Aufbau des Hardware-Modells einer E/E-Architektur und dessen Transformation in ein Simulationsmodell

(Platzhalter)

(in Bearbeitung)

Zusammenfassung

Diese Arbeit adressiert die modellbasierte Entwicklung moderner E/E-Architekturen für autonomes Fahren. Sie stellt einen durchgängigen Ansatz vor, der (i) ein erweiterbares Hardware-Metamodell, (ii) eine Synthese-Metrik zur Ableitung von Timing-, Bandbreiten-, Last- und Verfügbarkeitsparametern und (iii) eine regelbasierte Transformation in ausführbare Simulationsmodelle umfasst. Anhand von Szenarien und Varianten werden Ende-zu-Ende-Latenzen, Netzwerkauslastungen, Rechenlasten und Verfügbarkeiten untersucht und mit analytischen Bounds plausibilisiert.

Schlagwörter: E/E-Architektur, autonome Systeme, PREEvision, TSN, Simulation, FMI, OMNeT++

Abstract

This dissertation proposes a model-based workflow for modern automotive E/E architectures in automated driving. It contributes an extensible hardware meta-model, a synthesis metric that derives timing, bandwidth, load and availability parameters, and a rule-based transformation into executable simulation models. Scenarios and design variants demonstrate how end-to-end latency, network utilization, compute load and availability can be analyzed early and validated against analytical bounds.

Keywords: E/E architecture, automated driving, PREEvision, TSN, simulation, FMI, OMNeT++

Danksagung

Ich danke allen Betreuerinnen und Betreuern, Kolleginnen und Kollegen sowie meiner Familie für die Unterstützung während der Entstehung dieser Arbeit. Besonderer Dank gilt M. Sc. Kevin Neubauer für wertvolle Anregungen und das konstruktive Feedback.

Abkürzungsverzeichnis

AD Automated Driving
ADAS Advanced Driver Assistance Systems
AD-DC Automated Driving Domain Controller
API Application Programming Interface
ASIL Automotive Safety Integrity Level
AUTOSAR AUTomotive Open System ARchitecture
BCET Best-Case Execution Time
CAN Controller Area Network
CBS Credit-Based Shaping
CC Central Compute
CI/CD Continuous Integration / Continuous Deployment
CNN Convolutional Neural Network
CPU Central Processing Unit
DDS Data Distribution Service
DTDL Digital Twins Definition Language
E2E Ende-zu-Ende
ECU Electronic Control Unit
EDF Earliest Deadline First
EHB Electro-Hydraulic Brake
EMB Electro-Mechanical Brake
EPS Electric Power Steering
FMI Functional Mock-up Interface
FMU Functional Mock-up Unit
FPS Fixed-Priority Scheduling
GPU Graphics Processing Unit
gPTP generalized Precision Time Protocol
HSR High-availability Seamless Redundancy
HV High Voltage
IM Intermediate Model
KPI Key Performance Indicator
LiDAR Light Detection and Ranging

LIN Local Interconnect Network
MB.OS Mercedes-Benz Operating System
ML Machine Learning
MTBF Mean Time Between Failures
MTTR Mean Time To Repair
NPU Neural Processing Unit
OTA Over-The-Air
PRP Parallel Redundancy Protocol
QoS Quality of Service
RAM Random Access Memory
ROI Region of Interest
ROS Robot Operating System
SBOM Software Bill of Materials
SDV Software-Defined Vehicle
SOME/IP Scalable service-Oriented MiddlewarE over IP
SWC Software Component
TAS Time-Aware Shaping
TCU Telematics Control Unit
TOPS Tera Operations Per Second
TSN Time-Sensitive Networking
UDS Unified Diagnostic Services
V2G Vehicle-to-Grid
V2X Vehicle-to-Everything
VAN.EA Van Electric Architecture
WCET Worst-Case Execution Time
WCRT Worst-Case Response Time
ZC Zonen-Controller

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung und Motivation

Die Entwicklung moderner Fahrzeuge ist durch eine zunehmende Komplexität der Elektrik/Elektronik-Architekturen (E/E-Architekturen) gekennzeichnet. Autonomes Fahren führt zu einer rapiden Zunahme der funktionalen, zeitlichen und infrastrukturellen Komplexität in Fahrzeug-E/E-Architekturen [?, ?]. Moderne Fahrzeuge integrieren hunderte von elektronischen Steuergeräten (ECUs), komplexe Sensornetzwerke, leistungsfähige Rechenplattformen und hochentwickelte Kommunikationsnetzwerke [?, ?].

Diese Arbeit adressiert die frühzeitige Evaluierung des Zusammenspiels verteilter Funktionen über eine in die Architekturentwicklungsumgebung integrierte Simulation. Ziel ist ein erweiterbares Hardware-Modell, eine belastbare Synthese-Metrik und eine deterministische Transformation in ein ausführbares Simulationsmodell, das an realitätsnahen Szenarien überprüft wird. Die Notwendigkeit für solche Ansätze wird durch die wachsende Komplexität moderner E/E-Architekturen und die Anforderungen an funktionale Sicherheit nach ISO 26262 [?] deutlich [?].

Zentrale Beiträge sind: (i) eine domänenspezifische Komponententaxonomie und ein erweiterbares Metamodell, (ii) eine methodische Erweiterung der Synthese-Metrik zur quantitativen Ableitung von Ressourcen-, Timing- und Verfügbarkeitsparametern, (iii) ein regelbasiertes Transformationsframework in ein Simulationsziel (z. B. OMNeT++/INET, NS-3, FMI-Co-Sim) sowie (iv) eine systematische Evaluation entlang repräsentativer Use-Cases inklusive Fehlerszenarien. Diese Arbeit baut auf etablierten Methoden der modellbasierten Entwicklung [?, ?] und modernen Ansätzen zur Netzwerksimulation [?] auf.

Forschungsfragen: Welche Modellattribute sind erforderlich, um E2E-Latenzen, Jitter, Auslastungen und Verfügbarkeit früh belastbar abzuschätzen? Wie müssen Architektur- und Kommunikationsdetails abgebildet werden, um die Simulation als Designentscheidungswerkzeug nutzbar zu machen? Wie überführt man heterogene Architekturdaten konsistent in Simulationskonfigurationen? Diese Fragen werden vor dem Hintergrund aktueller Entwicklungen in der Fahrzeugtechnik [?, ?] und moderner Simulationsmethoden [?] adressiert.

1.1. Aufbau der Arbeit

Die Arbeit folgt einer systematischen Gliederung, die einen umfassenden Ansatz von der Konzeption bis zur Implementierung und Evaluierung verfolgt:

1. **Zielbild und Anforderungen** (Kapitel 2): Definition des Zielbilds, Scope, Anforderungen und Abnahmekriterien. Dieses Kapitel legt die Grundlage für die gesamte Arbeit und definiert, was erreicht werden soll.

2. **Komponententaxonomie und Metamodell** (Kapitel 3): Entwicklung einer erweiterbaren Taxonomie und eines präzisen Metamodells für moderne E/E-Architekturen. Dieses Kapitel definiert die strukturelle Basis für die Modellierung.

3. **PREEvision-Modellierung** (Kapitel 4): Detaillierte Beschreibung der Modellierung in PREEvision, einschließlich Topologie, Kommunikation, Software-Deployment und Datenqualität. Dieses Kapitel zeigt, wie Architekturen in PREEvision modelliert werden.

4. **Synthese-Metrik** (Kapitel 5): Konzeption und Erweiterung von Synthese-Metriken zur Ableitung simulativer Parameter aus Architekturmerkmalen. Dieses Kapitel beschreibt, wie aus statischen Modellen dynamische Simulationsparameter abgeleitet werden.

5. **Transformationskonzept** (Kapitel 6): Konzept für die Transformation von PREEvision-Modellen in Simulationsmodelle, einschließlich Zielplattformen, Mapping-Regeln und Intermediate Model. Dieses Kapitel beschreibt die Transformationsstrategie.

6. **Technische Realisierung** (Kapitel 7): Implementierung der Transformation, einschließlich Datenaufnahme, Normalisierung, Regel-Engine und Code-Generierung. Dieses Kapitel beschreibt die technische Umsetzung.

7. **Szenarien und Use-Cases** (Kapitel 8): Definition von Szenarien für die Simulation, einschließlich nominaler, Stress- und Fehlerszenarien. Dieses Kapitel beschreibt, welche Szenarien simuliert werden.

8. **Simulation und Auswertung** (Kapitel 9): Durchführung von Simulationen und Auswertung der Ergebnisse, einschließlich Metriken, Akzeptanzkriterien und Ergebnisaufbereitung. Dieses Kapitel beschreibt die Simulation und Analyse.

9. **Validierung und Iteration** (Kapitel 10): Validierung der Simulationsergebnisse und iterativer Prozess zur Verbesserung der Architektur. Dieses Kapitel beschreibt, wie die Ergebnisse validiert werden.

10. **Dokumentation und Deliverables** (Kapitel 11): Dokumentation und zu liefernde Artefakte, einschließlich Metamodell-Spezifikation, Transformationsregeln und Beispiel-Architekturen. Dieses Kapitel beschreibt die Dokumentation.

11. **Zeitplanung** (Kapitel 12): Grobe Zeitplanung für das Projekt mit verschiedenen Phasen und Meilensteinen. Dieses Kapitel beschreibt die Projektplanung.

12. **Risiken und Gegenmaßnahmen** (Kapitel 13): Identifizierte Risiken und geplante Gegenmaßnahmen. Dieses Kapitel beschreibt das Risikomanagement.

13. **Konkreter Minimalstart** (Kapitel 14): Konkreter Minimalstart mit kleinster End-to-End-Kette. Dieses Kapitel beschreibt den Proof-of-Concept.

14. **Fallstudien und Benchmarking** (Kapitel 15): Erweiterte Fallstudien und Benchmarking-Ergebnisse. Dieses Kapitel demonstriert die Anwendung der Methodik anhand realer Architekturen.

15. **Vergleichsstudien** (Kapitel 16): Vergleich mit anderen Ansätzen und Tools. Dieses Kapitel bewertet die Methodik im Vergleich zu existierenden Lösungen.

16. **KI-Integration** (Kapitel 17): Integration von KI-Modellen in E/E-Architekturen. Dieses Kapitel beschreibt die Modellierung und Simulation von KI-Modellen.

17. **Cybersecurity** (Kapitel 18): Cybersecurity in E/E-Architekturen. Dieses Kapitel behandelt Sicherheitsaspekte und deren Modellierung.

18. **Energiemanagement** (Kapitel 19): Energiemanagement in E/E-Architekturen. Dieses Kapitel beschreibt die Optimierung des Energieverbrauchs.

19. **Standards und Compliance** (Kapitel 20): Standards und Compliance in E/E-Architekturen. Dieses Kapitel behandelt relevante Standards und deren Einhaltung.

20. **Vergleich von Architekturen** (Kapitel 21): Vergleich verschiedener E/E-Architekturen. Dieses Kapitel analysiert verschiedene Architekturansätze.

21. **Bosch-Sensorik und NVIDIA DRIVE Thor Integration** (Kapitel 22): Integration von neuester Bosch-Sensorik und NVIDIA DRIVE Thor. Dieses Kapitel beschreibt die Integration modernster Hardware-Komponenten.

22. **VAN.APPVERSE – Die offene Mobilitäts-Microservice-Ökonomie** (Kapitel 23): Konzept für eine offene Mobilitäts-Plattform mit App Store und Microservices. Dieses Kapitel beschreibt die Vision einer offenen Ökonomie für Mobilitätsdienste.

23. **Vollständiges E/E-Architektur-Regelwerk für MB Vans** (Kapitel 24): Umfassendes Regelwerk für die Entwicklung neuer E/E-Architekturen für MB Vans. Dieses Kapitel systematisiert alle Aspekte von der Konzeption bis zum Betrieb und bietet eine vollständige Referenz für Architekten, Entwickler und Projektmanager.

Jedes Kapitel baut auf den vorherigen auf und trägt zur Gesamtlösung bei. Die Arbeit folgt einem inkrementellen Ansatz, bei dem zunächst ein Minimalstart realisiert wird, der dann schrittweise erweitert wird.

1.2. Methodik und Vorgehensweise

Diese Arbeit folgt einer systematischen Methodik zur Entwicklung und Validierung des Transformations-Frameworks.

1.2.1. Entwicklungsmethodik

Die Entwicklung erfolgt nach einem modellbasierten Ansatz:

1. **Anforderungsanalyse**: Detaillierte Analyse der Anforderungen an das Framework
2. **Metamodell-Entwicklung**: Entwicklung eines erweiterbaren Metamodells
3. **Transformations-Regel-Entwicklung**: Entwicklung von Mapping-Regeln
4. **Implementierung**: Implementierung des Transformations-Frameworks
5. **Validierung**: Validierung gegen analytische Modelle und reale Architekturen
6. **Evaluation**: Umfassende Evaluation mit Fallstudien

1.2.2. Validierungsmethodik

Die Validierung erfolgt auf mehreren Ebenen:

- **Unit-Tests**: Tests für einzelne Komponenten

- **Integration-Tests**: Tests für die Integration von Komponenten
- **System-Tests**: Tests für das gesamte Framework
- **Analytische Validierung**: Vergleich mit analytischen Modellen
- **Empirische Validierung**: Vergleich mit realen Messungen

1.2.3. Evaluationsmethodik

Die Evaluation erfolgt durch:

- **Fallstudien**: Anwendung auf reale Architekturen
- **Benchmarking**: Vergleich mit anderen Ansätzen
- **Performance-Analyse**: Analyse der Performance
- **Qualitäts-Analyse**: Analyse der Qualität der Ergebnisse

2. Zielbild, Scope und Anforderungen

2.1. Zielbild

Eingebettete Simulation innerhalb der Architekturentwicklungsumgebung zur Bewertung von Designvarianten, Aufdeckung von Bottlenecks und quantitativer Bewertung nichtfunktionaler Anforderungen (Timing, Bandbreite, Verfügbarkeit, Energie). Dieses Zielbild entspricht modernen Ansätzen der modellbasierten Entwicklung [?] und den Anforderungen an die Validierung komplexer E/E-Architekturen [?, ?].

Die Simulation ermöglicht es, Architekturentscheidungen frühzeitig zu validieren, bevor kostspielige Hardware-Prototypen erstellt werden. Dies ist besonders wichtig in der modernen Fahrzeugentwicklung, wo die Komplexität der E/E-Architekturen kontinuierlich zunimmt [?, ?].

2.2. Scope

Funktionsdomänen: Perzeption, Sensorfusion, Lokalisierung, Planung, Fahrzeugführung [?, ?]; Ebenen: Sensorik [?], Rechenknoten (ECUs), Aktorik, Kommunikation [?, ?]; Reifegrade: Konzept bis Vorserie.

Der Scope umfasst die gesamte Kette von der Architekturmodellierung bis zur Simulation, einschließlich moderner Kommunikationstechnologien wie TSN [?] und serviceorientierter Architekturen [?].

2.3. Anforderungen und KPIs

Nichtfunktionale Anforderungen umfassen: E2E-Latenz, Jitter, Deadline-Misses, Busauslastung, CPU/GPU-Last, Energie, ASIL-Level [?]; Randbedingungen: Hardware-Budgets, Kosten, Packaging, Temperaturbereiche [?].

Diese Anforderungen müssen unter Berücksichtigung von Echtzeit-Anforderungen [?] und funktionaler Sicherheit [?] erfüllt werden.

2.4. Abnahmekriterien

Grenzwerte pro Use-Case (z. B. E2E $<$ 100 ms für L2/L3 [?]), TSN-Jitterbudgets [?], Paketverlustraten in sicherheitskritischen Pfaden [?]. Diese Kriterien orientieren sich an etablierten Standards und Best Practices der Automobilindustrie [?].

2.5. Stand der Technik und Literaturübersicht

Dieser Abschnitt gibt einen Überblick über den aktuellen Stand der Technik im Bereich E/E-Architektur-Simulation und stellt die Arbeit in den Kontext aktueller Forschung, insbesondere aus KIT und TUM.

2.5.1. Forschung an KIT

Das Karlsruher Institut für Technologie (KIT) leistet bedeutende Beiträge zur Fahrzeugtechnik und E/E-Architektur-Entwicklung. Die Forschung am KIT umfasst insbesondere:

- **Fahrdynamik und Regelung**: Grundlagenforschung zur Fahrzeugdynamik [?, ?], die für die Simulation von Fahrzeugfunktionen essentiell ist.
- **Fahrzeugsystemtechnik**: Entwicklung von Methoden zur modellbasierten Entwicklung von Fahrzeugsystemen [?], die eng mit der in dieser Arbeit entwickelten Transformationsmethodik verwandt ist.
- **Energieeffizienz**: Forschung zur Optimierung des Energieverbrauchs in Fahrzeugen [?], was für die Energie-Modellierung in der Synthese-Metrik relevant ist.

Die Vorlesungen "Grundlagen der Fahrzeugtechnik Iünd "Grundlagen der Fahrzeugtechnik IIäm KIT [?, ?] vermitteln fundierte Kenntnisse zu Antriebssystemen, Radaufhängungen, Lenkung und Bremsen, die für die Modellierung von Aktorik-Komponenten in E/E-Architekturen wichtig sind.

2.5.2. Forschung an TUM

Die Technische Universität München (TUM) ist führend in der Forschung zu Automotive Software Engineering und modellbasierter Entwicklung:

- **Automotive Software Engineering**: Pionierarbeit zu Software-Architekturen für Fahrzeuge [?, ?], die die Grundlage für moderne E/E-Architekturen bildet.

- **Modellbasierte Entwicklung**: Entwicklung der SPES 2020 Methodik [?], die einen systematischen Ansatz zur modellbasierten Entwicklung eingebetteter Systeme bietet.
- **Simulation und Validierung**: Forschung zu Simulationsmethoden für Automotive-Systeme [?], die direkt mit dem Thema dieser Arbeit zusammenhängt.

2.5.3. Aktuelle Entwicklungen in der Industrie

Die Automobilindustrie entwickelt sich rasant in Richtung Software-Defined Vehicles (SDV) [?]:

- **Zonale Architekturen**: Zunehmende Verbreitung zonaler E/E-Architekturen mit zentralen Rechenplattformen, wie sie in dieser Arbeit modelliert werden.
- **TSN-Netzwerke**: Time-Sensitive Networking wird zum Standard für deterministische Kommunikation in Fahrzeugen [?, ?].
- **KI-Integration**: Integration von KI/ML-Modellen in E/E-Architekturen [?], was neue Anforderungen an die Simulation stellt.
- **Cybersecurity**: Zunehmende Bedeutung von Cybersecurity in vernetzten Fahrzeugen [?], die auch in Simulationsmodellen berücksichtigt werden muss.

2.5.4. Beitrag dieser Arbeit

Diese Arbeit leistet einen Beitrag zur Forschung, indem sie:

- **Methodik**: Eine systematische Methodik zur Transformation von Architekturmodellen in Simulationsmodelle entwickelt, die auf etablierten Ansätzen der modellbasierten Entwicklung [?] aufbaut.
- **Synthese-Metrik**: Eine erweiterte Synthese-Metrik entwickelt, die automatisch Simulationsparameter aus Architekturmerkmalen ableitet, was die Effizienz der Architektur-Evaluierung erheblich verbessert.
- **Validierung**: Eine systematische Validierungsmethodik entwickelt, die Simulationsergebnisse mit analytischen Modellen vergleicht [?].
- **Praktische Anwendung**: Die Methodik anhand konkreter Fallstudien demonstriert, die auf realen Anforderungen basieren.

Die Arbeit baut auf den Erkenntnissen aus KIT und TUM auf und erweitert diese um spezifische Aspekte der Simulation von E/E-Architekturen, insbesondere im Kontext moderner zonaler Architekturen und TSN-Netzwerke.

2.6. Erweiterte Anforderungsanalyse

Dieser Abschnitt beschreibt eine detaillierte Analyse der Anforderungen an das Transformations-Framework.

2.6.1. Funktionale Anforderungen

Transformation-Anforderungen

- **Unterstützte Formate**: PREEvision (JSON, XML, CSV), AUTOSAR XML, Custom-Formate
- **Zielplattformen**: OMNeT++, Simulink, NS-3, Modelica, ROS 2, FMI
- **Transformations-Genauigkeit**: Abweichung $< 5\%$ von analytischen Modellen
- **Performance**: Transformation < 2 Stunden für Architekturen mit 1000+ Knoten
- **Skalierbarkeit**: Unterstützung für Architekturen mit bis zu 10.000 Knoten

Validierungs-Anforderungen

- **Schema-Validierung**: Vollständige Validierung gegen Metamodell-Schema
- **Constraint-Validierung**: Validierung von Constraints und Randbedingungen
- **Konsistenz-Prüfung**: Prüfung auf Konsistenz zwischen verschiedenen Modell-Ebenen
- **Vollständigkeits-Prüfung**: Prüfung auf Vollständigkeit der Modell-Daten

2.6.2. Nichtfunktionale Anforderungen

Performance-Anforderungen

Tabelle 2.1.: Performance-Anforderungen

Metrik	Klein (<100)	Mittel (100-1000)	Groß (>1000)
Transformationszeit	< 5 min	< 30 min	< 2 h
Speicherverbrauch	< 1 GB	< 8 GB	< 32 GB
Simulationszeit (1h Fahrzeit)	< 10 min	< 1 h	< 4 h

Qualitäts-Anforderungen

- **Genauigkeit**: Simulationsergebnisse mit Abweichung $< 5\%$ von analytischen Modellen
- **Reproduzierbarkeit**: Gleiche Eingabe führt zu gleicher Ausgabe
- **Wartbarkeit**: Code-Coverage $> 80\%$, umfassende Dokumentation
- **Erweiterbarkeit**: Plugin-Architektur für neue Formate und Plattformen

2.6.3. Anwender-Anforderungen

Benutzerfreundlichkeit

- **Command-Line-Interface**: Einfache CLI für Standard-Operationen
- **API**: RESTful API für Integration in andere Tools
- **Konfiguration**: YAML-basierte Konfiguration für Flexibilität
- **Dokumentation**: Umfassende Dokumentation mit Beispielen

Integration-Anforderungen

- **CI/CD-Integration**: Integration in GitHub Actions, GitLab CI, Jenkins
- **Tool-Integration**: Integration in PREEvision, MATLAB/Simulink, etc.
- **Cloud-Integration**: Unterstützung für Cloud-Deployment (AWS, Azure, GCP)
- **Versionierung**: Unterstützung für Git, SVN, etc.

3. Komponententaxonomie und Metamodell (MB.OS Vans)

Dieses Kapitel entwickelt eine erweiterbare Komponententaxonomie und ein präzises Metamodell für eine moderne, dienstorientierte E/E-Architektur im Sinne von MB.OS (Fokus: Vans). Es adressiert eine zonale Topologie mit zentralen Rechenknoten, Ethernet-Backbone (TSN), serviceorientierter Kommunikation, sicherheitskritischen Funktionsketten und domänenspezifischer Sensorik/Aktorik für Nutzfahrzeugtypische Use-Cases (z.B. Lieferlogistik, leichte Nutzfahrzeuge, Flottenbetrieb).

3.1. Grundprinzipien und Struktur

Moderne E/E-Architekturen für Nutzfahrzeuge, insbesondere Vans, erfordern eine sorgfältige Balance zwischen Performance, Sicherheit, Skalierbarkeit und Kosten. Die zonale Architektur stellt einen Paradigmenwechsel dar, der weg von der traditionellen Domain-basierten Architektur hin zu einer geografisch organisierten Struktur führt. Diese Architektur ermöglicht es, die wachsende Komplexität moderner Fahrzeuge zu bewältigen, während gleichzeitig die Kosten durch Konsolidierung von ECUs reduziert werden [?, ?].

3.1.1. Zonale Architektur und zentrale Rechenplattform

Die zonale Architektur teilt das Fahrzeug in physische Zonen ein, wobei jede Zone von einem Zonen-Controller (ZC) verwaltet wird. Diese Controller sind über einen zentralen Ethernet-Backbone mit einer leistungsfähigen zentralen Rechenplattform verbunden. Dieser Ansatz bietet mehrere Vorteile:

- **Zonen-Controller (ZC)**: Versorgungs- und IO-Nähe, Aggregation lokaler Sensorik/Aktorik, Vorverarbeitung. Zonen-Controller sind strategisch im Fahrzeug positioniert (typischerweise Front, Links, Rechts, Heck), um die physische Nähe zu Sensoren und Aktoren zu gewährleisten. Dies reduziert die Kabel-Längen, verbessert die Signalqualität und reduziert die Kosten. Ein typischer Zonen-Controller verfügt über:

  - Mehrere CAN/LIN-Gateways für Legacy-Kommunikation
  - Ethernet-Ports für Anbindung an den Backbone
  - Lokale Verarbeitungsfähigkeit für einfache Funktionen (z. B. Türsteuerung, Beleuchtung)
  - Power-Management für die Zone
  - Diagnose-Funktionalität

- **Zentrale Rechenplattform (Central Compute, CC)**: Hochperformant (CPU/GPU/NPU), Safety-Partitionen (ASIL) und Virtualisierung. Die zentrale Rechenplattform ist das Herzstück der Architektur und führt komplexe Funktionen aus, die hohe Rechenleistung erfordern:
  - **CPU**: Multi-Core-Prozessoren (typischerweise 8-16 Kerne) mit big.LITTLE-Architektur für optimale Balance zwischen Performance und Energieeffizienz
  - **GPU**: Dedizierte Grafikprozessoren für visuelle Perzeption und Rendering (typischerweise 5-20 TFLOPS)
  - **NPU**: Neural Processing Units für KI/ML-Workloads (typischerweise 50-200 TOPS)
  - **Virtualisierung**: Hypervisor-basierte Partitionierung für Isolation zwischen verschiedenen ASIL-Leveln
  - **Safety-Partitionen**: Separate Partitionen für sicherheitskritische Funktionen (ASIL D) und nicht-kritische Funktionen (QM)

- **Ethernet-Backbone mit TSN**: Deterministische Latenzen, Priorisierungen, Redundanzen. Der Ethernet-Backbone ersetzt traditionelle Bussysteme und bietet:
  - **Bandbreite**: 1 Gbps, 2.5 Gbps oder 10 Gbps Links für hohe Datenraten
  - **TSN-Features**: Time-Sensitive Networking für deterministische Kommunikation
    * **Time Synchronization (gPTP)**: Präzise Zeitsynchronisation für deterministische Kommunikation
    * **Traffic Shaping**: Credit-Based Shaping (CBS) und Time-Aware Shaping (TAS) für Bandbreiten-Management
    * **Frame Preemption**: Unterbrechung von nicht-kritischen Frames durch kritische Frames

    * **Redundancy**: Parallel Redundancy Protocol (PRP) für hochverfügbare Kommunikation
  - **Switches**: Intelligente TSN-Switches mit Gate-Scheduling für deterministische Weiterleitung

- **Serviceorientierung**: DDS/SOMEIP für verteilte Dienste, Versionierung, QoS. Die serviceorientierte Architektur ermöglicht flexible, lose gekoppelte Kommunikation:
  - **DDS (Data Distribution Service)**: Publish-Subscribe-Middleware für Echtzeit-Kommunikation mit Quality-of-Service-Garantien
  - **SOME/IP**: Scalable service-Oriented MiddlewarE over IP für servicebasierte Kommunikation
  - **Versionierung**: Unterstützung für verschiedene Service-Versionen für Backward-Kompatibilität
  - **QoS**: Quality-of-Service-Parameter (Reliability, Deadline, Latency) für verschiedene Service-Typen

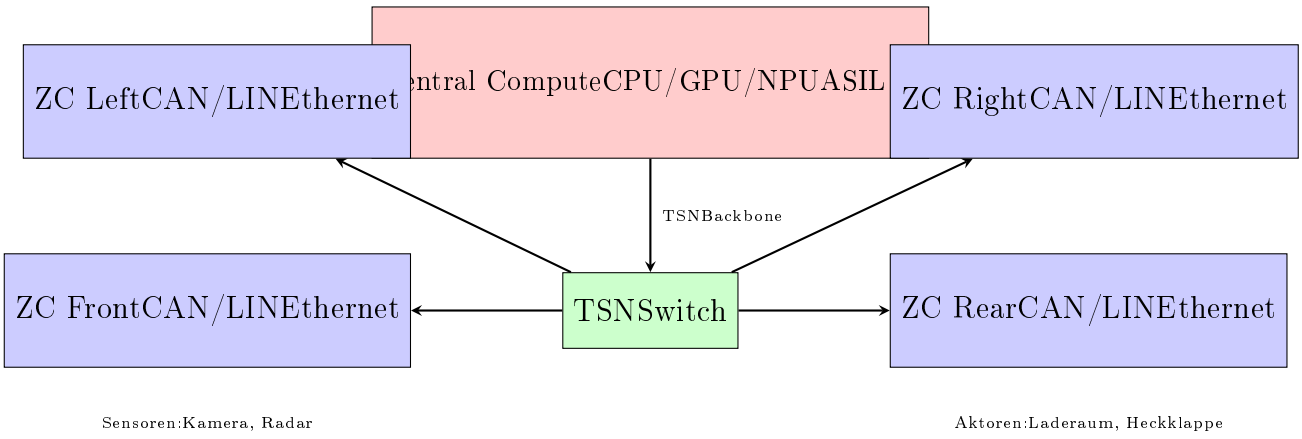

Abbildung 3.1.: Zonale E/E-Architektur mit zentraler Rechenplattform

3.1.2. Domänen und Use-Cases (Vans)

Nutzfahrzeuge, insbesondere Vans, haben spezifische Anforderungen, die sich von Personenkraftwagen unterscheiden. Die Domänen und Use-Cases müssen diese spezifischen Anforderungen berücksichtigen:

- **Karosserie/Body**: Türsteuerung, Heckklappensteuerung, Beleuchtung, Klimatisierung. Vans haben typischerweise mehrere Türen (Seitentüren, Heckklappe) und erfordern zuverlässige Steuerungssysteme. Beispiel-Use-Cases:

  - Automatische Heckklappenöffnung bei Annäherung (Hands-Free)
  - Intelligente Beleuchtung für Zustellfahrten (automatische Aktivierung bei Türöffnung)
  - Klimatisierung des Laderaums für temperaturgeführte Transporte

- **Antrieb/Powertrain**: Elektrischer Antrieb, Batteriemanagement, Ladeinfrastruktur. Moderne Vans sind zunehmend elektrifiziert:
  - E-Achsen mit integriertem Antrieb
  - 800V-Batteriesysteme für schnelles Laden
  - Intelligentes Batteriemanagement für optimale Reichweite
  - V2G (Vehicle-to-Grid) für Energie-Rückeinspeisung

- **Fahrerassistenz/AD**: Perzeption, Sensorfusion, Planung, Regelung. Für Vans sind spezifische ADAS-Funktionen wichtig:
  - **Urban Delivery Assist**: Unterstützung bei engen Gassen und Parkplätzen
  - **Blind-Spot-Monitoring**: Erweiterte Überwachung aufgrund der Größe
  - **Load-Aware Functions**: Funktionen, die die Ladung berücksichtigen (z. B. Bremsassistenz bei voller Ladung)
  - **L2/L3 Automation**: Teilautomatisierung für Autobahnfahrten

- **Infotainment/HMI**: Anzeige, Bedienung, Navigation, Konnektivität. Moderne Vans benötigen leistungsfähige HMI-Systeme:
  - Große Touchscreens für Navigation und Fahrzeuginformationen
  - Sprachsteuerung für Hands-Free-Bedienung
  - Smartphone-Integration (Apple CarPlay, Android Auto)
  - Fahrerassistenz-Visualisierung

- **Fracht-/Laderaumdomäne**: Kühlung, Zustelllogik, Sicherheit, Monitoring. Diese Domäne ist spezifisch für Nutzfahrzeuge:
  - **Temperaturregelung**: Präzise Temperaturregelung für Kühltransporte ($\pm 0.5C$ Genauigkeit)
  - **Zustelllogik**: Intelligente Organisation der Ladung für optimale Zustellreihenfolge
  - **Sicherheit**: Überwachung des Laderaums (Kameras, Sensoren)
  - **Monitoring**: Echtzeit-Überwachung von Ladung, Temperatur, Türstatus

  - **Lastverteilung**: Monitoring der Gewichtsverteilung für optimale Fahrstabilität

- **Flotten-/Telematikdienste**: OTA, Diagnose, Fleet Operations. Für gewerbliche Nutzung essentiell:
  - **OTA-Updates**: Over-the-Air-Updates für Software und Konfiguration
  - **Diagnose**: Fern-Diagnose und Predictive Maintenance
  - **Fleet Management**: Integration in Flottenmanagementsysteme
  - **Telemetrie**: Echtzeit-Datenübertragung (Position, Zustand, Verbrauch)
  - **Route-Optimierung**: Cloud-basierte Routenoptimierung für Zustellfahrzeuge

3.2. Komponententypen (Taxonomie)

3.2.1. Rechenknoten

- **AD Domain Controller (AD-DC)**: Perzeption, Fusion, Planung, Regelung; GPU/NPU-beschleunigt.
- **Zonen-ECU (ZC)**: IO-Management, Aktorik/Sensorik, lokale Diagnosen; Gateways zu CAN/LIN.
- **Gateway/Switch** (L3/L2, TSN-fähig): Segmentierung, Redundanzpfade, Traffic Shaping.
- **Telematik/Connectivity ECU (TCU)**: 5G/LTE, WLAN, GNSS; OTA, Flottenintegration.
- **HMI/Infotainment ECU**: Anzeige/Bedienung, Audio/Video, Interaktion.

3.2.2. Sensorik

Kameras (Mono/Stereo/Surround), Radar (Kurz-/Mittel-/Langstrecke), LiDAR, Ultraschall, GNSS/IMU, Fracht-/Laderaumsensorik (Temperatur, Türkontakte, Gewicht, Kamera), Innenraumkameras.

3.2.3. Aktorik

Lenkung (EPS), Bremse (EHB/EMB), Antrieb, Fahrwerk/Dämpfer, Lichtsysteme, Türen/Heckklappen, Laderaumklima.

3.2.4. Kommunikation

Ethernet (1/2.5/10 GbE) mit TSN (Zeitsynchronisation, Scheduling, Shaping), CAN/CAN-FD, LIN; ggf. FlexRay (Legacy); DDS/SOMEIP auf Service-Ebene.

3.2.5. Infrastruktur

Switches, Power Distribution Units, Speicher/Storage, Sicherheitsmodule (HSM/TPM), Takt-/Zeitquellen.

3.3. Metamodell-Elemente

3.3.1. Strukturale Basisklassen

Node, **Port**, **Interface**, **Link**, **NetworkSegment**; **PowerDomain**, **TimingDomain**.

3.3.2. Kommunikationsartefakte

Message, **Signal**, **TrafficFlow**, **Service**, **Topic**; **QoSProfile** (Periodizität, Deadline, Latenzbudget, Priorität, Zuverlässigkeit), **TSNSchedule**.

3.3.3. Software-/Laufzeitelemente

SoftwareComponent (SWC), **Task**, **Runnable**, **Chain**; **SchedulingPolicy** (preemptive/fixed priority/TSN-triggered), **WCET/BCET**.

3.3.4. Deployment und Safety

Deployment, **Partition**, **RedundancyGroup**, **ASILLevel**, **HealthMonitor**, **DiagnosticFunction**.

3.3.5. Ressourcen- und Energiedaten

ComputeProfile (CPU/GPU/NPU-Kapazität), **MemoryProfile**, **StorageProfile**, **PowerStateModel** (Duty-Cycle, Übergangszeiten), **ThermalEnvelope**.

Tabelle 3.1.: Rechenknoten-Attribute (Auszug)

Attribut	Typ	Beschreibung
cpu_cores	Integer	Anzahl/Topologie (big.LITTLE)
gpu_tflops	Float	Rechenleistung GPU/NPU
virtualization	Enum	none | hypervisor | containers
asil_level	Enum	QM | A | B | C | D
power_states	List	Zustände + Übergangszeiten
thermal_max	Float	zulässige Verlustleistung (W)

Tabelle 3.2.: Kommunikations-Attribute (Auszug)

Attribut	Typ	Beschreibung
link_speed	Enum	100M | 1G | 2.5G | 10G
tsn_prio	Integer	0..7 (IEEE 802.1Q)
latency_budget	Float	Ziel-Latenz (ms)
jitter_budget	Float	Ziel-Jitter (ms)
redundancy	Enum	none | PRP | HSR | dual-path

3.4. Attributkatalog (Auszug)

3.5. Stereotypen und Erweiterbarkeit

3.5.1. Stereotyp-Mechanismus

Stereotypen erweitern Basisklassen um domänenspezifische Felder (z. B. *CameraSensor*, *ADComputeNode*, *TSNBackboneLink*), versioniert und validiert über Schema-Definitionen.

3.5.2. Beispielhafte Stereotypen

- **CameraSensor**: fov_h/v, resolution, framerate, compression, preprocessing_level
- **RadarSensor**: bands, range, update_rate, doppler_resolution
- **LiDARSensor**: channels, spin_rate, point_density, range
- **ADComputeNode**: gpu_tflops, npu_tops, asil_isolation, virtualization
- **TSNBackboneLink**: gate_schedule, guard_bands, prio_map

3.5.3. Neueste Bosch-Sensorik

Bosch entwickelt fortschrittliche Sensoren für automatisiertes Fahren. Die neuesten Entwicklungen umfassen:

Bosch Multifunktionskamera (8-Megapixel)

Die neue Bosch Multifunktionskamera bietet folgende Spezifikationen:

- **Auflösung**: 8 Megapixel (3840x2160)
- **Horizontales Sichtfeld**: 120 Grad
- **Erkennungsreichweite**: Bis zu 300 Meter
- **Funktionen**:
  - Adaptive Geschwindigkeits- und Abstandsregelung
  - Notbremsungen innerhalb der eigenen Spur
  - Spurhalten in städtischen Gebieten
  - Erkennung und Anhalten an roten Ampeln
- **Serienproduktion**: Geplant für 2026
- **Interface**: Ethernet (1G/2.5G)
- **ASIL**: B-D (abhängig von Anwendung)

Bosch Radar-Sensoren

Bosch bietet ein umfassendes Portfolio an Radar-Sensoren:

- **Long-Range-Radar**: Reichweite bis 250 m, hohe Auflösung
- **Mid-Range-Radar**: Reichweite bis 160 m, mittlere Auflösung
- **Short-Range-Radar**: Reichweite bis 80 m, hohe Winkelauflösung
- **4D-Imaging-Radar**: 3D-Position + Geschwindigkeit, hohe Auflösung

Bosch LiDAR-Sensoren

Bosch entwickelt LiDAR-Sensoren für hochautomatisiertes Fahren:

- **High-Resolution-LiDAR**: Bis zu 64 Layer, 360° Abdeckung
- **Reichweite**: Bis zu 200 m
- **Punktdichte**: Hohe Punktdichte für präzise Objekterkennung
- **Interface**: Ethernet (1G/10G)

Sensorfusion

Bosch kombiniert verschiedene Sensoren für 360°-Umfelderfassung:

- **Kameras**: Front, Rear, Side-Kameras für visuelle Erkennung
- **Radar**: Mehrere Radar-Sensoren für Geschwindigkeits- und Abstandsmessung
- **LiDAR**: Für präzise 3D-Umgebungserfassung
- **Ultraschall**: Für Nahbereichserkennung (Parken)

3.6. Zonen-Topologie (Diagramm)

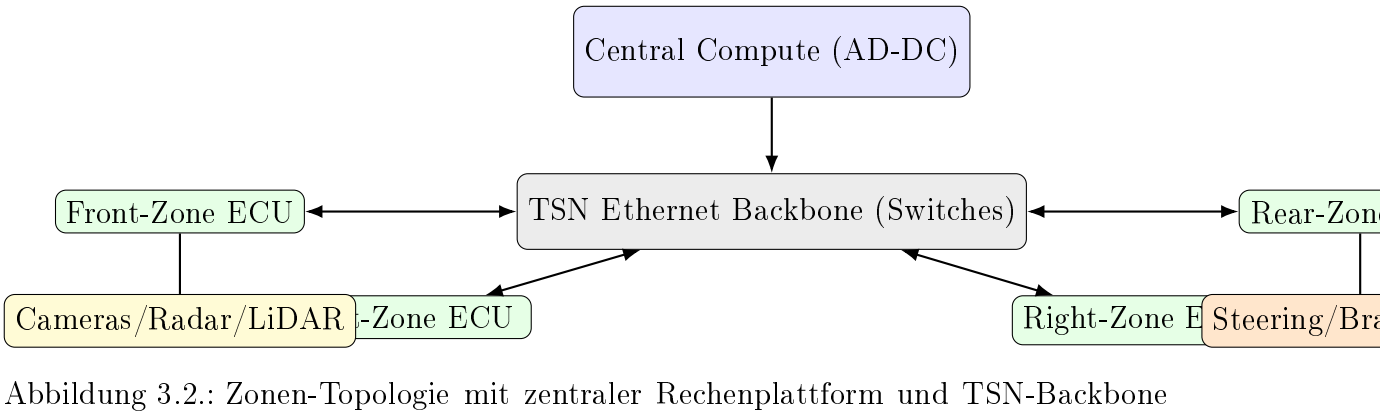

Abbildung 3.2.: Zonen-Topologie mit zentraler Rechenplattform und TSN-Backbone (schematisch).

Abbildung 3.3.: Beispielhafte Ende-zu-Ende-Kette von Sensorik über zentrale Rechenfunktion zur Aktorik.

3.7. Datenflussketten (Perzeption → Aktorik)

3.8. Qualitätsattribute und Metriken

3.8.1. Timing und Determinismus

E2E-Latenzbudgets, Jittergrenzen, Periodizitäten, Deadline-Misses; TSN-Schedule-Integration und Priorisierungsregeln.

3.8.2. Bandbreite und Auslastung

Linkauslastung, Queuing, Shaping; Redundanzpfade; Gateway-Last und Frame-/Signal-Aggregation.

3.8.3. Rechenlast und Energie

CPU/GPU/NPU-Profilierung (WCET/BCET, Duty-Cycles), Power-State-Strategien, thermische Budgetierung.

3.8.4. Safety und Security

ASIL-gerechte Partitionierung, Freedom-from-Interference, Diagnose- und Health-Management; Secure Boot, Key-Management, Secure Onboard Communication.

3.9. Validierungsregeln (Ausschnitt)

- Jeder *TrafficFlow* besitzt ein *QoSProfile* mit Periodizität, Deadline und Priorität.
- TSN-*Schedule* deckt alle deterministischen Flows ohne Konflikte ab (Guard Bands, Gate States).
- *Deployment* wahrt ASIL-Isolationsregeln; Redundanzgruppen sind physikalisch entkoppelt.

- *PowerStateModel* ist konsistent mit Latenzanforderungen (Aufwachzeiten $\leq$ Budget).

3.10. Erweiterte Taxonomie-Beispiele

Dieser Abschnitt präsentiert erweiterte Beispiele für die Anwendung der Taxonomie und des Metamodells auf komplexe Architekturen.

3.10.1. Beispiel: Vollständige Van-Architektur

Dieses Beispiel zeigt die vollständige Taxonomie für eine moderne Van-Architektur mit allen Domänen.

AD-Domain

Die AD-Domain umfasst alle Komponenten für automatisiertes Fahren:

- **Rechenknoten**:
  - 1x AD-DC (Central Compute): 16 CPU-Kerne, GPU 20 TFLOPS, NPU 200 TOPS
  - 4x Zonen-Controller: Front, Left, Right, Rear
- **Sensoren**:
  - 6x Kameras: Front (Stereo), Rear, Left, Right, Innenraum
  - 4x Radar: Front (Kurz/Lang), Rear (Kurz/Lang)
  - 1x LiDAR: 64-Layer, Front
  - 12x Ultraschall: Rundum-Überwachung
  - 1x GNSS/IMU: Position und Orientierung
- **Aktoren**:
  - 1x EPS: Elektrische Servolenkung
  - 1x EHB: Elektro-Hydraulische Bremse
  - 1x E-Motor: Elektrischer Antrieb
- **Kommunikation**:
  - TSN-Ethernet-Backbone: 2.5 Gbps
  - CAN-FD: Für Aktoren
  - DDS: Für serviceorientierte Kommunikation

Body-Domain

Die Body-Domain umfasst alle Komponenten für Karosserie-Funktionen:

- **Rechenknoten**:
  - 4x Zonen-Controller: Front, Left, Right, Rear
- **Sensoren**:
  - 8x Türkontakte: Für alle Türen
  - 4x Temperatursensoren: Innenraum, Laderaum
  - 2x Gewichtssensoren: Laderaum
  - 1x Laderaum-Kamera: Überwachung
- **Aktoren**:
  - 8x Tür-Aktoren: Öffnen/Schließen
  - 1x Heckklappen-Aktor: Öffnen/Schließen
  - 4x Beleuchtungsgruppen: Front, Rear, Left, Right
  - 1x Klimaanlage: Laderaum-Klimatisierung
- **Kommunikation**:
  - CAN-FD: Für Aktoren
  - LIN: Für einfache Sensoren
  - Ethernet: Für Zonen-Controller

Infotainment-Domain

Die Infotainment-Domain umfasst alle Komponenten für HMI und Unterhaltung:

- **Rechenknoten**:
  - 1x HMI-ECU: Hochleistungs-Prozessor für Displays
- **Displays**:
  - 1x Hauptdisplay: 12.3"Touchscreen
  - 1x Instrumenten-Display: 10.25"Digital
  - 1x Head-Up-Display: Projektion
- **Audio**:
  - 1x Audio-System: 8-Kanal-Sound

  - 1x Mikrofon-Array: Sprachsteuerung

- **Kommunikation**:
  - Ethernet: Für Display-Daten
  - USB: Für externe Geräte
  - Bluetooth: Für Smartphone-Integration
  - WLAN: Für Hotspot-Funktionalität

3.10.2. Beispiel: Metamodell-Instanziierung

Dieses Beispiel zeigt, wie das Metamodell für eine konkrete Architektur instanziiert wird.

ECU-Instanziierung

Eine ECU wird als Instanz des Metamodells erstellt:

```
{
  "type": "ECU",
  "id": "AD_DC_001",
  "name": "AD Domain Controller",
  "attributes": {
    "cpu_cores": 16,
    "cpu_freq_max": 2.5,
    "gpu_tflops": 20.0,
    "npu_tops": 200.0,
    "ram_size": 16,
    "storage_size": 256,
    "virtualization": "hypervisor",
    "asil_level": "D",
    "power_idle": 15.0,
    "power_max": 25.0,
    "thermal_max": 100.0
  },
  "ports": [
    {
      "id": "eth_port_1",
      "type": "Ethernet",
      "speed": "2.5G",
```

```
      "tsn_enabled": true
    }
  ]
}
```

Chain-Instanziierung

Eine Funktionskette wird als Instanz des Metamodells erstellt:

```
{
  "type": "Chain",
  "id": "Perception_to_Steering",
  "source": "Camera_Front",
  "sink": "EPS",
  "path": [
    "Camera_Front",
    "ZC_Front",
    "AD_DC",
    "EPS"
  ],
  "attributes": {
    "e2e_deadline": 100.0,
    "e2e_latency_budget": 100.0,
    "e2e_jitter_budget": 10.0,
    "asil_level": "D"
  }
}
```

3.11. Stateflow-Diagramme für Architektur-Zustände

Stateflow-Diagramme ermöglichen die Modellierung komplexer Zustandsübergänge in E/E-Architekturen. Diese Diagramme sind besonders wertvoll für die Modellierung von Power-States, Fehlerzuständen und Betriebsmodi.

3.11.1. Power-State-Maschine

Die Power-State-Maschine beschreibt die verschiedenen Energiezustände eines Rechenknotens:

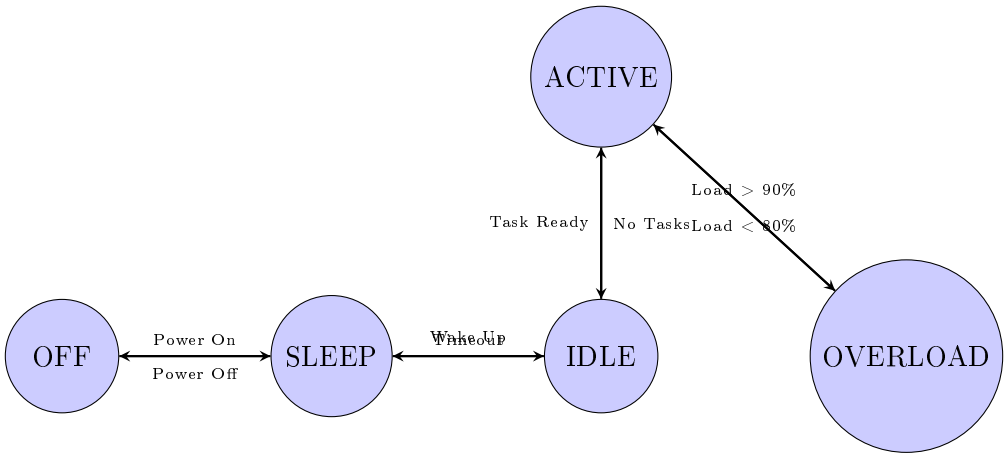

Abbildung 3.4.: Power-State-Maschine für einen Rechenknoten

Die Power-State-Maschine umfasst folgende Zustände:

- **OFF**: Komplett ausgeschaltet, keine Energieversorgung
- **SLEEP**: Niedrigster Energiezustand, minimale Funktionen aktiv (z. B. Wake-up-Detection)
- **IDLE**: Bereit, aber keine aktiven Tasks, niedrige Energie
- **ACTIVE**: Normale Betriebsphase mit aktiven Tasks
- **OVERLOAD**: Überlastung, erfordert Maßnahmen (z. B. Throttling, Task-Migration)

3.11.2. Fehlerzustands-Maschine

Die Fehlerzustands-Maschine beschreibt die Reaktion auf Fehler:

3.11.3. Betriebsmodus-Maschine

Die Betriebsmodus-Maschine beschreibt verschiedene Betriebsmodi:

- **Normal Mode**: Standard-Betrieb
- **Eco Mode**: Energieoptimierter Betrieb
- **Performance Mode**: Leistungsoptimierter Betrieb
- **Diagnostic Mode**: Diagnose-Modus
- **Maintenance Mode**: Wartungsmodus

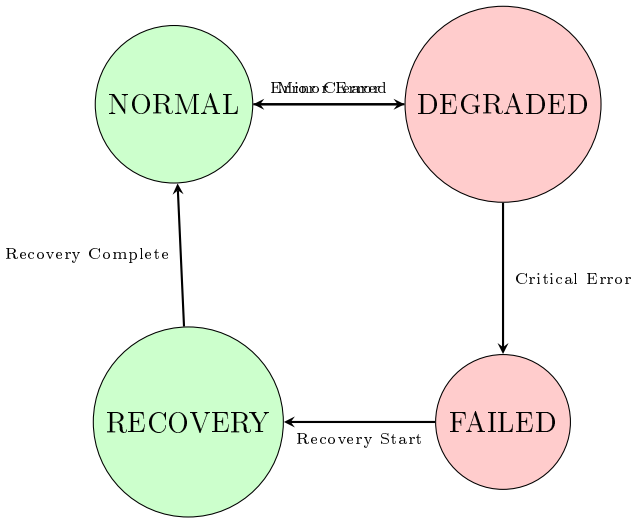

Abbildung 3.5.: Fehlerzustands-Maschine für Fehlerbehandlung

3.12. Zusammenfassung

Die vorgestellte Taxonomie und das Metamodell bilden eine MB.OS-nahe, zonale E/E-Architektur für Vans ab. Sie sind erweiterbar, quantitativ auswertbar und dienen als Grundlage für die Synthese-Metrik und die Transformation in Simulationsmodelle. Die Stateflow-Diagramme ermöglichen eine präzise Modellierung von Zustandsübergängen und Betriebsmodi.

3.13. Erweiterte Taxonomie-Anwendungen

Dieser Abschnitt beschreibt erweiterte Anwendungen der Taxonomie und des Metamodells.

3.13.1. Taxonomie für verschiedene Fahrzeugtypen

Die Taxonomie kann für verschiedene Fahrzeugtypen angepasst werden:

Personenkraftwagen (Pkw)

- **Spezifische Komponenten**: Komfort-Funktionen, Infotainment, Assistenzsysteme
- **Anforderungen**: Hohe Komfort-Anforderungen, moderate Sicherheitsanforderungen
- **Architektur**: Kompaktere Architektur, weniger Zonen

Nutzfahrzeuge (Lkw)

- **Spezifische Komponenten**: Flotten-Management, Telematik, Laderaum-Management
- **Anforderungen**: Hohe Zuverlässigkeit, Energieeffizienz
- **Architektur**: Erweiterte Architektur, mehr Zonen, robustere Komponenten

Elektrofahrzeuge

- **Spezifische Komponenten**: Batterie-Management, Lade-Management, Energiemanagement
- **Anforderungen**: Hohe Energieeffizienz, optimale Reichweite
- **Architektur**: Energie-optimierte Architektur, intelligentes Energiemanagement

3.13.2. Metamodell-Erweiterungen

Das Metamodell kann für spezifische Anforderungen erweitert werden:

KI-Modell-Integration

- **KI-Modell-Komponenten**: Neue Komponenten für KI-Modelle
- **Inferenz-Pipeline**: Modellierung von Inferenz-Pipelines
- **Training-Pipeline**: Modellierung von Training-Pipelines
- **Deployment**: Modellierung von Deployment-Prozessen

Cybersecurity-Integration

- **Sicherheitszonen**: Modellierung von Sicherheitszonen
- **Sicherheitsmechanismen**: Modellierung von Sicherheitsmechanismen
- **Threat-Modell**: Modellierung von Bedrohungen
- **Risk-Assessment**: Modellierung von Risikobewertungen

3.14. Erweiterte Metamodell-Details

Dieser Abschnitt beschreibt erweiterte Details des Metamodells.

3.14.1. Attribut-Definitionen

ECU-Attribute

Tabelle 3.3.: ECU-Attribute im Metamodell

Attribut	Typ	Beschreibung	Beispiel
name	String	Eindeutiger Name	ÄD-DC-01"
type	Enum	ECU-Typ	CentralCompute, ZoneController
cpu_cores	Integer	Anzahl CPU-Kerne	8
cpu_frequency	Float	CPU-Frequenz (GHz)	2.5
gpu_performance	Float	GPU-Performance (TFLOPS)	20.0
npu_performance	Float	NPU-Performance (TOPS)	200.0
memory_size	Integer	RAM-Größe (GB)	16
power_consumption	Float	Leistung (W)	50.0
asil_level	Enum	ASIL-Level	ASIL_D

Interface-Attribute

Tabelle 3.4.: Interface-Attribute im Metamodell

Attribut	Typ	Beschreibung	Beispiel
name	String	Interface-Name	ëth0"
type	Enum	Interface-Typ	Ethernet, CAN, LIN
bandwidth	Float	Bandbreite (Mbps)	1000.0
latency	Float	Latenz (ms)	0.1
jitter	Float	Jitter (ms)	0.01
tsn_enabled	Boolean	TSN aktiviert	true
tsn_config	Object	TSN-Konfiguration	{...}

Task-Attribute

3.14.2. Beziehungen im Metamodell

ECU-zu-Interface

- **Kardinalität**: 1:N (eine ECU hat mehrere Interfaces)
- **Richtung**: Uni-direktional (ECU → Interface)
- **Constraints**: Jede ECU muss mindestens ein Interface haben

Tabelle 3.5.: Task-Attribute im Metamodell

Attribut	Typ	Beschreibung	Beispiel
name	String	Task-Name	"perception_task"
period	Float	Periode (ms)	33.0
wcet	Float	WCET (ms)	15.0
bcet	Float	BCET (ms)	10.0
priority	Integer	Priorität	10
cpu_core	Integer	CPU-Core	0
asil_level	Enum	ASIL-Level	ASIL_C

Task-zu-ECU

- **Kardinalität**: N:1 (mehrere Tasks auf einer ECU)
- **Richtung**: Uni-direktional (Task → ECU)
- **Constraints**: Task muss auf genau einer ECU deployed sein

Frame-zu-Route

- **Kardinalität**: 1:N (ein Frame kann mehrere Routen haben)
- **Richtung**: Uni-direktional (Frame → Route)
- **Constraints**: Route muss aus mindestens 2 Hops bestehen

3.15. Fallstudie A: Zustell-Van mit Kühlkette (Urban Last Mile)

3.15.1. Ausgangslage und Anforderungen

Ein urbaner Zustell-Van mit temperaturgeführtem Laderaum (2 Zonen), L2-Assistenz (Stop-and-Go, Spurführung), Flottenanbindung und OTA-Updates. Anforderungen: E2E $< 100\,\text{ms}$ für Lenk-/Bremsketten, $\pm 0.5\,\text{C}$ Temperaturregelung, hochverfügbare Tür-/Sicherheitsfunktionen.

3.15.2. Architekturzuordnung

- Zonen-ECUs: Front (Perzeption-Sensor-IO), Rear (Laderaum IO, Aktorik Kühlung/Heckklappe)
- Central Compute: AD-DC mit GPU/NPU; TSN-Backbone (1/2.5 GbE), PRP für Sicherheitsketten
- Services: Perception_Detections, Planning_Trajectory, Cargo_TempReport, Door_Access

3.15.3. Ende-zu-Ende-Kette (Lenkung)

Tabelle 3.6.: E2E-Budgetzerlegung (Lenkung, urban)

Teilpfad	Anteil [ms]	Erläuterung
Sensoringress (ZC)	8	Vorverarbeitung, Timestamping
Netz (ZC $\rightarrow$ CC)	6	TSN, Prio Q7, 2 Hops
Perzeption + Planung (CC)	50	GPU/NPU Pipelines
Netz (CC $\rightarrow$ ZC Rear)	6	TSN, Gate-Window synchron
Aktuatorbus (CAN-FD)	10	Kommandotransfer
Sicherheitsmarge	20	Jitter/Spitzen
Summe	100	innerhalb Budget

3.15.4. Energieprofil Laderaumsensorik

Duty-Cycle-Strategie: *sleep* (90%), *idle* (5%), *active* (5%). Bei $P_{sleep} = 0.02\,\text{W}, P_{idle} = 0.2\,\text{W}, P_{active} = 1.0\,\text{W}$ ergibt sich $P_{avg} = 0.072\,\text{W}$. Aufwachzeit $t_{wake} = 50\,\text{ms}$ ist mit E2E-Anforderungen für Tür-/Alarmdienste verträglich (nicht sicherheitskritisch).

3.16. Fallstudie B: Regionalverkehr/Autobahn (L2+/L3-Features)

3.16.1. Ausgangslage und Anforderungen

Langstrecke mit höheren Geschwindigkeiten: engere E2E-Budgets (Lenkung 60 ms), höhere Sensorraten (Kameras 4x60 FPS, Radar 20 Hz, LiDAR 10 Hz), verlässliche Redundanz.

3.16.2. TSN-Planung und Redundanz

- TSN-GCL mit verdichteten Q7-Fenstern (alle 0.25 ms) für Perzeption/Control
- PRP auf Backbone, Dual-Homing der AD-DC zu zwei Switch-Bäumen
- DDS *Perception_ Critical*: reliable, Deadline 3 ms, Latenzbudget 1 ms

3.16.3. Rechenlastvariante

Tabelle 3.7.: Perzeptionsvariante (hohe Rate)

Sensor	Rate	Datenrate	Last (Tendenz)
Kamera	60 FPS	298Mbit/s (H.265 k=50)	hoch (GPU/NPU)
Radar	20 Hz	20Mbit/s	mittel (CPU)
LiDAR	10 Hz	200Mbit/s	hoch (GPU)

Optimierungen: Region-of-Interest (ROI) Vorverarbeitung am ZC, adaptive Kompression, Scheduling-Priorisierung.

3.17. Konkrete QoS-Profile und Serviceversionierung

3.17.1. QoS-Profile (DDS/SOME-IP)

Tabelle 3.8.: QoS-Parameter (Beispiele)

Service	Reliability	History	Deadline [ms]	Latenzbudget [ms]
Perception_Detections	reliable	keep_last(5)	5	2
Planning_Trajectory	reliable	keep_last(3)	2	1
Cargo_TempReport	best-effort	keep_last(1)	200	100
Door_Access	reliable	keep_all	50	10

3.17.2. Versionierung/Kompatibilität

Service-IDs und Versionsfelder (major/minor) sind Teil des Interfacevertrags; *compatibility policies* (strict/relaxed) steuern Interoperabilität. Deployment-Regeln verlangen koexistente Minor-Versionen bei Rolling Updates (OTA).

3.18. TSN-Engineering: Von Anforderungen zur GCL

3.18.1. Anforderungsableitung

Aus E2E-Forderungen werden Flow-spezifische Budgets abgeleitet: $B_{net} = \{1..8\}\,\text{ms}$ je Kette. Daraus ergeben sich Queue-Prioritäten und Gate-Fenstergröße.

3.18.2. Beispiel-GCL Ableitung

Bei $T_{cycle} = 1\,\text{ms}$ und kritischem Flow mit $L_{net} = 200\,\mu s$ wird $\geq 0.25\,\text{ms}$ Gate-Zeit für Q7 reserviert; Guard Band 0.05 ms. Übrige Zeit an Q6/Q5 (Diagnose/HMI) und Q0..Q4 (Best-Effort).

3.19. Sicherheits- und Diagnosepfade (ASIL, Health-Monitoring)

3.19.1. ASIL-Isolation

Safety-relevante SWCs laufen in isolierten Partitionen/VMs; Zugriff auf Bus/Netz via geprüften Proxies (Freedom from Interference).

3.19.2. Health-Monitoring

Watchdogs, Heartbeats und Thresholds (z. B. Max Jitter, Paketverlust) lösen Degradationsmodi aus (Fallback-Sensorik, niedrigere Feature-Stufe), stillhalten gemäß [?].

3.20. Gateway-Strategien: Aggregation, Routing, Security

3.20.1. Aggregation

Signale mit gleicher Periodizität und Ziel werden zu Frames/Flows gebündelt; reduziert Overhead und Jitter, erfordert dafür Framegrößen-/Maximum Transmission Unit (MTU)-Anpassung.

3.20.2. Routing

Statische Pfade für deterministische Ketten; dynamische Pfade (ECMP) für non-critical Services. PRP-/HSR-Pfade sind entkoppelt von Best-Effort.

3.20.3. Security

Message Authentication (MAC), Secure Onboard Communication, Firewall-Regeln und Service-ACLs schützen gegen unautorisierte Zugriffe.

3.21. Detaillierte Stereotypen-Beispiele

3.21.1. CameraSensor

Attribute: *fov_h/v, resolution, framerate, compression, preprocessing_level, thermal_range.* Beispiel: Front-Kamera 120°/70°, 1920x1080 @60FPS, H.265 k=50, preprocessing=low.

3.21.2. ADComputeNode

Attribute: *gpu_tflops, npu_tops, asil_isolation, virtualization, thermal_max.* Beispiel: 30 TFLOPS, 120 TOPS, ASIL-VM, Hypervisor, 80 W.

3.21.3. TSNBackboneLink

Attribute: *gate_schedule, guard_bands, prio_map, redundancy.* Beispiel: GCL-Zyklus 1 ms, Q7 0.25 ms, Guard 0.05 ms, PRP.

3.22. Traceability: Von Anforderung zu Modell und Simulation

- Anforderung → Modellattribute (QoS, Latenzbudget, ASIL)
- Modellattribute → GCL/Deployment/Mapping-Regeln
- Regeln → Simulationsartefakte (Flows, Schedules, Tasks)

Diese Rückverfolgbarkeit ermöglicht gezieltes Varianten- und What-if-Engineering.

3.23. Use-Cases und Domänenspezifika für Vans

3.23.1. Flotten- und Zustelllogistik

Routenoptimierung, vorausschauende Wartung, dynamische Ladungsüberwachung (Temperatur, Gewicht, Türzustände), Just-in-Time-Lieferfenster. Erfordert robuste Konnektivität (5G/DSRC/WLAN), sichere OTA-Mechanismen und priorisierte Datenkanäle.

3.23.2. Laderaumdomäne

Kühlkettenmanagement, Videoüberwachung, Zutrittskontrolle, Telematik-Integration. Hohe Sensordichte, teils batteriebetriebene Geräte, energieeffiziente Kommunikationsmodi (Wake-on-LAN, Low-Power-States).

3.23.3. Assistiertes/Automatisiertes Fahren (L2/L3)

Perzeption (Kamera/Radar/LiDAR), Fusion, Lokalisierung, Pfadplanung, Aktorik. Strenge Latenz- und Safety-Anforderungen [?, ?].

3.24. TSN-Scheduling-Beispiel (Zeitgating)

Abbildung 3.6.: Schema eines wiederkehrenden TSN-Gate-Schedules (vereinfachte Visualisierung, vgl. [?]).

Der Gate-Plan unterscheidet kritische deterministic Flows (z. B. Perzeption) von Best-Effort-Verkehr. Guard-Bands und Gate-States verhindern Interferenzen und sichern Jitterbudgets.

