerkzeuge und Modelle (Silva und Antonino 2024) zusammenzuführen. Dieser Austausch erfolgt durch eine Beschreibung der Schnittstelle als XML, wobei die konkrete Funktion als ausführbare Functional Mock-up Unit (FMU) implementiert wird (Brinkmann 2018, S.88). Insbesondere die Verwendung von FMUs erlaubt eine standardisierte Integration von Modellen und Co-Simulationen in verschiedenen Umgebungen, was auch für das Konzept dieser Arbeit relevant sein könnte. Dennoch wird in dieser Arbeit der Schwerpunkt auf eine einfachere, direkte Implementierung unter Verwendung von Python und ROS gelegt, die im Kapitel 7 erläutert wird. Gründe dafür sind die spezifischen Anforderungen der Arbeit, die eine schnelle und flexible Entwicklung erforderten, sowie der zusätzliche Aufwand, der mit der vollständigen Implementierung einer FMI-basierten Architektur verbunden war. Das vorgestellte Konzept bleibt jedoch offen für zukünftige Erweiterungen, die auf FMI-basierte Module zurückgreifen könnten, primär um die Wiederverwendbarkeit und Interoperabilität mit anderen Simulationssystemen zu fördern. Die vorgestellte Architektur orientiert sich daher an einigen Grundprinzipien von FMI, ohne jedoch den Standard explizit zu implementieren.

1. https://fmi-standard.org/ (Abgerufen am 11.01.2025)

6.4. System under Test (SUT)

Das SUT ist eine Kombination aus Hard- und Software, die das ADS widerspiegelt. Im Gegensatz zur vorgestellten Testarchitektur von Brinkmann (2018) wird neben der Kommunikation vom/zum SUT auch die interne Netzwerkstruktur in der Systemarchitektur vorgestellt.

Wie bereits im Handlungsbedarf der verwandten Arbeiten beschrieben und von A5 gefordert, findet eine strikte Trennung des zu testenden Systems und der Testumgebung statt. Anders als in den vorgestellten Arbeiten von Sievers et al. (2018) und Feilhauer et al. (2016) wird das SUT von der Generierung der Eingabedaten (Sensordaten) getrennt. Die von ihnen vorgestellten Sensor-ECUs bilden die gesamte Verarbeitungskette, von der Datengenerierung des Sensors bis zur Berechnung der Trajektorie, ab. Dies steht im Widerspruch zu der geforderten losen Kopplung und der Möglichkeit einzelne Komponenten auf verschiedener Hardware (HiL, SiL) auszuführen (A11). In diesem Sinne wird das zu testende System in dieser Arbeit als eine Softwarekomponente betrachtet, die auf einer bestimmten Hardware ausgeführt wird. Gleichzeitig wird das SUT von der Generierung der Sensordaten (durch physische oder virtuelle Sensoren) und deren Modifikation durch Sensormodelle getrennt. Es enthält auf einem konkreten Kommunikationskanal definierte Sensordaten und gibt auf einem anderen Kanal die berechneten Steuerbefehle zurück. Soll beispielsweise ein Lane-Follower als SUT getestet werden, so sendet die Testumgebung die erforderlichen Kameradaten (Pixeldaten) an das SUT. Es ist möglich, dass diese Daten in der Testumgebung bereits durch ein Sensormodell modifiziert wurden, um realitätsnähere Daten zu erhalten, es handelt sich jedoch um Rohdaten. Diese Option wird in der Literatur von Feilhauer et al. (2016) als „Raw Video Data“ und von Sievers et al. (2018) als „Raw Data“ bezeichnet. Bezogen auf einen physischen Sensor sind dies die Daten nach dem Bildsensor, der Kameralinse und dem Objektiv. Im SUT wird auf Basis dieser Kameradaten die Verarbeitungskette gestartet, in der z. B. das Bild über eine Vorverarbeitung in ein Schwarz-Weiß-Bild umgewandelt wird, dann eine Linienerkennung stattfindet und daraus der Lenkwinkel berechnet wird. Abbildung 6.7 zeigt eine mögliche Aufteilung vom SUT, bei der unterschiedliche Softwarekomponenten im SiL und HiL ausgeführt werden. Diese Abbildung ist eine Erweiterung der bereits gezeigten Abbildung 6.2. In der Literatur werden HiL und SiL häufig als Ansätze zur Entwicklung beispielsweise von CPSs dargestellt (Riedmaier et al. 2018). Wie bereits in Unterabschnitt 2.2.2 beschrieben, können damit dynamische Tests mit unterschiedlichen Simulationsstufen durchgeführt werden. In dieser Arbeit wird eine eingeschränkte Definition dieser Begriffe verwendet. Die HiL- und die SiL-Umgebung unterscheiden sich lediglich in der verfügbaren Ressourcenkapazität. Die HiL-Umgebung beschreibt eine reale Fahrzeughardware mit Ressourcenbeschränkungen. In ihr können einzelne oder alle Komponenten des zu testenden Systems ausgeführt werden, um den Ressourcenbedarf der Komponenten zu testen. Da in dieser Umgebung ein eingebettetes

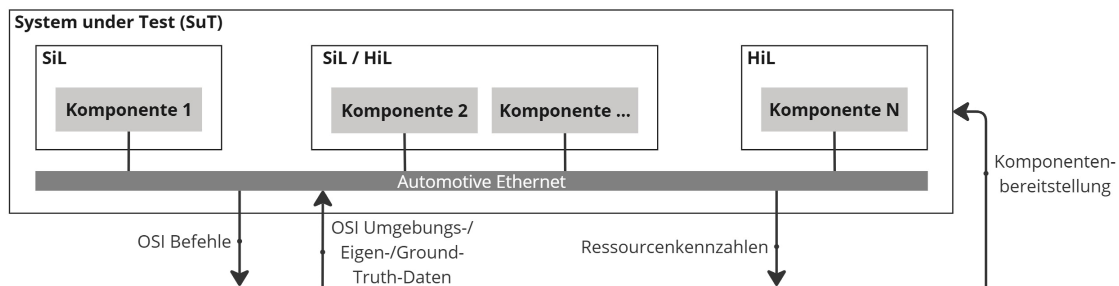


Abbildung 6.7.: Softwarekomponenten des Systems under Test (SUT) und Aufteilung in die HiL- und SiL-Umgebung

System verwendet wird, kann der Begriff HiL präziser als PiL konkretisiert werden. Die SiL-Umgebung hingegen wird als Plattform genutzt, auf der die Softwarekomponenten nahezu ohne Ressourcenbeschränkung ausgeführt werden können, z. B. auf einem leistungsfähigen Server/PC. Wie bereits beschrieben und durch Anforderung A11 gefordert, müssen die Softwarekomponenten des SUT ohne Anpassung in der HiL- und SiL-Umgebung bereitgestellt werden. Ebenfalls wird der Einsatz von Containertechnologien vorausgesetzt, welches die Portierung und Funktionen der Interoperabilität (A6) ermöglicht. Diese Entscheidung bringt einige Einschränkungen für die Entwicklung und Konfiguration des zu testenden Systems mit sich. Echtzeitfunktionen wie Interrupts lassen sich zwar auch in Containern umsetzten (z. B. über Real-Time Operating System (RTOS) (Jaikamal 2009)), können jedoch nicht alle Funktionen einer nativen Entwicklung abbilden. Ebenso ist die Trennung der HiL-Hardware in verschiedene Partitionen für unterschiedliche Komponenten des ADS nicht ohne weitere Konfiguration möglich (Casini et al. 2022). Dieser durch die Anforderungen erzwungene Trade-off für eine nahtlose Bereitstellung in den verschiedenen Umgebungen wird im Fazit erneut aufgegriffen.

Automotive Ethernet. In Abbildung 6.7 ist neben den Komponenten auch das Netzwerk des SUT, Automotive Ethernet zu erkennen. Wie bereits in Unterabschnitt 2.4.2 beschrieben, ist Automotive Ethernet eine spezielle Weiterentwicklung auf Basis des Ethernet-Standards IEEE 802.3. Aufgrund der einfachen Implementierung und der im Vergleich zu CAN oder FlexRay deutlich höheren Bandbreite wird Automotive Ethernet in der Literatur als Zukunftskandidat für die fahrzeuginterne Kommunikation bezeichnet (Lo Bello et al. 2023; Zinner 2020; Deng et al. 2022). Auch in dieser Architektur des zu testenden Systems wird Automotive Ethernet als Netzwerkstandard eingesetzt. Im Gegensatz zu CAN oder FlexRay ermöglicht Automotive Ethernet die Übertragung großer Datenmengen, wie z. B. Sensordaten, zwischen den Komponenten im SUT. Diese Fähigkeit, Sensordaten oder verarbeitete Sensordaten zu übertragen, erhöht die Flexibilität des Systems. Sie ermöglicht den Einsatz

einer mikroserviceorientierten Architektur, in der spezifische Services für die Vorverarbeitung definiert werden können. Über diese Services werden die OSI-Sensordaten vom Adaptermodul gesendet und die OSI-Steuerbefehle empfangen. Gleichzeitig dient das Ethernet-Netzwerk als Plattform für die interne Kommunikation des zu testenden Systems. Die Entwickler des zu testenden Systems können frei entscheiden, welches konkrete Ethernet-basierte Protokoll (Transmission Control Protocol (TCP), User Datagram Protocol (UDP), DDS, Hypertext Transfer Protocol (HTTP), usw.) sie für die Kommunikation innerhalb der Komponenten verwenden.

6.5. Adaptermodul

Aus den in Kapitel 4 definierten Anforderungen und den bereits gezeigten Konzepten dieser Architektur folgt der Bedarf eines Adapters (im folgenden Adaptermodul bezeichnet). Diese Komponente agiert als Schnittstelle zwischen der Testumgebung und dem zu testenden System. Ziel ist es, die Daten über die nachrichtenorientierte Middleware an das lose gekoppelte SUT zu senden und umgekehrt. Ein Adapter bezeichnet in der Softwareentwicklung ein Entwurfsmuster und agiert als Kommunikator zwischen zwei Schnittstellen. Der Adapter übernimmt dabei bestimmte Übersetzungen, die zwischen den Schnittstellen notwendig sind (Gamma et al. 2015, S.171 f.). Der Fokus liegt dabei auf einer protokolltechnischen Adaption zwischen den Schnittstellen, bei dem das Adaptermodul zwei inkompatible Kommunikationsmechanismen und Protokolle miteinander verbindet. Er ermöglicht zusätzlich eine physische und netzwerkorientierte Trennung zwischen dem zu testenden System (SUT) und der Testumgebung. Auf der Seite der Testumgebung empfängt und versendet das Adaptermodul Informationen über das nachrichtenorientierte Publish-Subscribe-Kommunikationsmuster (Middleware). Aufgrund der geforderten und bereits beschriebenen strikten Trennung zwischen Testumgebung und zu testendem System wird diese Kommunikation nicht direkt an das SUT weitergeleitet. Wie bereits in Abschnitt 6.4 beschrieben, agiert das zu testende System auf einem separaten Bussystem, für das möglicherweise spezielle Anforderungen erfüllt werden müssen. Ein Beispiel hierfür sind die Standards des TSN nach IEEE 802.1, die die Echtzeitfähigkeit von Netzwerken gewährleisten (Deng et al. 2022). Erst durch die Netzwerktrennung, die durch das Adaptermodul gewährleistet wird, ist es möglich, spezielle Anforderungen wie die Echtzeitfähigkeit nach den TSN-Standards zu erfüllen.

Durch den Einsatz des Adaptermoduls wird die Kommunikationskette zwischen der Datenquelle und der Datensenke aus Abbildung 6.5 erweitert. Die Erweiterung dieser Kette ist in Abbildung 6.8 dargestellt. Von der Datenquelle bis zum Subscribe der nachrichtenorientierten Middleware bleibt die Kommunikation gleich. Nach dem Subscribe werden die Daten jedoch nicht deserialisiert, sondern vom Adaptermodul über ein spezifisches Kommunikationsproto-

koll an das SUT gesendet. Erst dort wird das OSI-Objekt deserialisiert und transformiert.

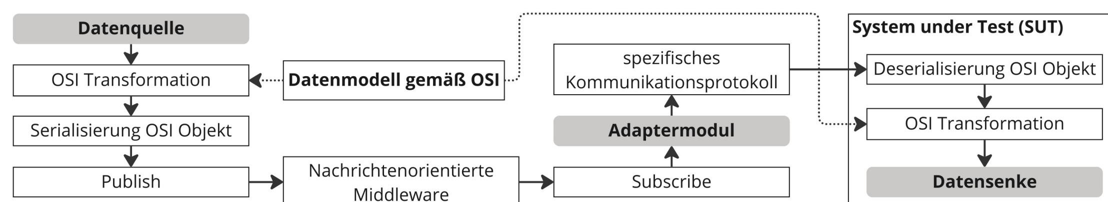


Abbildung 6.8.: Erweiterung der Nachrichtenkommunikationskette zwischen Datenquelle und ADS-Funktion durch das Adapatermodul

Dieses spezifische Kommunikationsprotokoll läuft auf dem bereits erwähnten Netzwerk/Bussystem, über das bestimmte Anforderungen realisiert werden können. Das Adaptermodul wird auf der Seite des SUT an dieses Netzwerk/Bussystem angeschlossen. Konkret wird ein separates (Automotive) Ethernet als spezifisches Netzwerk/Bussystem verwendet. Dieser Ansatz ermöglicht Flexibilität bei der Wahl des Bussystems. Beispielsweise wäre es auch möglich, die Sensordaten über Automotive Ethernet zu kommunizieren und die Steuersignale über einen CAN-Bus zu empfangen. Dies ermöglicht nicht nur eine logische, sondern auch eine physische und netzwerktechnische Trennung der Testumgebung vom SUT. Somit ist es auch möglich, andere Bussysteme oder Kommunikationsprotokolle in Richtung SUT zu verwenden, ohne die Testumgebung anpassen zu müssen.

Die in der Anforderung A1.1 geforderte Auswahl der verschiedenen Eingabedaten (perfekte Kameradaten, realitätsnähere Kameradaten und OTA-Kameradaten) wird ebenfalls vom Adaptermodul übernommen. Diese Funktion wird in der Nachrichtentechnik als Multiplexer bezeichnet und sorgt dafür, dass die Sensordaten immer über den gleichen Kanal an das SUT gesendet werden. Dadurch ist es möglich, verschiedene Testläufe mit dem SUT durchzuführen und die Eingabedaten für jeden Testlauf zu ändern, ohne die Testumgebung neu bereitstellen zu müssen. Abbildung 6.9 zeigt die beiden genannten Funktionen des Adaptermoduls als Blockdiagramm. Ebenfalls kann durch das Adaptermodul eine hybride Testumgebung ermöglicht werden. Wie in dem von Feilhauer et al. (2016) vorgestellten Konzept erlaubt ein hybrider Ansatz die Verknüpfung von Sensordaten aus virtuellen Komponenten mit Daten aus physischen Komponenten der Testumgebung. Beispielsweise kann ein ADS mit Kameradaten aus einem OTA-Sensor und Radardaten aus einem rein virtuellen Sensor getestet werden.

6.6. Test-, Verifikations- und Validierungs-Management

Zwei weitere Teilsysteme, welches durch die Referenzarchitektur von ENBALE-S3 (Nickovic et al. 2017) vorgestellt werden, sind das Test-Management und das V+V-Management. In

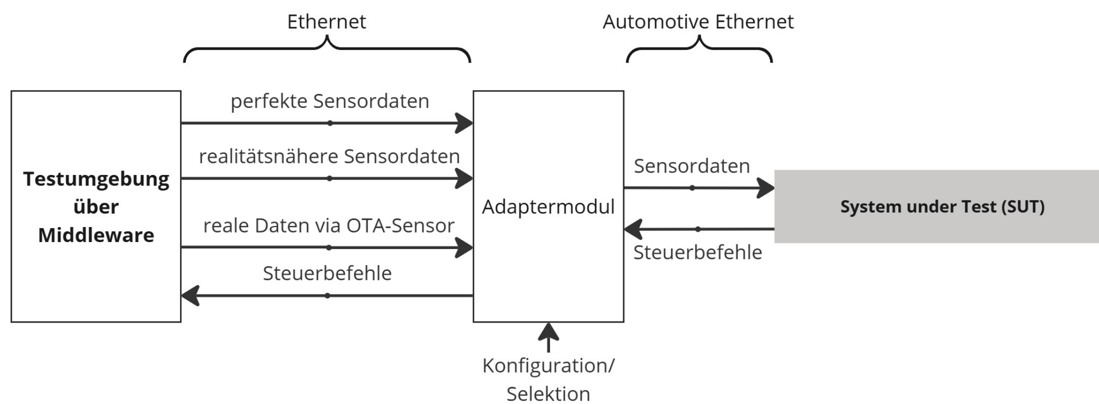


Abbildung 6.9.: Trennung zwischen der Testumgebung und dem zu testenden System durch das Adaptermodul

der Testfeld-Architektur von Brinkmann (2018) werden diese beiden Teilsystem in ein System zusammengefasst. Wie bereits in der Konzeptübernahme beschrieben, sind die Methoden des V+V nicht der Fokus dieser Arbeit. In diesem Teilsystem befinden sich Komponenten und Schnittstellen, die dem Test- & V+V Management beitragen. Die Konzepte und Ansätze dieser Komponenten werden im Folgenden erläutert.

6.6.1. Monitoring

Ein zentraler Aspekt des V+V-Managements ist die Überwachung (engl. „monitoring“) der gesamten Testumgebung und des zu testenden Systems. Diese Komponente hat direkten Zugriff auf die nachrichtenorientierte Middleware und kann somit alle gesendeten Informationen (Umgebungsdaten, Eigendaten, Ground-Truth-Daten, Messdaten und Steuerbefehle) empfangen. Da die Bewertung und Überwachung eines SUT stark von der konkreten Aufgabenstellung des zu testenden Systems abhängt, ist das gesamte Monitoring als Schnittstelle konzipiert (A8). Dies ermöglicht es dem Entwickler des SUT, eigene Metriken zu implementieren und zu überwachen, um die Leistung oder Korrektheit des Systems zu überprüfen. Durch die Verwendung des standardisierten Datenformats OSI und der definierten Nachrichtenstruktur ist eine einfache Nutzung der Schnittstelle möglich. Neben der Beobachtung der Ein- und Ausgabedaten des SUT wird von A10 auch eine Visualisierung dieser Daten gefordert. Dies ermöglicht die Beobachtung der Fahrfunktion und des Verhaltens des Fahrzeugs in der Simulation zur Laufzeit, was insbesondere bei Closed-Loop-Tests von Bedeutung ist.

Durch die geforderte Portierbarkeit (A11) einzelner Komponenten des SUT in eine HiL-Umgebung mit begrenzten Ressourcen sind Ressourcenkennzahlen für die Entwicklung dieser Komponenten relevant. Dabei kann der Entwickler des zu testenden Systems über die Monitoring-Schnittstelle auf Kennzahlen wie CPU-Auslastung, Speicherauslastung, Netz-

werkauslastung etc. zugreifen. Mit diesen Informationen ist es möglich, den Ressourcenverbrauch einzelner Komponenten des SUT zu beobachten und zu analysieren.

6.6.2. Szenarien

Das Testen von Fahrfunktionen in der Simulation erfordert die Definition und Ausführung spezifischer Szenarien, die verschiedene Verkehrssituationen und Umweltbedingungen repräsentieren. Der in der Literatur verwendete Begriff des „szenariobasierten Testens" beschreibt einen Ansatz, bei dem Tests auf der Grundlage vordefinierter Szenarien durchgeführt werden, um das Verhalten des SUT unter realistischen Bedingungen zu evaluieren (Brade et al. 2021). Obwohl sich diese Arbeit an Konzepten des szenariobasierten Testens orientiert, wird nicht der vollständige Ansatz in das Konzept der Testumgebung übernommen. Anstelle einer umfassenden Abdeckung aller möglichen Szenarien, steht die gezielte Bewertung spezifischer Fahrfunktionen unter definierten Bedingungen im Vordergrund. Dabei werden zentrale Prinzipien des szenariobasierten Testens, wie die Verwendung standardisierter Szenarien zur Reproduzierbarkeit und Vergleichbarkeit, berücksichtigt, jedoch keine Methoden zur systematischen Generierung und Variation betrachtet.

Ähnlich wie beim Monitoring hängen die Szenarien stark vom zu testenden System und den darin enthaltenen Funktionen ab. Für die Erweiterbarkeit und einfache Anpassbarkeit der Testumgebung wird in dieser Arbeit der Standard OpenSCENARIO verwendet (ASAM 2024b). Wie das OSI-Format wurde auch dieser Standard von ASAM entwickelt. Er enthält die Beschreibung komplexer, synchronisierter Manöver, an denen verschiedene Verkehrsteilnehmer (Fahrzeuge, Fußgänger, Objekte) beteiligt sind (siehe Unterabschnitt 2.4.2). Für die logische Straßenstruktur wird der Standard OpenDRIVE verwendet, der ebenfalls von ASAM stammt. OpenDRIVE bietet ein Modell zur präzisen Beschreibung von Straßenlayouts, Kreuzungen und anderen infrastrukturellen Details und schafft damit eine konsistente und standardisierte Grundlage für Szenarien (ASAM 2024a). In Kombination ermöglichen diese Standards eine umfassende Modellierung eines Szenarios. Für die Ausführung und Definition der Szenarien ist eine Szenario-Komponente vorgesehen, die in Abbildung 6.2 dargestellt ist. Sie startet, stoppt und führt das OpenSCENARIO auf der OpenDRIVE-Karte im Simulator der Testumgebung aus. Entwickler können somit für ihre Fahrfunktion angepasste Szenarien im OpenSCENARIO-Format definieren und hinzufügen. Beim manuellen Start kann für jeden Testlauf ein bestimmtes Szenario ausgewählt werden.

6.6.3. Bereitstellungsorchestrator und Automatisierungsdienst

Entsprechend dem Use-Case 2A und den dafür definierten Anforderungen (A11, A12), müssen die einzelnen Komponenten des zu testenden Systems in verschiedenen Umgebungen (HiL

und SiL) bereitgestellt werden. Diese Entscheidung trifft der Entwickler für jede einzelne Komponente des SUT. Dies ermöglicht reproduzierbare Testläufe, bei denen einzelne Komponenten des zu testenden Systems schrittweise von der SiL-Umgebung in die HiL-Umgebung portiert werden, um die Ressourcennutzung einzelner Komponenten zu testen. Der Bereitstellungsorchestrator benötigt dafür direkten Zugriff auf das laufende Betriebssystem der HiL- und SiL-Umgebung. Wie bereits erläutert, werden die Softwarekomponenten des SUT als einzelne Container entwickelt. Eine Orchestrierung dieser Container ist über Technologien wie Kubernetes[1] oder Docker Swarm[2] möglich. Dieser Ansatz der Orchestrierung gilt insbesondere in mikroserviceorientierten Architekturen als aktueller Stand der Technik (Vayghan et al. 2019; Berger et al. 2017) (Vohra 2016, S.41f).

Für das Ziel einer verbesserten Entwicklung und Evaluation von CPS spielt die Automatisierung der Testumgebung eine große Rolle (A13). Diese ermöglicht es dem Entwickler, z. B. nach der Entwicklung einer neuen Version, schnelle und automatisierte Testläufe durchzuführen (siehe Use-Case 2B). Nach Abschluss eines Testlaufs wird ein Testprotokoll zur Verfügung gestellt, das die Leistung des SUT aufzeigt. Für die Umsetzung einer solchen Testautomatisierung wird ein Automatisierungsdienst zusammen mit einem Versionskontrollsystem benötigt. Diese müssen folgende Funktionen erfüllen:

1. Das Starten des SUT über den Bereitstellungsorchestrator
2. Das Starten aller Komponenten der Testumgebung (Adaptermodul, virtuelle und physische Komponenten)
3. Das Laden und Starten des OpenSCENARIOS im Simulator
4. Nach Durchlauf des Szenarios, die Generierung eines Testprotokolls

Wie in Abbildung 6.2 illustriert, startet diese Komponente das Szenario und den oben erläuterten Bereitstellungsorchestrator. Dabei hat der Entwickler die Möglichkeit, für jeden Testlauf eine Auswahl zu treffen: die Art der Eingabedaten, das verwendete Szenario und die Zielumgebung für die einzelnen Komponenten des SUT (A13.1). Diese Auswahl wird als Konfigurationsdatei im Versionskontrollsystem gespeichert und vom Automatisierungsdienst gelesen. Die Automatisierung wird durch das Versionskontrollsystem gestartet, beispielsweise nach einem Commit auf einem bestimmten Branch. Eine intern konfigurierte CI/CD-Pipeline benachrichtigt den Automatisierungsdienst und der Testlauf wird durchgeführt. Mit diesem Automatisierungskonzept ist es auch möglich, Remote Tests durchzuführen, selbst wenn sich die Testumgebung physisch an einem anderen Ort befindet. Dies wird durch die Rolle des Versionskontrollsystems ermöglicht, da alle benötigten Konfigurationsdateien und Szenarien

1. https://kubernetes.io/ (Abgerufen am 28.01.2025)
2. https://docs.docker.com/engine/swarm/ (Abgerufen am 28.01.2025)

in einem Repository online gespeichert werden. Der Entwickler kann somit über das Repository des SUT den Testlauf starten, die Ergebnisse einsehen und die Leistung des SUT analysieren, ohne direkt vor Ort sein zu müssen.

6.7. Zusammenfassung der Systemarchitektur

Die in Kapitel 6 dargestellte Systemarchitektur der Testumgebung stellt eine Lösung für die in Abschnitt 1.2 genannten Ziele und die in Kapitel 4 definierten Anforderungen dar. Dabei wurde eine technische und logische Sicht auf die verwendeten Komponenten dargestellt. Der Kernaspekt der Testumgebung ist die Bereitstellung und Entgegennahme von Ein-/Ausgabedaten zum/vom SUT. Dies geschieht über eine nachrichtenorientierte Middleware und die Verwendung des standardisierten Datenformats OSI. Diese Aufgabe übernimmt ein Adaptermodul, das zusätzlich eine Netzwerktrennung zwischen dem SUT und der Testumgebung ermöglicht. Die Komponenten der Testumgebung werden in physische und virtuelle Komponenten unterteilt. So können Sensoren und Aktoren als virtuelle Komponenten aus einer Simulation oder als reale Komponenten verwendet und ausgetauscht werden. Ebenso bietet die Testumgebung Konzepte des Test- & V+V-Managements, wie eine Monitoring-Schnittstelle, eine Automatisierung und die Möglichkeit, verschiedene Szenarien zu verwenden. Zum Abschluss des Kapitels werden im Folgenden die Funktionen der beschriebenen Testumgebung zusammengefasst:

- Bereitstellung von Sensordaten, Eigendaten des Fahrzeugs sowie Ground-Truth-Daten aus der Testumgebung an das SUT.
- Empfang von Steuerbefehlen vom SUT und Rückkopplung in die Simulation.
- Durchführung von Closed-Loop-Tests.
- Auswahl zwischen verschiedenen Qualitätsstufen der Sensordaten.
- Bereitstellung einzelner Komponenten des SUT in einer HiL oder SiL Umgebung.
- Vollständige Automatisierung der Testumgebung.
- Verwendung von Standards wie OSI und OpenSCENARIO, um Erweiterbarkeit und Interoperabilität zu gewährleisten.