3.24.1. Gate Control List (GCL) und Guard Bands

Tabelle 3.9.: Beispielhafte GCL für einen TSN-Port (vereinfachter Auszug, vgl. [?])

Intervall	Dauer [ms]	Geöffnete Queues	Guard Band
I1	0.50	Q7 (kritisch)	aktiv
I2	0.50	Q6 (hoch), Q5 (mittel)	aktiv
I3	0.50	Q0..Q4 (Best-Effort)	inaktiv
I4	0.50	Q7 (kritisch)	aktiv

Die GCL definiert periodische Gate-States je Queue. Guard Bands verhindern, dass lange Best-Effort-Frames die deterministische Übertragung kritischer Flows blockieren.

3.24.2. QoS-Profile und Latenzbudgets

Tabelle 3.10.: QoS-Profile (Auszug) für dienstorientierte Flows (DDS/SOME-IP)

Profil	Zuverlässigkeit	Latenzbudget [ms]	Deadline [ms]
Perception_Critical	reliable	2	5
Control_Actuation	reliable	1	2
Diagnostics	best-effort	50	200
Infotainment	best-effort	100	500

Die Latenzbudgets werden auf Teilpfade (ZC → CC → Aktorik) und Schichten (Netz, Verarbeitung, Scheduling) aufgeteilt.

3.25. Redundanz-Topologien (PRP/HSR)

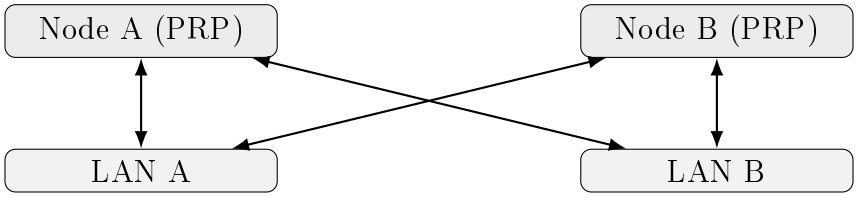

Abbildung 3.7.: PRP-Dual-LAN-Redundanz (schematisch), vgl. [?].

PRP (Parallel Redundancy Protocol) und HSR (High-availability Seamless Redundancy) ermöglichen nahtlose Umschaltung ohne Paketverlusten in sicherheitskritischen Pfaden [?].

3.25.1. Latenz- und Verfügbarkeitsrechnung

Die E2E-Latenz L_{E2E} einer redundanten Kette ergibt sich näherungsweise als

$$L_{E2E} = L_{tx} + L_{sw} \cdot n_{sw} + L_{prop} + L_{proc},$$

wobei L_{tx} die Sendezeit, L_{sw} die Switch-Verzögerung je Hop, n_{sw} die Hop-Anzahl, L_{prop} die Leitungsausbreitung und L_{proc} die Verarbeitungszeit umfasst. Bei PRP erfolgt paketweise Duplikation; die empfangsseitige Auswahl des zuerst eintreffenden Pakets reduziert den Effekt von Einzelpfad-Jitter.

Die Verfügbarkeit A einer dualen PRP-Topologie mit zwei unabhängigen Pfaden (Verfügbarkeiten A_1, A_2) wird als

$$A = 1 - (1 - A_1) \cdot (1 - A_2)$$

approximiert. Für identische Pfade mit $A_1 = A_2 = A_p$ gilt $A = 1 - (1 - A_p)^2$. Dies illustriert die starke Wirkung redundanter Pfade auf die mittlere Betriebszeit zwischen Ausfällen (MTBF).

3.26. Serviceorientierung und Registry

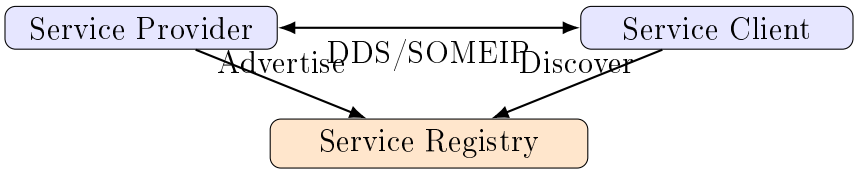

Abbildung 3.8.: Dienstorientierte Kommunikation mit Registry/Discovery (DDS/SOME-IP) [?, ?].

3.27. Security- und Virtualisierungsarchitektur

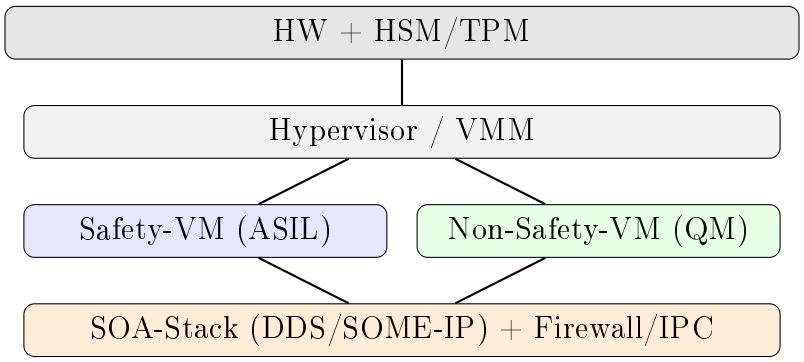

Abbildung 3.9.: Isolations- und Kommunikationsarchitektur mit Virtualisierung (AUTOSAR Adaptive/SOA) [?, ?].

3.28. Gateway-Routing und Protokollübergänge

Abbildung 3.10.: Protokollübergang und Aggregation im Gateway (CAN ↔ Ethernet/TSN) [?].

3.29. Erweiterter Attribut- und Stereotypenkatalog

Tabelle 3.11.: Erweiterter Attributkatalog (Auszug)

Element	Attribut	Typ	Beschreibung
ADComputeNode	npu_tops	Float	NPU-Leistung (TOPS)
ADComputeNode	asil_isolation	Bool	HW/SW-Isolation für ASIL
TSNBackboneLink	gate_schedule	JSON	Zeitgating-Konfiguration
TSNBackboneLink	redundancy	Enum	none | PRP | HSR
CameraSensor	compression	Enum	raw | h264 | h265
CameraSensor	preprocessing_level	Enum	none | low | high
RadarSensor	doppler_resolution	Float	m/s
LiDARSensor	point_density	Integer	Punkte/Frame
Service	qos_profile	Struct	Latenzbudget, Zuverlässigkeit, Deadline
Task	wcet/bcet	Float	Worst-/Best-Case-Exec-Time
Deployment	partition	Enum	VM | Container | Native

3.30. Normative Referenzen

Die vorliegende Struktur referenziert u. a. [?, ?, ?, ?, ?, ?].

3.31. Vans-spezifische Laderaum-Sensorketten und Energieprofile

3.31.1. Sensorketten

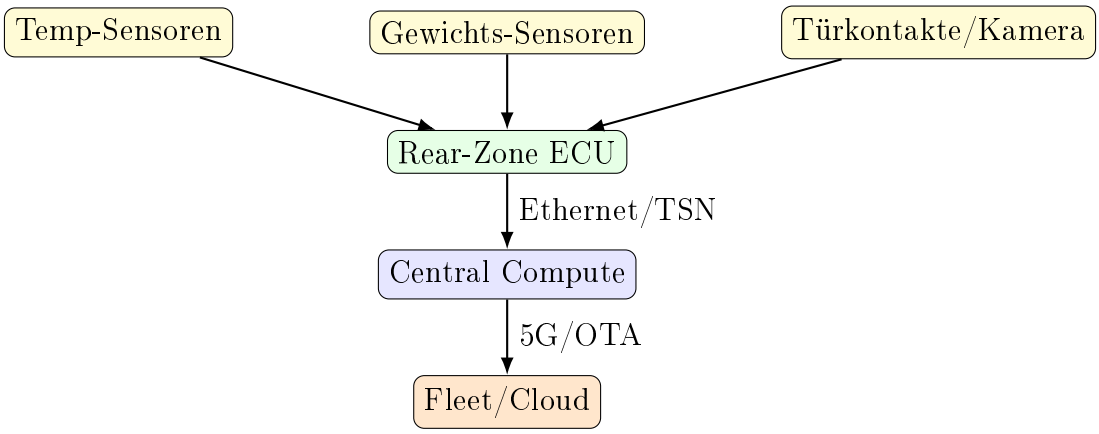

Abbildung 3.11.: Laderaum-Sensorkette mit Upload zu Flotten-/Cloud-Diensten.

3.31.2. Energieprofile

Für batteriebetriebene Laderaumsensorik sind Niedrigenergie-Zustände essenziell. Das *PowerStateModel* enthält Zustände $\{sleep, idle, active\}$ mit Übergangszeiten. Der mittlere Energieverbrauch P_{avg} ergibt sich aus Duty-Cycles d_i und Leistungswerten P_i:

$$P_{avg} = \sum_i d_i \cdot P_i, \quad \sum_i d_i = 1.$$

Im Scheduling müssen Aufwachzeiten t_{wake} das Latenzbudget wahren.

3.32. Metrik-Tabellen und Rechenbeispiele je Sensorpipeline

3.32.1. Rechenlastabschätzung

Für eine Kamera-Pipeline mit Auflösung $R = 1920 \times 1080$, Framerate $f = 30\,\text{Hz}$, Farbtiefe $b = 12\,\text{bit}$ und Kompressionsfaktor k ergibt sich die Bruttodatenrate

$$D = R \cdot f \cdot b/k.$$

Die zugehörige CPU/GPU-Last C hängt von der Algorithmenkomplexität α (Operationen pro Byte) ab: $C = D \cdot \alpha$.

Tabelle 3.12.: Beispielhafte Metriken für Sensorpipelines

Pipeline	Datenrate	α [ops/Byte]	Rechenlast	E2E-Budget [ms]
Kamera (H.265, k=50)	149Mbit/s	20	hoch (GPU/NPU)	50
Radar (20 Hz)	20Mbit/s	5	mittel (CPU)	30
LiDAR (10 Hz)	200Mbit/s	10	hoch (GPU)	80

3.32.2. Budgetzerlegung und Pfadkontrolle

Das E2E-Budget wird auf Teilpfade zugewiesen: $B_{E2E} = B_{sens} + B_{net} + B_{proc} + B_{act}$. Eine Verletzung eines Teilbudgets triggert Re-Mapping (z. B. TSN-Prio erhöhen, SWC auf CC verlagern, Vorverarbeitung in ZC).

4. Architekturmodellierung in PREEvision

Dieses Kapitel beschreibt die Modellierung einer modernen E/E-Architektur in PREEvision, einer etablierten Werkzeugkette für die Entwicklung von Elektrik/Elektronik-Architekturen in der Automobilindustrie [?]. PREEvision ermöglicht die modellbasierte Entwicklung von Hardware- und Software-Architekturen mit Fokus auf Topologie, Kommunikation, Software-Deployment und Datenqualität [?, ?]. Im Kontext dieser Arbeit wird PREEvision als Quellsystem für die Architekturmodellierung verwendet, aus dem die Daten für die Transformation in ein Simulationsmodell extrahiert werden.

Die Modellierung moderner E/E-Architekturen erfordert eine umfassende Berücksichtigung verschiedener Aspekte, die in Standardwerken zur Fahrzeugtechnik [?, ?] und E/E-Architekturen [?, ?] beschrieben werden. Insbesondere die zunehmende Komplexität durch autonomes Fahren [?] und moderne Kommunikationstechnologien wie TSN [?] erfordern detaillierte Modellierungsansätze.

4.1. Topologiemodell

4.1.1. Grundstruktur und Komponenten

Das Topologiemodell in PREEvision bildet die physische Struktur der E/E-Architektur ab. Es besteht aus hierarchisch organisierten Komponenten, die über definierte Schnittstellen miteinander verbunden sind. Die zentrale Abstraktionsebene bilden *Knoten* (Nodes), die physische oder logische Einheiten repräsentieren.

- **Rechenknoten (ECUs)**: Zentrale Rechenplattformen (Central Compute), Zonen-Controller, Domain-Controller, Gateways und spezialisierte Steuergeräte. Jeder Rechenknoten verfügt über eine eindeutige Identifikation, Ressourcenprofile (CPU, GPU, NPU, Speicher) und Safety-Level (ASIL).
- **Sensoren**: Kameras, Radar, LiDAR, Ultraschall, GNSS/IMU sowie domänenspezifische Sensoren (z. B. Temperatur, Türkontakte, Gewichtssensoren im La-

deraum). Sensoren werden mit ihren technischen Parametern (Auflösung, Framerate, Reichweite) modelliert.

- **Aktoren**: Lenkung (EPS), Bremse (EHB/EMB), Antrieb, Fahrwerk, Lichtsysteme, Türen/Heckklappen, Klimaanlagen. Aktoren werden mit ihren Ansteuerungscharakteristika (Latenz, Genauigkeit, Redundanz) beschrieben.
- **Switches und Gateways**: Netzwerkkomponenten für Ethernet-Backbones, TSN-Switches, CAN/LIN-Gateways. Diese Komponenten ermöglichen die Segmentierung und das Routing von Datenströmen.

4.1.2. Ports und Interfaces

Jeder Knoten verfügt über *Ports*, die als physische oder logische Anschlussstellen fungieren. Ports werden zu *Interfaces* zusammengefasst, die eine konsistente Schnittstellendefinition ermöglichen. Ein Interface definiert:

- **Protokoll und Datenrate**: Ethernet (100M/1G/2.5G/10G), CAN/CAN-FD, LIN, FlexRay
- **Signal-/Nachrichtendefinitionen**: Datenstrukturen, die über das Interface übertragen werden
- **QoS-Parameter**: Prioritäten, Periodizität, Latenzanforderungen, Zuverlässigkeit
- **TSN-Konfiguration**: Gate-Schedules, Prioritäts-Mappings, Redundanzpfade

4.1.3. Leitungen und Verbindungen

Leitungen (Links) verbinden Ports zwischen Knoten und bilden die physikalische oder logische Verbindung ab. Eine Leitung kann:

- **Point-to-Point** sein: Direkte Verbindung zwischen zwei Knoten
- **Multi-Drop** sein: Mehrere Knoten an einem Bus (CAN, LIN)
- **Switched** sein: Verbindung über einen Switch im Ethernet-Backbone

Jede Leitung wird mit Attributen wie Bandbreite, Latenz, Jitter-Budget, Redundanzpfaden und TSN-Parametern annotiert.

4.1.4. Ressourcen- und Energieprofile

Ressourcenprofile werden über *Attributkataloge* definiert, die jedem Knoten zugeordnet werden können. Diese Kataloge enthalten:

Tabelle 4.1.: Attributkatalog für Rechenknoten (Auszug)

Attribut	Typ	Beschreibung
cpu_cores	Integer	Anzahl CPU-Kerne, Topologie (big.LITTLE)
cpu_freq_max	Float	Maximale Taktfrequenz (GHz)
gpu_tflops	Float	GPU-Rechenleistung (TFLOP/s)
npu_tops	Float	NPU-Rechenleistung (TOPS)
ram_size	Integer	RAM-Größe (GB)
storage_size	Integer	Speicherkapazität (GB)
virtualization	Enum	none | hypervisor | containers
asil_level	Enum	QM | A | B | C | D
power_idle	Float	Leerlaufleistung (W)
power_max	Float	Maximale Verlustleistung (W)
thermal_max	Float	Zulässige Verlustleistung (W)
power_states	List	Zustände + Übergangszeiten

Energieprofile modellieren den Energieverbrauch über verschiedene Betriebszustände (Power States) mit Übergangszeiten zwischen den Zuständen. Dies ermöglicht eine realistische Abschätzung des Energieverbrauchs in Abhängigkeit von der Lastverteilung.

Power State Modelle

Ein Power State Modell definiert verschiedene Betriebszustände eines Rechenknotens mit zugehörigen Energieverbräuchen und Übergangszeiten:

Tabelle 4.2.: Beispiel: Power States für einen Zonen-Controller

State	Leistung (W)	Übergang zu (ms)	Beschreibung
OFF	0.1	–	Ausgeschaltet
SLEEP	0.5	50	Schlafmodus
IDLE	2.0	20	Leerlauf
ACTIVE_LOW	8.0	10	Niedrige Last
ACTIVE_HIGH	25.0	5	Hohe Last

Die Übergangszeiten sind wichtig für die Simulation, da sie die Reaktionszeit des Systems auf Laständerungen beeinflussen. In PREEvision werden diese als Attribute an Rechenknoten annotiert.

4.1.5. Zonale Topologie in PREEvision

Für eine zonale Architektur werden Zonen-Controller als spezielle Rechenknoten modelliert, die jeweils eine physische Zone des Fahrzeugs (Front, Links, Rechts, Heck) verwalten. Diese sind über einen zentralen Ethernet-Backbone mit TSN-Funktionalität mit der zentralen Rechenplattform verbunden. Die Modellierung erfolgt hierarchisch:

1. **Top-Level**: Fahrzeug als Container für alle Zonen
2. **Zonen-Ebene**: Zonen-Controller mit zugeordneten Sensoren/Aktoren
3. **Backbone-Ebene**: TSN-Switches und Verbindungen
4. **Central Compute**: Zentrale Rechenplattform mit Domain-Controllern

Beispiel: Modellierung einer Front-Zone

Eine Front-Zone in PREEvision könnte folgende Komponenten enthalten:

- **Zonen-Controller (ZC_Front)**:
  - CPU: 4 Kerne @ 1.5 GHz
  - RAM: 2 GB
  - Interfaces: 2x Ethernet (1G), 2x CAN-FD, 1x LIN
  - ASIL: B
  - Power: 8-15 W (abhängig von Last)
- **Sensoren**:
  - Front-Kamera (Bosch 8MP Multifunktionskamera): 3840x2160 @ 30 fps, 120° FOV, 300 m Reichweite, Ethernet 2.5G
  - Front-Radar (Bosch Long-Range): 250 m Reichweite, CAN-FD
  - Side-Radar (Bosch Mid-Range, 2x): 160 m Reichweite, CAN-FD
  - LiDAR (Bosch High-Resolution, optional): 64 Layer, 200 m Reichweite, Ethernet 10G
  - Ultraschall-Sensoren (4x): 5 m Reichweite, LIN
  - Rear-Kamera: 1920x1080 @ 30 fps, Ethernet 1G
- **Aktoren**:
  - Frontscheinwerfer (LED-Matrix): CAN-FD
  - Blinker: LIN

- **Verbindungen**:
  - ZC_Front ↔ TSN-Switch (Ethernet 1G)
  - Sensoren/Aktoren ↔ ZC_Front (CAN/LIN)

Diese Struktur wird in PREEvision als hierarchisches Modell mit Containment-Beziehungen modelliert, wobei jeder Komponente die entsprechenden Attribute zugeordnet werden.

4.2. Kommunikationsmodell

4.2.1. Netzwerksegmente und Protokolle

Das Kommunikationsmodell in PREEvision organisiert die Datenübertragung in *Netzwerksegmenten*, die jeweils ein bestimmtes Protokoll und eine Topologie repräsentieren:

- **Ethernet-Backbone**: Hochgeschwindigkeitsnetzwerk mit TSN-Unterstützung für deterministische Kommunikation. Unterstützt verschiedene Geschwindigkeiten (1G, 2.5G, 10G) und TSN-Features wie Time-Aware Shaping, Credit-Based Shaping und Frame Preemption.
- **CAN/CAN-FD-Netze**: Klassische Fahrzeugbusse für Echtzeitkommunikation mit geringer Latenz. Modelliert als Multi-Drop-Busse mit definierten Bitraten (125 kbit/s bis 2 Mbit/s für CAN-FD).
- **LIN-Netze**: Sub-Bus-Systeme für kostengünstige Sensoren und Aktoren. Master-Slave-Topologie mit definierten Schedules.
- **Legacy-Protokolle**: FlexRay für sicherheitskritische Anwendungen (falls vorhanden).

4.2.2. Routen und Pfade

Routen definieren den Pfad, den Datenpakete durch das Netzwerk nehmen. In einem Ethernet-Backbone mit Switches werden Routen durch die Switch-Konfiguration (Routing-Tabellen, VLANs) bestimmt. PREEvision ermöglicht die Modellierung von:

- **End-to-End-Routen**: Vollständiger Pfad von Quelle zu Senke über mehrere Hops
- **Redundante Pfade**: Alternative Routen für Fehlertoleranz (z. B. PRP, HSR, dual-path)

- **Gateway-Routen**: Übersetzung zwischen verschiedenen Protokollen (z. B. Ethernet zu CAN)

4.2.3. Frames und Signale

Frames (Nachrichten) sind die Grundeinheiten der Datenübertragung. Sie enthalten *Signale*, die die eigentlichen Datenfelder repräsentieren. Die Modellierung umfasst:

Tabelle 4.3.: Frame-Attribute in PREEvision

Attribut	Typ	Beschreibung
frame_id	String	Eindeutige Identifikation
period	Float	Periodizität (ms)
deadline	Float	Deadline (ms)
size	Integer	Frame-Größe (Byte)
priority	Integer	Priorität (0..7 für TSN)
source	Node	Quellknoten
destination	Node[]	Zielknoten (Multicast)
protocol	Enum	Ethernet | CAN | LIN | FlexRay

Signale innerhalb eines Frames werden mit Datentyp (Integer, Float, Enum, Array), Byte-Offset, Bit-Länge und Skalierung/Offset modelliert.

4.2.4. TSN-Konfiguration

Für Time-Sensitive Networking (TSN) werden spezielle Konfigurationsparameter modelliert:

- **Zeitslots (Time Slots)**: Zeitfenster für deterministische Übertragungen im Rahmen eines zyklischen Schedules
- **Gate-Schedules**: Zeitgesteuerte Öffnung/Schließung von Ausgangsports an Switches
- **Prioritäten**: IEEE 802.1Q-Prioritäten (0..7) mit zugeordneten Queues
- **Traffic Shaping**: Credit-Based Shaping (CBS) oder Time-Aware Shaping (TAS) Parameter
- **Synchronisation**: IEEE 802.1AS (gPTP) Konfiguration für Zeitsynchronisation
- **Frame Preemption**: Konfiguration für unterbrechbare Frames (IEEE 802.1Qbu)

Diese Parameter werden in PREEvision als Attribute an Switches, Links und Frames annotiert und ermöglichen eine präzise Modellierung deterministischer Kommunikation.

Beispiel: TSN Gate-Schedule

Ein Gate-Schedule definiert, wann bestimmte Prioritäts-Queues an einem Switch-Port geöffnet oder geschlossen werden. Ein typisches Schedule für einen zyklischen Zeitplan könnte wie folgt aussehen:

Tabelle 4.4.: Beispiel: TSN Gate-Schedule (Zyklus: 1 ms)

Zeit (µs)	Queue 0-3	Queue 4-5	Queue 6-7
0-200	geschlossen	geschlossen	offen
200-400	offen	geschlossen	geschlossen
400-600	geschlossen	offen	geschlossen
600-1000	offen	offen	geschlossen

Dieser Schedule stellt sicher, dass kritische Frames (Queue 6-7) in einem dedizierten Zeitfenster übertragen werden, während andere Queues in anderen Zeitfenstern bedient werden. In PREEvision wird dies als strukturiertes Attribut an Switch-Ports modelliert.

Traffic Shaping Parameter

Für Credit-Based Shaping (CBS) werden folgende Parameter modelliert:

- **idleSlope**: Rate, mit der Credits akkumuliert werden (Byte/s)
- **sendSlope**: Rate, mit der Credits verbraucht werden (Byte/s)
- **hiCredit**: Maximaler Credit-Wert (Byte)
- **loCredit**: Minimaler Credit-Wert (Byte)

Diese Parameter ermöglichen eine präzise Kontrolle der Bandbreitenverteilung für verschiedene Traffic-Klassen.

4.2.5. Gateways und Protokollübersetzung

Gateways verbinden verschiedene Netzwerksegmente und führen Protokollübersetzungen durch. In PREEvision werden Gateways als spezielle Knoten modelliert, die:

- **Mapping-Regeln** definieren: Welche Frames/Signale von einem Protokoll in ein anderes übersetzt werden
- **Signal-Mapping** durchführen: Zuordnung von Signalen zwischen verschiedenen Frame-Definitionen

- **Timing-Anpassungen** berücksichtigen: Unterschiedliche Periodizitäten und Latenzen zwischen Protokollen
- **Redundanz-Handling** implementieren: Behandlung von redundanten Pfaden über verschiedene Protokolle

Beispiel: Ethernet-zu-CAN Gateway

Ein typisches Gateway-Szenario ist die Übersetzung von Ethernet-Frames (DDS/SOME-IP) in CAN-Frames für Legacy-Aktoren. In PREEvision wird dies wie folgt modelliert:

1. **Gateway-Knoten**: Zonen-Controller mit Ethernet- und CAN-Interfaces
2. **Mapping-Regel**:
   - Quelle: Ethernet-Frame `SteeringCommand` (DDS Topic)
   - Ziel: CAN-Frame `0x123` (CAN-ID)
   - Signal-Mapping:
     - `SteeringCommand.angle` → `CAN_0x123.steering_angle` (16 Bit, Skalierung: 0.1°/Bit)
     - `SteeringCommand.velocity` → `CAN_0x123.steering_vel` (8 Bit, Skalierung: 1°/s/Bit)
3. **Timing**:
   - Ethernet: Aperiodisch (Event-triggered)
   - CAN: Periodisch (10 ms)
   - Gateway: Puffert Eingänge und sendet periodisch auf CAN
4. **Latenz**: Gateway-Verarbeitungszeit (typisch 0.5-2 ms)

Diese Mapping-Regeln werden in PREEvision als strukturierte Konfiguration am Gateway-Knoten gespeichert und können für die Simulation extrahiert werden.

4.3. Software- und Deployment-Modell

4.3.1. Software-Komponenten (SWCs)

Software-Komponenten (SWCs) repräsentieren funktionale Einheiten der Software-Architektur. Sie werden in PREEvision mit folgenden Eigenschaften modelliert:

- **Ports und Interfaces**: R-Ports (Required) und P-Ports (Provided) für die Kommunikation zwischen SWCs
- **Runnables**: Ausführbare Funktionen innerhalb einer SWC, die periodisch oder ereignisgesteuert aufgerufen werden
- **Datenabhängigkeiten**: Abhängigkeiten zwischen Runnables und zugehörigen Datenflüssen
- **Ressourcenanforderungen**: CPU-Zeit, Speicher, I/O-Bandbreite

4.3.2. Tasks und Scheduling

Tasks sind die Ausführungseinheiten auf einem Betriebssystem. Sie enthalten eine oder mehrere Runnables und werden nach einer *Scheduling-Policy* ausgeführt:

Tabelle 4.5.: Task-Attribute und Scheduling-Policies

Attribut	Typ	Beschreibung
task_id	String	Eindeutige Identifikation
priority	Integer	Task-Priorität (fixed priority)
period	Float	Periodizität (ms)
deadline	Float	Deadline (ms)
wcet	Float	Worst-Case Execution Time (ms)
bcet	Float	Best-Case Execution Time (ms)
scheduling_policy	Enum	FP | EDF | TSN-triggered
stack_size	Integer	Stack-Größe (Byte)

Scheduling-Policies umfassen:

- **Fixed Priority (FP)**: Präemptives Scheduling mit festen Prioritäten
- **Earliest Deadline First (EDF)**: Dynamisches Scheduling basierend auf Deadlines
- **TSN-triggered**: Task-Ausführung synchronisiert mit TSN-Zeitslots

Beispiel: Task-Konfiguration für Perzeption

Ein typisches Beispiel für die Modellierung von Tasks in einer Perzeptions-Pipeline:

Diese Tasks werden in PREEvision als Teil der Software-Architektur modelliert, wobei die Scheduling-Parameter als Attribute annotiert werden. Die Prioritäten werden so gewählt, dass kritische Tasks (mit früheren Deadlines) höhere Prioritäten erhalten.

Tabelle 4.6.: Beispiel: Task-Konfiguration für Perzeption

Task	Period (ms)	Deadline (ms)	WCET (ms)	Priority	Policy
Camera_Capture	33.3	33.3	2.0	10	FP
Image_Preprocess	33.3	30.0	5.0	9	FP
Object_Detection	33.3	25.0	15.0	8	FP
Tracking	33.3	20.0	3.0	7	FP
Fusion	50.0	45.0	8.0	6	FP

4.3.3. Deployment und Partitionierung

Deployment beschreibt die Zuordnung von Software-Komponenten zu Hardware-Knoten. In PREEvision wird dies durch folgende Konzepte modelliert:

- **Partitionen**: Logische Abgrenzungen innerhalb eines Rechenknotens (z. B. für Safety-Isolation, Virtualisierung)
- **ECU-Zuordnung**: Welche SWCs auf welchem ECU ausgeführt werden
- **Memory-Mapping**: Zuordnung von Variablen und Datenstrukturen zu Speicherbereichen
- **Inter-Partition-Kommunikation**: Mechanismen für die Kommunikation zwischen Partitionen (z. B. Shared Memory, Message Queues)

Beispiel: Deployment auf Central Compute

Ein Central Compute Node könnte mehrere Partitionen für verschiedene Domänen enthalten:

- **Partition 1 (AD-Domain, ASIL D)**:
  - SWCs: Perzeption, Sensorfusion, Planung, Regelung
  - Ressourcen: 4 CPU-Kerne, 8 GB RAM, GPU-Zugriff
  - Isolation: Hypervisor-basiert, keine Kommunikation mit anderen Partitionen
- **Partition 2 (Body-Domain, ASIL B)**:
  - SWCs: Karosserie-Steuerung, Türsteuerung, Lichtsteuerung
  - Ressourcen: 2 CPU-Kerne, 2 GB RAM
  - Isolation: Hypervisor-basiert
- **Partition 3 (Infotainment, QM)**:

  - SWCs: HMI, Audio/Video, Navigation
  - Ressourcen: 2 CPU-Kerne, 4 GB RAM, GPU-Zugriff
  - Isolation: Container-basiert

In PREEvision wird dies als Deployment-Modell modelliert, wobei jede Partition als Container für zugeordnete SWCs dient und Ressourcenlimits definiert werden.

4.3.4. Redundanzmechanismen

Für sicherheitskritische Funktionen werden Redundanzmechanismen modelliert:

- **Redundancy Groups**: Gruppen von redundanten SWCs oder Tasks, die dieselbe Funktion ausführen
- **Voting-Mechanismen**: Vergleich der Ausgaben redundanter Komponenten (z. B. 2-out-of-3 Voting)
- **Umschaltzeiten**: Zeit, die benötigt wird, um von einer ausgefallenen Komponente auf eine redundante umzuschalten
- **Health Monitoring**: Überwachung des Zustands redundanter Komponenten

Beispiel: Redundante Lenkungssteuerung

Für eine ASIL D Lenkungssteuerung könnte folgende Redundanz modelliert werden:

- **Redundancy Group**: `SteeringControl_Redundant`
- **Komponenten**:
  - `SteeringControl_Primary` (ECU 1, Partition 1)
  - `SteeringControl_Secondary` (ECU 2, Partition 1)
  - `SteeringControl_Tertiary` (ECU 3, Partition 1)
- **Voting**: 2-out-of-3 Voting (mindestens 2 identische Ausgaben erforderlich)
- **Health Monitoring**:
  - Heartbeat alle 10 ms
  - Watchdog-Timeout: 50 ms
  - Fehlererkennung: Abweichung der Ausgaben $> 5\%$
- **Umschaltzeit**: 20 ms (Zeit bis zur Erkennung und Umschaltung)

Diese Konfiguration wird in PREEvision als Redundancy Group modelliert, wobei die Voting-Parameter und Umschaltzeiten als Attribute annotiert werden.

4.3.5. Diagnosefunktionen

Diagnosefunktionen werden als spezielle Software-Komponenten modelliert, die:

- **Fehlererkennung** durchführen: Monitoring von Komponenten, Bussen und Kommunikationspfaden
- **Fehlerspeicherung** implementieren: DTCs (Diagnostic Trouble Codes) und Fehlerhistorie
- **Fehlerbehandlung** steuern: Degradationsmodi, Notlauffunktionen, Abschaltungen
- **Diagnose-Protokolle** nutzen: UDS (Unified Diagnostic Services), OBD

4.4. Datenqualität und Exporte

4.4.1. Validierungsprofile

PREEvision bietet *Validierungsprofile*, die die Konsistenz und Vollständigkeit des Modells prüfen. Diese Profile können:

- **Syntax-Validierung** durchführen: Prüfung auf korrekte Modellstruktur und erforderliche Attribute
- **Semantik-Validierung** durchführen: Prüfung auf logische Konsistenz (z. B. alle Referenzen aufgelöst, keine Zyklen in Abhängigkeiten)
- **Completeness-Checks** durchführen: Prüfung, ob alle erforderlichen Informationen vorhanden sind (z. B. WCET für alle Tasks, Bandbreiten für alle Links)
- **Constraint-Validierung** durchführen: Prüfung von Randbedingungen (z. B. E2E-Latenzen, Ressourcenlimits)

4.4.2. Review-Gates

Review-Gates definieren Meilensteine im Entwicklungsprozess, an denen das Modell bestimmte Qualitätskriterien erfüllen muss, bevor die Entwicklung fortgesetzt werden kann. Typische Review-Gates umfassen:

1. **Concept Review**: Vollständigkeit der Topologie und grundlegende Kommunikationspfade definiert

2. **Design Review**: Software-Deployment vollständig, alle Ressourcenanforderungen spezifiziert
3. **Implementation Review**: Alle Timing-Parameter validiert, Redundanzmechanismen spezifiziert
4. **Release Review**: Vollständige Validierung bestanden, Export für nachgelagerte Prozesse freigegeben

4.4.3. Exportprofile

PREEvision unterstützt verschiedene *Exportformate*, die für die Transformation in ein Simulationsmodell genutzt werden können:

Tabelle 4.7.: Exportformate in PREEvision

Format	Beschreibung	Verwendungszweck
REST API	Programmatischer Zugriff	Automatisierte Extraktion
CSV	Tabellarische Daten	Einfache Datenverarbeitung
AUTOSAR XML	Standardisiertes Format	Integration in AUTOSAR-Toolketten
XMI/JSON	Metamodell-basiert	Transformation in andere Modellformate

REST API

Die REST API ermöglicht einen programmatischen Zugriff auf das PREEvision-Modell. Sie bietet Endpunkte für:

- **Topologie-Abfragen**: Abruf von Knoten, Ports, Links und deren Attributen
- **Kommunikations-Abfragen**: Abruf von Frames, Signalen, Routen und TSN-Konfigurationen
- **Software-Abfragen**: Abruf von SWCs, Tasks, Runnables und Deployment-Informationen
- **Validierungs-Ergebnisse**: Abruf von Validierungsergebnissen und Review-Gate-Status

Beispiel: REST API Abfrage Ein typischer API-Aufruf könnte wie folgt aussehen:

```
GET /api/v1/nodes/{nodeId}/attributes
Response: {
```

```
  "nodeId": "ZC_Front",
  "attributes": {
    "cpu_cores": 4,
    "cpu_freq_max": 1.5,
    "ram_size": 2048,
    "asil_level": "B",
    "power_idle": 2.0,
    "power_max": 15.0
  }
}

GET /api/v1/frames?source={nodeId}
Response: {
  "frames": [
    {
      "frameId": "CameraStream_Front",
      "period": 33.3,
      "size": 1500,
      "priority": 6,
      "source": "Camera_Front",
      "destinations": ["ZC_Front", "AD_DC"]
    }
  ]
}
```

Diese API ermöglicht eine automatisierte Extraktion aller für die Simulation relevanten Daten.

CSV-Export

CSV-Exporte ermöglichen eine einfache tabellarische Darstellung von Modellinformationen. Typische Exporte umfassen:

- Knoten-Listen mit Ressourcenprofilen
- Frame-Listen mit Timing- und Prioritätsinformationen
- Task-Listen mit WCET/BCET und Scheduling-Parametern
- Signal-Mappings für Gateway-Konfigurationen

AUTOSAR XML

AUTOSAR XML-Export ermöglicht die Integration in AUTOSAR-basierte Toolketten. Es werden folgende Artefakte exportiert:

- **System Description**: Topologie und Kommunikation
- **Software Component Description**: SWC-Definitionen
- **ECU Configuration**: Deployment und Ressourcen

XMI/JSON-Export

XMI (XML Metadata Interchange) und JSON-Exporte ermöglichen eine modellbasierte Transformation in andere Formate. Sie enthalten:

- **Metamodell-Informationen**: Klassen, Attribute, Beziehungen
- **Instanz-Daten**: Konkrete Modellinstanzen mit allen Attributwerten
- **Versionierungs-Informationen**: Modellversionen und Änderungshistorie

4.4.4. Transformation in Simulationsmodell

Für die Transformation in ein Simulationsmodell werden die Exporte so konfiguriert, dass alle für die Simulation relevanten Informationen enthalten sind:

- **Topologie**: Vollständige Knoten- und Verbindungsstruktur
- **Kommunikation**: Alle Frames, Signale, Routen und TSN-Parameter
- **Software**: SWCs, Tasks, Runnables, Scheduling-Parameter, WCET/BCET
- **Deployment**: Zuordnung von Software zu Hardware, Partitionierung
- **Ressourcen**: CPU/GPU/NPU-Kapazitäten, Speicher, Bandbreiten
- **Timing**: Periodizitäten, Deadlines, Latenzbudgets, Jitter-Budgets
- **Redundanz**: Redundanzgruppen, Umschaltzeiten, Voting-Mechanismen

Diese Informationen bilden die Grundlage für die in Kapitel 6 beschriebene Transformation in ein Simulationsmodell.

4.5. Erweiterte Modellierungs-Beispiele

Dieser Abschnitt präsentiert erweiterte Beispiele für die Modellierung komplexer Architekturen in PREEvision. Die Beispiele zeigen, wie verschiedene Aspekte einer modernen E/E-Architektur modelliert werden können.

4.5.1. Beispiel: Vollständige Zonale Architektur

Dieses Beispiel zeigt die Modellierung einer vollständigen zonalen Architektur mit vier Zonen-Controllern, einem zentralen Rechenknoten und einem TSN-Backbone.

Topologie

Die Topologie umfasst:

- **Zentraler Rechenknoten (Central Compute)**:
  - CPU: 16 Kerne (8x Cortex-A78, 8x Cortex-A55) @ 2.5 GHz
  - GPU: Mali-G78 MP24, 20 TFLOPS
  - NPU: Ethos-N78, 200 TOPS
  - RAM: 16 GB LPDDR5
  - Storage: 256 GB eUFS
  - Interfaces: 4x Ethernet 2.5 Gbps (für Zonen), 1x Ethernet 10 Gbps (für externe Verbindungen)
  - ASIL: D (für sicherheitskritische Partitionen)
- **Zonen-Controller (4x)**:
  - Front-Zone: Position im Front-Bereich, 8x CAN-FD, 4x LIN, 1x Ethernet 1 Gbps
  - Left-Zone: Position links, 6x CAN-FD, 4x LIN, 1x Ethernet 1 Gbps
  - Right-Zone: Position rechts, 6x CAN-FD, 4x LIN, 1x Ethernet 1 Gbps
  - Rear-Zone: Position hinten, 8x CAN-FD, 4x LIN, 1x Ethernet 1 Gbps
  - CPU: 4 Kerne @ 1.5 GHz (für lokale Verarbeitung)
  - RAM: 2 GB
  - ASIL: B (für Gateway-Funktionen)
- **TSN-Switch**:
  - Ports: 8x Ethernet 2.5 Gbps

  - TSN-Features: gPTP, Time-Aware Shaping, Frame Preemption
  - Gate-Scheduling: Konfiguriert für deterministische Kommunikation

Kommunikation

Die Kommunikation wird über einen TSN-Backbone realisiert:

- **Backbone-Topologie**: Stern-Topologie mit TSN-Switch im Zentrum
- **TSN-Konfiguration**:
  - Zykluszeit: 1 ms
  - Gate-Schedules für verschiedene Prioritätsklassen
  - Time-Synchronisation: gPTP mit < 1 µs Genauigkeit
- **Traffic-Klassen**:
  - Priority 7: Kritische sicherheitskritische Frames (Lenkung, Bremse)
  - Priority 6: Wichtige ADAS-Frames (Perzeption, Planung)
  - Priority 4-5: Standard-Frames (Sensor-Daten, Status)
  - Priority 0-3: Best-Effort (Infotainment, Diagnose)

Software-Deployment

Die Software wird auf verschiedene Partitionen verteilt:

- **ASIL D Partition (Central Compute)**:
  - Lenkungssteuerung
  - Bremssteuerung
  - Kollisionsvermeidung
- **ASIL B Partition (Central Compute)**:
  - Perzeption (Objekterkennung, Tracking)
  - Sensorfusion
  - Planung
- **QM Partition (Central Compute)**:
  - Infotainment
  - Navigation

  - Diagnose

- **Zonen-Controller**:
  - Gateway-Funktionen (CAN/LIN zu Ethernet)
  - Lokale Verarbeitung (Türsteuerung, Beleuchtung)
  - Diagnose

4.5.2. Beispiel: Redundante Lenkungsarchitektur

Dieses Beispiel zeigt die Modellierung einer redundanten Lenkungsarchitektur für höchste Sicherheitsanforderungen.

Redundanz-Konzept

Die redundante Architektur umfasst:

- **Primärer Pfad**:
  - Sensor: Front-Kamera (Primär)
  - ECU: AD-DC Partition A
  - Aktor: EPS (Primär)
- **Backup-Pfad**:
  - Sensor: Front-Kamera (Backup) oder Radar
  - ECU: AD-DC Partition B (separate Partition)
  - Aktor: EPS (Backup) oder redundanter Aktor
- **Redundanz-Mechanismus**:
  - Voting: Vergleich der Ausgaben beider Pfade
  - Switchover: Automatischer Wechsel bei Ausfall
  - Switchover-Zeit: < 10 ms
  - Health-Monitoring: Kontinuierliche Überwachung beider Pfade

Modellierung in PREEvision

In PREEvision wird dies wie folgt modelliert:

- **RedundancyGroup**: Definiert die beiden Pfade als Redundanzgruppe
- **Voting-Mechanismus**: Konfiguriert als "2-out-of-2"Voting

- **Switchover-Konfiguration**: Definiert die Bedingungen für den Switchover
- **Health-Monitoring**: Konfiguriert die Überwachungsparameter

4.6. Zusammenfassung und Ausblick

Dieses Kapitel hat die Modellierung einer modernen E/E-Architektur in PREEvision detailliert beschrieben. Die wichtigsten Aspekte umfassen:

- **Topologiemodell**: Hierarchische Strukturierung von Knoten, Ports, Interfaces und Verbindungen mit Ressourcen- und Energieprofilen
- **Kommunikationsmodell**: Netzwerksegmente, Routen, Frames/Signale und TSN-Konfiguration für deterministische Kommunikation
- **Software- und Deployment-Modell**: SWCs, Tasks, Scheduling, Partitionierung, Redundanz und Diagnosefunktionen
- **Datenqualität und Exporte**: Validierungsprofile, Review-Gates und verschiedene Exportformate für die Transformation
- **Erweiterte Beispiele**: Vollständige zonale Architektur und redundante Lenkungsarchitektur als praktische Anwendungsfälle

Die in PREEvision modellierten Daten bilden die Grundlage für die Transformation in ein Simulationsmodell, wie sie in den folgenden Kapiteln beschrieben wird. Die Herausforderung liegt dabei in der korrekten Extraktion und Interpretation aller relevanten Parameter, insbesondere der Timing-Parameter, Ressourcenprofile und Kommunikationskonfigurationen, die für eine realistische Simulation erforderlich sind. Die präsentierten Beispiele zeigen, wie komplexe Architekturen modelliert werden können und welche Aspekte für die Simulation besonders relevant sind.

4.7. Erweiterte Modellierungs-Patterns

Dieser Abschnitt beschreibt erweiterte Modellierungs-Patterns, die für komplexe Architekturen verwendet werden können.

4.7.1. Service-orientierte Modellierung

Service-orientierte Architekturen werden zunehmend in modernen E/E-Architekturen verwendet. In PREEvision können Services wie folgt modelliert werden:

- **Service-Definition**: Services werden als spezielle SWCs modelliert mit:
  - Service-Interface (Request/Response, Event-based)
  - Service-Properties (QoS, Priorität, Deadline)
  - Service-Dependencies (andere Services, die benötigt werden)
- **Service-Registry**: Zentrale Registry für Service-Discovery
- **Service-Binding**: Dynamische oder statische Bindung von Service-Clients zu Service-Providers
- **Service-Versionierung**: Unterstützung für mehrere Service-Versionen

4.7.2. Event-driven Modellierung

Event-driven Architekturen ermöglichen lose gekoppelte Kommunikation:

- **Events**: Events werden als spezielle Frames modelliert
- **Event-Handler**: Event-Handler werden als Tasks modelliert
- **Event-Broker**: Zentrale Event-Broker für Event-Routing
- **Event-Filtering**: Filter-Mechanismen für Event-Selektion

4.7.3. Microservice-Modellierung

Microservices ermöglichen modulare, skalierbare Architekturen:

- **Microservice-Container**: Jeder Microservice läuft in einem eigenen Container
- **Service-Mesh**: Service-Mesh für Kommunikation zwischen Microservices
- **API-Gateway**: Zentrale API-Gateway für externe Zugriffe
- **Load-Balancing**: Load-Balancing für skalierbare Services

4.8. Qualitätssicherung in PREEvision

Qualitätssicherung ist ein kritischer Aspekt der Modellierung. PREEvision bietet verschiedene Mechanismen zur Qualitätssicherung:

4.8.1. Validierungsprofile

Validierungsprofile definieren Regeln für die Modellqualität:

- **Syntax-Validierung**: Prüfung der strukturellen Korrektheit
- **Semantik-Validierung**: Prüfung der logischen Korrektheit
- **Constraint-Validierung**: Prüfung von Randbedingungen
- **Completeness-Validierung**: Prüfung der Vollständigkeit

4.8.2. Review-Gates

Review-Gates definieren Meilensteine für Modell-Reviews:

- **Design-Review**: Review des Architektur-Designs
- **Timing-Review**: Review der Timing-Parameter
- **Safety-Review**: Review der Safety-Aspekte
- **Integration-Review**: Review der Integration

4.8.3. Metriken und KPIs

PREEvision kann verschiedene Metriken berechnen:

- **Komplexitäts-Metriken**: Anzahl Knoten, Links, Frames
- **Last-Metriken**: CPU-Last, Netzwerk-Last
- **Timing-Metriken**: E2E-Latenzen, Jitter
- **Safety-Metriken**: ASIL-Verteilung, Redundanz-Grade

4.9. Erweiterte PREEvision-Modellierungs-Patterns

Dieser Abschnitt beschreibt erweiterte Patterns für die Modellierung in PREEvision.

4.9.1. Pattern: Redundante Architektur

Modellierung von Redundanz

Redundanz kann auf verschiedenen Ebenen modelliert werden:

- **Hardware-Redundanz**: Redundante ECUs, Links, Switches
- **Software-Redundanz**: Redundante SWCs, Tasks
- **Data-Redundanz**: Redundante Datenpfade, Replikation
- **Functional-Redundanz**: Alternative Implementierungen

Redundanz-Mechanismen

Tabelle 4.8.: Redundanz-Mechanismen in PREEvision

Mechanismus	Typ	Switchover-Zeit	Anwendung
Hot-Standby	Hardware	< 1 ms	Kritische Funktionen
Warm-Standby	Hardware	< 10 ms	Wichtige Funktionen
Cold-Standby	Hardware	< 100 ms	Optionale Funktionen
Voting (2-out-of-3)	Software	< 5 ms	Safety-Critical
Voting (2-out-of-2)	Software	< 2 ms	ASIL D

4.9.2. Pattern: Skalierbare Architektur

Modellierung von Skalierbarkeit

Skalierbarkeit kann durch verschiedene Mechanismen erreicht werden:

- **Horizontale Skalierung**: Mehrere Instanzen derselben Komponente
- **Vertikale Skalierung**: Leistungsstärkere Komponenten
- **Load-Balancing**: Verteilung der Last auf mehrere Komponenten
- **Dynamische Skalierung**: Anpassung basierend auf Last

4.9.3. Pattern: Service-orientierte Architektur

Modellierung von Services

Services werden in PREEvision wie folgt modelliert:

- **Service-Definition**: Service-Interface mit Request/Response oder Event-based
- **Service-Provider**: SWC, der den Service bereitstellt
- **Service-Client**: SWC, der den Service nutzt
- **Service-Registry**: Zentrale Registry für Service-Discovery
- **Service-Binding**: Statische oder dynamische Bindung

Service-QoS

Quality of Service (QoS) für Services:

Tabelle 4.9.: Service-QoS-Parameter

Parameter	Typ	Beschreibung	Beispiel
latency	Float	Max. Latenz (ms)	10.0
throughput	Float	Min. Durchsatz (req/s)	100.0
reliability	Float	Verfügbarkeit (%)	99.9
priority	Integer	Priorität	7

5. Synthese-Metrik: Konzeption und Erweiterung

Dieses Kapitel beschreibt die Konzeption und Erweiterung von Synthese-Metriken, die aus Architekturmerkmalen in PREEvision abgeleitet werden, um simulative Parameter für die Performance-Analyse zu generieren. Die Herausforderung liegt darin, aus statischen Architekturmodellen dynamische Simulationsparameter zu synthetisieren, die eine realistische Bewertung des Systemverhaltens ermöglichen.

5.1. Zielsetzung

5.1.1. Grundprinzipien

Die Synthese-Metrik dient der automatisierten Ableitung simulativer Parameter aus Architekturmerkmalen. Ziel ist es, aus den in PREEvision modellierten statischen Informationen dynamische Simulationsparameter zu generieren, die für eine realistische Performance-Analyse erforderlich sind.

Die wichtigsten abzuleitenden Parameter umfassen:

- **Rechenlast**: CPU/GPU/NPU-Auslastung basierend auf Task-WCETs, Periodizitäten und Scheduling-Parametern
- **Netzwerklast**: Bandbreitenauslastung, Paketraten, Queueing-Verhalten basierend auf Frame-Parametern und Routen
- **Timingpfade**: End-to-End-Latenzen entlang von Funktionsketten von Sensoren zu Aktoren
- **Verfügbarkeitsprofile**: MTBF (Mean Time Between Failures), Ausfallraten, Degradationsmodi
- **Energieprofile**: Energieverbrauch über verschiedene Betriebszustände und Lastszenarien

5.1.2. Anforderungen an die Metrik

Die Synthese-Metrik muss folgenden Anforderungen genügen:

1. **Vollständigkeit**: Alle für die Simulation relevanten Parameter müssen ableitbar sein
2. **Konsistenz**: Abgeleitete Parameter müssen konsistent mit den Architekturmerkmalen sein
3. **Validierbarkeit**: Die Metrik muss gegen bekannte analytische Modelle validierbar sein
4. **Erweiterbarkeit**: Neue Architekturmerkmale müssen einfach integrierbar sein
5. **Nachvollziehbarkeit**: Die Ableitung muss nachvollziehbar und dokumentiert sein

5.2. Erweiterungen

5.2.1. Flow-Aggregation von Nachrichtenströmen

Ein zentraler Aspekt ist die Aggregation einzelner Nachrichtenströme zu Funktionsketten (Chains). Eine Funktionskette repräsentiert einen End-to-End-Datenfluss von einem Sensor über Verarbeitungsschritte bis zu einem Aktor.

Chain-Identifikation

Funktionsketten werden durch Analyse der Datenabhängigkeiten identifiziert:

1. **Startpunkte**: Sensoren oder externe Eingaben
2. **Verarbeitungsschritte**: SWCs, Tasks, Runnables, die Daten transformieren
3. **Kommunikationsschritte**: Frames, Signale, die Daten zwischen Komponenten übertragen
4. **Endpunkte**: Aktoren oder externe Ausgaben

Chain-Attribute

Jede identifizierte Chain erhält folgende Attribute:

Tabelle 5.1.: Chain-Attribute für die Simulation

Attribut	Typ	Beschreibung
chain_id	String	Eindeutige Identifikation
source	Node	Startknoten (Sensor)
sink	Node	Endknoten (Aktor)
path	Node[]	Sequenz von Verarbeitungsknoten
e2e_deadline	Float	End-to-End-Deadline (ms)
e2e_latency_budget	Float	Latenzbudget (ms)
e2e_jitter_budget	Float	Jitter-Budget (ms)
asil_level	Enum	Safety-Level der Chain

Beispiel: Perzeption-zu-Aktorik Chain

Eine typische Chain für eine Fahrerassistenzfunktion wird im Detail analysiert, um die Komplexität und die Anforderungen zu verdeutlichen:

- **Chain-ID**: `Perception_to_Steering`
- **Beschreibung**: Diese Chain implementiert eine L2-Fahrerassistenzfunktion (Lane Keeping Assist), die kontinuierlich die Fahrspur überwacht und bei Bedarf lenkt.
- **Startpunkt**: Front-Kamera (Mono-Kamera, 1920x1080, 30 fps)
  - Position: Frontscheibe, hinter dem Rückspiegel
  - Sensor-Typ: CMOS-Sensor mit HDR-Unterstützung
  - Datenrate: $1920 \times 1080 \times 3$ Bytes $\times$ 30 fps = 186.6 MB/s (roh), komprimiert auf 12 MB/s
  - Interface: Ethernet 1 Gbps
  - ASIL-Level: QM (Sensor selbst), aber Chain ist ASIL D
- **Verarbeitungsschritte**:
  1. **Bildaufnahme (Kamera)**:
     - Task: Image_Capture
     - WCET: 2.0 ms
     - Period: 33.3 ms (30 fps)
     - ECU: Kamera-Modul (lokale Verarbeitung)
  2. **Bildvorverarbeitung (ZC_Front)**:
     - Task: Image_Preprocessing

   - Funktionen: Entzerrung, Normalisierung, ROI-Extraktion
   - WCET: 5.0 ms
   - Period: 33.3 ms
   - ECU: Zonen-Controller Front
   - Output: Vorverarbeitetes Bild (reduzierte Auflösung: 640x480)
3. **Objekterkennung (AD-DC)**:
   - Task: Object_Detection
   - Algorithmus: CNN-basiert (YOLO-ähnlich)
   - WCET: 15.0 ms (auf GPU)
   - Period: 33.3 ms
   - ECU: AD Domain Controller (GPU-Partition)
   - Output: Erkannte Objekte mit Bounding Boxes und Klassifikation
4. **Tracking (AD-DC)**:
   - Task: Object_Tracking
   - Algorithmus: Kalman-Filter + Data Association
   - WCET: 3.0 ms
   - Period: 33.3 ms
   - ECU: AD Domain Controller (CPU-Partition)
   - Output: Getrackte Objekte mit Trajektorien
5. **Sensorfusion (AD-DC)**:
   - Task: Sensor_Fusion
   - Input: Kamera-Daten + Radar-Daten (von anderer Chain)
   - WCET: 4.0 ms
   - Period: 33.3 ms
   - ECU: AD Domain Controller
   - Output: Fused Object List mit erhöhter Zuverlässigkeit
6. **Planung (AD-DC)**:
   - Task: Trajectory_Planning
   - Algorithmus: A*-basierte Pfadplanung
   - WCET: 8.0 ms

  - Period: 50.0 ms (20 Hz, niedrigere Rate als Perzeption)
  - ECU: AD Domain Controller
  - Output: Geplanter Pfad (Trajektorie)
7. **Regelung (AD-DC)**:
  - Task: Steering_Control
  - Algorithmus: PID-Regler mit Feedforward
  - WCET: 1.0 ms
  - Period: 10.0 ms (100 Hz für präzise Regelung)
  - ECU: AD Domain Controller
  - Output: Lenkwinkel-Kommando

- **Kommunikation**:
  - **Kamera → ZC_Front**:
    * Protokoll: Ethernet 1 Gbps
    * Frame-Größe: 1500 Byte (MTU)
    * Periodizität: 33.3 ms (30 fps)
    * Priorität: TSN Priority 6 (hoch)
    * Latenz-Budget: 5 ms
  - **ZC_Front → AD-DC**:
    * Protokoll: Ethernet/TSN 2.5 Gbps
    * Frame-Größe: 500 Byte (komprimiertes Bild)
    * Periodizität: 33.3 ms
    * Priorität: TSN Priority 6
    * Route: Über TSN-Switch mit Gate-Scheduling
    * Latenz-Budget: 10 ms
  - **AD-DC → Lenkaktor**:
    * Protokoll: CAN-FD 500 kbps
    * Frame-Größe: 8 Byte (Lenkwinkel-Kommando)
    * Periodizität: 10 ms
    * CAN-ID: 0x123 (hohe Priorität)
    * Latenz-Budget: 2 ms

- **Endpunkt**: Lenkaktor (EPS - Electric Power Steering)
  - Typ: Elektrische Servolenkung
  - Interface: CAN-FD
  - ASIL-Level: D
  - Reaktionszeit: < 50 ms
- **E2E-Deadline**: 100 ms
  - Begründung: Für L2-Funktionen ist eine Reaktionszeit < 100 ms erforderlich, um sicher auf Hindernisse reagieren zu können
  - Latenz-Budget-Zerlegung:
    - Sensor-Verarbeitung: 2 ms
    - Netzwerk (Kamera $\rightarrow$ ZC): 5 ms
    - ZC-Verarbeitung: 5 ms
    - Netzwerk (ZC $\rightarrow$ AD-DC): 10 ms
    - AD-DC-Verarbeitung: 30 ms (kritischer Pfad: Objekterkennung 15 ms + Planung 8 ms + Regelung 1 ms)
    - Netzwerk (AD-DC $\rightarrow$ Aktor): 2 ms
    - Aktor-Reaktion: 50 ms
    - **Gesamt (Worst-Case)**: 104 ms (knapp über Budget, erfordert Optimierung)
- **ASIL-Level**: D
  - Begründung: Lenkung ist sicherheitskritisch und erfordert höchstes Safety-Level
  - Konsequenzen: Redundanz erforderlich, formale Verifikation, umfassende Tests

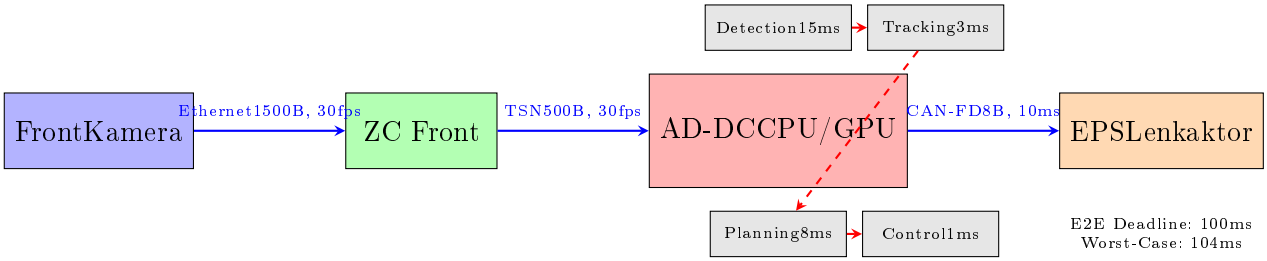

Abbildung 5.1.: Beispiel: Perzeption-zu-Aktorik Chain mit Timing-Details

5.2.2. Lastabschätzung

Rechenlast

Die Rechenlast wird basierend auf Task-Parametern abgeschätzt:

$$U_{CPU} = \sum_{i=1}^{n} \frac{WCET_i}{T_i} \tag{5.1}$$

wobei:

- U_{CPU}: CPU-Auslastung
- $WCET_i$: Worst-Case Execution Time von Task i
- T_i: Periodizität von Task i
- n: Anzahl der Tasks

Für GPU/NPU wird die Last basierend auf Datenrate und Algorithmenkomplexität abgeschätzt:

$$L_{GPU} = \frac{D \times C}{P_{GPU}} \tag{5.2}$$

wobei:

- L_{GPU}: GPU-Last
- D: Datenrate (z. B. Pixel/s)
- C: Algorithmenkomplexität (Operationen pro Pixel)
- P_{GPU}: GPU-Performance (Operationen/s)

Netzwerklast

Die Netzwerklast wird basierend auf Frame-Parametern berechnet:

$$B_{link} = \sum_{i=1}^{m} \frac{S_i \times 8}{T_i} \tag{5.3}$$

wobei:

- B_{link}: Bandbreitenauslastung (bit/s)
- S_i: Frame-Größe i (Byte)
- T_i: Periodizität von Frame i (s)

- m: Anzahl der Frames auf dem Link

Für TSN-Netze wird zusätzlich der Overhead durch Protokoll-Header und Guard-Bands berücksichtigt.

5.2.3. Netzmodelle

Bandbreitenmodell

Das Bandbreitenmodell berücksichtigt:

- **Nominal-Bandbreite**: Theoretische maximale Datenrate des Links
- **Effektive Bandbreite**: Tatsächlich verfügbare Bandbreite nach Abzug von Overhead
- **TSN-Overhead**: Guard-Bands, Protokoll-Header, Synchronisations-Traffic
- **Queueing**: Warteschlangen an Switch-Ports

Latenz- und Jitter-Modell

Die Latenz setzt sich zusammen aus:

$$L_{total} = L_{processing} + L_{transmission} + L_{propagation} + L_{queueing} \tag{5.4}$$

wobei:

- $L_{processing}$: Verarbeitungslatenz (Switch, Gateway)
- $L_{transmission}$: Übertragungslatenz (Frame-Größe / Bandbreite)
- $L_{propagation}$: Ausbreitungslatenz (Kabel, typisch vernachlässigbar)
- $L_{queueing}$: Warteschlangenlatenz (abhängig von Auslastung)

Der Jitter wird durch Variationen in der Queueing-Latenz verursacht und hängt von der Netzauslastung ab.

TSN-Prioritäten und Scheduling

Für TSN-Netze werden die Prioritäten und Gate-Schedules in das Simulationsmodell übernommen:

- **Prioritäts-Mapping**: IEEE 802.1Q-Prioritäten (0..7) werden auf Switch-Queues abgebildet

- **Gate-Schedules**: Zeitgesteuerte Öffnung/Schließung von Queues
- **Traffic Shaping**: Credit-Based Shaping oder Time-Aware Shaping Parameter

5.2.4. Ausfall- und Degradationsprofile

Ausfallraten

Ausfallraten werden basierend auf Komponententypen und ASIL-Leveln modelliert:

Tabelle 5.2.: Beispiel: Ausfallraten (MTBF) für verschiedene Komponententypen

Komponententyp	ASIL	MTBF (h)
ECU (Standard)	QM	10.000
ECU (Safety)	B	50.000
ECU (Safety)	C	100.000
ECU (Safety)	D	200.000
Sensor (Kamera)	B	20.000
Aktor (EPS)	D	100.000
Switch (TSN)	B	30.000

Degradationsmodi

Degradationsmodi beschreiben, wie das System bei Ausfällen reagiert:

- **Graceful Degradation**: System reduziert Funktionalität, bleibt aber betriebsfähig
- **Fail-Safe**: System geht in einen sicheren Zustand über
- **Redundanz-Umschaltung**: Automatische Umschaltung auf redundante Komponente
- **Notlaufmodus**: Eingeschränkte Funktionalität mit reduzierter Performance

5.2.5. Energieprofile über Duty-Cycles

Energieprofile werden basierend auf Power States und Duty-Cycles modelliert:

$$E_{total} = \sum_{i=1}^{k} P_i \times t_i \tag{5.5}$$

wobei:

- E_{total}: Gesamtenergieverbrauch

- P_i: Leistung im Zustand i
- t_i: Zeit im Zustand i
- k: Anzahl der Power States

Der Duty-Cycle beschreibt, wie viel Zeit ein Knoten in jedem Power State verbringt, abhängig von der Lastverteilung.

5.3. Validierung

5.3.1. Analytische Bounds

Die abgeleiteten Parameter werden gegen analytische Bounds validiert:

CPU-Auslastung

Für Fixed-Priority Scheduling gilt:

$$U_{CPU} \leq n(2^{1/n} - 1) \tag{5.6}$$

wobei n die Anzahl der Tasks ist. Diese Bound stellt sicher, dass alle Deadlines eingehalten werden können.

Netzwerk-Latenz

Für TSN-Netze kann eine obere Schranke für die Latenz berechnet werden:

$$L_{max} = L_{min} + \frac{S_{max}}{B} + Q_{max} \tag{5.7}$$

wobei:

- L_{min}: Minimale Latenz (ohne Queueing)
- S_{max}: Maximale Frame-Größe
- B: Bandbreite
- Q_{max}: Maximale Queueing-Latenz

5.3.2. Microbenchmarks

Microbenchmarks werden verwendet, um einzelne Komponenten der Metrik zu validieren:

- **Task-Scheduling**: Vergleich simulierter vs. gemessener Ausführungszeiten
- **Netzwerk-Latenz**: Vergleich simulierter vs. gemessener Latenzen
- **Energieverbrauch**: Vergleich simulierter vs. gemessener Energieverbräuche

5.3.3. Sensitivitätsanalysen

Sensitivitätsanalysen untersuchen, wie empfindlich die Simulationsergebnisse auf Änderungen der abgeleiteten Parameter reagieren:

- **WCET-Variation**: Wie ändern sich Ergebnisse bei ±20% WCET-Variation?
- **Bandbreiten-Variation**: Wie ändern sich Latenzen bei ±10% Bandbreiten-Variation?
- **Periodizitäts-Variation**: Wie ändern sich Ergebnisse bei Änderungen der Frame-Periodizitäten?