7. Implementierung

Die in dieser Arbeit vorgestellte Testumgebung wurde im Rahmen eines Proof of Concept implementiert, um zu überprüfen, ob das entwickelte Konzept in der Praxis funktioniert. Der Fokus lag dabei auf einer prototypischen Implementierung, die alle Aspekte der Architektur abbildet, sodass ein Testen des SUT durch die Testumgebung möglich ist. Die Implementierung umfasst verschiedene Aspekte. Da die gesamte Testumgebung als Docker-Projekt realisiert und gestartet wird, folgt in Abschnitt 7.1 eine Erläuterung der verwendeten Container und deren Abhängigkeiten. Anschließend folgt die Implementierung der nachrichtenorientierten Middleware und der benutzerdefinierten Nachrichtenklasse (Abschnitt 7.2 und Abschnitt 7.3). Um die Komponenten der virtuellen und physischen Testumgebung (Abschnitt 7.4) mit dem SUT zu verbinden, folgt die Implementierung des Adaptermoduls in Abschnitt 7.5. Danach folgt die Beschreibung des Basisimages, welches von dem Bereitstellungsorchestrator genutzt wird, um die Komponenten des SUT in der SiL- und HiL-Umgebung bereitzustellen (Abschnitt 7.7, Abschnitt 7.8). Abschließend folgt die Beschreibung des manuellen Starts der Testumgebung (Abschnitt 7.9) und der automatisierten Testläufe durch den Automatisierungsdienst (Abschnitt 7.10).

7.1. Dockerprojekt

Aus Gründen der Reproduzierbarkeit, der einfachen Bereitstellung und der Betriebssystemunabhängigkeit werden alle Komponenten der Testumgebung als Docker-Container virtualisiert. Für die Bereitstellung der benötigten Docker-Container wird das Werkzeug `docker-compose` verwendet. Damit können verschiedene Container als Services definiert und deren Abhängigkeiten und Parameter festgelegt werden. Die Gesamtheit dieser Services wird in Abbildung 7.1 visualisiert. Gleichzeitig dient die Darstellung der verschiedenen Container als übergeordnete Struktur, um die verschiedenen Komponenten der Testumgebung und deren Abhängigkeiten zu verstehen. Jeder Container erfüllt dabei eine bestimmte Funktion in der Systemarchitektur aus Kapitel 6, die in diesem Abschnitt erläutert wird. Es ist zu erkennen, dass alle Container über Abhängigkeiten von `healthchecks` miteinander verbunden sind. Dies ermöglicht z. B., dass der Container `scenario` erst dann gestartet wird, wenn der Container `carla` den Zustand `healthy` erreicht hat (erfolgreich gestartet wurde). Ebenfalls erkennbar

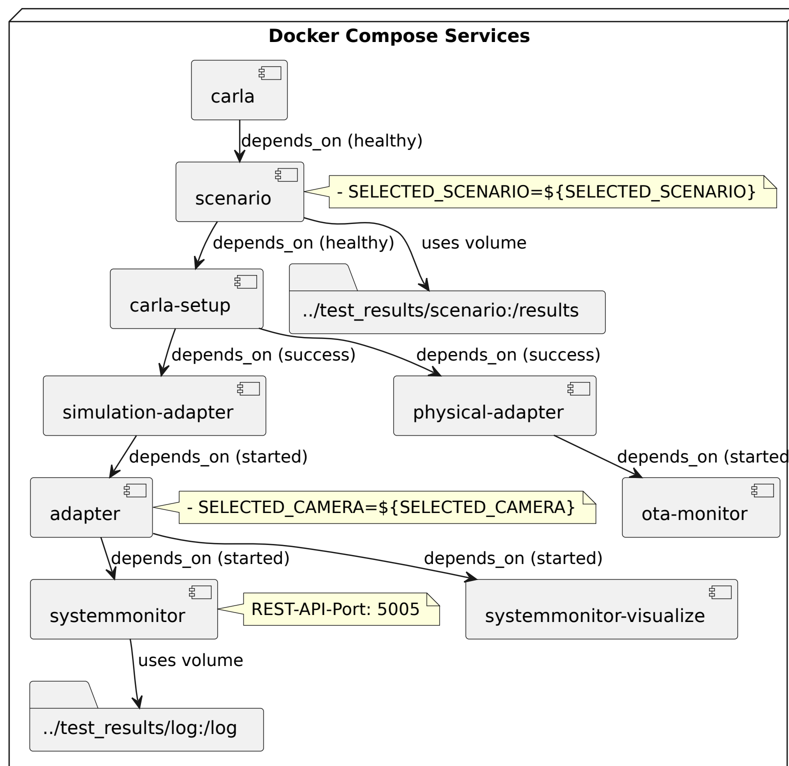


Abbildung 7.1.: Visualisierung des Docker-Stacks der Testumgebung mit Umgebungsvariablen, Ports und Volumen

sind die Umgebungsvariablen `SELECTED_SCENARIO` und `SELECTED_CAMERA`, mit denen das Szenario und die Eingabedaten (A1.1, A9) ausgewählt werden können. Es folgt eine kurze Beschreibung der einzelnen Container.

carla Der Container `carla` startet und verwaltet eine Instanz des Simulators Car Learning to Act (CARLA). Dabei wird standardmäßig eine vordefinierte Karte geladen. CARLA ist ein Open-Source-Fahrsimulator mit Fokus auf autonomes Fahren, der seit 2017 entwickelt wird (Dosovitskiy et al. 2017). Mit CARLA wird eine realistische Test- und Entwicklungsplattform im Bereich autonomes Fahren auf Basis der Unreal Engine 4 angeboten. Über eine Python-API kann direkt auf die Simulation zugegriffen werden, um sie zu verwalten, Daten abzufragen oder Änderungen vorzunehmen. In dieser Arbeit wird die Version 0.9.15 verwendet, die als Docker-Image [1] zur Verfügung gestellt wird. Für die korrekte Funktion müssen verschiedene Parameter und Umgebungsvariablen in der `docker-compose` Datei gesetzt werden, wie z. B. die Verwendung von `runtime: nvidia`, eine Weiterleitung des X11-Videotreibers und die Übergabe der Variable `DISPLAY`.

1. https://hub.docker.com/r/carlasim/carla/tags (Abgerufen am 24.01.2025)

scenario Dieser Container startet und verwaltet das Szenario. Um das ausgewählte OpenSCENARIO in CARLA auszuführen, wird das Open-Source-Projekt ScenarioRunner verwendet[1]. Dies ermöglicht es OpenSCENARIO-XML-Dateien in CARLA auszuführen. Über das Szenario (OpenSCENARIO-Datei) kann eine CARLA-Karte ausgewählt, die Position des Ego-Fahrzeugs und weiteren Fahrzeugen festgelegt werden, usw. (siehe Unterabschnitt 2.4.2). Der `scenario`-Container wechselt die Karte und erzeugt die angegebenen Fahrzeuge (engl. „spawn") in der Simulationsumgebung. Nach Beendigung des Szenarios erzeugt der ScenarioRunner ein Testprotokoll, das auf einem Volume gespeichert wird. Die benutzerdefinierten Szenarien werden in den Container geladen, sodass sie über die Umgebungsvariable ausgewählt werden können. Diese Variable enthält den Dateinamen des Szenarios. Der folgende Befehl wird im Container `scenario` ausgeführt.

```
python3 scenario_runner.py --openscenario custom/$SELECTED_SCENARIO \
    --json --outputDir /results &
```

Mit dem Parameter `--openscenario` kann dann der Dateipfad angegeben werden. Mit den Parametern `--outputDir` und `--json` wird der Protokollpfad (zum Volumen) und die Dateikonvertierung angegeben.

carla-setup Das `carla-setup` ermöglicht es, weitere Einstellungsparameter der Simulationsumgebung zu treffen. Während das Szenario bereits die Karte wechselt und die Fahrzeuge erzeugt, können hier weitere Einstellungen getroffen werden, die in einer OpenSCENARIO-Datei nicht beschrieben werden können. Beispielsweise das Anbringen verschiedener Sensoren am Ego-Fahrzeug (Kamera, Radar, LiDAR, Ultraschall). Im Gegensatz zur Szenariodatei sind diese zusätzlichen Parameter und Einstellungen abhängig vom verwendeten Simulator (in diesem Fall CARLA). Sollte dieser gewechselt werden, ist eine Anpassung dieses Containers erforderlich. Das bereits in der Simulation vorhandene Ego-Fahrzeug kann über den eindeutigen Rollennamen „ego_vehicle" und der Python-API abgefragt werden. Anschließend wird eine RGB-Frontkamera mit einer Auflösung von 1920 * 1080 Pixel und einem FOV von 90° am Ego-Fahrzeug angebracht.

simulations-adapter, physical-adapter und adapter sind einzelne Container, die die im Konzept genannten Funktionen (Simulationsadapter und Datenverteiler aus Abschnitt 6.3, Adaptermodul aus Abschnitt 6.5) ausführen. Auf diese Komponenten wird im späteren Verlauf dieses Kapitels konkreter eingegangen.

1. https://github.com/carla-simulator/scenario_runner (Abgerufen am 24.01.2025)

systemmonitor Der Container `systemmonitor` startet und verwaltet die Monitoring-Schnittstelle (A8). Über den Port 5005 kann auf die zugehörige REST-API zugegriffen werden. Bei Anfragen auf die definierten REST-Endpunkte werden diese Daten über die nachrichtenorientierte Middleware abgefragt.

systemmonitor-visualize Zusätzlich zu der Monitoring-Schnittstelle wird von A10 eine Visualisierung der Ein- und Ausgabedaten gefordert, welche vom Container `systemmonitor-visualize` umgesetzt wird. Dieser greift direkt auf die nachrichtenorientierte Middleware zu. Abbildungsbeispiel für diese Visualisierung folgen in der Evaluation in Kapitel 8.

7.2. Nachrichtenorientierte Middleware mit ROS

Wie im Konzept beschrieben, erfolgt die gesamte Kommunikation in der Testumgebung über eine nachrichtenorientierte Middleware. Dabei hat sich das Publish-Subscribe-Kommunikationsmuster als geeigneter Ansatz erwiesen, um eine zeitliche und räumliche Entkopplung zu gewährleisten. Zur Realisierung dieser Middleware wird auf das Open-Source-Framework ROS in der Version 2 zurückgegriffen, das bereits in Unterabschnitt 2.4.2 erläutert wurde. ROS unterstützt das Kommunikationsmuster, in dem vordefinierte oder benutzerdefinierte Nachrichtenformate verwendet werden können. Als Programmiersprache wird C++ und Python unterstützt. ROS 2 (im Folgenden ROS genannt) unterscheidet sich von ROS 1 insbesondere durch das verwendete Nachrichtenprotokoll DDS. Damit ist es möglich, Synchronisierungen, QoS-Funktionen und Echtzeitfunktionen zu implementieren. Darüber hinaus ermöglicht ROS die Erweiterbarkeit und Skalierbarkeit der Testumgebung. Neue Publisher oder Subscriber können hinzugefügt werden, ohne die gesamte Verarbeitungskette anpassen zu müssen.

In Kapitel 6 wurde bereits das einheitliche Datenformat OSI erläutert. Dieses beschreibt die inhaltliche Struktur der Nachrichten innerhalb der Testumgebung. ROS wird als Middleware verwendet, um diese Nachrichten versenden zu können. Dazu muss die OSI-Nachricht in eine ROS-Nachricht verpackt werden. Als ROS-Nachrichtentyp wird dabei der Typ `std_msgs/String` verwendet. Dazu muss die OSI-Nachricht serialisiert und anschließend in einen String konvertiert werden. Quellcodeauszug 7.1 zeigt einen Ausschnitt aus dem Quellcode, der diese Konvertierung und Paketierung darstellt. Zunächst wird die leere OSI-Nachricht (in diesem Beispiel `osi3::CameraSensorView`) erstellt, und anschließend mit Daten gefüllt. Über die Methode `SerializeToString()` wird die OSI-Nachricht serialisiert, über `hex()` in eine hexadezimale Zeichenkette umgewandelt und abschließend über den ROS-Publisher veröffentlicht.

```python
osi_msg = osi_sensorview_pb2.CameraSensorView() # create OSI msg
osi_msg.image_data = img_bytes
osi_msg.view_configuration.number_of_pixels_horizontal = 640
osi_msg.view_configuration.number_of_pixels_vertical = 360
serialized_msg = osi_msg.SerializeToString()
msg = std_msgs.msg.String() # create empty ROS msg
msg.data = serialized_msg.hex() # convert into hexadecimal string
self.camera_image_publisher.publish(msg) # publish ROS msg
```

Quellcodeauszug 7.1: Verpacken einer OSI-Nachricht in eine ROS-Nachricht

7.3. Benutzerdefinierte OSI-Nachrichtenklasse

OSI bietet keine vordefinierten Nachrichtenformate für Steuerbefehle. Wie bereits in Kapitel 6 beschrieben, besteht jedoch die Möglichkeit, benutzerdefinierte Nachrichtenklassen zu erstellen. OSI basiert auf dem von Google entwickelten Framework `protobuf`, das eine plattform- und programmiersprachenunabhängige Beschreibung strukturierter Daten ermöglicht. Diese Datenstrukturen werden als `.proto` Datei gespeichert. Die benutzerdefinierte Nachrichtenklasse für die benötigten Steuerbefehle, genannt `osi3::ControlCommand`, ist nach Quellcodeauszug 7.2 strukturiert. Sie enthält neben Signalen wie Beschleunigung,

```protobuf
import "osi_version.proto";
import "osi_common.proto";
message ControlCommand
{
    optional double steering_angle = 1;
    optional double throttle = 2;
    optional double braking = 3;
    optional bool low_beam_on = 4;
    optional bool high_beam_on = 5;
    optional bool right_blinker_on = 6;
    optional bool left_blinker_on = 7;
}
```

Quellcodeauszug 7.2: Struktur der benutzerdefinierten OSI-Nachrichtenklasse ControlCommand

Bremsen und Lenkbefehle auch einer Licht- und Blinkersteuerung. Um diese Nachrichtenklasse in der Testumgebung verwenden zu können, muss sie vor dem Build-Prozess von OSI im Docker-Container hinzugefügt werden. Diese Nachrichtenklasse stellt eine vereinfachte Ver-

sion, der tatsächlich möglichen Steuerbefehle eines ADS dar. Sollte das zu testende System, weitere Befehle benötigen, die in die Simulation rückgekoppelt werden, kann die Nachricht durch weitere Signale erweitert werden.

7.4. Komponenten der virtuellen Testumgebung

Die Komponenten der virtuellen Testumgebung bilden die Grundlage der gesamten Testumgebung. Wie bereits in Abschnitt 6.3 erläutert, besteht diese aus einem Simulationsadapter, Sensormodellen, Sensorik und Aktorik. Der Simulationsadapter dient als Kommunikationsschnittstelle zwischen der Sensorik/Aktorik und der nachrichtenorientierten Middleware ROS. Der Simulationsadapter unterteilt sich in einen Publisher und einen Subscriber. Um den Nachrichtenfluss zwischen den Komponenten und dem SUT zu verdeutlichen, ist in Anhang C ein Sequenzdiagramm dieser Verarbeitungskette dargestellt. Abbildung 7.2 zeigt ein Unified Modeling Language (UML)-Klassendiagramm des Publishers. Der `SimulationsAdapterPublisher` ist als ROS-Node definiert und erbt von der Klasse `rclpy.Node`. Dabei verläuft die Verarbeitungskette wie folgt: Die Klasse `Sensory` stellt

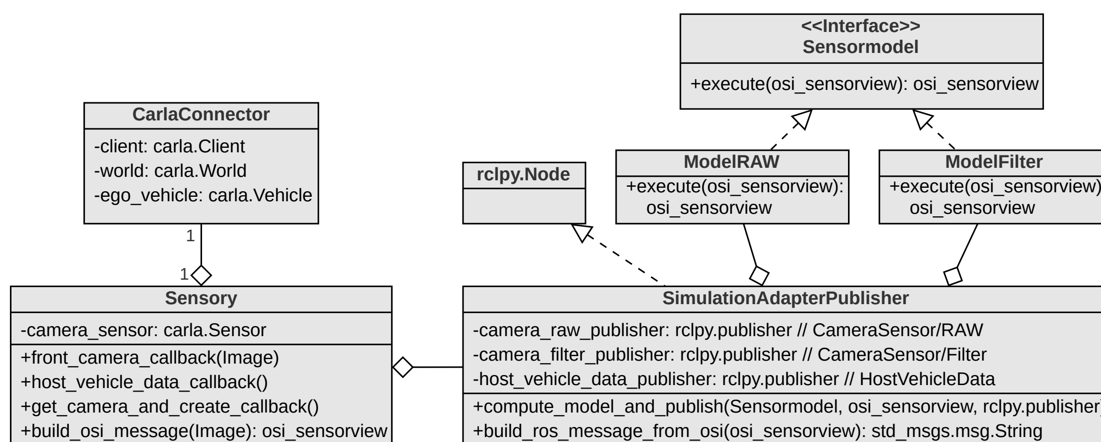


Abbildung 7.2.: UML Klassendiagramm des Simulationsadapters (Publisher)

über den `CarlaConnector` eine Verbindung zum Simulator her. Über die Methode `get_camera_and_create_callback` wird die `listen` Methode des Sensors aufgerufen. Dabei wird die `front_camera_callback` als Lambda-Funktion übergeben, sodass beim Eintreffen neuer Sensordaten diese Callback-Methode aufgerufen wird[1]. Um aus den Kameradaten von CARLA eine Nachricht im OSI-Format zu erzeugen, wird die Methode `build_osi_message` verwendet. Sie verwendet die Klasse `SimulationAdapterPublisher`, um die Sensordaten über die Sensormodelle zu verarbeiten und über die Middleware zu versenden. Das Sensor-

1. https://carla.readthedocs.io/en/latest/core_sensors (Abgerufen am 27.01.2025)

modell ist als Interface definiert, in dem Implementierungen die Methode `execute` bereitstellen müssen. In dieser wird ein `osi_sensorview` übergeben und das bearbeitete Objekt zurückgegeben. Durch die Verwendung der Nachrichtenklasse `osi3::SensorView` kann das Interface für Sensormodelle verschiedener Sensoren (Kamera, LiDAR, Radar und Ultraschall) verwendet werden, da ein SensorView-Objekt Daten aller Sensoren enthalten kann. Durch die Verwendung des Interfaces können weitere Sensormodelle hinzugefügt werden, die für die Entwicklung eines ADS benötigt werden. In der aktuellen Implementierung wird im `ModelRAW` das Objekt im Original zurückgegeben. Im `ModelFilter` wird ein Teil des in Modrakowski et al. (2024) vorgestellte Unschärfe-Effekt-Modells verwendet, um die Kameradaten zu bearbeiten. Alle Daten der Sensormodelle werden anschließend über ROS-Publisher an die Middleware veröffentlicht.

Die umgekehrte Richtung, d. h. der Empfang der Steuerbefehle über die Middleware und die Anwendung auf die Aktorik, erfolgt durch eine ähnliche Modellierung. In der Klasse `SimulationsAdapterSubscriber` werden die Daten über einen ROS-Subscriber empfangen, die darin enthaltene benutzerdefinierte OSI-Nachricht (siehe Abschnitt 7.3) extrahiert und die Steuerbefehle auf das Ego-Fahrzeug in der Simulation angewendet. Quellcodeauszug 7.3 zeigt den entsprechenden Quellcode. Die Anwendung der Steuerbefehle auf das Ego-Fahrzeug erfolgt über die CARLA-Python-API und die Methoden `get_control` und `apply_control`.

```
1 def control_callback(self, control_data: osi_control_command.ControlCommand()):
2     carla_control: carla.VehicleControl = self.vehicle.get_control()
3     carla_control.steer = control_data.steering_angle
4     carla_control.throttle = control_data.throttle
5     carla_control.brake = control_data.braking
6     self.vehicle.apply_control(carla_control)
```

Quellcodeauszug 7.3: Anwenden der Steuerbefehle auf das Ego-Fahrzeug in der Simulation über die CARLA-Python-API

Die Komponenten der physischen Testumgebung sind ebenfalls als Python-Applikation implementiert. Als physische Komponente wird ein OTA-Sensor verwendet, der durch eine Visualisierung der Ego-Fahrzeug-Kameradaten stimuliert wird. Der OTA-Sensor, in diesem Fall eine USB-Webcam, ist mit einer separaten Rechnerplattform verbunden. Über das Framework OpenCV[1] können die USB-Kameradaten empfangen und über die nachrichtenorientierte Middleware ROS veröffentlicht werden. Ein weiteres Python-Programm (`ota-monitor.py`) empfängt die Kamerarohdaten von der nachrichtenorientierten Middleware und visualisiert

1. https://opencv.org/ (Abgerufen am 04.02.2025)

diese auf einem externen PC-Monitor im Vollbildmodus. Eine detaillierte Beschreibung der Hardwarekonfiguration ist in der Evaluation in Kapitel 8 enthalten. Das im Folgenden beschriebene Adaptermodul empfängt die Daten der virtuellen und physischen Testumgebung.

7.5. Adaptermodul

In diesem Abschnitt folgt die Implementierung des Adaptermoduls, das in Abschnitt 6.5 konzeptionell vorgestellt wurde. Wie dort beschrieben, hat das Adaptermodul die Aufgabe, das zu testende System (SUT) mit der Testumgebung zu verbinden. Dabei müssen die ausgewählten Sensordaten übertragen und die Steuerbefehle empfangen werden. Auf der Seite der Testumgebung werden die Daten über die nachrichtenorientierte Middleware gesendet und empfangen. Auf der Seite des SUT wird ein spezielles Kommunikationsprotokoll verwendet, das eine lose Kopplung sowie die Unabhängigkeit von der verwendeten Programmiersprache gewährleistet. Das Protokoll basiert auf der Netzwerktechnologie (Automotive) Ethernet (siehe Abschnitt 6.4). Wie bereits in Unterabschnitt 2.4.2 beschrieben, ist Automotive Ethernet eine Weiterentwicklung des IEEE 802.3 Standards. In der Realisierung und dem in der Evaluierung verwendeten Hardware-Setup wird Ethernet als Netzwerktechnologie verwendet. Die zusätzlich möglichen Mechanismen, wie beispielsweise ein TSN, sind nicht der Fokus dieser Arbeit und werden dementsprechend nicht verwendet.

7.5.1. Auswahl des Kommunikationsprotokolls

Für die Auswahl eines spezifischen Kommunikationsprotokolls müssen verschiedene Punkte beachtet werden. Das gewählte Protokoll bestimmt, wie Nachrichten zwischen dem Adaptermodul und dem SUT übertragen werden.

- Das Protokoll muss weit verbreitet sein.
- Das Protokoll muss Programmiersprachenunabhängig implementiert werden können.
- Das Protokoll muss die Übertragung von serialisierten OSI-Nachrichten ermöglichen.
- Das Protokoll muss auf dem TCP-Protokoll basieren.
- Das Protokoll muss eine geringe Latenz und eine persistente Verbindung aufweisen.

Für diese Anforderungen kommen verschiedene Kommunikationsprotokolle infrage. Ein weit verbreitetes Protokoll ist eine Representational State Transfer (REST)-Verbindung. Diese ist sehr einfach zu implementieren, bietet jedoch keine persistente Verbindung. Das SUT müsste in regelmäßigen Intervallen polling-basierte Anfragen senden, um neue Daten zu erhalten, was weniger effizient ist als eine persistente Verbindung. Das Message Queuing Telemetry

Transport (MQTT)-Protokoll basiert auf einem Publish-Subscribe-Modell und ermöglicht eine Kommunikation mit geringer Latenz. Es erfordert jedoch einen separaten MQTT-Broker, was zusätzliche Infrastruktur benötigt. Zudem kann es komplexer sein, wenn Sicherheitsmechanismen und Verteilungsaspekte berücksichtigt werden müssen. Ein weiterer Kandidat für das Kommunikationsprotokoll sind Server-Sent Events (SSE). Diese basieren auf HTTP, bieten eine breite Unterstützung von Technologien und ermöglichen eine einfache Implementierung. Sie bieten jedoch eine limitierte Skalierbarkeit, da jede offene Verbindung eine dedizierte HTTP-Session belegt. Zudem ist keine direkte Wiederverbindung nach Verbindungsverlust möglich (Hassan 2024).

Ein weiteres Protokoll, das die oben genannten Punkte unterstützt, ist WebSocket. WebSockets bieten eine bidirektionale, persistente Verbindung zwischen einem Client und einem Server. Im Vergleich zu REST oder SSE ermöglicht die persistente Verbindung eine effizientere Kommunikation, da nicht regelmäßig eine neue Verbindung aufgebaut werden muss. Es basiert ebenfalls auf TCP und kann in allen gängigen Programmiersprachen und Frameworks implementiert werden. Durch die Erfüllung der oben genannten Punkte wird in dieser Implementierung das Kommunikationsprotokoll Websocket verwendet. Es ist jedoch zu beachten, dass aufgrund der losen Kopplung zwischen Testumgebung und zu testendem System mit wenig Aufwand auch ein anderes Protokoll verwendet werden kann. Eine Aufgabe des Adaptermoduls ist es, die Daten der nachrichtenorientierten Middleware ROS als WebSockets dem SUT zur Verfügung zu stellen. Dazu ist ein Mapping der ROS-Topics, der WebSocket-Uniform Resource Locator (URL) und der zugehörigen OSI-Nachricht notwendig (siehe Tabelle 7.1). Beispielsweise werden die Kameradaten, die über das bereits beschriebene Sensormodell `Filter` und über das ROS-Topic `/CameraSensor/Filter` der Middleware verarbeitet werden, über die WebSocket-URL `ws://localhost:8765` gesendet. In der ge-

Tabelle 7.1.: Übersicht der WebSocket-Verbindungen, ROS-Topics und verwendeten OSI-Nachrichten

Inhalt	**WebSocket URL**	**ROS-Topic**	**OSI-Nachricht**
CameraRAW	ws://localhost:8765	/CameraSensor/RAW	osi_sensorview
CameraFilter	ws://localhost:8765	/CameraSensor/Filter	osi_sensorview
CameraOTA	ws://localhost:8765	/CameraSensor/OTA	osi_sensorview
ControlCommand	ws://localhost:8766	/ControlCommand	osi_control_command
HostVehicleData	ws://localhost:8767	/HostVehicleData	osi_hostvehicledata
GroundTruth	ws://localhost:8768	/GroundTruth	osi_groundtruth

zeigten Implementierung werden jeweils drei Modelle eines Kamerasensors verwendet. Es ist

jedoch möglich, weitere Sensoren wie LiDAR oder Radar hinzuzufügen und einen neuen Port zu definieren.

7.5.2. Auswahl der Eingabedaten.

Wie in Tabelle 7.1 zu erkennen, werden die Daten der verschiedenen Sensormodelle und Sensoren der Kamera (RAW, Filter und OTA) über einzelne ROS-Topics vom Adaptermodul empfangen. Dabei übernimmt das Adaptermodul eine weitere Aufgabe, die eines Multiplexers. Über die im Docker-Container gesetzte Umgebungsvariable `SELECTED_CAMERA` werden die Daten der gewünschten Kamera/Modell auf der WebSocket-URL mit dem Port 8765 bereitgestellt. Durch diese Entscheidung wird keine Änderung durch verschiedene Sensormodell am Port durchgeführt. Somit muss im SUT nur ein WebSocket-Client für diesen Port existieren und das Sensormodell/Kamera kann zur Laufzeit geändert werden, ohne das zu testende System anpassen zu müssen.

7.5.3. Modellierung

Analog zum Simulationsadapter ist auch das Adaptermodul in einen Subscriber und einen Publisher (zur Middleware) aufgeteilt. Der Subscriber empfängt Daten von der nachrichtenorientierten Middleware (ROS) und sendet diese über die definierten WebSockets. Der Publisher nimmt die Steuerbefehle des SUT über ein WebSocket entgegen und publiziert die Daten über ROS. In diesem Abschnitt folgt die Modellierung des Subscribers, der aufgrund der verschiedenen Verbindungen komplexer ist als der Publisher. Abbildung 7.3 zeigt ein UML-Klassendiagramm des Subscribers des Adaptermoduls. Zu erkennen ist das Interface `WebSocket`, das die benötigten Methoden definiert. Dieses Interface wird durch `CameraWebSocket`, `VehicleDataWebSocket` und `GroundtruthWebSocket` realisiert. Zu erkennen ist auch die Klasse `AdapterSubscriber`, die über eine Event-Klasse mit den WebSocket-Klassen verbunden ist. Diese Modellierung ist notwendig, da jede WebSocket-Kommunikation in einem eigenen Thread abläuft. Dadurch ist es möglich, Daten parallel als WebSocket bereitzustellen, falls diese gleichzeitig als ROS-Nachricht eintreffen. Sobald neue Daten über den ROS-Subscriber eintreffen, wird das entsprechende Event über die Methode `set` gesetzt. Mit der Methode `handle_client` können sich WebSocket-Clients anmelden und in die Liste `clients` eingetragen werden. Die Methode `broadcast_data` sendet für jeden registrierten Client die benötigten Daten, sobald das entsprechende Event gesetzt wurde. Das Warten auf das Event wird durch die Methode `wait` realisiert. Damit ist eine Entkopplung der ROS-Subscriber und der zugehörigen Callback-Methoden vom Versenden der Nachrichten über die WebSockets möglich.

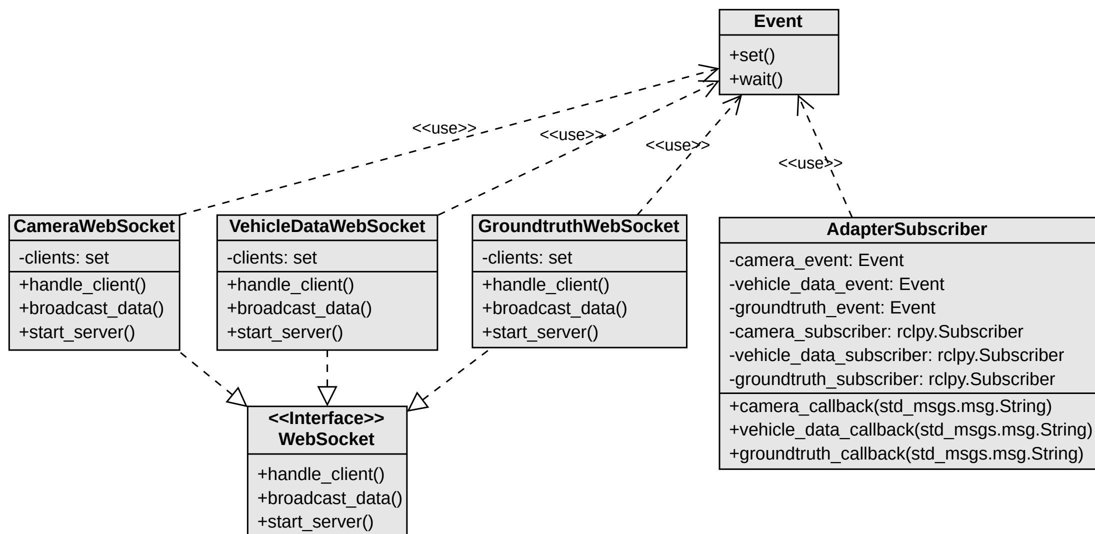


Abbildung 7.3.: UML Klassendiagramm des Adaptermoduls (Subscriber)

7.6. Monitoring-Schnittstelle

Für die Beobachtung der Testumgebung und des SUT wird nach A8 eine Schnittstelle gefordert. Wie bereits in Unterabschnitt 6.6.1 erläutert, bezieht diese Schnittstelle die Daten aus der nachrichtenorientierten Middleware. Sie wird als REST-Application Programming Interface (API) implementiert und ist über die URL `localhost:5005` erreichbar. Es werden verschiedene Endpunkte der API implementiert, bei sowohl auf Simulationsdaten, als auch auf Monitordaten des SUT zugegriffen wird. Dabei können Events wie Kollisionen oder das Überfahren von Fahrspuren, aber auch Ressourcenkennzahlen abgefragt werden. Ebenso ist es möglich über die Endpunkte `GET /monitor/sut/input` und `GET /monitor/sut/output` die Ein- und Ausgabedaten des SUT zu beobachten. Über den Image-Endpunkt kann das Kamerabild, welches als Eingabe zum SUT verwendet wird, direkt als `mimetype=image/jpeg` visualisiert werden. Tabelle 7.2 zeigt alle implementierten Endpunkte der REST-API.

Tabelle 7.2.: Endpunkte der REST-API

HTTP	Endpunkt	Beschreibung
GET	/actors/ego_vehicle	Gibt Informationen des Ego-Fahrzeugs (ID, Typ, Name, Position, Rotation) zurück
GET	/monitor/lane_invasion_events	Gibt eine Liste von allen bisherigen Events zurück, in denen das Ego-Fahzeug über eine Spur gefahren ist.

HTTP	Endpunkt	Beschreibung
GET	/monitor/collision_events	Gibt eine Liste von allen bisherigen Events zurück, in denen eine Kollision mit dem Ego-Fahrzeug stattgefunden hat.
GET	/monitor/sut/sil/ressources	Gibt Informationen über Ressourcenauslastung (CPU, MEM, etc.) der im SiL laufenden Container des SUT zurück.
GET	/monitor/sut/hil/ressources	Gibt Informationen über Ressourcenauslastung (CPU, MEM, etc.) der im HiL laufenden Container des SUT zurück.
GET	/monitor/sut/input	Gibt Informationen über die Eingabedaten zurück.
GET	/monitor/sut/input/image	Gibt das Kamerabild der Eingabedaten als JPG zurück.
GET	/monitor/sut/output	Gibt Informationen über die Ausgabedaten (Steuerbefehle) zurück.
GET	/monitor/sut/runtime	Gibt Laufzeit des SUT zurück, gemessen vom Empfang der Eingabedaten bis zum Versenden der Steuerbefehle.

7.7. Erstellung eines Basisimage für das SUT

Um die Entwicklung neuer ADS zu erleichtern und eine einfache Integration in die Testumgebung zu ermöglichen, wird ein standardisiertes Basisimage als Docker-Container zur Verfügung gestellt. Dieses Image enthält alle notwendigen Dateien und Abhängigkeiten. Entwickler können dieses Image dann als Basisimage für ihre Docker-Container verwenden. Durch die Verwendung des standardisierten Datenformats OSI ist es notwendig, eine funktionierende OSI-Implementierung bereitzustellen. Diese kann im Dockerfile des Basisimages definiert werden. OSI erlaubt eine Installation als C++ und Python Programm[1]. Das Basisimage kann dann unter dem Namen `sut-base-osi:latest` gebaut werden. Um das Image verwenden zu können, muss es im Dockerfile des SUT als Basisimage über `FROM sut-base-osi:latest` angegeben werden. Auf diese Weise muss der Entwickler des ADS nur das Basisimage einbinden und kann darauf aufbauend das spezifische Dockerfile des SUT definieren.

1. https://github.com/OpenSimulationInterface/open-simulation-interface (Abgerufen am 18.02.2025)

7.8. Bereitstellungsorchestrator

Wie bereits in Unterabschnitt 6.6.3 erläutert und von A12 gefordert, ist eine Bereitstellung der verschiedenen Docker-Container des SUT auf unterschiedliche Hardware (HiL und SiL) notwendig. Diese Aufgabe übernimmt der sogenannte Bereitstellungsorchestrator. Dieser kann durch verschiedene Werkzeuge implementiert werden. Die am weitesten verbreiteten Softwarewerkzeuge sind Kubernetes und Docker Swarm. Während Kubernetes vor allem im Bereich der Skalierung mehr Features bietet, ist Docker Swarm durch seine einfache Handhabung in kleineren Projekten die geeignete Wahl für diese Arbeit. Bei der Verwendung von Docker Swarm wird zwischen verschiedenen Nodes (Geräten) unterschieden. Es gibt immer einen Manager und eine beliebige Anzahl von Arbeiter (engl. „worker"). Nach der Initialisierung von Docker Swarm kann der ein gesamter Stack (Definition von mehreren Docker-Containern) bereitgestellt werden. Der gesamte Durchlauf dieses Prozesses mit einem konkreten Beispiel folgt in Abschnitt 8.3.

Um die Docker-Images nicht jeweils in der HiL- und in der SiL-Umgebung bauen zu müssen, kann ein lokales Docker-Registry verwendet werden. Diese verwaltet die Images und ermöglicht den Zugriff von verschiedene Geräten. Dadurch wird sichergestellt, dass stets konsistente und getestete Images verwendet werden, was den Aufwand für die Bereitstellung reduziert. Diese Registry kann als Docker-Service definiert und gestartet werden. Um das Image vom SUT von dieser Registry verwalten zu lassen, muss ein Tag definiert und das Image hochgeladen werden (siehe Quellcodeauszug 7.4).

```
docker service create --name registry --publish 5000:5000 registry:2
docker tag sut:latest 192.168.2.2:5000/sut:latest
docker push 192.168.2.2:5000/sut:latest
```

Quellcodeauszug 7.4: Definition der Registry als Docker-Service und Hochladen des Images

Dabei ist es notwendig die IP-Adresse des Managers anzugeben, damit die anderen Geräte (z. B. das Gerät der HiL-Umgebung) die Registry finden. Ohne eine zentrale Registry müsste jedes Gerät lokal eigene Builds erstellen oder manuell mit Images versorgt werden, was fehleranfällig und ineffizient wäre. Anschließend kann das Image in der docker-compose Datei des SUT angegeben werden (`image: 192.168.2.2:500/sut:latest`)[1]. Es ist ebenfalls möglich, verschiedene Images für verschiedene Teile des SUT zu definieren und anschließend via Docker Swarm bereitzustellen.

1. https://codeblog.dotsandbrackets.com/private-registry-swarm/ (Abgerufen am 20.12.2024)

7.9. Starten und Stoppen der Testumgebung

Um die gesamte Testumgebung mit den bereits gezeigten Komponenten (siehe Abschnitt 7.1) manuell starten und stoppen zu können, werden Bash-Skripte definiert. Über ein Startskript (siehe Quellcodeauszug 7.5) können Argumente für die Quelle der Eingabedaten und die Auswahl des Szenarios übergeben werden (A9, A1.1). In dem gezeigten Beispiel wird die Testumgebung mit dem Sensormodell `RAW` und dem Szenario `FolloLane.xosc` gestartet.

```
$ ./manual_test_run.sh --help
Usage: ./manual_test_run.sh [OPTIONS]
Options:
  --SELECT_CAMERA=<value>      Set the selected camera (RAW, Filter, OTA).
  --SELECT_SCENARIO=<value>    Set the selected scenario. (e.g., FollowLane.xosc)
  --help                       Display this help message and exit.
Available scenarios:
  - FollowLane.xosc
  - FollowLeadingVehicle.xosc
  - FollowStraightLane.xosc

$ ./manual_test_run.sh --SELECT_CAMERA=RAW --SELECT_SCENARIO=FollowLane.xosc
```

Quellcodeauszug 7.5: Verwendung des Startskripts zum Hochfahren der Testumgebung und Starten eines Testlaufs

Dieses Skript exportiert die definierten Umgebungsvariablen und startet das gesamte Dockerprojekt über die `docker-compose` Datei. Um am Ende des Szenarios alle Docker-Container herunterzufahren, wird in einer while-Schleife der Status des Szenario-Containers überprüft. Sollte dieser nach Beendigung des Szenarios stoppen, wird das gesamte Projekt mit dem Befehl `docker compose down` heruntergefahren und das Skript beendet. Für eine vereinfachte Entwicklung ist es über ein dediziertes Stopp-Skript möglich, die Testumgebung zu jeder Zeit herunterzufahren.

7.10. Automatisierungsdienst

Das manuelle Testen des SUT ist zeitaufwendig und fehleranfällig. Durch eine Automatisierung können neue Versionen des SUT unter identischen Bedingungen getestet werden, was eine hohe Reproduzierbarkeit ermöglicht. Diese automatisierten Tests können vom Entwickler auch dann durchgeführt werden, wenn er sich nicht am Standort der Testumgebung befindet (Remote Testing). Für die Implementierung des beschriebenen Automatisierungsdienstes

wird zusätzlich ein Versionskontrollsystem mit einer CI/CD-Pipeline benötigt. Ziel ist es, automatisierte Testläufe durchführen zu können, wenn eine neue Version des SUT hochgeladen wird. In dieser Implementierung wird das Versionskontrollsystem GitLab und der zugehörige GitLab-Runner verwendet. Dieser Runner führt die in der `.gitlab-ci.yml` definierten Jobs aus, muss auf einem Gerät installiert und dem Repository des SUT hinzugefügt werden.

Die CI/CD enthält vier Jobs: `build-sut`, `deploy-sut`, `deploy-test-env` und `validate`. Im ersten Job wird zunächst das Repository des SUT im GitLab-Runner bereitgestellt, die Docker-Images werden gebaut und in die lokale Registry hochgeladen (siehe Abschnitt 7.8). Im zweiten Job wird der gesamte Docker-Stack des SUT über ein eigenes Startskript gestartet. Der dritte Job startet die Testumgebung mit der gewünschten Konfiguration und speichert das Szenarioprotokoll als Artefakt. Im letzten Schritt der Pipeline wird dieses Protokoll analysiert und ausgewertet. Nur wenn alle definierten Tests des Szenarios bestanden wurden, wird die Pipeline als bestanden markiert. Zusätzlich ist es möglich, die Pipeline nach Abschluss erneut zu starten, um z. B. ein bestimmtes Ergebnis zu reproduzieren. In der aktuellen Implementierung der Pipeline wird mit dem Parameter `only: - run_test` festgelegt, dass die automatisierten Tests nur in dem dedizierten Git-Branch `run_test` durchgeführt werden. Dies hat den Vorteil, dass der Entwickler in verschiedenen Branches entwickeln kann, um dann den Stand der getestet werden soll auf diesen Branch zu mergen. Eine Parallelisierung der automatisierten Tests ist bei der Realisierung der Testumgebung nicht möglich. Eine beispielhafte Durchführung eines automatisierten Testlaufs wird in der Evaluation in Abschnitt 8.4 beschrieben. Die vollständige Definition der CI/CD ist in Anhang E zu finden.

8. Evaluation der Testumgebung

Dieses Kapitel beschreibt die Evaluation der implementierten Testumgebung. Für die folgende Evaluation wird zunächst das verwendete SUT in Abschnitt 8.1 mit den notwendigen Anpassungen vorgestellt. Anschließend wird das System anhand der definierten Use-Cases (Abschnitt 8.2, Abschnitt 8.3, Abschnitt 8.4) sowie der zugrundeliegenden Anforderungen verifiziert. Dazu werden ein Closed-Loop-Test eines Lane-Followers (Abschnitt 8.2), eine Portierung des Lane-Followers auf realer Hardware (Abschnitt 8.3) und die Verifikation von automatisierten Testläufen über eine CI/CD (Abschnitt 8.4) durchgeführt. Abschließend erfolgt in Abschnitt 8.5 eine Validierung des Gesamtsystems, die auch über die Anforderungen hinausgehende Aspekte berücksichtigt.

8.1. Beschreibung des verwendeten SUT

Um eine Evaluation anhand der Use-Cases und Anforderungen durchführen zu können, wird ein geeignetes SUT benötigt. Konkret handelt es sich bei dem zu testenden System um einen einfachen Lane-Follower, der mittels kamerabasierter Fahrspurerkennung die aktuelle Fahrspur halten kann. Dieser von Modrakowski et al. (2024) vorgestellte Lane-Follower (dort als LKA bezeichnet) wird in dieser Arbeit als SUT verwendet. Die Verarbeitungskette des Lane-Followers besteht aus den folgenden Schritten:

- Empfang (über ROS) und Vorverarbeitung der Kamerabilder (Preprocessing)
- Erkennung der Fahrspur (Lane-Detection)
- Die Berechnung des Lenkwinkels (Lane-Keeping-System)
- Berechnen und Senden (über ROS) von Steuerbefehlen (Control-Command)

Die ursprüngliche Implementierung des Lane-Followers aus Modrakowski et al. (2024) verwendet verschiedene ROS-Nachrichten für die interne Kommunikation sowie für den Empfang und das Senden von Signalen. Da das Datenformat (OSI) und das Kommunikationsprotokoll (TCP-Websockets) durch die Testumgebung vorgegeben sind, muss das SUT entsprechend angepasst werden. Dies konnte durch die Verwendung des Basis-Images (siehe Abschnitt 7.7)

und durch eine Anpassung des Python-Quellcodes erreicht werden. So benötigt die Komponente `Preprocessing` ein zusätzliches TCP-Websocket für den Empfang der Kamerabilder und eine Methode zur Deserialisierung der Nachrichten im OSI-Format. Gleiches gilt für die Komponente `Control-Command`, die die berechneten Steuerbefehle in das benutzerdefinierte OSI-Format (siehe Abschnitt 7.3) konvertiert und über ein weiteres TCP-Websocket an die Testumgebung sendet. Die gesamte interne Implementierung und Nachrichtenkommunikation nach dem Lesen der Eingabedaten bis zum Senden der Steuerbefehle kann unverändert übernommen werden.

8.2. Closed-Loop-Test eines Lane-Followers – Use-Case 1

An Anlehnung an den Use-Case 1 wird die Testumgebung zunächst mit dem beschriebenen Lane-Follower in einem Closed-Loop-Test getestet. Abbildung 8.1 zeigt den Aufbau der instanziierten Testumgebung zusammen mit dem Lane-Follower als SUT. Hier lassen sich die bereits im Konzept (Kapitel 6) erläuterten Teilsysteme: SUT, Adaptermodul, Test- & V+V-Management, physische und virtuelle Testumgebung. Im Teilsystem SUT werden die bereits erläuterten Komponenten des Lane-Followers dargestellt, welche vollständig in einer SiL-Umgebung ausgeführt werden. Im Adaptermodul wird die nachrichtenorientierte Middleware durch das ROS-Framework instanziiert, das Nachrichten im OSI-Format sendet und empfängt. In der virtuellen Testumgebung sind die beiden unterschiedlichen Sensormodelle erkennbar: Im Modell `RAW` werden die Daten der CARLA-Frontkamera lediglich als Rohdaten an den Simulationsadapter übergeben. Im Modell `Filter` werden diese mit einem Ausschnitt der in Modrakowski et al. (2024) vorgestellten Effekte eines digitalen Zwillings bearbeitet. Der `OTA-Sensor` in der physischen Testumgebung wird durch eine USB-Webcam instanziiert. Dieser Sensor wird über einen `Visualisierungs`-Monitor mit Kamerarohdaten aus dem Sensormodell `RAW` stimuliert. Im Gegensatz zu dem Aufbau der Testumgebung, die im Konzept (Kapitel 6) vorgestellt wurde, fehlt in dieser Ansicht der Automatisierungsdienst und der Bereitstellungsorchestrator. Diese Komponenten werden erst bei der Betrachtung von Use-Case 2A und Use-Case 2B benötigt.

8.2.1. Durchführung

Um die Closed-Loop-Tests des Lane-Followers durchführen zu können, muss zunächst die Testumgebung gestartet werden. Dazu wird das in Abschnitt 7.9 beschriebene Startskript verwendet. Die Testumgebung und das SUT werden auf einem leistungsfähigen Rechner, im Folgenden Simulations-PC genannt, ausgeführt. Dieser startet eine Instanz von CARLA und den ScenarioRunner. Das ausgewählte Szenario verändert die Parameter der Simulation, wie z. B. die Karte, das Wetter, das Ego-Fahrzeug, andere Fahrzeuge, etc. Die in

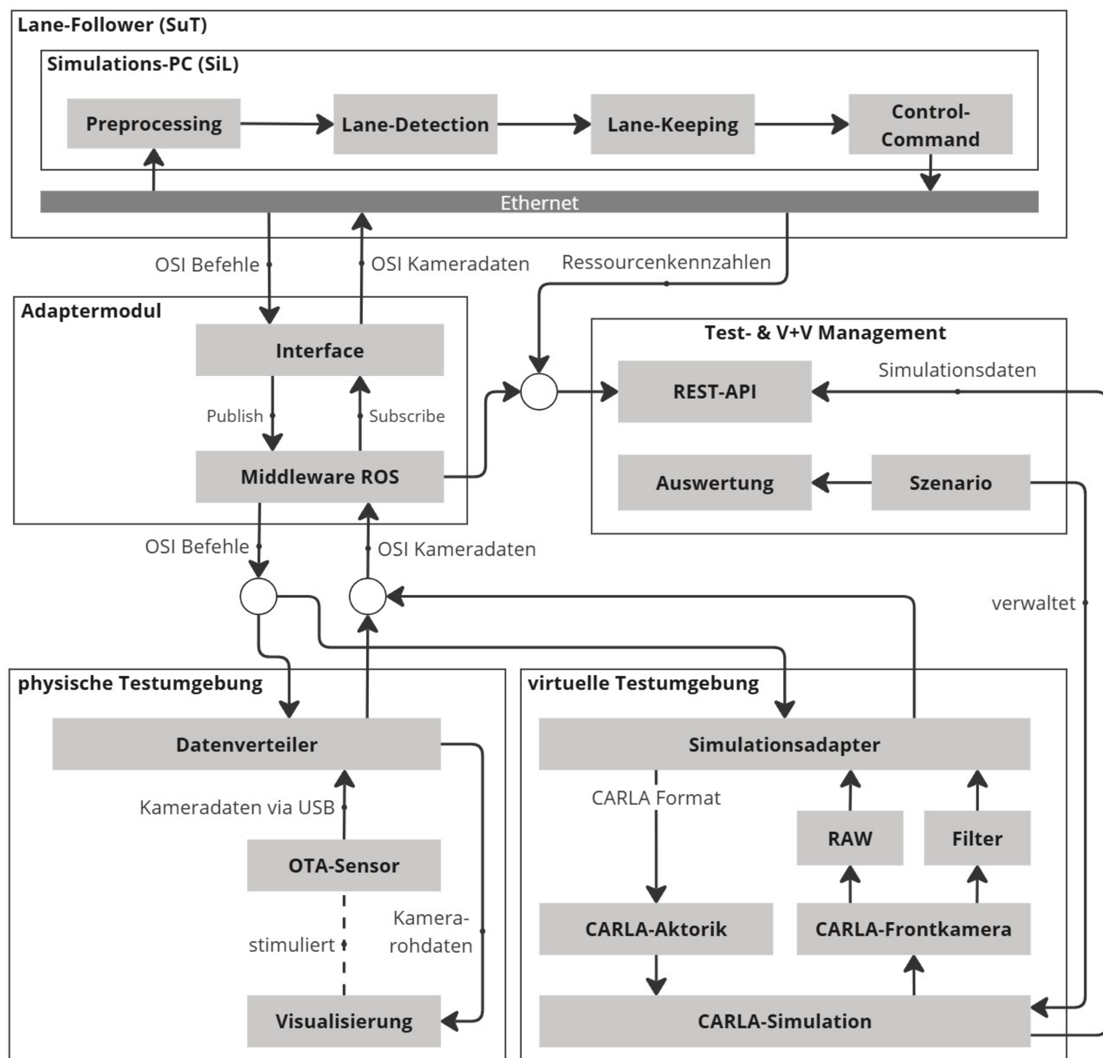


Abbildung 8.1.: Aufbau der Testumgebung mit Integration des Lane-Followers als SUT

der Simulation benötigten Sensoren, wie z. B. der Frontkamerasensor, werden dem bestehenden Ego-Fahrzeug in der Simulation hinzugefügt. Die Bereitstellung dieser Kameradaten und der Eigendaten des Ego-Fahrzeugs erfolgt durch das Adaptermodul, welcher die Daten über die Middleware vom Simulations-Adapter empfängt. Gleichzeitig werden zwei Visualisierungsfenster gestartet. Abbildung 8.3 zeigt die Sensordaten, wie sie im zu testenden System vorliegen, Abbildung 8.2 ermöglicht die Beobachtung des Ego-Fahrzeugs in der Simulation. Mithilfe dieser Visualisierung kann der Entwickler seine Fahrfunktion und die Eingabedaten live überprüfen. Zusätzlich ermöglicht die implementierte Schnittstelle für Systemmonitore die Abfrage verschiedenster Daten zur Laufzeit des Szenarios.

Sobald die Testumgebung bereit ist, kann der Entwickler den Lane-Follower starten. Durch die Festlegung des standardisierten OSI-Formats für die Ein- und Ausgabedaten erfolgt

Abbildung 8.2.: Visualisierung des Ego-Fahrzeugs in der Simulation

eine nahtlose Integration des SUT in die Testumgebung. Das Adaptermodul stellt die Kameradaten im OSI-Format über ein TCP-Websocket zur Verfügung, das von der Vorverarbeitung des Lane-Followers empfangen wird. Ebenso empfängt das Adaptermodul die berechneten Steuerbefehle des SUT und leitet sie über den Simulationsadapter an die CARLA-Simulation weiter. Während der Ausführung des Szenarios kann der Entwickler über die Schnittstellen der Systemmonitore und der verschiedenen Visualisierungen den Lane-Follower überprüfen. Das ausgewählte Szenario (in diesem Fall `FollowLane.xosc`) endet, nachdem das Ego-Fahrzeug eine Strecke von 100 m zurückgelegt hat. Nach Beendigung wird die Testumgebung heruntergefahren und ein Abschlussbericht des Szenarios erstellt.