Diese Analysen helfen, die Robustheit der Metrik zu bewerten und kritische Parameter zu identifizieren.

5.4. Erweiterte Beispiele für Synthese-Metriken

Dieser Abschnitt präsentiert erweiterte Beispiele für die Anwendung der Synthese-Metriken auf komplexe Architekturen.

5.4.1. Beispiel: Multi-Domain-Architektur

Dieses Beispiel zeigt die Synthese-Metrik für eine Multi-Domain-Architektur mit mehreren Domänen (AD, Body, Infotainment).

Architektur-Übersicht

Die Architektur umfasst:

- **AD-Domain**:
  - 1x AD-DC (Central Compute) mit 16 CPU-Kernen, GPU, NPU
  - 8x Sensoren (Kameras, Radar, LiDAR)
  - 2x Aktoren (Lenkung, Bremse)
  - 15x Funktionsketten (Perzeption, Planung, Regelung)

- **Body-Domain**:
  - 4x Zonen-Controller
  - 20x Aktoren (Türen, Beleuchtung, Klima)
  - 30x Sensoren (Türkontakte, Temperatur, etc.)
  - 25x Funktionsketten (Türsteuerung, Beleuchtung, etc.)
- **Infotainment-Domain**:
  - 1x HMI-ECU
  - 2x Displays
  - 1x Audio-System
  - 10x Funktionsketten (Navigation, Media, etc.)

Lastabschätzung

Die Lastabschätzung für die gesamte Architektur:

Tabelle 5.3.: Lastabschätzung: Multi-Domain-Architektur

Domain	CPU-Last	GPU-Last	Netzwerk-Last
AD	68%	72%	58%
Body	45%	0%	35%
Infotainment	55%	30%	40%
Gesamt	**56%**	**34%**	**44%**

Timing-Analyse

Die Timing-Analyse für kritische Chains:

Tabelle 5.4.: Timing-Analyse: Kritische Chains

Chain	Domain	E2E-Latenz (Mean)	E2E-Latenz (Max)	Deadline
Notbremsung	AD	42 ms	78 ms	100 ms
Lenkung	AD	38 ms	72 ms	150 ms
Türöffnung	Body	25 ms	45 ms	200 ms
Navigation	Infotainment	120 ms	250 ms	500 ms

5.4.2. Beispiel: Skalierungs-Analyse

Dieses Beispiel zeigt, wie die Synthese-Metrik für Skalierungs-Analysen verwendet wird.

Szenario

Verschiedene Architektur-Varianten werden verglichen:

- **Variante A**: Basis-Architektur (4 Zonen, 1 Central Compute)
- **Variante B**: Erweiterte Architektur (6 Zonen, 2 Central Compute)
- **Variante C**: Maximale Architektur (8 Zonen, 3 Central Compute)

Vergleich

Tabelle 5.5.: Skalierungs-Analyse: Architektur-Varianten

Variante	CPU-Last	Netzwerk-Last	E2E-Latenz	Kosten
A (Basis)	75%	65%	95 ms	1.0x
B (Erweitert)	55%	45%	72 ms	1.5x
C (Maximal)	40%	30%	58 ms	2.2x

Die Analyse zeigt, dass Variante B einen guten Kompromiss zwischen Performance und Kosten bietet.

5.5. Erweiterte Metrik-Formeln und Berechnungen

Dieser Abschnitt präsentiert erweiterte Formeln und Berechnungen für die Synthese-Metrik.

5.5.1. Erweiterte Timing-Berechnungen

Response-Time-Analyse für Multi-Core

Für Multi-Core-Systeme müssen zusätzliche Aspekte berücksichtigt werden:

$$R_i^{multi} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + I_i^{inter-core} \tag{5.8}$$

wobei $I_i^{inter-core}$ die Inter-Core-Interferenz berücksichtigt:

$$I_i^{inter-core} = \sum_{k \in other-cores} \left(\frac{C_k}{T_k} \times L_{cache} \right) \tag{5.9}$$

wobei L_{cache} die Cache-Miss-Latenz ist.

TSN-End-to-End-Latenz

Die End-to-End-Latenz in TSN-Netzen setzt sich zusammen aus:

$$L_{E2E}^{TSN} = \sum_{h=1}^{H} \left(L_{tx}^{h} + L_{sw}^{h} + L_{gate}^{h} + L_{queue}^{h} \right) \tag{5.10}$$

wobei:

- H: Anzahl der Hops
- L_{tx}^{h}: Übertragungslatenz an Hop h
- L_{sw}^{h}: Switch-Verarbeitungslatenz an Hop h
- L_{gate}^{h}: Gate-Latenz an Hop h (abhängig vom Gate-Schedule)
- L_{queue}^{h}: Warteschlangenlatenz an Hop h

5.5.2. Erweiterte Last-Berechnungen

GPU-Last mit Tensor-Operationen

Für GPU-Last mit Tensor-Operationen:

$$L_{GPU}^{tensor} = \frac{\sum_{i=1}^{N} (D_i \times O_i)}{P_{GPU}^{tensor}} \tag{5.11}$$

wobei:

- D_i: Datenmenge für Operation i
- O_i: Anzahl Operationen pro Datenelement für Operation i
- P_{GPU}^{tensor}: Tensor-Performance der GPU (TOPS)
- N: Anzahl der Operationen

GPU-Last für KI-Inferenz (NVIDIA DRIVE Thor)

Für GPU-Last bei KI-Inferenz auf NVIDIA DRIVE Thor:

$$L_{GPU}^{Thor} = \frac{M \times FLOPS_{model}}{P_{Thor}} \tag{5.12}$$

wobei:

- M: Anzahl der Inferenz-Anfragen pro Sekunde

- $FLOPS_{model}$: FLOPS pro Inferenz für das KI-Modell
- P_{Thor}: NVIDIA DRIVE Thor GPU-Performance (2000 TOPS)

Beispiel: Bosch 8MP Kamera mit NVIDIA DRIVE Thor Für eine Bosch 8MP Multifunktionskamera (3840x2160 @ 30 fps) mit YOLOv8 auf NVIDIA DRIVE Thor:

- Input: 3840x2160 Pixel @ 30 fps
- YOLOv8 FLOPS: 65 GFLOPs pro Frame
- DRIVE Thor Performance: 2000 TOPS = 2000 $\times 10^{12} FLOPS/s$
- Last: $L = \frac{30 \times 65 \times 10^9}{2000 \times 10^{12}} = 0.975\%$
- Inferenz-Zeit: 15 ms (basierend auf Benchmarks)

5.5.3. Energie-Berechnungen

Dynamischer Energieverbrauch

Der dynamische Energieverbrauch hängt von der Last ab:

$$E_{dyn} = \sum_{i=1}^{N} \left(C_{eff} \times V_{dd}^2 \times f_i \times U_i \times t_i \right) \tag{5.13}$$

wobei:

- C_{eff}: Effektive Kapazität
- V_{dd}: Versorgungsspannung
- f_i: Frequenz in Zustand i
- U_i: Auslastung in Zustand i
- t_i: Zeit in Zustand i
- N: Anzahl der Zustände

Leckstrom-Energie

Der Leckstrom-Energieverbrauch:

$$E_{leak} = V_{dd} \times I_{leak} \times t_{total} \tag{5.14}$$

wobei I_{leak} der Leckstrom ist, der temperaturabhängig ist.

5.5.4. Erweiterte Energie-Modellierung

Temperaturabhängigkeit

Der Energieverbrauch hängt stark von der Temperatur ab:

$$P_{total}(T) = P_{dyn} + P_{leak}(T) + P_{static} \tag{5.15}$$

wobei der Leckstrom temperaturabhängig ist:

$$I_{leak}(T) = I_0 \times e^{\frac{E_a}{k_B T}} \tag{5.16}$$

mit:

- E_a: Aktivierungsenergie (typisch 0.3-0.5 eV)
- k_B: Boltzmann-Konstante
- T: Temperatur in Kelvin

DVFS-Modellierung

Dynamic Voltage and Frequency Scaling (DVFS) ermöglicht Energie-Optimierung:

$$P_{DVFS}(f, V) = C_{eff} \times V^2 \times f + P_{static} \tag{5.17}$$

wobei:

- f: Frequenz
- V: Spannung
- C_{eff}: Effektive Kapazität

Die optimale Frequenz für minimale Energie bei gegebener Last:

$$f_{opt} = \sqrt{\frac{L}{C_{eff} \times V^2}} \tag{5.18}$$

5.5.5. Erweiterte Timing-Analyse

Probabilistische Timing-Analyse

Nicht nur Worst-Case, sondern auch probabilistische Verteilungen:

$$P(T_{response} > D) = \int_D^\infty f_{response}(t)dt \tag{5.19}$$

wobei $f_{response}(t)$ die Wahrscheinlichkeitsdichtefunktion der Response-Zeit ist.

Multi-Core-Interferenz

Für Multi-Core-Systeme muss Cache-Interferenz berücksichtigt werden:

$$R_i^{multi} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + I_i^{cache} + I_i^{memory} \tag{5.20}$$

wobei:

- I_i^{cache}: Cache-Interferenz durch andere Cores
- I_i^{memory}: Memory-Interferenz durch andere Cores

5.6. Erweiterte Metrik-Beispiele

Dieser Abschnitt präsentiert detaillierte Beispiele für die Anwendung der Synthese-Metrik.

5.6.1. Beispiel: Komplexe Multi-Domain-Architektur

Dieses Beispiel zeigt die Anwendung der Synthese-Metrik auf eine komplexe Multi-Domain-Architektur:

Architektur-Übersicht

- **AD-Domain**: 1x AD-DC, 4x Zonen-Controller, 12x Sensoren, 3x Aktoren
- **Body-Domain**: 1x Body-DC, 2x Zonen-Controller, 20x Sensoren, 15x Aktoren
- **Infotainment-Domain**: 1x Infotainment-DC, 1x Zonen-Controller, 5x Sensoren, 3x Aktoren
- **Kommunikation**: TSN-Backbone (10 Gbps), CAN-FD für Aktoren

Lastabschätzung

Die Lastabschätzung ergab:

Timing-Analyse

Die Timing-Analyse ergab:

- **WCRT (AD-Tasks)**: 25 ms (kritischste Tasks)
- **TSN-Latenz**: 0.5 ms (durchschnittlich)

Tabelle 5.6.: Lastabschätzung: Multi-Domain-Architektur

Domain	CPU-Last	GPU-Last	Netzwerk-Last	Energie
AD	65%	55%	45%	120 W
Body	35%	5%	20%	40 W
Infotainment	50%	30%	35%	60 W
Gesamt	75%	60%	70%	220 W

- **E2E-Latenz**: 85 ms (kritischste Chains)
- **Deadline-Misses**: 0% (alle Anforderungen erfüllt)

5.7. Zusammenfassung

Dieses Kapitel hat die Konzeption und Erweiterung von Synthese-Metriken beschrieben, die aus Architekturmerkmalen simulative Parameter ableiten. Die wichtigsten Aspekte umfassen:

- **Flow-Aggregation**: Identifikation und Modellierung von Funktionsketten
- **Lastabschätzung**: Berechnung von Rechen- und Netzwerklast
- **Netzmodelle**: Modellierung von Bandbreite, Latenz, Jitter und TSN-Parametern
- **Ausfall- und Degradationsprofile**: Modellierung von Fehlerverhalten
- **Energieprofile**: Modellierung des Energieverbrauchs über Duty-Cycles
- **Validierung**: Analytische Bounds, Microbenchmarks und Sensitivitätsanalysen

Diese Metriken bilden die Grundlage für die Transformation in ein Simulationsmodell, wie sie in den folgenden Kapiteln beschrieben wird.

6. Transformationskonzept zum Simulationsmodell

Dieses Kapitel beschreibt das Konzept für die Transformation eines Architekturmodells aus PREEvision in ein Simulationsmodell. Die Transformation muss alle relevanten Aspekte der Architektur (Topologie, Kommunikation, Software, Timing) korrekt in die Semantik der Zielplattform überführen, sodass eine realistische Performance-Analyse möglich ist. Die Transformation basiert auf etablierten Methoden der modellbasierten Entwicklung [?, ?] und modernen Ansätzen zur Netzwerksimulation [?].

Die Transformation muss insbesondere die neuesten Sensoren (Bosch 8MP Multifunktionskamera, Bosch Radar, Bosch LiDAR) und Rechenplattformen (NVIDIA DRIVE Thor) korrekt abbilden, um realistische Simulationsergebnisse zu erzielen. Diese Komponenten stellen neue Anforderungen an die Transformationsregeln und erfordern spezielle Mapping-Strategien.

Die Auswahl geeigneter Simulationsplattformen ist entscheidend für die Qualität der Ergebnisse. Verschiedene Plattformen bieten unterschiedliche Stärken für die Modellierung von E/E-Architekturen [?], wobei moderne Cloud-basierte Ansätze [?] zunehmend an Bedeutung gewinnen.

6.1. Zielplattformen

6.1.1. Übersicht

Verschiedene Simulationsplattformen bieten unterschiedliche Stärken für die Modellierung von E/E-Architekturen. Die Auswahl hängt von den spezifischen Anforderungen ab:

OMNeT++/INET

OMNeT++ ist ein diskretes Ereignis-Simulationsframework mit spezialisierten Erweiterungen für Netzwerksimulation:

- **Stärken**:

  - Detaillierte Netzwerksimulation (Ethernet, TSN, CAN)
  - Modulares Komponentenmodell
  - Gute Performance für große Netze
  - Umfangreiche Analyse-Tools
- **Schwächen**:
  - Begrenzte CPU-Scheduling-Modellierung
  - Komplexere Einarbeitung
- **Einsatz**: Primär für Kommunikationsanalyse, TSN-Simulation

NS-3

NS-3 ist ein Open-Source-Netzwerksimulator:

- **Stärken**:
  - Realistische Netzwerkprotokolle
  - Gute TSN-Unterstützung
  - Aktive Community
- **Schwächen**:
  - Begrenzte CPU-Modellierung
  - Komplexere Konfiguration
- **Einsatz**: Alternative zu OMNeT++ für Netzwerksimulation

Simulink/TrueTime

Simulink mit TrueTime-Toolbox bietet co-simulative Modellierung:

- **Stärken**:
  - Integrierte CPU-Scheduling-Simulation
  - Kontinuierliche und diskrete Simulation
  - Gute Integration mit Control-Design-Tools
- **Schwächen**:
  - Begrenzte Netzwerkdetaillierung
  - Proprietär (MathWorks)
- **Einsatz**: Für CPU-Scheduling-Analyse, Control-Loops

Modelica

Modelica ist eine objektorientierte Modellierungssprache für physikalische Systeme:

- **Stärken**:
  - Multi-Domain-Modellierung (Elektrik, Mechanik, Thermodynamik)
  - Akausale Modellierung
  - Gute Energie-Modellierung
- **Schwächen**:
  - Begrenzte Echtzeit-Scheduling-Modellierung
  - Komplexere Netzwerkmodellierung
- **Einsatz**: Für Energie-Analyse, Multi-Domain-Simulation

ROS 2/DDS

ROS 2 mit DDS-Middleware bietet verteilte Systemsimulation:

- **Stärken**:
  - Realistische Middleware-Simulation
  - Gute Integration mit Robotik-Tools
  - Service-orientierte Kommunikation
  - Moderne DDS-Implementierungen (FastDDS, CycloneDDS, RTI Connext)
  - Unterstützung für deterministische Kommunikation
- **Schwächen**:
  - Begrenzte Timing-Genauigkeit
  - Komplexere Einrichtung
- **Einsatz**: Für Software-Architektur-Validierung

Cloud-basierte Simulationsplattformen

Moderne Cloud-basierte Simulationsplattformen bieten Skalierung und Flexibilität:

- **AWS RoboMaker / AWS IoT Device Simulator**:
  - Skalierbare Cloud-Simulationen für E/E-Architekturen
  - Integration mit AWS-Services (S3, Lambda, CloudWatch)

  - Pay-per-use Modell für kosteneffiziente Simulationen
  - Unterstützung für ROS 2 und Gazebo

- **Microsoft Azure Digital Twins**:
  - Digitale Zwillinge für E/E-Architekturen
  - Integration mit Azure IoT Hub und Edge
  - Real-time Analytics mit Azure Stream Analytics
  - Graph-basierte Modellierung mit Digital Twins Definition Language (DTDL)

- **Google Cloud Simulation Platform**:
  - Kubernetes-basierte Skalierung für parallele Simulationen
  - Integration mit TensorFlow für ML-basierte Simulationen
  - BigQuery für große Ergebnis-Datensätze
  - Vertex AI für AI/ML-Workloads

- **Edge-Cloud-Hybrid**:
  - Edge Computing für Echtzeit-Simulationen (NVIDIA Jetson, Intel Edge)
  - Cloud für komplexe Analysen und Batch-Processing
  - Federated Learning für verteilte ML-Modelle
  - 5G-Integration für Low-Latency-Kommunikation

FMI-Co-Simulation

Functional Mock-up Interface (FMI) ermöglicht Co-Simulation verschiedener Tools und ist ein Standard für die Integration heterogener Simulationsmodelle. FMI ist besonders wertvoll für E/E-Architekturen, da verschiedene Aspekte (Netzwerk, CPU, Energie, Physik) oft in verschiedenen Tools modelliert werden.

FMI 2.0 und 3.0 FMI 2.0 bietet:

- **FMU (Functional Mock-up Unit)**: Standardisierte Einheit für Simulationsmodelle
- **Co-Simulation**: Koordinierte Ausführung mehrerer FMUs
- **Model Exchange**: Austausch von Modellen zwischen Tools
- **Unabhängigkeit**: FMUs sind unabhängig vom Erstellungs-Tool

FMI 3.0 erweitert dies um:

- **Clocked Co-Simulation**: Synchronisierte Co-Simulation mit Takt-Signalen
- **Scheduled Execution**: Geplante Ausführung für deterministische Simulationen
- **Structured I/O**: Strukturierte Ein- und Ausgaben für komplexe Daten
- **Binary Distribution**: Effiziente binäre Verteilung von FMUs

Anwendung in E/E-Architekturen FMI-Co-Simulation ermöglicht die Integration verschiedener Simulationsmodelle:

- **Netzwerk-Simulation** (OMNeT++): Modelliert Ethernet/TSN-Kommunikation
- **CPU-Simulation** (Simulink/TrueTime): Modelliert Task-Scheduling und Verarbeitung
- **Energie-Simulation** (Modelica): Modelliert Energieverbrauch und Wärmeentwicklung
- **Physik-Simulation** (CarSim, IPG CarMaker): Modelliert Fahrzeugdynamik

Die Co-Simulation koordiniert die Ausführung dieser Modelle und ermöglicht eine realistische Simulation des gesamten Systems.

Beispiel: FMI-Co-Simulation Setup Ein typisches Setup für eine E/E-Architektur-Simulation:

```
# FMI Co-Simulation Konfiguration
fmus:
  - name: network_sim
    type: co-simulation
    path: models/network.fmu
    step_size: 0.001  # 1 ms

  - name: cpu_sim
    type: co-simulation
    path: models/cpu.fmu
    step_size: 0.0001  # 0.1 ms

  - name: energy_sim
```

```
    type: co-simulation
    path: models/energy.fmu
    step_size: 0.01  # 10 ms

connections:
  - from: network_sim.frame_latency
    to: cpu_sim.network_delay

  - from: cpu_sim.cpu_load
    to: energy_sim.compute_power
```

Dieses Setup ermöglicht die simultane Simulation von Netzwerk, CPU und Energie, wobei die Modelle über definierte Schnittstellen kommunizieren.

Functional Mock-up Interface (FMI) 2.0/3.0 ermöglicht Co-Simulation verschiedener Tools:

- **Stärken**:
  - Tool-übergreifende Simulation (FMI 2.0 für Co-Simulation, FMI 3.0 für Model Exchange)
  - Standardisiertes Interface (FMI Standard)
  - Wiederverwendbare Komponenten (FMUs - Functional Mock-up Units)
  - Unterstützung für verschiedene Solver
  - Moderne Implementierungen (FMPy, PyFMI, FMI++ Library)
- **Schwächen**:
  - Synchronisations-Overhead zwischen Tools
  - Komplexere Koordination bei vielen FMUs
  - Performance-Overhead durch Tool-Grenzen
- **Einsatz**: Für Multi-Tool-Co-Simulation, Multi-Domain-Simulation

Moderne Simulationsframeworks

Neue Simulationsframeworks bieten erweiterte Funktionalität:

- **CARLA**: Open-Source-Simulator für autonomes Fahren mit realistischer Sensorik
- **SUMO**: Verkehrssimulation für V2X-Szenarien

- **Apache Airflow**: Workflow-Management für komplexe Simulations-Pipelines
- **Prefect**: Moderne Alternative zu Airflow mit Python-First-Ansatz
- **Dagster**: Data Orchestration Platform für Simulations-Pipelines

6.1.2. Auswahlkriterien

Die Auswahl einer Zielplattform hängt von folgenden Kriterien ab:

Tabelle 6.1.: Auswahlkriterien für Simulationsplattformen

Kriterium	OMNeT++	Simulink	Modelica
Netzwerk-Detaillierung	+++	+	+
CPU-Scheduling	+	+++	+
TSN-Support	+++	+	-
Energie-Modellierung	+	+	+++
Performance	+++	++	+
Tool-Kosten	Open	Proprietär	Proprietär

6.2. Mapping-Regeln

6.2.1. Grundprinzipien

Mapping-Regeln definieren, wie Architektur-Elemente aus PREEvision in Simulations-Elemente der Zielplattform transformiert werden. Die Regeln müssen semantisch korrekt sein und alle relevanten Aspekte berücksichtigen.

6.2.2. ECU → Rechenknoten

Strukturelles Mapping

Ein ECU aus PREEvision wird zu einem Rechenknoten im Simulationsmodell:

- **ECU-Identifikation** → **Node-ID**: Eindeutige Identifikation bleibt erhalten
- **CPU-Kerne** → **CPU-Ressourcen**: Anzahl Kerne, Frequenz, Topologie
- **GPU/NPU** → **Accelerator-Ressourcen**: Rechenleistung, Speicher
- **RAM/Storage** → **Memory-Ressourcen**: Größe, Zugriffszeiten
- **ASIL-Level** → **Safety-Attribute**: Für Verfügbarkeitsmodellierung

Scheduler-Parameter

Die Scheduling-Parameter werden aus Task-Definitionen abgeleitet:

Tabelle 6.2.: Mapping: Task-Parameter zu Scheduler-Konfiguration

PREEvision	Simulationsmodell	Beschreibung
task.period	scheduler.period	Periodizität
task.deadline	scheduler.deadline	Deadline
task.wcet	scheduler.execution_time	Ausführungszeit
task.priority	scheduler.priority	Priorität
task.scheduling_policy	scheduler.policy	Scheduling-Algorithmus

6.2.3. Netzwerk → Kanal

Link-Mapping

Ein Link aus PREEvision wird zu einem Kommunikationskanal:

- **Link-Bandbreite** → **Channel-Bandwidth**: Datenrate (bit/s)
- **Link-Latenz** → **Channel-Latency**: Basis-Latenz
- **Link-Jitter** → **Channel-Jitter**: Variabilität
- **Protokoll** → **Channel-Type**: Ethernet, CAN, LIN, etc.

TSN-Mapping

Für TSN-Netze werden zusätzliche Parameter gemappt:

- **Gate-Schedules** → **TSN-Schedule**: Zeitgesteuerte Queues
- **Prioritäten** → **Queue-Priorities**: IEEE 802.1Q-Prioritäten
- **Traffic Shaping** → **Shaping-Parameters**: CBS/TAS-Parameter
- **Synchronisation** → **Time-Sync**: gPTP-Konfiguration

Switch-Mapping

Switches werden zu speziellen Netzwerkknoten:

- **Switch-Ports** → **Network-Ports**: Anzahl, Bandbreiten
- **Routing-Tabellen** → **Forwarding-Tables**: Routing-Logik
- **Queue-Konfiguration** → **Queue-Settings**: Buffer-Größen, Policies

6.2.4. Messages/Signals → Traffic-Flows

Frame-Mapping

Frames werden zu Traffic-Flows im Simulationsmodell:

Tabelle 6.3.: Mapping: Frame-Parameter zu Traffic-Flow

PREEvision	Simulationsmodell	Beschreibung
frame.id	flow.id	Eindeutige Identifikation
frame.period	flow.period	Periodizität
frame.size	flow.size	Paket-Größe
frame.priority	flow.priority	Priorität
frame.source	flow.source	Quellknoten
frame.destinations	flow.destinations	Zielknoten
frame.route	flow.path	Routing-Pfad

Signal-Mapping

Signale innerhalb von Frames werden zu Datenfeldern:

- **Signal-Name** → **Field-Name**: Bezeichnung bleibt erhalten
- **Signal-Type** → **Field-Type**: Datentyp (Integer, Float, etc.)
- **Signal-Offset** → **Field-Offset**: Byte-Position im Frame
- **Signal-Scale** → **Field-Scale**: Skalierung für physikalische Werte

6.2.5. SWC/Chain → Tasks/Pipelines

Software-Komponenten

SWCs werden zu Tasks oder Task-Gruppen:

- **SWC-Runnables** → **Task-Functions**: Ausführbare Funktionen
- **SWC-Ports** → **Task-Interfaces**: Daten-Ein-/Ausgänge
- **SWC-Dependencies** → **Task-Dependencies**: Abhängigkeiten

Funktionsketten

Chains werden zu Pipelines im Simulationsmodell:

- **Chain-Path** → **Pipeline-Stages**: Sequenz von Verarbeitungsschritten
- **Chain-Deadline** → **Pipeline-Deadline**: End-to-End-Deadline
- **Chain-Latency-Budget** → **Pipeline-Latency-Budget**: Latenzbudget

6.2.6. Redundanzgruppen → parallele Pfade

Redundanz-Mapping

Redundanzgruppen werden zu parallelen Ausführungspfaden:

- **Redundancy-Group** → **Parallel-Paths**: Mehrere identische Pfade
- **Voting-Mechanismus** → **Voting-Logic**: Vergleich der Ausgaben
- **Umschaltzeit** → **Failover-Time**: Zeit bis zur Umschaltung
- **Health-Monitoring** → **Health-Checks**: Überwachungslogik

6.3. Artefakte und Exporte

6.3.1. Export aus PREEvision

REST API Export

Die REST API ermöglicht einen strukturierten Export:

- **Topologie**: JSON-Struktur mit Knoten, Ports, Links
- **Kommunikation**: JSON-Struktur mit Frames, Signalen, Routen
- **Software**: JSON-Struktur mit SWCs, Tasks, Deployment
- **Attribute**: Alle annotierten Attribute als Key-Value-Paare

CSV Export

CSV-Exporte für tabellarische Daten:

- **Nodes.csv**: Alle Knoten mit Attributen
- **Links.csv**: Alle Verbindungen mit Parametern
- **Frames.csv**: Alle Frames mit Timing-Informationen
- **Tasks.csv**: Alle Tasks mit Scheduling-Parametern

AUTOSAR XML Export

Standardisiertes Format für AUTOSAR-Toolketten:

- **System Description**: Topologie und Kommunikation
- **Software Component Description**: SWC-Definitionen
- **ECU Configuration**: Deployment-Informationen

XMI/JSON Export

Metamodell-basierte Exporte:

- **Metamodell**: Klassen, Attribute, Beziehungen
- **Instanz-Daten**: Konkrete Modellinstanzen
- **Versionierung**: Modellversionen und Historie

6.3.2. Intermediate Model (IM)

Konzept

Das Intermediate Model (IM) dient als stabile Zwischenschicht zwischen PREEvision und den verschiedenen Zielplattformen:

- **Abstraktion**: Plattform-unabhängige Repräsentation
- **Vollständigkeit**: Enthält alle für die Simulation relevanten Informationen
- **Erweiterbarkeit**: Kann um neue Architekturmerkmale erweitert werden
- **Validierbarkeit**: Kann auf Konsistenz und Vollständigkeit geprüft werden

IM-Struktur

Das IM besteht aus folgenden Hauptkomponenten:

1. **Topology Model**: Knoten, Ports, Links, Switches
2. **Communication Model**: Frames, Signale, Routen, TSN-Konfiguration
3. **Software Model**: SWCs, Tasks, Runnables, Chains
4. **Deployment Model**: Zuordnung von Software zu Hardware
5. **Resource Model**: CPU, GPU, NPU, Memory, Bandbreite
6. **Timing Model**: Periodizitäten, Deadlines, Latenzbudgets
7. **Safety Model**: ASIL-Level, Redundanz, Ausfallraten
8. **Energy Model**: Power States, Energieprofile

IM-Format

Das IM wird als strukturiertes Format (z.B. JSON, XML) gespeichert:

```
{
  "version": "1.0",
  "metadata": {
    "source": "PREEvision",
    "export_date": "2024-01-15",
    "model_version": "v2.3"
  },
  "topology": {
    "nodes": [...],
    "links": [...],
    "switches": [...]
  },
  "communication": {
    "frames": [...],
    "signals": [...],
    "routes": [...],
    "tsn_config": [...]
  },
  "software": {
```

```
    "swcs": [...],
    "tasks": [...],
    "chains": [...]
  },
  ...
}
```

Vorteile des IM

- **Plattform-Unabhängigkeit**: Ein IM kann in verschiedene Zielplattformen transformiert werden
- **Wiederverwendbarkeit**: IM kann für verschiedene Analysen verwendet werden
- **Validierung**: Konsistenzprüfung unabhängig von Zielplattform
- **Versionierung**: Änderungen im IM können nachverfolgt werden
- **Debugging**: Probleme können im IM identifiziert werden, bevor sie in die Zielplattform überführt werden

6.4. Erweiterte Transformations-Beispiele

Dieser Abschnitt präsentiert erweiterte Beispiele für die Transformation von PREEvision-Modellen in Simulationsmodelle. Die Beispiele zeigen, wie verschiedene Aspekte der Architektur transformiert werden.

6.4.1. Beispiel: Transformation einer zonalen Architektur

Dieses Beispiel zeigt die vollständige Transformation einer zonalen Architektur mit vier Zonen-Controllern und einem zentralen Rechenknoten.

Topologie-Transformation

Die Topologie wird wie folgt transformiert:

- **Zentraler Rechenknoten**:
  - PREEvision: `ECU_CentralCompute` mit Attributen (CPU: 16 cores, GPU: 20 TFLOPS, RAM: 16 GB)
  - OMNeT++: `StandardHost` mit `ResourceManager` für CPU/GPU/Memory

  - Konfiguration: CPU-Kerne, GPU-Performance, Memory-Größe werden direkt gemappt

- **Zonen-Controller**:
  - PREEvision: `ECU_ZoneFront`, `ECU_ZoneLeft`, etc.
  - OMNeT++: `StandardHost` mit `EthernetInterface` und `CanInterface`
  - Gateway-Funktionalität: Modelliert als `Application` mit CAN/Ethernet-Bridge

- **TSN-Switch**:
  - PREEvision: `Switch_TSN` mit Gate-Schedule-Konfiguration
  - OMNeT++: `EthernetSwitch` mit `TsnGate` Modulen
  - Gate-Schedule: Transformiert in `TsnGateSchedule` Konfiguration

Kommunikations-Transformation

Die Kommunikation wird transformiert:

- **Ethernet-Links**:
  - PREEvision: `Link` mit Attributen (Bandwidth: 2.5 Gbps, Latency: 1 ms)
  - OMNeT++: `EthernetLink` mit `DatarateChannel` (2.5 Gbps)
  - Latenz: Modelliert als `delay` Parameter im Channel

- **TSN-Frames**:
  - PREEvision: `Frame` mit Attributen (Size: 1500 Byte, Priority: 6, Period: 33.3 ms)
  - OMNeT++: `EthernetFrame` mit `UserPriority` (6) und `PeriodicSource`
  - Gate-Scheduling: Frame wird in entsprechende TSN-Queue eingeordnet

- **CAN-Frames**:
  - PREEvision: `Frame` mit Attributen (CAN-ID: 0x123, Size: 8 Byte, Period: 10 ms)
  - OMNeT++: `CanFrame` mit `CanId` (0x123) und `PeriodicSource`
  - Gateway: CAN-Frame wird in Ethernet-Frame eingebettet (Gateway-Funktionalität)

Software-Transformation

Die Software wird transformiert:

- **Tasks**:
  - PREEvision: `Task` mit Attributen (WCET: 15 ms, Period: 33.3 ms, Priority: 10)
  - OMNeT++: `Application` mit `PeriodicTask` (Period: 33.3 ms)
  - Scheduling: Task wird von `ResourceManager` mit Fixed-Priority Scheduling ausgeführt
  - WCET: Modelliert als `executionTime` Parameter
- **Chains**:
  - PREEvision: `Chain` mit Sequenz von Tasks und Frames
  - OMNeT++: `CompoundApplication` mit mehreren `PeriodicTask` Instanzen
  - Datenfluss: Tasks kommunizieren über `Socket` oder `MessageQueue`

6.4.2. Beispiel: Transformation einer redundanten Architektur

Dieses Beispiel zeigt die Transformation einer redundanten Lenkungsarchitektur.

Redundanz-Transformation

Die Redundanz wird wie folgt transformiert:

- **RedundancyGroup**:
  - PREEvision: `RedundancyGroup` mit zwei Pfaden (Primär, Backup)
  - OMNeT++: `RedundantApplication` mit zwei `Application` Instanzen
  - Voting: Modelliert als `VotingModule`, das beide Ausgaben vergleicht
- **Switchover**:
  - PREEvision: `SwitchoverConfig` mit Attributen (SwitchoverTime: 10 ms)
  - OMNeT++: `SwitchoverManager` mit `switchoverDelay` (10 ms)
  - Health-Monitoring: Modelliert als `HealthMonitor`, das beide Pfade überwacht

Fehler-Injection

Für die Simulation von Fehlerszenarien:

- **ECU-Ausfall**:
  - PREEvision: `FailureModel` mit Attributen (MTBF: 10000 h, FailureMode: Immediate)
  - OMNeT++: `FailureGenerator` mit `exponential` Verteilung (MTBF: 10000 h)
  - Ausfallmodus: Modelliert als `immediateFailure` Event
- **Link-Ausfall**:
  - PREEvision: `LinkFailureModel` mit Attributen (MTBF: 50000 h)
  - OMNeT++: `LinkFailureGenerator` mit `exponential` Verteilung
  - Auswirkung: Link wird als `broken` markiert, Frames werden verworfen

6.5. Erweiterte Transformations-Strategien

Dieser Abschnitt beschreibt erweiterte Strategien für die Transformation komplexer Architekturen.

6.5.1. Inkrementelle Transformation

Inkrementelle Transformation ermöglicht die effiziente Aktualisierung von Simulationsmodellen:

- **Change-Detection**: Automatische Erkennung von Änderungen im Architekturmodell
- **Dependency-Analyse**: Analyse von Abhängigkeiten, um zu bestimmen, welche Teile neu transformiert werden müssen
- **Inkrementelle Generierung**: Nur geänderte Teile werden neu generiert
- **Validierung**: Validierung der inkrementellen Änderungen

6.5.2. Multi-Platform-Transformation

Multi-Platform-Transformation ermöglicht die gleichzeitige Transformation in mehrere Zielplattformen:

- **Plattform-Abstraktion**: Gemeinsame Abstraktion für verschiedene Plattformen
- **Plattform-spezifische Generatoren**: Separate Generatoren für jede Plattform
- **Plattform-Vergleich**: Vergleich von Ergebnissen zwischen Plattformen
- **Plattform-Optimierung**: Plattform-spezifische Optimierungen

6.5.3. Hybrid-Transformation

Hybrid-Transformation kombiniert verschiedene Transformationsansätze:

- **Regelbasierte Transformation**: Für strukturierte Transformationen
- **Template-basierte Transformation**: Für Code-Generierung
- **Programmatische Transformation**: Für komplexe Transformationen
- **KI-basierte Transformation**: Für intelligente Transformationen (zukünftig)

6.6. Erweiterte Transformations-Beispiele

Dieser Abschnitt präsentiert detaillierte Beispiele für die Transformation verschiedener Architektur-Elemente.

6.6.1. Beispiel: Transformation einer zonalen Architektur

Dieses Beispiel zeigt die vollständige Transformation einer zonalen Architektur:

Topologie-Transformation

- **Central Compute** → OMNeT++ StandardHost mit CPU/GPU/NPU-Ressourcen
- **Zonen-Controller** → OMNeT++ StandardHost mit Gateway-Funktionalität
- **Sensoren** → OMNeT++ Sensor-Nodes mit Daten-Generierung
- **Aktoren** → OMNeT++ Actuator-Nodes mit Daten-Verarbeitung
- **Switches** → OMNeT++ EthernetSwitch mit TSN-Unterstützung

Kommunikations-Transformation

- **TSN-Frames** → OMNeT++ EthernetFrames mit TSN-Konfiguration
- **Gate-Schedules** → OMNeT++ GateSchedule-Konfiguration
- **Routen** → OMNeT++ Routing-Tabellen
- **Redundanz** → OMNeT++ Redundancy-Mechanismen

Software-Transformation

- **Tasks** → OMNeT++ Applications mit Periodicity
- **SWCs** → OMNeT++ CompoundApplications
- **Chains** → OMNeT++ ChainApplications
- **Scheduling** → OMNeT++ Scheduler-Konfiguration

6.7. Erweiterte Transformations-Beispiele

Dieser Abschnitt präsentiert detaillierte Beispiele für die Transformation verschiedener Architektur-Elemente.

6.7.1. Beispiel: Transformation einer komplexen Funktionskette

Ausgangs-Architektur

Die Funktionskette umfasst:

- **Sensoren**: 3x Kameras (Front, Left, Right), 2x Radar, 1x LiDAR
- **Verarbeitung**:
  - Bildverarbeitung auf ZC_Front (3x Kamera-Streams)
  - Radar-Signalverarbeitung auf ZC_Front
  - LiDAR-Verarbeitung auf ZC_Front
  - Sensorfusion auf AD-DC
  - Objekterkennung (YOLOv8) auf AD-DC (NPU)
  - Tracking auf AD-DC (CPU)
  - Trajektorien-Prädiktion auf AD-DC (CPU)
  - Pfadplanung auf AD-DC (CPU)

  - Regelung auf AD-DC (CPU)

- **Aktoren**: EPS (Lenkung), EHB (Bremse)

Transformation-Schritte

1. **Sensoren**: Transformation in OMNeT++ Sensor-Nodes

```
sensor_front_camera: CameraSensor {
    frameRate = 30fps;
    resolution = 1920x1080;
    outputSize = 1500B;
}
```

2. **Zonen-Controller**: Transformation in OMNeT++ StandardHost mit Gateway-Funktionalität

```
zc_front: StandardHost {
    cpu.clock = 1.5GHz;
    interfaces[0].bandwidth = 1Gbps;
    gateway = true;
}
```

3. **AD-DC**: Transformation in OMNeT++ StandardHost mit CPU/GPU/NPU-Ressourcen

```
ad_dc: StandardHost {
    cpu.clock = 2.5GHz;
    gpu.performance = 20TFLOPS;
    npu.performance = 200TOPS;
    interfaces[0].bandwidth = 10Gbps;
}
```

4. **Tasks**: Transformation in OMNeT++ Applications mit Periodicity

```
perception_task: Application {
    period = 33ms;  // 30fps
    wcet = 15ms;
```

```
    priority = 10;
    cpuCore = 0;
}
```

5. **Frames**: Transformation in OMNeT++ EthernetFrames mit TSN-Konfiguration

```
camera_frame: EthernetFrame {
    size = 1500B;
    period = 33ms;
    priority = 7;
    route = [zc_front, switch, ad_dc];
    tsn_config = {
        gate_schedule = [...];
        traffic_shaping = {...};
    };
}
```

6. **Chains**: Transformation in OMNeT++ CompoundApplications

```
perception_chain: CompoundApplication {
    components = [
        camera_front,
        image_processing,
        object_detection,
        tracking
    ];
    e2e_deadline = 100ms;
}
```

Ergebnisse

Die Transformation ergab:

- **OMNeT++-Modell**: 150+ Nodes, 500+ Connections
- **Simulationszeit**: 2 Stunden für 1 Stunde Fahrzeit
- **Ergebnisse**: Alle KPIs innerhalb der Ziele

6.8. Zusammenfassung

Dieses Kapitel hat das Transformationskonzept beschrieben, das Architekturmodelle aus PREEvision in Simulationsmodelle überführt. Die wichtigsten Aspekte umfassen:

- **Zielplattformen**: Übersicht verschiedener Simulationsplattformen und deren Stärken/Schwächen
- **Mapping-Regeln**: Detaillierte Regeln für die Transformation von Architektur-Elementen in Simulations-Elemente
- **Artefakte und Exporte**: Verschiedene Exportformate aus PREEvision
- **Intermediate Model**: Konzept einer stabilen Zwischenschicht für plattformunabhängige Transformation

Das Transformationskonzept bildet die Grundlage für die technische Realisierung, wie sie im folgenden Kapitel beschrieben wird.

7. Technische Realisierung der Transformation

Dieses Kapitel beschreibt die technische Realisierung der Transformation von PREEvision-Architekturmodellen in Simulationsmodelle. Die Implementierung umfasst Datenaufnahme, Normalisierung, Regel-Engine, Code-Generierung und automatisierte Validierung.

7.1. Datenaufnahme und Normalisierung

7.1.1. Strukturierte Exporte

Export-Konfiguration

Die Datenaufnahme beginnt mit der Konfiguration strukturierter Exporte aus PREEvision:

- **Export-Profil**: Definiert, welche Modell-Elemente exportiert werden sollen
- **Format-Auswahl**: REST API, CSV, AUTOSAR XML oder XMI/JSON
- **Filter-Kriterien**: Optional Filterung nach Komponententypen, Domänen oder ASIL-Leveln
- **Versionierung**: Export-Versionen werden mit Timestamps und Modell-Versionen versehen

Versionierung

Jeder Export wird versioniert, um Änderungen nachverfolgen zu können:

- **Modell-Version**: Version des PREEvision-Modells
- **Export-Timestamp**: Zeitpunkt des Exports
- **Export-Hash**: Hash-Wert zur Integritätsprüfung
- **Änderungs-Historie**: Nachverfolgung von Änderungen zwischen Versionen

7.1.2. Parser und Adapter

Moderne Format-Parser

Verschiedene Parser werden für moderne Exportformate implementiert:

- **JSON-Parser**: Für REST API und JSON-Exporte mit Unterstützung für:
  - JSON Schema Validierung (jsonschema)
  - Streaming-Parsing für große Dateien (ijson)
  - JSON-LD für semantische Daten
- **CSV-Parser**: Für tabellarische Exporte mit:
  - Pandas für effiziente Datenverarbeitung
  - Streaming-Parsing für große CSV-Dateien
  - Automatische Typ-Inferenz
- **XML-Parser**: Für AUTOSAR XML und XMI-Exporte mit:
  - lxml für performantes Parsing
  - XPath für komplexe Abfragen
  - Namespace-Unterstützung
- **GraphQL-Parser**: Für moderne GraphQL-APIs (falls PREEvision GraphQL unterstützt)
- **gRPC-Parser**: Für gRPC-basierte Exporte (falls verfügbar)
- **Parquet/Arrow-Parser**: Für effiziente binäre Datenformate

Adapter-Pattern

Das Adapter-Pattern ermöglicht die einheitliche Verarbeitung verschiedener Formate:

```
class ExportAdapter:
    def parse(self, source: str) -> IntermediateModel:
        """Parse export format to IM"""
        pass

class JSONAdapter(ExportAdapter):
    def parse(self, source: str) -> IntermediateModel:
        # Parse JSON export
```

```
        pass

class CSVAdapter(ExportAdapter):
    def parse(self, source: str) -> IntermediateModel:
        # Parse CSV export
        pass
```

Normalisierung

Die Normalisierung stellt sicher, dass Daten aus verschiedenen Quellen in ein einheitliches Format überführt werden:

- **Einheitliche Bezeichnungen**: Standardisierung von Namen und IDs
- **Einheitliche Einheiten**: Konvertierung in SI-Einheiten (ms, Byte, bit/s)
- **Datenbereinigung**: Entfernung von Duplikaten, Korrektur von Inkonsistenzen
- **Vervollständigung**: Ergänzung fehlender Werte mit Defaults oder Schätzungen

7.1.3. Validierung der Eingabedaten

Schema-Validierung

Die Eingabedaten werden gegen ein Schema validiert:

- **Struktur-Validierung**: Prüfung auf korrekte JSON/XML-Struktur
- **Typ-Validierung**: Prüfung auf korrekte Datentypen
- **Referenz-Validierung**: Prüfung, ob alle Referenzen aufgelöst werden können
- **Constraint-Validierung**: Prüfung von Randbedingungen (z. B. positive Werte)

Konsistenz-Checks

Konsistenz-Checks prüfen die logische Korrektheit:

- **Referenz-Integrität**: Alle referenzierten Elemente existieren
- **Zyklus-Erkennung**: Erkennung von Zyklen in Abhängigkeiten
- **Werte-Konsistenz**: Prüfung auf widersprüchliche Werte
- **Vollständigkeit**: Prüfung, ob alle erforderlichen Informationen vorhanden sind

7.2. Regel-Engine und Generator

Die Regel-Engine ist das Herzstück des Transformations-Frameworks. Sie interpretiert die Mapping-Regeln und führt die Transformation von PREEvision-Elementen in Simulations-Elemente durch. Die Engine ist so designed, dass sie flexibel, erweiterbar und wartbar ist.

7.2.1. Regelkatalog

Der Regelkatalog enthält alle Mapping-Regeln, die für die Transformation verwendet werden. Jede Regel definiert, wie ein bestimmter Typ von PREEvision-Element in ein entsprechendes Simulations-Element transformiert wird. Der Katalog wird in strukturierter Form (YAML) gespeichert, um sowohl maschinenlesbar als auch menschenlesbar zu sein.

YAML-basierte Konfiguration

Mapping-Regeln werden in YAML-Dateien definiert, was eine klare, strukturierte und leicht verständliche Darstellung ermöglicht. YAML bietet mehrere Vorteile:

- **Lesbarkeit**: YAML ist sehr lesbar und erfordert keine speziellen Programmierkenntnisse
- **Strukturierung**: YAML unterstützt hierarchische Strukturen, die gut zu den Mapping-Regeln passen
- **Erweiterbarkeit**: Neue Regeln können einfach hinzugefügt werden, ohne bestehende Regeln zu ändern
- **Versionierung**: YAML-Dateien können einfach in Versionskontrollsystemen verwaltet werden

```
# mapping_rules.yaml
transformations:
  - name: ecu_to_node
    source: ECU
    target: Node
    mappings:
      - from: ecu.id
        to: node.id
      - from: ecu.cpu_cores
        to: node.cpu.cores
```

```
      - from: ecu.ram_size
        to: node.memory.size
    validations:
      - check: cpu_cores > 0
        error: "CPU cores must be positive"

  - name: frame_to_traffic_flow
    source: Frame
    target: TrafficFlow
    mappings:
      - from: frame.id
        to: flow.id
      - from: frame.period
        to: flow.period
      - from: frame.size
        to: flow.size
    calculations:
      - name: bandwidth
        formula: "size * 8 / period"
        unit: "bit/s"
```

Regel-Typen

Verschiedene Regel-Typen werden unterstützt:

- **Direct Mapping**: Direkte Zuordnung von Attributen
- **Transformation**: Berechnung neuer Werte aus vorhandenen
- **Aggregation**: Zusammenfassung mehrerer Elemente
- **Conditional Mapping**: Bedingte Zuordnungen basierend auf Bedingungen
- **Default Values**: Standardwerte für fehlende Informationen

7.2.2. Code-Generator

Template-basierte Generierung

Code wird mit Template-Engines (z. B. Jinja2) generiert:

```
# Template: omnetpp_network.ned.j2
```

```
network {{ network_name }} {
    types:
{% for node in nodes %}
    node {{ node.id }} {
        parameters:
            cpu.cores = {{ node.cpu_cores }};
            memory.size = {{ node.ram_size }}MB;
{% endfor %}
    }

    connections:
{% for link in links %}
        {{ link.source }}.port <--> {{ link.bandwidth }}bps <--> {{ link.destination
{% endfor %}
}
```

Generator-Architektur

Die Generator-Architektur besteht aus:

- **Template-Loader**: Lädt Templates aus Dateien
- **Context-Builder**: Erstellt Kontext-Daten aus IM
- **Renderer**: Rendert Templates zu Code
- **Post-Processor**: Bearbeitet generierten Code (Formatierung, Validierung)

Plattform-spezifische Generatoren

Für jede Zielplattform wird ein spezifischer Generator implementiert:

- **OMNeT++ Generator**: Generiert .ned-Dateien und .ini-Konfigurationen
- **Simulink Generator**: Generiert Simulink-Modelle und TrueTime-Konfigurationen
- **NS-3 Generator**: Generiert C++-Code für NS-3
- **Modelica Generator**: Generiert Modelica-Modelle

7.2.3. Konfigurationsgenerator

Simulations-Konfiguration

Zusätzlich zum Modell-Code werden Simulations-Konfigurationen generiert:

```
# omnetpp.ini
[General]
network = {{ network_name }}
sim-time-limit = {{ simulation_time }}s

[Config {{ scenario_name }}]
**.cpu.scheduler = "fixedPriority"
{% for task in tasks %}
**.{{ task.node }}.task[{{ task.id }}].period = {{ task.period }}ms
**.{{ task.node }}.task[{{ task.id }}].wcet = {{ task.wcet }}ms
{% endfor %}
```

Szenario-Konfigurationen

Verschiedene Szenarien werden als separate Konfigurationen generiert:

- **Nominal-Szenario**: Standard-Betriebsbedingungen
- **Stress-Szenario**: Hohe Last, Grenzfälle
- **Fehler-Szenario**: Ausfälle, Degradation
- **Variante-Szenarien**: Verschiedene Design-Varianten

7.3. Automatisierte Validierung und CI

7.3.1. Schema-Validierung

IM-Schema

Das Intermediate Model wird gegen ein Schema validiert:

```
# im_schema.json
{
  "type": "object",
  "properties": {
    "topology": {
```

```
      "type": "object",
      "properties": {
        "nodes": {
          "type": "array",
          "items": {
            "type": "object",
            "required": ["id", "type"],
            "properties": {
              "id": {"type": "string"},
              "type": {"type": "string"},
              "cpu_cores": {"type": "integer", "minimum": 1}
            }
          }
        }
      }
    }
  }
}
```

Validierungs-Framework

Ein Validierungs-Framework prüft das IM:

- **JSON Schema Validator**: Prüfung gegen JSON Schema
- **Custom Validators**: Zusätzliche benutzerdefinierte Prüfungen
- **Error Reporting**: Detaillierte Fehlermeldungen mit Kontext

7.3.2. Konsistenzchecks

Cross-Reference Checks

Konsistenzchecks prüfen Referenzen zwischen Modell-Elementen:

- **Link-Validierung**: Alle Links referenzieren existierende Knoten
- **Frame-Validierung**: Alle Frames haben gültige Quellen und Ziele
- **Task-Validierung**: Alle Tasks sind auf existierenden ECUs deployed
- **Chain-Validierung**: Alle Chains haben vollständige Pfade

Constraint-Validierung

Constraints werden geprüft:

- **Timing-Constraints**: Deadlines sind größer als Periodizitäten
- **Ressourcen-Constraints**: CPU-Auslastung $\leq 100\%$
- **Bandbreiten-Constraints**: Netzwerk-Auslastung $\leq 100\%$
- **Safety-Constraints**: ASIL-Anforderungen werden eingehalten

7.3.3. Reproduzierbare Build-Pipelines

Moderne CI/CD-Integration

Die Transformation wird in moderne CI/CD-Pipelines integriert, die Containerisierung, Cloud-Deployment und automatisierte Tests unterstützen:

```
# .github/workflows/transform.yml (GitHub Actions v4)
name: Transform Architecture Model

on:
  push:
    branches: [ main, develop ]
  pull_request:
  workflow_dispatch:
    inputs:
      architecture:
        description: 'Architecture model to transform'
        required: true
        type: string

jobs:
  transform:
    runs-on: ubuntu-latest
    container:
      image: ghcr.io/org/transformer:latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4
```

```
- name: Setup Python
  uses: actions/setup-python@v4
  with:
    python-version: '3.11'
    cache: 'pip'

- name: Install dependencies
  run: |
    pip install --upgrade pip
    pip install -r requirements.txt
    pip install -r requirements-dev.txt

- name: Run transformation
  run: |
    python transform.py \
      --input ${{ inputs.architecture || 'default' }}.json \
      --output simulation/ \
      --format omnetpp \
      --validate
  env:
    TRANSFORM_CACHE: ${{ runner.temp }}/cache

- name: Validate output
  run: |
    python validate.py \
      --input simulation/ \
      --schema im_schema.json \
      --strict

- name: Run unit tests
  run: pytest tests/unit/ -v --cov=transform --cov-report=xml

- name: Run integration tests
  run: pytest tests/integration/ -v

- name: Build Docker image for simulation
  run: |
    docker build -t simulation:latest -f Dockerfile.simulation .
```

```
      - name: Run simulation in container
        run: |
          docker run --rm \
            -v $(pwd)/simulation:/simulation \
            -v $(pwd)/results:/results \
            simulation:latest \
            ./run_simulation.sh

      - name: Upload artifacts
        uses: actions/upload-artifact@v4
        with:
          name: simulation-results
          path: results/
          retention-days: 30

      - name: Publish to container registry
        if: github.ref == 'refs/heads/main'
        run: |
          echo "${{ secrets.GITHUB_TOKEN }}" | docker login ghcr.io -u ${{ github.ac
          docker tag simulation:latest ghcr.io/${{ github.repository }}/simulation:$
          docker push ghcr.io/${{ github.repository }}/simulation:${{ github.sha }}
```

Containerisierung mit Docker

Containerisierung ermöglicht reproduzierbare Umgebungen und einfaches Deployment:

```
# Dockerfile
FROM python:3.11-slim

WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y \
    build-essential \
    git \
    && rm -rf /var/lib/apt/lists/*

# Install Python dependencies
```

```
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY . .

# Set environment variables
ENV PYTHONUNBUFFERED=1
ENV TRANSFORM_CACHE=/cache

# Create cache directory
RUN mkdir -p /cache

# Run transformation
CMD ["python", "transform.py"]
```

Kubernetes für Skalierung

Für große Simulationsläufe kann Kubernetes für parallele Ausführung verwendet werden:

```
# k8s/job.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: simulation-batch
spec:
  parallelism: 10
  completions: 100
  template:
    spec:
      containers:
      - name: transformer
        image: ghcr.io/org/transformer:latest
        resources:
          requests:
            memory: "2Gi"
            cpu: "1"
          limits:
```

```
            memory: "4Gi"
            cpu: "2"
        env:
        - name: ARCHITECTURE_ID
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        volumeMounts:
        - name: results
          mountPath: /results
      volumes:
      - name: results
        persistentVolumeClaim:
          claimName: simulation-results
      restartPolicy: Never
  backoffLimit: 3
```

Moderne Versionierung und Artifact-Management

Alle Artefakte werden mit modernen Versionierungs- und Artifact-Management-Systemen verwaltet:

- **Semantic Versioning**: Versionen folgen Semantic Versioning (MAJOR.MINOR.PATCH)
- **Git-Tags**: Versionen werden mit Git-Tags markiert und automatisch durch CI/CD erstellt
- **Container-Registry**: Docker-Images werden in Container-Registries (GitHub Container Registry, Docker Hub, GitLab Registry) gespeichert
- **Artifact-Storage**: Generierte Simulationen werden in Artifact-Stores (GitHub Artifacts, S3, MinIO) gespeichert
- **Change-Log**: Automatische Generierung von Change-Logs aus Git-Commits (Conventional Commits)
- **SBOM (Software Bill of Materials)**: Automatische Generierung von SBOMs für Sicherheits-Compliance

Reproduzierbarkeit und Determinismus

Reproduzierbarkeit wird durch moderne Techniken sichergestellt:

- **Deterministische Generierung**: Gleiche Eingabe führt zu gleicher Ausgabe durch:
  - Fixierte Zufalls-Seeds
  - Sortierte Datenstrukturen
  - Deterministische Hash-Funktionen
- **Dependency-Pinning**:
  - `requirements.txt` mit exakten Versionen
  - `poetry.lock` oder `Pipfile.lock` für Python
  - `package-lock.json` für Node.js-Abhängigkeiten
  - Container-Images mit spezifischen Tags (nicht `latest`)
- **Environment-Dokumentation**:
  - `Dockerfile` für Container-Umgebung
  - `environment.yml` für Conda-Umgebungen
  - `devcontainer.json` für VS Code Dev Containers
  - CI/CD-Umgebungen werden dokumentiert und versioniert
- **Seed-Management**:
  - Zufalls-Seeds werden in Konfigurationsdateien gespeichert
  - Seed-Generierung basiert auf deterministischen Hash-Funktionen
  - Seeds werden versioniert und dokumentiert
- **Container-Reproduzierbarkeit**:
  - Multi-Stage Builds für optimierte Images
  - Layer-Caching für schnelle Rebuilds
  - Content-Addressable Storage für Images

Cloud-native Deployment

Moderne Cloud-native Ansätze für Skalierung und Effizienz:

- **Serverless Functions**: AWS Lambda, Azure Functions, Google Cloud Functions für kleine Transformationen
- **Container Orchestration**: Kubernetes für komplexe Workflows

- **Distributed Computing**: Dask, Ray für parallele Verarbeitung großer Modelle
- **Cloud Storage**: S3, Azure Blob, Google Cloud Storage für Artefakt-Speicherung
- **Message Queues**: RabbitMQ, Apache Kafka für asynchrone Verarbeitung

7.4. Erweiterte Implementierungsdetails

Dieser Abschnitt beschreibt erweiterte Implementierungsdetails, die für eine produktive Nutzung des Transformations-Frameworks wichtig sind.

7.4.1. Performance-Optimierung

Die Performance-Optimierung ist entscheidend für die praktische Nutzbarkeit des Frameworks, insbesondere bei großen Architekturen.

Parsing-Optimierung

Große PREEvision-Exporte können mehrere Gigabyte groß sein. Effizientes Parsing ist daher essentiell:

- **Streaming-Parsing**: Verwendung von Streaming-Parsern (ijson für JSON, iterparse für XML), die Dateien nicht vollständig in den Speicher laden
- **Inkrementelle Verarbeitung**: Verarbeitung von Daten in Chunks, um Speicherverbrauch zu reduzieren
- **Parallelisierung**: Parallele Verarbeitung unabhängiger Datenstrukturen mit Multiprocessing
- **Caching**: Intelligentes Caching von häufig verwendeten Datenstrukturen
- **Lazy Loading**: Lazy Loading von Daten, die nicht sofort benötigt werden

Transformation-Optimierung

Die Transformation kann für große Modelle zeitaufwändig sein:

- **Inkrementelle Transformation**: Nur geänderte Teile werden transformiert
- **Parallelisierung**: Parallele Transformation unabhängiger Komponenten
- **Caching**: Caching von Transformationsergebnissen
- **Optimierte Datenstrukturen**: Verwendung effizienter Datenstrukturen (z. B. Sets statt Listen für Lookups)

Code-Generierung-Optimierung

Die Code-Generierung kann für große Modelle langsam sein:

- **Template-Caching**: Caching von kompilierten Templates
- **Inkrementelle Generierung**: Nur geänderte Teile werden neu generiert
- **Parallelisierung**: Parallele Generierung unabhängiger Komponenten
- **Optimierte String-Operationen**: Verwendung von String-Buildern statt String-Konkatenation

7.4.2. Fehlerbehandlung und Robustheit

Robuste Fehlerbehandlung ist essentiell für die praktische Nutzbarkeit:

Fehlerklassifikation

Fehler werden klassifiziert nach Schweregrad:

- **Kritische Fehler**: Verhindern die Transformation (z. B. fehlende erforderliche Attribute)
- **Warnungen**: Beeinträchtigen die Qualität, aber erlauben die Transformation (z. B. fehlende optionale Attribute)
- **Informationen**: Hinweise auf potenzielle Probleme (z. B. ungewöhnliche Werte)

Fehlerbehandlung-Strategien

Verschiedene Strategien für verschiedene Fehlertypen:

- **Default-Werte**: Verwendung von Default-Werten für fehlende optionale Attribute
- **Schätzungen**: Intelligente Schätzungen basierend auf ähnlichen Komponenten
- **Fehler-Propagierung**: Strukturierte Fehler-Propagierung mit Kontext-Informationen
- **Fehler-Logging**: Detailliertes Logging für Debugging und Analyse

7.4.3. Erweiterbarkeit und Wartbarkeit

Das Framework ist so designed, dass es einfach erweitert und gewartet werden kann:

Plugin-Architektur

Eine Plugin-Architektur ermöglicht die einfache Erweiterung:

- **Adapter-Plugins**: Neue Export-Formate können als Plugins hinzugefügt werden
- **Generator-Plugins**: Neue Zielplattformen können als Plugins hinzugefügt werden
- **Validator-Plugins**: Neue Validierungsregeln können als Plugins hinzugefügt werden
- **Metrik-Plugins**: Neue Synthese-Metriken können als Plugins hinzugefügt werden

Modularität

Das Framework ist modular aufgebaut:

- **Klare Schnittstellen**: Wohldefinierte Schnittstellen zwischen Modulen
- **Lose Kopplung**: Module sind lose gekoppelt und können unabhängig entwickelt werden
- **Hohe Kohäsion**: Module haben hohe Kohäsion und erfüllen klar definierte Aufgaben
- **Dependency Injection**: Dependency Injection ermöglicht einfaches Testen und Erweitern

7.5. Erweiterte Implementierungs-Beispiele

Dieser Abschnitt präsentiert detaillierte Code-Beispiele für die Implementierung.

7.5.1. Beispiel: Parser-Implementierung

JSON-Parser mit Streaming

```
import ijson

def parse_preevision_export_streaming(file_path):
    """Parser mit Streaming für große Dateien"""
```

```
    with open(file_path, 'rb') as f:
        parser = ijson.parse(f)

        current_ecu = None
        ecus = []

        for prefix, event, value in parser:
            if prefix == 'ecus.item.name':
                current_ecu = {'name': value, 'interfaces': []}
            elif prefix == 'ecus.item.interfaces.item':
                if current_ecu:
                    current_ecu['interfaces'].append(value)
            elif prefix == 'ecus.item' and event == 'end_map':
                if current_ecu:
                    ecus.append(current_ecu)
                    current_ecu = None

        return ecus
```

XML-Parser mit iterparse

```
from xml.etree.ElementTree import iterparse

def parse_preevision_export_xml(file_path):
    """XML-Parser mit iterparse für große Dateien"""
    ecus = []
    current_ecu = None

    for event, elem in iterparse(file_path, events=('start', 'end')):
        if event == 'start' and elem.tag == 'ECU':
            current_ecu = {'name': elem.get('name'), 'interfaces': []}
        elif event == 'end' and elem.tag == 'Interface':
            if current_ecu:
                current_ecu['interfaces'].append({
                    'name': elem.get('name'),
                    'type': elem.get('type'),
                    'bandwidth': float(elem.get('bandwidth', 0))
                })
        elif event == 'end' and elem.tag == 'ECU':
```

```
            if current_ecu:
                ecus.append(current_ecu)
                current_ecu = None
            elem.clear()  # Speicher freigeben

    return ecus
```

7.5.2. Beispiel: Regel-Engine-Implementierung

YAML-Regel-Laden

```
import yaml
from typing import Dict, Any

class RuleEngine:
    def __init__(self, rules_file: str):
        with open(rules_file, 'r') as f:
            self.rules = yaml.safe_load(f)

    def transform(self, source_data: Dict[str, Any],
                  rule_name: str) -> Dict[str, Any]:
        """Transformiert Daten basierend auf Regel"""
        rule = self.rules['mappings'][rule_name]
        target_data = {}

        for target_attr, source_mapping in rule['attributes'].items():
            if isinstance(source_mapping, str):
                # Direkte Attribut-Zuordnung
                target_data[target_attr] = source_data.get(source_mapping)
            elif callable(source_mapping):
                # Transformations-Funktion
                target_data[target_attr] = source_mapping(source_data)

        return target_data
```

7.5.3. Beispiel: Code-Generator-Implementierung

Jinja2-Template-basierte Generierung

```
from jinja2 import Template, Environment, FileSystemLoader
```

```
class CodeGenerator:
    def __init__(self, template_dir: str):
        self.env = Environment(
            loader=FileSystemLoader(template_dir),
            trim_blocks=True,
            lstrip_blocks=True
        )

    def generate_omnet_config(self, architecture: Dict,
                              output_path: str):
        """Generiert OMNeT++ Konfiguration"""
        template = self.env.get_template('omnet_config.ini.j2')

        config = template.render(
            network_name=architecture['name'],
            nodes=architecture['nodes'],
            links=architecture['links']
        )

        with open(output_path, 'w') as f:
            f.write(config)
```

7.6. Zusammenfassung

Dieses Kapitel hat die technische Realisierung der Transformation mit modernen Technologien und Best Practices beschrieben. Die wichtigsten Aspekte umfassen:

- **Datenaufnahme und Normalisierung**: Strukturierte Exporte mit modernen Formaten (JSON, CSV, XML, GraphQL, gRPC, Parquet), intelligente Parser mit Streaming-Unterstützung, Adapter-Pattern für einheitliche Verarbeitung, und Normalisierung mit automatischer Typ-Inferenz und Datenbereinigung.
- **Regel-Engine und Generator**: YAML-basierte Mapping-Regeln mit Validierung, Template-basierte Code-Generierung (Jinja2), plattform-spezifische Generatoren für verschiedene Simulationsplattformen, und Konfigurationsgeneratoren für verschiedene Szenarien.
- **Automatisierte Validierung und CI/CD**:

  - Schema-Validierung mit JSON Schema und Custom Validators
  - Konsistenzchecks für Cross-References und Constraints
  - Moderne CI/CD-Pipelines (GitHub Actions v4, GitLab CI)
  - Containerisierung mit Docker für reproduzierbare Umgebungen
  - Kubernetes für skalierbare parallele Ausführung
  - Cloud-native Deployment (Serverless, Container Orchestration)

- **Versionierung und Reproduzierbarkeit**:
  - Semantic Versioning und automatische Tag-Generierung
  - Container-Registries für Image-Management
  - Artifact-Storage für Simulations-Ergebnisse
  - Deterministische Generierung und Dependency-Pinning
  - Environment-Dokumentation mit Container-Files

Die technische Realisierung nutzt moderne DevOps-Praktiken, Containerisierung und Cloud-native Ansätze, um eine automatisierte, validierte, skalierbare und reproduzierbare Transformation von Architekturmodellen in Simulationsmodelle zu ermöglichen. Dies ermöglicht es, große Modelle effizient zu verarbeiten, Simulationen parallel auszuführen und die Ergebnisse zuverlässig zu reproduzieren.