Durchführung mit OTA-Sensor Wie bereits in Use-Case 1 beschrieben, können auch Testläufe durchgeführt werden, bei dem die Eingabedaten aus einem OTA-Sensor stammen. Der verwendete Hardwareaufbau orientiert sich an Modrakowski et al. (2024). Dabei wird ein physischer Sensor, in diesem Fall eine Webcam, an den Simulations-PC angeschlossen. Die in Abschnitt 7.4 dargestellte Visualisierung der Sensordaten aus der Simulation wird auf dem PC-Monitor angezeigt, auf den die Webcam gerichtet ist. Mit diesem Aufbau ist es möglich, einen realen Sensor in die Testumgebung zu integrieren und CiL-Ansätze zu testen (Sievers et al. 2018; Reway et al. 2018).
Über `./manual_test_run.sh --SELECT_CAMERA=OTA --SELECT_SCENARIO=FollowLane.xosc` kann die Testumgebung mit der Auswahl des Szenarios und dem OTA-Sensor gestartet wer-

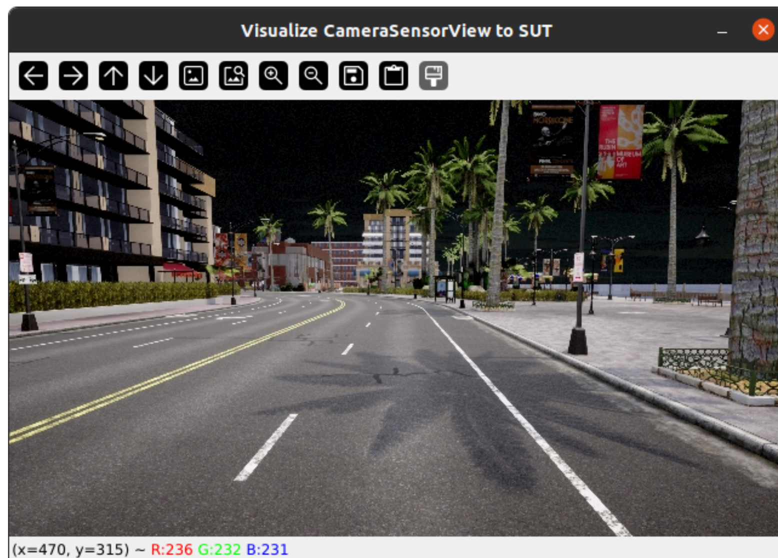


Abbildung 8.3.: Visualisierung der beim SUT vorliegenden Eingabedaten

den. Abbildung 8.4 zeigt den Hardwareaufbau des OTA-Sensors und die Visualisierung der zugehörigen Eingangsdaten. Im linken Bild ist der Visualisierungsmonitor zu sehen, der die Kamera-Rohdaten aus der Simulation im Vollbildmodus anzeigt. Die rechte Seite des Bildes zeigt die USB-Webcam, die in einem bestimmten Abstand zum Monitor ausgerichtet ist. Im rechten Bild sind die resultierenden Eingabedaten der Webcam zu sehen, in denen der Rand des Monitors und der Qualitätsverlust gegenüber den Kamerarohdaten zu erkennen sind.

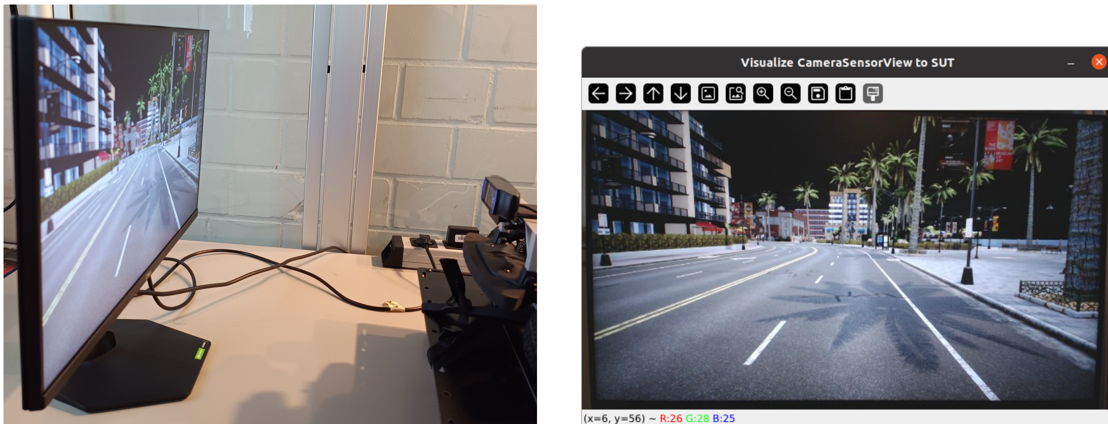


Abbildung 8.4.: Hardware-Aufbau des OTA-Sensors (links) und die daraus resultierenden Eingabedaten (rechts)

8.2.2. Erfüllung der Anforderungen

Anforderung A1: *Die Testumgebung muss in der Lage sein, die vom zu testenden System benötigten Eingangsdaten (Sensordaten) bereitzustellen. Je nach Konfiguration der Testumgebung stammen diese Daten von simulierten Sensoren in der Simulation oder von realen Sensoren über die HiL-Umgebung.*

Die Umsetzung erfolgt durch den Einsatz eines Adaptermoduls in Kombination mit dem Simulationsadapter und dem Datenverteiler. Je nach Art des Sensors (virtuell oder physisch) werden die Sensordaten entweder an den Datenverteiler (physisch) oder an den Simulationsadapter (virtuell) gesendet. Beide Komponenten übertragen die Daten in einem standardisierten Format über die nachrichtenorientierte Middleware ROS, wobei dedizierte Kommunikationskanäle verwendet werden. Der Datenverteiler und der Simulationsadapter agieren als Publisher, während das Adaptermodul als Subscriber fungiert. Diese Publish-Subscribe-Kommunikation ermöglicht die Bereitstellung der Eingabedaten, die im ausgeführten Szenario verifiziert werden können (siehe Unterabschnitt 8.2.1). Zur Verifikation des hybriden Ansatzes wird ein separater Test durchgeführt. Dabei werden simulierte Radardaten aus der virtuellen Testumgebung zusammen mit realen OTA-Daten aus der physischen Testumgebung an das Adaptermodul übertragen. Abbildung 8.5 zeigt den Aufbau der Testumgebung für diesen Test. Die Daten des virtuellen CARLA-Radarsensor werden im OSI-Format (Klasse `osi3::RadarSensorView` [1]) via ROS an das Adaptermodul gesendet und von dort über ein TCP-Websocket an das SUT verschickt. Die Daten des physischen OTA-Sensors werden über den Datenverteiler an das Adaptermodul gesendet, das sie wiederum dem SUT zur Verfügung stellt. In diesem reduzierten Test empfängt das zu testende System nur die Kamera- und Radardaten und zeigt diese auf der Konsole an. Das zu testende System konnte die Daten aus der hybriden Testumgebung erfolgreich empfangen. Die gezeigte Publish-Subscribe-Kommunikation ermöglicht die Bereitstellung der Eingangsdaten auch im vorgestellten hybriden Ansatz und erfüllt damit die Anforderungen vollständig.

Anforderung A1.1: *Die zu versendenden Sensordaten aus A1 stammen aus den virtuellen und physischen Komponenten der Testumgebung: Eingabedaten direkt aus dem Simulationssensor, Eingabedaten aus einer realitätsnäheren Variante des Simulationssensors und Eingabedaten aus einem OTA-Sensor.*

Auch diese Anforderung wird von der Testumgebung erfüllt. Zunächst erfolgt die Auswahl der Eingangsdaten über das Argument im Startskript (siehe Quellcodeauszug 7.5). Die Eingabedaten aus dem Simulationssensor werden vom Simulationsadapter an das Adaptermodul unter dem ROS-Topic `/CameraSensor/RAW` gesendet. Der Simulationsadapter

1. https://opensimulationinterface.github.io/osi-antora-generator/asamosi/latest/gen/structosi3_1_1RadarSensorView.html (Abgerufen am 26.11.2024)

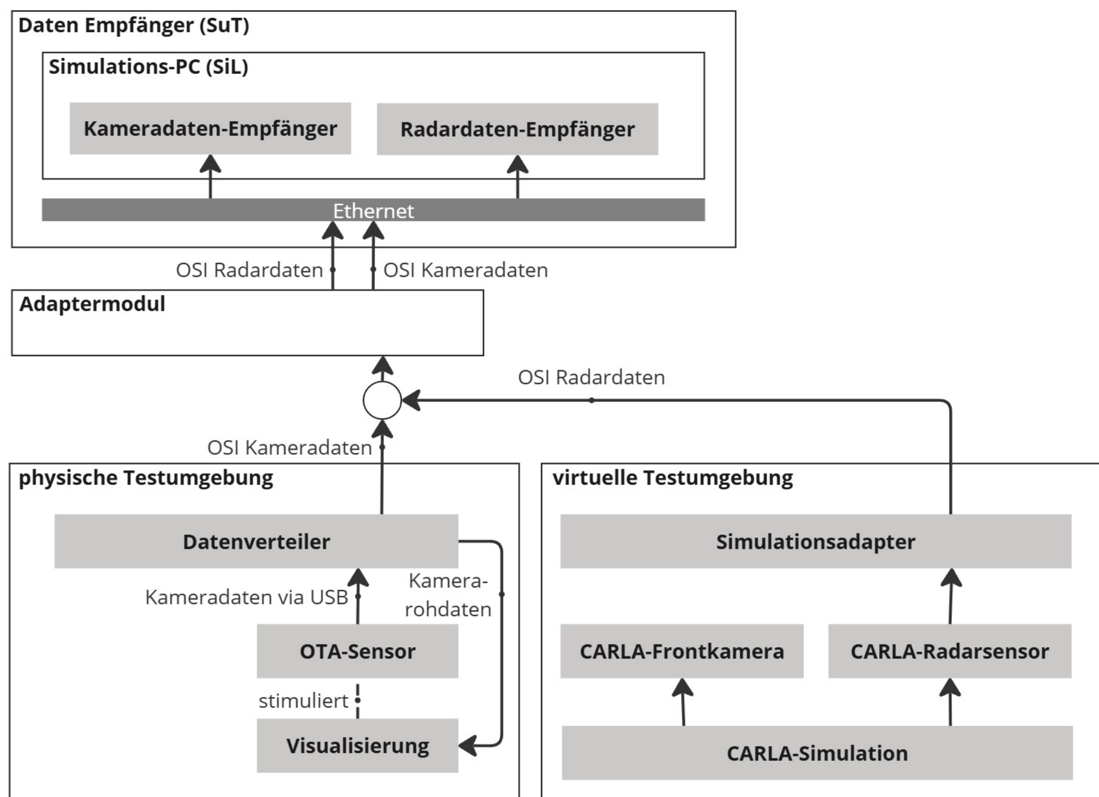


Abbildung 8.5.: Aufbau der Testumgebung im hybriden Ansatz, in dem virtuelle Radardaten mit Kameradaten aus einem realen Sensor verbunden werden

greift direkt auf das rohe Kamerabild aus der Simulation zu und veröffentlicht es unter dem genannten Topic. Die realitätsnähere Variante des Simulationssensors wird ebenfalls vom Simulationsadapter unter dem ROS-Topic `/CameraSensor/Filter` veröffentlicht. Allerdings durchläuft das rohe Kamerabild zunächst ein Sensormodell, in dem ein leichter Unschärfeeffekt des Kameraobjektivs simuliert wird. Die Bereitstellung der Eingangsdaten des OTA-Sensors übernimmt der Datenverteiler in der physischen Testumgebung. Dieser greift über USB direkt auf die verwendete Webcam zu und veröffentlicht die Daten unter dem ROS-Topic `/CameraSensor/OTA`. Das Adaptermodul sendet die ausgewählte Quelle der Eingabedaten an das SUT. Eine vollständige Liste dieser Eingangsdaten und der zugehörigen ROS-Topics ist in der Implementierung unter Abschnitt 7.5 zu finden.

Anforderung A1.2: *Die Testumgebung muss die genannten Sensordaten über eine definierte Schnittstelle, in einem standardisierten Format an das SUT übertragen.*

Diese Anforderung wird durch die Verwendung des OSI-Formats und des TCP-Websockets erfüllt. Wie bereits in Unterabschnitt 6.2.1 erläutert, wird das OSI-Format in vielen Projekten verwendet und kann daher als standardisiertes Datenformat angesehen werden. Für die

Sensordaten wird die OSI-Klasse osi3::CameraSensorView [1] verwendet. Zusätzlich zu den Kameradaten wird eine Kamerakonfiguration übertragen (siehe Quellcodeauszug 8.6).

```
1 import osi_sensorview_pb2
2 camera_sensor_view = osi_sensorview_pb2.CameraSensorView()
3 camera_sensor_view.image_data = img_bytes
4 camera_sensor_view.view_configuration.number_of_pixels_horizontal = 640
5 camera_sensor_view.view_configuration.number_of_pixels_vertical = 360
6 camera_sensor_view.view_configuration.mounting_position.position.x = 1.5
7 camera_sensor_view.view_configuration.mounting_position.position.z = 2.0
8 camera_sensor_view.view_configuration.sensor_id.value = 0
```

Quellcodeauszug 8.6: Verwendung des OSI-Formats als standardisiertes Format für Eingabedaten

Anforderung A2: *Zusätzlich zu den Sensordaten müssen dem zu testenden System auch Eigendaten des Fahrzeugs zur Verfügung gestellt werden.*

Diese Anforderung wird durch den Simulationsadapter und das Adaptermodul erfüllt. Der Simulationsadapter veröffentlicht eine Nachricht im OSI-Format der Klasse osi3::HostVehicleData. Diese enthält den aktuellen Lenkwinkel, die Beschleunigung und die Geschwindigkeit des Ego-Fahrzeugs in der Simulation. Der Simulationsadapter greift über die CARLA-Python-API auf die Daten des Simulationsfahrzeugs zu. Das Adaptermodul abonniert diese Daten und stellt sie zusammen mit den Sensordaten dem SUT zur Verfügung. Das zu testende System (Lane-Follower) in der beschriebenen Version verwendet keine der hier genannten Daten. Um diese Anforderung zu überprüfen, wird das SUT um einen Datenempfänger der Eigendaten erweitert, welche die Daten lediglich auf der Konsole ausgibt. Die erfolgreiche Implementierung und Beobachtung der Datenausgabe des SUT auf der Konsole unterstreicht die Erfüllung dieser Anforderung.

Anforderung A3: *Die Testumgebung muss neben den Sensor- und Eigendaten auch Ground-Truth-Daten an das zu testende System übertragen.*

Diese Anforderung soll die Erstellung von Perzeption-Mockups im SUT durch den Entwickler ermöglichen. Die Mockups verwenden die Ground-Truth-Daten, um die Informationen in das vom Entwickler gewählte spezifische Format der weiteren Planungskomponente umzuwandeln. Um diese Anforderung überprüfen zu können, wird ein Perzeption-Mockup benötigt.

1. https://opensimulationinterface.github.io/osi-antora-generator/asamosi/latest/gen/structosi3_1_1CameraSensorView.html (Abgerufen am 25.11.2024)

Da die Perzeptionskomponente des Lane-Followers (`lane-detection`) eine hohe Komplexität aufweist, wird der Lane-Follower um eine weitere Perzeptionskomponente erweitert. Aufgrund der Einfachheit wird ein Adaptive Cruise Control (ACC) als SUT eingesetzt. Die von der Testumgebung gesendeten Ground-Truth-Daten enthalten bereits die relative Position des vorausfahrenden Fahrzeugs. Das Perzeption-Mockup wandelt diese empfangenen Daten in das spezifische Format des zu testenden Systems um. In diesem Beispiel erwartet die Planungskomponente des ACC die Daten als ROS-Nachricht im Format `geometry_msgs/Pose`. Um die Funktionalität des Mockups testen zu können, wird in der Planungskomponente eine einfache Abfrage durchgeführt: Unterschreitet der Abstand den Grenzwert von 50 m, wird ein Bremssignal gesendet. Dies konnte im Szenario `FollowLeadingVehicle` erfolgreich getestet werden, in der das Ego-Fahrzeug einem vorausfahrenden Fahrzeug folgt, den Grenzwert jedoch nicht unterschreitet. Sollte in Zukunft eine auf Sensordaten basierende Perzeptions-Komponente zur Verfügung stehen, kann das Mockup durch diese ersetzt werden. Durch die Definition des spezifischen Nachrichtenformats muss die Planungskomponente nicht angepasst werden. Die Anforderung gilt somit als erfüllt.

Anforderung A4: *Um in der Testumgebung Closed-Loop-Tests zu ermöglichen, müssen die vom SUT berechnet Steuerbefehle empfangen und verarbeitet werden.*

Diese Befehle werden vom SUT im OSI-Format der benutzerdefinierten Klasse `osi3::ControlCommand` über den TCP-Websocket übertragen. Das Adaptermodul nimmt diese Steuerbefehle entgegen und leitet sie über ROS an den Simulationsadapter weiter. Dieser wandelt die Befehle in ein für CARLA verständliches Format um und überträgt die Befehle an das Ego-Fahrzeug in der Simulation. Damit ist die Anforderung erfüllt. Das OSI-Format kann als standardisiertes Format angesehen werden, jedoch stammt `osi3::ControlCommand` nicht aus den vordefinierten Klassen, sondern wurde hinzugefügt. Bei Closed-Loop-Tests bestehender Testumgebungen, z. B. von Lotz et al. (2019), werden die Steuerbefehle über einen CAN-Bus gesendet und von den Aktoren des Fahrzeugs empfangen. Diese im Kapitel 6 begründete Entscheidung hat den Vorteil, dass nur ein Nachrichtenprotokoll (OSI) verwendet wird, das vom SUT-Entwickler zusätzlich editiert werden kann. Der erfolgreiche Durchlauf des Closed-Loop-Tests (siehe Unterabschnitt 8.2.1), bei dem der Lane-Follower die Spur halten kann, zeigt ebenfalls die Erfüllung der Anforderungen A1, A1.2 und A4.

Anforderung A5: *Das SUT und die Testumgebung müssen lose gekoppelt voneinander arbeiten. Dabei wird das zu testende System aus Sicht der Testumgebung als Blackbox betrachtet, die Eingabedaten empfängt und Steuerbefehle sendet.*

Abgesehen vom Senden der Sensordaten und dem Empfangen von Steuerbefehlen wird das SUT als Blackbox betrachtet, in der die interne Architektur, die Kommunikation und der

verwendete Quellcode unabhängig bleiben müssen. Die Verwendung des hier gezeigten zu testenden Systems (Lane-Follower) zeigt diese lose Kopplung. Für die Integration des SUT musste lediglich der Empfang der Sensordaten und das Senden der Steuerbefehle durch eine Implementierung des OSI-Formats über TCP-Websockets ersetzt werden. Abbildung 8.6 zeigt die Ein- und Ausgabedaten des Lane-Followers und visualisiert die mögliche Betrachtung des SUT als Blackbox. Die interne Architektur der Fahrfunktion, über verschiedene Microser-

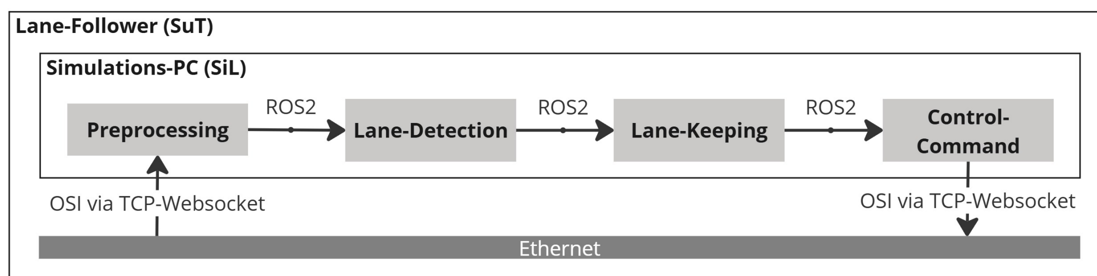


Abbildung 8.6.: Notwendige Änderungen (Eingabe- und Ausgabedaten) des zu testenden Systems für die Integration in die Testumgebung

vices in verschiedenen Containern und dem Einsatz von ROS als Middleware konnten ohne Probleme in die Testumgebung integriert werden.

Anforderung A6: *Das standardisierte Datenformat (A1.2) zum und vom zu testenden System muss von einer Schnittstelle entgegengenommen werden und in die für die jeweilige Umgebung spezifische Datenrepräsentation umgewandelt werden, um eine Interoperabilität zu ermöglichen.*

Diese Anforderung wird in der physischen Testumgebung durch den Datenverteiler und in der virtuellen Testumgebung durch den Simulationsadapter erfüllt. Sie agieren jeweils als Schnittstelle und Übersetzer von Datenformaten. Die Kameradaten der Webcam werden vom Datenverteiler über USB empfangen, in das OSI-Format umgewandelt und über die nachrichtenorientierte Middleware publiziert. Die berechneten Steuerbefehle des SUT kommen im OSI-Format im Simulationsadapter an und werden dort in für CARLA verständliche Signale umgewandelt. Diese Interoperabilität über den Datenverteiler und den Simulationsadapter ermöglicht es auch, weitere Sensoren/Aktoren in die Umgebungen einzubinden (siehe Abbildung 8.5). Beispielsweise wäre es möglich, Steuerbefehle nicht nur an das Ego-Fahrzeug in der Simulation zu senden, sondern auch an einen realen ViL-Prüfstand. Der Datenverteiler müsste dann die Nachrichten im OSI-Format in das spezifische Format des Prüfstands übersetzen.

Anforderung A7: *Um eine flexible Nutzung der Testumgebung zu gewährleisten, muss die Integration der verteilten Komponenten lose gekoppelt sein.*

Durch den Einsatz der nachrichtenorientierten Middleware ROS in der Version 2 wird diese Anforderung erfüllt. Diese ermöglicht eine geräteübergreifende Netzwerkkommunikation zwischen den einzelnen Komponenten. Da die verschiedenen Komponenten alle containerisiert sind und ROS die Ports dynamisch zuweist, muss das gesamte Hostnetzwerk in den Container geroutet werden. Dies wurde getestet, indem der Datenverteiler der physischen Testumgebung auf einem Raspberry Pi installiert und die USB-Webcam an den Pi angeschlossen wurde. Dieser sendet die OTA-Sensordaten der Webcam über die Middleware an das Adaptermodul, das sich auf einem anderen Gerät befindet. Die beiden Geräte sind über ein lokales Netzwerk verbunden, eine lose Kopplung der Komponenten ist somit möglich. Wie in der Anforderung als Beispiel angegeben, ist es zusätzlich möglich, die Simulation auf einem externen Server im gleichen Netzwerk laufen zu lassen. Bei der genannten Dezentralisierung der Testumgebung auf verschiedene Komponenten müssen große Datenmengen, wie die Sensordaten, über das Netzwerk übertragen werden. Da im Konzept der Testumgebung ein Ethernet-Netzwerk mit einer standardmäßigen Datenübertragungsrate von 1 GBit/s verwendet wird, darf diese Grenze nicht überschritten werden. Um die Netzwerkgeschwindigkeit zu testen, werden zwei verschiedene PCs verwendet. Die gesamte Testumgebung mit Ausnahme des Adaptermoduls wird auf PC A und das Adaptermodul auf PC B ausgeführt. Der gesamte Nachrichtenverkehr über die Middleware erfolgt somit über das lokale Ethernet-Netzwerk. Dabei konnte auf PC A eine Netzwerklast von ca. 105 MiB/s (ca. 110 MB/s) gemessen werden, welche unter der maximalen Übertragungsrate liegt. Durch das Hinzufügen weiterer Sensoren (LiDAR, Radar, etc.), neuer Sensormodelle oder die Erhöhung der Auflösung der Kamerasensoren kann die Grenze von 1 GBit/s überschritten werden. Die Anforderung gilt daher als erfüllt.

Anforderung A8: *Die Testumgebung muss eine generische Schnittstelle bereitstellen, die eine flexible Anbindung von Systemmonitoren ermöglicht.*

Diese Schnittstellen müssen Daten über das zu testende System (Ergebnisdaten der Fahrfunktion), als auch Daten über Ressourcenkennzahlen (CPU-Auslastung) bereitstellen. Mithilfe der implementierten REST-API können auf Daten zugegriffen werden. Die Verwendung einer REST-API ermöglicht eine flexible Anbindung von Systemmonitoren. Gleichzeitig können die Endpunkte durch benutzerdefinierte Datenabfragen erweitert werden. Eine Liste aller Endpunkte ist in Abschnitt 7.6 zu finden, die Anforderung gilt somit als erfüllt.

Anforderung A9: *Das zu testende System muss mithilfe von Szenarien innerhalb der Simulation getestet werden können, welche sich am OpenSCENARIO-Standard orientieren.*

Zusätzlich verlangt die Anforderung, dass der Entwickler vordefinierte Szenarien aus einer Liste auswählt. Wie bereits in Abschnitt 7.1 erläutert, erfolgte die Umsetzung mithilfe des Scenario-Runners [1] von CARLA. Dieser ermöglicht das Laden und Ausführen von Szenarien im OpenSCENARIO-Format. Es unterstützt einige Funktionen für `Maneuvers`, `Actions`, `Conditions`, `Stories` und `Storyboards`. Eine Liste der gesamten OpenSCENARIO-Unterstützung ist online verfügbar[2]. Als Beispiel werden zwei Szenarien als `.xosc`-Datei zur Verfügung gestellt: ein Szenario zum Testen des Lane-Followers ohne andere Fahrzeuge und ein Szenario zum Folgen eines vorausfahrenden Fahrzeugs. Die Anforderung verlangt zusätzlich, dass der Entwickler eigene Szenarien definieren und verwenden kann. Dies ist durch die Verwendung des OpenSCENARIO Standards möglich. Der Entwickler kann eigene Szenarien über eine `.xosc`-Datei erstellen und in die Testumgebung laden. Eine Liste der anschließend verfügbaren Szenarien kann über `./manual_test_run.sh --help` ausgegeben werden (siehe Quellcodeauszug 7.5). Dieses sucht dynamisch nach Szenariendateien und zeigt alle verfügbaren Szenarien an. Zusätzlich beschreibt die Anforderung, dass nach Beendigung des Szenarios ein Report generiert werden soll. Dieser wird ebenfalls vom Scenario-Runner erzeugt und enthält Informationen über den Ablauf des Szenarios. Dabei werden die folgenden Tests geprüft: `RunningStopTest`, `RunningRedLightTest`, `WrongLaneTest`, `OnSidewalkTest`, `CheckKeepLane`, `CollisionTest`, `CheckDrivenDistance` und `Duration`. Quellcodeauszug 8.7 zeigt einen Auszug eines beispielhaften Berichts des Szenarios `FollowLane`. Es ist

```json
{
    "scenario": "FollowLane",
    "success": true,
    "criteria": [ {
            "name": "CheckKeepLane",
            "actor": "lincoln.mkz_2017-24",
            "optional": false,
            "expected": 0,
            "actual": 0,
            "success": true
        }, ]
}
```

Quellcodeauszug 8.7: Auszug aus einem beispielhaften Bericht des LaneFollow-Szenarios

zu erkennen, dass das Szenario erfolgreich durchlaufen wurde, da alle Kriterien positiv bewer-

1. https://github.com/carla-simulator/scenario_runner (Abgerufen am 25.11.2024)
2. https://github.com/carla-simulator/scenario_runner/blob/master/Docs/openscenario_support.md (Abgerufen am 25.11.2024)

tet wurden. Diese Dokumentation wird am Ende des Testlaufs generiert und dem Entwickler zur Verfügung gestellt. Damit ist auch diese Anforderung als erfüllt gekennzeichnet.

Anforderung A10: *Die Testumgebung muss verschiedene Visualisierungen zur Verfügung stellen, mit denen die Ein-/Ausgabedaten des SUT und das Verhalten des Fahrzeugs in der Simulation beobachtet werden können.*

Diese Anforderung wird durch einer Python-Anwendung erfüllt, welche die direkten Eingabedaten (Kameradaten) über das TCP-Websocket abruft und visualisiert (siehe Abbildung 8.3 und Abbildung 8.4). Die Ausgabedaten, d. h. die berechneten Steuerbefehle, können in der Visualisierung des Ego-Fahrzeugs im Szenarios entnommen werden (siehe Abbildung 8.7). Die Durchführung des Szenarios mit den erkennbaren Visualisierungen erfüllt diese Anforderung.