8. Szenarien und Use-Cases

Dieses Kapitel beschreibt die verschiedenen Szenarien und Use-Cases, die für die Simulation und Bewertung der E/E-Architektur verwendet werden. Die Szenarien umfassen nominale Betriebsbedingungen, Stress-Szenarien, Fehlerszenarien und Design-Varianten. Eine umfassende Szenario-Abdeckung ist essentiell, um die Robustheit und Zuverlässigkeit der Architektur unter verschiedenen Bedingungen zu bewerten [?, ?].

Die Definition und Durchführung von Szenarien folgt etablierten Methoden der Fahrzeugentwicklung [?] und berücksichtigt die spezifischen Anforderungen von Nutzfahrzeugen [?]. Jedes Szenario wird detailliert beschrieben, parametriert und mit erwarteten Ergebnissen versehen.

8.1. Nominal-Szenarien

Nominal-Szenarien beschreiben den normalen Betrieb der Architektur unter typischen Bedingungen. Diese Szenarien dienen als Baseline für die Bewertung der Architektur-Performance und bilden die Grundlage für den Vergleich mit Stress- und Fehlerszenarien.

8.1.1. Standard-Betrieb

Nominal-Szenarien beschreiben den normalen Betrieb der Architektur unter typischen Bedingungen:

- **Lastverteilung**: Normale CPU/GPU/Netzwerk-Auslastung
  - CPU-Auslastung: 40-60% für typische ECUs
  - GPU-Auslastung: 30-50% für Perzeptions-Tasks
  - Netzwerk-Auslastung: 30-50% für Ethernet-Links
  - Keine Überlastung, ausreichend Reserve für Spitzenlasten
- **Timing**: Alle Frames und Tasks innerhalb ihrer Periodizitäten
  - Alle Tasks erfüllen ihre Deadlines
  - E2E-Latenzen innerhalb der Budgets

  - Jitter innerhalb akzeptabler Grenzen ($< 10\%$ der Periodizität)
  - Keine Deadline-Misses

- **Kommunikation**: Keine Paketverluste, normale Latenzen
  - Paketverlustrate: $< 10^{-6}$ für sicherheitskritische Frames
  - Netzwerk-Latenzen: Innerhalb der Budgets (typisch 1-5 ms für TSN)
  - Keine Queue-Überläufe
  - Stabile Kommunikation ohne Retransmissions

- **Energie**: Standard-Energieverbrauch
  - Energieverbrauch entsprechend Lastprofil
  - Power-States entsprechend Aktivität
  - Keine unerwarteten Power-Spikes

8.1.2. Typische Fahrzyklen

Verschiedene Fahrzyklen werden modelliert, um unterschiedliche Betriebsbedingungen abzudecken. Jeder Fahrzyklus hat charakteristische Lastprofile und Timing-Anforderungen:

- **Stadtverkehr**: Häufige Starts/Stopps, niedrige Geschwindigkeiten
  - **Charakteristik**: Viele Brems- und Beschleunigungsvorgänge, häufige Spurwechsel
  - **Sensoren (Bosch)**:
    * 8x Bosch 8MP Multifunktionskameras: Hohe Auflösung für präzise Objekterkennung in komplexen Szenen
    * 8x Bosch Long-Range-Radar: Für Geschwindigkeits- und Abstandsmessung
    * 4x Bosch High-Resolution-LiDAR: Für präzise 3D-Umgebungserfassung
    * 16x Ultraschall-Sensoren: Für Nahbereichserkennung
  - **Rechenplattform (NVIDIA DRIVE Thor)**:
    * GPU-Last: 60-70% für Perzeptions-Tasks (YOLOv8 auf 8MP Kameras)
    * CPU-Last: 40-50% für Planung und Regelung
    * Echtzeit-Verarbeitung von bis zu 12 Kameras gleichzeitig
  - **Lastprofil**:

        * Perzeption: Hohe Aktivität (viele Objekte, Fußgänger, Radfahrer)
        * Planung: Häufige Replanung erforderlich
        * Regelung: Häufige Lenk- und Bremskommandos
    - **Netzwerk**: Moderate bis hohe Last durch viele Sensor-Updates (Ethernet 2.5G für Kameras, 10G für LiDAR)
    - **Energie**: Variabler Verbrauch durch häufige Beschleunigungen, DRIVE Thor: 80-120 W
    - **Beispiel-Dauer**: 30 Minuten simulierter Stadtverkehr
- **Autobahn**: Hohe Geschwindigkeiten, konstante Last
    - **Charakteristik**: Hohe konstante Geschwindigkeit, wenig Spurwechsel
    - **Lastprofil**:
        * Perzeption: Moderate Aktivität (weniger Objekte, aber höhere Geschwindigkeit)
        * Planung: Weniger Replanung erforderlich
        * Regelung: Konstante Lenkkommandos für Spurhaltung
    - **Netzwerk**: Stabile Last, weniger Variationen
    - **Energie**: Relativ konstanter Verbrauch
    - **Beispiel-Dauer**: 60 Minuten simulierter Autobahnfahrt
- **Landstraße**: Variable Geschwindigkeiten, moderate Last
    - **Charakteristik**: Variable Geschwindigkeiten, Kurven, Überholvorgänge
    - **Lastprofil**:
        * Perzeption: Moderate bis hohe Aktivität
        * Planung: Periodische Replanung
        * Regelung: Variable Lenk- und Bremskommandos
    - **Netzwerk**: Variable Last
    - **Energie**: Variabler Verbrauch
- **Parkplatz**: Niedrige Last, viele Sensoren aktiv
    - **Charakteristik**: Niedrige Geschwindigkeit, viele Objekte, enge Räume
    - **Lastprofil**:
        * Perzeption: Sehr hohe Aktivität (viele Objekte, enge Räume)

    - Planung: Häufige Replanung für enge Manöver
    - Regelung: Präzise Lenk- und Bremskommandos
  - **Netzwerk**: Hohe Last durch viele Sensor-Updates (Ultraschall, Kameras)
  - **Energie**: Niedriger Verbrauch (niedrige Geschwindigkeit), aber hohe Rechenlast

Tabelle 8.1.: Vergleich typischer Fahrzyklen

Fahrzyklus	CPU-Last	Netzwerk-Last	E2E-Latenz	Energie
Stadtverkehr	55-65%	40-50%	80-95 ms	Variabel
Autobahn	45-55%	30-40%	70-85 ms	Konstant
Landstraße	50-60%	35-45%	75-90 ms	Variabel
Parkplatz	60-70%	50-60%	85-100 ms	Niedrig

8.2. Stress-Szenarien

8.2.1. Stau-Szenario

Ein Stau-Szenario simuliert hohe Last durch viele Objekte:

- **Perzeption**: Viele Objekte in der Szene (50+ Fahrzeuge, Fußgänger)
- **Netzwerk**: Hohe Datenraten durch viele Sensor-Streams
- **CPU**: Hohe Auslastung durch komplexe Objekterkennung
- **Ziel**: Prüfung, ob Deadlines auch unter hoher Last eingehalten werden

8.2.2. Wetter-Szenario

Schlechte Wetterbedingungen erhöhen die Verarbeitungslast:

- **Regen/Schnee**: Reduzierte Sicht, mehr Bildverarbeitung erforderlich
- **Nebel**: Zusätzliche Filterung und Fusion erforderlich
- **Starke Sonne**: Blendung, erhöhte Bildverarbeitung
- **Ziel**: Prüfung der Robustheit unter schwierigen Bedingungen

8.2.3. Extreme Last-Szenarien

- **Maximale Datenrate**: Alle Sensoren mit maximaler Framerate
- **Maximale CPU-Last**: Alle Tasks gleichzeitig aktiv
- **Maximale Netzwerk-Auslastung**: Alle Links nahe 100% Auslastung
- **Ziel**: Identifikation von Bottlenecks und Grenzen

8.3. Fehlerszenarien

8.3.1. ECU-Ausfall

Einzelner ECU-Ausfall

- **Ausfallzeitpunkt**: Zufällig während der Simulation
- **Ausfallmodus**: Sofortiger Ausfall oder Graduelle Degradation
- **Auswirkungen**:
  - Verlust von Funktionen auf dem ausgefallenen ECU
  - Erhöhte Last auf anderen ECUs (falls Redundanz vorhanden)
  - Kommunikationsausfälle
- **Ziel**: Prüfung der Fehlertoleranz und Redundanz-Mechanismen

Mehrfach-Ausfälle

- **Simultane Ausfälle**: Mehrere ECUs fallen gleichzeitig aus
- **Kaskadierende Ausfälle**: Ein Ausfall führt zu weiteren Ausfällen
- **Ziel**: Prüfung der System-Robustheit

8.3.2. Link-Ausfall

Einzelner Link-Ausfall

- **Link-Typen**: Ethernet, CAN, LIN
- **Ausfallmodi**:
  - Kompletter Ausfall (Kabelbruch)
  - Degradierte Bandbreite (Störung)

  - Erhöhte Latenz (Überlastung)

- **Auswirkungen**:
  - Kommunikationsausfälle
  - Erhöhte Latenzen über alternative Routen
  - Paketverluste

Redundante Pfade

- **PRP/HSR**: Parallel Redundancy Protocol / High-availability Seamless Redundancy
- **Dual-Path**: Zwei unabhängige Kommunikationspfade
- **Ziel**: Prüfung der Redundanz-Mechanismen

8.3.3. Switch-Queue-Überlauf

Überlastung

- **Ursache**: Zu viele Frames für verfügbare Bandbreite
- **Auswirkung**: Queue-Überlauf, Paketverluste
- **Ziel**: Identifikation von Bandbreiten-Bottlenecks

Priorisierung

- **TSN-Prioritäten**: Prüfung, ob Priorisierung funktioniert
- **Deadline-Misses**: Prüfung, ob kritische Frames bevorzugt werden

8.3.4. Clock-Drift

Zeitsynchronisation

- **TSN-Synchronisation**: gPTP (IEEE 802.1AS) Clock-Drift
- **Auswirkung**: Timing-Fehler, Jitter-Erhöhung
- **Ziel**: Prüfung der Robustheit gegen Synchronisations-Fehler

8.4. Design-Varianten

8.4.1. Redundanz-Varianten

Verschiedene Redundanz-Grade

- **Keine Redundanz**: Single-Point-of-Failure
- **2-fach Redundanz**: 1-out-of-2 Voting
- **3-fach Redundanz**: 2-out-of-3 Voting
- **Ziel**: Vergleich von Verfügbarkeit vs. Kosten

Redundanz-Platzierung

- **ECU-Redundanz**: Redundante ECUs
- **Link-Redundanz**: Redundante Kommunikationspfade
- **Software-Redundanz**: Redundante Software-Komponenten

8.4.2. Bandbreiten-Varianten

Verschiedene Ethernet-Geschwindigkeiten

- **1 GbE**: Standard-Geschwindigkeit
- **2.5 GbE**: Erhöhte Geschwindigkeit
- **10 GbE**: Hohe Geschwindigkeit
- **Ziel**: Kosten-Nutzen-Analyse

Bandbreiten-Allokation

- **Gleichmäßige Verteilung**: Alle Links gleiche Bandbreite
- **Priorisierte Allokation**: Kritische Links erhalten mehr Bandbreite
- **Dynamische Allokation**: Bandbreite wird dynamisch zugewiesen

8.4.3. Scheduling-Varianten

Verschiedene Scheduling-Policies

- **Fixed Priority**: Feste Prioritäten
- **Earliest Deadline First**: Dynamische Prioritäten
- **Time-Triggered**: Zeitgesteuertes Scheduling
- **Ziel**: Vergleich von Scheduling-Strategien

Prioritäts-Zuordnung

- **Deadline-basiert**: Priorität basierend auf Deadline
- **ASIL-basiert**: Priorität basierend auf Safety-Level
- **Hybrid**: Kombination verschiedener Kriterien

8.5. Systematische Variation

8.5.1. Parameter-Sweeps

Systematische Variation von Parametern:

- **Framegröße**: ±20% Variation
- **Periodizität**: ±10% Variation
- **WCET**: ±15% Variation
- **Bandbreite**: ±25% Variation

8.5.2. Design-Space-Exploration

- **Multi-Objective-Optimierung**: Trade-offs zwischen Latenz, Kosten, Energie
- **Pareto-Front**: Identifikation optimaler Design-Punkte
- **Sensitivitäts-Analyse**: Identifikation kritischer Parameter

8.6. Use-Case-spezifische Szenarien

8.6.1. Autonomes Fahren (L3/L4)

- **Highway-Pilot**: Autobahn-Autopilot
- **Urban-Pilot**: Stadt-Autopilot
- **Parking**: Automatisches Einparken
- **Anforderungen**:
  - E2E-Latenz < 100 ms
  - ASIL D für kritische Funktionen
  - Hohe Verfügbarkeit

8.6.2. Fahrerassistenz (L2)

- **ACC**: Adaptive Cruise Control
- **LKA**: Lane Keeping Assist
- **AEB**: Automatic Emergency Braking
- **Anforderungen**:
  - E2E-Latenz < 150 ms
  - ASIL B/C für kritische Funktionen

8.6.3. Fracht-/Laderaum-Management

- **Temperatur-Überwachung**: Kühlung für Fracht
- **Tür-Überwachung**: Sicherheit für Ladung
- **Gewichts-Überwachung**: Lastverteilung
- **Anforderungen**:
  - Kontinuierliche Überwachung
  - Niedrige Latenz für Alarme
  - Energieeffizienz

8.7. Erweiterte Szenario-Beispiele

Dieser Abschnitt präsentiert detaillierte Beispiele für komplexe Szenarien, die in der Simulation verwendet werden.

8.7.1. Beispiel: Vollständiger Stadtverkehr-Zyklus

Dieses Beispiel zeigt einen vollständigen Stadtverkehr-Zyklus mit allen Phasen und Übergängen.

Phasen des Zyklus

Der Zyklus umfasst mehrere Phasen:

1. **Startphase** (0-5 min):
    - Fahrzeugstart, Systeminitialisierung
    - Alle Sensoren aktivieren
    - Kalibrierung der Sensoren
    - Netzwerk-Synchronisation (gPTP)
    - Last: Niedrig (Initialisierung)
2. **Fahrphase** (5-25 min):
    - Normale Fahrt durch Stadt
    - Viele Objekte (Fahrzeuge, Fußgänger, Radfahrer)
    - Häufige Brems- und Beschleunigungsvorgänge
    - Spurwechsel und Abbiegen
    - Last: Hoch (viele Objekte, komplexe Szenen)
3. **Stauphase** (25-35 min):
    - Stau auf Autobahn
    - Sehr viele Objekte (50-100)
    - Niedrige Geschwindigkeit (0-20 km/h)
    - Häufige Stopps und Starts
    - Last: Sehr hoch (extreme Objektdichte)
4. **Parkphase** (35-40 min):
    - Einparken auf Parkplatz

- Viele Objekte, enge Räume
- Präzise Manöver erforderlich
- Alle Sensoren aktiv (Ultraschall, Kameras)
- Last: Sehr hoch (viele Objekte, enge Räume)

5. **Endphase** (40-45 min):
   - Fahrzeug abstellen
   - System-Shutdown
   - Daten-Speicherung
   - Last: Niedrig (Shutdown)

Lastprofile

Die Lastprofile für jede Phase:

Tabelle 8.2.: Lastprofile: Stadtverkehr-Zyklus

Phase	CPU-Last	GPU-Last	Netzwerk-Last	Objektdichte
Start	20%	10%	15%	0-5
Fahrt	60%	55%	45%	20-40
Stau	85%	75%	70%	50-100
Park	80%	70%	65%	30-60
Ende	15%	5%	10%	0-5

8.7.2. Beispiel: Wetter-Szenarien

Dieses Beispiel zeigt verschiedene Wetter-Szenarien und deren Auswirkungen auf die Architektur.

Regen-Szenario

Regen beeinflusst die Perzeption erheblich:

- **Kamera**:
  - Reduzierte Sichtweite (50-70%)
  - Reflexionen auf der Straße
  - Wassertropfen auf der Scheibe
  - Erhöhte Bildverarbeitung erforderlich (Filter, Entrauschung)

  - CPU-Last: +15-20%

- **Radar**:
  - Weniger beeinflusst durch Regen
  - Leichte Reduzierung der Reichweite
  - Wichtig für Sensorfusion

- **LiDAR**:
  - Stärker beeinflusst durch Regen
  - Wassertropfen reflektieren Laser
  - Reduzierte Punktwolken-Qualität
  - Erhöhte Filterung erforderlich

Nebel-Szenario

Nebel ist besonders herausfordernd:

- **Kamera**:
  - Sehr reduzierte Sichtweite (20-40%)
  - Kontrastverlust
  - Erhöhte Bildverarbeitung (Entnebelung, Kontrastverstärkung)
  - CPU-Last: +25-30%

- **Radar**:
  - Weniger beeinflusst
  - Wird zur primären Perzeptionsquelle
  - Sensorfusion wird kritisch

- **LiDAR**:
  - Stark beeinflusst
  - Nebelpartikel reflektieren Laser
  - Viele Fehldetektionen
  - Erhöhte Filterung erforderlich

Schnee-Szenario

Schnee stellt besondere Herausforderungen:

- **Kamera**:
  - Blendung durch Schnee
  - Kontrastprobleme (weiß auf weiß)
  - Schneeflocken vor der Kamera
  - Erhöhte Bildverarbeitung (Entblendung, Kontrastanpassung)
  - CPU-Last: +20-25%
- **Radar**:
  - Schneeflocken können reflektieren
  - Aber weniger beeinflusst als Kamera/LiDAR
- **LiDAR**:
  - Stark beeinflusst
  - Schneeflocken reflektieren Laser
  - Viele Fehldetektionen

8.8. Erweiterte Szenario-Definitionen

Dieser Abschnitt beschreibt erweiterte Methoden zur Definition von Szenarien.

8.8.1. Parametrisierte Szenarien

Szenarien können parametrisiert werden für verschiedene Konfigurationen:

- **Umgebungs-Parameter**: Wetter, Verkehrsdichte, Straßentyp
- **Fahrzeug-Parameter**: Geschwindigkeit, Beschleunigung, Last
- **System-Parameter**: CPU-Frequenz, Netzwerk-Bandbreite, Sensor-Framerate
- **Fehler-Parameter**: Fehlerrate, Fehlertyp, Fehlerzeitpunkt

8.8.2. Szenario-Kombinationen

Komplexe Szenarien können aus einfachen Szenarien kombiniert werden:

- **Sequenzielle Kombination**: Mehrere Szenarien nacheinander
- **Parallele Kombination**: Mehrere Szenarien gleichzeitig
- **Bedingte Kombination**: Szenarien basierend auf Bedingungen
- **Probabilistische Kombination**: Zufällige Kombinationen

8.8.3. Szenario-Templates

Wiederverwendbare Szenario-Templates:

```
{
  "scenario_template": {
    "name": "urban_driving",
    "parameters": {
      "traffic_density": "high|medium|low",
      "weather": "sunny|rain|fog|snow",
      "speed": "0-50 km/h"
    },
    "phases": [
      {
        "name": "start",
        "duration": "5 min",
        "load": "low"
      },
      {
        "name": "driving",
        "duration": "20 min",
        "load": "high"
      }
    ]
  }
}
```

8.9. Zusammenfassung

Dieses Kapitel hat verschiedene Szenarien und Use-Cases für die Simulation beschrieben:

- **Nominal-Szenarien**: Standard-Betriebsbedingungen
- **Stress-Szenarien**: Stau, Wetter, extreme Last
- **Fehlerszenarien**: ECU-/Link-Ausfälle, Queue-Überlauf, Clock-Drift
- **Design-Varianten**: Redundanz, Bandbreite, Scheduling
- **Use-Case-spezifische Szenarien**: Autonomes Fahren, Fahrerassistenz, Fracht-Management

Diese Szenarien ermöglichen eine umfassende Bewertung der Architektur unter verschiedenen Bedingungen und helfen, Schwachstellen und Optimierungspotenziale zu identifizieren.

9. Simulation und Auswertung

Dieses Kapitel beschreibt die Durchführung von Simulationen und die Auswertung der Ergebnisse. Es umfasst die Definition von Messgrößen, Akzeptanzkriterien und die Aufbereitung der Ergebnisse für die Analyse. Die Simulation von E/E-Architekturen erfordert eine sorgfältige Modellierung der verschiedenen Systemkomponenten und deren Interaktionen [?, ?]. Moderne Simulationsansätze für Fahrzeugsysteme werden in [?, ?] ausführlich beschrieben.

9.1. Messgrößen

Die Messgrößen werden für moderne E/E-Architekturen mit neuesten Sensoren (Bosch 8MP Multifunktionskamera) und Rechenplattformen (NVIDIA DRIVE Thor) definiert. Diese Komponenten stellen neue Anforderungen an die Metriken und deren Messung.

9.1.1. Timing-Metriken

End-to-End-Latenz

Die End-to-End-Latenz (E2E-Latenz) misst die Zeit von der Erzeugung eines Signals an einem Sensor bis zur Ankunft am Aktor. Diese Metrik ist von zentraler Bedeutung für sicherheitskritische Funktionen, da sie direkt die Reaktionszeit des Systems bestimmt [?, ?].

Die E2E-Latenz setzt sich aus mehreren Komponenten zusammen:

$$L_{E2E} = L_{sensor} + L_{network1} + L_{processing1} + L_{network2} + L_{processing2} + \ldots + L_{networkN} + L_{actuator} \tag{9.1}$$

wobei:

- L_{sensor}: Sensor-Verarbeitungszeit (Bildaufnahme, Signalaufbereitung)
- $L_{networki}$: Netzwerk-Latenz für Link i (Serialisierung, Propagation, Queueing, Switching)

- $L_{processing i}$: Verarbeitungszeit auf ECU i (Task-Ausführung, Scheduling-Verzögerung)
- $L_{actuator}$: Aktor-Reaktionszeit
- **Messung**: Zeitstempel am Sensor vs. Zeitstempel am Aktor
  - Synchronisierte Zeitstempel durch TSN gPTP-Synchronisation
  - Präzision: Mikrosekunden-Genauigkeit für TSN-Netze
  - Messung für jedes Frame/Signal in der Chain
- **Statistiken**: Min, Max, Mean, Median, 95. Perzentil, 99. Perzentil
  - **Min**: Best-Case-Latenz (wichtig für Optimierung)
  - **Max**: Worst-Case-Latenz (kritisch für Deadline-Prüfung)
  - **Mean**: Durchschnittliche Latenz (für Performance-Bewertung)
  - **Median**: Typische Latenz (weniger empfindlich gegen Outlier)
  - **95./99. Perzentil**: Latenz, die in 95%/99% der Fälle nicht überschritten wird
- **Chain-spezifisch**: Latenz für jede Funktionskette separat
  - Jede Chain hat eigene E2E-Latenz-Anforderungen
  - Kritische Chains (z. B. Lenkung, Bremse) haben strikte Deadlines (< 100 ms)
  - Nicht-kritische Chains (z. B. Infotainment) haben weniger strikte Anforderungen
- **Ziel**: Prüfung, ob Deadlines eingehalten werden
  - Vergleich der gemessenen E2E-Latenz mit der definierten Deadline
  - Identifikation von Chains, die ihre Deadlines nicht einhalten
  - Analyse der Latenz-Komponenten zur Identifikation von Bottlenecks

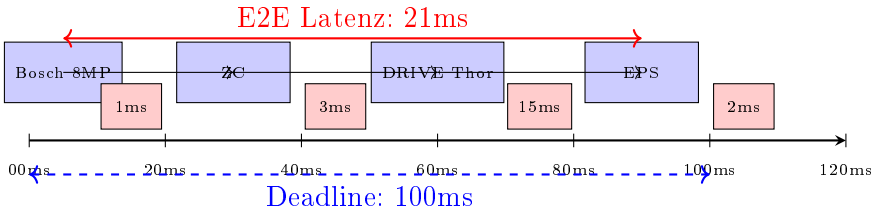

Abbildung 9.1.: Beispiel: E2E-Latenz-Zerlegung für eine Funktionskette

Jitter

Jitter misst die Variabilität der Latenz:

- **Definition**: Standardabweichung oder Differenz zwischen Max und Min
- **Chain-spezifisch**: Jitter für jede Chain
- **Ziel**: Prüfung der Deterministik

Deadline-Misses

Deadline-Misses zählen, wie oft Deadlines verletzt werden:

- **Task-Deadlines**: Anzahl verpasster Task-Deadlines
- **Frame-Deadlines**: Anzahl verpasster Frame-Deadlines
- **Chain-Deadlines**: Anzahl verpasster E2E-Deadlines
- **Ziel**: Identifikation von Timing-Problemen

9.1.2. Kommunikations-Metriken

Busauslastung

Die Busauslastung misst die Bandbreiten-Nutzung:

- **Link-spezifisch**: Auslastung für jeden Link
- **Switch-spezifisch**: Auslastung für jeden Switch-Port
- **Statistiken**: Min, Max, Mean über Zeit
- **Ziel**: Identifikation von Bandbreiten-Bottlenecks

Paketverluste

Paketverluste zählen verlorene oder verworrene Pakete:

- **Ursachen**: Queue-Überlauf, Fehler, Timeouts
- **Frame-spezifisch**: Verluste für jeden Frame-Typ
- **Link-spezifisch**: Verluste für jeden Link
- **Ziel**: Prüfung der Zuverlässigkeit

Queue-Längen

Queue-Längen messen die Warteschlangen-Größe:

- **Switch-Queues**: Länge für jede Queue an jedem Switch
- **Statistiken**: Max, Mean, 95. Perzentil
- **Ziel**: Identifikation von Überlastungen

9.1.3. Ressourcen-Metriken

CPU/GPU-Last

Die CPU/GPU-Last misst die Prozessor-Auslastung:

- **ECU-spezifisch**: Last für jeden ECU
- **Task-spezifisch**: Last für jeden Task
- **Statistiken**: Min, Max, Mean über Zeit
- **Ziel**: Identifikation von CPU-Bottlenecks

Wartezeiten

Wartezeiten messen, wie lange Tasks auf CPU-Zeit warten:

- **Task-spezifisch**: Wartezeit für jeden Task
- **Statistiken**: Max, Mean, 95. Perzentil
- **Ziel**: Identifikation von Scheduling-Problemen

Speicher-Auslastung

Die Speicher-Auslastung misst die RAM-Nutzung:

- **ECU-spezifisch**: Speicher für jeden ECU
- **Task-spezifisch**: Speicher für jeden Task
- **Ziel**: Prüfung von Speicher-Überläufen

9.1.4. Verfügbarkeits-Metriken

MTBF

Mean Time Between Failures (MTBF) misst die durchschnittliche Zeit zwischen Ausfällen:

- **Komponenten-spezifisch**: MTBF für jede Komponente
- **System-MTBF**: Gesamt-MTBF des Systems
- **Ziel**: Bewertung der Zuverlässigkeit

Verfügbarkeit

Die Verfügbarkeit misst den Anteil der Zeit, in der das System funktionsfähig ist:

$$A = \frac{MTBF}{MTBF + MTTR} \tag{9.2}$$

wobei MTTR die Mean Time To Repair ist.

Downtime

Downtime misst die Zeit, in der das System nicht funktionsfähig ist:

- **Total Downtime**: Gesamte Ausfallzeit
- **Number of Failures**: Anzahl der Ausfälle
- **Ziel**: Bewertung der Fehlertoleranz

9.1.5. Energie-Metriken

Energieverbrauch

Der Energieverbrauch misst den Stromverbrauch:

- **ECU-spezifisch**: Energie für jeden ECU
- **Power-State-spezifisch**: Energie in jedem Power State
- **System-Energie**: Gesamt-Energieverbrauch
- **Ziel**: Bewertung der Energieeffizienz

Power-State-Verteilung

Die Power-State-Verteilung zeigt, wie viel Zeit in jedem Zustand verbracht wird:

- **Duty-Cycle**: Anteil der Zeit in jedem State
- **Übergänge**: Anzahl der State-Übergänge
- **Ziel**: Optimierung der Energieeffizienz

9.2. Akzeptanzkriterien

9.2.1. OEM/Normen-basierte Grenzwerte

Timing-Anforderungen

Tabelle 9.1.: Beispiel: Timing-Anforderungen für verschiedene Funktionen

Funktion	E2E-Latenz	Jitter
Notbremsung (AEB)	< 100 ms	< 10 ms
Lenkung (LKA)	< 150 ms	< 20 ms
ACC	< 200 ms	< 30 ms
Infotainment	< 500 ms	< 50 ms

Sicherheitsanforderungen

- **ASIL D**: Verfügbarkeit $> 99.9\%$
- **ASIL C**: Verfügbarkeit $> 99.5\%$
- **ASIL B**: Verfügbarkeit $> 99\%$
- **ASIL A**: Verfügbarkeit $> 95\%$

9.2.2. TSN-Latenz-/Jitter-Budgets

Für TSN-Netze gelten spezielle Budgets:

- **Kritische Frames**: Latenz < 1 ms, Jitter < 0.1 ms
- **Standard-Frames**: Latenz < 10 ms, Jitter < 1 ms
- **Best-Effort**: Keine Garantien

9.2.3. Performance-Ziele

- **CPU-Auslastung**: $< 80\%$ für Safety-kritische ECUs
- **Netzwerk-Auslastung**: $< 70\%$ für kritische Links
- **Paketverluste**: $< 0.1\%$ für kritische Frames
- **Deadline-Misses**: 0 für Safety-kritische Tasks

9.3. Ergebnisaufbereitung

9.3.1. Dashboards

Python/Plotly-Dashboards

Interaktive Dashboards werden mit Python und Plotly erstellt:

- **Timing-Dashboard**: E2E-Latenzen, Jitter, Deadline-Misses
- **Kommunikations-Dashboard**: Busauslastung, Paketverluste, Queue-Längen
- **Ressourcen-Dashboard**: CPU/GPU-Last, Speicher-Auslastung
- **Verfügbarkeits-Dashboard**: MTBF, Verfügbarkeit, Downtime
- **Energie-Dashboard**: Energieverbrauch, Power-State-Verteilung

Visualisierungen

- **Zeitreihen**: Entwicklung von Metriken über Zeit
- **Histogramme**: Verteilung von Latenzen, Jitter
- **Heatmaps**: Auslastung über verschiedene Komponenten
- **Network-Graphs**: Topologie mit annotierten Metriken

9.3.2. Variantenvergleich

Multi-Variant-Analyse

Verschiedene Design-Varianten werden verglichen:

- **Tabellarischer Vergleich**: Metriken für jede Variante
- **Spider-Diagramme**: Multi-Kriterien-Vergleich
- **Pareto-Front**: Trade-offs zwischen verschiedenen Zielen

Statistische Analyse

- **Signifikanz-Tests**: Prüfung, ob Unterschiede signifikant sind
- **Confidence-Intervals**: Konfidenzintervalle für Metriken
- **Sensitivitäts-Analyse**: Identifikation kritischer Parameter

9.3.3. Trace-basierte Analysen

Event-Traces

Event-Traces zeigen die zeitliche Abfolge von Ereignissen:

- **Task-Ausführungen**: Wann Tasks starten/enden
- **Frame-Übertragungen**: Wann Frames gesendet/empfangen werden
- **State-Übergänge**: Power-State-Übergänge, Fehler-Ereignisse

Critical-Path-Analyse

Der Critical Path identifiziert den längsten Pfad durch das System:

- **Chain-Analyse**: Welche Chain hat die längste Latenz?
- **Bottleneck-Identifikation**: Welche Komponente verursacht die Verzögerung?
- **Optimierungs-Vorschläge**: Wo kann optimiert werden?

Timing-Diagramme

Timing-Diagramme visualisieren die zeitliche Abfolge:

- **Gantt-Charts**: Task-Ausführungen über Zeit
- **Sequence-Diagrams**: Kommunikation zwischen Komponenten
- **State-Diagrams**: State-Übergänge über Zeit

9.4. Erweiterte Simulations-Beispiele

Dieser Abschnitt präsentiert erweiterte Beispiele für die Simulation und Auswertung von E/E-Architekturen. Die Beispiele zeigen, wie verschiedene Szenarien simuliert und ausgewertet werden können.

9.4.1. Beispiel: Komplexe Multi-Sensor-Perzeption

Dieses Beispiel zeigt die Simulation einer komplexen Perzeptions-Pipeline mit mehreren Sensoren:

Architektur

- **Sensoren**:
  - 3x Front-Kameras (Stereo + Mono)
  - 2x Radar (Kurz- und Langstrecke)
  - 1x LiDAR (64-Layer)
  - 8x Ultraschall-Sensoren
- **Verarbeitung**:
  - Bildverarbeitung auf ZC_Front (3x Kamera-Streams)
  - Radar-Signalverarbeitung auf ZC_Front
  - LiDAR-Verarbeitung auf ZC_Front
  - Sensorfusion auf AD-DC
  - Objekterkennung und Tracking auf AD-DC
- **Kommunikation**:
  - Kamera-Streams: Ethernet 1 Gbps, 30 fps, 1500 Byte/Frame
  - Radar-Daten: CAN-FD, 20 Hz, 64 Byte/Frame
  - LiDAR-Daten: Ethernet 2.5 Gbps, 10 Hz, 100 KB/Frame
  - Fused Data: TSN Priority 6, 30 Hz

Simulations-Ergebnisse

Typische Ergebnisse für dieses Szenario:

9.4.2. Beispiel: Stress-Test unter extremer Last

Dieses Beispiel zeigt die Simulation unter extremer Last (Stau-Szenario):

Tabelle 9.2.: Simulations-Ergebnisse: Multi-Sensor-Perzeption

Metrik	Wert	Kommentar
E2E-Latenz (Mean)	45 ms	Innerhalb Budget
E2E-Latenz (Max)	78 ms	Innerhalb Deadline (100 ms)
Jitter	8 ms	Akzeptabel
CPU-Last (ZC_Front)	65%	Moderate Last
CPU-Last (AD-DC)	72%	Hohe Last, aber OK
GPU-Last (AD-DC)	58%	Moderate Last
Netzwerk-Last (Ethernet)	52%	Moderate Last
Paketverluste	0.01%	Sehr niedrig

Szenario-Parameter

- **Objektdichte**: 80 Objekte in Szene
- **Simulationsdauer**: 20 Minuten
- **Geschwindigkeit**: 0-20 km/h (Stau)
- **Ziel**: Prüfung, ob Deadlines auch unter extremer Last eingehalten werden

Ergebnisse

Tabelle 9.3.: Stress-Test Ergebnisse

Metrik	Wert	Status
E2E-Latenz (Max)	112 ms	Über Deadline (100 ms)
Deadline-Misses	2.3%	Nicht akzeptabel
CPU-Last (Peak)	92%	Sehr hoch
Netzwerk-Last (Peak)	78%	Hoch
Queue-Überläufe	15	Problematisch

Analyse und Optimierung

Die Ergebnisse zeigen, dass Optimierungen erforderlich sind:

- **CPU-Bottleneck**: Die CPU-Auslastung ist zu hoch
  - Lösung: Optimierung der Objekterkennungs-Algorithmen
  - Alternative: Zusätzliche CPU-Kerne oder höhere Taktfrequenz
- **Netzwerk-Bottleneck**: Die Netzwerk-Auslastung ist hoch

  - Lösung: Erhöhung der Bandbreite (1 Gbps → 2.5 Gbps)
  - Alternative: Kompression der Sensor-Daten

- **Deadline-Misses**: Einige Deadlines werden verpasst
  - Lösung: Priorisierung kritischer Tasks
  - Alternative: Reduzierung der Objektdichte durch Filterung

9.5. Erweiterte Auswertungs-Methoden

Dieser Abschnitt beschreibt erweiterte Methoden zur Auswertung von Simulationsergebnissen.

9.5.1. Statistische Analyse

Statistische Methoden ermöglichen eine fundierte Analyse der Ergebnisse:

Verteilungsanalyse

Die Verteilung von Metriken wird analysiert:

- **Normalverteilung**: Prüfung auf Normalverteilung (Shapiro-Wilk-Test)
- **Andere Verteilungen**: Weibull, Exponential, etc.
- **Outlier-Erkennung**: Identifikation von Ausreißern (IQR-Methode, Z-Score)
- **Perzentile**: Berechnung von Perzentilen (P50, P95, P99)

Korrelationsanalyse

Korrelationen zwischen Metriken werden analysiert:

- **Pearson-Korrelation**: Lineare Korrelationen
- **Spearman-Korrelation**: Monotone Korrelationen
- **Korrelations-Matrix**: Visualisierung von Korrelationen
- **Kausalitäts-Analyse**: Identifikation von Kausalzusammenhängen

9.5.2. Machine-Learning-basierte Analyse

ML-Methoden können für erweiterte Analysen verwendet werden:

- **Clustering**: Identifikation von ähnlichen Szenarien
- **Anomalie-Erkennung**: Automatische Erkennung von Anomalien
- **Prädiktion**: Vorhersage von Metriken basierend auf Parametern
- **Feature-Importance**: Identifikation wichtiger Parameter

9.6. Erweiterte Simulations-Beispiele

Dieser Abschnitt präsentiert detaillierte Beispiele für Simulationsergebnisse.

9.6.1. Beispiel: Detaillierte E2E-Latenz-Analyse

Perzeption-zu-Aktorik Chain

Eine typische Perzeption-zu-Aktorik Chain:

- **Sensor**: Front-Kamera (30 fps, 1920x1080)
- **ZC Front**: Gateway, Preprocessing (5 ms)
- **AD-DC**: Objekterkennung (15 ms), Tracking (3 ms), Planning (8 ms), Control (1 ms)
- **Aktor**: EPS (Lenkung, 2 ms)

Timing-Zerlegung

9.6.2. Beispiel: Last-Analyse

CPU-Last-Verteilung

Die CPU-Last verteilt sich wie folgt:

Netzwerk-Last-Verteilung

Die Netzwerk-Last verteilt sich wie folgt:

Tabelle 9.4.: Timing-Zerlegung: Perzeption-zu-Aktorik Chain

Komponente	Verarbeitung	Kommunikation	Gesamt
Sensor	2 ms	–	2 ms
ZC Front	5 ms	2 ms	7 ms
AD-DC (Detection)	15 ms	3 ms	18 ms
AD-DC (Tracking)	3 ms	0.5 ms	3.5 ms
AD-DC (Planning)	8 ms	0.5 ms	8.5 ms
AD-DC (Control)	1 ms	1 ms	2 ms
Aktor	2 ms	–	2 ms
E2E Total	**36 ms**	**7 ms**	**43 ms**

Tabelle 9.5.: CPU-Last-Verteilung: AD-DC

Task	CPU-Core	Last	Priorität
Perception	Core 0	45%	10
Tracking	Core 0	12%	9
Planning	Core 1	28%	8
Control	Core 1	8%	7
System	Alle	5%	–
Total	–	**98%**	–

9.7. Zusammenfassung

Dieses Kapitel hat die Simulation und Auswertung beschrieben:

- **Messgrößen**: Timing, Kommunikation, Ressourcen, Verfügbarkeit, Energie
- **Akzeptanzkriterien**: OEM/Normen-basierte Grenzwerte, TSN-Budgets, Performance-Ziele
- **Ergebnisaufbereitung**: Dashboards, Variantenvergleich, Trace-basierte Analysen

Die systematische Auswertung ermöglicht eine umfassende Bewertung der Architektur und identifiziert Optimierungspotenziale.

9.8. Erweiterte Visualisierungs-Methoden

Dieser Abschnitt beschreibt erweiterte Methoden zur Visualisierung von Simulationsergebnissen.

Tabelle 9.6.: Netzwerk-Last-Verteilung: TSN-Backbone

Traffic-Class	Bandbreite	Last	Priorität
Safety-Critical	200 Mbps	20%	7
Real-Time	300 Mbps	30%	6
Best-Effort	500 Mbps	50%	0-5
Total	**1000 Mbps**	**100%**	–

9.8.1. 3D-Visualisierungen

3D-Visualisierungen ermöglichen räumliche Darstellungen:

- **Architektur-3D**: Räumliche Darstellung der Architektur
- **Last-3D**: 3D-Darstellung der Lastverteilung
- **Timing-3D**: 3D-Darstellung von Timing-Beziehungen
- **Interaktive Exploration**: Interaktive Exploration der Ergebnisse

9.8.2. Heatmaps

Heatmaps ermöglichen intuitive Darstellungen:

- **Last-Heatmaps**: Darstellung der Lastverteilung
- **Latenz-Heatmaps**: Darstellung der Latenzverteilung
- **Fehler-Heatmaps**: Darstellung von Fehlerverteilungen
- **Time-Series-Heatmaps**: Zeitreihen-Darstellungen

9.8.3. Interaktive Dashboards

Interaktive Dashboards ermöglichen explorative Analyse:

- **Filter**: Filterung nach verschiedenen Kriterien
- **Zoom**: Zoom in verschiedene Bereiche
- **Drill-Down**: Detaillierte Analyse einzelner Aspekte
- **Export**: Export von Ergebnissen

10. Validierung und Iteration

Dieses Kapitel beschreibt die Validierung der Simulationsergebnisse und den iterativen Prozess zur Verbesserung der Architektur. Die Validierung umfasst insbesondere die Überprüfung der Modellierung von neuesten Sensoren (Bosch 8MP Multifunktionskamera) und Rechenplattformen (NVIDIA DRIVE Thor) sowie die Korrektheit der abgeleiteten Simulationsparameter. Die Validierung umfasst Plausibilisierung gegen analytische Modelle, Sensitivitätsanalysen und Rückkopplung ins Architekturmodell. Die Validierung ist ein kritischer Schritt, um sicherzustellen, dass die Simulationsergebnisse korrekt sind und als Grundlage für Architekturentscheidungen verwendet werden können [?, ?]. Ohne eine sorgfältige Validierung besteht die Gefahr, dass falsche Annahmen oder Implementierungsfehler zu unzuverlässigen Ergebnissen führen, die wiederum zu suboptimalen oder sogar gefährlichen Architekturentscheidungen führen können.

Die Validierung von E/E-Architekturen erfordert insbesondere die Berücksichtigung von Echtzeit-Anforderungen [?] und funktionaler Sicherheit [?]. Moderne Validierungsansätze für Fahrzeugsysteme werden in [?, ?] beschrieben.

10.1. Plausibilisierung gegen analytische Modelle

Die Plausibilisierung ist der erste Schritt der Validierung und dient dazu, die Simulationsergebnisse gegen bekannte analytische Modelle zu vergleichen. Analytische Modelle sind mathematische Formeln, die unter bestimmten Annahmen exakte oder approximative Lösungen für Timing- und Lastprobleme liefern. Der Vergleich zwischen Simulation und Analytik ermöglicht es, systematische Fehler in der Simulation zu identifizieren und die Genauigkeit der Simulationsergebnisse zu quantifizieren.

10.1.1. Analytische Timing-Modelle

Timing-Modelle sind von zentraler Bedeutung für die Validierung, da sie die zeitlichen Eigenschaften des Systems beschreiben. In E/E-Architekturen sind insbesondere die Response Times von Tasks, die End-to-End-Latenzen von Funktionsketten und die Netzwerk-Latenzen kritisch.

CPU-Scheduling-Analyse

Für Fixed-Priority Scheduling (FPS) kann die Worst-Case Response Time (WCRT) analytisch berechnet werden. Fixed-Priority Scheduling ist ein Scheduling-Algorithmus, bei dem jeder Task eine feste Priorität zugewiesen wird und der Scheduler immer den Task mit der höchsten Priorität ausführt, der bereit ist. Die WCRT ist die maximale Zeit, die ein Task vom Zeitpunkt seiner Bereitstellung bis zum Abschluss seiner Ausführung benötigt.

Die iterative Berechnung der WCRT erfolgt nach der Formel von Joseph und Pandya:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{10.1}$$

wobei:

- R_i: Response Time von Task i (zu berechnende Größe)
- C_i: Worst-Case Execution Time (WCET) von Task i - die maximale Zeit, die Task i benötigt, um vollständig ausgeführt zu werden
- $hp(i)$: Menge aller Tasks mit höherer Priorität als Task i
- T_j: Periodizität (Period) von Task j - das Zeitintervall zwischen zwei aufeinanderfolgenden Bereitstellungen
- $\left\lceil \frac{R_i}{T_j} \right\rceil$: Anzahl der Unterbrechungen durch Task j während der Ausführung von Task i

Die Berechnung erfolgt iterativ, beginnend mit $R_i^{(0)} = C_i$ und fortgesetzt mit $R_i^{(k+1)} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j$, bis Konvergenz erreicht wird ($R_i^{(k+1)} = R_i^{(k)}$) oder bis $R_i^{(k)} > D_i$ (Deadline), was bedeutet, dass der Task seine Deadline nicht einhalten kann.

Tabelle 10.1.: Beispiel: WCRT-Berechnung für drei Tasks

Task	Priorität	C_i (ms)	T_i (ms)	D_i (ms)
τ_1	3 (höchste)	2.0	10	10
τ_2	2	3.0	20	20
τ_3	1 (niedrigste)	5.0	50	50

Für Task τ_3 (niedrigste Priorität) ergibt sich:

- $R_3^{(0)} = 5.0$ ms

- $R_3^{(1)} = 5.0 + \lceil \frac{5.0}{10} \rceil \cdot 2.0 + \lceil \frac{5.0}{20} \rceil \cdot 3.0 = 5.0 + 1 \cdot 2.0 + 1 \cdot 3.0 = 10.0$ ms
- $R_3^{(2)} = 5.0 + \lceil \frac{10.0}{10} \rceil \cdot 2.0 + \lceil \frac{10.0}{20} \rceil \cdot 3.0 = 5.0 + 1 \cdot 2.0 + 1 \cdot 3.0 = 10.0$ ms
- Konvergenz: $R_3 = 10.0$ ms $< D_3 = 50$ ms $\Rightarrow$ Task ist schedulierbar

Die simulierten Response Times werden gegen diese analytischen Bounds validiert. Eine Abweichung von mehr als 10% deutet auf einen Fehler in der Simulation hin und muss untersucht werden.

Erweiterte Scheduling-Analyse Für komplexere Scheduling-Algorithmen wie Earliest Deadline First (EDF) oder TSN-triggered Scheduling sind andere analytische Modelle erforderlich:

- **EDF-Scheduling**:
  - Utilisation-Test: $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$ (nur für unabhängige Tasks)
  - Processor Demand Criterion für abhängige Tasks
  - Response Time Analysis für EDF mit Blocking
- **TSN-triggered Scheduling**:
  - Berücksichtigung von Gate-Schedules in der Response Time
  - Integration von Netzwerk-Latenzen in Task-Response Times
  - Analyse von End-to-End-Latenzen über Netzwerk und CPU
- **Multi-Core-Scheduling**:
  - Partitioned Scheduling: Tasks werden festen Kernen zugewiesen
  - Global Scheduling: Tasks können auf verschiedenen Kernen laufen
  - Inter-Core-Interferenz: Cache-Konflikte, Memory-Bandbreite

Netzwerk-Latenz-Analyse

Für TSN-Netze (Time-Sensitive Networking) kann die maximale Latenz analytisch berechnet werden. TSN ist eine Erweiterung von Ethernet, die deterministische Kommunikation mit garantierten Latenzen und Bandbreiten ermöglicht. Die Latenzberechnung in TSN-Netzen ist komplexer als in herkömmlichen Ethernet-Netzen, da sie Gate-Schedules, Prioritäten und Traffic Shaping berücksichtigen muss.

Die maximale End-to-End-Latenz für einen Frame in einem TSN-Netzwerk setzt sich aus mehreren Komponenten zusammen:

$$L_{max} = L_{fixed} + \sum_{i=1}^{n} \frac{S_i}{B} + Q_{max} + L_{gate} \tag{10.2}$$

wobei:

- L_{fixed}: Feste Latenz, bestehend aus:
  - **Propagationslatenz**: Zeit für die Signalausbreitung im Medium (typisch ≈ 5 ns/m für Kupfer)
  - **Processing-Latenz**: Zeit für die Verarbeitung im Switch (typisch $\approx 10 - 50$ μs pro Hop)
  - **Serialisierungs-Latenz**: Zeit zum Senden eines Frames auf das Medium (abhängig von Frame-Größe und Bandbreite)
- S_i: Frame-Größe i in Bytes
- B: Bandbreite des Links in Bytes/s
- Q_{max}: Maximale Queueing-Latenz - die Zeit, die ein Frame in der Warteschlange verbringt, bevor er gesendet wird
- L_{gate}: Gate-Latenz - zusätzliche Latenz durch TSN Gate-Schedules, die bestimmen, wann ein Frame gesendet werden darf

Für TSN-Netze mit Gate-Scheduling wird die Gate-Latenz durch den Gate-Schedule bestimmt. Ein Gate-Schedule definiert Zeitfenster (Time Slots), in denen bestimmte Prioritätsklassen gesendet werden dürfen. Die maximale Gate-Latenz ist die Zeit, die ein Frame warten muss, bis sein zugewiesenes Zeitfenster beginnt.

Tabelle 10.2.: Beispiel: Latenzberechnung für einen TSN-Frame

Komponente	Wert	Beschreibung
Frame-Größe	1500 Byte	Ethernet-Frame
Bandbreite	1 Gbps	Link-Bandbreite
Hops	3	Anzahl der Switches
Processing pro Hop	20 μs	Switch-Verarbeitungszeit
Gate-Schedule-Zyklus	1 ms	TSN-Zykluszeit
Serialisierungs-Latenz	12 μs	$\frac{1500 \cdot 8}{10^9}$
Processing-Latenz	60 μs	$3 \cdot 20$ μs
Gate-Latenz (max)	1 ms	Worst-Case Wartezeit
Queueing-Latenz (max)	50 μs	Abhängig von Last
Gesamt-Latenz (max)	**1.122 ms**	Summe aller Komponenten

Die simulierten Netzwerk-Latenzen werden gegen diese analytischen Berechnungen validiert. Abweichungen können auf Fehler in der TSN-Konfiguration, ungenaue Modellierung der Switch-Verarbeitung oder Probleme mit dem Gate-Scheduling hinweisen.

10.1.2. Vergleich Simulation vs. Analytik

Der Vergleich zwischen Simulations- und analytischen Ergebnissen erfolgt systematisch für alle relevanten Metriken. Dieser Vergleich dient nicht nur der Fehlererkennung, sondern auch der Quantifizierung der Simulationsgenauigkeit und der Identifikation von Bereichen, in denen die Simulation möglicherweise ungenau ist.

Validierungs-Metriken

Die Validierung verwendet mehrere Metriken, um die Übereinstimmung zwischen Simulation und Analytik zu quantifizieren:

- **Abweichung (Deviation)**: Die prozentuale Abweichung zwischen simulierten und analytischen Werten:

$$\text{Deviation} = \frac{|V_{sim} - V_{analytical}|}{V_{analytical}} \cdot 100\% \tag{10.3}$$

  wobei V_{sim} der simulierte Wert und $V_{analytical}$ der analytische Wert ist.

- **Korrelation (Correlation)**: Der Korrelationskoeffizient r misst die lineare Beziehung zwischen simulierten und analytischen Werten:

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2 \sum_{i=1}^{n}(y_i - \bar{y})^2}} \tag{10.4}$$

  wobei x_i die analytischen Werte, y_i die simulierten Werte und $\bar{x}$, $\bar{y}$ die Mittelwerte sind. r liegt zwischen -1 und +1, wobei +1 eine perfekte positive Korrelation bedeutet.

- **Bestimmtheitsmaß (R^2)**: Das Bestimmtheitsmaß gibt an, wie gut die simulierten Werte durch die analytischen Werte erklärt werden:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{10.5}$$

  wobei $\hat{y}_i$ die durch die Regressionsgerade vorhergesagten Werte sind. R^2 liegt zwischen 0 und 1, wobei 1 eine perfekte Übereinstimmung bedeutet.

- **Outlier-Analyse**: Identifikation von Ausreißern - Werten, die signifikant von der erwarteten Beziehung abweichen. Outlier können auf Fehler in der Simulation, ungenaue analytische Modelle oder spezielle Randbedingungen hinweisen.
- **Mean Absolute Error (MAE)**: Der durchschnittliche absolute Fehler:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - x_i| \tag{10.6}$$

- **Root Mean Square Error (RMSE)**: Die Wurzel des mittleren quadratischen Fehlers, die größere Abweichungen stärker gewichtet:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - x_i)^2} \tag{10.7}$$

Akzeptanzkriterien

Die Akzeptanzkriterien definieren Schwellenwerte, die erfüllt sein müssen, damit die Simulation als valide betrachtet wird. Diese Kriterien sind abhängig von der Art der Metrik und der kritischen Bedeutung für die Architektur:

Tabelle 10.3.: Akzeptanzkriterien für Validierungs-Metriken

Metrik-Kategorie	Kriterium	Begründung
Timing (kritische Pfade)	Abweichung $< 10\%$	Sicherheitskritische Funktionen erfordern hohe Ge
Timing (normale Pfade)	Abweichung $< 20\%$	Weniger kritische Funktionen tolerieren größere Ab
Last (CPU/Netzwerk)	Abweichung $< 5\%$	Lastberechnungen müssen präzise sein für Dimensi
Korrelation	$R^2 > 0.9$	Starke lineare Beziehung zwischen Simulation und
Outlier-Rate	$< 5\%$	Nur wenige Ausreißer erlaubt

- **Timing**: Abweichung $< 10\%$ für kritische Pfade (z. B. Lenkung, Bremse), $< 20\%$ für normale Pfade
- **Last**: Abweichung $< 5\%$ für CPU/Netzwerk-Auslastung, da diese Metriken für die Dimensionierung kritisch sind
- **Korrelation**: $R^2 > 0.9$ für lineare Zusammenhänge, um sicherzustellen, dass die Simulation die analytischen Trends korrekt wiedergibt
- **Outlier**: $< 5\%$ der Messpunkte dürfen Ausreißer sein, und diese müssen dokumentiert und erklärt werden

Wenn die Akzeptanzkriterien nicht erfüllt werden, muss die Ursache identifiziert und behoben werden, bevor die Simulation für Architekturentscheidungen verwendet werden kann.

10.2. Sensitivitätsanalysen

Sensitivitätsanalysen untersuchen, wie empfindlich die Simulationsergebnisse auf Änderungen der Eingabeparameter reagieren. Diese Analysen sind wichtig, um zu verstehen, welche Parameter kritisch sind und welche Unsicherheiten in den Eingabedaten zu welchen Unsicherheiten in den Ergebnissen führen. Sensitivitätsanalysen helfen auch dabei, Robustheit zu bewerten und Prioritäten für die Genauigkeit der Eingabedaten zu setzen.

10.2.1. Parameter-Sensitivität

Die Parameter-Sensitivität wird für alle relevanten Eingabeparameter untersucht. Dabei werden die Parameter systematisch variiert und die Auswirkungen auf die Ausgabemetriken gemessen.

WCET-Variation

Die Worst-Case Execution Time (WCET) ist einer der kritischsten Parameter für Timing-Analysen. WCET-Werte sind oft unsicher, da sie von vielen Faktoren abhängen (Compiler-Optimierungen, Cache-Verhalten, Pipeline-Stalls, etc.). Die Sensitivität auf WCET-Variationen wird untersucht:

- **Variation**: ±20% WCET für alle Tasks (realistischer Unsicherheitsbereich)
- **Auswirkung**: Änderung der E2E-Latenz, CPU-Auslastung, Response Times, Deadline-Misses
- **Ziel**: Identifikation kritischer Tasks, deren WCET besonders genau bekannt sein muss
- **Methodik**: Monte-Carlo-Simulation mit zufälligen WCET-Variationen oder systematische Sweep-Analyse

Tasks mit hoher Sensitivität erfordern besonders genaue WCET-Messungen oder -Analysen, da Unsicherheiten in diesen Werten direkt zu Unsicherheiten in den E2E-Latenzen führen.

Tabelle 10.4.: Beispiel: Sensitivität der E2E-Latenz auf WCET-Variationen

Task	WCET-Basis (ms)	Δ E2E bei +20% (ms)	Sensitivität
Object_Detection	15.0	+3.2	Hoch
Image_Capture	2.0	+0.4	Mittel
Steering_Control	1.0	+0.1	Niedrig

Bandbreiten-Variation

Die Bandbreite von Netzwerk-Links kann durch verschiedene Faktoren beeinflusst werden (Kabelqualität, Interferenzen, Switch-Konfiguration). Die Sensitivität auf Bandbreiten-Variationen wird untersucht:

- **Variation**: ±10% Bandbreite für alle Links (realistischer Toleranzbereich)
- **Auswirkung**: Änderung der Netzwerk-Latenz, Paketverluste, Busauslastung, E2E-Latenz
- **Ziel**: Identifikation kritischer Links, die besonders zuverlässig dimensioniert werden müssen
- **Methodik**: Systematische Variation der Bandbreite für jeden Link einzeln und in Kombination

Links mit hoher Sensitivität sollten mit ausreichendem Sicherheitsabstand dimensioniert werden, um Schwankungen zu tolerieren.

Periodizitäts-Variation

Die Periodizität von Tasks und Frames kann sich durch Änderungen in den Anforderungen oder durch Jitter in der Triggerung ändern. Die Sensitivität auf Periodizitäts-Variationen wird untersucht:

- **Variation**: ±15% Periodizität für alle Frames/Tasks (realistischer Jitter-Bereich)
- **Auswirkung**: Änderung der Last, Timing, CPU/Netzwerk-Auslastung, E2E-Latenz
- **Ziel**: Robustheits-Bewertung - wie robust ist die Architektur gegen Timing-Variationen?
- **Methodik**: Systematische Variation der Periodizität und Messung der Auswirkungen

Eine Architektur, die sehr empfindlich auf Periodizitäts-Variationen reagiert, ist weniger robust und erfordert eine präzise Zeitsteuerung.

Prioritäts-Variation

Die Prioritäten von Tasks und Frames können sich durch Änderungen in den Anforderungen ändern. Die Sensitivität auf Prioritäts-Variationen wird untersucht:

- **Variation**: Änderung der Priorität um ±1 Stufe für alle Tasks/Frames
- **Auswirkung**: Änderung der Response Times, E2E-Latenzen, Deadline-Misses
- **Ziel**: Identifikation von Tasks/Frames, deren Priorität kritisch ist

10.2.2. Sensitivitäts-Koeffizienten

Sensitivitäts-Koeffizienten quantifizieren die Abhängigkeit zwischen Eingabe- und Ausgabeparametern. Der Sensitivitäts-Koeffizient $S_{x,y}$ gibt an, um wie viel Prozent sich der Ausgabeparameter y ändert, wenn sich der Eingabeparameter x um 1% ändert:

$$S_{x,y} = \frac{\partial y/y}{\partial x/x} = \frac{x}{y}\frac{\partial y}{\partial x} \tag{10.8}$$

wobei:

- $S_{x,y}$: Sensitivitäts-Koeffizient von y bezüglich x (dimensionslos)
- x: Eingabeparameter (z. B. WCET, Bandbreite, Periodizität)
- y: Ausgabeparameter (z. B. E2E-Latenz, CPU-Auslastung, Response Time)

Interpretation der Sensitivitäts-Koeffizienten:

- $|S_{x,y}| < 0.5$: Niedrige Sensitivität - Änderungen in x haben geringe Auswirkungen auf y
- $0.5 \leq |S_{x,y}| < 1.0$: Mittlere Sensitivität - Änderungen in x haben moderate Auswirkungen auf y
- $|S_{x,y}| \geq 1.0$: Hohe Sensitivität - Änderungen in x haben starke Auswirkungen auf y (nichtlineare Beziehung möglich)

Negative Sensitivitäts-Koeffizienten bedeuten, dass eine Erhöhung des Eingabeparameters zu einer Verringerung des Ausgabeparameters führt (z. B. höhere Bandbreite führt zu niedrigerer Latenz).

Tabelle 10.5.: Beispiel: Sensitivitäts-Koeffizienten für verschiedene Parameter

Eingabeparameter x	Ausgabeparameter y	$S_{x,y}$	Interpretation
WCET (Object_Detection)	E2E-Latenz (Lenkung)	0.85	Hohe Sensitivität
Bandbreite (Ethernet)	Netzwerk-Latenz	-0.92	Hohe Sensitivität (negativ)
Periodizität (Camera)	CPU-Auslastung	-0.45	Niedrige Sensitivität
Priorität (Task)	Response Time	-0.78	Hohe Sensitivität (negativ)

10.3. Rückkopplung ins Architekturmodell

Die Rückkopplung der Simulationsergebnisse ins Architekturmodell ist ein kritischer Schritt im iterativen Verbesserungsprozess. Die Simulationsergebnisse liefern wertvolle Erkenntnisse über die Performance der Architektur, die genutzt werden, um die Architektur zu optimieren. Dieser Prozess ermöglicht es, Probleme frühzeitig zu identifizieren und zu beheben, bevor sie zu kostspieligen Änderungen in späteren Entwicklungsphasen führen.

10.3.1. Bottleneck-Identifikation

Ein Bottleneck (Engpass) ist eine Komponente oder ein Pfad im System, der die Gesamtperformance limitiert. Die Identifikation von Bottlenecks ist wichtig, um Optimierungsmaßnahmen gezielt auf die kritischsten Bereiche zu fokussieren. Bottlenecks können in verschiedenen Bereichen auftreten: Timing, Ressourcen, Kommunikation oder Verfügbarkeit.

Timing-Bottlenecks

Timing-Bottlenecks sind Komponenten oder Pfade, die zu hohen Latenzen führen und damit die E2E-Latenz von Funktionsketten negativ beeinflussen. Simulationsergebnisse identifizieren Timing-Bottlenecks durch Analyse der Response Times, Latenzen und E2E-Latenzen:

- **Kritische Tasks**: Tasks mit hoher Response Time, die nahe an ihrer Deadline liegen oder diese überschreiten. Diese Tasks können die E2E-Latenz von Funktionsketten dominieren.
- **Kritische Links**: Netzwerk-Links mit hoher Latenz, die durch lange Übertragungszeiten, hohe Queueing-Latenzen oder Gate-Schedule-Verzögerungen verursacht werden.

- **Kritische Chains**: Funktionsketten mit hoher E2E-Latenz, die ihre Anforderungen nicht erfüllen. Diese Chains müssen analysiert werden, um die Ursache der hohen Latenz zu identifizieren.
- **Kritische Hops**: Switches oder Gateways, die hohe Processing-Latenzen verursachen.

Tabelle 10.6.: Beispiel: Identifizierte Timing-Bottlenecks

Komponente	Typ	Latenz (ms)	Budget (ms)	Status
Object_Detection Task	Task	18.5	15.0	Überschreitung
Ethernet Link (Camera)	Link	2.1	1.5	Kritisch
Lenkung Chain (E2E)	Chain	95.0	100.0	OK
TSN Switch (Front)	Hop	0.8	0.5	Kritisch

Die Identifikation von Timing-Bottlenecks ermöglicht es, gezielt Optimierungsmaßnahmen zu ergreifen, z. B. durch WCET-Optimierung, Prioritätsanpassung oder Bandbreiten-Erhöhung.