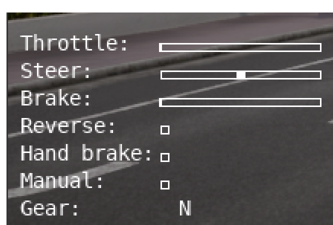


Abbildung 8.7.: Visualisierung der berechneten Steuerbefehle des SUT

8.3. Migration und Tests auf realer Hardware – Use-Case 2A

Basierend auf der erfolgreichen Durchführung eines Closed-Loop-Tests mithilfe eines Lane-Followers folgt in diesem Abschnitt die Verifikation der definierten Anforderungen für Use-Case 2A. Dieser Use-Case versetzt sich wiederum in die Lage des Entwicklers der Fahrfunktion, der einzelne Komponenten des ADS aus der SiL-Umgebung in die HiL-Umgebung portieren möchte. Im vorherigen Use-Case werden alle Komponenten des Lane-Followers (Preprocessing, Lane-Detection, Lane-Keeping, Control-Command) auf einem leistungsfähigen Simulations-PC (SiL-Umgebung) ausgeführt. Um im nächsten Schritt der realen Ausführung im realen Fahrzeug näherzukommen, werden einzelne Komponenten auf realer Fahrzeughardware ausgeführt (HiL-Umgebung). Für diesen Test wird ein NVIDIA Jetson als HiL-Umgebung verwendet. Abbildung 8.8 zeigt die genannte Konfiguration des zu testenden Systems. Die Bereitstellung in der gewünschten Umgebung übernimmt ein Bereitstellungsorchestrator. Es ist zu erkennen, dass die Komponenten `Preprocessing` und `Control-Command` auf dem Simulations-PC ausgeführt werden, während die Komponenten `Lane-Detection` und `Lane-Keeping` auf dem Jetson (HiL) ausgeführt werden. Damit wird der Entwicklungsprozess des Entwicklers simuliert. Die Komponente `Lane-Detection` benötigt durch

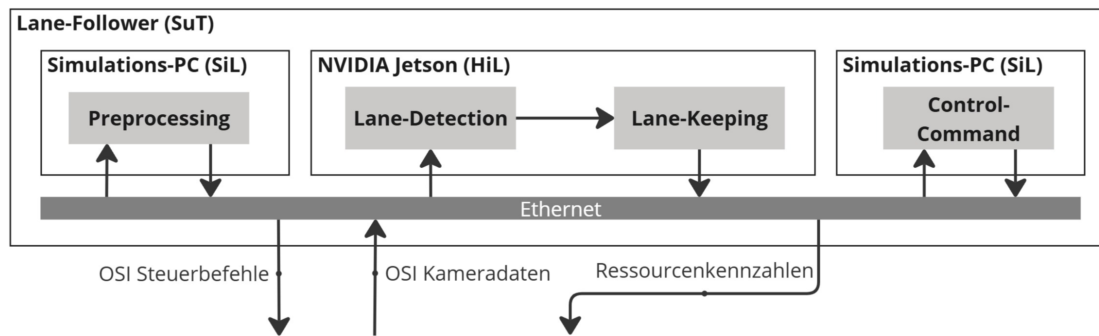


Abbildung 8.8.: Konfiguration des zu testenden Systems und Portierung von der SiL- in die HiL-Umgebung

die Bildverarbeitung und die Erkennung der Linien die meisten Ressourcen der genannten Komponenten. Durch eine Portierung auf reale Hardware (NVIDIA-Jetson) kann überprüft werden, wie sich die Komponente auf Hardware mit begrenzten Ressourcen verhält. Weiterhin zeigt Abbildung 8.8, dass die Kommunikation der Komponenten zwischen der HiL- und SiL-Umgebung über Ethernet erfolgt. Für die Testumgebung wird der unveränderte Aufbau aus Abbildung 8.1 verwendet.

8.3.1. Durchführung

Für die Umsetzung folgt zunächst eine konkrete Beschreibung der verwendeten Umgebungen. Die HiL-Umgebung wird durch den Einsatz eines NVIDIA-Jetson TX1[1] simuliert. Als Betriebssystem ist Ubuntu in der Version 20.04 mit Docker installiert. Der Prozessor des NVIDIA-Jetson ist ein ARM-Cortex-A57 zusammen mit 4 GB Arbeitsspeicher. Die Betriebssystemarchitektur ist somit `linux/arm64`. Als SiL-Umgebung wird ein leistungsfähiger Simulations-PC verwendet, der mit einem 12th Gen Intel(R) Core(TM) i9-12900K, einer NVIDIA-RTX-3090 und 64 GB RAM ausgestattet ist. Die Betriebssystemarchitektur ist x86, auch `linux/amd64` genannt. Der Simulations-PC und der NVIDIA-Jetson sind über ein lokales Ethernet-Netzwerk miteinander verbunden.

Wie bereits beschrieben, sind alle Komponenten der Fahrfunktion (Lane-Follower) als Docker-Container implementiert und verwenden `sut-base-osi:latest` als Basis-Image (A5). Da in diesem Aufbau des zu testenden Systems mehrere Architekturen verwendet werden (`linux/arm64` und `linux/amd64`), müssen das Basis-Image und das fertige Image für beide Architekturen kompiliert werden (engl. „cross-compile“). Um dies nicht einzeln zu tun und manuell einzustellen, welches Image für welche Umgebung verwendet werden soll, kann das Image für beide Architekturen gleichzeitig gebaut werden (siehe Quellcodeauszug 8.8).

1. https://developer.nvidia.com/embedded/jetson-tx1 (Abgerufen am 10.12.2024)

Über das Argument -platform können die beiden benötigten Architekturen angegeben wer-

```
$ docker buildx build --platform linux/amd64,linux/arm64
    -t lane-follower:latest -f docker/Dockerfile .
```

Quellcodeauszug 8.8: Kompilieren der Docker-Images für die Architekturen linux/amd64 und linux/arm64

den. Über `-t lane-follower:latest` wird der Name und Tag des Images und über `-f docker/Dockerfile` das für den Lane-Follower verwendete Dockerfile angegeben.

Bereitstellungsorchestrator. Für die Bereitstellung der einzelnen Komponenten in der gewünschten Zielumgebung (HiL oder SiL) kommt ein Bereitstellungsorchestrator zum Einsatz (A12). Wie in Abschnitt 7.8 beschrieben, wird hierfür das Werkzeug Docker-Swarm verwendet. Der Simulations-PC dient in diesem Beispiel gleichzeitig als Manager und Worker des Swarms. Nach der Initialisierung über `docker swarm init` wird auf der Konsole der Befehl zum Hinzufügen von Workern und das benötigte Token angezeigt. Dieser muss auf der Konsole des Jetson eingegeben werden. Um entscheiden zu können, welche Komponenten auf welcher Umgebung ausgeführt werden sollen (A11), erhalten die Docker-Nodes (Simulations-PC und Jetson) die Rollennamen `SiL` und `HiL`. Diese Rollennamen müssen in der Docker-Compose-Datei des SUT für die jeweiligen Komponenten gesetzt werden. Quellcodeauszug 8.9 zeigt einen Ausschnitt der Datei, in der dies zu sehen ist. Über das Argument `deploy`

```
1  lane-detection:
2      image: lane-follower:latest
3      deploy:
4        placement:
5          constraints:
6            - node.labels.role == HiL
7      networks:
8        - outside
9      command: ros2 run dmv_primeapp dmv_lane_detection
```

Quellcodeauszug 8.9: Auszug aus dem Dockerfile des zu testenden Systems, in dem der Rollenname festgelegt wird

`- placement - constraints` wird festgelegt, dass der Container `lane-detection` in der HiL-Umgebung bereitgestellt wird. Die letztendliche Bereitstellung des gesamten Docker-Stack geschieht über den Befehl:

```
docker stack deploy --compose-file docker/docker-compose.yaml sut
```

Über eine webbasierte Visualisierung kann die Bereitstellung der einzelnen Komponenten in den unterschiedlichen Umgebungen beobachtet werden (siehe Abbildung 8.9). Über den Endpunkt `GET /monitor/sut/container_stats/hil` der REST-API lässt sich verifizieren, dass die Komponente `lane-detection` tatsächlich in der HiL-Umgebung ausgeführt wird. Um die korrekte Funktionsweise des SUT zu testen, kann ein Testlauf des Closed-Loop-Tests

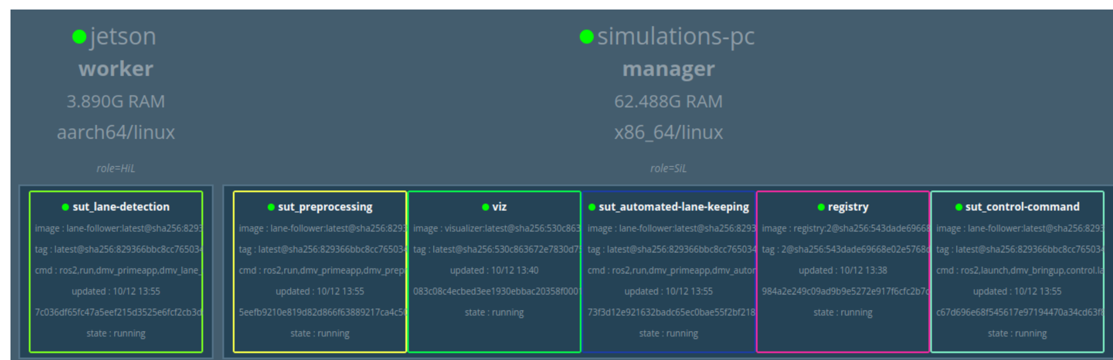


Abbildung 8.9.: Visualisierung des Docker-Swarms und der zugehörigen Services

aus Abschnitt 8.2 verwendet werden. Nach dem Start eines beispielhaften Testlaufs über `./start_test_run.sh` kann der erfolgreiche Abschluss des Szenarios beobachtet werden.

8.3.2. Erfüllung der Anforderungen

Anforderung A11: *Für einen verbesserten Entwicklungsprozess muss es möglich sein, die einzelnen Komponenten des zu testende Systems in der HiL- oder in der SiL-Umgebung bereitzustellen.*

Die Anforderung verlangt, dass einzelne Komponenten des SUT im HiL und andere im SiL ausgeführt werden können, sodass das SUT als Ganzes funktionsfähig bleibt. Dies konnte mit dem in Unterabschnitt 8.3.1 gezeigten Beispiel gezeigt werden. Für einen weiteren Testlauf werden die Komponenten `Preprocessing` und `Control-Command` auf dem Simulations-PC und die Komponenten `Lane-Detection` und `Lane-Keeping` auf dem Jetson ausgeführt. Mit dem Befehl `docker node ps` können die einzelnen Prozesse des Nodes angezeigt werden. Um die Prozesse auf dem Jetson anzuzeigen, wird der Befehl `docker ps` verwendet. Quellcodeauszug 8.10 zeigt die Ausgabe beider Befehle. Diese Aufteilung der Komponenten auf die beiden Umgebungen kann beliebig verändert werden, ohne dass Änderungen am Quellcode vorgenommen werden müssen. Dies wird durch die Verwendung von Containern und die Kompilierung für beide Systemarchitekturen (x86 und ARM) ermöglicht.

```
# auf dem Simulations-PC
[...]  NAME                    IMAGE                  NODE            [...]
[...]  sut_control-command.1   lane-follower:latest   simulations-pc  [...]
[...]  sut_preprocessing.1     lane-follower:latest   simulations-pc  [...]
# auf dem NVIDIA-Jetson
[...]  IMAGE                 [...]  NAMES
[...]  lane-follower:latest  [...]  sut_automated-lane-keeping.1
[...]  lane-follower:latest  [...]  sut_lane-detection.1
```

Quellcodeauszug 8.10: Anzeige der laufenden Docker-Prozesse auf dem Simulations-PC und dem NVIDIA-Jetson

Verschiedene Aufteilungen der Komponenten wurden getestet: Ein hybrider Aufbau wie hier beschrieben, ein Testlauf mit allen Komponenten im SiL und einer mit allen Komponenten im HiL. Da der Ressourcenbedarf des Lane-Followers die Ressourcen der hier verwendeten HiL-Umgebung (NVIDIA Jetson) nicht übersteigt, ist neben der Funktionalität auch ein gleiches Verhalten des SUT erkennbar. Diese Anforderung wird als erfüllt angesehen.

Anforderung A12: *Die Verteilung der einzelnen Services des SUT in den Zielumgebungen muss ein Bereitstellungsorchestrator übernehmen.*

Wie bereits erwähnt, wird Docker-Swarm als Orchestrator verwendet. Zusätzlich verlangt die Anforderung, dass der Entwickler für jede Komponente entscheiden kann, in welcher Umgebung sie ausgeführt werden soll. Dies ist über die Konfiguration in der Datei `docker-compose.yaml` möglich. Quellcodeauszug 8.9 zeigt, dass diese Einstellung für jeden Container vorgenommen werden kann. Der Container-Ansatz setzt voraus, dass alle Komponenten als eigenständige Container zur Verfügung stehen. Die Verwendung des bereits erläuterten Basis-Image (siehe Abschnitt 7.7) und einem gemeinsamen Dockerfile, minimieren diese Hürde.

8.4. Automatisierte Testläufe über eine CI/CD – Use-Case 2B

Es konnte bereits gezeigt werden, dass neben Closed-Loop-Tests in der Testumgebung auch eine Portierung einzelner Komponenten der Fahrfunktion möglich ist. Use-Case 2B erweitert Use-Case 2A um eine vollständige Automatisierung der Testumgebung. Ziel ist es, neue Versionen der Fahrfunktion automatisiert zu testen. Dies soll durch einen Automatisierungsdienst (A13) ermöglicht werden, der die Testläufe mit der neuen Version der Fahrfunktion durchführt und die Ergebnisse dem Entwickler zur Verfügung stellt. Wie in Abschnitt 7.10

beschrieben, wird dies über eine in GitLab laufende CI/CD umgesetzt. Abbildung 8.10 zeigt einen Ausschnitt des instanziierten Konzepts. Die dafür vorgesehenen Komponenten befinden sich im Test- & V+V-Management. Es ist zu erkennen, dass der GitLab-Runner, der von der

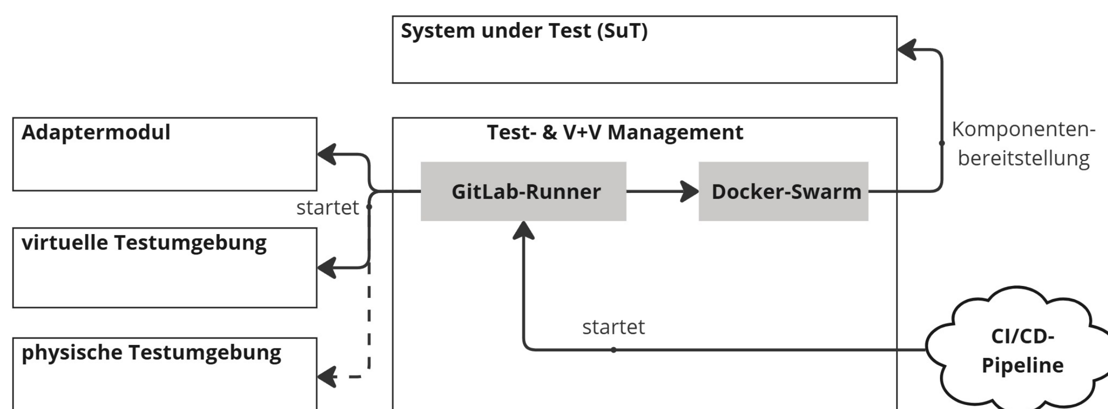


Abbildung 8.10.: Ausschnitts des instanziierten Konzeptes für Use-Case 2B

externen CI/CD-Pipeline im GitLab gestartet wird, das SUT via Docker-Swarm bereitstellt und die gesamte Testumgebung startet. Der gestrichelte Pfeil zur physischen Testumgebung symbolisiert, dass diese nur bei Auswahl der OTA-Eingabedaten gestartet wird. Der Entwickler muss in der Lage sein, den Testlauf zu konfigurieren (A13.1), d. h. die Art der Eingabedaten (A1.1), das verwendete Szenario (A9) und die Zielumgebung der einzelnen Komponenten des SUT (A12).

8.4.1. Durchführung

Für die Durchführung folgt zunächst eine Beschreibung des Ablaufs der automatisierten Testläufe (siehe Abbildung 8.11). Das in der Abbildung gezeigte GitLab-Repository enthält das SUT und die CI/CD-Pipeline. Der GitLab-Runner muss auf einem öffentlichen Server mit Internetzugang laufen. Dieser Runner kann über Secure Shell (SSH) auf einen privaten Server, z. B. im lokalen Netzwerk, zugreifen und dort die Jobs der Pipeline ausführen. In diesem Beispiel werden der GitLab-Runner und die definierten Jobs auf dem Simulations-PC ausgeführt. Die benötigten Docker-Images für die Testumgebung sind auf dem PC vorhanden. Die erforderliche Konfiguration des Testlaufs (A13.1) kann vom Entwickler über eine Konfigurationsdatei und die `docker-compose`-Datei des SUT festgelegt werden. In der Konfigurationsdatei kann die Art der Eingabedaten und das verwendete Szenario festgelegt werden. Es können die gleichen Parameter eingegeben werden wie beim manuellen Start der Testumgebung (siehe Quellcodeauszug 7.5). Welche Komponenten der Fahrfunktion in der SiL- und welche in der HiL-Umgebung bereitgestellt werden, kann über den Rollennamen in der `docker-compose.yaml` des SUT festgelegt werden. Lädt der Entwickler die neue

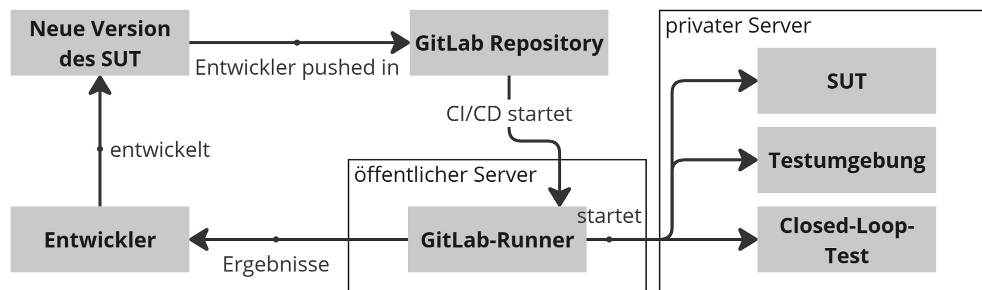


Abbildung 8.11.: Ablauf der automatisierten Testläufe über einen GitLab-Runner

Version zusammen mit der Konfigurationsdatei in das Repository hoch, startet die hinterlegte CI/CD-Pipeline. Der in Abbildung 8.11 zu sehende GitLab-Runner läuft direkt auf dem Simulations-PC und arbeitet die offenen Jobs der Pipeline ab. Dabei wird zunächst das zu testende System (z. B. der Lane-Follower) gestartet und die einzelnen Komponenten in der richtigen Umgebung bereitgestellt. Anschließend startet der Runner die Testumgebung und führt einen Testlauf mit dem vorgegebenen Szenario durch. Wie in der Evaluation zu Use Case 1 beschrieben, werden die ausgewählten Eingabedaten (z. B. Kamerarohdaten) aus der CARLA-Simulation über die Testumgebung an das SUT übergeben und die Steuerbefehle entgegengenommen. Im letzten Schritt der CI/CD-Pipeline werden die Ergebnisse des Testlaufs analysiert und dem Entwickler zur Verfügung gestellt.

Hochladen einer fehlerhaften Version des Lane-Followers. Um das Verhalten der Pipeline zu demonstrieren, folgt zunächst ein Beispiel, bei dem das Szenario nicht erfolgreich durchlaufen wird. Dazu wird die Komponente `Lane-Keeping` des SUT so modifiziert, dass in Kurven die innere Fahrlinie überfahren wird. Abbildung 8.12 zeigt diesen beispielhaften Durchlauf einer Pipeline, der durch das Hochladen der fehlerhaften Version der Fahrfunktion gestartet wird. Es ist zu erkennen, dass das zu testende System und die Testumgebung erfolg-

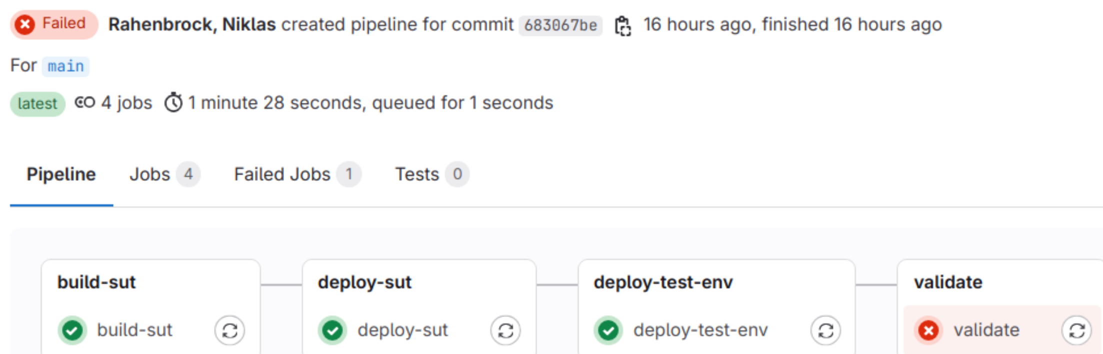


Abbildung 8.12.: Durchlauf der verschiedenen Jobs der Pipeline

reich bereitgestellt werden, aber die Validierung der Ergebnisse fehlschlägt. Mit einem Klick auf den Job `validate` kann das Ergebnisprotokoll des Szenarios (siehe Quellcodeauszug 8.7) angezeigt werden. Die vom Entwickler hochgeladene Version des Lane-Followers kreuzt in Kurven die Fahrlinie. Dies ist im Protokoll am Kriterium `CheckKeepLane` zu erkennen, das mit `success: false` bewertet wurde. Wird eines der Kriterien mit `false` bewertet, ist die Pipeline nicht erfolgreich.

Hochladen einer überarbeiteten Version des Lane-Followers. Der Entwickler der Fahrfunktion sieht nun im Ergebnisprotokoll, dass einige Fahrlinien überfahren wurden. Für eine genauere Analyse kann auch die bereitgestellte Schnittstelle der Systemmonitore verwendet werden. Nach der Entwicklung einer neuen Version des Lane-Followers, die dieses Problem behebt, wird die Pipeline beim Upload erneut gestartet. Da diese Version des Lane-Followers die Fahrlinien in Kurven nicht überfährt, wird auch der Job `validate` mit einem grünen Haken versehen und die Pipeline ist bestanden.

8.4.2. Erfüllung der Anforderungen

Anforderung A13: *Der Automatisierungsdienst muss automatisierte Testläufe von neuen Versionen des zu testenden Systems durchführen können.*

Die Umsetzung erfolgt über die Verwendung eines GitLab-Repositorys und einen GitLab-Runner. Wie bereits in der Implementierung gezeigt, ist es möglich, automatisierte Testläufe durchzuführen. Dazu wird über den GitLab-CI/CD eine Pipeline gestartet, die mehrere Schritte enthält. Im ersten Schritt wird das SUT auf der Zielumgebung bereitgestellt. Dies geschieht über den bereits beschriebenen Bereitstellungsorchestrator. Im zweiten Schritt wird die Testumgebung auf dem Simulations-PC gestartet. Nach Beendigung des Szenarios wird das Ergebnisprotokoll generiert und die Testumgebung heruntergefahren. Dieses Protokoll wird im letzten Schritt der Pipeline verifiziert. Die Testumgebung und das SUT sind vollständig containerisiert. In einer standardmäßige CI/CD-Pipeline, werden zunächst die Docker-Images gebaut und in die Container-Registry des Repositorys hochgeladen. Diese Option ist in der verwendeten GitLab-Instanz nicht möglich. Daher ist es notwendig, dass das Image der Testumgebung in der lokalen Registry des Simulations-PCs vorhanden ist.

Zusätzlich verlangt die Anforderung, dass die Ergebnisprotokolle dem Entwickler zur Verfügung gestellt werden. Diese Anforderung wird durch den letzten Schritt der Pipeline (`validate`) erfüllt. Hier wird der Inhalt des Protokolls angezeigt und analysiert (siehe Quellcodeauszug 8.7). Mit diesem Ansatz ist es möglich, Testläufe durchzuführen, mit denen neue Versionen der Fahrfunktion automatisiert getestet werden können, was den Entwicklungsprozess einer solchen Funktion verbessert.

Anforderung A13.1: *Für die automatisierten Testläufe muss eine Konfiguration der Testumgebung durch den Entwickler möglich sein.*

Diese Anforderung erweitert A13 um eine Konfiguration der automatisierten Testläufe. Der Entwickler muss die Möglichkeit haben, die Art der Eingabedaten, das verwendete Szenario und die Zielumgebung der einzelnen Komponenten des SUT auszuwählen. Dies wird durch eine Konfigurationsdatei ermöglicht, die im Repository der Fahrfunktion vorhanden sein muss. Quellcodeauszug 8.11 zeigt eine beispielhafte Konfigurationsdatei. In dieser Datei

```
1 {
2     "SELECTED_CAMERA": "RAW",
3     "SELECTED_SCENARIO": "FollowLane.xosc"
4 }
```

Quellcodeauszug 8.11: Konfigurationsdatei für automatisierte Testläufe

kann der Entwickler angeben, mit welchen Eingabedaten und welchem Szenario der Testlauf durchgeführt werden soll. Beim automatisierten Testlauf werden diese Informationen über Umgebungsvariablen an die Testumgebung übergeben. Die Angabe der Zielumgebungen der einzelnen Komponenten der Fahrfunktion erfolgt über die `docker-compose.yml` Datei des SUT, wie bereits in Quellcodeauszug 8.9 beschrieben. Damit gilt diese Anforderung als erfüllt.

8.5. Validierung

Nachdem im vorangegangenen Kapitel die Erfüllung der definierten Anforderungen durch die Verifikation nachgewiesen wurde, folgt in diesem Kapitel die Validierung der entwickelten Test- und Simulationsumgebung. Dabei wird untersucht, inwieweit die Umgebung effektiv zur Verbesserung der Entwicklungsprozesse und zur Evaluierung sicherheitsrelevanter Funktionen beiträgt. Zunächst wird ein Vergleich zwischen einer monolithischen und einer mikroserviceorientierten Architektur angestellt, um die Vorteile und Herausforderungen der modularen Softwarearchitektur im Kontext der Automobilindustrie zu analysieren. Anschließend werden die konkreten Vorteile einer losen Kopplung zwischen Testumgebung und zu testendem System in Verbindung mit der Blackbox-Sichtweise näher erläutert. Abschließend wird die Durchführung von Open-Loop-Tests beschrieben, die in der bisherigen Evaluierung noch nicht behandelt wurde.