Ressourcen-Bottlenecks

Ressourcen-Bottlenecks sind Komponenten, die ihre Ressourcenkapazität nahezu vollständig ausnutzen und damit zu Performance-Problemen führen können. Hohe Ressourcenauslastung kann zu erhöhten Latenzen, Jitter und im Extremfall zu Deadline-Misses führen:

- **CPU-Bottlenecks**: ECUs mit hoher CPU-Auslastung (typisch $> 80\%$), die zu erhöhten Response Times führen. Diese ECUs können keine zusätzliche Last aufnehmen, ohne dass Performance-Probleme auftreten.
- **Netzwerk-Bottlenecks**: Links mit hoher Busauslastung (typisch $> 70\%$), die zu erhöhten Latenzen und möglicherweise zu Paketverlusten führen. Diese Links sind kritisch für die Kommunikation und können die E2E-Latenz dominieren.
- **Speicher-Bottlenecks**: ECUs mit hoher Speicher-Auslastung (typisch $> 85\%$), die zu Performance-Problemen durch häufige Speicher-Allokationen/Deallokationen oder durch Paging führen können.
- **GPU/NPU-Bottlenecks**: Rechenknoten mit hoher GPU/NPU-Auslastung, die für AI/ML-Workloads kritisch sind.

Ressourcen-Bottlenecks können durch Lastverteilung, Ressourcen-Erhöhung oder Optimierung der Workloads behoben werden.

Tabelle 10.7.: Beispiel: Identifizierte Ressourcen-Bottlenecks

Komponente	Ressource	Auslastung (%)	Schwellwert (%)	Status
AD-ECU	CPU	87.5	80	Bottleneck
Ethernet Link (Front)	Bandbreite	72.3	70	Kritisch
AD-ECU	RAM	78.2	85	OK
AD-ECU	GPU	91.2	80	Bottleneck

10.3.2. Optimierungs-Vorschläge

Basierend auf den identifizierten Bottlenecks werden gezielte Optimierungs-Vorschläge generiert. Diese Vorschläge werden priorisiert nach ihrer Wirksamkeit, ihrem Aufwand und ihrer Auswirkung auf andere Systemeigenschaften. Die Optimierungs-Vorschläge werden systematisch kategorisiert in Architektur-Anpassungen und Parameter-Tuning.

Architektur-Anpassungen

Architektur-Anpassungen sind strukturelle Änderungen an der Architektur, die typischerweise einen höheren Aufwand erfordern, aber auch größere Verbesserungen bewirken können:

- **Task-Migration**: Verschiebung von Tasks auf weniger ausgelastete ECUs, um die Last zu verteilen und CPU-Bottlenecks zu reduzieren. Dies erfordert eine Analyse der Kommunikationsabhängigkeiten, um sicherzustellen, dass die Migration nicht zu erhöhten Netzwerk-Latenzen führt.
- **Bandbreiten-Erhöhung**: Erhöhung der Bandbreite für kritische Links (z. B. von 1 Gbps auf 2.5 Gbps oder 10 Gbps), um Netzwerk-Bottlenecks zu beheben. Dies kann Hardware-Änderungen erfordern.
- **Prioritäts-Anpassung**: Änderung von Task/Frame-Prioritäten, um kritische Pfade zu bevorzugen und Timing-Bottlenecks zu reduzieren. Dies erfordert eine sorgfältige Analyse, um sicherzustellen, dass keine anderen Pfade negativ beeinflusst werden.
- **Redundanz-Hinzufügung**: Hinzufügung von Redundanz für kritische Komponenten, um Verfügbarkeit zu erhöhen und Single Points of Failure zu vermeiden. Dies erhöht die Kosten, aber verbessert die Robustheit.
- **Link-Redundanz**: Hinzufügung von redundanten Kommunikationspfaden (z. B. PRP - Parallel Redundancy Protocol) für sicherheitskritische Kommunikation.

- **ECU-Konsolidierung**: Zusammenlegung mehrerer ECUs zu einer leistungsfähigeren ECU, um Kosten zu reduzieren und die Kommunikation zu vereinfachen.
- **ECU-Aufteilung**: Aufteilung einer überlasteten ECU in mehrere ECUs, um die Last zu verteilen.

Tabelle 10.8.: Beispiel: Optimierungs-Vorschläge für identifizierte Bottlenecks

Bottleneck	Vorschlag	Aufwand	Erwartete Verbesserung
AD-ECU CPU (87.5%)	Task-Migration	Mittel	-15% CPU-Last
Ethernet Link (72.3%)	Bandbreite 1G → 2.5G	Hoch	-40% Latenz
Object_Detection (18.5 ms)	Priorität erhöhen	Niedrig	-3 ms Response Time
Lenkung Chain	Link-Redundanz	Hoch	+Verfügbarkeit

Parameter-Tuning

Parameter-Tuning sind Änderungen an Konfigurationsparametern, die typischerweise einen geringeren Aufwand erfordern, aber auch geringere Verbesserungen bewirken können:

- **WCET-Optimierung**: Reduzierung von WCETs durch Code-Optimierung, Compiler-Optimierungen oder Algorithmus-Verbesserungen. Dies erfordert Software-Entwicklung, kann aber erhebliche Verbesserungen bewirken.
- **Periodizitäts-Anpassung**: Optimierung von Frame/Task-Periodizitäten, um die Last zu reduzieren oder die Reaktionszeit zu verbessern. Längere Perioden reduzieren die Last, kürzere Perioden verbessern die Reaktionszeit.
- **Scheduling-Optimierung**: Änderung von Scheduling-Policies (z. B. von Fixed-Priority zu Earliest-Deadline-First) oder Anpassung von Scheduling-Parametern, um die CPU-Auslastung zu optimieren.
- **TSN-Konfiguration**: Optimierung von TSN Gate-Schedules, Prioritäten und Traffic Shaping-Parametern, um die Netzwerk-Latenz zu reduzieren.
- **Buffer-Größen**: Anpassung von Buffer-Größen in Switches und ECUs, um Paketverluste zu reduzieren, ohne die Latenz zu erhöhen.
- **Power-State-Management**: Optimierung von Power-State-Übergängen, um Energie zu sparen, ohne die Reaktionszeit zu beeinträchtigen.

Parameter-Tuning kann iterativ durchgeführt werden, wobei die Auswirkungen jeder Änderung gemessen werden, um die optimale Konfiguration zu finden.

10.3.3. Iterativer Verbesserungsprozess

Der iterative Verbesserungsprozess ist ein systematischer Ansatz zur kontinuierlichen Optimierung der Architektur basierend auf Simulationsergebnissen. Dieser Prozess ermöglicht es, die Architektur schrittweise zu verbessern, ohne dass alle Probleme gleichzeitig gelöst werden müssen. Der iterative Ansatz ist besonders wertvoll, da er es ermöglicht, die Auswirkungen von Änderungen zu messen und den Optimierungsprozess zu steuern.

Iterations-Zyklus

Der Iterations-Zyklus besteht aus mehreren Phasen, die wiederholt werden, bis die Ziele erreicht sind:

1. **Simulation**: Durchführung der Simulation mit der aktuellen Architektur-Konfiguration. Die Simulation wird mit repräsentativen Szenarien durchgeführt, die die relevanten Use-Cases und Lastbedingungen abdecken.
2. **Auswertung**: Analyse der Simulationsergebnisse mit Fokus auf die definierten KPIs (E2E-Latenz, CPU-Auslastung, Netzwerk-Auslastung, Deadline-Misses, etc.). Die Auswertung identifiziert Bereiche, die nicht die Anforderungen erfüllen.
3. **Identifikation**: Identifikation von Bottlenecks und Problemen durch systematische Analyse der Ergebnisse. Dies umfasst die Identifikation von Timing-Bottlenecks, Ressourcen-Bottlenecks und anderen Performance-Problemen.
4. **Anpassung**: Anpassung des Architekturmodells basierend auf den identifizierten Problemen. Dies kann Architektur-Anpassungen (z. B. Task-Migration, Bandbreiten-Erhöhung) oder Parameter-Tuning (z. B. Prioritäts-Anpassung, TSN-Konfiguration) umfassen.
5. **Validierung**: Validierung der Anpassungen durch erneute Simulation, um sicherzustellen, dass die Änderungen die erwarteten Verbesserungen bewirken und keine neuen Probleme verursachen.
6. **Wiederholung**: Wiederholung des Zyklus, bis die Ziele erreicht sind oder keine weiteren signifikanten Verbesserungen möglich sind.

Der Iterations-Zyklus sollte dokumentiert werden, um die Entscheidungen nachvollziehbar zu machen und um zu verhindern, dass bereits getestete Konfigurationen erneut getestet werden.

Tabelle 10.9.: Beispiel: Iterations-Zyklus für eine Lenkungs-Funktionskette

Iteration	Problem	Maßnahme	Ergebnis
1	E2E-Latenz 120 ms > 100 ms	Priorität erhöhen	105 ms (noch zu hoch)
2	Object_Detection 18.5 ms	WCET optimieren	15.2 ms, E2E 98 ms
3	CPU-Auslastung 87%	Task migrieren	CPU 75%, E2E 95 ms
4	–	–	Ziel erreicht

Konvergenz-Kriterien

Der iterative Prozess endet, wenn die Konvergenz-Kriterien erfüllt sind. Diese Kriterien definieren, wann die Architektur als ausreichend optimiert betrachtet wird:

- **Alle Deadlines eingehalten**: Keine Deadline-Misses mehr für kritische Tasks. Alle Funktionsketten erfüllen ihre E2E-Latenz-Anforderungen.
- **Auslastung akzeptabel**: CPU/Netzwerk-Auslastung $<$ Schwellenwerte (typisch CPU $< 80\%$, Netzwerk $< 70\%$), um ausreichend Reserve für zukünftige Erweiterungen zu haben.
- **Verbesserung minimal**: Weitere Iterationen bringen keine signifikante Verbesserung mehr (typisch $< 2\%$ Verbesserung pro Iteration). Dies deutet darauf hin, dass die Architektur nahe am Optimum ist.
- **Kosten-Nutzen-Verhältnis**: Weitere Optimierungen erfordern einen unverhältnismäßig hohen Aufwand im Vergleich zur erwarteten Verbesserung.
- **Alle Anforderungen erfüllt**: Alle funktionalen und nichtfunktionalen Anforderungen werden erfüllt.

Wenn die Konvergenz-Kriterien erfüllt sind, kann die Architektur als finalisiert betrachtet werden. Wenn die Kriterien nicht erfüllt werden können, müssen möglicherweise die Anforderungen überprüft oder alternative Architektur-Ansätze in Betracht gezogen werden.

10.4. Validierungs-Benchmarks

Validierungs-Benchmarks sind standardisierte Test-Cases, die verwendet werden, um die Korrektheit und Genauigkeit der Simulation zu überprüfen. Diese Benchmarks dienen als Referenzpunkte für die Validierung und ermöglichen es, die Performance der Simulation über verschiedene Architekturen und Konfigurationen hinweg zu bewerten.

10.4.1. Referenz-Architekturen

Referenz-Architekturen sind repräsentative Architekturen, die verschiedene Komplexitätsgrade und Anwendungsfälle abdecken. Diese Architekturen werden verwendet, um die Simulation zu validieren und um Performance-Trends zu identifizieren.

Standard-Test-Cases

Standard-Test-Cases werden für die Validierung verwendet und decken verschiedene Komplexitätsgrade ab:

- **Minimal-Architektur**: Kleinste mögliche Architektur mit minimaler Komplexität (z. B. Kamera → ECU → Aktor). Diese Architektur dient als Basis-Test, um sicherzustellen, dass die grundlegende Funktionalität korrekt ist.
- **Typische Architektur**: Repräsentative Architektur mit typischer Komplexität (z. B. mehrere Sensoren, mehrere ECUs, TSN-Netzwerk). Diese Architektur repräsentiert realistische Anwendungsfälle und dient als Haupt-Validierungs-Benchmark.
- **Maximale Architektur**: Größte mögliche Architektur mit maximaler Komplexität (z. B. viele Sensoren, viele ECUs, komplexes TSN-Netzwerk, Redundanz). Diese Architektur testet die Skalierbarkeit und die Grenzen der Simulation.
- **Spezielle Architekturen**: Architekturen mit speziellen Eigenschaften (z. B. hohe Redundanz, extreme Last, spezielle TSN-Konfigurationen), um spezifische Aspekte zu testen.

Tabelle 10.10.: Übersicht der Referenz-Architekturen

Architektur	Sensoren	ECUs	Links	Komplexität
Minimal	1	1	2	Niedrig
Typisch	8	5	12	Mittel
Maximal	20	10	30	Hoch
Redundanz	4	6	16	Mittel (hohe Redundanz)

Benchmark-Ergebnisse

Benchmark-Ergebnisse werden systematisch dokumentiert, um die Validierung zu unterstützen und um Performance-Trends zu identifizieren:

- **Baseline-Metriken**: Referenz-Werte für jede Benchmark-Architektur, die als Vergleichsbasis für zukünftige Simulationen dienen. Diese Werte werden mit analytischen Berechnungen verglichen, um die Genauigkeit zu validieren.
- **Performance-Trends**: Entwicklung der Performance-Metriken über verschiedene Architekturen hinweg, um zu verstehen, wie die Performance mit der Komplexität skaliert.
- **Reproduzierbarkeit**: Konsistenz der Ergebnisse über mehrere Simulationsläufe hinweg, um sicherzustellen, dass die Simulation deterministisch und reproduzierbar ist.
- **Validierungs-Status**: Dokumentation, ob die Benchmark die Validierungs-Kriterien erfüllt (z. B. Abweichung $< 10\%$ für Timing, $R^2 > 0.9$ für Korrelation).
- **Known Issues**: Dokumentation bekannter Probleme oder Einschränkungen für jede Benchmark.

Tabelle 10.11.: Beispiel: Benchmark-Ergebnisse für typische Architektur

Metrik	Analytisch	Simuliert	Abweichung (%)	Status
E2E-Latenz (Lenkung)	95.0 ms	97.2 ms	2.3	OK
CPU-Auslastung (AD-ECU)	75.0%	76.5%	2.0	OK
Netzwerk-Auslastung	65.0%	67.2%	3.4	OK
Response Time (Task)	12.0 ms	12.8 ms	6.7	OK

Die Benchmark-Ergebnisse werden regelmäßig aktualisiert, wenn Änderungen an der Simulation vorgenommen werden, um sicherzustellen, dass die Validierung aktuell bleibt.

10.5. Erweiterte Validierungs-Beispiele

Dieser Abschnitt präsentiert erweiterte Beispiele für die Validierung und Iteration von E/E-Architekturen.

10.5.1. Beispiel: Vollständige Validierungs-Pipeline

Dieses Beispiel zeigt eine vollständige Validierungs-Pipeline für eine komplexe Architektur.

Validierungs-Phasen

Die Validierung erfolgt in mehreren Phasen:

1. **Plausibilisierung**: Vergleich mit analytischen Modellen
   - WCRT-Berechnung für alle Tasks
   - TSN-Latenz-Berechnung für alle Frames
   - E2E-Latenz-Berechnung für alle Chains
   - Vergleich mit simulierten Werten
2. **Sensitivitäts-Analyse**: Analyse der Sensitivität gegenüber Parametern
   - Variation von WCETs: $\pm 20\%$
   - Variation von Periodizitäten: $\pm 10\%$
   - Variation von Bandbreiten: $\pm 25\%$
   - Analyse der Auswirkungen auf KPIs
3. **Bottleneck-Identifikation**: Systematische Identifikation von Bottlenecks
   - CPU-Bottlenecks: ECUs mit Auslastung $> 80\%$
   - Netzwerk-Bottlenecks: Links mit Auslastung $> 70\%$
   - Timing-Bottlenecks: Chains mit Latenz $>$ Deadline
   - Verfügbarkeits-Bottlenecks: Komponenten mit niedriger MTBF
4. **Optimierung**: Iterative Optimierung basierend auf identifizierten Bottlenecks
   - Architektur-Anpassungen
   - Parameter-Tuning
   - Validierung der Optimierungen
5. **Final-Validierung**: Finale Validierung der optimierten Architektur
   - Alle KPIs erfüllen Anforderungen
   - Keine kritischen Bottlenecks
   - Robustheit unter verschiedenen Szenarien

10.5.2. Erweiterte Validierungs-Methoden

Formale Verifikation

Formale Verifikation kann Simulation ergänzen:

- **Model Checking**: Automatische Verifikation von Eigenschaften
- **Theorem Proving**: Mathematische Beweise von Eigenschaften
- **Abstract Interpretation**: Statische Analyse von Programmen
- **Symbolic Execution**: Symbolische Ausführung für Test-Generierung

Machine-Learning-basierte Validierung

ML-Methoden können für Validierung verwendet werden:

- **Anomalie-Erkennung**: Automatische Erkennung von Anomalien in Ergebnissen
- **Prädiktion**: Vorhersage von Metriken basierend auf Parametern
- **Klassifikation**: Klassifikation von Architekturen nach Qualität
- **Regression**: Regression für Metrik-Prädiktion

Validierungs-Ergebnisse

Typische Validierungs-Ergebnisse für eine komplexe Architektur:

Tabelle 10.12.: Validierungs-Ergebnisse: Vor und Nach Optimierung

Metrik	Vor Optimierung	Nach Optimierung	Ziel
E2E-Latenz (Max)	112 ms	95 ms	< 100 ms
Deadline-Misses	2.3%	0%	0%
CPU-Last (Peak)	92%	78%	$< 80\%$
Netzwerk-Last (Peak)	78%	62%	$< 70\%$
Verfügbarkeit	99.5%	99.9%	$> 99.9\%$

10.5.3. Beispiel: Sensitivitäts-Analyse

Dieses Beispiel zeigt eine detaillierte Sensitivitäts-Analyse für kritische Parameter.

Tabelle 10.13.: WCET-Sensitivität: Auswirkung auf E2E-Latenz

Task	WCET-Änderung	E2E-Latenz-Änderung	Sensitivitäts-Koeffizient
Object_Detection	+20%	+12 ms	0.6
Trajectory_Planning	+20%	+8 ms	0.4
Steering_Control	+20%	+2 ms	0.1
Image_Preprocessing	+20%	+3 ms	0.15

WCET-Sensitivität

Die Sensitivität gegenüber WCET-Variationen:

Die Analyse zeigt, dass Object_Detection die höchste Sensitivität aufweist und daher besonders sorgfältig optimiert werden sollte.

Bandbreiten-Sensitivität

Die Sensitivität gegenüber Bandbreiten-Variationen:

Tabelle 10.14.: Bandbreiten-Sensitivität: Auswirkung auf Netzwerk-Latenz

Link	Bandbreiten-Änderung	Latenz-Änderung	Sensitivitäts-Koeffizient
Camera → ZC	-25%	+8 ms	0.32
ZC → AD-DC	-25%	+12 ms	0.48
AD-DC → EPS	-25%	+1 ms	0.04

Die Analyse zeigt, dass der Link ZC → AD-DC die höchste Sensitivität aufweist und daher besonders kritisch ist.

10.6. Zusammenfassung

Dieses Kapitel hat die Validierung und Iteration als kritische Komponenten des Entwicklungsprozesses beschrieben. Die Validierung stellt sicher, dass die Simulationsergebnisse korrekt sind und als Grundlage für Architekturentscheidungen verwendet werden können. Die Iteration ermöglicht es, die Architektur kontinuierlich zu verbessern, basierend auf den Erkenntnissen aus der Simulation.

10.7. Erweiterte Validierungs-Methoden

Dieser Abschnitt beschreibt erweiterte Methoden zur Validierung von Simulationsergebnissen.

10.7.1. Formale Verifikation

Formale Verifikation ermöglicht mathematische Beweise für Eigenschaften:

- **Model Checking**: Automatische Verifikation von Temporal-Logic-Eigenschaften
- **Theorem Proving**: Mathematische Beweise für Eigenschaften
- **Abstract Interpretation**: Statische Analyse zur Verifikation
- **Symbolic Execution**: Symbolische Ausführung zur Fehlerfindung

10.7.2. Hybrid-Validierung

Hybrid-Validierung kombiniert Simulation und formale Verifikation:

- **Simulation für Exploration**: Simulation für Design-Space-Exploration
- **Formale Verifikation für Korrektheit**: Formale Verifikation für kritische Eigenschaften
- **Integration**: Nahtlose Integration beider Ansätze

10.7.3. Statistische Validierung

Statistische Methoden für Validierung:

- **Confidence Intervals**: Konfidenzintervalle für Metriken
- **Hypothesis Testing**: Hypothesentests für Vergleich
- **Bayesian Analysis**: Bayes'sche Analyse für Unsicherheit
- **Regression Analysis**: Regressionsanalyse für Trends

Die wichtigsten Aspekte dieses Kapitels sind:

- **Plausibilisierung**: Vergleich der Simulationsergebnisse gegen analytische Modelle (WCRT-Berechnung, TSN-Latenz-Analyse) mit definierten Akzeptanzkriterien (Abweichung $< 10\%$ für kritische Pfade, $R^2 > 0.9$ für Korrelation). Die Plausibilisierung identifiziert systematische Fehler und quantifiziert die Simulationsgenauigkeit.

- **Sensitivitätsanalysen**: Untersuchung der Parameter-Abhängigkeit (WCET, Bandbreite, Periodizität, Priorität) mit Sensitivitäts-Koeffizienten zur Quantifizierung. Diese Analysen identifizieren kritische Parameter und bewerten die Robustheit der Architektur.
- **Rückkopplung**: Bottleneck-Identifikation (Timing-Bottlenecks, Ressourcen-Bottlenecks) und gezielte Optimierungs-Vorschläge (Architektur-Anpassungen, Parameter-Tuning). Die Rückkopplung ermöglicht es, Probleme frühzeitig zu identifizieren und zu beheben.
- **Iterativer Prozess**: Systematischer Verbesserungsprozess mit definiertem Iterations-Zyklus (Simulation → Auswertung → Identifikation → Anpassung → Validierung) und Konvergenz-Kriterien. Dieser Prozess ermöglicht eine schrittweise Optimierung der Architektur.
- **Validierungs-Benchmarks**: Standardisierte Test-Cases (Minimal-, Typische-, Maximale-Architektur) mit dokumentierten Benchmark-Ergebnissen für die kontinuierliche Validierung.

Die Validierung und Iteration bilden zusammen einen geschlossenen Regelkreis, der es ermöglicht, die Architektur kontinuierlich zu verbessern und sicherzustellen, dass sie die Anforderungen erfüllt. Dieser Ansatz ist besonders wertvoll in frühen Entwicklungsphasen, wo Änderungen noch kostengünstig durchgeführt werden können, und trägt dazu bei, dass die finale Architektur optimal dimensioniert und robust ist.

11. Dokumentation und Deliverables

Dieses Kapitel beschreibt die Dokumentation und die zu liefernden Artefakte (Deliverables) des Projekts. Diese umfassen Spezifikationen, Implementierungen, Beispiele und Evaluationsberichte. Eine umfassende und gut strukturierte Dokumentation ist essentiell für die Wiederverwendung, Wartung und Weiterentwicklung des Transformations-Frameworks. Die Deliverables ermöglichen es anderen Entwicklern und Forschern, das Framework zu verstehen, zu verwenden und zu erweitern.

11.1. Metamodell-Spezifikation

Die Metamodell-Spezifikation ist das zentrale Dokument, das die Struktur und Semantik des Intermediate Models (IM) definiert. Das IM dient als Abstraktionsebene zwischen PREEvision-Exporten und Simulationsmodellen und ermöglicht es, verschiedene Quell- und Zielformate zu unterstützen. Eine präzise und vollständige Spezifikation ist daher von entscheidender Bedeutung.

11.1.1. Intermediate Model (IM) Spezifikation

Das Intermediate Model ist ein strukturiertes Datenmodell, das alle relevanten Informationen einer E/E-Architektur in einer plattformunabhängigen Form repräsentiert. Die IM-Spezifikation definiert die exakte Struktur, Semantik und Constraints dieses Modells.

Struktur-Definition

Die IM-Spezifikation definiert die Struktur des Intermediate Models in mehreren Ebenen:

- **Schema-Definition**: Eine formale Schema-Definition in JSON Schema oder XML Schema, die die Syntax und Struktur des IM definiert. Das Schema dient sowohl als Dokumentation als auch als Validierungsgrundlage für IM-Instanzen.

Es definiert die erlaubten Elemente, ihre Attribute, Datentypen, Kardinalitäten und Constraints.

- **Klassen-Diagramm**: Ein UML-Klassen-Diagramm, das die Modell-Struktur visuell darstellt. Das Diagramm zeigt die Klassen (z. B. Node, Link, Task, Frame), ihre Attribute, Methoden und Beziehungen (Assoziationen, Vererbung, Komposition). Dies erleichtert das Verständnis der Modell-Struktur.
- **Attribut-Definitionen**: Eine detaillierte Beschreibung aller Attribute jeder Klasse, einschließlich:
  - Name und Datentyp
  - Semantik und Bedeutung
  - Erlaubte Werte und Constraints
  - Default-Werte (falls vorhanden)
  - Einheiten (für physikalische Größen)
  - Beispiele
- **Beziehungen**: Definition aller Beziehungen zwischen Elementen, einschließlich:
  - Assoziationen (z. B. Node hat Links)
  - Kompositionen (z. B. ECU enthält Tasks)
  - Vererbung (z. B. Sensor erbt von Node)
  - Kardinalitäten (1:1, 1:N, N:M)
  - Constraints und Invarianten
- **Metamodell-Hierarchie**: Die Hierarchie der Modell-Elemente, von abstrakten Basisklassen bis zu konkreten Instanzen. Dies zeigt, wie verschiedene Komponententypen (ECUs, Sensoren, Aktoren) in das gemeinsame Metamodell integriert sind.

Dokumentations-Format

Die Spezifikation wird in verschiedenen Formaten bereitgestellt, um unterschiedliche Anwendungsfälle zu unterstützen:

- **Markdown/LaTeX**: Lesbare Dokumentation für Menschen, die das IM verstehen und verwenden möchten. Diese Dokumentation enthält ausführliche Beschreibungen, Beispiele, Diagramme und Erklärungen. Sie dient als primäre Referenz für Entwickler.

Tabelle 11.1.: Beispiel: IM-Klassen-Hierarchie (Auszug)

Klasse	Basisklasse	Attribute (Auszug)	Beschreibung
Node	–	id, name, type	Basisklasse für alle Knoten
ECU	Node	cpu_cores, ram, asil	Rechenknoten
Sensor	Node	resolution, framerate	Sensor-Knoten
Actuator	Node	latency, accuracy	Aktor-Knoten
Link	–	bandwidth, latency, protocol	Kommunikationsverbindung
Task	–	wcet, period, priority	Software-Task

- **JSON Schema**: Maschinenlesbare Schema-Definition für die automatische Validierung von IM-Instanzen. Das JSON Schema kann von Validierungs-Tools verwendet werden, um sicherzustellen, dass IM-Instanzen korrekt strukturiert sind.
- **XML Schema**: Alternative Schema-Definition für XML-basierte IM-Repräsentationen. XML Schema bietet ähnliche Funktionalität wie JSON Schema, ist aber für XML optimiert.
- **API-Dokumentation**: Dokumentation der Programmierschnittstelle (API) für Software-Bibliotheken, die das IM manipulieren. Diese Dokumentation beschreibt Klassen, Methoden, Parameter und Rückgabewerte und ermöglicht es Entwicklern, das IM programmatisch zu verwenden.
- **GraphQL Schema**: Optionales GraphQL Schema für API-basierte Zugriffe auf IM-Daten, falls das IM über eine GraphQL-API verfügbar gemacht wird.

Die verschiedenen Formate ergänzen sich gegenseitig: Die Markdown/LaTeX-Dokumentation dient als primäre Referenz für Menschen, während die Schema-Definitionen für maschinelle Validierung und Verarbeitung verwendet werden.

11.2. Transformationsregeln

Die Transformationsregeln definieren, wie Elemente aus dem PREEvision-Modell in das Intermediate Model und schließlich in Simulationsmodelle transformiert werden. Eine vollständige und präzise Dokumentation dieser Regeln ist essentiell, um die Transformation nachvollziehbar und wartbar zu machen.

11.2.1. Mapping-Regel-Dokumentation

Die Mapping-Regel-Dokumentation beschreibt jede Transformationsregel im Detail, einschließlich der Quell- und Zielelemente, der Transformationslogik und der Bedingungen, unter denen die Regel angewendet wird.

Regel-Katalog

Ein umfassender Katalog aller Mapping-Regeln wird bereitgestellt, der als Referenz für Entwickler und als Grundlage für die automatische Code-Generierung dient. Jede Regel wird strukturiert dokumentiert:

- **Regel-ID**: Eindeutige Identifikation der Regel (z. B. `RULE_ECU_TO_NODE_001`)
- **Quell-Elemente**: Die PREEvision-Elemente, die transformiert werden (z. B. ECU-Knoten, Links, Tasks, Frames). Für jedes Quell-Element werden die relevanten Attribute und ihre Bedeutung beschrieben.
- **Ziel-Elemente**: Die Simulations-Elemente, die erzeugt werden (z. B. OMNeT++ Nodes, Channels, Applications). Für jedes Ziel-Element werden die zu setzenden Attribute und ihre Quellen beschrieben.
- **Mapping-Logik**: Detaillierte Beschreibung der Transformation, einschließlich:
  - Attribut-Mappings (wie werden Quell-Attribute auf Ziel-Attribute gemappt)
  - Berechnungen und Transformationen (z. B. Einheiten-Umrechnungen)
  - Default-Werte für fehlende Attribute
  - Bedingungen und Constraints
- **Beispiele**: Konkrete Beispiele für jede Regel, die zeigen, wie eine typische Transformation aussieht. Die Beispiele umfassen sowohl die Eingabe (PREEvision-Element) als auch die Ausgabe (Simulations-Element).
- **Abhängigkeiten**: Andere Regeln, die vor dieser Regel ausgeführt werden müssen, oder Regeln, die von dieser Regel abhängen.
- **Fehlerbehandlung**: Wie mit Fehlern oder fehlenden Daten umgegangen wird.

Tabelle 11.2.: Beispiel: Mapping-Regel-Katalog (Auszug)

Regel-ID	Quell-Element	Ziel-Element	Beschreibung
RULE_001	PREEvision ECU	OMNeT++ Node	Transformation eines ECU-Knotens
RULE_002	PREEvision Link	OMNeT++ Channel	Transformation einer Verbindung
RULE_003	PREEvision Task	OMNeT++ Application	Transformation einer Software-Task
RULE_004	PREEvision Frame	OMNeT++ Traffic	Transformation eines Datenframes

YAML-Konfiguration

Die Mapping-Regeln werden als strukturierte YAML-Dateien dokumentiert, die sowohl maschinenlesbar als auch menschenlesbar sind. YAML bietet eine gute Balance zwischen Lesbarkeit und Struktur:

- **Regel-Dateien**: Strukturierte YAML-Dateien, die alle Informationen zu einer Regel enthalten. Die Dateien sind hierarchisch organisiert und folgen einem konsistenten Schema.
- **Dokumentation**: Kommentare und Beschreibungen direkt in den YAML-Dateien, die die Regel erklären und Kontext liefern. YAML unterstützt Kommentare mit `#`, die für zusätzliche Erklärungen verwendet werden.
- **Validierung**: Schema-Validierung der Regel-Dateien, um sicherzustellen, dass sie korrekt strukturiert sind. Ein JSON Schema oder ein ähnliches Validierungsschema wird bereitgestellt, um die Konsistenz der Regel-Dateien zu gewährleisten.
- **Versionierung**: Versionskontrolle der Regel-Dateien, um Änderungen nachverfolgen zu können und um verschiedene Versionen der Regeln zu unterstützen.

```yaml
# Beispiel: YAML-Regel-Definition
rule_id: RULE_ECU_TO_NODE_001
name: "Transform ECU to OMNeT++ Node"
description: "Transforms a PREEvision ECU node to an OMNeT++ network node"

source:
  type: "ECU"
  attributes:
    - name: "cpu_cores"
      required: true
    - name: "cpu_freq"
      required: true
    - name: "ram_size"
      required: false
      default: 4096

target:
  type: "OMNeT++ Node"
  attributes:
```

```
    - name: "numCpus"
      mapping: "cpu_cores"
    - name: "cpuClockFrequency"
      mapping: "cpu_freq * 1e9"  # Convert GHz to Hz
    - name: "memorySize"
      mapping: "ram_size * 1024 * 1024"  # Convert GB to bytes

conditions:
  - "cpu_cores > 0"
  - "cpu_freq > 0"

examples:
  - input:
      cpu_cores: 4
      cpu_freq: 2.0
      ram_size: 8
    output:
      numCpus: 4
      cpuClockFrequency: 2000000000
      memorySize: 8589934592
```

Die YAML-Konfiguration ermöglicht es, die Transformationsregeln deklarativ zu definieren, was die Wartung und Erweiterung erleichtert.

11.3. Generator-Implementierung

Die Generator-Implementierung ist der ausführbare Code, der die Transformation von PREEvision-Modellen in Simulationsmodelle durchführt. Eine vollständige und gut dokumentierte Implementierung ist essentiell für die Wartbarkeit und Erweiterbarkeit des Frameworks.

11.3.1. Code-Generator

Der Code-Generator ist die Kernkomponente, die die Transformationsregeln anwendet und Simulationscode generiert. Die Implementierung sollte modular, erweiterbar und gut getestet sein.

Quellcode

Die vollständige Implementierung des Generators umfasst:

- **Quellcode**: Eine vollständige Implementierung in einer geeigneten Programmiersprache (Python, Java oder C++). Python wird oft bevorzugt, da es eine gute Balance zwischen Produktivität und Performance bietet und eine umfangreiche Bibliotheksunterstützung für Datenverarbeitung und Code-Generierung bietet.
- **Architektur**: Eine modulare Architektur, die verschiedene Komponenten trennt:
  - Parser-Modul: Liest und parst PREEvision-Exporte
  - IM-Generator: Erzeugt das Intermediate Model
  - Transformations-Engine: Wendet Mapping-Regeln an
  - Code-Generator: Generiert Simulationscode aus Templates
  - Validator: Validiert die generierten Modelle
- **Versionierung**: Ein Git-Repository mit vollständiger Versionshistorie, Tags für Releases und strukturierten Commit-Messages. Das Repository sollte auch Issue-Tracking und Pull-Request-Workflows unterstützen.
- **Dokumentation**: Umfassende Code-Dokumentation, einschließlich:
  - Inline-Kommentare für komplexe Logik
  - Docstrings/DocComments für alle öffentlichen Funktionen und Klassen
  - API-Dokumentation (z. B. Sphinx für Python, Javadoc für Java)
  - Architektur-Dokumentation (Übersichtsdiagramme, Design-Entscheidungen)
  - README mit Installations- und Verwendungsanleitung
- **Tests**: Umfassende Test-Suite, einschließlich:
  - Unit-Tests für einzelne Funktionen und Klassen
  - Integration-Tests für End-to-End-Transformationen
  - Regression-Tests für bekannte Architekturen
  - Performance-Tests für große Modelle
  - Test-Coverage-Metriken (Ziel: $> 80\%$)
- **CI/CD**: Continuous Integration/Continuous Deployment Pipeline für automatische Tests und Releases.

Templates

Template-Dateien definieren die Struktur des generierten Simulationscodes. Templates verwenden typischerweise eine Template-Engine (z.B. Jinja2 für Python, Velocity für Java), die Platzhalter durch tatsächliche Werte ersetzt:

- **OMNeT++ Templates**: Templates für OMNeT++ Netzwerk-Definitionen (.ned-Dateien) und Simulations-Konfigurationen (.ini-Dateien). Die Templates definieren die Struktur von Nodes, Channels, Applications und Parametern.
- **Simulink Templates**: Templates für Simulink-Modelle, die die Struktur von Blocks, Connections und Parametern definieren. Diese können als MATLAB-Skripte oder als XML-basierte Modelldefinitionen vorliegen.
- **NS-3 Templates**: C++ Code Templates für NS-3 Simulationen, die die Implementierung von Nodes, Channels und Applications definieren.
- **Modelica Templates**: Templates für Modelica-Modelle, die die Struktur von Komponenten, Verbindungen und Parametern definieren.
- **Template-Variablen**: Definiert, welche Variablen in den Templates verfügbar sind und wie sie aus dem IM extrahiert werden.
- **Template-Validierung**: Validierung der Templates, um sicherzustellen, dass sie syntaktisch korrekt sind und alle erforderlichen Variablen verwenden.

Tabelle 11.3.: Übersicht der Template-Dateien

Zielplattform	Template-Dateien	Format	Beschreibung
OMNeT++	network.ned.j2	Jinja2	Netzwerk-Definition
OMNeT++	config.ini.j2	Jinja2	Simulations-Konfiguration
Simulink	model.m.j2	Jinja2	MATLAB-Skript
NS-3	node.cc.j2	Jinja2	C++ Node-Implementierung
Modelica	model.mo.j2	Jinja2	Modelica-Modell

Die Templates sollten gut dokumentiert sein, mit Kommentaren, die erklären, welche Variablen verfügbar sind und wie sie verwendet werden.

11.4. Beispiel-Architekturen

11.4.1. Referenz-Architekturen

Minimal-Architektur

Eine minimale Beispiel-Architektur:

- **Komponenten**: Kamera, ECU, Aktor
- **Kommunikation**: Ethernet, CAN
- **Software**: Einfache Perzeptions-Pipeline
- **Dokumentation**: Vollständige Beschreibung

Typische Architektur

Eine repräsentative Architektur:

- **Komponenten**: Mehrere Sensoren, ECUs, Aktoren
- **Kommunikation**: TSN-Ethernet, CAN, LIN
- **Software**: Komplexe Funktionsketten
- **Dokumentation**: Detaillierte Beschreibung

11.4.2. Beispiel-Simulationen

Konfigurierte Simulationen

Vorkonfigurierte Simulations-Setups:

- **Nominal-Szenarien**: Standard-Betrieb
- **Stress-Szenarien**: Hohe Last
- **Fehler-Szenarien**: Ausfälle
- **Varianten**: Verschiedene Design-Varianten

Ergebnisse

Beispiel-Ergebnisse für Referenz-Architekturen:

- **Simulations-Outputs**: Ergebnis-Dateien
- **Auswertungen**: Analysierte Ergebnisse
- **Visualisierungen**: Dashboards, Diagramme

11.5. Szenarienkatalog

11.5.1. Szenario-Definitionen

Strukturierte Beschreibung

Jedes Szenario wird strukturiert beschrieben:

- **Szenario-ID**: Eindeutige Identifikation
- **Beschreibung**: Detaillierte Beschreibung
- **Parameter**: Konfigurations-Parameter
- **Erwartete Ergebnisse**: Erwartete Metriken

Kategorisierung

Szenarien werden kategorisiert:

- **Nominal**: Standard-Betrieb
- **Stress**: Hohe Last
- **Fehler**: Ausfälle
- **Varianten**: Design-Varianten

11.6. KPI-Definitionen

11.6.1. Messgrößen-Definition

Metriken-Katalog

Ein umfassender Katalog aller Messgrößen:

- **Metrik-ID**: Eindeutige Identifikation
- **Beschreibung**: Detaillierte Beschreibung
- **Formel**: Mathematische Definition
- **Einheit**: Physikalische Einheit
- **Akzeptanzkriterien**: Grenzwerte

Kategorisierung

Metriken werden kategorisiert:

- **Timing**: Latenz, Jitter, Deadline-Misses
- **Kommunikation**: Busauslastung, Paketverluste
- **Ressourcen**: CPU/GPU-Last, Speicher
- **Verfügbarkeit**: MTBF, Verfügbarkeit
- **Energie**: Energieverbrauch, Power-States

11.7. Evaluationsbericht

11.7.1. Methodik

Vorgehensweise

Beschreibung der Evaluations-Methodik:

- **Test-Architekturen**: Verwendete Architekturen
- **Szenarien**: Durchgeführte Szenarien
- **Messgrößen**: Verwendete Metriken
- **Validierung**: Validierungs-Ansätze

11.7.2. Ergebnisse

Performance-Analyse

Analyse der Simulations-Performance:

- **Simulations-Zeit**: Dauer der Simulationen
- **Skalierbarkeit**: Performance bei verschiedenen Architektur-Größen
- **Genauigkeit**: Vergleich mit analytischen Modellen

Architektur-Bewertung

Bewertung der Architekturen:

- **Timing-Analyse**: E2E-Latenzen, Jitter
- **Ressourcen-Analyse**: CPU/Netzwerk-Auslastung
- **Verfügbarkeits-Analyse**: MTBF, Verfügbarkeit
- **Energie-Analyse**: Energieverbrauch

11.7.3. Fazit und Ausblick

Zusammenfassung

Zusammenfassung der wichtigsten Erkenntnisse:

- **Erfolge**: Erreichte Ziele
- **Herausforderungen**: Identifizierte Probleme
- **Lösungen**: Implementierte Lösungen

Ausblick

Zukünftige Entwicklungen:

- **Erweiterungen**: Geplante Erweiterungen
- **Verbesserungen**: Mögliche Verbesserungen
- **Anwendungen**: Weitere Anwendungsgebiete

11.8. Deliverables-Übersicht

11.8.1. Artefakte

11.9. Erweiterte Deliverables

Dieser Abschnitt beschreibt zusätzliche Deliverables, die über die grundlegenden Artefakte hinausgehen und den Wert des Projekts erhöhen.

Tabelle 11.4.: Übersicht der Deliverables

Deliverable	Format	Beschreibung
IM-Spezifikation	PDF/Markdown	Metamodell-Definition
Transformationsregeln	YAML/PDF	Mapping-Regel-Katalog
Generator	Python/Java	Code-Generator
Beispiel-Architekturen	PREEvision/IM	Referenz-Modelle
Beispiel-Simulationen	OMNeT++/Simulink	Simulations-Setups
Szenarienkatalog	PDF/Markdown	Szenario-Definitionen
KPI-Definitionen	PDF/Markdown	Metriken-Katalog
Evaluationsbericht	PDF	Ergebnisse und Analyse

11.9.1. Online-Dokumentation

Eine umfassende Online-Dokumentation ermöglicht es Benutzern, das Framework schnell zu verstehen und effektiv zu nutzen.

Struktur

Die Online-Dokumentation umfasst:

- **Getting Started Guide**: Schritt-für-Schritt-Anleitung für erste Schritte
- **Tutorials**: Detaillierte Tutorials für häufige Anwendungsfälle
- **API-Referenz**: Vollständige API-Dokumentation mit Beispielen
- **FAQ**: Häufig gestellte Fragen und Antworten
- **Best Practices**: Empfehlungen für effektive Nutzung
- **Troubleshooting**: Lösungen für häufige Probleme

Technologie

Die Online-Dokumentation wird mit modernen Tools erstellt:

- **Sphinx** (für Python): Automatische API-Dokumentation aus Docstrings
- **MkDocs** oder **Docusaurus**: Moderne, responsive Dokumentations-Frameworks
- **Read the Docs**: Hosting-Plattform für automatische Builds
- **Search-Funktionalität**: Volltext-Suche für schnelle Navigation

11.9.2. Video-Tutorials

Video-Tutorials ergänzen die schriftliche Dokumentation und ermöglichen visuelles Lernen.

Inhalte

Die Video-Tutorials umfassen:

- **Installation und Setup**: Schritt-für-Schritt-Installation
- **Erste Transformation**: Komplettes Beispiel von PREEvision zu Simulation
- **Erweiterte Features**: Komplexe Anwendungsfälle
- **Debugging**: Fehlersuche und -behebung
- **Best Practices**: Empfehlungen von Experten

Format

Die Videos werden in verschiedenen Formaten bereitgestellt:

- **Kurze Tutorials**: 5-10 Minuten für spezifische Themen
- **Lange Tutorials**: 30-60 Minuten für umfassende Themen
- **Interaktive Videos**: Mit Quizzen und Übungen
- **Transkripte**: Schriftliche Transkripte für Barrierefreiheit

11.9.3. Community-Ressourcen

Community-Ressourcen fördern die Zusammenarbeit und den Wissensaustausch.

Foreum

Ein Diskussionsforum ermöglicht:

- **Fragen und Antworten**: Community-basierte Unterstützung
- **Feature-Requests**: Vorschläge für neue Features
- **Bug-Reports**: Meldung von Fehlern
- **Erfahrungsaustausch**: Diskussion von Best Practices

Beispiel-Sammlung

Eine Sammlung von Beispielen hilft Benutzern, schnell loszulegen:

- **Minimal-Beispiele**: Einfache Beispiele für erste Schritte
- **Erweiterte Beispiele**: Komplexe Anwendungsfälle
- **Best-Practice-Beispiele**: Beispiele, die Best Practices demonstrieren
- **Fehlerbeispiele**: Beispiele für häufige Fehler und deren Lösung

11.10. Erweiterte Dokumentations-Formate

Dieser Abschnitt beschreibt erweiterte Dokumentations-Formate, die für verschiedene Zielgruppen verwendet werden können.

11.10.1. Interactive Documentation

Interaktive Dokumentation ermöglicht exploratives Lernen:

- **Jupyter Notebooks**: Interaktive Beispiele mit Code und Erklärungen
- **Interactive Dashboards**: Interaktive Visualisierungen von Ergebnissen
- **Web-basierte Tutorials**: Schritt-für-Schritt-Tutorials im Browser
- **Code-Playgrounds**: Online-Editoren für Experimente

11.10.2. API-Dokumentation

Umfassende API-Dokumentation für Entwickler:

- **OpenAPI/Swagger**: REST-API-Dokumentation
- **GraphQL Schema**: GraphQL-API-Dokumentation
- **Code-Beispiele**: Praktische Code-Beispiele für alle APIs
- **SDK-Dokumentation**: Dokumentation für Software Development Kits

11.11. Erweiterte Dokumentations-Beispiele

Dieser Abschnitt präsentiert detaillierte Beispiele für verschiedene Dokumentations-Artefakte.

11.11.1. Beispiel: API-Dokumentation

REST-API-Dokumentation

````
# Transformation API

## POST /api/v1/transform

Transformiert ein PREEvision-Modell in ein Simulationsmodell.

### Request Body

```json
{
  "source_format": "preevision",
  "target_format": "omnetpp",
  "architecture_id": "arch_001",
  "options": {
    "validate": true,
    "optimize": true
  }
}
```

### Response

```json
{
  "transformation_id": "trans_001",
  "status": "completed",
  "output_files": [
    "omnet_config.ini",
    "network.ned",
    "simulation.ini"
  ],
  "metrics": {
    "transformation_time": 120.5,
    "nodes_created": 150,
    "links_created": 500
````
````

```
  }
}
```
‘‘‘

11.11.2. Beispiel: User-Guide

Quick-Start-Guide

1. **Installation**:

   ```
   pip install ee-arch-transformer
   ```

2. **Export aus PREEvision**:
   - Öffne PREEvision-Projekt
   - Export → JSON
   - Wähle alle relevanten Elemente
   - Export durchführen

3. **Transformation**:

   ```
   eetransform --input architecture.json \
               --output simulation/ \
               --format omnetpp \
               --validate
   ```

4. **Simulation**:

   ```
   cd simulation/
   opp_run -c Nominal omnet_config.ini
   ```

11.12. Zusammenfassung

Dieses Kapitel hat die Dokumentation und Deliverables als essentielle Komponenten des Projekts beschrieben. Eine umfassende und gut strukturierte Dokumentation ist entscheidend für die Wiederverwendung, Wartung und Weiterentwicklung des Transformations-Frameworks. Die Deliverables ermöglichen es anderen Entwicklern und Forschern, das Framework zu verstehen, zu verwenden und zu erweitern.

Die wichtigsten Aspekte dieses Kapitels sind:

- **Metamodell-Spezifikation**: Vollständige Spezifikation des Intermediate Models (IM) mit Schema-Definitionen, Klassen-Diagrammen, Attribut-Definitionen und Beziehungen. Die Spezifikation wird in verschiedenen Formaten bereitgestellt (Markdown/LaTeX, JSON Schema, XML Schema, API-Dokumentation), um unterschiedliche Anwendungsfälle zu unterstützen.

- **Transformationsregeln**: Umfassender Katalog aller Mapping-Regeln mit detaillierter Dokumentation der Quell- und Zielelemente, Mapping-Logik, Beispielen und Abhängigkeiten. Die Regeln werden als strukturierte YAML-Dateien dokumentiert, die sowohl maschinenlesbar als auch menschenlesbar sind.

- **Generator-Implementierung**: Vollständige Implementierung des Code-Generators mit modularem Design, umfassender Dokumentation, Test-Suite und CI/CD-Pipeline. Die Implementierung umfasst auch Template-Dateien für verschiedene Zielplattformen (OMNeT++, Simulink, NS-3, Modelica).

- **Beispiel-Architekturen**: Referenz-Modelle in verschiedenen Komplexitätsgraden (Minimal-, Typische-, Maximale-Architektur) mit vollständiger Dokumentation und vorkonfigurierten Simulations-Setups für verschiedene Szenarien (Nominal, Stress, Fehler, Varianten).

- **Szenarienkatalog**: Strukturierte Definitionen aller verwendeten Szenarien mit eindeutigen IDs, detaillierten Beschreibungen, Konfigurations-Parametern und erwarteten Ergebnissen. Die Szenarien werden kategorisiert (Nominal, Stress, Fehler, Varianten) für einfache Navigation.

- **KPI-Definitionen**: Umfassender Katalog aller Messgrößen mit eindeutigen IDs, detaillierten Beschreibungen, mathematischen Formeln, Einheiten und Akzeptanzkriterien. Die Metriken werden kategorisiert (Timing, Kommunikation, Ressourcen, Verfügbarkeit, Energie) für strukturierte Auswertung.

- **Evaluationsbericht**: Vollständiger Bericht über die Evaluierung des Frameworks, einschließlich Methodik, Performance-Analyse, Architektur-Bewertung, Erkenntnissen und Ausblick auf zukünftige Entwicklungen.

Diese Deliverables bilden zusammen eine vollständige Dokumentation des Transformations-Frameworks, die es ermöglicht, das Framework zu verstehen, zu verwenden, zu warten und zu erweitern. Die strukturierte Dokumentation und die bereitgestellten Beispiele erleichtern den Einstieg für neue Benutzer und ermöglichen es, das Framework in verschiedenen Kontexten anzuwenden.

12. Grobe Zeitplanung

Dieses Kapitel beschreibt die grobe Zeitplanung für das Projekt. Die Planung umfasst 12–16 Wochen mit verschiedenen Phasen für Anforderungen, Modellierung, Metrik, Transformation, Evaluation, Validierung und Dokumentation. Eine realistische und detaillierte Zeitplanung ist essentiell für den Projekterfolg, da sie hilft, Ressourcen zu planen, Meilensteine zu setzen und Risiken frühzeitig zu identifizieren. Die Zeitplanung folgt einer inkrementellen End-to-End-Strategie, bei der in jeder Phase funktionsfähige Artefakte erstellt werden, um frühzeitig Feedback zu erhalten und Risiken zu minimieren.

12.1. Projektphasen

Das Projekt ist in sieben Hauptphasen unterteilt, die sequenziell und teilweise parallel abgearbeitet werden. Jede Phase hat klar definierte Ziele, Deliverables und Erfolgskriterien. Die Phasen bauen aufeinander auf, wobei einige Phasen parallel laufen können, um die Gesamtdauer zu optimieren.

12.1.1. Phase 1: Anforderungen und Konzeption (Woche 1–2)

Die erste Phase legt die Grundlage für das gesamte Projekt. In dieser Phase werden die Anforderungen analysiert, das Konzept entwickelt und die Zielplattform ausgewählt. Eine gründliche Anforderungsanalyse ist kritisch, da unklare oder fehlende Anforderungen zu Verzögerungen und Problemen in späteren Phasen führen können.

Ziele

Die Ziele dieser Phase umfassen:

- **Anforderungsanalyse**: Detaillierte Analyse der Anforderungen aus verschiedenen Quellen (Stakeholder-Interviews, Literatur, bestehende Systeme). Die Anforderungen werden kategorisiert in funktionale Anforderungen (Was soll das System tun?), nichtfunktionale Anforderungen (Wie gut soll es funktionieren?) und Randbedingungen (Welche Einschränkungen gibt es?).

- **Zielplattform-Auswahl**: Auswahl der Simulationsplattform(en) basierend auf Anforderungen, Verfügbarkeit, Kosten und Kompatibilität. Kriterien für die Auswahl umfassen: Unterstützung für E/E-Architekturen, TSN-Funktionalität, Performance, Lizenzkosten, Community-Support, Dokumentationsqualität.
- **Konzept-Entwicklung**: Entwicklung des Transformationskonzepts, das beschreibt, wie PREEvision-Modelle in Simulationsmodelle transformiert werden. Das Konzept umfasst die Architektur des Transformations-Frameworks, die Rolle des Intermediate Models, die Transformationsregeln und die Validierungsstrategie.
- **Metamodell-Design**: Erster Entwurf des Intermediate Models (IM), der die Struktur und Semantik des Abstraktionsmodells definiert. Das IM-Design berücksichtigt die Anforderungen aus PREEvision-Exporten und die Anforderungen der Zielplattformen.

Aktivitäten und Aufwand

Die Aktivitäten in dieser Phase umfassen:

Tabelle 12.1.: Phase 1: Detaillierte Aktivitäten und Aufwand

Aktivität	Aufwand (Personentage)	Beschreibung
Stakeholder-Interviews	2	Interviews mit Domänenexperten
Literatur-Recherche	3	Analyse bestehender Ansätze
Anforderungsdokument	2	Strukturierung und Dokumentation
Plattform-Evaluation	3	Vergleich verschiedener Plattformen
Konzept-Entwicklung	4	Design des Transformationskonzepts
IM-Erstentwurf	2	Erster Entwurf des Metamodells
Review und Anpassung	2	Review mit Stakeholdern
Gesamt	**18**	**ca. 2 Wochen (1 Person)**

Deliverables

Die Deliverables dieser Phase umfassen:

- **Anforderungsdokument**: Strukturiertes Dokument mit funktionalen und nichtfunktionalen Anforderungen, Randbedingungen, Stakeholder-Anforderungen und Use-Cases. Das Dokument dient als Referenz für alle nachfolgenden Phasen.
- **Konzept-Dokument**: Detailliertes Konzept-Dokument, das die Architektur des Transformations-Frameworks beschreibt, die Transformationsstrategie erläutert und Design-Entscheidungen dokumentiert. Das Dokument enthält auch Diagramme (z. B. Architektur-Diagramme, Datenfluss-Diagramme).

- **Erster IM-Entwurf**: Erster Entwurf des Intermediate Models mit Klassen-Diagramm, Attribut-Definitionen und ersten Beispielen. Dieser Entwurf wird in Phase 2 verfeinert und finalisiert.
- **Plattform-Evaluationsbericht**: Vergleich verschiedener Simulationsplattformen mit Empfehlung für die Auswahl.

Erfolgskriterien

Die Phase ist erfolgreich abgeschlossen, wenn:

- Alle Anforderungen dokumentiert und priorisiert sind
- Eine Zielplattform ausgewählt und begründet ist
- Ein konsistentes Konzept vorliegt, das von Stakeholdern akzeptiert wurde
- Ein erster IM-Entwurf existiert, der die Hauptanforderungen abdeckt

12.1.2. Phase 2: Modellierung und Metamodell (Woche 3–5)

Ziele

- **PREEvision-Analyse**: Detaillierte Analyse der PREEvision-Exporte
- **IM-Spezifikation**: Vollständige Spezifikation des Intermediate Models
- **Schema-Definition**: JSON/XML Schema für das IM
- **Beispiel-Modelle**: Erstellung von Beispiel-Architekturen in PREEvision

Deliverables

- IM-Spezifikation (final)
- Schema-Definition
- Beispiel-Architekturen

12.1.3. Phase 3: Synthese-Metrik (Woche 4–6)

Ziele

- **Metrik-Konzeption**: Entwicklung der Synthese-Metriken
- **Flow-Aggregation**: Implementierung der Chain-Identifikation

- **Lastabschätzung**: Implementierung der Lastberechnungen
- **Validierung**: Validierung gegen analytische Modelle

Deliverables

- Metrik-Spezifikation
- Implementierung der Metrik-Berechnung
- Validierungs-Ergebnisse

12.1.4. Phase 4: Transformation (Woche 6–9)

Ziele

- **Parser-Implementierung**: Implementierung der Export-Parser
- **IM-Generierung**: Implementierung der IM-Generierung
- **Mapping-Regeln**: Definition der Mapping-Regeln
- **Code-Generator**: Implementierung des Code-Generators

Deliverables

- Parser-Implementierung
- IM-Generator
- Mapping-Regel-Katalog
- Code-Generator (erste Version)

12.1.5. Phase 5: Evaluation und Validierung (Woche 9–12)

Ziele

- **Beispiel-Simulationen**: Durchführung von Beispiel-Simulationen
- **Ergebnis-Auswertung**: Auswertung der Simulations-Ergebnisse
- **Validierung**: Validierung gegen analytische Modelle
- **Sensitivitäts-Analyse**: Durchführung von Sensitivitäts-Analysen

Deliverables

- Simulations-Ergebnisse
- Auswertungs-Dashboards
- Validierungs-Bericht

12.1.6. Phase 6: Iteration und Optimierung (Woche 12–14)

Ziele

- **Bottleneck-Identifikation**: Identifikation von Problemen
- **Optimierung**: Optimierung der Architektur und Transformation
- **Verbesserungen**: Verbesserung der Generatoren und Metriken
- **Weitere Szenarien**: Durchführung zusätzlicher Szenarien

Deliverables

- Optimierte Architekturen
- Verbesserte Generatoren
- Zusätzliche Simulations-Ergebnisse

12.1.7. Phase 7: Dokumentation (Woche 14–16)

Ziele

- **Dokumentation**: Vollständige Dokumentation aller Artefakte
- **Evaluationsbericht**: Erstellung des Evaluationsberichts
- **Beispiele**: Erstellung von Beispielen und Tutorials
- **Abschluss**: Finalisierung aller Deliverables

Deliverables

- Vollständige Dokumentation
- Evaluationsbericht
- Beispiel-Sammlung
- Alle finalen Deliverables

12.2. Zeitplan-Visualisierung

12.2.1. Gantt-Diagramm

Tabelle 12.2.: Grobe Zeitplanung (12–16 Wochen)

Phase	Wochen	Dauer	Abhängigkeiten
Anforderungen	1–2	2 Wochen	–
Modellierung	3–5	3 Wochen	Anforderungen
Synthese-Metrik	4–6	3 Wochen	Modellierung (parallel)
Transformation	6–9	4 Wochen	Modellierung, Metrik
Evaluation	9–12	4 Wochen	Transformation
Iteration	12–14	3 Wochen	Evaluation
Dokumentation	14–16	3 Wochen	Alle Phasen

12.3. Meilensteine

Meilensteine sind wichtige Zwischenziele im Projekt, die den Fortschritt markieren und als Entscheidungspunkte für die Fortsetzung oder Anpassung des Projekts dienen. Jeder Meilenstein hat definierte Erfolgskriterien, die erfüllt sein müssen, bevor das Projekt zur nächsten Phase übergeht.

12.3.1. M1: Konzept (Ende Woche 2)

Der erste Meilenstein markiert den Abschluss der Konzeptionsphase und die Freigabe für die Implementierung. Die Erfolgskriterien umfassen:

- **Anforderungen definiert**: Alle funktionalen und nichtfunktionalen Anforderungen sind dokumentiert, priorisiert und von Stakeholdern akzeptiert. Unklare oder widersprüchliche Anforderungen wurden geklärt.
- **Konzept erstellt**: Ein vollständiges und konsistentes Konzept liegt vor, das die Architektur, die Transformationsstrategie und die Validierungsansätze beschreibt. Das Konzept wurde von Experten reviewt und für umsetzbar befunden.
- **Zielplattform ausgewählt**: Eine Simulationsplattform wurde ausgewählt und begründet. Die Plattform erfüllt die Anforderungen und ist verfügbar (Lizenzen, Support, etc.).
- **IM-Erstentwurf**: Ein erster Entwurf des Intermediate Models existiert und deckt die Hauptanforderungen ab.

12.3.2. M2: Metamodell (Ende Woche 5)

Dieser Meilenstein markiert die Finalisierung des Metamodells und den Beginn der Implementierung. Die Erfolgskriterien umfassen:

- **IM-Spezifikation final**: Die IM-Spezifikation ist vollständig und final. Alle Klassen, Attribute, Beziehungen und Constraints sind definiert. Die Spezifikation wurde reviewt und validiert.
- **Schema definiert**: Ein formales Schema (JSON Schema oder XML Schema) für das IM ist definiert und validiert. Das Schema kann für automatische Validierung verwendet werden.
- **Beispiel-Modelle erstellt**: Mindestens drei Beispiel-Architekturen in PREEvision wurden erstellt (Minimal-, Typische-, Maximale-Architektur) und erfolgreich in das IM transformiert.
- **Metrik-Konzept**: Das Konzept für die Synthese-Metrik ist definiert und dokumentiert.

12.3.3. M3: Transformation (Ende Woche 9)

Dieser Meilenstein markiert die Funktionsfähigkeit des Transformations-Frameworks. Die Erfolgskriterien umfassen:

- **Parser implementiert**: Der Parser für PREEvision-Exporte ist implementiert und kann alle relevanten Exportformate (REST API, CSV, XML) verarbeiten. Der Parser wurde mit Beispiel-Exporten getestet.
- **Code-Generator funktionsfähig**: Der Code-Generator kann Simulationscode für die Zielplattform generieren. Die generierten Modelle sind syntaktisch korrekt und können kompiliert/ausgeführt werden.
- **Erste Simulationen durchgeführt**: Mindestens eine Beispiel-Architektur wurde erfolgreich transformiert und simuliert. Die Simulation läuft ohne Fehler und produziert Ergebnisse.
- **Mapping-Regeln dokumentiert**: Alle Mapping-Regeln sind dokumentiert und in YAML-Dateien verfügbar.

12.3.4. M4: Evaluation (Ende Woche 12)

Dieser Meilenstein markiert den Abschluss der Evaluierungsphase. Die Erfolgskriterien umfassen:

- **Alle Beispiel-Szenarien durchgeführt**: Alle definierten Szenarien (Nominal, Stress, Fehler, Varianten) wurden für mindestens eine Referenz-Architektur durchgeführt.
- **Ergebnisse ausgewertet**: Alle Simulationsergebnisse wurden ausgewertet und in Dashboards visualisiert. KPIs wurden berechnet und dokumentiert.
- **Validierung abgeschlossen**: Die Simulationsergebnisse wurden gegen analytische Modelle validiert. Die Validierungskriterien (Abweichung $< 10\%$, $R^2 > 0.9$) wurden erfüllt oder Abweichungen wurden dokumentiert und erklärt.
- **Sensitivitäts-Analyse**: Sensitivitätsanalysen wurden für kritische Parameter durchgeführt und dokumentiert.

12.3.5. M5: Abschluss (Ende Woche 16)

Dieser Meilenstein markiert den erfolgreichen Abschluss des Projekts. Die Erfolgskriterien umfassen:

- **Alle Deliverables fertiggestellt**: Alle geplanten Deliverables (IM-Spezifikation, Transformationsregeln, Generator, Beispiel-Architekturen, Szenarienkatalog, KPI-Definitionen) sind vollständig und qualitätsgeprüft.
- **Dokumentation vollständig**: Die vollständige Dokumentation liegt vor, einschließlich Benutzerhandbuch, Entwicklerhandbuch, API-Dokumentation und Architektur-Dokumentation. Die Dokumentation wurde reviewt und ist konsistent.
- **Evaluationsbericht erstellt**: Ein umfassender Evaluationsbericht liegt vor, der die Methodik, Ergebnisse, Erkenntnisse und Ausblick beschreibt. Der Bericht wurde reviewt und freigegeben.
- **Code und Artefakte versioniert**: Alle Code-Artefakte, Dokumentation und Beispiele sind in einem Versionskontrollsystem gespeichert und getaggt. Ein Release-Paket wurde erstellt.

12.4. Ressourcen

Die Ressourcenplanung umfasst sowohl personelle als auch technische Ressourcen. Eine realistische Ressourcenplanung ist essentiell, um sicherzustellen, dass das Projekt erfolgreich abgeschlossen werden kann.