8.5.1. Vergleich der Laufzeit des Lane-Followers

In komplexen ADS spielt die Softwarearchitektur eine wichtige Rolle für die Entwicklung, Wartung und Performance des Systems. Wie bereits in Abschnitt 2.4 beschrieben, kann grundsätzlich zwischen monolithischer und modularer Softwarearchitektur unterschieden werden. Während Monolithen für ihre Effizienz und Laufzeit bekannt sind, gewinnen mikroserviceorientierte und modulare Architekturen zunehmend an Bedeutung (Berger et al. 2017; Blinowski et al. 2022; Lotz et al. 2019). Angelehnt an die klassische Softwareentwicklung versprechen sie eine bessere Skalierbarkeit, Wartbarkeit und Testbarkeit, können aber auch Nachteile in der Performance mit sich bringen. Ein wichtiger Aspekt für das hier vorgestellte System ist die Laufzeit der Verarbeitungskette. Zum Vergleich der beiden Architekturen wird die Laufzeit der gesamten Fahrfunktion (Lane-Follower) gemessen, d. h. die Zeit vom Empfang der Eingangsdaten bis zum Senden der Steuerbefehle. Der im Closed-Loop-Test verwendete Lane-Follower orientiert sich durch die Verwendung verschiedener Komponenten in unterschiedlichen Containern bereits an einer mikroserviceorientierten Architektur. Um eine monolithische Architektur zu simulieren, aber die Fahrfunktion nicht zu verändern, erfolgt eine Verschmelzung der einzelnen Funktionen zu einer großen Komponente. Die interne Kommunikation über ROS2 wird durch Funktionsaufrufe ersetzt. Wie bereits in der Verifikation (siehe Unterabschnitt 8.2.2) gezeigt, ist eine Änderung der internen Architektur des SUT ohne Änderung der Testumgebung möglich. Die Messdaten der Laufzeit für beide Testläufe können über die REST-API (`GET /monitor/sut/runtime`) abgefragt werden. Abbildung 8.13 zeigt den Vergleich der Messergebnisse für die monolithische und die mikroserviceorientierte Architektur des Lane-Followers. Auf der x-Achse ist die Testdauer von 1000 Sekunden aufgetragen, wobei die Messdaten alle 100 ms erfasst werden. Die y-Achse zeigt die Laufzeit der beiden SUT. Es ist deutlich zu erkennen, dass die Laufzeit der monolithischen Architektur (blau) deutlich unter der der mikroserviceorientierten Architektur (orange) liegt. Der Monolith weist eine mittlere Laufzeit von 8,69 ms, einen Median von 8,59 ms und einen Maximalwert von 22,21 ms auf. Im Vergleich dazu erreichen die Mikroservices eine höhere durchschnittliche Laufzeit von 13,57 ms, einen Median von 13,36 ms und einen Maximalwert von 42,23 ms.

Ergebnisbegründung. Die unterschiedliche Laufzeit des Lane-Followers in den beiden Architekturen lässt sich wie folgt begründen. Die Aufteilung der Fahrfunktion in mehrere Komponenten hat zur Folge, dass eine Kommunikation zwischen den Komponenten notwendig ist. Diese erfolgt in diesem Beispiel über die Middleware ROS. Im Verlauf der Verarbeitungskette müssen die Daten in ROS Nachrichten verpackt, versendet und wieder entpackt werden. Obwohl in diesem Beispiel alle Dienste auf einem Gerät ausgeführt werden, simuliert ROS eine netzwerkbasierte Übertragung. ROS basiert zwar auf DDS, jedoch können

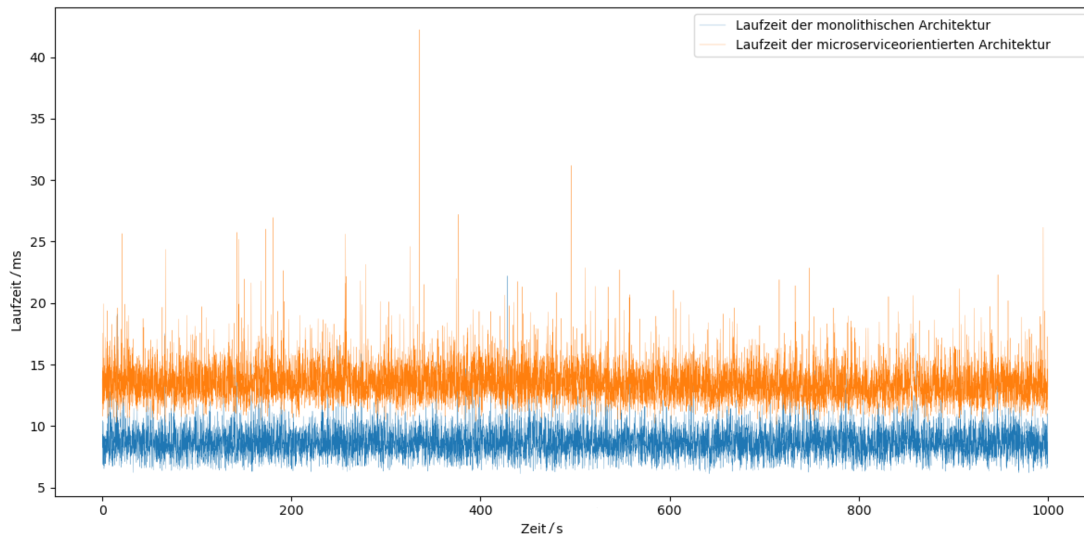


Abbildung 8.13.: Vergleich der Laufzeit des Lane-Followers zwischen monolithischer und mikroserviceorientierter Architektur

Ansätze wie die Verwendung eines geteilten Speichers (engl. „shared-memory“) durch eine mögliche Verteilung auf verschiedene Geräte (Use-Case 2A) nicht genutzt werden (Paul et al. 2024). Diese zusätzlichen Schritte sind zeitaufwendiger als einfache Funktionsaufrufe und Speicherzuweisungen in der Implementierung des Monolithen.

Ergebnisbewertung. Für die Bewertung des Ergebnisses muss zunächst die verwendete Fahrfunktion und die Implementierung des Lane-Followers kritisch betrachtet werden. Die Verarbeitungskette vom Empfang der Sensordaten bis zum Senden der Steuerbefehle ist rein seriell. Das bedeutet, dass jede Komponente auf dem Ergebnis der vorhergehenden Funktion aufbaut. Beispielsweise benötigt die Komponente `Lane-Detection` das vorverarbeitete Kamerabild aus `Preprocessing`. Dadurch ist es nicht möglich, Funktionen zu parallelisieren oder zu skalieren, was normalerweise ein großer Vorteil von mikroserviceorientierten Architekturen ist. Diese Parallelisierung wird von aktuellen Architekturen von ADSs, wie PARA-Drive von Weng et al. (2024) oder dem Framework Adaptive AUTOSAR (siehe Unterabschnitt 2.4.1) unterstützt.

Ein zentraler Aspekt in diesem Zusammenhang ist die Abwägung zwischen geringer Laufzeit und Verbesserung der Entwicklungsmöglichkeiten (engl. „trade-off“). Modulare oder mikroserviceorientierte Architekturen bieten in der Entwicklung viele Vorteile, wie die Förderung der Wartbarkeit und Skalierbarkeit des Systems, erhöhen jedoch auch die Laufzeit einer solchen Verarbeitungskette (Berger et al. 2017). Außerdem können einzelne Services unabhängig voneinander getestet werden (A11), was insbesondere für sicherheitsrelevante

Anwendungen interessant ist, da somit abgekapselte Tests möglich sind. Im Bereich der Entwicklung hat eine monolithische Architektur einige Nachteile. Die enge Kopplung der Komponenten erschwert die Wartbarkeit und die Entwicklung neuer Funktionen. Dies steht ebenfalls im Widerspruch zu Anforderungen des Software-Definied Vehicles, wie den OTA-Updates und der Mitnahme von Softwareteilen für neue Versionen (Slama et al. 2023). Diese Aspekte im Hinblick auf die Laufzeit werden auch in der Literatur kritisch diskutiert. So zeigt Blinowski et al. (2022), dass eine Mikroservice-Architektur im Vergleich zu einer monolithischen Struktur zu einer höheren Laufzeit führt, insbesondere wenn die Tests auf einer einzelnen Maschine durchgeführt werden. Um diese beiden Architekturen zu kombinieren, hat Google ein Framework namens „Service Weaver“[1] vorgestellt, das in Su und Li (2024) untersucht wird. Dieses Framework ermöglicht einen modularen Monolithen, bei dem die Software als Monolith entwickelt und in Form von Mikroservices bereitgestellt wird. Das Framework verspricht durch gezielte Serialisierung und direkte Methodenaufrufe eine hohe Performance, was durch Su und Li (2024) bestätigt werden konnte. Diese Bewertung sowie der Vergleich der Architekturen werden im Fazit Kapitel 9 erneut aufgegriffen und zusammengeführt.

8.5.2. Lose Kopplung und Blackbox-Ansatz

Wie auch in A5 beschrieben, wird eine lose Kopplung zwischen der Testumgebung und dem zu testenden System gefordert. In Absatz 8.2.2 wurde diese Anforderung bereits verifiziert und bestätigt. Das SUT wird als Blackbox betrachtet, welche Sensordaten empfängt und Steuerbefehle sendet. Dieser Ansatz hat sowohl Vor- als auch Nachteile. Der größte Vorteil ist die Unabhängigkeit der Fahrfunktion (SUT). Diese kann in einer beliebigen Programmiersprache auf einer beliebigen Architektur entwickelt werden, solange sie die Ein- und Ausgabedaten im richtigen Format (OSI) empfängt und sendet. Erfolgt die Kommunikation, z. B. zwischen verschiedenen Komponenten der Fahrfunktion, über Ethernet, kann der Entwickler frei entscheiden. Soll eine Fahrfunktion getestet werden, die nicht für diese Testumgebung entwickelt wurde und die Formate nicht unterstützt, kann dies mithilfe eines Wrappers ermöglicht werden. Dieser dient als Schnittstelle für die spezifischen Sensordaten und Steuerbefehle. Ein schneller Test verschiedener Fahrfunktionen ist ebenfalls möglich. Anforderungen an unterschiedliche Sensorkonfigurationen können in der Testumgebung einfach angepasst und die Daten über das Adaptermodul übertragen werden. Durch den konkreten Rahmen über das standardisierte OSI-Format kann das SUT für jeden Testlauf ausgetauscht werden.

Ein wesentlicher Nachteil dieses Ansatzes ist die fehlende Unterstützung von Whitebox-Tests. Aufgrund der offenen internen Kommunikation und Architektur des SUT können keine

1. https://serviceweaver.dev/ (Abgerufen am 06.02.2025)

generischen Schnittstellen für Monitoring und Analyse zur Verfügung gestellt werden. Diese müssen für jede Fahrfunktion vom Entwickler definiert und der Testumgebung hinzugefügt werden. Der Integrationsansatz des SUT in die Testumgebung orientiert sich an einer zentralen Architektur (siehe Abschnitt 2.4), in der das zu testende System auf einem leistungsfähigen PC läuft und die Sensordaten über das Netzwerk empfängt. In den Arbeiten von Sievers et al. (2018) oder Ruehl und Bronner (2024) wird ein dezentraler Ansatz mit verschiedenen Sensor-ECUs betrachtet. Dabei erfolgt beispielsweise die Objekterkennung direkt auf einer mit dem Kamerasensor integrierten ECU. Diese direkt in die Fahrzeughardware und Sensorik integrierten Ansätze können in der hier vorgestellten Testumgebung nicht direkt getestet werden. Das verwendete Konzept der Testumgebung ist darauf ausgelegt, Sensordaten in abstrahierter Form über Ethernet zur Verfügung zu stellen, wodurch der direkte Zugriff auf die Sensorhardware und die spezifischen Eigenschaften der ECUs entfällt. Es ist jedoch zu beachten, dass der Trend in der Automobilindustrie gerade im Bereich der SDV zunehmend in Richtung zentralisierter Computing-Architekturen geht, insbesondere durch den Einsatz von künstlicher Intelligenz und neuronalen Netzen für sicherheitsrelevante Anwendungen (Schleicher und Grigorescu 2020; Slama et al. 2023, S.32f).

8.5.3. Durchführung von Open-Loop-Tests

Die Verifikation der Anforderungen und Use-Cases erfolgt bisher über einen Closed-Loop-Test mithilfe eines einfachen Lane-Followers. Dieser berechnet auf Basis von Sensordaten spezifische Steuerbefehle, die in der Simulation an das Fahrzeug gesendet werden. Neben Closed-Loop-Tests werden in vielen Anwendungsfällen zusätzlich Open-Loop-Tests durchgeführt. Bei diesen Tests werden keine Steuerbefehle oder ähnliches Feedback an die Simulation zurückgegeben. Ein in der Literatur häufig genanntes Beispiel ist der Test der Objekterkennung, z. B. die Erkennung anderer Verkehrsteilnehmer (Reway et al. 2018; Qian et al. 2022; Chen et al. 2016). Um Open-Loop-Tests in der gezeigten Testumgebung zu ermöglichen, müssen einige Änderungen vorgenommen werden. Da keine Steuerbefehle an das Fahrzeug gesendet werden, müssen diese simuliert werden, z. B. über den CARLA-Autopiloten. Mit diesem kann das Fahrzeug in der Simulation gefahren und verschiedene Manöver wie Abbiegen, Überholen, Spurwechsel etc. durchgeführt werden. Damit kann das vorgestellte Konzept des szenariobasierten Testens (A9) beibehalten werden, mit dem auch verschiedene Szenarien für die Objekterkennung erstellt werden können. Lediglich die Ergebnisse der Open-Loop-Tests können nicht direkt von der Testumgebung empfangen werden. In der bisherigen Konfiguration werden die Steuerbefehle im standardisierten OSI-Format übertragen. Für das Beispiel der Objekterkennung könnte dies ebenfalls im OSI-Format über die Klas-

sen `osi3::DetectedMovingObject`[1] übertragen werden. Falls die Ergebnisse nicht in das Format einer OSI-Klasse überführt werden können, müssen Systemmonitore direkt auf die Websocket-Kommunikation des SUT zugreifen. Dies erfordert eine Anpassung des Adaptermoduls und die Erstellung einer neuen Systemmonitor-Schnittstelle. Über diese Schnittstellen kann auf die Ergebnisse der Objekterkennung zugegriffen und Analysen durchgeführt werden.

8.6. Zusammenfassung der Evaluierung und Zielabdeckung

Die in diesem Kapitel vorgestellte Evaluation basiert auf den bereits definierten Use-Cases aus Kapitel 3 und den daraus abgeleiteten Anforderungen aus Kapitel 4. Für diese Evaluation, wird für jeden Use-Case eine Instanziierung des vorgestellten Konzepts vorgenommen, die Durchführung der Experimente beschrieben und die Erfüllung der Anforderungen betrachtet. Die verwendeten Methoden und Ergebnisse werden ausführlich beschrieben. In der Evaluation von Use-Case 1 konnte bestätigt werden, dass Closed-Loop-Tests eines Lane-Followers in der entwickelten Testumgebung möglich sind. Der Use-Case 2A beschäftigt sich mit der Portierung einzelner Komponenten des SUT in eine Zielumgebung (HiL, SiL). Diese Funktionalität wurde mit einzelnen Komponenten des Lane-Followers erfolgreich getestet. Im Use-Case 2B wurde eine vollständige Automatisierung der Testumgebung beschrieben. Mithilfe einer CI/CD-Pipeline und eines GitLab-Runners konnte diese Automatisierung evaluiert werden. Die Verifikation in der Evaluation zeigt, dass alle definierten Anforderungen durch die Testumgebung abgedeckt werden. Da der Lane-Follower nicht repräsentativ für ein gesamtes ADS ist, mussten für die Verifikation einiger Anforderungen zusätzliche Komponenten bzw. Quellcode hinzugefügt werden. Während die Verifikation erfolgreich verlief, zeigte die Validierung des Systems einige Einschränkungen des Konzepts auf.

Die in der Einleitung unter Abschnitt 1.2 formulierte Forschungsfrage lautet: *„Wie kann eine Simulations- und Testumgebung die Entwicklung und frühzeitige Evaluation modularer Softwaresysteme in unterschiedlichen Entwicklungsstufen eines Software-Defined Vehicles ermöglichen?“* Die Ergebnisse der durchgeführten Evaluation belegen die Beantwortung der Forschungsfrage. Durch die möglichen Eingabedaten von Ground-Truth-Daten, über Kamerarohdaten aus der Simulation, über verschiedene Sensormodelle, bis hin zu Daten aus einem realen OTA-Sensor, kann eine inkrementelle Entwicklung eines ADS erfolgen. Insbesondere mit der Möglichkeit, Ground-Truth-Daten der Umgebung zu senden, können Mockups von Perzeptionskomponenten erstellt werden, um bereits mit der Entwicklung der weiteren Komponenten zu beginnen. Gleichzeitig können durch den Einsatz von OpenSCENARIO verschiedene standardisierte Szenarien verwendet werden, um reproduzierbare Tests zu er-

1. https://opensimulationinterface.github.io/osi-antora-generator/asamosi/latest/gen/structosi3_1_1DetectedTrafficSign.html (Abgerufen am 19.12.2024)

möglichen. Die dargestellte Portierung einzelner Komponenten des SUT von einer SiL- in eine HiL-Umgebung ermöglicht ebenfalls eine verbesserte Entwicklung. Komponenten können zunächst in der SiL-Umgebung auf korrekte Funktion getestet werden, bevor sie in einer HiL-Umgebung mit begrenzten Ressourcen eingesetzt werden. Dies konnte am Beispiel der Komponente `lane-detection` des Lane-Followers erfolgreich demonstriert werden. Neben der manuellen Durchführung der Testläufe mit den genannten Konfigurationen der Testumgebung sind auch automatisierte Testläufe mit dem Automatisierungsdienst möglich. Da die V+V eines ADS ein zentraler Aspekt der Entwicklung ist, trägt eine optimierte V+V durch die entwickelte Testumgebung zur Verbesserung des Entwicklungsprozesses bei.

Insgesamt lässt sich festhalten, dass die vorgestellte Simulations- und Testumgebung einen wesentlichen Beitrag zur Verbesserung der Entwicklung modularer Softwaresysteme leisten kann und somit die in der Einleitung formulierte Forschungsfrage erfolgreich beantwortet werden kann.

9. Zusammenfassung

9.1. Ergebnisse und Fazit

Diese Arbeit beschäftigt sich mit der Entwicklung einer Simulations- und Testumgebung zur Unterstützung der V+V von sicherheitsrelevanten Funktionen eines SDVs. Mit der zunehmenden Bedeutung von Software in Fahrzeugen gewinnt das Testen solcher Systeme in der Entwicklung an Relevanz. Angesichts der hohen Anzahl der erforderlichen Testkilometer und der neuen Anforderungen an ein SDV steigt der Bedarf an einer wiederverwendbaren und erweiterbaren Testumgebung, die eine effizientere Entwicklung ermöglicht. Um diesen Entwicklungsprozess mit einer geeigneten Testumgebung zu unterstützen, beantwortet diese Arbeit die Frage: „Wie kann eine Simulations- und Testumgebung die Entwicklung und frühzeitige Evaluation modularer Softwaresysteme in unterschiedlichen Entwicklungsstufen eines Software-Defined Vehicles ermöglichen?“

Zur Zielerreichung werden zunächst die relevanten Grundlagen für hochautomatisierte Fahrzeuge, deren Entwicklung und die dafür notwendigen Testumgebungen dargestellt (Kapitel 2). Als Grundkonzepte dienen dabei die verschiedenen Stufen eines ADS (SAE Level) und das PPC-Schema. Anschließend werden SDVs und die wesentlichen Aspekte ihrer kontinuierlichen Weiterentwicklung beschrieben. Es folgt eine Einführung in die Entwicklung solcher Systeme, unterstützt durch XiL-Ansätze und geeignete Testumgebungen. Darüber hinaus wird die Referenzarchitektur von Nickovic et al. (2017) vorgestellt und erläutert, welchen Einfluss verschiedene Sensorsimulationen auf die Entwicklung haben können.

Auf Basis dieser Grundlagen werden Use-Cases (Kapitel 3) und Anforderungen (Kapitel 4) definiert. Die Use-Cases beschreiben aus Sicht eines ADS-Entwicklers die Durchführung von Closed-Loop-Tests eines Lane-Followers in der entwickelten Testumgebung. Anschließend wird die Portierung des SUT von der SiL- in die HiL-Umgebung dargestellt, gefolgt von der vollständigen Automatisierung der Testumgebung mittels CI/CD-Pipeline. Diese Use-Cases bilden die Grundlage für die definierten Anforderungen und veranschaulichen die Einsatzmöglichkeiten der entwickelten Testumgebung.

Basierend auf diesen Use-Cases und Anforderungen werden in Kapitel 5 verwandte Arbeiten untersucht, um anschließend einen Handlungsbedarf abzuleiten. Neben den Testfeldern für CPS werden auch Testumgebungen für ADS analysiert. Die Analyse zeigt, dass keine der

existierenden Arbeiten alle definierten Anforderungen erfüllt, wodurch der Bedarf für eine neue Testumgebung zur Verbesserung der ADS-Entwicklung entsteht.

Kapitel 6 leitet aus dem resultierenden Handlungsbedarf und den Anforderungen eine Systemarchitektur für eine Testumgebung als Lösungsansatz dieser Arbeit her. Diese Systemarchitektur basiert auf der beschriebenen Referenzarchitektur von Nickovic et al. (2017) und der physischen Testfeld-Architektur von Brinkmann (2018). Dabei waren einige Anpassungen von der maritimen Domäne auf das Testen eines ADS notwendig. Die Testumgebung nutzt ein Adaptermodul und das einheitliche Datenformat OSI, um das zu testende System nahtlos zu integrieren. Für die interne Kommunikation verwendet die Testumgebung eine nachrichtenorientierte Middleware mit einem Publish-Subscribe-Muster. Damit ist es möglich, sowohl virtuelle als auch physische Komponenten der Testumgebung, wie z. B. OTA-Sensoren, anzubinden. Durch die Verwendung von Standards wie OSI, OpenSCENARIO und OpenDRIVE wird die Reproduzierbarkeit von Testläufen verbessert und die Nähe zu industriellen Prozessen erhöht. Im Hinblick auf eine verbesserte Entwicklung von ADS können verschiedene Sensormodelle hinzugefügt werden, um realistischere Sensordaten als Eingabedaten für das SUT zu verwenden.

Ebenfalls ermöglicht das Konzept die Portierung einzelner Komponenten des SUT von einer SiL- in eine HiL-Umgebung. Dadurch können einzelne Funktionen des SUT auf einer Hardware mit begrenzten Ressourcen getestet werden. Gleichzeitig ermöglicht ein Automatisierungsdienst die vollständige Automatisierung von Testläufen, wodurch leicht reproduzierbare Tests möglich sind. Der in den Anforderungen geforderte Black-Box-Ansatz des zu testenden Systems hat neben der einfachen Integration verschiedener SUT auch einige Nachteile. Zwar ermöglicht dieser Ansatz durch die standardisierte Ethernet-Kommunikation eine Portierung in eine HiL-Umgebung, jedoch können einige Konzepte nicht umgesetzt werden. Beispielsweise ist eine tiefe Hardwareabstraktion mit unabhängigen Partitionen für verschiedene Teilsysteme des SUT nicht möglich. Dies ist der Funktionsweise des Bereitstellungsorchestrators geschuldet, der in diesem Konzept lediglich Docker-Container auf verschiedenen Geräte bereitstellen kann. In Kapitel 7 wird das vorgestellte Konzept der Systemarchitektur durch eine prototypische Implementierung umgesetzt, welche als Basis der Evaluation genutzt wird.

Anhand der Anforderungen wurde in der Evaluation (Kapitel 8) eine Verifikation dieser Implementierung und des vorgestellten Konzepts durchgeführt, wobei die Testumgebung zum Testen eines beispielhaften Lane-Followers als SUT verwendet wurde. Dabei konnten alle definierten Anforderungen durch die Testumgebung als erfüllt gekennzeichnet werden. In den gezeigten Closed-Loop-Tests des Lane-Followers konnte gezeigt werden, dass Testläufe mit unterschiedlichen Eingabedaten, mit unterschiedlichen Szenarien, mit einer Portierung des SUT in eine HiL-Umgebung und mit einer vollständigen Automatisierung über eine GitLab

CI/CD möglich sind. Diese Aspekte unterstützen die Entwicklung eines ADS und ermöglichen reproduzierbare Tests, die sich schrittweise von der vollständigen Simulation hin zu Tests mit realer Hardware und Sensoren entwickeln. Die Evaluierung wird mit einer Validierung des Systems abgeschlossen, bei der Aspekte betrachtet werden, die nicht Teil der Anforderungen sind. Durch die Möglichkeit, einzelne Komponenten des SUT in unterschiedliche Zielumgebungen zu portieren, wird ein Vergleich zwischen zwei verschiedenen Versionen des Lane-Followers durchgeführt. Dabei wird die Laufzeit zwischen einer monolithischen und einer mikroserviceorientierten Softwarearchitektur verglichen. Die Ergebnisse zeigen eine Erhöhung der Laufzeit bei einer Aufteilung des SUT in verschiedene Komponenten. Die Analyse der Messergebnisse verdeutlicht den Einfluss der gewählten Softwarearchitektur auf die Systemperformance. Während der monolithische Ansatz eine niedrigere Laufzeit aufweist, bietet die mikroserviceorientierte Architektur Vorteile hinsichtlich Modularität, Skalierbarkeit und Wartbarkeit. Besonders für die Entwicklung sicherheitskritischer Funktionen in SDVs sind diese Aspekte essenziell, da sie eine isolierte Validierung einzelner Komponenten und die iterative Entwicklung des Systems ermöglichen. Gleichzeitig zeigt die Evaluation, dass moderne Architekturen wie PARA-Drive oder Adaptive AUTOSAR Mechanismen zur Parallelisierung und Optimierung der Verarbeitungskette nutzen, um die Performanceeinbußen modularer Architekturen zu minimieren. Diese Erkenntnisse verdeutlichen die Komplexität des Abwägens zwischen Performance und Entwicklungsmöglichkeiten, wobei die Testumgebung einen klaren Rahmen für die Evaluierung und Verbesserung solcher Architekturen bietet.

Die Evaluation bestätigt, dass die in der Einleitung definierten Ziele und Anforderungen durch die implementierte Testumgebung erfolgreich erfüllt werden. Gleichzeitig liefert sie eine fundierte Antwort auf die in Abschnitt 1.2 formulierte Forschungsfrage.

9.2. Ausblick

Die durchgeführte Evaluation zeigt die Zielerreichung und die Erfüllung der Anforderungen der entwickelten Testumgebung. Die definierten Anforderungen sind jedoch nicht allumfassend, da in dieser Arbeit nur eine prototypische Implementierung als „Proof of Concept“ gezeigt wird. Beispielsweise müssen ADS im realen Einsatz Echtzeitanforderungen erfüllen, um eine zuverlässige Steuerung in der Umgebung zu gewährleisten. Ohne die Erfüllung dieser Anforderungen kann die Sicherheit und Effektivität eines ADS im praktischen Betrieb erheblich beeinträchtigt werden. Insbesondere bei der Verwendung einer HiL-Umgebung sind korrekte Tests von der Echtzeitfähigkeit abhängig. Um diese Anforderungen zu unterstützen, müssten Änderungen am SUT, am Ethernet-Netzwerk und an der Testumgebung vorgenommen werden. Die im Konzept vorgestellte nachrichtenorientierte Middleware konnte mit dem Framework ROS2 realisiert werden. Diese Version basiert auf dem Protokoll DDS und bietet

damit die Möglichkeit, Echtzeitprogramme zu entwickeln. Um dies zu ermöglichen, muss das System verschiedene Voraussetzungen erfüllen, die in dieser Arbeit nicht betrachtet wurden. Als Betriebssystem für ein solches System muss ein Linux-Kernelpatch wie `PREEMPT_RT` oder ein Echtzeitbetriebssystem wie RTOS verwendet werden. Da die gesamte Testumgebung containerisiert ist, müssen die Basis-Images an den Patch `PREEMPT_RT` angepasst werden. Gleichzeitig muss die Implementierung der ROS-Anwendungen in C++ und nicht in Python erfolgen. Das verwendete Ethernet-Netzwerk zwischen der Testumgebung und dem SUT kann durch den Einsatz des Adaptermoduls durch ein Netzwerk mit TSN-Fähigkeiten ersetzt werden, welches bereits in Abschnitt 6.5 vorgestellt wurde. Die vorgestellte Testumgebung bietet durch die Modularität über die Middleware und die vollständige Containerisierung bereits Ansätze, um ein echtzeitfähiges System entwickeln zu können.

Die Fähigkeit der Testumgebung, das SUT mit unterschiedlichen Qualitätsstufen von Eingabedaten (Ground-Truth, Simulationsdaten, Sensormodelle und OTA-Stimulation) zu testen, lässt die Vergleichbarkeit dieser Eingabedaten offen. Um dies zu ermöglichen, sind standardisierte Methoden zur Qualifizierung und Kalibrierung dieser Daten erforderlich. Die Eingabedaten können sich nicht nur in der visuellen Qualität, sondern auch im FOV oder in der Brennweite unterscheiden, was zu einem unterschiedlichen Verhalten des SUT führen kann. Diese Abweichung kann bei der Verwendung des OTA-Sensors im Vergleich zu den Kamerarohdaten aus der Simulation in Unterabschnitt 8.2.1 beobachtet werden. In einer zukünftigen Arbeit sollte dieser Unterschied und der konkrete Einfluss auf die Testläufe weiter untersucht werden. Wie bereits im Konzept der Testumgebung beschrieben, können durch die Middleware und das einheitliche Datenformat OSI neue Sensoren hinzugefügt werden. Diese Möglichkeit wirft die Frage nach der Synchronisation der Sensordaten auf, welche besonders im Bereich der Sensor-Fusion von Bedeutung hat. Sollen beispielsweise Kamera- und Radardaten gleichzeitig oder zeitversetzt vom SUT empfangen werden? In der aktuellen Implementierung werden die Daten vom Adaptermodul in der Testumgebung an das SUT gesendet, sobald sie verfügbar sind. Eine mögliche Anforderung, dass die Daten synchronisiert gesendet werden, erfordert eine Änderung des Adaptermoduls und der verwendeten Simulation, da diese in einen synchronen Modus versetzt werden muss.

Neben der Synchronisation kann auch die bereits bestehende Möglichkeit der Dezentralisierung auf eine neue Stufe gehoben werden. Bisher ist es durch die Middleware und die Verwendung eines standardisierten Netzwerkprotokolls möglich, einzelne Komponenten der Testumgebung auf unterschiedlichen Geräten auszuführen. Dies wurde bereits in Abschnitt 8.2 bei der Verifikation einer Anforderung erfolgreich demonstriert. Wird dieser Gedanke der Dezentralisierung auf eine Verteilung auf verschiedene Standorte ausgeweitet, öffnet dies neue Türen für die Entwicklung und den Test von ADS. Dies könnte es Unternehmen ermöglichen, sich auf bestimmte Teile des SUT oder der Testumgebung zu konzentrieren, während das

System als Ganzes genutzt werden kann. Ein Beispiel hierfür wäre: Unternehmen A fokussiert sich auf ViL-Teststände mit Anbindung von Sensoren, Aktoren und einer Simulation dieser, Unternehmen B auf die V+V dieser und Unternehmen C auf die Entwicklung von ADS. Bei einer vollständigen Dezentralisierung können Tests durchgeführt werden, die diese Komponenten der verschiedenen Unternehmen kombinieren, ohne dass die entwickelte Hard- oder Software dem anderen Unternehmen zur Verfügung gestellt werden muss.

Literaturverzeichnis

Abhishek, Manish Kumar, D Rajeswara Rao und K Subrahmanyam. 2022. „Framework to deploy containers using kubernetes and ci/cd pipeline“. *International Journal of Advanced Computer Science and Applications* 13 (4). ISSN: 2156-5570. https://doi.org/10.14569/ijacsa.2022.0130460.

Acatech. 2012. „Cyber-Physical Systems: Innovationsmotoren für Mobilität, Gesundheit“. *Energie und Produktion (acatech POSITION).*

Antony, Maria Merin und Ruban Whenish. 2021. „Advanced Driver Assistance Systems (ADAS)“. In *Automotive Embedded Systems: Key Technologies, Innovations, and Applications,* 165–181. Cham: Springer International Publishing. ISBN: 978-3-030-59897-6. https://doi.org/10.1007/978-3-030-59897-6_9.

ASAM. 2024a. *Asam OpenDRIVE® 1.8.1.* https://publications.pages.asam.net/standards/ASAM%5C%5FOpenDRIVE/ASAM%5C%5FOpenDRIVE%5C%5FSpecification/latest/specification/index.html.

———. 2024b. *Asam OpenSCENARIO® 2.0.0.* https://www.asam.net/static%5C%5Fdownloads/public/asam-openscenario/2.0.0/welcome.html.

———. 2024c. *Asam OSI® (Open Simulation Interface) V3.7.0-rc2.* https://opensimulationinterface.github.io/osi-antora-generator/asamosi/latest/specification/index.html.

Baic, Drazen, P. Langjahr, W. Haas und A. Fessard. 2018. „Safe computing with central ECUs“. In *18. Internationales Stuttgarter Symposium,* 155–163. Wiesbaden: Springer Fachmedien Wiesbaden. ISBN: 978-3-658-21194-3. https://doi.org/10.1007/978-3-658-21194-3_14.

Baumann, Gerd. 2006. *Was verstehen wir unter Tests? Abstraktionsebenen, Begriffe und Definitionen.* https://www.yumpu.com/de/document/read/8390972/pdf-download-fkfs.

Behere, Sagar und Martin Törngren. 2015. „A functional architecture for autonomous driving“. In *Proceedings of the first international workshop on automotive software architecture,* 3–10.

Benckendorff, Tenny, Andreas Lapp, Thomas Oexner und Thomas Thiel. 2019. „Comparing current and future E/EArchitecture trends of commercial vehicles and passenger cars“. In *19. Internationales Stuttgarter Symposium,* 1190–1200. Wiesbaden: Springer Fachmedien Wiesbaden. ISBN: 978-3-658-25939-6. https://doi.org/10.1007/978-3-658-25939-6_95.

Berger, Christian, Björnborg Nguyen und Ola Benderius. 2017. „Containerized Development and Microservices for Self-Driving Vehicles: Experiences and Best Practices“. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW),* 7–12. IEEE, April. https://doi.org/10.1109/ICSAW.2017.56.

Betz, Tobias, Long Wen, Fengjunjie Pan, Gemb Kaljavesi, Alexander Zuepke, Andrea Bastoni, Marco Caccamo, Alois Knoll und Johannes Betz. 2024. „A Containerized Microservice Architecture for a ROS 2 Autonomous Driving Software: An End-to-End Latency Evaluation“. *arXiv preprint arXiv:2404.12683* (August): 57–66. ISSN: 1533-2306. https://doi.org/10.1109/rtcsa62462.2024.00018.

Blinowski, Grzegorz, Anna Ojdowska und Adam Przybyłek. 2022. „Monolithic vs. microservice architecture: A performance and scalability evaluation“. *IEEE Access* 10:20357–20374. ISSN: 2169-3536. https://doi.org/10.1109/access.2022.3152803.

Brade, Tino, Birte Kramer und Christian Neurohr. 2021. „Paradigms in scenario-based testing for automated driving“. In *2021 International Symposium on Electrical, Electronics and Information Engineering,* 108–114.

Brinkmann, Marius. 2018. „Physikalische Testfeld-Architektur für die Unterstützung der Entwicklung von automatisierten Schiffsführungssystemen“. Diss., Universität Oldenburg.

Burge, J., John Millar Carroll, Ray McCall und I. Mistrík. 2008. „Rationale and Requirements Engineering“. In *Rationale-Based Software Engineering,* 139–153. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-540-77583-6. https://doi.org/10.1007/978-3-540-77583-6_11.

Busch, Jean-Pierre, Lennart Reiher und Lutz Eckstein. 2024. „Enabling the deployment of any-scale robotic applications in microservice architectures through automated containerization“. In *2024 IEEE International Conference on Robotics and Automation (ICRA),* 17650–17656. IEEE. https://doi.org/10.1109/icra57147.2024.10611586.

Carlson, Alexandra, Katherine A Skinner, Ram Vasudevan und Matthew Johnson-Roberson. 2018. „Modeling camera effects to improve visual learning from synthetic data“. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshops,* 11129:505–520. Cornell University, März. https://doi.org/10.1007/978-3-030-11009-3_31.

Casini, Daniel, Paolo Pazzaglia, Alessandro Biondi und Marco Di Natale. 2022. „Optimized partitioning and priority assignment of real-time applications on heterogeneous platforms with hardware acceleration“. *Journal of Systems Architecture* 124 (Februar): 102416. ISSN: 1383-7621. https://doi.org/10.1016/j.sysarc.2022.102416.

Chen, Xiaozhi, Kaustav Kundu, Ziyu Zhang, Huimin Ma, Sanja Fidler und Raquel Urtasun. 2016. „Monocular 3d object detection for autonomous driving“. In *Proceedings of the IEEE conference on computer vision and pattern recognition,* 2147–2156. IEEE Computer Society, Juni. https://doi.org/10.1109/cvpr.2016.236.

Deng, Libing, Guoqi Xie, Hong Liu, Yunbo Han, Renfa Li und Keqin Li. 2022. „A survey of real-time ethernet modeling and design methodologies: From AVB to TSN“. *ACM Computing Surveys (CSUR)* 55, Nr. 2 (Januar): 1–36. ISSN: 0360-0300. https://doi.org/10.1145/3487330.

Ding, Shengxuan, Mohamed Abdel-Aty, Natalia Barbour, Dongdong Wang, Zijin Wang und Ou Zheng. 2024. „Exploratory analysis of injury severity under different levels of driving automation (SAE Level 2-5) using multi-source data“, https://arxiv.org/abs/2303.17788.

Dosovitskiy, Alexey, German Ros, Felipe Codevilla, Antonio Lopez und Vladlen Koltun. 2017. „CARLA: An Open Urban Driving Simulator“. In *Proceedings of the 1st Annual Conference on Robot Learning,* 78:1–16. Cornell University, November. https://doi.org/10.48550/arxiv.1711.03938.

dSPACE. 2024. „Blueprint for Testing Software: Thoughtful Verification and Validation“. Zugriff am 08. Februar 2025, *dSPACE Engineers' Insights,* https://www.dspace.com/en/inc/home/news/engineers-insights/thoughtful-verification-and-va/blueprint-for-testing-software.cfm.

Eigner, Martin, Daniil Roubanov und Radoslav Zafirov. 2014. *Modellbasierte virtuelle produktentwicklung.* 2014. Aufl. New York, NY: Springer, Januar. https://doi.org/10.1007/978-3-662-43816-9.

Farkas, Janos, Lucia Lo Bello und Craig Gunther. 2018. „Time-sensitive networking standards“. *IEEE Communications Standards Magazine* 2, Nr. 2 (Juni): 20–21. ISSN: 2471-2825. https://doi.org/10.1109/mcomstd.2018.8412457.

Feilhauer, Marius, Juergen Haering und Sean Wyatt. 2016. „Current approaches in HiL-based ADAS testing“. *SAE International Journal of Commercial Vehicles* 9, Nrn. 2016-01-8013 (September): 63–69. ISSN: 1946-391X.

Fremont, Daniel J, Edward Kim, Yash Vardhan Pant, Sanjit A Seshia, Atul Acharya, Xantha Bruso, Paul Wells, Steve Lemke, Qiang Lu und Shalin Mehta. 2020. „Formal scenario-based testing of autonomous vehicles: From simulation to the real world“. In *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC),* 1–8. IEEE, September. https://doi.org/10.1109/itsc45102.2020.9294368.

Frigerio, Alessandro, Bart Vermeulen und Kees GW Goossens. 2021. „Automotive architecture topologies: Analysis for safety-critical autonomous vehicle applications“. *IEEE Access* 9:62837–62846. ISSN: 2169-3536. https://doi.org/10.1109/access.2021.3074813.

Fürst, Simon und Markus Bechter. 2016. „AUTOSAR for connected and autonomous vehicles: The AUTOSAR adaptive platform“. In *2016 46th annual IEEE/IFIP international conference on Dependable Systems and Networks Workshop (DSN-W),* 215–217. IEEE Computer Society, Juni. https://doi.org/10.1109/dsn-w.2016.24.

Gamma, E., R. Helm, R. Johnson und J. Vlissides. 2015. *Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software.* mitp Professional. MITP. ISBN: 9783826699047.

Garikapati, Divya und Sneha Sudhir Shetiya. 2024. „Autonomous Vehicles: Evolution of Artificial Intelligence and the Current Industry Landscape“. *Big Data and Cognitive Computing* 8, Nr. 4 (April): 42. ISSN: 2504-2289. https://doi.org/10.3390/bdcc8040042.

Giaimo, Federico und Christian Berger. 2017. „Design criteria to architect continuous experimentation for self-driving vehicles“. In *2017 ieee international conference on software architecture (icsa),* 203–210. IEEE, April. https://doi.org/10.1109/icsa.2017.36.

Gliwa, Peter. 2021. „AUTOSAR“ [auf ger]. In *Embedded Software Timing,* 317–342. Germany: Springer Fachmedien Wiesbaden GmbH. ISBN: 3658264799. https://doi.org/10.1007/978-3-658-26480-2_10.

Gomaa, Hassan. 2011. *Software modeling and design: UML, use cases, patterns, and software architectures.* 49:49–0322. 01. Cambridge University Press, September. ISBN: 9781139494731. https://doi.org/10.5860/choice.49-0322.

Gruyer, Dominique, Valentin Magnier, Karima Hamdi, Laurène Claussmann, Olivier Orfila und Andry Rakotonirainy. 2017. „Perception, information processing and modeling: Critical stages for autonomous driving applications“. *Annual Reviews in Control* 44:323–341. ISSN: 1367-5788. https://doi.org/10.1016/j.arcontrol.2017.09.012.

Hagen, Lars. 2020. *Neue Möglichkeiten für die Motorsteuergeräte-Software durch Car-to-Cloud-Vernetzung.* 1–115. Springer. ISBN: 978-3-658-31565-8. https://doi.org/10.1007/978-3-658-31565-8.

Hahn, Axel. 2015. „Simulation Environment for Risk Assessment of E-Navigation Systems“. In *Volume 3: Structures, Safety and Reliability.* American Society of Mechanical Engineers, Mai. https://doi.org/10.1115/OMAE2015-41498.

Hahn, Axel und Thoralf Noack. 2016. „eMaritime Integrated Reference Platform“, https://www.dglr.de/publikationen/2016/420297.pdf.

Hahn, Axel, Sören Schweigert, Volker Gollücke und Carsten Buschmann. 2015. „Virtual test bed for maritime safety assessment“. *16h Marine Engineering Conference,* 116–122. https://yadda.icm.edu.pl/yadda/element/bwmeta1.element.baztech-81241a76-1979-43ea-8a20-a8c714f228ad.

Halstenberg, Jürgen, Bernd Pfitzinger und Thomas Jestädt. 2020. *DevOps.* Springer.

Harris, Chandler. 2017. *Microservices und monolithische Architektur im Vergleich.* Abgerufen am 21.01.2025. https://www.atlassian.com/de/microservices/microservices-architecture/microservices-vs-monolith.

Hassan, Mohamed. 2024. „Choosing the Right Communication Protocol for your Web Application“. *arXiv preprint arXiv:2409.07360* abs/2409.07360 (September). ISSN: 2331-8422. https://doi.org/10.48550/arxiv.2409.07360.

Hills, Laguna. 2020. *Recap of 2020 recalls reveals impact of pandemic on compliance and the continuing threat.* https://www.recallmasters.com/sor/. Zugriff am 17. Januar 2025.

Hong, Dongwon und Changjoo Moon. 2024. „Autonomous Driving System Architecture with Integrated ROS2 and Adaptive AUTOSAR“. *Electronics* 13, Nr. 7 (März): 1303. ISSN: 2079-9292. https://doi.org/10.3390/electronics13071303.

IEEE. 1990. *IEEE Standard Glossary of Software Engineering Terminology.* 610.12-1990. Accessed: 2025-01-20. New York, USA: IEEE Standards Association, September. https://doi.org/10.1109/ieeestd.1990.101064. https://standards.ieee.org/.

———. 1998. *IEEE Recommended Practice for Software Requirements Specifications.* IEEE, Juni. https://doi.org/10.1109/IEEESTD.1998.88286.

———. 2010. *IEEE Standard for Modeling and Simulation (M and S) High Level Architecture (HLA)– Framework and Rules.* IEEE Std 1516-2010. Accessed: 2025-01-23. IEEE Standards Association. https://standards.ieee.org/ieee/1516/3744/.

———. 2020. *IEEE Standard for Ethernet - Physical Layer Specifications and Management Parameters for Greater than 1 Gb/s Automotive Ethernet.* IEEE Std 802.3ch-2020. Accessed: 2025-01-23. IEEE Standards Association. https://standards.ieee.org/ieee/802.3ch/7058/.

Isermann, R. 2016. *Fahrerassistenzsysteme 2016: Von der Assistenz zum automatisierten Fahren 2. Internationale ATZ-Fachtagung.* Proceedings. Springer Fachmedien Wiesbaden. ISBN: 9783658214449.

ISO. 2018. *Road vehicles – Functional safety – Part 1: Vocabulary.* ISO 26262-1:2018. Geneva, Switzerland: International Organization for Standardization (ISO). https://www.iso.org/standard/68383.html.

Jaikamal, Vivek. 2009. *Model-based ecu development–an integrated mil-sil-hil approach.* Technischer Bericht. SAE Technical Paper, April.

Al-Jaroodi, Jameela, Nader Mohamed, Imad Jawhar und Sanja Lazarova-Molnar. 2016. „Software engineering issues for cyber-physical systems“. In *2016 IEEE International Conference on Smart Computing (SMARTCOMP),* 1–6. IEEE Computer Society, Mai. https://doi.org/10.1109/smartcomp.2016.7501717.

Johansson, Axel und Simon Paulsson. 2024. „Microservice integration testing with hardware-in-the-loop in CI/CD pipelines“. Master's thesis, Chalmers University of Technology.

Jüstel, Benjamin und Ulrich Stöckmann. 2018. „Key component dynamic motion controller–longitudinal and lateral vehicle control for automated driving functions“. In *Fahrerassistenzsysteme 2016: Von der Assistenz zum automatisierten Fahren 2. Internationale ATZ-Fachtagung,* 63–69. Springer. https://doi.org/10.1007/978-3-658-21444-9_5.

Kallweit, Roland, Uwe Gropengießer, Jörn Männel und Rajanpreet Singh. 2020. „Safe and Robust Function Development for Urban Autonomous Driving Based on Agile Methodology and DevOps“. In *Automatisiertes Fahren 2020: Von der Fahrerassistenz zum autonomen Fahren 6. Internationale ATZ-Fachtagung,* 1–9. Springer. https://doi.org/10.1007/978-3-658-34752-9_1.

Kang, Min-Su, Jae-Hoon Ahn, Ji-Ung Im und Jong-Hoon Won. 2022. „Lidar- and V2X-Based Cooperative Localization Technique for Autonomous Driving in a GNSS-Denied Environment“. *Remote Sensing* 14, Nr. 22 (November): 5881. ISSN: 2072-4292. https://doi.org/10.3390/rs14225881.

Koopman, Philip und Michael Wagner. 2016. „Challenges in autonomous vehicle testing and validation“. *SAE International Journal of Transportation Safety* 4, Nr. 1 (April): 15–24. ISSN: 2327-5626.

Kulak, D. und E. Guiney. 2012. *Use Cases: Requirements in Context.* Pearson Education. ISBN: 9780133085150.

Lo Bello, Lucia, Gaetano Patti und Luca Leonardi. 2023. „A perspective on ethernet in automotive communications—Current status and future trends“. *Applied Sciences* 13, Nr. 3 (Januar): 1278. ISSN: 2076-3417. https://doi.org/10.3390/app13031278.

Lotz, Jannik, Andreas Vogelsang, Ola Benderius und Christian Berger. 2019. „Microservice Architectures for Advanced Driver Assistance Systems: A Case-Study“. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C),* 45–52. IEEE, März. https://doi.org/10.1109/ICSA-C.2019.00016.

Lou, Guannan, Yao Deng, Xi Zheng, Mengshi Zhang und Tianyi Zhang. 2022. „Testing of autonomous driving systems: where are we and where should we go?“ In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering,* 31–43. ACM, November. https://doi.org/10.1145/3540250.3549111.

Lu, Sidi und Weisong Shi. 2024. *Vehicle Computing: From Traditional Transportation to Computing on Wheels.* 1–242. Springer Nature Switzerland. ISBN: 9783031599637. https://doi.org/10.1007/978-3-031-59963-7.

Lunkeit, Armin und Wolf Zimmer. 2021. *Security by Design: Security Engineering informationstechnischer Systeme.* Springer.

Macenski, Steven, Tully Foote, Brian Gerkey, Chris Lalancette und William Woodall. 2022. „Robot Operating System 2: Design, architecture, and uses in the wild“. *Science Robotics* 7, Nr. 66 (Mai): eabm6074. ISSN: 2470-9476. https://doi.org/10.1126/scirobotics.abm6074.

Magosi, Zoltan Ferenc, Hexuan Li, Philipp Rosenberger, Li Wan und Arno Eichberger. 2022. „A Survey on Modelling of Automotive Radar Sensors for Virtual Test and Validation of Automated Driving“. *Sensors* 22, Nr. 15 (Juli): 5693. ISSN: 1424-8220. https://doi.org/10.3390/s22155693.

Marko, Nadja, Jonas Ruebsam, Andreas Biehn und Hannes Schneider. 2019. „Scenario-based Testing of ADAS-Integration of the Open Simulation Interface into Co-simulation for Function Validation.“ In *SIMULTECH,* 255–262. SCITEPRESS - Science / Technology Publications, Juli. https://doi.org/10.5220/0007838302550262.

Maurer, Markus, J Christian Gerdes, Barbara Lenz und Hermann Winner. 2015. *Autonomes Fahren: technische, rechtliche und gesellschaftliche Aspekte.* Springer Nature.

McKinsey. 2022. *Autonomes Fahren: Die Zukunft der Mobilität gestalten.* Zugriff am 08. Februar 2025. https://www.mckinsey.de/publikationen/2022-01-03-autonomes-fahren.

Mina, J, Z Flores, E López, A Pérez und J-H Calleja. 2016. „Processor-in-the-loop and hardware-in-the-loop simulation of electric systems based in FPGA“. In *2016 13th International Conference on Power Electronics (CIEP),* 172–177. IEEE.

Modrakowski, Elias, Niklas Rahenbrock, Eike Möhlmann und Henning Schlender. 2024. „Small scale, big impact: experiences from a miniature ViL testbed and digital twin development“. In *International Symposium on Leveraging Applications of Formal Methods,* 15223:83–106. Springer Science+Business Media, Oktober. https://doi.org/10.1007/978-3-031-75390-9_6.

Mustyala, Anirudh. 2022. „CI/CD Pipelines in Kubernetes: Accelerating Software Development and Deployment“. *EPH-International Journal of Science And Engineering* 8 (3): 1–11.

Ness, A.J. 2013. *Eine Systemarchitektur für die Gestaltung und das Management verteilter Informationssysteme.* 1–203. IPA-IAO - Forschung und Praxis. Springer Berlin Heidelberg. ISBN: 9783662068472. https://doi.org/10.1007/978-3-662-06847-2.

Newman, S. und T. Demmig. 2020. *Vom Monolithen zu Microservices: Patterns, um bestehende Systeme Schritt für Schritt umzugestalten.* O'Reilly. ISBN: 9783960104247.

Nguyen, Trung-Hieu, Truong-Giang Vuong, Hong-Nam Duong, Son Nguyen, Hieu Dinh Vo, Toshiaki Aoki und Thu-Trang Nguyen. 2024. „Generating Critical Scenarios for Testing Automated Driving Systems". *arXiv preprint arXiv:2412.02574* abs/2412.02574 (Dezember). ISSN: 2331-8422. https://doi.org/10.48550/arxiv.2412.02574.

Nickovic, Dejan, Wolfgang Herzner, Eike Möhlmann, Sebastian Gerwinn, Gustavo García Padilla, Fabian Diegmann, Sadri Hakki und Colin Snook. 2017. *D3.2.2 v1 V&V Methodology.* Technischer Bericht. ENABLE-S3 - D3.2.2 v1 V&V Methodology. ENABLE-S3. https://ec.europa.eu/research/participants/documents/downloadPublic?documentIds=080166e5b2068457&appId=PPGMS.

Nouri, Ali, Christian Berger und Fredrik Törner. 2022. „An Industrial Experience Report about Challenges from Continuous Monitoring, Improvement, and Deployment for Autonomous Driving Features". In *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA),* 358–365. IEEE, August. https://doi.org/10.1109/seaa56994.2022.00063.

Oddi, Angelo, Riccardo Rasconi, Vieri Giuliano Santucci, Gabriele Sartor, Emilio Cartoni, Francesco Mannella und Gianluca Baldassarre. 2020. „Integrating open-ended learning in the sense-plan-act robot control paradigm". In *ECAI 2020,* 325:2417–2424. IOS Press. https://doi.org/10.3233/faia200373.

Oruganti, Pradeep Sharma, Matt Appel und Qadeer Ahmed. 2019. „Hardware-in-loop based Automotive Embedded Systems Cybersecurity Evaluation Testbed". In *Proceedings of the ACM Workshop on Automotive Cybersecurity, AutoSec@CODASPY 2019, Richardson, TX, USA, March 27, 2019,* 41–44. AutoSec '19. Richardson, Texas, USA: Association for Computing Machinery, März. ISBN: 9781450361804. https://doi.org/10.1145/3309171.3309173.

Österle, H., R. Riehm und P. Vogler. 2013. *Middleware: Grundlagen, Produkte und Anwendungsbeispiele für die Integration heterogener Welten.* Vieweg+Teubner Verlag. ISBN: 9783322872449.

Paul, Sumit, Danh Lephuoc und Manfred Hauswirth. 2024. „Performance Evaluation of ROS2-DDS middleware implementations facilitating Cooperative Driving in Autonomous Vehicle". *arXiv preprint arXiv:2412.07485* abs/2412.07485 (Dezember). ISSN: 2331-8422. https://doi.org/10.48550/arxiv.2412.07485.