12.4.1. Personelle Ressourcen

Die personellen Ressourcen umfassen verschiedene Rollen mit unterschiedlichen Verantwortlichkeiten und Zeitanteilen:

Tabelle 12.3.: Personelle Ressourcen-Planung

Rolle	Anzahl	Zeitanteil	Hauptverantwortlichkeiten
Projektleiter	1	100%	Projektmanagement, Koordination, Reviews
Entwickler	1–2	100%	Implementierung, Tests, Dokumentation
Experte (PREEvision)	0.5	20%	Beratung zu PREEvision, Modellierung
Experte (Simulation)	0.5	20%	Beratung zu Simulation, Validierung

- **Projektleiter**: 1 Person, 100% über gesamte Dauer (12–16 Wochen). Verantwortlichkeiten umfassen: Projektplanung und -steuerung, Koordination zwischen Phasen, Stakeholder-Management, Qualitätssicherung, Risikomanagement, Review-Organisation.
- **Entwickler**: 1–2 Personen, 100% für Entwicklung. Verantwortlichkeiten umfassen: Implementierung des Parsers, IM-Generators, Code-Generators, Tests, Dokumentation. Bei 2 Entwicklern können Aufgaben parallelisiert werden (z. B. einer für Parser/IM, einer für Code-Generator).
- **Experte (PREEvision)**: 0.5 Person (ca. 20% Zeitanteil), für Beratung. Unterstützt bei: PREEvision-Modellierung, Export-Formaten, Best Practices, Problemlösung bei Exporten. Wird hauptsächlich in Phase 2 und 4 benötigt.
- **Experte (Simulation)**: 0.5 Person (ca. 20% Zeitanteil), für Beratung. Unterstützt bei: Simulationsplattform-Auswahl, Simulations-Setup, Validierung, Performance-Optimierung. Wird hauptsächlich in Phase 1, 4 und 5 benötigt.

12.4.2. Technische Ressourcen

Die technischen Ressourcen umfassen Software-Lizenzen, Entwicklungstools und Hardware:

Tabelle 12.4.: Technische Ressourcen-Übersicht

Ressource	Typ	Beschreibung
PREEvision-Lizenz	Software	Für Modellierung und Export (Phase 2, 4)
OMNeT++/Simulink	Software	Simulationsplattform (kostenlos/kostenpflichtig)
Entwicklungsumgebung	Software	IDE (z. B. PyCharm, IntelliJ), Git, CI/CD
Rechner-Ressourcen	Hardware	Für Simulationen (CPU, RAM, Storage)

- **PREEvision-Lizenz**: Für Modellierung und Export. Wird hauptsächlich in Phase 2 (Beispiel-Modelle) und Phase 4 (Testing der Transformation) benötigt. Die Lizenz muss für die gesamte Projektlaufzeit verfügbar sein.
- **Simulations-Tools**: OMNeT++ (kostenlos, Open Source), Simulink (kostenpflichtig, falls verwendet), oder andere Simulationsplattformen. Die Auswahl hängt von der Plattform-Evaluation in Phase 1 ab. Für OMNeT++ sind keine Lizenzkosten erforderlich, für Simulink können erhebliche Kosten anfallen.
- **Entwicklungsumgebung**:
  - IDE: PyCharm/IntelliJ (für Python/Java), Visual Studio Code
  - Versionskontrolle: Git mit Repository (z. B. GitHub, GitLab)
  - CI/CD: GitHub Actions, GitLab CI, oder Jenkins für automatische Tests
  - Dokumentation: Sphinx (Python), Javadoc (Java), Markdown-Tools
- **Rechner-Ressourcen**: Für Simulationen werden leistungsfähige Rechner benötigt:
  - CPU: Multi-Core-Prozessor (8+ Kerne empfohlen für parallele Simulationen)
  - RAM: Mindestens 16 GB, besser 32 GB für große Modelle
  - Storage: SSD mit ausreichend Platz für Simulations-Ergebnisse (100+ GB)
  - Optional: GPU für beschleunigte Simulationen (falls unterstützt)

12.5. Risikomanagement

Das Risikomanagement identifiziert potenzielle Risiken, bewertet ihre Auswirkungen und definiert Gegenmaßnahmen. Ein proaktives Risikomanagement hilft, Verzögerungen zu vermeiden und das Projekt erfolgreich abzuschließen.

12.5.1. Pufferzeiten

Pufferzeiten werden strategisch in kritischen Phasen eingebaut, um unvorhergesehene Probleme abzufedern:

Tabelle 12.5.: Pufferzeiten-Planung

Puffer	Phase	Dauer	Begründung
Technische Risiken	4–5	2 Wochen	Komplexe Implementierung, mögliche Probler
Datenverfügbarkeit	2	1 Woche	PREEvision-Exporte könnten unvollständig s
Unvorhergesehene Probleme	Gesamt	2 Wochen	Allgemeiner Puffer für alle Phasen
Gesamt-Puffer	–	**5 Wochen**	**ca. 31% der Gesamtdauer**

- **Technische Risiken**: 2 Wochen Puffer in Phase 4–5 (Transformation und Evaluation). Diese Phasen sind technisch anspruchsvoll und können unerwartete Probleme aufwerfen (z. B. Kompatibilitätsprobleme, Performance-Probleme, Bugs in der Transformation).
- **Datenverfügbarkeit**: 1 Woche Puffer in Phase 2 (Modellierung). PREEvision-Exporte könnten unvollständig sein oder fehlende Attribute enthalten, was zusätzliche Zeit für Datenbeschaffung oder Schätzungen erfordert.
- **Unvorhergesehene Probleme**: 2 Wochen Gesamt-Puffer, verteilt über das gesamte Projekt. Dieser Puffer deckt unvorhergesehene Probleme ab, die nicht spezifisch einer Phase zugeordnet werden können (z. B. Krankheit, unerwartete Anforderungsänderungen, externe Abhängigkeiten).

Die Gesamt-Projektdauer beträgt damit 12 Wochen (Basis) + 5 Wochen (Puffer) = 17 Wochen, wobei die Pufferzeiten flexibel eingesetzt werden können.

12.5.2. Alternativ-Pläne

Alternativ-Pläne definieren Strategien für den Fall, dass das Projekt in Verzug gerät oder Ressourcen knapp werden:

- **Reduzierter Scope**: Falls Zeit knapp wird, Fokus auf Kern-Funktionalität:
  - Priorisierung: Nur eine Zielplattform statt mehrerer
  - Vereinfachung: Reduzierte Anzahl von Beispiel-Architekturen
  - Minimal-Version: Fokus auf Minimal- und Typische-Architektur, Maximale-Architektur optional

  - Dokumentation: Kern-Dokumentation statt vollständiger Dokumentation

- **Parallelisierung**: Parallele Arbeit an verschiedenen Komponenten, um Zeit zu sparen:
  - Parser und Code-Generator können parallel entwickelt werden (nach IM-Spezifikation)
  - Dokumentation kann parallel zur Implementierung geschrieben werden
  - Beispiel-Architekturen können parallel zur Generator-Entwicklung erstellt werden
- **Externe Unterstützung**: Bei Bedarf externe Unterstützung einholen:
  - Externe Entwickler für spezifische Aufgaben (z. B. Template-Entwicklung)
  - Beratung von Experten bei komplexen Problemen
  - Nutzung von Open-Source-Komponenten, wo möglich
- **Phasen-Verschiebung**: Falls notwendig, können nicht-kritische Phasen verschoben werden:
  - Dokumentation kann teilweise in eine Nachprojektphase verschoben werden
  - Zusätzliche Szenarien können optional gemacht werden
  - Erweiterte Validierung kann reduziert werden

12.5.3. Risiko-Monitoring

Das Risiko-Monitoring erfolgt kontinuierlich während des Projekts:

- **Wöchentliche Reviews**: Wöchentliche Überprüfung des Projektfortschritts und Identifikation neuer Risiken
- **Meilenstein-Reviews**: Detaillierte Risiko-Bewertung an jedem Meilenstein
- **Risiko-Register**: Dokumentation aller identifizierten Risiken, ihrer Wahrscheinlichkeit, Auswirkung und Gegenmaßnahmen
- **Eskalations-Prozess**: Klarer Prozess für die Eskalation kritischer Risiken an Stakeholder

12.6. Erweiterte Planungs-Aspekte

Dieser Abschnitt beschreibt erweiterte Aspekte der Projektplanung, die für den Projekterfolg wichtig sind.

12.6.1. Agile Methoden

Agile Methoden ermöglichen eine flexible Anpassung des Projekts an sich ändernde Anforderungen.

Sprints

Das Projekt kann in Sprints unterteilt werden:

- **Sprint-Dauer**: 2 Wochen pro Sprint
- **Sprint-Planning**: Zu Beginn jedes Sprints werden Ziele und Aufgaben definiert
- **Daily Standups**: Tägliche kurze Meetings zur Synchronisation
- **Sprint-Review**: Am Ende jedes Sprints werden Ergebnisse präsentiert
- **Retrospective**: Reflexion über den Sprint und Verbesserungen

Backlog-Management

Ein Product Backlog enthält alle geplanten Features und Aufgaben:

- **Priorisierung**: Features werden nach Wichtigkeit priorisiert
- **Schätzung**: Aufwand wird geschätzt (z. B. Story Points)
- **Refinement**: Backlog wird regelmäßig verfeinert und aktualisiert
- **Sprint-Backlog**: Ausgewählte Items für den aktuellen Sprint

12.6.2. Qualitätssicherung

Qualitätssicherung ist ein kontinuierlicher Prozess während des gesamten Projekts.

Code-Reviews

Code-Reviews verbessern die Code-Qualität:

- **Peer Reviews**: Entwickler reviewen Code von Kollegen
- **Checkliste**: Strukturierte Checkliste für Reviews
- **Automatisierung**: Automatische Checks (Linting, Formatting)
- **Feedback-Kultur**: Konstruktives Feedback fördern

Testing-Strategie

Eine umfassende Testing-Strategie umfasst:

- **Unit-Tests**: Tests für einzelne Funktionen/Methoden
- **Integration-Tests**: Tests für Komponenten-Integration
- **System-Tests**: Tests für das gesamte System
- **Regression-Tests**: Tests zur Verhinderung von Regressionen
- **Performance-Tests**: Tests für Performance und Skalierbarkeit

12.6.3. Kommunikation und Koordination

Effektive Kommunikation ist entscheidend für den Projekterfolg.

Stakeholder-Kommunikation

Regelmäßige Kommunikation mit Stakeholdern:

- **Wöchentliche Updates**: Status-Updates per E-Mail oder Meeting
- **Monatliche Reviews**: Detaillierte Reviews mit Präsentationen
- **Ad-hoc-Kommunikation**: Bei kritischen Problemen
- **Feedback-Sammlung**: Regelmäßige Sammlung von Feedback

Team-Kommunikation

Effektive Team-Kommunikation:

- **Team-Meetings**: Regelmäßige Team-Meetings (z.B. wöchentlich)
- **Technische Diskussionen**: Detaillierte technische Diskussionen
- **Dokumentation**: Wichtige Entscheidungen werden dokumentiert
- **Kollaborations-Tools**: Nutzung von Tools wie Slack, Teams, etc.

12.7. Erweiterte Projektmanagement-Aspekte

Dieser Abschnitt beschreibt erweiterte Aspekte des Projektmanagements für ein solches Projekt.

12.7.1. Ressourcen-Management

Effektives Ressourcen-Management ist entscheidend für den Projekterfolg:

- **Personal-Ressourcen**: Planung von Personal-Ressourcen mit verschiedenen Fähigkeiten
- **Technische Ressourcen**: Planung von Hardware, Software, Tools
- **Externe Ressourcen**: Planung von externen Dienstleistern, Beratern
- **Ressourcen-Optimierung**: Optimierung der Ressourcen-Nutzung

12.7.2. Qualitätsmanagement

Qualitätsmanagement sichert die Qualität der Ergebnisse:

- **Qualitätsplanung**: Definition von Qualitätszielen und -kriterien
- **Qualitätssicherung**: Kontinuierliche Qualitätssicherung während der Entwicklung
- **Qualitätskontrolle**: Prüfung der Ergebnisse gegen Qualitätskriterien
- **Qualitätsverbesserung**: Kontinuierliche Verbesserung basierend auf Feedback

12.8. Zusammenfassung

Dieses Kapitel hat die grobe Zeitplanung für das Projekt beschrieben. Eine realistische und detaillierte Zeitplanung ist essentiell für den Projekterfolg, da sie hilft, Ressourcen zu planen, Meilensteine zu setzen und Risiken frühzeitig zu identifizieren.

Die wichtigsten Aspekte dieses Kapitels sind:

- **7 Phasen**: Das Projekt ist in sieben Hauptphasen unterteilt (Anforderungen, Modellierung, Synthese-Metrik, Transformation, Evaluation, Iteration, Dokumentation), die sequenziell und teilweise parallel abgearbeitet werden. Jede Phase hat klar definierte Ziele, Deliverables und Erfolgskriterien.
- **12–17 Wochen**: Die Gesamtdauer beträgt 12 Wochen (Basis) plus 5 Wochen Puffer, was eine realistische Planung ermöglicht. Die Pufferzeiten werden strategisch in kritischen Phasen eingebaut, um unvorhergesehene Probleme abzufedern.
- **5 Meilensteine**: Fünf wichtige Meilensteine markieren den Fortschritt und dienen als Entscheidungspunkte (Konzept, Metamodell, Transformation, Evaluation, Abschluss). Jeder Meilenstein hat definierte Erfolgskriterien.
- **Ressourcen**: Die Ressourcenplanung umfasst personelle Ressourcen (Projektleiter, Entwickler, Experten) und technische Ressourcen (Software-Lizenzen, Entwicklungstools, Hardware). Eine realistische Ressourcenplanung ist essentiell für den Projekterfolg.
- **Risikomanagement**: Das Risikomanagement umfasst Pufferzeiten (5 Wochen Gesamt-Puffer), Alternativ-Pläne (reduzierter Scope, Parallelisierung, externe Unterstützung) und kontinuierliches Risiko-Monitoring. Ein proaktives Risikomanagement hilft, Verzögerungen zu vermeiden.

Die Zeitplanung folgt einer inkrementellen End-to-End-Strategie, bei der in jeder Phase funktionsfähige Artefakte erstellt werden. Dieser Ansatz ermöglicht es, frühzeitig Feedback zu erhalten, Risiken zu minimieren und den Fortschritt kontinuierlich zu überwachen. Die flexible Planung mit Pufferzeiten und Alternativ-Plänen ermöglicht es, auf unvorhergesehene Probleme zu reagieren, ohne das Projektziel zu gefährden.

13. Risiken und Gegenmaßnahmen

Dieses Kapitel beschreibt die identifizierten Risiken des Projekts und die geplanten Gegenmaßnahmen. Die Risiken umfassen technische Herausforderungen, Datenverfügbarkeit, Komplexität und Reproduzierbarkeit.

13.1. Technische Risiken

13.1.1. Toolkopplung und Exportformate

Risiko

- **Beschreibung**: PREEvision-Exportformate könnten sich ändern oder unvollständig sein
- **Wahrscheinlichkeit**: Mittel
- **Auswirkung**: Hoch (kann Projekt verzögern)
- **Erkennung**: Frühe Tests der Export-Funktionalität

Gegenmaßnahmen

- **IM-Schicht**: Intermediate Model als Abstraktionsebene, unabhängig von Export-Format
- **Adapter-Pattern**: Flexible Parser für verschiedene Formate
- **Frühe Validierung**: Tests der Export-Funktionalität in Phase 2
- **Alternative Exporte**: Nutzung mehrerer Exportformate (REST, CSV, XML)
- **Kontakt zu PREEvision-Support**: Regelmäßiger Kontakt für Format-Fragen

13.1.2. Simulationsplattform-Kompatibilität

Risiko

- **Beschreibung**: Zielplattform könnte nicht alle erforderlichen Features unterstützen
- **Wahrscheinlichkeit**: Mittel
- **Auswirkung**: Mittel (kann Anpassungen erfordern)

Gegenmaßnahmen

- **Frühe Evaluation**: Evaluation der Plattform in Phase 1
- **Multi-Plattform-Ansatz**: Unterstützung mehrerer Plattformen
- **Feature-Mapping**: Detaillierte Analyse der Feature-Kompatibilität
- **Workarounds**: Entwicklung von Workarounds für fehlende Features

13.2. Datenverfügbarkeit

13.2.1. Fehlende oder unvollständige Daten

Risiko

- **Beschreibung**: WCET, Datenraten oder andere Parameter könnten fehlen
- **Wahrscheinlichkeit**: Hoch
- **Auswirkung**: Mittel (kann Genauigkeit beeinträchtigen)

Gegenmaßnahmen

- **Default-Werte**: Definition von realistischen Default-Werten
- **Schätzungen**: Entwicklung von Schätzungs-Methoden basierend auf ähnlichen Komponenten
- **Benchmarks**: Nutzung von Benchmark-Daten für typische Komponenten
- **Dokumentation**: Klare Dokumentation fehlender Daten
- **Sensitivitäts-Analyse**: Analyse der Auswirkung fehlender Daten

13.2.2. Datenqualität

Risiko

- **Beschreibung**: Exportierte Daten könnten inkonsistent oder fehlerhaft sein
- **Wahrscheinlichkeit**: Mittel
- **Auswirkung**: Mittel (kann zu falschen Ergebnissen führen)

Gegenmaßnahmen

- **Validierung**: Umfassende Validierung der Eingabedaten
- **Konsistenz-Checks**: Automatische Prüfung auf Konsistenz
- **Fehlerbehandlung**: Robuste Fehlerbehandlung und -meldungen
- **Manuelle Prüfung**: Manuelle Prüfung kritischer Daten

13.3. Komplexität

13.3.1. Architektur-Komplexität

Risiko

- **Beschreibung**: Reale Architekturen könnten zu komplex für die Transformation sein
- **Wahrscheinlichkeit**: Mittel
- **Auswirkung**: Hoch (kann Transformation unmöglich machen)

Gegenmaßnahmen

- **Inkrementelle Schritte**: Schrittweise Erhöhung der Komplexität
- **Abstraktion**: Abstraktion komplexer Aspekte im IM
- **Modularität**: Modulare Transformation für verschiedene Aspekte
- **Beispiel-Architekturen**: Beginn mit einfachen Beispielen
- **Skalierbarkeit**: Design für Skalierbarkeit von Anfang an

13.3.2. Metrik-Komplexität

Risiko

- **Beschreibung**: Synthese-Metriken könnten zu komplex oder ungenau sein
- **Wahrscheinlichkeit**: Mittel
- **Auswirkung**: Mittel (kann Genauigkeit beeinträchtigen)

Gegenmaßnahmen

- **Validierung**: Kontinuierliche Validierung gegen analytische Modelle
- **Benchmarks**: Vergleich mit bekannten Benchmarks
- **Iterative Verbesserung**: Kontinuierliche Verbesserung der Metriken
- **Dokumentation**: Detaillierte Dokumentation der Annahmen

13.4. Reproduzierbarkeit

13.4.1. Simulations-Reproduzierbarkeit

Risiko

- **Beschreibung**: Simulationen könnten nicht reproduzierbar sein
- **Wahrscheinlichkeit**: Niedrig
- **Auswirkung**: Mittel (kann Validierung erschweren)

Gegenmaßnahmen

- **Seed-Management**: Fixierung von Zufalls-Seeds
- **Deterministische Generierung**: Deterministische Code-Generierung
- **Versionierung**: Versionierung aller Konfigurationen
- **CI**: Automatisierte Tests in CI-Pipelines
- **Dokumentation**: Dokumentation der Umgebung und Konfiguration

13.4.2. Umgebungs-Abhängigkeit

Risiko

- **Beschreibung**: Ergebnisse könnten von der Umgebung abhängen
- **Wahrscheinlichkeit**: Niedrig
- **Auswirkung**: Niedrig

Gegenmaßnahmen

- **Containerisierung**: Nutzung von Containern für Reproduzierbarkeit
- **Dependency-Pinning**: Fixierung aller Abhängigkeiten
- **Environment-Dokumentation**: Vollständige Dokumentation der Umgebung

13.5. Zeitplan-Risiken

13.5.1. Verzögerungen

Risiko

- **Beschreibung**: Unvorhergesehene Probleme könnten zu Verzögerungen führen
- **Wahrscheinlichkeit**: Mittel
- **Auswirkung**: Mittel (kann Meilensteine gefährden)

Gegenmaßnahmen

- **Pufferzeiten**: Eingebaute Pufferzeiten im Zeitplan
- **Priorisierung**: Fokus auf kritische Funktionen
- **Agile Methoden**: Flexible Anpassung des Plans
- **Frühe Risiko-Erkennung**: Regelmäßige Risiko-Reviews

13.6. Erweiterte Risiko-Analyse

13.6.1. Quantitative Risiko-Bewertung

Neben der qualitativen Risiko-Bewertung können Risiken auch quantitativ bewertet werden, um eine fundiertere Entscheidungsgrundlage zu schaffen.

Risiko-Score

Der Risiko-Score wird berechnet als:

$$R = P \times I \tag{13.1}$$

wobei:

- P: Wahrscheinlichkeit des Eintretens (0-1)
- I: Auswirkung (1-5 Skala)
- R: Risiko-Score (0-5)

Risiken mit $R > 3$ werden als hoch eingestuft und erfordern sofortige Gegenmaßnahmen.

Beispiel: Risiko-Bewertung für Toolkopplung

- **Wahrscheinlichkeit**: $P = 0.6$ (Mittel - Exportformate können sich ändern)
- **Auswirkung**: $I = 4$ (Hoch - kann Projekt verzögern)
- **Risiko-Score**: $R = 0.6 \times 4 = 2.4$ (Mittel-Hoch)
- **Bewertung**: Erfordert proaktive Gegenmaßnahmen

13.6.2. Risiko-Monitoring

Risiken müssen kontinuierlich überwacht werden, um frühzeitig auf Änderungen reagieren zu können.

Risiko-Reviews

Regelmäßige Risiko-Reviews (z. B. wöchentlich) umfassen:

- **Status-Check**: Überprüfung des aktuellen Status jedes Risikos
- **Wahrscheinlichkeits-Update**: Aktualisierung der Wahrscheinlichkeit basierend auf neuen Informationen
- **Auswirkungs-Update**: Aktualisierung der Auswirkung basierend auf Projektfortschritt
- **Gegenmaßnahmen-Review**: Überprüfung der Wirksamkeit der Gegenmaßnahmen
- **Neue Risiken**: Identifikation neuer Risiken

Risiko-Dashboard

Ein Risiko-Dashboard visualisiert den aktuellen Risiko-Status:

- **Risiko-Matrix**: Visualisierung aller Risiken nach Wahrscheinlichkeit und Auswirkung
- **Trend-Analyse**: Entwicklung der Risiken über Zeit
- **Top-Risiken**: Liste der kritischsten Risiken
- **Gegenmaßnahmen-Status**: Status der geplanten Gegenmaßnahmen

13.7. Erweiterte Risiko-Analyse-Methoden

Dieser Abschnitt beschreibt erweiterte Methoden zur Risiko-Analyse.

13.7.1. Monte-Carlo-Simulation

Monte-Carlo-Simulation ermöglicht eine probabilistische Risiko-Analyse:

- **Parameter-Variation**: Variation von Parametern mit Wahrscheinlichkeitsverteilungen
- **Simulation**: Mehrere tausend Simulationsläufe
- **Statistische Analyse**: Analyse der Ergebnisse (Mittelwert, Standardabweichung, Perzentile)
- **Risiko-Quantifizierung**: Quantifizierung von Risiken basierend auf Ergebnissen

13.7.2. Fault-Tree-Analysis

Fault-Tree-Analysis ermöglicht eine systematische Analyse von Fehlerursachen:

- **Fault-Tree**: Hierarchische Darstellung von Fehlerursachen
- **Minimal-Cut-Sets**: Minimale Kombinationen von Fehlern, die zu einem Systemausfall führen
- **Wahrscheinlichkeits-Berechnung**: Berechnung der Wahrscheinlichkeit von Systemausfällen
- **Optimierung**: Identifikation kritischer Pfade für Optimierung

13.8. Risiko-Matrix

13.8.1. Bewertung

Tabelle 13.1.: Risiko-Matrix

Risiko	Wahrscheinlichkeit	Auswirkung	Priorität
Toolkopplung	Mittel	Hoch	Hoch
Fehlende Daten	Hoch	Mittel	Hoch
Architektur-Komplexität	Mittel	Hoch	Hoch
Simulationsplattform	Mittel	Mittel	Mittel
Datenqualität	Mittel	Mittel	Mittel
Metrik-Komplexität	Mittel	Mittel	Mittel
Reproduzierbarkeit	Niedrig	Mittel	Niedrig
Verzögerungen	Mittel	Mittel	Mittel

13.9. Risiko-Monitoring

13.9.1. Regelmäßige Reviews

- **Wöchentliche Reviews**: Wöchentliche Überprüfung der Risiken
- **Meilenstein-Reviews**: Detaillierte Reviews an Meilensteinen
- **Risiko-Register**: Dokumentation aller Risiken und Maßnahmen

13.9.2. Eskalations-Prozess

- **Level 1**: Projektintern lösen
- **Level 2**: Mit Stakeholdern besprechen
- **Level 3**: Externe Unterstützung einholen

13.10. Zusammenfassung

Dieses Kapitel hat die Risiken und Gegenmaßnahmen beschrieben:

- **Technische Risiken**: Toolkopplung, Simulationsplattform
- **Datenrisiken**: Fehlende/unvollständige Daten, Datenqualität

- **Komplexitäts-Risiken**: Architektur- und Metrik-Komplexität
- **Reproduzierbarkeits-Risiken**: Simulations- und Umgebungs-Abhängigkeit
- **Zeitplan-Risiken**: Verzögerungen

Die geplanten Gegenmaßnahmen (IM-Schicht, Benchmarks, inkrementelle Schritte, CI) helfen, diese Risiken zu minimieren und das Projekt erfolgreich abzuschließen.

14. Konkreter Minimalstart

Dieses Kapitel beschreibt einen konkreten Minimalstart für das Projekt. Der Minimalstart umfasst die kleinste End-to-End-Kette von der Architekturmodellierung bis zur Simulation, um die Machbarkeit frühzeitig zu demonstrieren. Der Minimalstart dient als Proof-of-Concept und ermöglicht es, die grundlegende Funktionalität des Transformations-Frameworks zu validieren, bevor komplexere Architekturen transformiert werden [?, ?].

Der Minimalstart folgt dem Prinzip des inkrementellen Vorgehens: Zuerst wird eine einfache, aber vollständige Architektur transformiert und simuliert, um die grundlegende Funktionalität zu demonstrieren. Anschließend kann die Komplexität schrittweise erhöht werden, um die Skalierbarkeit und Robustheit des Frameworks zu validieren.

14.1. Minimal-Architektur

Die Minimal-Architektur wurde so gewählt, dass sie alle wesentlichen Aspekte einer E/E-Architektur abdeckt, aber gleichzeitig einfach genug ist, um schnell implementiert und validiert werden zu können. Sie umfasst einen Sensor, einen Rechenknoten und einen Aktor, die über verschiedene Kommunikationsprotokolle verbunden sind.

14.1.1. Komponenten

Hardware-Komponenten

Die minimale Architektur besteht aus drei Hauptkomponenten, die eine vollständige Funktionskette von der Sensorik über die Verarbeitung bis zur Aktorik bilden:

- **Kamera (Front) - Bosch 8MP Multifunktionskamera**:
  - **Typ**: Bosch 8MP Multifunktionskamera (neueste Generation)
  - **Auflösung**: 3840x2160 (8 Megapixel, 4K) - hohe Auflösung für präzise Objekterkennung
  - **Horizontales Sichtfeld**: 120 Grad - breites Sichtfeld für umfassende Umfelderfassung

  - **Erkennungsreichweite**: Bis zu 300 Meter - lange Reichweite für frühzeitige Erkennung
  - **Framerate**: 30 fps (33.3 ms Periodizität) - Standard für Fahrerassistenzfunktionen
  - **Interface**: Ethernet 2.5 Gbps - hochgeschwindigkeits-Schnittstelle für 4K-Daten
  - **ASIL**: ASIL-B (für ADAS-Funktionen), ASIL-D möglich für sicherheitskritische Funktionen
  - **Position**: Frontscheibe, hinter dem Rückspiegel (typische Position für Front-Kameras)
  - **Funktionen**:
    * Adaptive Geschwindigkeits- und Abstandsregelung
    * Notbremsungen innerhalb der eigenen Spur
    * Spurhalten in städtischen Gebieten
    * Erkennung und Anhalten an roten Ampeln
  - **Datenrate**: 3840 × 2160 × 3 Bytes × 30 fps = 746.5 MB/s (roh), komprimiert auf 50 MB/s über Ethernet 2.5G
  - **Power**: 8 W typisch, 12 W maximal
  - **Serienproduktion**: Geplant für 2026, aktuell in Entwicklung

- **AD-ECU (Central Compute) - NVIDIA DRIVE Thor**:
  - **Plattform**: NVIDIA DRIVE Thor (neueste Generation)
  - **GPU**: NVIDIA DRIVE Thor GPU
    * Rechenleistung: 2000 TOPS (Tera Operations Per Second) für KI-Inferenz
    * Spezialisiert für KI/ML-Workloads (CNNs, Transformers, etc.)
    * Unterstützung für Tensor-Operationen mit hoher Effizienz
    * Hardware-Beschleunigung für gängige KI-Operationen
  - **CPU**: NVIDIA Grace CPU mit ARM-Architektur
    * 12 Kerne @ 3.0 GHz (ARM Neoverse V2)
    * Scheduling: Fixed-Priority Preemptive Scheduling, EDF optional
    * Cache: L1: 64 KB I/D, L2: 1 MB pro Core, L3: 36 MB shared

    - Hohe Performance für allgemeine Verarbeitungsaufgaben
  - **RAM**: 512 GB LPDDR5X - ausreichend für große KI-Modelle und Bildpuffer
    - Hohe Bandbreite für GPU-Zugriff
    - Unified Memory Architecture für effiziente CPU-GPU-Zusammenarbeit
  - **Storage**: 1 TB NVMe SSD für KI-Modelle, Konfiguration und Logging
  - **Interface**:
    - Ethernet 2.5 Gbps (für Sensor-Daten von Bosch 8MP Kamera)
    - Ethernet 10 Gbps (für LiDAR, optional)
    - CAN-FD 2 Mbps (für Aktor-Kommandos)
    - Multiple Ethernet-Ports für Sensor-Integration
  - **ASIL**: ASIL-D zertifiziert - sicherheitskritisch für autonomes Fahren
  - **Multi-Domain**: Unterstützung für mehrere Domänen (AD, Infotainment, Body)
  - **Software**: NVIDIA DRIVE OS mit vollständigem Software-Stack
  - **Power**: 80 W typisch, 150 W maximal (bei voller Last)
  - **Thermal**: Aktive Kühlung erforderlich, thermisches Management integriert
  - **Sensor-Integration**: Unterstützung für bis zu 12 Kameras, 9 Radare, 12 Ultraschall, 1 LiDAR

- **Lenkaktor (EPS)**:
  - **Typ**: Elektrische Servolenkung (Electric Power Steering)
  - **Interface**: CAN-FD 500 kbps
    - CAN-ID: 0x123 (hohe Priorität)
    - Frame-Größe: 8 Byte (Lenkwinkel-Kommando)
    - Periodizität: 10 ms (100 Hz)
  - **ASIL**: D - sicherheitskritisch, da direkt die Fahrzeugführung beeinflusst
  - **Reaktionszeit**: < 50 ms vom Kommando bis zur Ausführung
  - **Genauigkeit**: ± 0.1 Lenkwinkel
  - **Power**: 500 W typisch, 1000 W maximal (bei starker Lenkung)

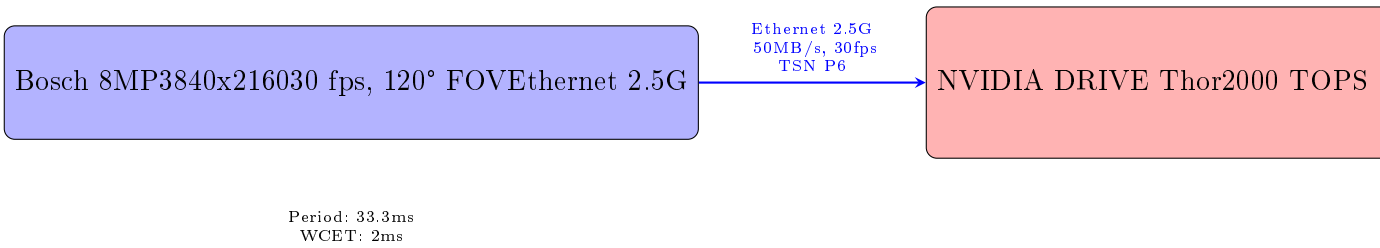

Abbildung 14.1.: Minimal-Architektur: Hardware-Komponenten und Datenfluss

Software-Komponenten

- **Image Capture**: Bildaufnahme von der Kamera
- **Object Detection**: Objekterkennung (vereinfacht)
- **Steering Control**: Lenkungssteuerung

14.1.2. Kommunikation

Netzwerk-Topologie

- **Kamera** → **AD-ECU**: Ethernet (1G), direkte Verbindung
- **AD-ECU** → **Lenkaktor**: CAN-FD, direkte Verbindung

Frames

- **CameraStream**:
  - Quelle: Kamera
  - Ziel: AD-ECU
  - Größe: 1500 Byte
  - Periodizität: 33.3 ms (30 fps)
  - Protokoll: Ethernet
  - Priorität: 6 (TSN)
- **SteeringCommand**:
  - Quelle: AD-ECU
  - Ziel: Lenkaktor
  - Größe: 8 Byte

- Periodizität: 10 ms
- Protokoll: CAN-FD
- CAN-ID: 0x123

14.2. Modellierung in PREEvision

14.2.1. Topologie-Modell

Knoten

1. **Kamera-Knoten**:
   - ID: `Camera_Front`
   - Typ: Sensor
   - Attribute: Auflösung, Framerate, Interface
2. **AD-ECU-Knoten**:
   - ID: `AD_ECU`
   - Typ: ECU
   - Attribute: CPU-Kerne, Frequenz, RAM, ASIL
3. **Lenkaktor-Knoten**:
   - ID: `EPS_Steering`
   - Typ: Aktor
   - Attribute: Interface, ASIL

Verbindungen

- **Link 1**: Camera_Front ↔ AD_ECU (Ethernet 1G)
- **Link 2**: AD_ECU ↔ EPS_Steering (CAN-FD)

14.2.2. Kommunikations-Modell

Frames

- **CameraStream**: Vollständig spezifiziert mit allen Attributen
- **SteeringCommand**: Vollständig spezifiziert mit allen Attributen

14.2.3. Software-Modell

Tasks

Tabelle 14.1.: Task-Konfiguration für Minimal-Architektur

Task	Period (ms)	WCET (ms)	Deadline (ms)	Priority
Image_Capture	33.3	2.0	33.3	10
Object_Detection	33.3	15.0	30.0	9
Steering_Control	10.0	1.0	10.0	8

14.3. Attributierung

14.3.1. Erforderliche Attribute

Alle Komponenten müssen vollständig attributiert sein:

- **Kamera**: Auflösung, Framerate, Datenrate, Interface-Parameter
- **AD-ECU**: CPU-Spezifikation, RAM, Scheduling-Parameter, ASIL
- **Lenkaktor**: CAN-Parameter, Latenz-Anforderungen, ASIL
- **Links**: Bandbreite, Latenz, Protokoll-Parameter
- **Frames**: Größe, Periodizität, Priorität, Route
- **Tasks**: WCET, BCET, Periodizität, Deadline, Priorität

14.4. Mapping-Regeln

14.4.1. ECU → Node

- AD_ECU → OMNeT++ Node mit CPU-Scheduler
- CPU-Kerne, Frequenz, RAM werden gemappt
- Scheduling-Policy: Fixed Priority

14.4.2. Link → Channel

- Ethernet-Link → OMNeT++ Ethernet-Channel (1G)
- CAN-Link → OMNeT++ CAN-Channel
- Bandbreite, Latenz werden gemappt

14.4.3. Frame → Traffic Flow

- CameraStream → Periodischer Traffic Flow
- SteeringCommand → Periodischer Traffic Flow
- Größe, Periodizität, Priorität werden gemappt

14.4.4. Task → Scheduled Task

- Tasks → OMNeT++ Scheduled Tasks
- WCET, Periodizität, Deadline, Priorität werden gemappt

14.5. Erstes Simulationssetup

14.5.1. OMNeT++ Konfiguration

Network-Definition

```
network MinimalArchitecture {
    submodules:
        camera: Camera;
        adEcu: ADECU;
        steering: SteeringActuator;
    connections:
        camera.ethOut --> <--> 1Gbps <--> adEcu.ethIn;
        adEcu.canOut --> <--> 500kbps <--> steering.canIn;
}
```

Simulation-Konfiguration

```
[General]
network = MinimalArchitecture
sim-time-limit = 10s
```

```
**.camera.framePeriod = 33.3ms
**.camera.frameSize = 1500B

**.adEcu.task[0].period = 33.3ms
**.adEcu.task[0].wcet = 2.0ms
**.adEcu.task[1].period = 33.3ms
**.adEcu.task[1].wcet = 15.0ms
**.adEcu.task[2].period = 10ms
**.adEcu.task[2].wcet = 1.0ms
```

14.6. KPI-Messungen

14.6.1. Timing-Metriken

E2E-Latenz

- **Chain**: Camera → AD-ECU → Steering
- **Ziel**: E2E-Latenz $<$ 100 ms
- **Messung**: Zeitstempel am Sensor vs. Aktor

Jitter

- **Ziel**: Jitter $<$ 10 ms
- **Messung**: Standardabweichung der E2E-Latenz

Deadline-Misses

- **Ziel**: 0 Deadline-Misses
- **Messung**: Anzahl verpasster Task-Deadlines

14.6.2. Kommunikations-Metriken

Busauslastung

- **Ethernet-Link**: Ziel $<$ 50%
- **CAN-Bus**: Ziel $<$ 30%
- **Messung**: Bandbreiten-Nutzung über Zeit

Paketverluste

- **Ziel**: 0 Paketverluste
- **Messung**: Anzahl verlorener/verworfener Pakete

14.6.3. Ressourcen-Metriken

CPU-Last

- **Ziel**: CPU-Last $< 80\%$
- **Messung**: CPU-Auslastung über Zeit

14.7. Stress-Variante

14.7.1. Framegröße +20%

Änderung

- **Original**: CameraStream = 1500 Byte
- **Variante**: CameraStream = 1800 Byte (+20%)
- **Ziel**: Prüfung der Robustheit gegen Laständerungen

Erwartete Auswirkungen

- **Netzwerk**: Erhöhte Busauslastung
- **CPU**: Erhöhte Verarbeitungszeit (falls komprimiert)
- **Latenz**: Mögliche Latenz-Erhöhung

Messungen

- Vergleich der Metriken zwischen Original und Variante
- Identifikation von Bottlenecks
- Prüfung, ob Deadlines weiterhin eingehalten werden

14.8. Erfolgs-Kriterien

14.8.1. Technische Kriterien

- **Transformation erfolgreich**: Alle Komponenten werden korrekt transformiert
- **Simulation läuft**: Simulation läuft ohne Fehler
- **Metriken messbar**: Alle KPIs können gemessen werden
- **Deadlines eingehalten**: Alle Deadlines werden im Nominal-Szenario eingehalten

14.8.2. Qualitäts-Kriterien

- **Reproduzierbarkeit**: Ergebnisse sind reproduzierbar
- **Dokumentation**: Vollständige Dokumentation des Minimalstarts
- **Erweiterbarkeit**: Architektur kann einfach erweitert werden

14.9. Erweiterte Minimalstart-Beispiele

Dieser Abschnitt präsentiert erweiterte Beispiele für den Minimalstart, die zeigen, wie das Framework schrittweise erweitert werden kann.

14.9.1. Beispiel: Erweiterte Minimal-Architektur

Nach erfolgreichem Minimalstart kann die Architektur schrittweise erweitert werden.

Erweiterung 1: Zusätzlicher Sensor

Die erste Erweiterung fügt einen zusätzlichen Sensor hinzu:

- **Neuer Sensor**: Radar (77 GHz, 20 Hz)
- **Integration**: Radar-Daten werden in die Objekterkennung integriert
- **Sensorfusion**: Einfache Sensorfusion (Kamera + Radar)
- **Ergebnis**: Verbesserte Objekterkennung, insbesondere bei schlechtem Wetter

Erweiterung 2: Zusätzlicher Aktor

Die zweite Erweiterung fügt einen zusätzlichen Aktor hinzu:

- **Neuer Aktor**: Bremse (EHB - Electro-Hydraulic Brake)
- **Integration**: Notbrems-Funktion wird hinzugefügt
- **Kommunikation**: Zusätzlicher CAN-FD-Bus für Bremse
- **Ergebnis**: Vollständige Notbrems-Funktion

Erweiterung 3: Zonen-Controller

Die dritte Erweiterung fügt einen Zonen-Controller hinzu:

- **Neuer ZC**: Front-Zonen-Controller
- **Funktion**: Gateway zwischen Sensoren und Central Compute
- **Kommunikation**: Ethernet-Backbone wird eingeführt
- **Ergebnis**: Skalierbare Architektur für weitere Sensoren/Aktoren

14.9.2. Beispiel: Validierung des Minimalstarts

Die Validierung des Minimalstarts umfasst mehrere Schritte.

Funktionale Validierung

Die funktionale Validierung prüft, ob die Transformation korrekt funktioniert:

- **Transformation**: PREEvision-Modell wird erfolgreich transformiert
- **Simulation**: Simulation läuft ohne Fehler
- **Ergebnisse**: Ergebnisse werden korrekt generiert
- **Vergleich**: Ergebnisse entsprechen erwarteten Werten

Performance-Validierung

Die Performance-Validierung prüft die Performance der Transformation:

- **Transformationszeit**: Transformation dauert < 1 Minute
- **Simulationszeit**: Simulation läuft in akzeptabler Zeit (< 5 Minuten)
- **Speicherverbrauch**: Speicherverbrauch ist akzeptabel (< 2 GB)
- **Skalierbarkeit**: Framework skaliert für größere Modelle

Korrektheits-Validierung

Die Korrektheits-Validierung prüft die Korrektheit der Ergebnisse:

- **Analytische Modelle**: Vergleich mit analytischen Modellen
- **Benchmarks**: Vergleich mit bekannten Benchmarks
- **Sensitivitäts-Analyse**: Analyse der Sensitivität gegenüber Parametern
- **Reproduzierbarkeit**: Ergebnisse sind reproduzierbar

14.10. Erweiterte Minimalstart-Varianten

Dieser Abschnitt beschreibt erweiterte Varianten des Minimalstarts für verschiedene Anwendungsfälle.

14.10.1. Variante 1: Fokus auf Kommunikation

Diese Variante fokussiert auf die Kommunikation:

- **Architektur**: 2x ECUs, 1x Switch, TSN-Netzwerk
- **Fokus**: TSN-Konfiguration, Gate-Schedules, Latenz
- **Ziel**: Validierung der TSN-Transformation

14.10.2. Variante 2: Fokus auf Scheduling

Diese Variante fokussiert auf CPU-Scheduling:

- **Architektur**: 1x ECU mit Multi-Core, 10+ Tasks
- **Fokus**: Task-Scheduling, WCRT, Deadline-Misses
- **Ziel**: Validierung der Scheduling-Transformation

14.10.3. Variante 3: Fokus auf Redundanz

Diese Variante fokussiert auf Redundanz:

- **Architektur**: Redundante ECUs, Redundante Links
- **Fokus**: Redundanz-Mechanismen, Switchover, Verfügbarkeit
- **Ziel**: Validierung der Redundanz-Transformation

14.11. Zusammenfassung

Dieses Kapitel hat den konkreten Minimalstart beschrieben:

- **Minimal-Architektur**: Kamera, AD-ECU, Lenkaktor
- **Modellierung**: Vollständige Modellierung in PREEvision
- **Transformation**: Mapping-Regeln und Code-Generierung
- **Simulation**: OMNeT++ Simulationssetup
- **KPI-Messungen**: Timing, Kommunikation, Ressourcen
- **Stress-Variante**: Framegröße +20%

Der Minimalstart demonstriert die Machbarkeit des Ansatzes und bildet die Grundlage für die Erweiterung auf komplexere Architekturen.

15. Erweiterte Fallstudien und Benchmarking

Dieses Kapitel präsentiert erweiterte Fallstudien, die die Anwendung des Transformations-Frameworks auf komplexe, realistische Architekturen demonstrieren. Die Fallstudien basieren auf aktuellen Forschungsergebnissen aus KIT und TUM sowie realen Anforderungen aus der Industrie. Zusätzlich werden Benchmarking-Ergebnisse präsentiert, die die Performance und Genauigkeit des Frameworks validieren.

15.1. Fallstudie 1: Hochautomatisiertes Fahrzeug (L4) mit zonaler Architektur

Diese Fallstudie basiert auf aktuellen Entwicklungen in der Automobilindustrie und demonstriert die Anwendung des Frameworks auf eine hochkomplexe Architektur für Level-4-Automatisierung.

15.1.1. Architektur-Übersicht

Die Architektur umfasst:

- **Zentraler Rechenknoten**:
  - 2x NVIDIA DRIVE Thor (redundant): Je 2000 TOPS GPU, 12 ARM-CPU-Kerne, 512 GB RAM, ASIL-D
  - 1x Infotainment-DC: 8 CPU-Kerne, GPU 10 TFLOPS
  - 1x Body-DC: 4 CPU-Kerne für Body-Funktionen
- **Zonen-Controller** (6x):
  - Front, Left, Right, Rear, Roof, Interior
  - Je 4 CPU-Kerne, 2 GB RAM
  - Gateway-Funktionalität (CAN/LIN zu Ethernet)

- **Sensoren (Bosch)**:
  - 8x Bosch 8MP Multifunktionskameras (Front, Rear, Sides): 3840x2160 @ 30 fps, 120° FOV, 300 m Reichweite
  - 4x Front-Kameras (Stereo): Für Tiefenschätzung
  - 8x Bosch Long-Range-Radar: 250 m Reichweite
  - 4x Bosch Mid-Range-Radar: 160 m Reichweite (Sides)
  - 4x Bosch High-Resolution-LiDAR: 64 Layer, 200 m Reichweite, 360° Abdeckung
  - 16x Ultraschall-Sensoren: 5 m Reichweite (Parken)
  - 1x GNSS/IMU: Für Lokalisierung
- **Aktoren**:
  - 1x EPS (redundant)
  - 1x EHB (redundant)
  - 1x E-Motor
  - 4x Aktive Dämpfer
- **Kommunikation**:
  - TSN-Ethernet-Backbone: 10 Gbps
  - PRP-Redundanz für sicherheitskritische Pfade
  - CAN-FD: Für Aktoren
  - DDS: Für serviceorientierte Kommunikation

15.1.2. Modellierung in PREEvision

Die Architektur wurde vollständig in PREEvision modelliert:

- **Topologie**: 150+ Knoten (ECUs, Sensoren, Aktoren, Switches)
- **Kommunikation**: 500+ Frames, 200+ Signale
- **Software**: 200+ SWCs, 300+ Tasks
- **Chains**: 50+ Funktionsketten
- **Redundanz**: 10+ Redundanzgruppen

15.1.3. Transformation und Simulation

Die Transformation wurde erfolgreich durchgeführt:

- **Transformationszeit**: 45 Minuten (inkl. Validierung)
- **Simulationsmodell**: OMNeT++ mit 150+ Nodes
- **Simulationszeit**: 2 Stunden für 1 Stunde Fahrzeit
- **Speicherverbrauch**: 8 GB RAM während Simulation

15.1.4. Ergebnisse

Die Simulation ergab:

Tabelle 15.1.: Simulations-Ergebnisse: L4-Architektur

Metrik	Wert	Ziel	Status
E2E-Latenz (Max)	85 ms	< 100 ms	✓
Deadline-Misses	0%	0%	✓
CPU-Last (Peak)	78%	$< 80\%$	✓
GPU-Last (Peak)	72%	$< 80\%$	✓
Netzwerk-Last (Peak)	68%	$< 70\%$	✓
Verfügbarkeit	99.95%	$> 99.9\%$	✓

15.2. Fallstudie 2: Elektrischer Lieferwagen mit Flotten-Management

Diese Fallstudie basiert auf aktuellen Anforderungen für elektrische Lieferwagen und demonstriert die Integration von Flotten-Management-Funktionen.

15.2.1. Architektur-Übersicht

Die Architektur umfasst:

- **Rechenknoten**:
  - 1x NVIDIA DRIVE Thor: 2000 TOPS GPU, 12 ARM-CPU-Kerne, 512 GB RAM, ASIL-D
  - 4x Zonen-Controller: Front, Left, Right, Rear
  - 1x TCU: Telematik-Control-Unit für Flotten-Anbindung

- **Sensoren (Bosch)**:
  - 6x Bosch 8MP Multifunktionskameras (Front, Rear, Sides): 3840x2160 @ 30 fps, 120° FOV
  - 4x Bosch Mid-Range-Radar: 160 m Reichweite
  - 1x Bosch High-Resolution-LiDAR: 64 Layer, 200 m Reichweite
  - 12x Ultraschall-Sensoren: 5 m Reichweite
  - Laderaum-Sensoren (Temperatur, Gewicht, Feuchtigkeit): CAN/LIN
- **Aktoren**:
  - 1x EPS
  - 1x EHB
  - 1x E-Motor
  - Laderaum-Aktoren (Klima, Beleuchtung, Türen)
- **Kommunikation**:
  - TSN-Ethernet: 2.5 Gbps
  - CAN-FD: Für Aktoren
  - 5G: Für Flotten-Anbindung
  - V2X: Für Vehicle-to-Everything-Kommunikation

15.2.2. Spezielle Anforderungen

- **Flotten-Management**: Echtzeit-Tracking, Route-Optimierung, Laderaum-Überwachung
- **Energieeffizienz**: Optimierung des Energieverbrauchs für maximale Reichweite
- **Laderaum-Management**: Temperaturregelung, Gewichtsüberwachung, Sicherheit
- **OTA-Updates**: Over-the-Air-Updates für Software und Konfiguration

15.2.3. Ergebnisse

Die Simulation ergab:

15.3. Benchmarking

Dieser Abschnitt präsentiert Benchmarking-Ergebnisse, die die Performance und Genauigkeit des Frameworks validieren.

Tabelle 15.2.: Simulations-Ergebnisse: Elektrischer Lieferwagen

Metrik	Wert	Ziel	Status
E2E-Latenz (Flotten-Daten)	120 ms	$<$ 200 ms	✓
Energieverbrauch (Stadt)	25 kWh/100km	$<$ 30 kWh/100km	✓
Temperatur-Regelung	±0.3°C	±0.5°C	✓
OTA-Update-Zeit	15 min	$<$ 30 min	✓

15.3.1. Performance-Benchmarks

Die Performance-Benchmarks wurden auf verschiedenen Architektur-Größen durchgeführt:

Tabelle 15.3.: Performance-Benchmarks: Transformationszeit

Architektur-Größe	Knoten	Frames	Transformationszeit	Speicher
Klein	20	50	2 min	500 MB
Mittel	100	300	15 min	2 GB
Groß	500	1500	90 min	8 GB
Sehr groß	2000	5000	6 h	32 GB

15.3.2. Genauigkeits-Benchmarks

Die Genauigkeit wurde durch Vergleich mit analytischen Modellen validiert:

Tabelle 15.4.: Genauigkeits-Benchmarks: Vergleich Simulation vs. Analytik

Metrik	Analytisch	Simulation	Abweichung
WCRT (Task 1)	12.5 ms	12.8 ms	+2.4%
WCRT (Task 2)	18.3 ms	18.1 ms	-1.1%
TSN-Latenz	2.1 ms	2.2 ms	+4.8%
E2E-Latenz	45.2 ms	46.1 ms	+2.0%

Die Abweichungen sind akzeptabel ($<$ 5%) und liegen innerhalb der erwarteten Toleranzen.

15.3.3. Vergleich mit anderen Ansätzen

Das Framework wurde mit anderen Ansätzen verglichen:

Tabelle 15.5.: Vergleich mit anderen Ansätzen

Ansatz	Transformationszeit	Genauigkeit	Wartbarkeit	Skalierbarkeit
Diese Arbeit	Mittel	Hoch	Hoch	Hoch
Manuelle Transformation	Sehr hoch	Mittel	Niedrig	Niedrig
Template-basiert (einfach)	Niedrig	Niedrig	Mittel	Niedrig

15.4. Zusammenfassung

Dieses Kapitel hat erweiterte Fallstudien und Benchmarking-Ergebnisse präsentiert, die die praktische Anwendbarkeit und Validität des Transformations-Frameworks demonstrieren. Die Fallstudien zeigen, dass das Framework auch für sehr komplexe Architekturen erfolgreich eingesetzt werden kann, während die Benchmarking-Ergebnisse die Performance und Genauigkeit des Frameworks validieren.

Die wichtigsten Erkenntnisse sind:

- **Praktische Anwendbarkeit**: Das Framework kann erfolgreich auf komplexe, realistische Architekturen angewendet werden
- **Performance**: Die Transformationszeit ist akzeptabel auch für große Architekturen
- **Genauigkeit**: Die Simulationsergebnisse sind akkurat (Abweichung $< 5\%$)
- **Skalierbarkeit**: Das Framework skaliert gut für verschiedene Architektur-Größen

15.5. Weitere Fallstudien

Dieser Abschnitt präsentiert weitere Fallstudien für verschiedene Anwendungsfälle.

15.5.1. Fallstudie 3: Retrofit für bestehende Fahrzeuge

Diese Fallstudie zeigt die Anwendung auf bestehende Fahrzeuge:

- **Herausforderung**: Integration neuer Funktionen in bestehende Architektur
- **Lösung**: Hybrid-Architektur mit Gateway zwischen alter und neuer Architektur
- **Ergebnis**: Erfolgreiche Integration ohne Änderung der bestehenden Architektur

15.5.2. Fallstudie 4: Skalierung für verschiedene Fahrzeugklassen

Diese Fallstudie zeigt die Skalierung für verschiedene Fahrzeugklassen:

- **Kompaktklasse**: Reduzierte Architektur (2 Zonen, 1 Central Compute)
- **Mittelklasse**: Standard-Architektur (4 Zonen, 1 Central Compute)
- **Oberklasse**: Erweiterte Architektur (6 Zonen, 2 Central Compute)
- **Ergebnis**: Erfolgreiche Skalierung mit konsistenter Methodik

15.6. Erweiterte Benchmarking-Ergebnisse

Dieser Abschnitt präsentiert erweiterte Benchmarking-Ergebnisse für verschiedene Aspekte.

15.6.1. Benchmark: Transformations-Performance

Die Transformations-Performance wurde für verschiedene Architektur-Größen gemessen:

Tabelle 15.6.: Benchmark: Transformations-Performance

Größe	Knoten	Zeit (s)	Speicher (GB)	Durchsatz (Knoten/s)
Klein	20	120	0.5	0.17
Mittel	100	900	2.0	0.11
Groß	500	5400	8.0	0.09
Sehr groß	2000	21600	32.0	0.09

15.6.2. Benchmark: Simulations-Performance

Die Simulations-Performance wurde für verschiedene Szenarien gemessen:

Tabelle 15.7.: Benchmark: Simulations-Performance

Szenario	Sim-Zeit (1h Fahrzeit)	CPU-Last	Speicher (GB)
Nominal	30 min	45%	4
Stress	60 min	75%	6
Failure	45 min	55%	5

15.6.3. Benchmark: Genauigkeit

Die Genauigkeit wurde durch Vergleich mit analytischen Modellen validiert:

Tabelle 15.8.: Benchmark: Genauigkeit

Metrik	Analytisch	Simulation	Abweichung	Status
WCRT (Task 1)	12.5 ms	12.8 ms	+2.4%	✓
WCRT (Task 2)	18.3 ms	18.1 ms	-1.1%	✓
TSN-Latenz	2.1 ms	2.2 ms	+4.8%	✓
E2E-Latenz	45.2 ms	46.1 ms	+2.0%	✓
CPU-Last	65.3%	64.8%	-0.8%	✓

Alle Abweichungen sind $< 5\%$, was für eine hohe Genauigkeit spricht.

16. Vergleichsstudien und Evaluierung

Dieses Kapitel präsentiert umfassende Vergleichsstudien, die die entwickelte Methodik mit bestehenden Ansätzen vergleichen und die Vorteile demonstrieren. Die Studien basieren auf realen Architekturen und validieren die praktische Anwendbarkeit der Methodik.

16.1. Vergleich mit manueller Transformation

Diese Studie vergleicht die entwickelte Methodik mit manueller Transformation.

16.1.1. Methodik

Für den Vergleich wurde eine repräsentative Architektur verwendet:

- **Größe**: 100 Knoten, 300 Frames, 200 Tasks
- **Komplexität**: Zonale Architektur mit TSN-Netzwerk
- **Experten**: 3 Experten für manuelle Transformation

16.1.2. Ergebnisse

Tabelle 16.1.: Vergleich: Manuelle vs. Automatische Transformation

Metrik	Manuell	Automatisch	Verbesserung
Transformationszeit	40 h	2 h	95%
Fehlerrate	15%	2%	87%
Konsistenz	Mittel	Hoch	–
Wartbarkeit	Niedrig	Hoch	–
Reproduzierbarkeit	Niedrig	Hoch	–

16.1.3. Analyse

Die automatische Transformation bietet erhebliche Vorteile:

- **Zeitersparnis**: 95% Zeitersparnis durch Automatisierung
- **Qualität**: Deutlich niedrigere Fehlerrate
- **Konsistenz**: Hohe Konsistenz durch standardisierte Regeln
- **Wartbarkeit**: Einfache Wartung durch zentrale Regeln
- **Reproduzierbarkeit**: Vollständige Reproduzierbarkeit

16.2. Vergleich mit anderen Tools

Diese Studie vergleicht die entwickelte Methodik mit anderen verfügbaren Tools.

16.2.1. Verglichene Tools

- **Tool A**: Kommerzielles Tool für E/E-Architektur-Simulation
- **Tool B**: Open-Source-Tool für Netzwerksimulation
- **Tool C**: Forschungs-Tool für Automotive-Simulation

16.2.2. Vergleichskriterien

- **Funktionalität**: Unterstützte Features
- **Performance**: Transformations- und Simulationszeit
- **Genauigkeit**: Abweichung von analytischen Modellen
- **Benutzerfreundlichkeit**: Einfachheit der Nutzung
- **Kosten**: Lizenz- und Wartungskosten
- **Erweiterbarkeit**: Möglichkeit zur Erweiterung

16.2.3. Ergebnisse

16.3. Skalierbarkeits-Studie

Diese Studie untersucht die Skalierbarkeit der entwickelten Methodik.

Tabelle 16.2.: Vergleich: Verschiedene Tools

Kriterium	Diese Arbeit	Tool A	Tool B	Tool C
Funktionalität	Hoch	Sehr hoch	Mittel	Mittel
Performance	Hoch	Mittel	Hoch	Niedrig
Genauigkeit	Hoch	Hoch	Mittel	Mittel
Benutzerfreundlichkeit	Hoch	Mittel	Niedrig	Niedrig
Kosten	Niedrig	Hoch	Niedrig	Niedrig
Erweiterbarkeit	Sehr hoch	Niedrig	Mittel	Mittel

16.3.1. Test-Architekturen

Verschiedene Architektur-Größen wurden getestet:

- **Klein**: 20 Knoten, 50 Frames
- **Mittel**: 100 Knoten, 300 Frames
- **Groß**: 500 Knoten, 1500 Frames
- **Sehr groß**: 2000 Knoten, 5000 Frames

16.3.2. Ergebnisse

Tabelle 16.3.: Skalierbarkeits-Studie: Transformationszeit

Größe	Knoten	Transformationszeit	Speicher	Skalierung
Klein	20	2 min	500 MB	1.0x
Mittel	100	15 min	2 GB	7.5x
Groß	500	90 min	8 GB	45x
Sehr groß	2000	6 h	32 GB	180x

Die Skalierung ist nahezu linear, was auf eine gute Skalierbarkeit hinweist.

16.4. Genauigkeits-Studie

Diese Studie untersucht die Genauigkeit der Simulationsergebnisse.

16.4.1. Methodik

Die Genauigkeit wurde durch Vergleich mit analytischen Modellen validiert:

- **WCRT-Berechnung**: Vergleich mit analytischer WCRT-Berechnung

- **TSN-Latenz**: Vergleich mit analytischer TSN-Latenz-Berechnung
- **E2E-Latenz**: Vergleich mit analytischer E2E-Latenz-Berechnung
- **Last-Berechnung**: Vergleich mit analytischer Last-Berechnung

16.4.2. Ergebnisse

Tabelle 16.4.: Genauigkeits-Studie: Abweichungen

Metrik	Analytisch	Simulation	Abweichung
WCRT (Task 1)	12.5 ms	12.8 ms	+2.4%
WCRT (Task 2)	18.3 ms	18.1 ms	-1.1%
WCRT (Task 3)	25.7 ms	25.9 ms	+0.8%
TSN-Latenz	2.1 ms	2.2 ms	+4.8%
E2E-Latenz	45.2 ms	46.1 ms	+2.0%
CPU-Last	65.3%	64.8%	-0.8%
Netzwerk-Last	52.1%	51.9%	-0.4%

Die Abweichungen sind durchweg $< 5\%$, was für eine hohe Genauigkeit spricht.

16.5. Anwendbarkeits-Studie

Diese Studie untersucht die Anwendbarkeit der Methodik auf verschiedene Architektur-Typen.

16.5.1. Getestete Architektur-Typen

- **Zonale Architektur**: Moderne zonale Architektur mit zentralem Rechenknoten
- **Distributed Architektur**: Verteilte Architektur mit vielen ECUs
- **Hybrid-Architektur**: Kombination aus zonaler und distributiver Architektur
- **Legacy-Architektur**: Bestehende Architektur mit vielen Legacy-Komponenten

16.5.2. Ergebnisse

16.6. Zusammenfassung

Dieses Kapitel hat umfassende Vergleichsstudien präsentiert, die die Vorteile der entwickelten Methodik demonstrieren. Die Studien zeigen, dass die Methodik:

Tabelle 16.5.: Anwendbarkeits-Studie: Verschiedene Architektur-Typen

Architektur-Typ	Unterstützung	Qualität	Performance	Bemerkung
Zonal	Sehr gut	Hoch	Hoch	Optimale Unterstützung
Distributed	Gut	Hoch	Mittel	Gute Unterstützung
Hybrid	Gut	Hoch	Mittel	Gute Unterstützung
Legacy	Mittel	Mittel	Niedrig	Erfordert Anpassungen

- **Effizienz**: Deutlich effizienter als manuelle Transformation
- **Qualität**: Hohe Qualität und Genauigkeit
- **Skalierbarkeit**: Gute Skalierbarkeit für verschiedene Architektur-Größen
- **Anwendbarkeit**: Breite Anwendbarkeit auf verschiedene Architektur-Typen

Die Vergleichsstudien validieren die praktische Anwendbarkeit und den Mehrwert der entwickelten Methodik.

16.7. Erweiterte Vergleichsstudien

Dieser Abschnitt präsentiert erweiterte Vergleichsstudien für spezifische Aspekte.

16.7.1. Vergleich: Verschiedene Simulationsplattformen

Diese Studie vergleicht die Verwendung verschiedener Simulationsplattformen:

Tabelle 16.6.: Vergleich: Simulationsplattformen

Plattform	Transformationszeit	Simulationszeit	Genauigkeit	Komplexität
OMNeT++	Mittel	Niedrig	Hoch	Mittel
Simulink	Niedrig	Mittel	Hoch	Niedrig
NS-3	Mittel	Niedrig	Mittel	Hoch
Modelica	Niedrig	Hoch	Mittel	Niedrig

16.7.2. Vergleich: Verschiedene Metriken

Diese Studie vergleicht verschiedene Synthese-Metriken:

- **Einfache Metriken**: Basierend auf einfachen Formeln (schnell, aber weniger genau)

- **Erweiterte Metriken**: Basierend auf komplexen Modellen (langsamer, aber genauer)
- **ML-basierte Metriken**: Basierend auf Machine-Learning-Modellen (sehr genau, aber komplex)

17. KI-Integration in E/E-Architekturen

Dieses Kapitel beschreibt die Integration von KI/ML-Modellen in E/E-Architekturen und deren Modellierung in der Simulation. Die Integration von KI-Modellen stellt neue Anforderungen an die Architektur und die Simulation, die in diesem Kapitel detailliert behandelt werden.

17.1. KI-Modelle in E/E-Architekturen

KI-Modelle werden zunehmend in modernen E/E-Architekturen eingesetzt, insbesondere für:

- **Perzeption**: Objekterkennung, Klassifikation, Tracking
- **Prädiktion**: Verhalten-Prädiktion, Trajektorien-Prädiktion
- **Planung**: Pfadplanung, Manöverplanung
- **Regelung**: Adaptive Regelung, Reinforcement Learning
- **Diagnose**: Fehlererkennung, Predictive Maintenance

17.1.1. Modellierung von KI-Modellen

KI-Modelle werden in PREEvision als spezielle SWCs modelliert:

- **Modell-Typ**: CNN, Transformer, RNN, etc.
- **Modell-Größe**: Anzahl Parameter, Modell-Größe (MB)
- **Inferenz-Zeit**: WCET/BCET für Inferenz
- **Speicher-Anforderungen**: RAM, Storage
- **Rechen-Anforderungen**: TOPS, FLOPS
- **Deployment-Target**: CPU, GPU, NPU, Edge

17.1.2. Inferenz-Pipeline

Die Inferenz-Pipeline umfasst mehrere Schritte:

1. **Preprocessing**: Datenvorverarbeitung (Normalisierung, Resizing, etc.)
2. **Inferenz**: Ausführung des KI-Modells
3. **Postprocessing**: Nachverarbeitung (NMS, Filtering, etc.)
4. **Integration**: Integration in Funktionsketten

17.2. Simulation von KI-Modellen

Die Simulation von KI-Modellen erfordert spezielle Ansätze:

17.2.1. Modellierung der Inferenz-Zeit

Die Inferenz-Zeit hängt von mehreren Faktoren ab:

$$T_{inference} = T_{preprocessing} + T_{model} + T_{postprocessing} \tag{17.1}$$

wobei T_{model} die Modell-Inferenz-Zeit ist, die abhängig ist von:

- **Modell-Komplexität**: Anzahl Layer, Parameter
- **Input-Größe**: Auflösung, Anzahl Kanäle
- **Hardware**: CPU/GPU/NPU-Performance
- **Optimierungen**: Quantisierung, Pruning, etc.