Pham, Tri Minh Triet, Bo Yang und Jinqiu Yang. 2024. „Perception-Guided Fuzzing for Simulated Scenario-Based Testing of Autonomous Driving Systems“. *arXiv preprint arXiv:2408.13686* abs/2408.13686 (August). ISSN: 2331-8422. https://doi.org/10.48550/arxiv.2408.13686.

Qian, Rui, Xin Lai und Xirong Li. 2022. „3D object detection for autonomous driving: A survey“. *Pattern Recognition* 130 (Mai): 108796. ISSN: 0031-3203. https://doi.org/10.1016/j.patcog.2022.108796.

Reway, Fabio, Werner Huber und Eduardo Ribeiro. 2018. „Test Methodology for Vision-Based ADAS Algorithms with an Automotive Camera-in-the-Loop“. In *2018 IEEE International Conference on Vehicular Electronics and Safety, ICVES 2018, Madrid, Spain, September 12-14, 2018,* 1–7. IEEE, September. https://doi.org/10.1109/ICVES.2018.8519598.

Riedmaier, Stefan, Jonas Nesensohn, Christian Gutenkunst, Bernhard Schick, Tobias Düser und Houssem Abdellatif. 2018. „Validation of x-in-the-loop approaches for virtual homologation of automated driving functions“. In *11. Grazer Symposium VIRTUAL VEHICLE (GSVF) Graz, 15./16.05. 2018.* https://mediatum.ub.tum.de/1546093.

Ruehl, Martin und Fabian Bronner. 2024. „Creation and Usage of Virtual Control Units (V-ECUs) in SIL and HIL for Development and Validation especially for Software-Defined-Vehicles (SDVs)“. In *2024 Stuttgart International Symposium on Automotive and Engine Technology,* 371–383. Wiesbaden: Springer Fachmedien Wiesbaden. ISBN: 978-3-658-45018-2. https://doi.org/10.1007/978-3-658-45018-2_27.

Rüssmeier, N, A Lamm und A Hahn. 2019. „A generic testbed for simulation and physical-based testing of maritime cyber-physical system of systems“. In *Journal of Physics: Conference Series,* 1357:012025. 1. IOP Publishing, Oktober. https://doi.org/10.1088/1742-6596/1357/1/012025.

Ruthardt, Jona und Thomas Michalke. 2022. „A System-driven Automatic Ground Truth Generation Method for DL Inner-City Driving Corridor Detectors“. In *2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC),* 2120–2127. IEEE, Oktober. https://doi.org/10.1109/itsc55140.2022.9921773.

SAE. 2021. *SAE J3016.* https://www.sae.org/standards/content/j3016%5C%5F202104/.

Schäuffele, J. und T. Zurawka. 2024. *Automotive Software Engineering: Grundlagen, Prozesse, Methoden und Werkzeuge.* ATZ/MTZ-Fachbuch. Springer Fachmedien Wiesbaden. ISBN: 9783658435431.

Schlager, Birgit, Stefan Muckenhuber, Simon Schmidt, Hannes Holzer, Relindis Rott, Franz Michael Maier, Kmeid Saad et al. 2020. „State-of-the-Art Sensor Models for Virtual Testing of Advanced Driver Assistance Systems/Autonomous Driving Functions“. *SAE International Journal of Connected and Automated Vehicles* 3, Nr. 3 (Oktober): 233–261. ISSN: 2574-075X. https://doi.org/10.4271/12-03-03-0018.

Schleicher, Martin und Sorin Mihai Grigorescu. 2020. „Wie neuronale Netze die Entwicklung von Automobilsoftware verändern“. *ATZelektronik* 15, Nr. 1 (Januar): 26–34. ISSN: 1862-1791. https://doi.org/10.1007/s35658-019-0151-0.

Siemens. 2024. „Powering Software-Defined Vehicle Insights Through Verification and Validation“. Zugriff am 17. Januar 2025, *Siemens Automotive Blog,* https://blogs.sw.siemens.com/automotive-transportation/2024/04/18/powering-software-defined-vehicle-insights-through-verification-and-validation/.

Sievers, Gregor, Caius Seiger, Michael Peperhowe, Holger Krumm, Sebastian Graf und H Hanselmann. 2018. „Driving simulation technologies for sensor simulation in sil and hil environments“. In *Proceedings of the DSC.*

Silva, Iron Prando da und Dr.-Ing. Pablo Oliveira Antonino. 2024. *Der FMI-Standard und seine Bedeutung für die Industrie.* Zugriff am 15.02.2025. https://www.iese.fraunhofer.de/blog/der-fmi-standard/.

Slama, Dirk, Achim Nonnenmacher und Thomas Irawan. 2023. *The software-defined vehicle: A digital-first approach to creating next-generation experiences.*

Srivastava, Ankit. 2019. „Sense-Plan-Act in Robotic Applications“, https://doi.org/10.13140/RG.2.2.21308.36481.

Stellwagen, J. 2024. *Hybride Direktkommunikation als Unterstützung für sicherheitskritische Anwendungsfälle der Vehicle-to-X Kommmunikation.* ISBN: 9783832581695.

Streichert, Thilo und Matthias Traub. 2012. *Elektrik/Elektronik-Architekturen im Kraftfahrzeug: Modellierung und Bewertung von Echtzeitsystemen.* Springer-Verlag.

Su, Ruoyu und Xiaozhou Li. 2024. *Modular Monolith: Is This the Trend in Software Architecture?,* April. https://doi.org/10.48550/ARXIV.2401.11867.

Vayghan, Leila Abdollahi, Mohamed Aymen Saied, Maria Toeroe und Ferhat Khendek. 2019. „Microservice based architecture: Towards high-availability for stateful applications with kubernetes“. In *2019 IEEE 19th international conference on software quality, reliability and security (QRS),* 176–185. IEEE, Juli. https://doi.org/10.1109/qrs.2019.00034.

Verein Deutscher Ingenieure (VDI). 2021. *VDI 2206: Entwicklung mechatronischer und cyber-physischer Systeme.* Erhältlich über den Beuth Verlag. Düsseldorf: Verein Deutscher Ingenieure (VDI), November. https://www.vdi.de/richtlinien/programme-zu-vdi-richtlinien/vdi-2206.

Villari, Massimo, Antonio Celesti, Giuseppe Tricomi, Antonino Galletta und Maria Fazio. 2017. „Deployment orchestration of microservices with geographical constraints for edge computing". In *2017 IEEE Symposium on Computers and Communications (ISCC),* 633–638. IEEE Computer Society, Juli. https://doi.org/10.1109/iscc.2017.8024599.

Vohra, Deepak. 2016. *Kubernetes microservices with Docker.* Apress. https://doi.org/10.1007/978-1-4842-1907-2.

Ward, Robert, Lee Alexander, Barrie Greenslade und Anthony Pharaoh. 2008. „IHO S-100: The new hydrographic geospatial standard for marine data and information".

Weng, Xinshuo, Boris Ivanovic, Yan Wang, Yue Wang und Marco Pavone. 2024. „PARA-Drive: Parallelized Architecture for Real-time Autonomous Driving". In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition,* 15449–15458. IEEE, Juni. https://doi.org/10.1109/cvpr52733.2024.01463.

Wolf, Fabian. 2018. *Fahrzeuginformatik.* Springer. https://doi.org/10.1007/978-3-658-21224-7.

Yeong, De Jong, Gustavo Velasco-Hernandez, John Barry und Joseph Walsh. 2021. „Sensor and Sensor Fusion Technology in Autonomous Vehicles: A Review". *Sensors* 21, Nr. 6 (März): 2140. ISSN: 1424-8220. https://doi.org/10.3390/s21062140.

Zinner, Helge. 2020. „Automotive Ethernet and SerDes in competition". *ATZelectronics worldwide* 15, Nr. 7 (Juli): 40–43. ISSN: 2524-8804. https://doi.org/10.1007/s38314-020-0232-0.

A. Vollständige Architektur der Testumgebung und des zu testenden Systems

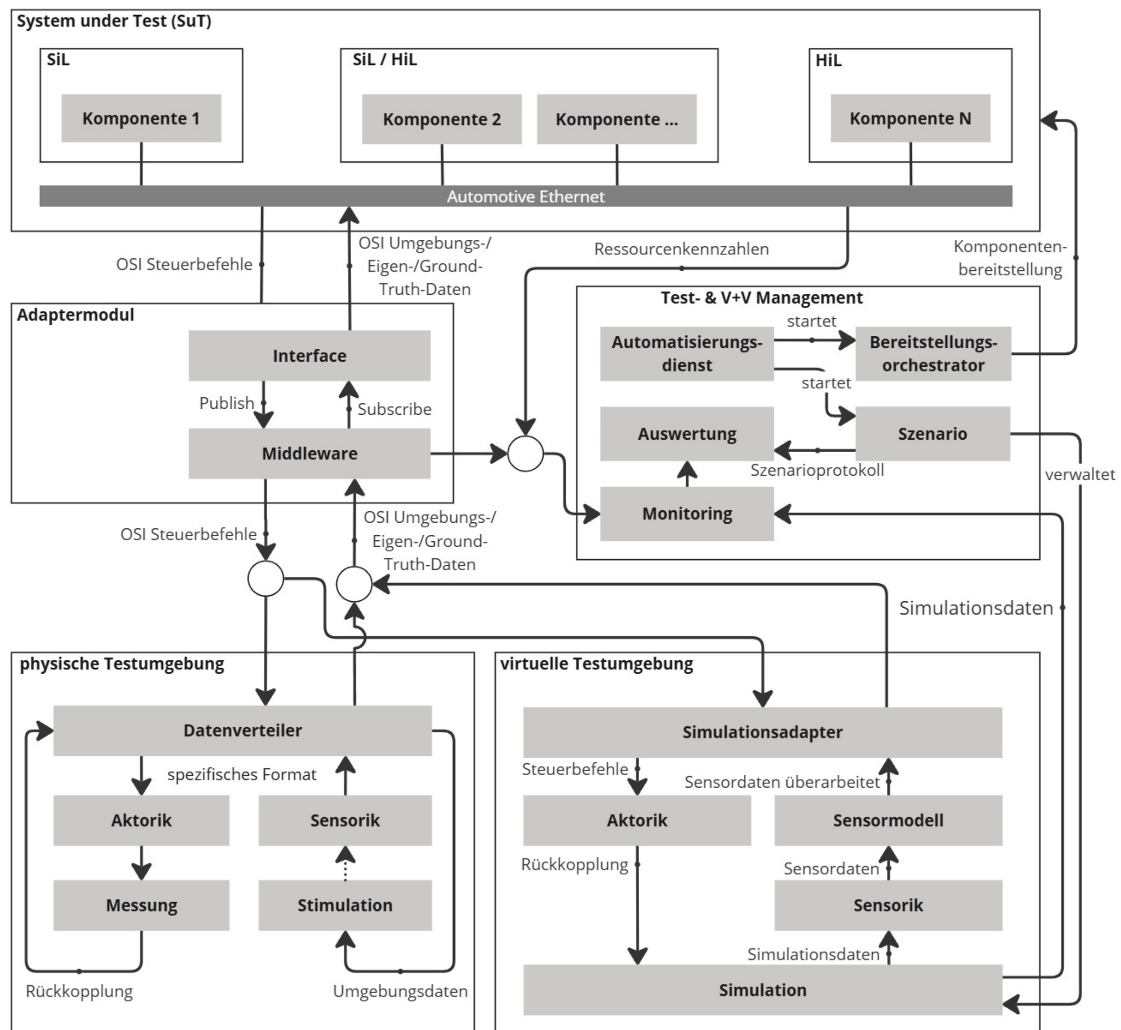


Abbildung A.1.: Vollständige Systemarchitektur der Testumgebung

B. Sequenzdiagramm für Use-Case 2A und 2B

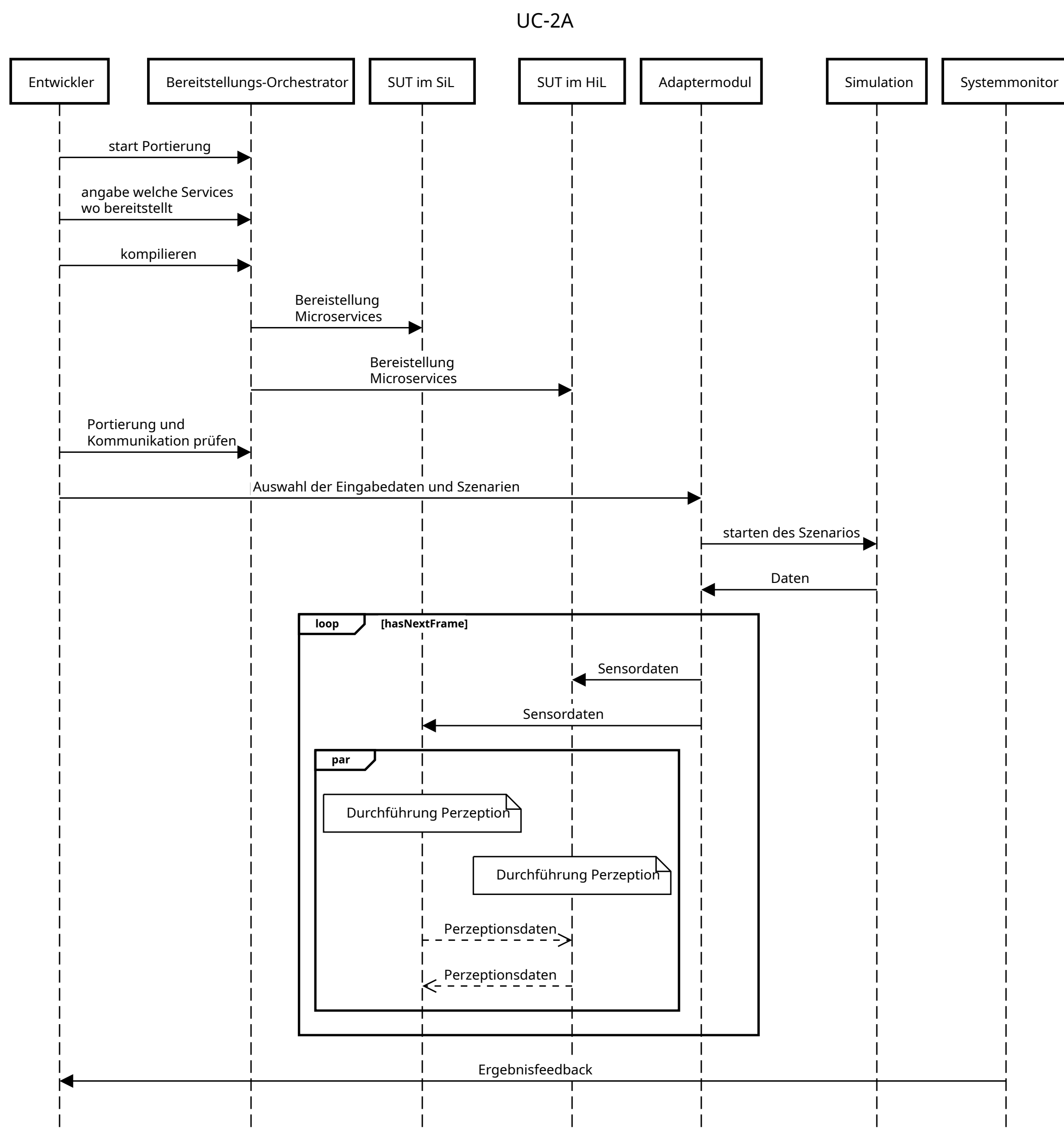


Abbildung B.1.: Sequenzdiagramm für UC-2A

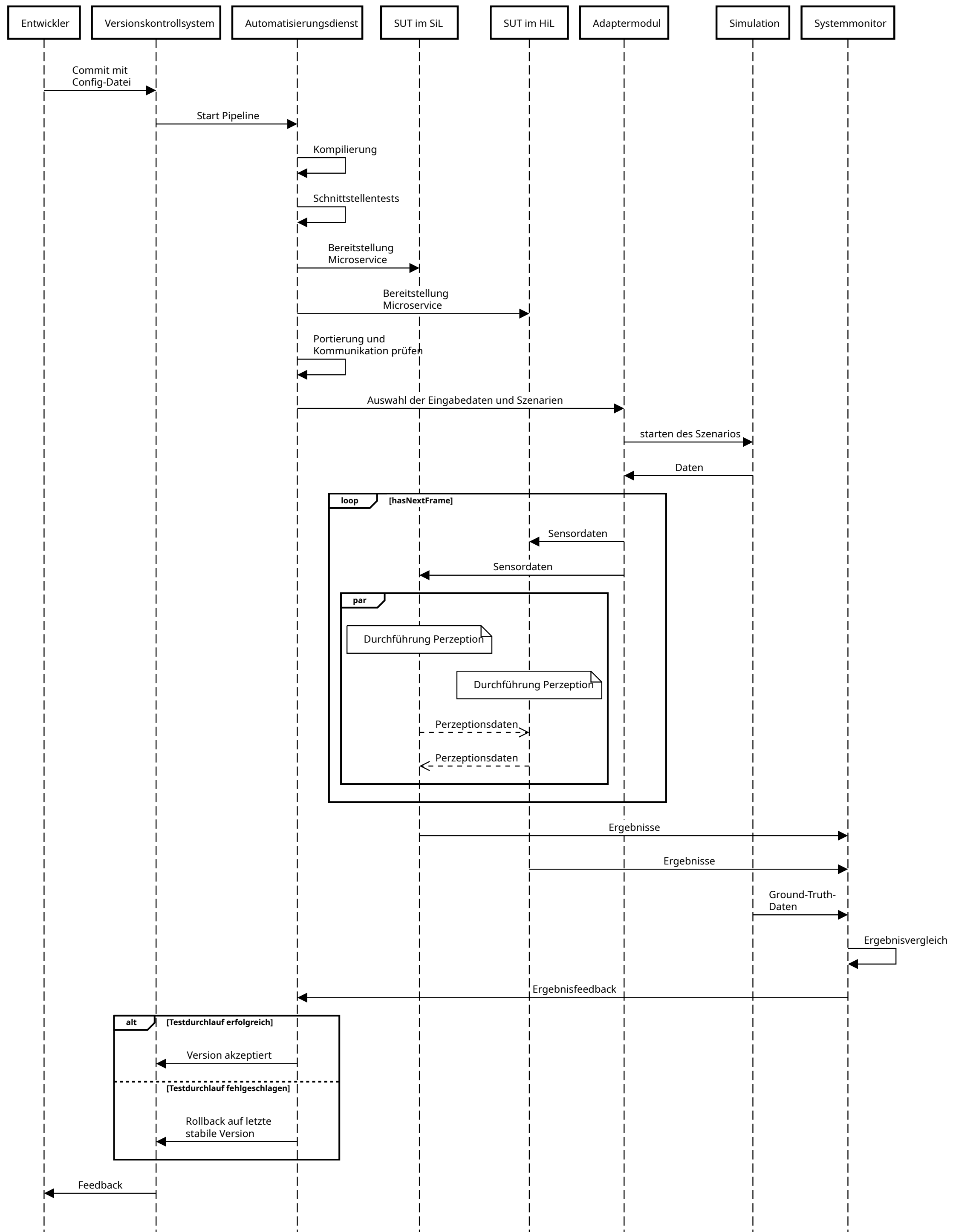


Abbildung B.2.: Sequenzdiagramm für UC-2B

C. Verarbeitungskette der Testumgebung von Closed-Loop-Tests

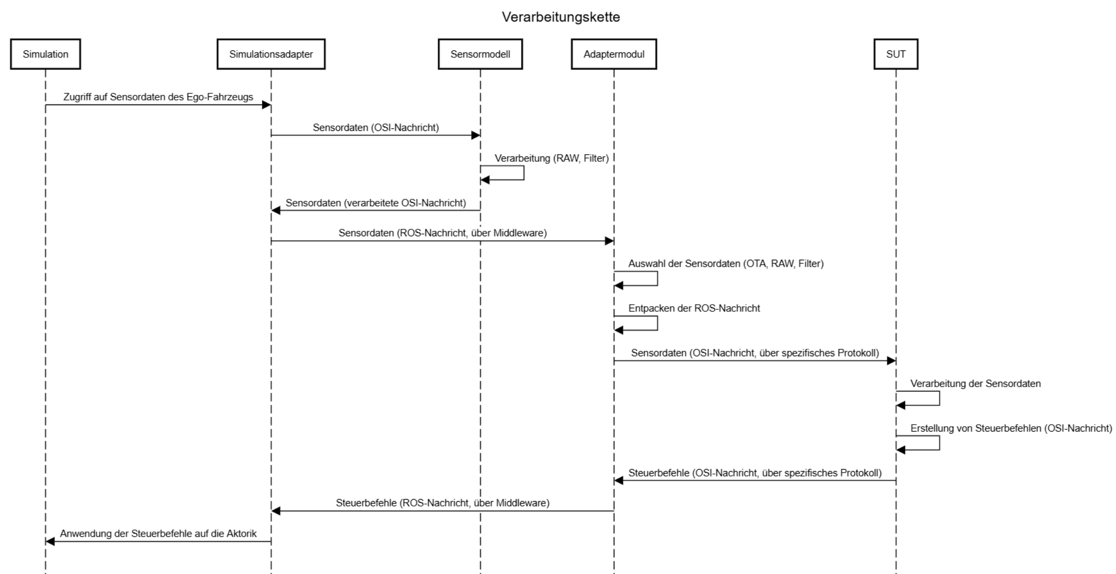


Abbildung C.1.: Verarbeitungskette der Testumgebung von Closed-Loop-Tests, zwischen der Simulation, dem Simulationsadapter, dem Sensormodell, dem Adaptermodul und dem SUT

D. Anwendung der Referenzarchitektur auf einen Highway Piloten

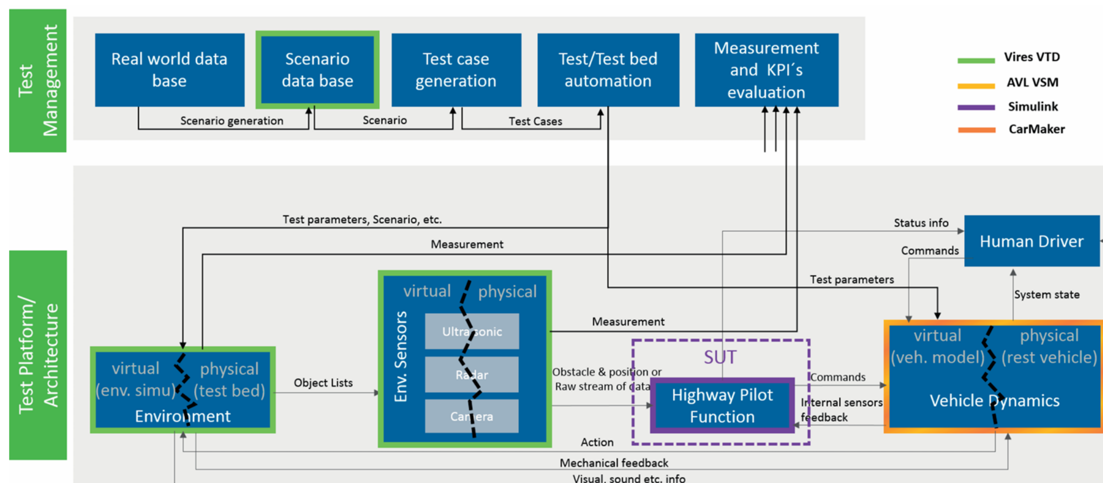


Abbildung D.1.: Anwendung der Referenzarchitektur auf einen Highway Piloten. Die schwarzen gestrichelten Linien zeigen, dass die verschiedene Komponenten der Testumgebung virtuell oder physisch existieren können (Nickovic et al. 2017)

E. Definition der CI/CD-Pipeline über eine gitlab-ci.yml

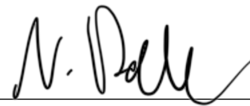
```
stages:
- build-sut
- deploy-sut
- deploy-test-env
- validate
build-sut:
  image: alpine:latest
  stage: build-sut
  script:
  - chmod og= $ID_RSA
  - apk update && apk add openssh-client rsync
  - ssh -i $ID_RSA [...] $SERVER_USER@$SERVER_IP "mkdir -p ~/repo"
  - rsync -avz --delete -e "ssh -i $ID_RSA [...]" ./ $SERVER_USER@$SERVER_IP:~/repo/
  - echo "Building SUT"
  - ssh -i $ID_RSA [...] $SERVER_USER@$SERVER_IP "cd ~/repo/system_under_test
    && docker buildx build --platform linux/amd64,linux/arm64 -t lane-follower:latest
    -f docker/Dockerfile ."
  - ssh -i $ID_RSA [...] $SERVER_USER@$SERVER_IP "docker tag
    lane-follower:latest localhost:5000/lane-follower:latest"
  - ssh -i $ID_RSA [...] $SERVER_USER@$SERVER_IP "docker push
    localhost:5000/lane-follower:latest"
  only:
  - run_test
deploy-sut:
  image: alpine:latest
  stage: deploy-sut
...
```

```
...
  script:
  - chmod og= $ID_RSA
  - apk update && apk add openssh-client rsync
  - ssh -i $ID_RSA [...] $SERVER_USER@$SERVER_IP "mkdir -p ~/repo"
  - rsync -avz --delete -e "ssh -i $ID_RSA [...]" ./ $SERVER_USER@$SERVER_IP:~/repo/
  - echo "Starting SUT"
  - ssh -i $ID_RSA [...] $SERVER_USER@$SERVER_IP "cd ~/repo
    && chmod +x start_sut.sh && ./start_sut.sh"
  only:
  - run_test
deploy-test-env:
  image: alpine:latest
  stage: deploy-test-env
  script:
  - chmod og= $ID_RSA
  - apk update && apk add openssh-client rsync
  - ssh -i $ID_RSA [...] $SERVER_USER@$SERVER_IP "mkdir -p ~/repo"
  - rsync -avz --delete -e "ssh -i $ID_RSA [...]" ./ $SERVER_USER@$SERVER_IP:~/repo/
  - echo "Reading test-config-file and save them as env variables"
  - ssh -i $ID_RSA [...] $SERVER_USER@$SERVER_IP "cd ~/repo
    && chmod +x read_test_config.sh && source ./read_test_config.sh
    && chmod +x auto_test_run.sh && ./auto_test_run.sh"
  - echo "Fetching log for validation stage"
  - ssh -i $ID_RSA [...] $SERVER_USER@$SERVER_IP "cd ~/repo
    && chmod +x show_latest_scenario_log.sh
    && ./show_latest_scenario_log.sh" > scenario_log.json
  - echo "Log fetched and saved for validation stage"
  artifacts:
    paths:
    - scenario_log.json
  only:
  - run_test
validate:
  image: alpine:latest
  stage: validate
  script:
  - apk update && apk add jq
  - echo "Contents of scenario_log.json:"
  - cat scenario_log.json
  - echo "Validating results from scenario_log.json"
  - jq -e '.criteria | all(.success == true)' scenario_log.json ||
  { echo "Validation failed! Not all criteria succeeded."; exit 1; }
  only:
  - run_test
```

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

Oldenburg, den 24. Februar 2025



Niklas Rahenbrock