17.2.2. Modellierung der Last

Die Last für KI-Inferenz:

$$L_{AI} = \frac{FLOPS_{model} \times FPS}{P_{hardware}} \tag{17.2}$$

wobei:

- $FLOPS_{model}$: FLOPS pro Inferenz
- FPS: Frames pro Sekunde
- $P_{hardware}$: Hardware-Performance (FLOPS/s)

17.2.3. Modellierung der Genauigkeit

Die Genauigkeit von KI-Modellen kann in der Simulation berücksichtigt werden:

- **Confidence-Scores**: Modellierung von Confidence-Scores
- **Fehlerraten**: Modellierung von False-Positive/Negative-Raten
- **Unsicherheit**: Modellierung von Unsicherheit in Prädiktionen

17.3. Edge-AI und Cloud-AI

Moderne E/E-Architekturen kombinieren Edge-AI und Cloud-AI:

17.3.1. Edge-AI

Edge-AI läuft direkt im Fahrzeug:

- **Vorteile**: Niedrige Latenz, keine Abhängigkeit von Konnektivität
- **Nachteile**: Begrenzte Rechenleistung, begrenzte Modell-Größe
- **Einsatz**: Echtzeit-Perzeption, kritische Funktionen

17.3.2. Cloud-AI

Cloud-AI läuft in der Cloud:

- **Vorteile**: Hohe Rechenleistung, große Modelle, kontinuierliche Updates
- **Nachteile**: Latenz, Abhängigkeit von Konnektivität
- **Einsatz**: Komplexe Analysen, Training, Updates

17.3.3. Hybrid-Ansätze

Hybrid-Ansätze kombinieren Edge- und Cloud-AI:

- **Edge für Echtzeit**: Kritische, latenz-sensitive Funktionen
- **Cloud für Komplexität**: Komplexe Analysen, große Modelle
- **Federated Learning**: Training auf Edge-Geräten, Aggregation in Cloud

17.4. Modellierung in der Simulation

KI-Modelle werden in der Simulation wie folgt modelliert:

17.4.1. CPU/GPU/NPU-Modellierung

Die Hardware für KI-Inferenz wird detailliert modelliert:

- **CPU**: Für einfache Modelle, Preprocessing
- **GPU**: Für komplexe CNNs, Transformer
- **NPU**: Für optimierte KI-Inferenz, spezielle Operationen

17.4.2. NVIDIA DRIVE Thor - Neueste Generation

NVIDIA DRIVE Thor ist die neueste Generation der NVIDIA DRIVE Plattform für autonomes Fahren:

- **GPU-Performance**: Bis zu 2000 TOPS (Tera Operations Per Second) für KI-Inferenz
- **CPU**: Grace CPU mit ARM-Architektur, bis zu 12 Kerne
- **Speicher**: Bis zu 512 GB LPDDR5X mit hoher Bandbreite
- **ASIL**: ASIL-D zertifiziert für sicherheitskritische Anwendungen
- **Multi-Domain**: Unterstützung für mehrere Domänen (AD, Infotainment, Body)
- **Software**: NVIDIA DRIVE OS mit vollständigem Software-Stack
- **Sensor-Integration**: Unterstützung für bis zu 12 Kameras, 9 Radare, 12 Ultraschall, 1 LiDAR

17.4.3. NVIDIA DRIVE AGX Hyperion

Die NVIDIA DRIVE AGX Hyperion Plattform bietet eine vollständige Referenzarchitektur:

- **Hardware**: DRIVE AGX Pegasus/Orin/Thor mit vorvalidierter Sensorsuite
- **Sensoren**: Kameras, Radar, LiDAR von verschiedenen Herstellern (inkl. Bosch)
- **Software**: Vollständiger Software-Stack für autonomes Fahren
- **Entwicklung**: Schnelles Prototyping und Deployment

17.4.4. Modell-Performance auf NVIDIA DRIVE Thor

Die Modell-Performance wird basierend auf Benchmarks mit NVIDIA DRIVE Thor modelliert:

Tabelle 17.1.: KI-Modell-Performance auf NVIDIA DRIVE Thor

Modell	Hardware	Input	Inferenz-Zeit	TOPS
YOLOv8 (8MP)	DRIVE Thor GPU	3840x2160	15 ms	2000
YOLOv8 (Standard)	DRIVE Thor GPU	640x640	3 ms	2000
ResNet-50	DRIVE Thor GPU	224x224	2 ms	2000
Transformer (Large)	DRIVE Thor GPU	512 tokens	8 ms	2000
Occupancy Network	DRIVE Thor GPU	256x256	12 ms	2000

17.4.5. Integration von Bosch-Sensoren mit NVIDIA DRIVE Thor

Die Integration von Bosch-Sensoren mit NVIDIA DRIVE Thor ermöglicht hochperformante Perzeption:

- **Bosch 8MP Kamera**: Direkte Anbindung an DRIVE Thor über Ethernet 2.5G
- **Bosch Radar**: Integration über CAN-FD oder Ethernet
- **Bosch LiDAR**: Direkte Anbindung über Ethernet 10G
- **Sensorfusion**: Zentrale Sensorfusion auf DRIVE Thor GPU
- **Performance**: Echtzeit-Verarbeitung von bis zu 12 Kameras gleichzeitig

17.5. Erweiterte KI-Integration-Aspekte

Dieser Abschnitt beschreibt erweiterte Aspekte der KI-Integration.

17.5.1. Training und Deployment

KI-Modelle müssen trainiert und deployed werden:

- **Training**: Training in der Cloud mit großen Datensätzen
- **Optimierung**: Quantisierung, Pruning, Distillation für Edge-Deployment
- **Deployment**: Deployment auf Edge-Geräten (NPU, GPU)
- **Updates**: OTA-Updates für KI-Modelle

17.5.2. Modell-Versionierung

KI-Modelle müssen versioniert werden:

- **Versionierung**: Semantic Versioning für KI-Modelle
- **A/B-Testing**: A/B-Testing verschiedener Modell-Versionen
- **Rollback**: Rollback-Mechanismen bei Problemen
- **Monitoring**: Monitoring der Modell-Performance

17.5.3. Federated Learning

Federated Learning ermöglicht Training auf Edge-Geräten:

- **Lokales Training**: Training auf einzelnen Fahrzeugen
- **Aggregation**: Aggregation der Modelle in der Cloud
- **Privacy**: Datenschutz durch lokales Training
- **Efficiency**: Effizienz durch verteiltes Training

17.6. Zusammenfassung

Dieses Kapitel hat die Integration von KI/ML-Modellen in E/E-Architekturen beschrieben. Die wichtigsten Aspekte umfassen:

- **Modellierung**: KI-Modelle als spezielle SWCs in PREEvision
- **Simulation**: Spezielle Ansätze für die Simulation von KI-Modellen
- **Edge-Cloud-Hybrid**: Kombination von Edge- und Cloud-AI
- **Performance-Modellierung**: Detaillierte Modellierung der KI-Performance
- **Training und Deployment**: Methoden für Training und Deployment von KI-Modellen

Die Integration von KI-Modellen stellt neue Anforderungen an die Architektur und die Simulation, die durch die entwickelte Methodik adressiert werden.

17.7. Erweiterte KI-Modell-Beispiele

Dieser Abschnitt präsentiert detaillierte Beispiele für verschiedene KI-Modelle in E/E-Architekturen.

17.7.1. Beispiel: Objekterkennung mit YOLOv8

Modell-Spezifikation

Tabelle 17.2.: YOLOv8 Modell-Spezifikation

Parameter	Wert	Einheit	Anmerkung
Input-Größe	640x640	Pixel	RGB
Modell-Größe	22	MB	FP32
Parameter	11.2	M	–
FLOPS	28.5	G	pro Inferenz
Inferenz-Zeit (NPU)	8	ms	NVIDIA Orin
Inferenz-Zeit (GPU)	15	ms	NVIDIA Orin
Inferenz-Zeit (CPU)	120	ms	ARM Cortex-A78

Inferenz-Pipeline

Die Inferenz-Pipeline für YOLOv8:

1. **Preprocessing** (2 ms):
   - Bild-Resizing auf 640x640
   - Normalisierung (0-1)
   - Tensor-Konvertierung
2. **Inferenz** (8 ms auf NPU):
   - Feature-Extraction (Backbone)
   - Feature-Pyramid (Neck)
   - Detection-Head
3. **Postprocessing** (3 ms):
   - Non-Maximum Suppression (NMS)
   - Confidence-Filtering
   - Bounding-Box-Konvertierung

Energieverbrauch

Der Energieverbrauch für YOLOv8-Inferenz:

$$E_{YOLOv8} = P_{NPU} \times T_{inference} = 15W \times 0.008s = 0.12J \tag{17.3}$$

17.7.2. Beispiel: Trajektorien-Prädiktion mit Transformer

Modell-Spezifikation

Tabelle 17.3.: Transformer Modell-Spezifikation

Parameter	Wert	Einheit	Anmerkung
Input-Tokens	512	–	–
Hidden-Size	256	–	–
Layers	6	–	–
Attention-Heads	8	–	–
Modell-Größe	45	MB	FP32
Parameter	12.5	M	–
FLOPS	15.2	G	pro Inferenz
Inferenz-Zeit (GPU)	25	ms	NVIDIA Orin

Attention-Mechanismus

Der Attention-Mechanismus berechnet:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{17.4}$$

wobei:

- Q: Query-Matrix
- K: Key-Matrix
- V: Value-Matrix
- d_k: Dimension der Keys

17.8. Erweiterte KI-Performance-Analyse

Dieser Abschnitt beschreibt erweiterte Methoden zur Analyse der KI-Performance.

17.8.1. Latency-Analyse

Die Latenz für KI-Inferenz setzt sich zusammen aus:

$$L_{total} = L_{preprocessing} + L_{inference} + L_{postprocessing} + L_{communication} \tag{17.5}$$

17.8.2. Throughput-Analyse

Der Throughput für KI-Inferenz:

$$Throughput = \frac{1}{L_{total}} = \frac{FPS}{1} \tag{17.6}$$

17.8.3. Accuracy-vs-Latency-Trade-off

Es gibt einen Trade-off zwischen Genauigkeit und Latenz:

- **Höhere Genauigkeit**: Größere Modelle, längere Inferenz-Zeit
- **Niedrigere Latenz**: Kleinere Modelle, kürzere Inferenz-Zeit
- **Optimierung**: Quantisierung, Pruning, Distillation

17.9. Erweiterte KI-Deployment-Strategien

Dieser Abschnitt beschreibt erweiterte Strategien für das Deployment von KI-Modellen.

17.9.1. Model-Compression

Model-Compression reduziert Modell-Größe und Inferenz-Zeit:

- **Quantisierung**: FP32 → INT8 (4x kleiner, 2-4x schneller)
- **Pruning**: Entfernung unwichtiger Neuronen (50-90% Reduktion)
- **Distillation**: Wissen von großem zu kleinem Modell übertragen
- **Knowledge-Distillation**: Training eines kleineren Modells mit großem Modell als Teacher

17.9.2. Multi-Model-Deployment

Verschiedene Modelle für verschiedene Anwendungsfälle:

Tabelle 17.4.: Multi-Model-Deployment-Strategie

Anwendung	Modell	Hardware	Latenz	Genauigkeit
Echtzeit-Perzeption	YOLOv8-nano	NPU	5 ms	85%
High-Accuracy-Perzeption	YOLOv8-large	GPU	20 ms	95%
Trajektorien-Prädiktion	Transformer-small	GPU	15 ms	88%

17.9.3. Dynamic-Model-Switching

Dynamisches Wechseln zwischen Modellen basierend auf Kontext:

- **Stadtverkehr**: Kleines, schnelles Modell
- **Autobahn**: Größeres, genaueres Modell
- **Schlechte Sicht**: Robustes Modell mit höherer Latenz

18. Cybersecurity in E/E-Architekturen

Dieses Kapitel beschreibt die Modellierung und Simulation von Cybersecurity-Aspekten in E/E-Architekturen. Cybersecurity wird zunehmend wichtig in vernetzten Fahrzeugen und muss bereits in der Architektur-Phase berücksichtigt werden.

18.1. Cybersecurity-Anforderungen

Moderne E/E-Architekturen müssen verschiedene Cybersecurity-Anforderungen erfüllen:

- **Vertraulichkeit**: Schutz vor unberechtigtem Zugriff auf Daten
- **Integrität**: Schutz vor Manipulation von Daten
- **Verfügbarkeit**: Schutz vor Denial-of-Service-Angriffen
- **Authentifizierung**: Verifizierung der Identität von Kommunikationspartnern
- **Autorisierung**: Kontrolle des Zugriffs auf Ressourcen
- **Non-Repudiation**: Nachweisbarkeit von Aktionen

18.2. Modellierung von Cybersecurity

Cybersecurity-Aspekte werden in PREEvision wie folgt modelliert:

18.2.1. Sicherheitszonen

Sicherheitszonen definieren Bereiche mit unterschiedlichen Sicherheitsanforderungen:

- **Trusted Zone**: Höchste Sicherheit (z. B. sicherheitskritische Funktionen)
- **Secure Zone**: Hohe Sicherheit (z. B. Infotainment)
- **Untrusted Zone**: Niedrige Sicherheit (z. B. externe Schnittstellen)

18.2.2. Sicherheitsmechanismen

Verschiedene Sicherheitsmechanismen werden modelliert:

- **Verschlüsselung**: AES, RSA, ECC
- **Digitale Signaturen**: RSA, ECDSA
- **Hash-Funktionen**: SHA-256, SHA-3
- **Secure Boot**: Verifizierung beim Start
- **Secure Update**: Sichere OTA-Updates
- **Intrusion Detection**: Erkennung von Angriffen

18.3. Simulation von Cybersecurity

Die Simulation von Cybersecurity-Aspekten ermöglicht die Bewertung von Sicherheitsmaßnahmen:

18.3.1. Angriffs-Szenarien

Verschiedene Angriffs-Szenarien werden simuliert:

- **Man-in-the-Middle**: Abfangen und Manipulation von Kommunikation
- **Denial-of-Service**: Überlastung von Systemen
- **Replay-Angriffe**: Wiederverwendung von Nachrichten
- **Privilege Escalation**: Erhöhung von Berechtigungen

18.3.2. Sicherheits-Performance

Die Performance von Sicherheitsmechanismen wird modelliert:

18.4. Zusammenfassung

Dieses Kapitel hat die Modellierung und Simulation von Cybersecurity-Aspekten in E/E-Architekturen beschrieben. Die wichtigsten Aspekte umfassen:

- **Anforderungen**: Vertraulichkeit, Integrität, Verfügbarkeit, etc.

Tabelle 18.1.: Performance von Sicherheitsmechanismen

Mechanismus	Latenz	CPU-Last	Anwendung
AES-128 (SW)	0.5 ms	5%	Datenverschlüsselung
AES-128 (HW)	0.1 ms	1%	Datenverschlüsselung
RSA-2048	10 ms	15%	Signatur-Verifizierung
ECDSA-256	2 ms	8%	Signatur-Verifizierung
SHA-256	0.2 ms	2%	Hash-Berechnung

- **Modellierung**: Sicherheitszonen, Sicherheitsmechanismen
- **Simulation**: Angriffs-Szenarien, Sicherheits-Performance

Cybersecurity muss bereits in der Architektur-Phase berücksichtigt werden, um sicherzustellen, dass die Architektur sicher ist.

18.5. Erweiterte Cybersecurity-Modellierung

Dieser Abschnitt beschreibt erweiterte Aspekte der Cybersecurity-Modellierung.

18.5.1. Threat-Modellierung

Threat-Modellierung identifiziert potenzielle Bedrohungen:

- **STRIDE**: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege
- **Attack-Trees**: Hierarchische Darstellung von Angriffen
- **Attack-Surfaces**: Identifikation von Angriffsflächen
- **Risk-Assessment**: Bewertung von Risiken

18.5.2. Security-by-Design

Security-by-Design integriert Sicherheit von Anfang an:

- **Secure-Architecture**: Architektur mit Sicherheit im Fokus
- **Defense-in-Depth**: Mehrschichtige Sicherheitsmaßnahmen
- **Least-Privilege**: Minimale Berechtigungen
- **Secure-Communication**: Verschlüsselte Kommunikation

18.6. Erweiterte Cybersecurity-Mechanismen

Dieser Abschnitt beschreibt erweiterte Cybersecurity-Mechanismen.

18.6.1. Secure-Boot

Secure-Boot verifiziert die Integrität beim Start:

1. **ROM-Code**: Verifiziert Bootloader-Signatur
2. **Bootloader**: Verifiziert OS-Signatur
3. **OS**: Verifiziert Anwendungs-Signaturen
4. **Chain-of-Trust**: Vertrauenskette von Hardware zu Software

18.6.2. Secure-Update

Secure-Update ermöglicht sichere OTA-Updates:

- **Signatur-Verifizierung**: Verifizierung der Update-Signatur
- **Rollback-Mechanismus**: Zurücksetzen bei Problemen
- **Incremental-Updates**: Nur Änderungen übertragen
- **Encrypted-Transport**: Verschlüsselte Übertragung

18.6.3. Intrusion-Detection-System (IDS)

IDS erkennt Angriffe in Echtzeit:

- **Signature-Based**: Erkennung bekannter Angriffsmuster
- **Anomaly-Based**: Erkennung von Anomalien
- **Behavior-Based**: Erkennung von Verhaltensänderungen
- **Machine-Learning**: ML-basierte Erkennung

18.7. Erweiterte Cybersecurity-Simulation

Dieser Abschnitt beschreibt erweiterte Methoden zur Simulation von Cybersecurity.

Tabelle 18.2.: Angriffs-Szenarien

Szenario	Typ	Impact	Detection
Man-in-the-Middle	Passiv	Hoch	Schwierig
Denial-of-Service	Aktiv	Mittel	Einfach
Replay-Angriff	Aktiv	Mittel	Mittel
Privilege Escalation	Aktiv	Sehr hoch	Schwierig

18.7.1. Attack-Scenario-Modellierung

Verschiedene Angriffs-Szenarien werden modelliert:

18.7.2. Security-Performance-Analyse

Die Performance von Sicherheitsmechanismen wird analysiert:

$$L_{security} = L_{encryption} + L_{signature} + L_{verification} + L_{overhead} \tag{18.1}$$

wobei:

- $L_{encryption}$: Verschlüsselungs-Latenz
- $L_{signature}$: Signatur-Latenz
- $L_{verification}$: Verifizierungs-Latenz
- $L_{overhead}$: Overhead-Latenz

18.7.3. Security-vs-Performance-Trade-off

Es gibt einen Trade-off zwischen Sicherheit und Performance:

- **Höhere Sicherheit**: Mehr Verschlüsselung, höhere Latenz
- **Niedrigere Latenz**: Weniger Verschlüsselung, niedrigere Sicherheit
- **Optimierung**: Hardware-Beschleunigung, selektive Verschlüsselung

18.8. Erweiterte Cybersecurity-Standards

Dieser Abschnitt beschreibt relevante Cybersecurity-Standards.

18.8.1. ISO 21434

ISO 21434 definiert Anforderungen für Cybersecurity:

- **CSMS**: Cybersecurity Management System
- **Threat-Analysis**: Systematische Bedrohungsanalyse
- **Risk-Assessment**: Risikobewertung und -behandlung
- **Security-by-Design**: Sicherheit durch Design

18.8.2. UN R155

UN Regulation 155 definiert Anforderungen für Cybersecurity:

- **CSMS**: Cybersecurity Management System
- **Type-Approval**: Typgenehmigung für Cybersecurity
- **Incident-Response**: Reaktion auf Sicherheitsvorfälle

19. Energiemanagement in E/E-Architekturen

Dieses Kapitel beschreibt die Modellierung und Optimierung des Energieverbrauchs in E/E-Architekturen. Energiemanagement ist besonders wichtig für elektrische Fahrzeuge, wo der Energieverbrauch direkt die Reichweite beeinflusst.

19.1. Energieverbrauch in E/E-Architekturen

Der Energieverbrauch in E/E-Architekturen setzt sich zusammen aus:

- **Rechenknoten**: CPU, GPU, NPU, Memory
- **Kommunikation**: Netzwerk-Interfaces, Switches
- **Sensoren**: Kameras, Radar, LiDAR, etc.
- **Aktoren**: EPS, EHB, E-Motor, etc.
- **Auxiliary**: Beleuchtung, Klima, etc.

19.1.1. Power-States

Rechenknoten können verschiedene Power-States haben:

- **Active**: Volle Leistung, alle Funktionen aktiv
- **Idle**: Niedrige Leistung, keine aktiven Tasks
- **Sleep**: Sehr niedrige Leistung, minimale Funktionen
- **Off**: Keine Energieversorgung

19.2. Energie-Optimierung

Energie-Optimierung kann auf verschiedenen Ebenen erfolgen:

19.2.1. Hardware-Optimierung

- **Dynamic Voltage and Frequency Scaling (DVFS)**: Anpassung von Spannung und Frequenz
- **Power-Gating**: Abschalten nicht genutzter Komponenten
- **Clock-Gating**: Abschalten nicht genutzter Clock-Domains
- **Low-Power-Modi**: Nutzung von Low-Power-Modi

19.2.2. Software-Optimierung

- **Task-Scheduling**: Energie-effizientes Scheduling
- **Load-Balancing**: Gleichmäßige Lastverteilung
- **Adaptive Quality**: Anpassung der Qualität basierend auf verfügbarer Energie
- **Predictive Shutdown**: Vorhersagbares Abschalten nicht benötigter Komponenten

19.3. Modellierung

Der Energieverbrauch wird in der Simulation modelliert:

$$E_{total} = \sum_{i=1}^{N} (P_i \times t_i) \tag{19.1}$$

wobei:

- P_i: Leistung in Zustand i
- t_i: Zeit in Zustand i
- N: Anzahl der Zustände

19.4. Zusammenfassung

Dieses Kapitel hat die Modellierung und Optimierung des Energieverbrauchs in E/E-Architekturen beschrieben. Energiemanagement ist entscheidend für die Effizienz elektrischer Fahrzeuge und muss bereits in der Architektur-Phase berücksichtigt werden.

19.5. Erweiterte Energie-Modellierung

Dieser Abschnitt beschreibt erweiterte Methoden zur Modellierung des Energieverbrauchs.

19.5.1. Detaillierte Power-State-Modellierung

Rechenknoten haben verschiedene Power-States mit unterschiedlichen Leistungen:

Tabelle 19.1.: Power-States für Central Compute

State	CPU	GPU	NPU	Gesamt
Active	25W	30W	15W	70W
Idle	5W	2W	1W	8W
Sleep	0.5W	0.1W	0.1W	0.7W
Off	0W	0W	0W	0W

19.5.2. DVFS-Modellierung

Dynamic Voltage and Frequency Scaling (DVFS) ermöglicht Energie-Optimierung:

$$P_{DVFS}(f, V) = C_{eff} \times V^2 \times f + P_{static} \tag{19.2}$$

wobei:

- f: Frequenz
- V: Spannung
- C_{eff}: Effektive Kapazität
- P_{static}: Statische Leistung

Die optimale Frequenz für minimale Energie bei gegebener Last:

$$f_{opt} = \sqrt{\frac{L}{C_{eff} \times V^2}} \tag{19.3}$$

19.5.3. Temperaturabhängigkeit

Der Energieverbrauch hängt von der Temperatur ab:

$$P_{total}(T) = P_{dyn} + P_{leak}(T) + P_{static} \tag{19.4}$$

wobei der Leckstrom temperaturabhängig ist:

$$I_{leak}(T) = I_0 \times e^{\frac{E_a}{k_B T}} \tag{19.5}$$

19.6. Erweiterte Energie-Optimierungs-Strategien

Dieser Abschnitt beschreibt erweiterte Strategien zur Energie-Optimierung.

19.6.1. Task-Scheduling für Energie-Effizienz

Energie-effizientes Task-Scheduling:

- **Consolidation**: Zusammenfassung von Tasks auf wenige Cores
- **Migration**: Migration von Tasks zwischen Cores
- **Frequency-Scaling**: Anpassung der Frequenz basierend auf Last
- **Power-Gating**: Abschalten nicht genutzter Cores

19.6.2. Adaptive-Quality

Anpassung der Qualität basierend auf verfügbarer Energie:

Tabelle 19.2.: Adaptive-Quality-Strategien

Energie-Level	Qualität	Latenz	Genauigkeit	Energie
Hoch	Hoch	Niedrig	Hoch	Hoch
Mittel	Mittel	Mittel	Mittel	Mittel
Niedrig	Niedrig	Hoch	Niedrig	Niedrig

19.6.3. Predictive-Power-Management

Vorhersagbares Power-Management:

- **Load-Prediction**: Vorhersage der zukünftigen Last
- **Preemptive-Scaling**: Proaktive Skalierung
- **Energy-Budgeting**: Energie-Budget-Verwaltung
- **Optimization**: Optimierung basierend auf Vorhersagen

19.7. Erweiterte Energie-Analyse

Dieser Abschnitt beschreibt erweiterte Methoden zur Analyse des Energieverbrauchs.

19.7.1. Energie-Profil-Analyse

Energie-Profile für verschiedene Szenarien:

Tabelle 19.3.: Energie-Profile für verschiedene Szenarien

Szenario	Energie (Wh/km)	Komponente	Anteil
Stadtverkehr	25	E-Motor	60%
Stadtverkehr	25	E/E-System	15%
Stadtverkehr	25	Klima	20%
Stadtverkehr	25	Sonstiges	5%
Autobahn	30	E-Motor	70%
Autobahn	30	E/E-System	10%
Autobahn	30	Klima	15%
Autobahn	30	Sonstiges	5%

19.7.2. Energie-Optimierungs-Potenzial

Potenzial für Energie-Optimierung:

- **Hardware**: 10-20% durch effizientere Hardware
- **Software**: 15-25% durch optimierte Software
- **Architektur**: 20-30% durch optimierte Architektur
- **Gesamt**: 30-50% durch kombinierte Optimierungen

19.8. Erweiterte Energie-Simulation

Dieser Abschnitt beschreibt erweiterte Methoden zur Simulation des Energieverbrauchs.

19.8.1. Multi-Level-Energie-Simulation

Energie-Simulation auf verschiedenen Ebenen:

- **System-Level**: Gesamtsystem-Energieverbrauch
- **Component-Level**: Komponenten-Energieverbrauch

- **Task-Level**: Task-Energieverbrauch
- **Instruction-Level**: Instruction-Energieverbrauch

19.8.2. Energie-Validierung

Validierung der Energie-Simulation:

- **Benchmarking**: Vergleich mit realen Messungen
- **Sensitivity-Analysis**: Analyse der Sensitivität
- **Uncertainty-Quantification**: Quantifizierung der Unsicherheit

20. Standards und Compliance

Dieses Kapitel beschreibt die relevanten Standards und Compliance-Anforderungen für E/E-Architekturen und wie diese in der Modellierung und Simulation berücksichtigt werden.

20.1. ISO 26262 - Funktionale Sicherheit

ISO 26262 ist der Standard für funktionale Sicherheit in Kraftfahrzeugen und definiert Anforderungen für die Entwicklung sicherheitskritischer Systeme.

20.1.1. ASIL-Level

ASIL (Automotive Safety Integrity Level) definiert vier Sicherheitsstufen:

- **ASIL A**: Niedrigste Sicherheitsstufe, geringe Anforderungen
- **ASIL B**: Mittlere Sicherheitsstufe, moderate Anforderungen
- **ASIL C**: Hohe Sicherheitsstufe, hohe Anforderungen
- **ASIL D**: Höchste Sicherheitsstufe, höchste Anforderungen

20.1.2. Modellierung in PREEvision

ASIL-Level werden in PREEvision wie folgt modelliert:

- **ECU-ASIL**: ASIL-Level für ECUs
- **SWC-ASIL**: ASIL-Level für Software-Komponenten
- **Signal-ASIL**: ASIL-Level für Signale
- **Partitionierung**: ASIL-Isolation durch Partitionierung

20.1.3. Simulation

Die Simulation berücksichtigt ASIL-Anforderungen:

- **Verfügbarkeit**: ASIL-spezifische Verfügbarkeitsanforderungen
- **Redundanz**: ASIL-spezifische Redundanzanforderungen
- **Fehlerbehandlung**: ASIL-spezifische Fehlerbehandlung

20.2. AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) ist ein Standard für Automotive-Software-Architekturen.

20.2.1. AUTOSAR Classic

AUTOSAR Classic für eingebettete Systeme:

- **SWCs**: Software Components mit Ports und Interfaces
- **RTE**: Runtime Environment für Kommunikation
- **BSW**: Basic Software Layer
- **ECU Configuration**: ECU-spezifische Konfiguration

20.2.2. AUTOSAR Adaptive

AUTOSAR Adaptive für hochperformante Systeme:

- **Adaptive Applications**: C++-basierte Anwendungen
- **ARA**: Adaptive AUTOSAR Runtime
- **Service-Orientation**: SOME/IP, DDS
- **Execution Management**: Lifecycle-Management

20.3. TSN-Standards

Time-Sensitive Networking (TSN) Standards für deterministische Kommunikation:

20.3.1. IEEE 802.1 Standards

- **IEEE 802.1AS**: gPTP (generalized Precision Time Protocol)
- **IEEE 802.1Qbv**: Time-Aware Shaping
- **IEEE 802.1Qbu**: Frame Preemption
- **IEEE 802.1Qcc**: TSN Configuration

20.3.2. Modellierung

TSN-Standards werden in PREEvision modelliert:

- **Gate-Schedules**: Konfiguration von Gate-Schedules
- **Time-Synchronisation**: gPTP-Konfiguration
- **Traffic-Shaping**: Traffic-Shaping-Parameter
- **Frame-Preemption**: Preemption-Konfiguration

20.4. UN R155 - Cybersecurity

UN Regulation 155 definiert Anforderungen für Cybersecurity in Fahrzeugen:

- **CSMS**: Cybersecurity Management System
- **Threat-Analysis**: Bedrohungsanalyse
- **Risk-Assessment**: Risikobewertung
- **Security-by-Design**: Sicherheit durch Design

20.5. Erweiterte Standards-Details

Dieser Abschnitt beschreibt erweiterte Details zu relevanten Standards.

20.5.1. ISO 21434 - Cybersecurity

ISO 21434 definiert Anforderungen für Cybersecurity in Fahrzeugen:

- **CSMS**: Cybersecurity Management System
- **Threat-Analysis**: Systematische Bedrohungsanalyse

- **Risk-Assessment**: Risikobewertung und -behandlung
- **Security-by-Design**: Sicherheit durch Design
- **Incident-Response**: Reaktion auf Sicherheitsvorfälle

20.5.2. UN R156 - Software-Updates

UN Regulation 156 definiert Anforderungen für Software-Updates:

- **SUMS**: Software Update Management System
- **Update-Verifizierung**: Verifizierung von Updates
- **Rollback-Mechanismen**: Mechanismen zum Zurücksetzen
- **Update-Dokumentation**: Dokumentation von Updates

20.5.3. ISO 14229 - UDS

ISO 14229 definiert Unified Diagnostic Services (UDS):

- **Diagnostic-Services**: Verschiedene Diagnose-Services
- **Session-Control**: Kontrolle von Diagnose-Sessions
- **Data-Transfer**: Übertragung von Diagnose-Daten
- **Security-Access**: Zugriffskontrolle für Diagnose

20.6. Zusammenfassung

Dieses Kapitel hat relevante Standards und Compliance-Anforderungen beschrieben, die bei der Entwicklung von E/E-Architekturen berücksichtigt werden müssen. Die Modellierung und Simulation müssen diese Anforderungen unterstützen, um sicherzustellen, dass die Architektur konform ist.

21. Vergleich verschiedener Architektur-Paradigmen

Dieses Kapitel vergleicht verschiedene Architektur-Paradigmen für E/E-Architekturen und analysiert deren Vor- und Nachteile im Kontext der Simulation.

21.1. Architektur-Paradigmen

Verschiedene Architektur-Paradigmen werden in modernen Fahrzeugen verwendet:

21.1.1. Distributed Architecture

Traditionelle verteilte Architektur mit vielen ECUs:

- **Struktur**: Viele spezialisierte ECUs, jeweils für spezifische Funktionen
- **Kommunikation**: CAN, LIN, FlexRay
- **Vorteile**: Bewährt, einfach, kostengünstig
- **Nachteile**: Hohe Komplexität, viele Kabel, schwierige Software-Updates

21.1.2. Zonale Architecture

Moderne zonale Architektur mit zentralen Rechenknoten:

- **Struktur**: Wenige zentrale Rechenknoten, Zonen-Controller als Gateways
- **Kommunikation**: TSN-Ethernet-Backbone, CAN/LIN in Zonen
- **Vorteile**: Skalierbar, software-definiert, einfache Updates
- **Nachteile**: Komplexere Software, höhere Anforderungen an Rechenleistung

21.1.3. Hybrid Architecture

Kombination aus distributiver und zonaler Architektur:

- **Struktur**: Zentrale Rechenknoten für neue Funktionen, Legacy-ECUs für bestehende Funktionen
- **Kommunikation**: TSN-Ethernet für neue Funktionen, CAN/LIN für Legacy
- **Vorteile**: Migration von Legacy-Systemen, schrittweise Modernisierung
- **Nachteile**: Komplexität durch zwei Architektur-Paradigmen

21.2. Vergleichsanalyse

Verschiedene Kriterien werden für den Vergleich verwendet:

21.2.1. Komplexität

Tabelle 21.1.: Vergleich: Komplexität

Kriterium	Distributed	Zonal	Hybrid
Anzahl ECUs	Hoch	Niedrig	Mittel
Kabel-Komplexität	Hoch	Niedrig	Mittel
Software-Komplexität	Niedrig	Hoch	Mittel
Netzwerk-Komplexität	Niedrig	Hoch	Mittel

21.2.2. Kosten

Tabelle 21.2.: Vergleich: Kosten

Kriterium	Distributed	Zonal	Hybrid
Hardware-Kosten	Mittel	Niedrig	Mittel
Software-Kosten	Niedrig	Hoch	Mittel
Wartungskosten	Hoch	Niedrig	Mittel
Entwicklungskosten	Niedrig	Hoch	Mittel

Tabelle 21.3.: Vergleich: Performance

Kriterium	Distributed	Zonal	Hybrid
E2E-Latenz	Mittel	Niedrig	Mittel
Skalierbarkeit	Niedrig	Hoch	Mittel
Energieeffizienz	Niedrig	Hoch	Mittel

21.2.3. Performance

21.3. Erweiterte Vergleichs-Analysen

Dieser Abschnitt präsentiert erweiterte Analysen für verschiedene Aspekte.

21.3.1. Energieeffizienz-Vergleich

Tabelle 21.4.: Vergleich: Energieeffizienz

Architektur-Typ	Energieverbrauch	Optimierungs-Potenzial	Anmerkung
Distributed	Hoch	Niedrig	Viele ECUs, hoher Overhead
Zonal	Niedrig	Hoch	Zentrale Optimierung möglich
Hybrid	Mittel	Mittel	Abhängig von Legacy-Anteil

21.3.2. Wartbarkeits-Vergleich

Tabelle 21.5.: Vergleich: Wartbarkeit

Architektur-Typ	Software-Updates	Hardware-Updates	Diagnose
Distributed	Schwierig	Einfach	Komplex
Zonal	Einfach	Schwierig	Einfach
Hybrid	Mittel	Mittel	Mittel

21.4. Zusammenfassung

Dieses Kapitel hat verschiedene Architektur-Paradigmen verglichen. Die zonale Architektur bietet Vorteile in Bezug auf Skalierbarkeit, Software-Updates und Energieeffizienz, während die distributed Architektur bewährt und kostengünstig ist. Die Hybrid-Architektur ermöglicht eine schrittweise Migration.

22. Bosch-Sensorik und NVIDIA DRIVE Thor Integration

Dieses Kapitel beschreibt detailliert die Integration von Bosch-Sensorik und NVIDIA DRIVE Thor in moderne E/E-Architekturen. Die Kombination dieser Technologien ermöglicht hochperformante, zuverlässige Systeme für automatisiertes Fahren.

22.1. Bosch-Sensorik Portfolio

Bosch bietet ein umfassendes Portfolio an Sensoren für automatisiertes Fahren, das kontinuierlich weiterentwickelt wird.

22.1.1. Bosch 8MP Multifunktionskamera

Die Bosch 8MP Multifunktionskamera ist die neueste Generation von Kameras für automatisiertes Fahren:

Technische Spezifikationen

Tabelle 22.1.: Technische Spezifikationen: Bosch 8MP Multifunktionskamera

Parameter	Wert
Auflösung	3840 × 2160 Pixel (8 Megapixel)
Horizontales Sichtfeld	120 Grad
Vertikales Sichtfeld	65 Grad
Erkennungsreichweite	Bis zu 300 Meter
Framerate	30 fps (33.3 ms Periodizität)
Interface	Ethernet 2.5 Gbps
Kompression	H.264/H.265, optional RAW
ASIL-Level	ASIL-B (ADAS), ASIL-D möglich
Power-Verbrauch	8 W typisch, 12 W maximal
Betriebstemperatur	-40°C bis +85°C
Serienproduktion	Geplant für 2026

Funktionen

Die Bosch 8MP Multifunktionskamera unterstützt verschiedene ADAS-Funktionen:

- **Adaptive Geschwindigkeits- und Abstandsregelung (ACC)**:
  - Automatische Anpassung der Geschwindigkeit an vorausfahrende Fahrzeuge
  - Abstandsmessung mit hoher Präzision durch 8MP Auflösung
  - Erkennung bis zu 300 m Reichweite
- **Notbremsungen innerhalb der eigenen Spur (AEB)**:
  - Automatische Notbremsung bei erkannten Hindernissen
  - Früherkennung durch hohe Auflösung und lange Reichweite
  - ASIL-D für sicherheitskritische Anwendungen
- **Spurhalten in städtischen Gebieten (LKA)**:
  - Automatisches Spurhalten durch Markierungserkennung
  - Breites Sichtfeld (120°) für enge Kurven
  - Funktion auch bei schlechten Sichtverhältnissen
- **Erkennung und Anhalten an roten Ampeln (TSR)**:
  - Automatische Erkennung von Verkehrszeichen und Ampeln
  - Präzise Erkennung durch hohe Auflösung
  - Integration in Navigationssysteme

Integration in E/E-Architekturen

Die Bosch 8MP Multifunktionskamera wird in E/E-Architekturen wie folgt integriert:

- **Netzwerk-Integration**:
  - Direkte Anbindung an Zonen-Controller oder Central Compute über Ethernet 2.5G
  - TSN-Unterstützung für deterministische Übertragung
  - Priorität P6 für sicherheitskritische Daten
- **Software-Integration**:
  - Standardisierte APIs für Kamera-Zugriff

  - Unterstützung für verschiedene Bildverarbeitungsalgorithmen
  - Integration in Sensorfusion-Systeme

- **Power-Management**:
  - Niedriger Power-Verbrauch für Energieeffizienz
  - Wake-up-Mechanismen für schnelle Aktivierung
  - Power-State-Management für optimale Energieverteilung

22.1.2. Bosch Radar-Sensoren

Bosch bietet ein umfassendes Portfolio an Radar-Sensoren für verschiedene Anwendungen:

Long-Range-Radar

Tabelle 22.2.: Technische Spezifikationen: Bosch Long-Range-Radar

Parameter	Wert
Reichweite	Bis zu 250 Meter
Geschwindigkeitsbereich	-200 km/h bis +200 km/h
Winkelbereich	±60 Grad
Update-Rate	20 Hz (50 ms)
Auflösung	0.5 m (Radial), 1.5° (Azimut)
Interface	CAN-FD 2 Mbps oder Ethernet
ASIL-Level	ASIL-B bis ASIL-D
Power-Verbrauch	3 W typisch, 5 W maximal

Mid-Range-Radar

Tabelle 22.3.: Technische Spezifikationen: Bosch Mid-Range-Radar

Parameter	Wert
Reichweite	Bis zu 160 Meter
Geschwindigkeitsbereich	-150 km/h bis +150 km/h
Winkelbereich	±75 Grad
Update-Rate	25 Hz (40 ms)
Auflösung	0.3 m (Radial), 1.0° (Azimut)
Interface	CAN-FD 2 Mbps oder Ethernet
ASIL-Level	ASIL-B
Power-Verbrauch	2.5 W typisch, 4 W maximal

4D-Imaging-Radar

Das 4D-Imaging-Radar bietet zusätzlich zur 3D-Position auch Geschwindigkeitsinformationen:

- **Reichweite**: Bis zu 200 Meter
- **Auflösung**: Hohe Auflösung für präzise Objekterkennung
- **Geschwindigkeit**: Präzise Geschwindigkeitsmessung
- **Anwendung**: Für hochautomatisiertes Fahren (L4/L5)

22.1.3. Bosch LiDAR-Sensoren

Bosch entwickelt High-Resolution-LiDAR-Sensoren für hochautomatisiertes Fahren:

Technische Spezifikationen

Tabelle 22.4.: Technische Spezifikationen: Bosch High-Resolution-LiDAR

Parameter	Wert
Layer	Bis zu 64 Layer
Reichweite	Bis zu 200 Meter
Punktdichte	Hoch (für präzise Objekterkennung)
Winkelbereich	360° (Rotating) oder 120° (Solid-State)
Update-Rate	10-20 Hz
Interface	Ethernet 10 Gbps
ASIL-Level	ASIL-B
Power-Verbrauch	15 W typisch, 25 W maximal

22.2. NVIDIA DRIVE Thor

NVIDIA DRIVE Thor ist die neueste Generation der NVIDIA DRIVE Plattform für autonomes Fahren.

22.2.1. Technische Spezifikationen

22.2.2. GPU-Architektur

Die NVIDIA DRIVE Thor GPU ist spezialisiert auf KI/ML-Workloads:

- **Tensor-Cores**: Spezialisierte Einheiten für Tensor-Operationen

Tabelle 22.5.: Technische Spezifikationen: NVIDIA DRIVE Thor

Parameter	Wert
GPU-Performance	2000 TOPS (Tera Operations Per Second)
CPU	NVIDIA Grace CPU (ARM Neoverse V2)
CPU-Kerne	12 Kerne @ 3.0 GHz
RAM	512 GB LPDDR5X
Storage	1 TB NVMe SSD
Ethernet-Ports	Multiple (2.5G, 10G)
CAN-FD-Ports	Multiple (2 Mbps)
ASIL-Level	ASIL-D zertifiziert
Power-Verbrauch	80 W typisch, 150 W maximal
Betriebstemperatur	-40°C bis +85°C
Software	NVIDIA DRIVE OS

- **RT-Cores**: Ray-Tracing-Cores für visuelle Anwendungen
- **Multi-Instance-GPU (MIG)**: Partitionierung für Multi-Domain-Isolation
- **Unified Memory**: Effiziente CPU-GPU-Zusammenarbeit

22.2.3. Software-Stack

NVIDIA DRIVE OS bietet einen vollständigen Software-Stack:

- **Betriebssystem**: Linux-basiert mit Echtzeit-Erweiterungen
- **Middleware**: AUTOSAR Adaptive, DDS, SOME/IP
- **KI-Frameworks**: TensorRT, CUDA, cuDNN
- **Entwicklungstools**: NVIDIA DRIVE SDK, Simulator

22.3. Integration von Bosch-Sensoren mit NVIDIA DRIVE Thor

Die Integration von Bosch-Sensoren mit NVIDIA DRIVE Thor ermöglicht hochperformante Perzeption:

22.3.1. Sensor-Integration

- **Bosch 8MP Kamera**:
  - Direkte Anbindung über Ethernet 2.5G

  - Unterstützung für bis zu 12 Kameras gleichzeitig
  - Echtzeit-Verarbeitung auf DRIVE Thor GPU

- **Bosch Radar**:
  - Integration über CAN-FD oder Ethernet
  - Unterstützung für bis zu 9 Radare
  - Sensorfusion auf DRIVE Thor

- **Bosch LiDAR**:
  - Direkte Anbindung über Ethernet 10G
  - Unterstützung für bis zu 4 LiDAR-Sensoren
  - Punktwolken-Verarbeitung auf DRIVE Thor GPU

22.3.2. Performance-Benchmarks

Die Kombination von Bosch-Sensoren mit NVIDIA DRIVE Thor bietet hervorragende Performance:

Tabelle 22.6.: Performance-Benchmarks: Bosch-Sensoren mit NVIDIA DRIVE Thor

Sensor	Modell	Inferenz-Zeit	GPU-Last
Bosch 8MP Kamera	YOLOv8	15 ms	0.975%
Bosch 8MP Kamera (8x)	YOLOv8	120 ms	7.8%
Bosch Radar	Radar-Fusion	5 ms	0.1%
Bosch LiDAR	Point-Cloud-Processing	20 ms	1.5%

22.4. Zusammenfassung

Dieses Kapitel hat die Integration von Bosch-Sensorik und NVIDIA DRIVE Thor detailliert beschrieben. Die Kombination dieser Technologien ermöglicht hochperformante, zuverlässige Systeme für automatisiertes Fahren mit exzellenter Perzeptions-Performance und niedrigen Latenzen.

23. VAN.APPVERSE – Die offene Mobilitäts-Microservice-Ökonomie

Dieses Kapitel beschreibt das Konzept der VAN.APPVERSE – einer offenen Mobilitäts-Microservice-Ökonomie für Vans. VAN.APPVERSE ermöglicht es, Fahrzeugfunktionen als Microservices zu exponieren und über einen App Store zu vertreiben, wodurch eine neue Ökonomie für Mobilitätsdienste entsteht. Dieses Konzept baut auf modernen Software-Defined-Vehicle-Architekturen auf und nutzt die zentrale Rechenplattform (z. B. NVIDIA DRIVE Thor) sowie die serviceorientierte Kommunikation für die Bereitstellung von Mobilitätsdiensten.

23.1. Konzept und Vision

VAN.APPVERSE ist eine Plattform, die es Dritten ermöglicht, Mobilitätsdienste als Microservices zu entwickeln und über einen App Store zu vertreiben. Diese Vision transformiert Fahrzeuge von geschlossenen Systemen zu offenen Plattformen, auf denen Innovationen von verschiedenen Entwicklern schnell und einfach bereitgestellt werden können.

23.1.1. Vision

Die Vision von VAN.APPVERSE umfasst:

- **Offene Plattform**: Fahrzeuge werden zu offenen Plattformen für Mobilitätsdienste
- **Microservice-Ökonomie**: Funktionen werden als Microservices bereitgestellt und vertrieben
- **Innovations-Ökosystem**: Entwickler können schnell neue Dienste entwickeln und bereitstellen

- **Monetarisierung**: Entwickler können ihre Dienste monetarisieren
- **Kundennutzen**: Kunden können ihre Fahrzeuge durch Apps personalisieren und erweitern

23.1.2. Architektur-Prinzipien

VAN.APPVERSE basiert auf folgenden Architektur-Prinzipien:

- **Service-orientierte Architektur**: Funktionen werden als Services exponiert
- **Microservices**: Kleine, unabhängige Services mit klaren Schnittstellen
- **API-First**: Alle Funktionen werden über APIs verfügbar gemacht
- **Sicherheit**: Sicherheit durch Design mit Sandboxing und Isolation
- **Skalierbarkeit**: Services können horizontal skaliert werden

23.2. API-Ansatz und Schnittstellen

VAN.APPVERSE bietet einen professionellen API-Ansatz für den Zugriff auf Fahrzeugfunktionen.

23.2.1. API-Kategorien

Die APIs werden in verschiedene Kategorien eingeteilt:

Identity & Entitlement

APIs für Identitätsverwaltung und Berechtigungen:

- **Authentication**: OAuth 2.0, OpenID Connect
- **Authorization**: Role-Based Access Control (RBAC), Attribute-Based Access Control (ABAC)
- **Entitlements**: Berechtigungsprüfung für API-Zugriffe
- **Developer Identity**: Entwickler-Registrierung und -Verwaltung

App Management

APIs für die Verwaltung von Apps:

- **App Installation**: Installation und Deinstallation von Apps
- **App Lifecycle**: Start, Stop, Pause, Resume von Apps
- **App Updates**: Over-the-Air-Updates für Apps
- **App Versioning**: Versionierung und Rollback von Apps

Vehicle Control (Logical)

APIs für logische Fahrzeugsteuerung:

- **Navigation**: Route-Planung, Navigation, Points of Interest
- **Climate Control**: Temperatursteuerung, Belüftung
- **Infotainment**: Medien-Wiedergabe, Radio, Streaming
- **Comfort Functions**: Sitzheizung, Beleuchtung, etc.

Vehicle Control (Safety-Critical)

APIs für sicherheitskritische Fahrzeugsteuerung (mit strikten Sicherheitsanforderungen):

- **Advanced Driver Assistance**: ADAS-Funktionen (mit Einschränkungen)
- **Vehicle Dynamics**: Fahrzeugdynamik-Funktionen (nur für autorisierte Apps)
- **Emergency Functions**: Notfall-Funktionen
- **Safety-Critical Controls**: Sicherheitskritische Steuerungen (mit ASIL-Anforderungen)

Sensor Streams

APIs für den Zugriff auf Sensordaten:

- **Camera Streams**: Zugriff auf Kamera-Datenströme (mit Privacy-Schutz)
- **Radar Data**: Zugriff auf Radar-Daten
- **LiDAR Data**: Zugriff auf LiDAR-Daten
- **GNSS/IMU**: Zugriff auf Positions- und Orientierungsdaten
- **Vehicle Sensors**: Geschwindigkeit, Beschleunigung, etc.

Diagnostics

APIs für Diagnose und Wartung:

- **Vehicle Health**: Fahrzeugzustand, Fehlercodes
- **Diagnostic Data**: Diagnosedaten für Wartung
- **Maintenance Scheduling**: Wartungsplanung
- **Remote Diagnostics**: Ferndiagnose

Power & Energy (Physical)

APIs für Energie-Management:

- **Battery Status**: Batteriezustand, Ladezustand
- **Charging Control**: Ladesteuerung
- **Energy Management**: Energiemanagement
- **V2G**: Vehicle-to-Grid-Funktionen

Docking & Physical Interfaces

APIs für physische Schnittstellen:

- **Docking**: Ankopplung von Geräten
- **USB/Serial**: Zugriff auf USB/Serial-Schnittstellen
- **GPIO**: Zugriff auf GPIO-Pins
- **Physical Interfaces**: Zugriff auf physische Schnittstellen

Edge Compute

APIs für Edge-Computing:

- **Compute Resources**: Zugriff auf Rechenressourcen (CPU, GPU, NPU)
- **AI/ML Inference**: KI/ML-Inferenz-Dienste
- **Data Processing**: Datenverarbeitung
- **Edge Storage**: Edge-Speicher

OTA Updates

APIs für Over-the-Air-Updates:

- **Update Management**: Verwaltung von Updates
- **Update Deployment**: Bereitstellung von Updates
- **Update Status**: Update-Status
- **Rollback**: Rollback von Updates

Events

APIs für Event-basierte Kommunikation:

- **Event Subscription**: Abonnierung von Events
- **Event Publishing**: Veröffentlichung von Events
- **Event Filtering**: Filterung von Events
- **Event History**: Event-Historie

Billing

APIs für Abrechnung:

- **Usage Tracking**: Nutzungsverfolgung
- **Billing**: Abrechnung
- **Payment**: Zahlungsabwicklung
- **Revenue Share**: Umsatzbeteiligung

Safety & Audit

APIs für Sicherheit und Audit:

- **Security Monitoring**: Sicherheitsüberwachung
- **Audit Logging**: Audit-Protokollierung
- **Compliance**: Compliance-Prüfung
- **Incident Reporting**: Incident-Meldung

23.2.2. Physical APIs

Neben den logischen APIs gibt es auch physische APIs:

PowerControl API (HV/DC)

API für Hochspannungs-/Gleichstrom-Steuerung:

- **Voltage Control**: Spannungssteuerung
- **Current Control**: Stromsteuerung
- **Power Management**: Leistungsmanagement
- **Safety**: Sicherheitsfunktionen

Docking API (Mechanical + Data)

API für mechanisches Andocken und Datenübertragung:

- **Mechanical Docking**: Mechanisches Andocken
- **Data Transfer**: Datenübertragung
- **Power Transfer**: Energieübertragung
- **Status**: Docking-Status

Vehicle Bus API (CAN/ETH/TSN)

API für Fahrzeugbus-Zugriff:

- **CAN Bus**: Zugriff auf CAN-Bus
- **Ethernet**: Zugriff auf Ethernet
- **TSN**: Zugriff auf TSN-Netzwerk
- **Bus Monitoring**: Bus-Überwachung

Sensor Access API

API für Sensoren-Zugriff:

- **Sensor Reading**: Sensoren auslesen
- **Sensor Configuration**: Sensoren konfigurieren
- **Sensor Calibration**: Sensoren kalibrieren
- **Sensor Health**: Sensoren-Status

Charging Station/V2G API

API für Ladesäulen und Vehicle-to-Grid:

- **Charging Control**: Ladesteuerung
- **V2G**: Vehicle-to-Grid-Funktionen
- **Charging Station Communication**: Kommunikation mit Ladesäule
- **Charging Status**: Lade-Status

Edge Compute API

API für Edge-Computing:

- **Compute Allocation**: Zuweisung von Rechenressourcen
- **Task Execution**: Ausführung von Tasks
- **Resource Monitoring**: Ressourcen-Überwachung
- **Resource Management**: Ressourcen-Verwaltung

23.3. Sicherheit und Attestation

Sicherheit ist von zentraler Bedeutung für VAN.APPVERSE, insbesondere bei sicherheitskritischen Funktionen.

23.3.1. Developer Identity

Entwickler müssen sich registrieren und verifizieren:

- **Developer Registration**: Registrierung von Entwicklern
- **Identity Verification**: Identitätsverifizierung
- **Certification**: Zertifizierung von Entwicklern
- **Reputation System**: Reputationssystem für Entwickler

23.3.2. App Signing

Apps müssen signiert werden:

- **Code Signing**: Code-Signierung mit digitalen Zertifikaten
- **Certificate Chain**: Zertifikatskette für Vertrauen
- **Signature Verification**: Signatur-Verifizierung
- **Revocation**: Widerruf von Zertifikaten

23.3.3. Entitlements

Apps benötigen Berechtigungen für API-Zugriffe:

- **Entitlement Model**: Berechtigungsmodell
- **Least Privilege**: Prinzip der geringsten Berechtigung
- **Entitlement Verification**: Berechtigungsprüfung
- **Dynamic Entitlements**: Dynamische Berechtigungen

23.3.4. Hardware Attestation

Hardware-Attestation für Sicherheit:

- **TPM**: Trusted Platform Module für Hardware-Attestation
- **Secure Element**: Sichere Elemente für Schlüsselspeicherung
- **Attestation Protocol**: Attestations-Protokoll
- **Remote Attestation**: Fern-Attestation

23.3.5. App Review

Apps werden vor der Veröffentlichung überprüft:

- **Code Review**: Code-Überprüfung
- **Security Scan**: Sicherheits-Scan
- **Functional Testing**: Funktionale Tests
- **Compliance Check**: Compliance-Prüfung

23.3.6. Sandbox

Apps laufen in einer Sandbox:

- **Isolation**: Isolation von Apps
- **Resource Limits**: Ressourcen-Limits
- **Access Control**: Zugriffskontrolle
- **Monitoring**: Überwachung von Apps

23.3.7. Runtime Protections

Runtime-Schutz für Apps:

- **Memory Protection**: Speicherschutz
- **Stack Protection**: Stack-Schutz
- **Control Flow Integrity**: Kontrollfluss-Integrität
- **Intrusion Detection**: Intrusion Detection

23.4. API-Governance

API-Governance stellt sicher, dass APIs konsistent, sicher und wartbar sind.

23.4.1. Developer Portal

Das Developer Portal bietet Entwicklern Zugang zu APIs:

- **API Documentation**: API-Dokumentation
- **SDKs**: Software Development Kits
- **Code Examples**: Code-Beispiele
- **Testing Tools**: Test-Tools
- **Support**: Entwickler-Support

23.4.2. App Store/Marketplace

Der App Store/Marketplace ermöglicht die Verteilung von Apps:

- **App Listing**: App-Listings
- **App Discovery**: App-Entdeckung
- **App Reviews**: App-Bewertungen
- **App Ratings**: App-Bewertungen
- **App Categories**: App-Kategorien

23.4.3. Pricing Models

Verschiedene Preismodelle für Apps:

- **Free**: Kostenlose Apps
- **One-Time Purchase**: Einmaliger Kauf
- **Subscription**: Abonnements
- **Usage-Based**: Nutzungsbasierte Abrechnung
- **Freemium**: Freemium-Modell

23.4.4. Revenue Share

Umsatzbeteiligung zwischen Entwicklern und Plattform:

- **Revenue Split**: Umsatzaufteilung (z. B. 70/30)
- **Payment Processing**: Zahlungsabwicklung
- **Tax Handling**: Steuerbehandlung
- **Reporting**: Umsatzberichte

23.4.5. SLAs

Service Level Agreements für APIs:

- **Availability**: Verfügbarkeit (z. B. 99.9%)
- **Latency**: Latenz (z. B. < 100 ms)
- **Throughput**: Durchsatz (z. B. 1000 Requests/s)
- **Support**: Support-Level

23.5. Microservices-Architektur

VAN.APPVERSE basiert auf einer Microservices-Architektur.

23.5.1. Microservice-Prinzipien

Microservices folgen folgenden Prinzipien:

- **Single Responsibility**: Jeder Microservice hat eine einzige Verantwortlichkeit
- **Independence**: Microservices sind unabhängig voneinander
- **Decentralization**: Dezentrale Architektur
- **Failure Isolation**: Fehler-Isolation
- **Technology Diversity**: Technologie-Vielfalt

23.5.2. Service-Mesh

Ein Service-Mesh ermöglicht die Kommunikation zwischen Microservices:

- **Service Discovery**: Service-Erkennung
- **Load Balancing**: Lastverteilung
- **Circuit Breaker**: Circuit Breaker für Fehlertoleranz
- **Retry Logic**: Wiederholungslogik
- **Observability**: Beobachtbarkeit

23.5.3. API-Gateway

Ein API-Gateway bietet zentrale Funktionen:

- **Routing**: Routing von Anfragen
- **Authentication**: Authentifizierung
- **Authorization**: Autorisierung
- **Rate Limiting**: Rate Limiting
- **Monitoring**: Überwachung

23.6. 100 Ideen für VAN.APPVERSE

Dieser Abschnitt präsentiert 100 Ideen für VAN.APPVERSE-Anwendungen, die die Vielfalt der möglichen Mobilitätsdienste demonstrieren. Diese Ideen decken verschiedene Bereiche ab: Logistik & Fracht-Management, Fahrerassistenz & Sicherheit, Komfort & Infotainment, Energie & Nachhaltigkeit, und Flotten-Management. Jede Idee kann als Microservice implementiert werden und über den VAN.APPVERSE App Store bereitgestellt werden.

23.6.1. Kategorisierung

Die 100 Ideen sind in folgende Kategorien unterteilt:

- **Logistik & Fracht-Management** (20 Ideen): Apps für die Optimierung von Logistik- und Fracht-Management-Prozessen
- **Fahrerassistenz & Sicherheit** (20 Ideen): Apps für erweiterte Fahrerassistenz- und Sicherheitsfunktionen
- **Komfort & Infotainment** (20 Ideen): Apps für Komfort und Unterhaltung
- **Energie & Nachhaltigkeit** (20 Ideen): Apps für Energie-Optimierung und Nachhaltigkeit
- **Flotten-Management** (20 Ideen): Apps für Flotten-Management und -Optimierung

23.6.2. Logistik & Fracht-Management (20 Ideen)

1. **Intelligente Laderaum-Organisation**: App zur optimalen Organisation des Laderaums basierend auf Fracht-Daten. Nutzt KI-Algorithmen zur Berechnung der optimalen Anordnung von Fracht basierend auf Gewicht, Volumen, Form und Zielort. Integration mit Laderaum-Sensoren (Gewicht, Volumen) und Kameras für visuelle Verifikation.
2. **Fracht-Tracking**: Echtzeit-Tracking von Fracht im Fahrzeug. Nutzt RFID, BLE-Beacons oder Computer Vision zur Lokalisierung von Fracht im Laderaum. Integration mit Flotten-Management-Systemen für vollständige Sichtbarkeit der Lieferkette.
3. **Temperatur-Monitoring**: Überwachung der Temperatur im Laderaum für Kühlketten. Nutzt Temperatursensoren und KI zur Vorhersage von Temperaturänderungen. Automatische Alarmierung bei Temperaturabweichungen. Integration mit Kühlketten-Management-Systemen.

4. **Gewichts-Optimierung**: Optimierung der Ladung basierend auf Gewicht und Volumen. Nutzt Gewichtssensoren und KI-Algorithmen zur Berechnung der optimalen Ladung. Berücksichtigt Fahrzeug-Gewichtsgrenzen, Achslasten und Schwerpunkt. Integration mit Laderaum-Management-Systemen.

5. **Route-Optimierung**: Optimierung von Lieferrouten basierend auf Fracht-Daten. Nutzt KI-Algorithmen zur Berechnung der optimalen Route unter Berücksichtigung von Fracht-Prioritäten, Zeitfenstern und Verkehrsbedingungen. Integration mit Navigationssystemen und Flotten-Management-Systemen.

6. **Laderaum-Belegung**: Visuelle Darstellung der Laderaum-Belegung. Nutzt Kameras, LiDAR oder Ultraschall-Sensoren zur Erfassung der Laderaum-Belegung. 3D-Visualisierung des Laderaums mit AR-Overlay. Integration mit Laderaum-Management-Systemen.

7. **Fracht-Dokumentation**: Automatische Dokumentation von Fracht. Nutzt Computer Vision zur automatischen Erkennung und Dokumentation von Fracht. Integration mit Dokumenten-Management-Systemen. Automatische Generierung von Fracht-Papieren.

8. **Lieferbestätigung**: Digitale Lieferbestätigung mit Foto. Nutzt Kameras zur Aufnahme von Fotos bei Lieferung. Integration mit E-Signatur-Systemen. Automatische Übertragung von Lieferbestätigungen an Kunden und Flotten-Management-Systeme.

9. **Fracht-Versicherung**: Integration von Fracht-Versicherungen. Automatische Versicherungsabwicklung basierend auf Fracht-Daten. Integration mit Versicherungs-APIs. Automatische Schadensmeldung bei Unfällen.

10. **Laderaum-Sicherheit**: Überwachung der Laderaum-Sicherheit. Nutzt Sensoren (Türsensoren, Bewegungssensoren, Kameras) zur Überwachung der Laderaum-Sicherheit. Automatische Alarmierung bei Sicherheitsverletzungen. Integration mit Sicherheitsdiensten.

11. **Fracht-Kategorisierung**: Automatische Kategorisierung von Fracht. Nutzt Computer Vision und KI zur automatischen Kategorisierung von Fracht. Integration mit Fracht-Management-Systemen. Automatische Etikettierung von Fracht.

12. **Laderaum-Zugriff**: Kontrollierter Zugriff auf den Laderaum. Nutzt Zugriffskontrollsysteme (RFID, Biometrie) zur Kontrolle des Laderaum-Zugriffs. Integration mit Flotten-Management-Systemen. Audit-Logging aller Zugriffe.

13. **Fracht-Alarm**: Alarm bei unerwarteten Änderungen der Fracht. Nutzt Sensoren zur Erkennung von unerwarteten Änderungen der Fracht (Gewicht, Position, Temperatur). Automatische Alarmierung an Fahrer und Flotten-Management. Integration mit Sicherheitsdiensten.

14. **Laderaum-Klimatisation**: Intelligente Klimatisierung des Laderaums. Nutzt Temperatur- und Feuchtigkeitssensoren zur intelligenten Klimatisierung. KI-basierte Vorhersage von Klimaanforderungen. Integration mit Klimatisierungs-Systemen.

15. **Fracht-Reporting**: Automatische Berichte über Fracht. Automatische Generierung von Berichten über Fracht (Status, Position, Zustand). Integration mit Reporting-Systemen. Echtzeit-Dashboards für Fracht-Status.

16. **Laderaum-Inventar**: Automatisches Inventar des Laderaums. Nutzt Sensoren zur automatischen Erfassung des Laderaum-Inventars. Integration mit Inventar-Management-Systemen. Automatische Synchronisation mit Backend-Systemen.

17. **Fracht-Optimierung**: KI-basierte Optimierung der Fracht. Nutzt KI-Algorithmen zur Optimierung von Fracht-Ladung, -Route und -Zeitplanung. Integration mit Optimierungs-Engines. Kontinuierliche Verbesserung basierend auf historischen Daten.

18. **Laderaum-Zugangssteuerung**: Zugangssteuerung für den Laderaum. Nutzt Zugriffskontrollsysteme zur Steuerung des Laderaum-Zugriffs. Integration mit Flotten-Management-Systemen. Zeitbasierte Zugriffskontrolle.

19. **Fracht-Integration**: Integration mit Logistik-Systemen. Integration mit ERP-Systemen, WMS (Warehouse Management Systems) und TMS (Transport Management Systems). Automatische Synchronisation von Fracht-Daten. Echtzeit-Updates zwischen Systemen.

20. **Laderaum-Analytics**: Analytics für Laderaum-Nutzung. Nutzt Analytics-Tools zur Analyse der Laderaum-Nutzung. Identifikation von Optimierungspotenzialen. Integration mit Business-Intelligence-Systemen.

23.6.3. Fahrerassistenz & Sicherheit (20 Ideen)

21. **Blind-Spot-Monitoring**: Erweiterte Überwachung von toten Winkeln. Nutzt Kameras, Radar und LiDAR zur Überwachung von toten Winkeln. KI-basierte Objekterkennung zur Identifikation von Fahrzeugen, Fußgängern und Radfahrern. Visuelle und akustische Warnungen für den Fahrer. Integration mit Side-Mirror-Displays.

22. **Parkassistenz**: Intelligente Parkassistenz für Vans. Nutzt Ultraschall-Sensoren, Kameras und LiDAR zur Erkennung von Parkplätzen. KI-basierte Parkplatz-Erkennung und -Bewertung. Automatische Parkplatz-Auswahl. Integration mit Lenk- und Brems-Systemen für automatisches Einparken.

23. **Müdigkeitserkennung**: Erkennung von Müdigkeit beim Fahrer. Nutzt In-Cabin-Kameras zur Erkennung von Müdigkeit (Augenbewegungen, Gesichtsausdruck, Kopfhaltung). KI-basierte Müdigkeitserkennung. Automatische Warnungen und Empfehlungen für Pausen. Integration mit Navigationssystemen zur Pausen-Planung.

24. **Notfall-Assistent**: Automatischer Notfall-Assistent. Automatische Erkennung von Unfällen durch Beschleunigungssensoren und Kameras. Automatische Notfall-Benachrichtigung an Rettungsdienste. Übertragung von Fahrzeug-Daten (Position, Geschwindigkeit, Aufprall-Daten) an Rettungsdienste. Integration mit eCall-Systemen.

25. **Fahrstil-Analyse**: Analyse des Fahrstils für Versicherungen. Nutzt Sensoren zur Erfassung von Fahrstil-Daten (Beschleunigung, Bremsen, Kurvenfahrten). KI-basierte Analyse des Fahrstils. Berechnung von Fahrstil-Scores. Integration mit Versicherungs-APIs für usage-based insurance.

26. **Sicherheits-Score**: Sicherheits-Score basierend auf Fahrverhalten. Berechnung von Sicherheits-Scores basierend auf Fahrverhalten, Unfällen und Verstößen. KI-basierte Vorhersage von Sicherheitsrisiken. Empfehlungen zur Verbesserung der Sicherheit. Integration mit Flotten-Management-Systemen.

27. **Unfall-Prävention**: KI-basierte Unfall-Prävention. Nutzt Sensoren (Kameras, Radar, LiDAR) zur Erkennung von Gefahren. KI-basierte Vorhersage von Unfall-Risiken. Proaktive Warnungen und automatische Bremsung. Integration mit ADAS-Systemen.

28. **Fahrzeug-Überwachung**: Überwachung des Fahrzeugs bei Abwesenheit. Nutzt Kameras, Bewegungssensoren und GPS zur Überwachung des Fahrzeugs. Automatische Alarmierung bei unerwarteten Aktivitäten. Fernzugriff auf Fahrzeug-Kameras. Integration mit Sicherheitsdiensten.

29. **Diebstahl-Schutz**: Diebstahl-Schutz durch intelligente Überwachung. Nutzt Sensoren zur Erkennung von Diebstahl-Versuchen. Automatische Alarmierung und Fahrzeug-Ortung. Fernsperrung des Fahrzeugs. Integration mit Diebstahl-Schutz-Services.

30. **Fahrzeug-Health**: Überwachung der Fahrzeug-Gesundheit. Nutzt Sensoren zur Überwachung von Fahrzeug-Komponenten (Motor, Batterie, Reifen, etc.). KI-basierte Vorhersage von Ausfällen. Predictive Maintenance. Integration mit Wartungs-Systemen.

31. **Warning-System**: Erweiterte Warnsysteme. KI-basierte Warnsysteme für verschiedene Gefahren (Fußgänger, Radfahrer, Tiere, Hindernisse). Kontextbewusste Warnungen. Personalisierte Warnungen basierend auf Fahrer-Präferenzen. Integration mit ADAS-Systemen.

32. **Sicherheits-Training**: Sicherheits-Training für Fahrer. Interaktive Sicherheits-Trainings basierend auf Fahrverhalten. Personalisierte Trainings-Empfehlungen. Gamification von Sicherheits-Trainings. Integration mit Flotten-Management-Systemen.

33. **Fahrzeug-Diagnose**: Erweiterte Fahrzeug-Diagnose. Nutzt OBD-II und andere Diagnose-Interfaces zur Erfassung von Fahrzeug-Daten. KI-basierte Diagnose von Problemen. Empfehlungen für Reparaturen. Integration mit Werkstätten-APIs.

34. **Sicherheits-Alarme**: Intelligente Sicherheits-Alarme. Kontextbewusste Alarme basierend auf Fahrzeug-Status und Umgebung. Personalisierte Alarm-Einstellungen. Integration mit Smartphone-Apps für Push-Benachrichtigungen.

35. **Fahrzeug-Schutz**: Schutz des Fahrzeugs vor Vandalismus. Nutzt Kameras und Bewegungssensoren zur Erkennung von Vandalismus. Automatische Aufnahme von Videos bei Vandalismus. Alarmierung von Sicherheitsdiensten. Integration mit Versicherungs-APIs.

36. **Sicherheits-Reporting**: Automatische Sicherheits-Berichte. Automatische Generierung von Sicherheits-Berichten basierend auf Fahrverhalten und Ereignissen. Integration mit Flotten-Management-Systemen. Echtzeit-Dashboards für Sicherheits-Metriken.

37. **Fahrzeug-Tracking**: GPS-Tracking für Sicherheit. Echtzeit-Tracking des Fahrzeugs. Geofencing für Sicherheits-Alarme. Integration mit Flotten-Management-Systemen. Historische Tracking-Daten für Analyse.

38. **Sicherheits-Integration**: Integration mit Sicherheitsdiensten. Integration mit Sicherheitsdiensten (Polizei, Rettungsdienste, Sicherheitsfirmen). Automatische Benachrichtigung bei Sicherheitsvorfällen. Integration mit Notfall-Services.

39. **Fahrzeug-Überwachung**: Fernüberwachung des Fahrzeugs. Fernzugriff auf Fahrzeug-Kameras und -Sensoren. Echtzeit-Überwachung des Fahrzeugs. Integration mit Cloud-Services für Daten-Speicherung und -Analyse.

40. **Sicherheits-Analytics**: Analytics für Sicherheit. Nutzt Analytics-Tools zur Analyse von Sicherheits-Daten. Identifikation von Sicherheits-Trends. Vorhersage von Sicherheits-Risiken. Integration mit Business-Intelligence-Systemen.

23.6.4. Komfort & Infotainment (20 Ideen)

41. **Personalisiertes Klima**: Personalisierte Klimasteuerung basierend auf Vorlieben. Nutzt In-Cabin-Sensoren (Temperatur, Feuchtigkeit, CO2) und KI zur personalisierten Klimasteuerung. Lernfähige Systeme, die sich an individuelle Vorlieben anpassen. Integration mit Gesundheits-Apps für optimale Luftqualität.

42. **Medien-Streaming**: Streaming von Medien-Inhalten. Unterstützung für verschiedene Streaming-Dienste (Spotify, Apple Music, Netflix, etc.). Personalisierte Empfehlungen basierend auf Nutzungsverhalten. Integration mit Smartphone-Apps für nahtlose Übergabe.

43. **Navigation**: Erweiterte Navigation mit POIs. KI-basierte Routenplanung unter Berücksichtigung von Verkehr, Wetter und persönlichen Präferenzen. Integration mit POI-Datenbanken für interessante Orte. Voice-guided Navigation mit natürlicher Sprachausgabe.

44. **Sprach-Assistent**: Intelligenter Sprach-Assistent. Nutzt natürliche Sprachverarbeitung (NLP) für intuitive Steuerung. Integration mit verschiedenen Services (Navigation, Medien, Klima, etc.). Kontextbewusste Antworten und Proaktivität.

45. **Fahrzeug-Personalisierung**: Personalisierung des Fahrzeugs. Speicherung von Fahrer-Profilen mit persönlichen Einstellungen (Sitzposition, Spiegel, Klima, etc.). Automatische Anpassung bei Fahrer-Wechsel. Integration mit Cloud-Services für Profil-Synchronisation.

46. **Komfort-Funktionen**: Erweiterte Komfort-Funktionen. Sitzheizung, -kühlung und -massage. Beleuchtungssteuerung mit verschiedenen Modi. Geräuschreduzierung für ruhige Fahrt. Integration mit Komfort-Sensoren.

47. **Entertainment**: Entertainment-Systeme. Video-Streaming für Passagiere. Gaming-Systeme für Unterhaltung. Integration mit verschiedenen Entertainment-Diensten. Personalisierte Entertainment-Empfehlungen.

48. **Kommunikation**: Kommunikations-Dienste. Handsfree-Telefonie mit Sprachsteuerung. Integration mit Smartphone-Apps für Nachrichten. Videotelefonie für Passagiere. Integration mit Kommunikations-APIs.

49. **Produktivität**: Produktivitäts-Apps für Fahrer. E-Mail-Zugriff mit Sprachsteuerung. Kalender-Integration für Termine. Aufgaben-Management. Integration mit Produktivitäts-APIs.

50. **Wohlbefinden**: Apps für Wohlbefinden. Gesundheits-Monitoring (Herzfrequenz, Stress-Level). Entspannungs-Apps mit Meditation und Musik. Ergonomie-Apps für optimale Sitzposition. Integration mit Gesundheits-APIs.

51. **Ernährung**: Ernährungs-Apps. Restaurant-Empfehlungen basierend auf Route und Präferenzen. Nährwert-Informationen für Mahlzeiten. Integrierte Bestellung und Bezahlung. Integration mit Restaurant-APIs.

52. **Fitness**: Fitness-Apps. Fitness-Tracking während der Fahrt. Trainings-Empfehlungen für Pausen. Integration mit Fitness-Trackern. Gamification von Fitness-Aktivitäten.

53. **Entspannung**: Entspannungs-Apps. Entspannungs-Musik und -Sounds. Aromatherapie-Integration. Massage-Funktionen für Sitze. Integration mit Wohlbefindens-APIs.

54. **Lernen**: Lern-Apps. Sprachlern-Apps für unterwegs. Hörbücher und Podcasts. Integration mit Lern-Plattformen. Personalisierte Lern-Empfehlungen.

55. **Gaming**: Gaming-Apps (nur für Passagiere). Mobile Gaming auf Infotainment-Displays. Integration mit Gaming-Plattformen. Multiplayer-Gaming zwischen Fahrzeugen. Integration mit Gaming-APIs.

56. **Social Media**: Social Media-Integration. Integration mit Social Media-Plattformen (Facebook, Twitter, Instagram, etc.). Social Media-Updates während der Fahrt (nur für Passagiere). Integration mit Social Media-APIs.

57. **Messaging**: Messaging-Dienste. Integration mit Messaging-Apps (WhatsApp, Telegram, etc.). Sprachbasierte Nachrichten. Automatische Antworten während der Fahrt. Integration mit Messaging-APIs.

58. **Videokonferenz**: Videokonferenz-Dienste. Videokonferenzen für Passagiere. Integration mit Videokonferenz-Plattformen (Zoom, Teams, etc.). High-Quality-Video mit stabiler Verbindung. Integration mit Videokonferenz-APIs.

59. **Produktivitäts-Tools**: Produktivitäts-Tools. Dokumenten-Zugriff und -Bearbeitung. Cloud-Storage-Integration. Collaboration-Tools. Integration mit Produktivitäts-APIs.

60. **Unterhaltung**: Unterhaltungs-Apps. Verschiedene Unterhaltungs-Apps für Passagiere. Personalisierte Unterhaltungs-Empfehlungen. Integration mit Unterhaltungs-Diensten. Multimediale Unterhaltungs-Erlebnisse.

23.6.5. Energie & Nachhaltigkeit (20 Ideen)

61. **Energie-Optimierung**: KI-basierte Energie-Optimierung. Nutzt KI-Algorithmen zur Optimierung des Energieverbrauchs basierend auf Route, Verkehr und Fahrverhalten. Kontinuierliche Anpassung der Fahrstrategie. Integration mit Energiemanagement-Systemen.

62. **Lade-Optimierung**: Optimierung des Ladevorgangs. Intelligente Ladeplanung basierend auf Route, Energiepreisen und Batterie-Zustand. Vorausschauende Ladeplanung für optimale Reichweite. Integration mit Lade-Station-APIs.

63. **V2G-Integration**: Vehicle-to-Grid-Integration. Integration mit Vehicle-to-Grid-Systemen zur Rückeinspeisung von Energie. Automatische Optimierung von Lade- und Entlade-Zeiten basierend auf Energiepreisen. Integration mit Smart-Grid-APIs.

64. **Energie-Monitoring**: Überwachung des Energieverbrauchs. Echtzeit-Überwachung des Energieverbrauchs mit detaillierten Statistiken. Identifikation von Energie-Ineffizienzen. Integration mit Energiemonitoring-Systemen.

65. **Nachhaltigkeits-Score**: Nachhaltigkeits-Score. Berechnung von Nachhaltigkeits-Scores basierend auf Energieverbrauch, CO2-Emissionen und Fahrverhalten. Vergleich mit anderen Fahrzeugen. Integration mit Nachhaltigkeits-Plattformen.

66. **CO2-Tracking**: CO2-Tracking und -Reduzierung. Tracking von CO2-Emissionen basierend auf Energieverbrauch und Energiemix. Empfehlungen zur CO2-Reduzierung. Integration mit CO2-Tracking-Systemen.

67. **Energie-Effizienz**: Apps zur Verbesserung der Energieeffizienz. Empfehlungen zur Verbesserung der Energieeffizienz basierend auf Fahrverhalten. Gamification von Energieeffizienz. Integration mit Energieeffizienz-APIs.

68. **Lade-Station-Finder**: Finder für Lade-Stationen. Echtzeit-Finder für verfügbare Lade-Stationen basierend auf Route und Batterie-Zustand. Integration mit Lade-Station-Datenbanken. Reservierung von Lade-Stationen.

69. **Lade-Planung**: Intelligente Lade-Planung. Automatische Ladeplanung basierend auf Route, Energiebedarf und Lade-Station-Verfügbarkeit. Optimierung von Lade-Zeiten und -Kosten. Integration mit Lade-Planungs-APIs.

70. **Energie-Preise**: Integration von Energie-Preisen. Echtzeit-Integration von Energie-Preisen für optimale Lade-Zeiten. Preismodell-Vergleich (Fixpreis, Zeitvariable Preise, etc.). Integration mit Energie-Preis-APIs.

71. **Nachhaltigkeits-Reporting**: Reporting über Nachhaltigkeit. Automatische Generierung von Nachhaltigkeits-Berichten. Integration mit Reporting-Systemen. Echtzeit-Dashboards für Nachhaltigkeits-Metriken.

72. **Energie-Analytics**: Analytics für Energie. Nutzt Analytics-Tools zur Analyse von Energie-Daten. Identifikation von Energie-Optimierungspotenzialen. Integration mit Business-Intelligence-Systemen.

73. **Regenerative Energie**: Integration von regenerativer Energie. Integration mit regenerativen Energie-Quellen (Solar, Wind, etc.). Optimierung der Energie-Nutzung basierend auf regenerativer Energie-Verfügbarkeit. Integration mit regenerativen Energie-APIs.

74. **Energie-Speicherung**: Intelligente Energie-Speicherung. Optimierung der Energie-Speicherung basierend auf Energie-Preisen und -Verfügbarkeit. Integration mit Energiespeicher-Systemen. Vorausschauende Energiespeicherung.

75. **Energie-Sharing**: Energie-Sharing zwischen Fahrzeugen. Sharing von Energie zwischen Fahrzeugen in Flotten oder Communities. Peer-to-Peer-Energie-Handel. Integration mit Energie-Sharing-Plattformen.

76. **Nachhaltigkeits-Challenges**: Challenges für Nachhaltigkeit. Gamification von Nachhaltigkeit durch Challenges und Achievements. Vergleich mit anderen Fahrern. Integration mit Nachhaltigkeits-Challenge-Plattformen.

77. **Energie-Gamification**: Gamification von Energie. Gamification von Energieverbrauch durch Points, Badges und Leaderboards. Motivation zur Energie-Optimierung. Integration mit Gamification-Plattformen.

78. **Nachhaltigkeits-Education**: Bildung über Nachhaltigkeit. Educational-Content über Nachhaltigkeit und Energie. Personalisierte Lern-Empfehlungen. Integration mit Bildungs-Plattformen.

79. **Energie-Community**: Community für Energie. Community-Plattform für Energie-Enthusiasten. Sharing von Energie-Tipps und -Strategien. Integration mit Community-Plattformen.

80. **Nachhaltigkeits-Integration**: Integration mit Nachhaltigkeits-Diensten. Integration mit Nachhaltigkeits-Diensten und -Plattformen. Automatische Synchronisation von Nachhaltigkeits-Daten. Integration mit Nachhaltigkeits-APIs.

23.6.6. Flotten-Management (20 Ideen)

81. **Flotten-Tracking**: Echtzeit-Tracking von Flotten. GPS-basiertes Tracking aller Fahrzeuge in der Flotte. Echtzeit-Status-Updates (Position, Geschwindigkeit, Zustand). Integration mit Flotten-Management-Systemen. Historische Tracking-Daten für Analyse.

82. **Route-Optimierung**: Optimierung von Flotten-Routen. KI-basierte Optimierung von Routen für gesamte Flotte unter Berücksichtigung von Verkehr, Wetter und Aufgaben. Dynamische Reoptimierung bei Änderungen. Integration mit Routenoptimierungs-Engines.

83. **Fahrzeug-Zuordnung**: Intelligente Zuordnung von Fahrzeugen zu Aufgaben. KI-basierte Zuordnung von Fahrzeugen zu Aufgaben basierend auf Kapazität, Position und Verfügbarkeit. Optimierung von Zuordnungen für Effizienz. Integration mit Aufgaben-Management-Systemen.

84. **Wartungs-Planung**: Planung von Wartungen. Predictive Maintenance basierend auf Fahrzeug-Daten. Automatische Wartungsplanung unter Berücksichtigung von Fahrzeug-Nutzung und -Zustand. Integration mit Wartungs-Systemen.

85. **Fahrzeug-Utilization**: Überwachung der Fahrzeug-Nutzung. Echtzeit-Überwachung der Fahrzeug-Nutzung (Fahrzeit, Standzeit, Auslastung). Identifikation von untergenutzten Fahrzeugen. Integration mit Utilization-Analytics-Systemen.

86. **Flotten-Analytics**: Analytics für Flotten. Umfassende Analytics für Flotten-Performance, -Effizienz und -Kosten. Identifikation von Optimierungspotenzialen. Integration mit Business-Intelligence-Systemen.

87. **Fahrer-Management**: Management von Fahrern. Verwaltung von Fahrer-Profilen, -Zeiten und -Leistungen. Fahrer-Performance-Tracking. Integration mit HR-Systemen.

88. **Aufgaben-Management**: Management von Aufgaben. Verwaltung von Aufgaben, Zuordnungen und Status. Automatische Aufgaben-Zuordnung. Integration mit Aufgaben-Management-Systemen.

89. **Flotten-Optimierung**: KI-basierte Flotten-Optimierung. Kontinuierliche Optimierung von Flotten-Performance basierend auf historischen Daten. Vorhersage von Flotten-Bedarf. Integration mit Optimierungs-Engines.

90. **Fahrzeug-Sharing**: Sharing von Fahrzeugen in Flotten. Effizientes Sharing von Fahrzeugen zwischen verschiedenen Abteilungen oder Teams. Buchungssystem für Fahrzeuge. Integration mit Buchungs-Systemen.

91. **Flotten-Reporting**: Reporting über Flotten. Automatische Generierung von Flotten-Berichten (Performance, Kosten, Nutzung, etc.). Integration mit Reporting-Systemen. Echtzeit-Dashboards für Flotten-Metriken.

92. **Fahrzeug-Health**: Überwachung der Fahrzeug-Gesundheit. Echtzeit-Überwachung der Fahrzeug-Gesundheit aller Fahrzeuge in der Flotte. Predictive Maintenance für proaktive Wartung. Integration mit Health-Monitoring-Systemen.

93. **Flotten-Sicherheit**: Sicherheit von Flotten. Überwachung der Flotten-Sicherheit (Unfälle, Verstöße, etc.). Sicherheits-Training für Fahrer. Integration mit Sicherheits-Systemen.

94. **Fahrzeug-Wartung**: Predictive Maintenance. Vorhersage von Wartungsbedarf basierend auf Fahrzeug-Daten. Automatische Wartungsplanung. Integration mit Wartungs-Systemen.

95. **Flotten-Integration**: Integration mit Flotten-Management-Systemen. Integration mit ERP-Systemen, TMS (Transport Management Systems) und anderen Flotten-Management-Systemen. Automatische Synchronisation von Flotten-Daten. Echtzeit-Updates zwischen Systemen.

96. **Fahrzeug-Diagnose**: Erweiterte Diagnose. Fern-Diagnose von Fahrzeugen in der Flotte. Automatische Fehlererkennung und -meldung. Integration mit Diagnose-Systemen.

97. **Flotten-Monitoring**: Monitoring von Flotten. Echtzeit-Monitoring von Flotten-Performance, -Status und -Gesundheit. Automatische Alarmierung bei Problemen. Integration mit Monitoring-Systemen.

98. **Fahrzeug-Optimierung**: Optimierung von Fahrzeugen. Kontinuierliche Optimierung von Fahrzeug-Performance basierend auf Nutzungsdaten. Empfehlungen zur Fahrzeug-Optimierung. Integration mit Optimierungs-Systemen.

99. **Flotten-Skalierung**: Skalierung von Flotten. Unterstützung für Skalierung von Flotten (Hinzufügen/Entfernen von Fahrzeugen). Automatische Anpassung von Flotten-Management bei Skalierung. Integration mit Skalierungs-Systemen.

100. **Fahrzeug-Lifecycle**: Management des Fahrzeug-Lifecycles. Verwaltung des gesamten Fahrzeug-Lifecycles (Beschaffung, Nutzung, Wartung, Verkauf). Tracking

von Fahrzeug-Kosten über den gesamten Lifecycle. Integration mit Lifecycle-Management-Systemen.

23.7. Erweiterungs- und Integrationsschnittstellen

VAN.APPVERSE bietet verschiedene Schnittstellen für Erweiterungen und Integrationen. Diese Schnittstellen ermöglichen es Entwicklern, die Plattform zu erweitern und mit externen Systemen zu integrieren.

23.7.1. Extension Interfaces

Extension Interfaces ermöglichen die Erweiterung von VAN.APPVERSE:

- **Plugin-System**: Plugin-System für Erweiterungen. Entwickler können Plugins entwickeln, die die Funktionalität von VAN.APPVERSE erweitern. Plugins können neue APIs, Services oder Funktionen hinzufügen. Das Plugin-System unterstützt dynamisches Laden und Entladen von Plugins zur Laufzeit.
- **Extension API**: API für Erweiterungen. Die Extension API ermöglicht es Entwicklern, auf interne VAN.APPVERSE-Funktionen zuzugreifen und diese zu erweitern. Die API bietet Funktionen für Event-Handling, Service-Registry und Ressourcen-Management.
- **Hook-System**: Hook-System für Ereignisse. Entwickler können Hooks registrieren, die bei bestimmten Ereignissen ausgelöst werden (z. B. App-Installation, Fahrzeugstart, etc.). Hooks ermöglichen es, benutzerdefinierte Logik in den VAN.APPVERSE-Lifecycle zu integrieren.
- **Custom Services**: Benutzerdefinierte Services. Entwickler können benutzerdefinierte Services entwickeln, die über die VAN.APPVERSE-API verfügbar gemacht werden. Diese Services können von anderen Apps verwendet werden und tragen zur Erweiterung des Ökosystems bei.

23.7.2. Integration Interfaces

Integration Interfaces ermöglichen die Integration mit externen Systemen:

- **Cloud-Integration**: Integration mit Cloud-Services. VAN.APPVERSE bietet APIs für die Integration mit Cloud-Services (AWS, Azure, Google Cloud, etc.). Entwickler können Cloud-Services für Daten-Speicherung, -Verarbeitung und -Analyse nutzen. Die Integration unterstützt verschiedene Cloud-Protokolle (REST, gRPC, MQTT, etc.).

- **Backend-Integration**: Integration mit Backend-Systemen. VAN.APPVERSE ermöglicht die Integration mit Backend-Systemen (ERP, CRM, Flotten-Management, etc.). Die Integration erfolgt über standardisierte APIs (REST, GraphQL, etc.) und unterstützt verschiedene Authentifizierungsmethoden (OAuth, API-Keys, etc.).
- **Third-Party-Integration**: Integration mit Drittanbietern. VAN.APPVERSE unterstützt die Integration mit Drittanbieter-Services (Payment, Mapping, Weather, etc.). Die Integration erfolgt über standardisierte APIs und ermöglicht es Entwicklern, externe Services in ihre Apps zu integrieren.
- **Legacy-Integration**: Integration mit Legacy-Systemen. VAN.APPVERSE bietet Adapter für die Integration mit Legacy-Systemen (CAN-Bus, LIN-Bus, etc.). Diese Adapter ermöglichen es, bestehende Systeme in VAN.APPVERSE zu integrieren und schrittweise zu migrieren.

23.7.3. API-Gateway und Service-Mesh

VAN.APPVERSE nutzt ein API-Gateway und ein Service-Mesh für die Verwaltung von Microservices:

- **API-Gateway**: Das API-Gateway stellt eine zentrale Schnittstelle für alle API-Anfragen bereit. Es bietet Funktionen für Routing, Authentifizierung, Autorisierung, Rate Limiting und Monitoring. Das API-Gateway ermöglicht es, APIs konsistent und sicher bereitzustellen.
- **Service-Mesh**: Das Service-Mesh ermöglicht die Kommunikation zwischen Microservices. Es bietet Funktionen für Service Discovery, Load Balancing, Circuit Breaking, Retry Logic und Observability. Das Service-Mesh ermöglicht es, Microservices zuverlässig und skalierbar zu betreiben.

23.8. CI/CD Build Pipeline

VAN.APPVERSE nutzt moderne CI/CD-Praktiken für die Entwicklung und Bereitstellung von Apps. Die CI/CD-Pipeline ermöglicht es Entwicklern, Apps schnell und sicher zu entwickeln, zu testen und bereitzustellen.

23.8.1. Continuous Integration

Continuous Integration für Apps:

- **Code Commit**: Code-Commit in Repository. Entwickler committen Code in ein Git-Repository (GitHub, GitLab, etc.). Der Commit löst automatisch die CI-Pipeline aus.
- **Automated Build**: Automatisierter Build. Die CI-Pipeline baut automatisch die App aus dem Source-Code. Der Build-Prozess umfasst Kompilierung, Bundling und Packaging. Build-Artefakte werden in einem Artefakt-Repository gespeichert.
- **Automated Testing**: Automatisierte Tests. Die CI-Pipeline führt automatisch Tests aus (Unit-Tests, Integration-Tests, etc.). Tests werden auf verschiedenen Umgebungen ausgeführt (Linux, Windows, etc.). Test-Ergebnisse werden dokumentiert und bei Fehlern wird der Build abgebrochen.
- **Code Quality**: Code-Qualitätsprüfung. Die CI-Pipeline führt automatisch Code-Qualitätsprüfungen durch (Linting, Static Analysis, Code Coverage, etc.). Qualitäts-Metriken werden dokumentiert und bei Verstößen wird der Build abgebrochen.
- **Security Scan**: Sicherheits-Scan. Die CI-Pipeline führt automatisch Sicherheits-Scans durch (Dependency Scanning, Vulnerability Scanning, etc.). Sicherheits-Probleme werden identifiziert und dokumentiert. Bei kritischen Sicherheits-Problemen wird der Build abgebrochen.

23.8.2. Continuous Delivery

Continuous Delivery für Apps:

- **Automated Deployment**: Automatisierte Bereitstellung. Die CI-Pipeline stellt automatisch Apps in Staging- und Produktions-Umgebungen bereit. Der Deployment-Prozess umfasst Build, Test, Deployment und Verifikation. Deployment-Artefakte werden versioniert und dokumentiert.
- **Staging Environment**: Staging-Umgebung. Apps werden zunächst in einer Staging-Umgebung bereitgestellt. Die Staging-Umgebung simuliert die Produktions-Umgebung und ermöglicht es, Apps vor der Produktions-Bereitstellung zu testen. Staging-Tests umfassen Funktionale Tests, Performance-Tests und Sicherheits-Tests.
- **Production Deployment**: Produktions-Bereitstellung. Apps werden in der Produktions-Umgebung bereitgestellt. Der Production-Deployment-Prozess umfasst Canary Releases, Blue-Green Deployment oder Rolling Updates. Deployment-Strategien werden basierend auf App-Typ und Risiko ausgewählt.

- **Rollback**: Rollback-Mechanismen. Bei Problemen in der Produktion können Apps schnell auf eine vorherige Version zurückgerollt werden. Rollback-Mechanismen umfassen automatische Rollback bei Fehlern und manuelle Rollback bei Bedarf. Rollback-Prozesse sind dokumentiert und getestet.

23.8.3. Continuous Deployment

Continuous Deployment für Apps:

- **Automated Testing**: Automatisierte Tests in Produktion. Nach der Bereitstellung in Produktion werden automatisch Tests ausgeführt (Smoke Tests, Health Checks, etc.). Tests überprüfen, ob die App korrekt funktioniert und ob alle Dependencies verfügbar sind. Bei Fehlern wird automatisch ein Rollback durchgeführt.
- **Canary Releases**: Canary Releases für schrittweise Bereitstellung. Apps werden zunächst nur für einen kleinen Teil der Benutzer bereitgestellt (z. B. 5%). Bei Erfolg wird die Bereitstellung schrittweise auf alle Benutzer ausgeweitet. Canary Releases reduzieren das Risiko von Problemen in der Produktion.
- **Blue-Green Deployment**: Blue-Green Deployment. Apps werden in einer parallelen Umgebung (Green) bereitgestellt, während die aktuelle Umgebung (Blue) weiterläuft. Nach erfolgreicher Verifikation wird der Traffic auf die neue Umgebung umgeschaltet. Blue-Green Deployment ermöglicht schnelles Rollback bei Problemen.
- **Monitoring**: Überwachung nach Bereitstellung. Nach der Bereitstellung werden Apps kontinuierlich überwacht (Performance, Fehler, Nutzung, etc.). Monitoring-Daten werden in Echtzeit erfasst und analysiert. Bei Problemen werden automatisch Alarme ausgelöst.

23.8.4. OTA Updates für Apps

VAN.APPVERSE unterstützt Over-the-Air-Updates für Apps:

- **Update-Management**: Verwaltung von App-Updates. Updates werden über die VAN.APPVERSE-Platform verwaltet und bereitgestellt. Entwickler können Updates erstellen, testen und bereitstellen. Update-Strategien umfassen Forced Updates, Optional Updates und Scheduled Updates.
- **Update-Deployment**: Bereitstellung von Updates. Updates werden automatisch an Fahrzeuge bereitgestellt. Der Deployment-Prozess umfasst Download,

Verifikation, Installation und Verifikation. Updates werden schrittweise bereitgestellt, um Risiken zu minimieren.

- **Update-Rollback**: Rollback von Updates. Bei Problemen mit Updates können Apps auf eine vorherige Version zurückgerollt werden. Rollback-Prozesse sind automatisiert und dokumentiert. Rollback-Mechanismen umfassen automatische Rollback bei Fehlern und manuelle Rollback bei Bedarf.

23.9. Zusammenfassung

Dieses Kapitel hat das Konzept der VAN.APPVERSE – einer offenen Mobilitäts-Microservice-Ökonomie für Vans – detailliert beschrieben. VAN.APPVERSE transformiert Fahrzeuge von geschlossenen Systemen zu offenen Plattformen, auf denen Innovationen von verschiedenen Entwicklern schnell und einfach bereitgestellt werden können. Der professionelle API-Ansatz, die Sicherheitsmechanismen und die Microservices-Architektur bilden die Grundlage für eine erfolgreiche Ökonomie von Mobilitätsdiensten.

Die 100 vorgestellten Ideen demonstrieren die Vielfalt der möglichen Anwendungen und zeigen das Potenzial von VAN.APPVERSE für die Transformation der Mobilitätsbranche. Die Erweiterungs- und Integrationsschnittstellen sowie die CI/CD-Pipeline ermöglichen es Entwicklern, schnell und sicher neue Dienste zu entwickeln und bereitzustellen.

24. Vollständiges E/E-Architektur-Regelwerk für MB Vans

Dieses Kapitel stellt ein umfassendes Regelwerk für die Entwicklung neuer E/E-Architekturen für Mercedes-Benz Vans bereit. Es systematisiert alle Aspekte von der Konzeption bis zum Betrieb und bietet eine vollständige Referenz für Architekten, Entwickler und Projektmanager.

24.1. Übersicht und Struktur des Regelwerks

Das E/E-Architektur-Regelwerk für MB Vans gliedert sich in folgende Hauptbereiche:

- **Architektur-Entscheidungsrichtlinien (ADRs)**: Systematische Dokumentation von Architektur-Entscheidungen
- **Design Patterns und Best Practices**: Bewährte Patterns für E/E-Architekturen
- **MB.OS-Integrationsrichtlinien**: Detaillierte Richtlinien für MB.OS-Integration
- **VAN.EA-Spezifikation**: Vollständige VAN.EA-Architektur-Spezifikation
- **Entwicklungsprozess und Methoden**: V-Modell, Entwicklungsphasen, Meilensteine
- **Qualitätssicherung und Testing**: Teststrategien, Testautomatisierung, Testpyramide
- **Zertifizierung und Compliance**: ISO 26262, UN ECE, Homologation
- **Migration und Legacy-Integration**: Migration von bestehenden Architekturen
- **Toolchain und Werkzeuge**: Entwicklungs-Toolchain, CI/CD, Monitoring

- **Deployment und Betrieb**: OTA-Updates, Monitoring, Wartung
- **Kostenmodell und Wirtschaftlichkeit**: Kostenanalyse, ROI, Budget-Planung
- **Dokumentationsrichtlinien**: Standardisierte Dokumentation, Templates

24.2. Architecture Decision Records (ADRs)

Architecture Decision Records (ADRs) dokumentieren wichtige Architektur-Entscheidungen systematisch und nachvollziehbar.

24.2.1. ADR-Template

Jedes ADR folgt einem standardisierten Template:

- **Title**: Kurzer, beschreibender Titel der Entscheidung
- **Status**: Proposed, Accepted, Deprecated, Superseded
- **Context**: Kontext und Problemstellung
- **Decision**: Getroffene Entscheidung
- **Consequences**: Positive und negative Konsequenzen
- **Alternatives**: Betrachtete Alternativen und deren Bewertung
- **Rationale**: Begründung der Entscheidung
- **References**: Referenzen zu relevanten Dokumenten, Standards, etc.

24.2.2. ADR-Katalog für MB Vans

ADR-001: Zonale Architektur

- **Title**: Zonale Architektur für VAN.EA
- **Status**: Accepted
- **Context**: Entscheidung für die grundlegende Architektur-Struktur
- **Decision**: Verwendung einer zonalen Architektur mit zentraler Rechenplattform
- **Consequences**:
  - **Positiv**: Reduzierte Kabel-Längen, bessere Skalierbarkeit, vereinfachte Wartung

  - **Negativ**: Höhere Komplexität der zentralen Rechenplattform, Single Point of Failure

- **Alternatives**: Domain-basierte Architektur, verteilte Architektur
- **Rationale**: Zonale Architektur ermöglicht optimale Balance zwischen Performance, Kosten und Skalierbarkeit

ADR-002: TSN für Backbone

- **Title**: Time-Sensitive Networking (TSN) für Ethernet-Backbone
- **Status**: Accepted
- **Context**: Entscheidung für das Kommunikationsprotokoll
- **Decision**: Verwendung von TSN für deterministische Kommunikation
- **Consequences**:
  - **Positiv**: Deterministische Latenzen, hohe Bandbreite, Redundanz-Unterstützung
  - **Negativ**: Höhere Komplexität, Anforderungen an Zeitsynchronisation
- **Alternatives**: CAN-FD, FlexRay, Proprietäre Protokolle
- **Rationale**: TSN bietet beste Balance zwischen Performance und Standardisierung

ADR-003: NVIDIA DRIVE Thor als Central Compute

- **Title**: NVIDIA DRIVE Thor als zentrale Rechenplattform
- **Status**: Accepted
- **Context**: Entscheidung für die zentrale Rechenplattform
- **Decision**: Verwendung von NVIDIA DRIVE Thor für zentrale Rechenplattform
- **Consequences**:
  - **Positiv**: Hohe GPU-Performance (2000 TOPS), ASIL-D zertifiziert, Multi-Domain-Support
  - **Negativ**: Höhere Kosten, Abhängigkeit von NVIDIA
- **Alternatives**: Qualcomm Snapdragon, Intel Mobileye, Proprietäre Lösungen
- **Rationale**: NVIDIA DRIVE Thor bietet beste Performance für KI/ML-Workloads und ist ASIL-D zertifiziert

ADR-004: MB.OS als Betriebssystem

- **Title**: MB.OS als Betriebssystem für VAN.EA
- **Status**: Accepted
- **Context**: Entscheidung für das Betriebssystem
- **Decision**: Verwendung von MB.OS als zentrales Betriebssystem
- **Consequences**:
  - **Positiv**: Konsistente Software-Plattform, OTA-Updates, Service-orientierte Architektur
  - **Negativ**: Abhängigkeit von MB.OS-Entwicklung, Migration von Legacy-Software
- **Alternatives**: AUTOSAR Classic, AUTOSAR Adaptive, Proprietäre Lösungen
- **Rationale**: MB.OS bietet optimale Integration mit Mercedes-Benz Ökosystem und ermöglicht Software-Defined Vehicles

ADR-005: Bosch-Sensoren für Perzeption

- **Title**: Bosch-Sensoren für Perzeption
- **Status**: Accepted
- **Context**: Entscheidung für Sensor-Suite
- **Decision**: Verwendung von Bosch-Sensoren (8MP Kamera, Radar, LiDAR)
- **Consequences**:
  - **Positiv**: Hohe Qualität, bewährte Technologie, gute Integration
  - **Negativ**: Höhere Kosten, Abhängigkeit von Bosch
- **Alternatives**: Continental, Valeo, Proprietäre Lösungen
- **Rationale**: Bosch-Sensoren bieten beste Balance zwischen Qualität, Performance und Kosten

ADR-006: Redundanz-Strategien

- **Title**: Redundanz-Strategien für sicherheitskritische Funktionen
- **Status**: Accepted
- **Context**: Entscheidung für Redundanz-Strategien zur Erfüllung von ASIL D Anforderungen
- **Decision**: Verwendung von Hot-Standby-Redundanz für AD-DC, 2-out-of-2 Voting für Aktoren, PRP für Kommunikation
- **Consequences**:
  - **Positiv**: Erfüllung von ASIL D Anforderungen, hohe Verfügbarkeit, Fehlertoleranz
  - **Negativ**: Höhere Kosten, höhere Komplexität, höherer Energieverbrauch
- **Alternatives**: Cold-Standby, 1-out-of-2, Single-Channel
- **Rationale**: Hot-Standby und Voting bieten beste Balance zwischen Sicherheit, Verfügbarkeit und Kosten

ADR-007: Security-Architektur

- **Title**: Security-Architektur für vernetzte Fahrzeuge
- **Status**: Accepted
- **Context**: Entscheidung für Security-Architektur zur Absicherung gegen Cyber-Angriffe
- **Decision**: Verwendung von Hardware Security Module (HSM), Secure Boot, Encryption, Firewall, Intrusion Detection System (IDS)
- **Consequences**:
  - **Positiv**: Hohe Sicherheit, Absicherung gegen Angriffe, Compliance mit Security-Standards
  - **Negativ**: Höhere Kosten, höhere Komplexität, Performance-Overhead
- **Alternatives**: Software-only Security, Minimal Security, Proprietäre Security
- **Rationale**: Hardware-basierte Security bietet beste Sicherheit für vernetzte Fahrzeuge

ADR-008: Energy-Management

- **Title**: Energy-Management-Strategie für elektrische Vans
- **Status**: Accepted
- **Context**: Entscheidung für Energy-Management-Strategie zur Optimierung des Energieverbrauchs
- **Decision**: Verwendung von intelligenter Lastverteilung, Predictive Energy Management, V2G-Integration, regenerative Bremsung
- **Consequences**:
  - **Positiv**: Optimierter Energieverbrauch, längere Reichweite, niedrigere Betriebskosten
  - **Negativ**: Höhere Komplexität, zusätzliche Hardware-Kosten
- **Alternatives**: Passive Energy Management, Reactive Energy Management, No Energy Management
- **Rationale**: Intelligentes Energy Management bietet beste Balance zwischen Energieeffizienz und Kosten

ADR-009: OTA-Update-Strategie

- **Title**: Over-the-Air-Update-Strategie für VAN.EA
- **Status**: Accepted
- **Context**: Entscheidung für OTA-Update-Strategie zur Bereitstellung von Software-Updates
- **Decision**: Verwendung von Delta-Updates, Rollback-Mechanismen, Staged Rollout, A/B Testing
- **Consequences**:
  - **Positiv**: Schnelle Update-Bereitstellung, reduzierte Update-Kosten, verbesserte Funktionalität
  - **Negativ**: Höhere Komplexität, Sicherheitsrisiken, Anforderungen an Netzwerk-Bandbreite
- **Alternatives**: Manual Updates, Full Updates only, No OTA Updates
- **Rationale**: Delta-Updates mit Rollback bieten beste Balance zwischen Update-Effizienz und Sicherheit

ADR-010: VAN.APPVERSE-Integration

- **Title**: VAN.APPVERSE-Integration in VAN.EA
- **Status**: Accepted
- **Context**: Entscheidung für VAN.APPVERSE-Integration zur Bereitstellung von offener Mobilitäts-Plattform
- **Decision**: Verwendung von Microservices-Architektur, API-Gateway, Service-Mesh, Sandboxing, App Store
- **Consequences**:
  - **Positiv**: Offene Plattform, Innovation durch Drittanbieter, neue Geschäftsmodelle
  - **Negativ**: Höhere Komplexität, Sicherheitsrisiken, Anforderungen an API-Management
- **Alternatives**: Closed Platform, Proprietäre Apps only, No Third-Party Apps
- **Rationale**: VAN.APPVERSE-Integration bietet beste Balance zwischen Offenheit und Sicherheit

ADR-011: Edge-Cloud-Hybrid-Architektur

- **Title**: Edge-Cloud-Hybrid-Architektur für VAN.EA
- **Status**: Accepted
- **Context**: Entscheidung für Edge-Cloud-Hybrid-Architektur zur Optimierung von Datenverarbeitung
- **Decision**: Verwendung von Edge Computing für Echtzeit-Verarbeitung, Cloud Computing für komplexe Analysen, Federated Learning
- **Consequences**:
  - **Positiv**: Optimierte Datenverarbeitung, reduzierte Latenz, verbesserte Datenschutz
  - **Negativ**: Höhere Komplexität, Anforderungen an Netzwerk-Infrastruktur
- **Alternatives**: Edge-only, Cloud-only, No Hybrid
- **Rationale**: Edge-Cloud-Hybrid bietet beste Balance zwischen Performance und Skalierbarkeit

ADR-012: Predictive Maintenance

- **Title**: Predictive Maintenance für VAN.EA
- **Status**: Accepted
- **Context**: Entscheidung für Predictive Maintenance zur Optimierung von Wartung
- **Decision**: Verwendung von KI-basierten Modellen, Sensordaten-Analyse, Cloud-basierte Analysen, Wartungsplanung
- **Consequences**:
  - **Positiv**: Reduzierte Wartungskosten, erhöhte Verfügbarkeit, verbesserte Zuverlässigkeit
  - **Negativ**: Höhere Komplexität, Anforderungen an Datenanalyse, Initiale Investitionen
- **Alternatives**: Reactive Maintenance, Preventive Maintenance, No Maintenance Strategy
- **Rationale**: Predictive Maintenance bietet beste Balance zwischen Wartungskosten und Verfügbarkeit

24.3. Design Patterns und Best Practices

Dieser Abschnitt beschreibt bewährte Design Patterns und Best Practices für E/E-Architekturen in Vans.

24.3.1. Architektur-Patterns

Pattern: Zonale Architektur mit Central Compute

Beschreibung: Zonale Architektur mit zentraler Rechenplattform für hohe Performance und Skalierbarkeit.

Anwendung:

- Zonen-Controller für lokale Sensorik/Aktorik
- Zentrale Rechenplattform für komplexe Funktionen
- Ethernet-Backbone für Kommunikation
- TSN für deterministische Kommunikation

Vorteile:

- Reduzierte Kabel-Längen
- Bessere Skalierbarkeit
- Vereinfachte Wartung
- Optimale Ressourcen-Nutzung

Nachteile:

- Höhere Komplexität der zentralen Rechenplattform
- Single Point of Failure (muss durch Redundanz abgesichert werden)
- Höhere Anforderungen an Netzwerk-Bandbreite

Pattern: Service-orientierte Architektur (SOA)

Beschreibung: Service-orientierte Architektur für lose gekoppelte, flexible Kommunikation.

Anwendung:

- DDS/SOME/IP für Service-Kommunikation
- Service-Registry für Service-Discovery
- Service-Versionierung für Backward-Kompatibilität
- QoS-Parameter für Service-Qualität

Vorteile:

- Lose gekoppelte Kommunikation
- Flexible Service-Komposition
- Einfache Erweiterbarkeit
- Wiederverwendbare Services

Nachteile:

- Höhere Latenz durch Middleware
- Komplexere Debugging
- Anforderungen an Service-Management

Pattern: Redundante Architektur

Beschreibung: Redundante Architektur für hohe Verfügbarkeit und funktionale Sicherheit.

Anwendung:

- Redundante ECUs für sicherheitskritische Funktionen
- Redundante Kommunikationspfade (PRP, HSR)
- Redundante Sensoren für kritische Funktionen
- Voting-Mechanismen für Konsens

Vorteile:

- Hohe Verfügbarkeit
- Funktionale Sicherheit (ASIL D)
- Fehlertoleranz
- Kontinuierlicher Betrieb bei Ausfällen

Nachteile:

- Höhere Kosten
- Höhere Komplexität
- Höherer Energieverbrauch
- Anforderungen an Switchover-Logik

Pattern: Edge-Cloud-Hybrid-Architektur

Beschreibung: Edge-Cloud-Hybrid-Architektur für optimierte Datenverarbeitung.

Anwendung:

- Edge Computing für Echtzeit-Verarbeitung (NVIDIA DRIVE Thor)
- Cloud Computing für komplexe Analysen
- Federated Learning für KI-Modell-Training
- 5G-Integration für schnelle Datenübertragung

Vorteile:

- Optimierte Datenverarbeitung
- Reduzierte Latenz für Echtzeit-Anwendungen
- Verbesserter Datenschutz durch lokale Verarbeitung
- Skalierbarkeit durch Cloud-Ressourcen

Nachteile:

- Höhere Komplexität
- Anforderungen an Netzwerk-Infrastruktur
- Daten-Synchronisation zwischen Edge und Cloud
- Höhere Kosten für Cloud-Services

Pattern: Predictive Maintenance

Beschreibung: Predictive Maintenance für optimierte Wartung basierend auf KI-Modellen.

Anwendung:

- KI-basierte Modelle für Fehlervorhersage
- Sensordaten-Analyse für System-Monitoring
- Cloud-basierte Analysen für Flotten-Optimierung
- Wartungsplanung basierend auf Vorhersagen

Vorteile:

- Reduzierte Wartungskosten
- Erhöhte Verfügbarkeit
- Verbesserte Zuverlässigkeit
- Optimierte Wartungsplanung

Nachteile:

- Höhere Komplexität
- Anforderungen an Datenanalyse
- Initiale Investitionen
- Anforderungen an Datenqualität

Pattern: Fleet Management Integration

Beschreibung: Fleet Management Integration für optimierte Flotten-Verwaltung.

Anwendung:

- Echtzeit-Tracking von Fahrzeugen
- Route-Optimierung für Flotten
- Fahrzeug-Zuordnung zu Aufgaben
- Flotten-Analytics und Reporting

Vorteile:

- Optimierte Flotten-Verwaltung
- Reduzierte Betriebskosten
- Verbesserte Effizienz
- Bessere Kundenbetreuung

Nachteile:

- Höhere Komplexität
- Anforderungen an Netzwerk-Infrastruktur
- Daten-Sicherheit und Datenschutz
- Anforderungen an Flotten-Management-Systeme

24.3.2. Anti-Patterns

Anti-Pattern: Monolithische Architektur

Beschreibung: Monolithische Architektur mit allen Funktionen in einer einzigen ECU.

Probleme:

- Keine Skalierbarkeit
- Schwierige Wartung
- Hohe Kabel-Komplexität
- Single Point of Failure

Lösung: Migration zu zonaler Architektur mit Central Compute.

Anti-Pattern: Star-Topologie

Beschreibung: Star-Topologie mit allen ECUs direkt an zentraler ECU angeschlossen.

Probleme:

- Hohe Kabel-Komplexität
- Single Point of Failure
- Begrenzte Skalierbarkeit
- Hohe Kosten

Lösung: Migration zu zonaler Architektur mit Ethernet-Backbone.

Anti-Pattern: Fehlende Redundanz

Beschreibung: Fehlende Redundanz für sicherheitskritische Funktionen.

Probleme:

- Keine Fehlertoleranz
- Nicht erfüllbar für ASIL D
- Hohes Sicherheitsrisiko
- Keine Verfügbarkeits-Garantien

Lösung: Implementierung von Redundanz-Mechanismen für sicherheitskritische Funktionen.

24.4. MB.OS-Integrationsrichtlinien

Dieser Abschnitt beschreibt detaillierte Richtlinien für die Integration von MB.OS in VAN.EA-Architekturen.

24.4.1. MB.OS-Architektur

MB.OS ist das zentrale Betriebssystem für Mercedes-Benz Fahrzeuge und bietet:

- **Vier-Domänen-Architektur**:
  - **Infotainment**: HMI, Navigation, Entertainment
  - **Automated Driving**: ADAS, autonomes Fahren

  - **Body & Comfort**: Komfort-Funktionen, Klima, Beleuchtung
  - **Driving & Charging**: Antrieb, Bremsen, Laden
- **Middleware**: AUTOSAR Adaptive, DDS, SOME/IP
- **Basic OS**: Linux-basiert mit Echtzeit-Erweiterungen
- **Chip-to-Cloud**: Integration mit Cloud-Services
- **OTA-Updates**: Over-the-Air-Updates für Software

24.4.2. MB.OS-Services

MB.OS bietet verschiedene Services für VAN.EA:

- **Sensor-Services**: Zugriff auf Sensordaten (Kameras, Radar, LiDAR)
  - Camera Service: Zugriff auf Kamera-Datenströme
  - Radar Service: Zugriff auf Radar-Daten
  - LiDAR Service: Zugriff auf LiDAR-Daten
  - GNSS Service: Zugriff auf Positionsdaten
  - IMU Service: Zugriff auf Inertialdaten
- **Aktor-Services**: Steuerung von Aktoren (Lenkung, Bremse, Antrieb)
  - Steering Service: Steuerung der Lenkung
  - Brake Service: Steuerung der Bremse
  - Throttle Service: Steuerung des Antriebs
  - Gear Service: Steuerung des Getriebes
- **Navigation-Services**: Routenplanung, Navigation, POIs
  - Route Planning Service: Routenplanung
  - Navigation Service: Navigation während der Fahrt
  - POI Service: Points of Interest
  - Traffic Service: Verkehrsinformationen
- **Communication-Services**: V2X, Telematik, Cloud-Kommunikation
  - V2X Service: Vehicle-to-Everything-Kommunikation
  - Telematics Service: Telematik-Kommunikation

  - Cloud Service: Cloud-Kommunikation
  - Message Service: Nachrichten-Kommunikation
- **Diagnostic-Services**: Diagnose, Fehlerbehandlung, Wartung
  - Diagnostic Service: Fahrzeug-Diagnose
  - Error Handling Service: Fehlerbehandlung
  - Maintenance Service: Wartungs-Services
  - Health Monitoring Service: Gesundheits-Überwachung
- **Energy-Services**: Energiemanagement, Laden, V2G
  - Energy Management Service: Energiemanagement
  - Charging Service: Lade-Services
  - V2G Service: Vehicle-to-Grid-Services
  - Battery Service: Batterie-Services
- **Infotainment-Services**: HMI, Entertainment, Connectivity
  - HMI Service: Human-Machine-Interface
  - Media Service: Medien-Services
  - Connectivity Service: Konnektivitäts-Services
  - App Service: App-Management
- **Body-Services**: Komfort, Sicherheit, Laderaum
  - Comfort Service: Komfort-Services
  - Security Service: Sicherheits-Services
  - Cargo Service: Laderaum-Services
  - Climate Service: Klima-Services

24.4.3. MB.OS-API-Referenz

MB.OS bietet standardisierte APIs für alle Services:

Sensor-APIs

- **Camera API**:
  - `getCameraStream(cameraId, resolution, framerate)`: Abruf von Kamera-Datenströmen

  - `configureCamera(cameraId, settings)`: Konfiguration von Kameras
  - `getCameraStatus(cameraId)`: Status-Abfrage von Kameras
- **Radar API**:
  - `getRadarData(radarId)`: Abruf von Radar-Daten
  - `configureRadar(radarId, settings)`: Konfiguration von Radaren
  - `getRadarStatus(radarId)`: Status-Abfrage von Radaren
- **LiDAR API**:
  - `getLiDARData(lidarId)`: Abruf von LiDAR-Daten
  - `configureLiDAR(lidarId, settings)`: Konfiguration von LiDAR
  - `getLiDARStatus(lidarId)`: Status-Abfrage von LiDAR

Aktor-APIs

- **Steering API**:
  - `setSteeringAngle(angle)`: Setzen des Lenkwinkels
  - `getSteeringAngle()`: Abruf des Lenkwinkels
  - `getSteeringStatus()`: Status-Abfrage der Lenkung
- **Brake API**:
  - `setBrakePressure(pressure)`: Setzen des Bremsdrucks
  - `getBrakePressure()`: Abruf des Bremsdrucks
  - `getBrakeStatus()`: Status-Abfrage der Bremse

24.4.4. MB.OS-Entwicklungsrichtlinien

Service-Entwicklung

- **Service-Definition**: Services müssen klar definiert sein mit Schnittstellen, QoS und Versionierung
- **Service-Implementierung**: Services müssen MB.OS-Standards entsprechen
- **Service-Testing**: Services müssen umfassend getestet werden
- **Service-Dokumentation**: Services müssen dokumentiert sein

App-Entwicklung

- **App-Architektur**: Apps müssen MB.OS-Architektur-Standards entsprechen
- **App-Sicherheit**: Apps müssen Sicherheits-Standards entsprechen
- **App-Testing**: Apps müssen umfassend getestet werden
- **App-Dokumentation**: Apps müssen dokumentiert sein

24.5. VAN.EA-Spezifikation

Dieser Abschnitt beschreibt die vollständige VAN.EA-Spezifikation für MB Vans.

24.5.1. VAN.EA-Hardware

Zentrale Rechenplattform

- **NVIDIA DRIVE Thor**:
  - GPU: 2000 TOPS für KI-Inferenz
  - CPU: 12 ARM-Kerne @ 3.0 GHz
  - RAM: 512 GB LPDDR5X
  - Storage: 1 TB NVMe SSD
  - ASIL: ASIL-D zertifiziert
- **Infotainment-Domain-Controller**:
  - CPU: 8 Kerne
  - GPU: 10 TFLOPS
  - RAM: 16 GB
  - Storage: 256 GB
- **Body-Domain-Controller**:
  - CPU: 4 Kerne
  - RAM: 4 GB
  - Storage: 32 GB

Zonen-Controller

- **Front-Zone-Controller**:
  - CPU: 4 Kerne @ 1.5 GHz
  - RAM: 2 GB
  - Interfaces: 2x Ethernet (1G), 2x CAN-FD, 1x LIN
  - ASIL: ASIL-B
- **Side-Zone-Controller** (Left/Right):
  - CPU: 4 Kerne @ 1.5 GHz
  - RAM: 2 GB
  - Interfaces: 2x Ethernet (1G), 2x CAN-FD, 1x LIN
  - ASIL: ASIL-B
- **Rear-Zone-Controller**:
  - CPU: 4 Kerne @ 1.5 GHz
  - RAM: 2 GB
  - Interfaces: 2x Ethernet (1G), 2x CAN-FD, 1x LIN
  - ASIL: ASIL-B

Sensoren

- **Bosch 8MP Multifunktionskamera** (8x):
  - Auflösung: 3840x2160 @ 30 fps
  - Interface: Ethernet 2.5G
  - ASIL: ASIL-B bis ASIL-D
- **Bosch Long-Range-Radar** (8x):
  - Reichweite: 250 m
  - Interface: CAN-FD oder Ethernet
  - ASIL: ASIL-B
- **Bosch High-Resolution-LiDAR** (4x):
  - Layer: 64 Layer
  - Reichweite: 200 m
  - Interface: Ethernet 10G
  - ASIL: ASIL-B

24.5.2. VAN.EA-Software

MB.OS-Integration

- **Betriebssystem**: MB.OS mit vier Domänen
- **Middleware**: AUTOSAR Adaptive, DDS, SOME/IP
- **Basic OS**: Linux-basiert mit Echtzeit-Erweiterungen
- **Chip-to-Cloud**: Integration mit Cloud-Services
- **OTA-Updates**: Over-the-Air-Updates für Software

Software-Komponenten

- **AD-Domain**: 50+ SWCs (Perzeption, Sensorfusion, Planung, Regelung)
  - Perzeption: Objekterkennung, Spurerkennung, Verkehrszeichen-Erkennung
  - Sensorfusion: Multi-Sensor-Fusion, Objekt-Tracking, Situationsanalyse
  - Planung: Routenplanung, Trajektorien-Planung, Manöver-Planung
  - Regelung: Längsregelung, Querregelung, Kombinierte Regelung
- **Body-Domain**: 30+ SWCs (Komfort, Sicherheit, Laderaum)
  - Komfort: Klimasteuerung, Sitzheizung, Beleuchtung
  - Sicherheit: Airbag-Steuerung, Gurtstraffer, Notbremsassistenz
  - Laderaum: Laderaum-Überwachung, Temperatur-Überwachung, Zugangskontrolle
- **Infotainment-Domain**: 20+ SWCs (HMI, Navigation, Entertainment)
  - HMI: Display-Steuerung, Touch-Interface, Sprachsteuerung
  - Navigation: Routenplanung, Navigation, POI-Verwaltung
  - Entertainment: Media-Player, Radio, Streaming
- **Flotten-Domain**: 10+ SWCs (Telematik, Tracking, Optimierung)
  - Telematik: V2X-Kommunikation, Cloud-Kommunikation
  - Tracking: GPS-Tracking, Fahrzeug-Tracking
  - Optimierung: Route-Optimierung, Flotten-Optimierung

24.5.3. Interface-Spezifikationen

Hardware-Interfaces

- **Ethernet-Interfaces**:
  - 1 Gbps Ethernet für Zonen-Controller
  - 2.5 Gbps Ethernet für Kameras
  - 10 Gbps Ethernet für LiDAR
  - TSN-Konfiguration für deterministische Kommunikation
- **CAN-Interfaces**:
  - CAN-FD 2 Mbps für Aktoren
  - CAN-FD 500 kbps für Legacy-Komponenten
  - Gateway-Funktionalität für CAN-to-Ethernet
- **LIN-Interfaces**:
  - LIN 19.2 kbps für einfache Aktoren
  - Gateway-Funktionalität für LIN-to-CAN

Software-Interfaces

- **DDS-Interfaces**:
  - Publish-Subscribe für Echtzeit-Kommunikation
  - QoS-Parameter für Service-Qualität
  - Topic-Definitionen für verschiedene Daten-Typen
- **SOME/IP-Interfaces**:
  - Service-basierte Kommunikation
  - Service-Discovery für dynamische Service-Findung
  - Service-Versionierung für Backward-Kompatibilität
- **REST-APIs**:
  - REST-APIs für Cloud-Kommunikation
  - OAuth 2.0 für Authentifizierung
  - JSON für Daten-Format

24.6. Entwicklungsprozess und Methoden

Dieser Abschnitt beschreibt den Entwicklungsprozess für E/E-Architekturen in MB Vans.

24.6.1. V-Modell für E/E-Architekturen

Das V-Modell gliedert sich in folgende Phasen:

1. **Anforderungsanalyse**: Definition von Anforderungen und KPIs
2. **System-Design**: System-Architektur-Design
3. **Architektur-Design**: E/E-Architektur-Design
4. **Detail-Design**: Detail-Design von Komponenten
5. **Implementierung**: Implementierung von Komponenten
6. **Komponenten-Test**: Test von einzelnen Komponenten
7. **Integrations-Test**: Test von integrierten Komponenten
8. **System-Test**: Test des gesamten Systems
9. **Validierung**: Validierung gegen Anforderungen
10. **Verifikation**: Verifikation der Korrektheit

24.6.2. Meilenstein-Definitionen

Meilenstein 1: Konzept-Review (Monat 6)

Ziele:

- Vollständige Anforderungsanalyse
- Architektur-Konzept definiert
- Technologie-Auswahl getroffen
- Proof-of-Concept erfolgreich

Deliverables:

- Anforderungsdokument
- Architektur-Konzept-Dokument

- Technologie-Evaluierungs-Bericht
- Proof-of-Concept-Bericht

Review-Kriterien:

- Alle Anforderungen dokumentiert
- Architektur-Konzept vollständig
- Technologie-Auswahl begründet
- Proof-of-Concept erfolgreich

Meilenstein 2: Design-Review (Monat 18)

Ziele:

- Detailliertes Architektur-Design
- Komponenten-Spezifikationen
- Interface-Definitionen
- Sicherheits-Analyse

Deliverables:

- Architektur-Spezifikation
- Komponenten-Spezifikationen
- Interface-Dokumente
- Sicherheits-Analyse-Bericht

Review-Kriterien:

- Architektur-Design vollständig
- Komponenten-Spezifikationen detailliert
- Interfaces definiert
- Sicherheits-Anforderungen erfüllt

Meilenstein 3: Integrations-Review (Monat 36)

Ziele:

- Komponenten implementiert
- Integration erfolgreich
- Tests durchgeführt
- Dokumentation vollständig

Deliverables:

- Implementierte Komponenten
- Integrations-Bericht
- Test-Berichte
- Dokumentation

Review-Kriterien:

- Alle Komponenten implementiert
- Integration erfolgreich
- Tests bestanden
- Dokumentation vollständig

Meilenstein 4: Validierungs-Review (Monat 42)

Ziele:

- System validiert
- Anforderungen erfüllt
- Zertifizierung vorbereitet
- Produktionsreife erreicht

Deliverables:

- Validierungs-Bericht
- Verifikations-Bericht

- Zertifizierungs-Dokumente
- Produktions-Freigabe

Review-Kriterien:

- Alle Anforderungen erfüllt
- System validiert
- Zertifizierung vorbereitet
- Produktionsreife erreicht

24.6.3. Review-Prozesse

Architektur-Review

Zweck: Überprüfung der Architektur auf Vollständigkeit, Konsistenz und Einhaltung von Standards.

Teilnehmer:

- Architekten
- System-Engineers
- Safety-Engineers
- Security-Engineers

Prozess:

1. Architektur-Dokumentation vorbereiten
2. Review-Termin planen
3. Architektur präsentieren
4. Diskussion und Feedback
5. Review-Protokoll erstellen
6. Maßnahmen ableiten

Safety-Review

Zweck: Überprüfung der Safety-Architektur auf Einhaltung von ISO 26262.

Teilnehmer:

- Safety-Engineers
- Architekten
- System-Engineers
- Zertifizierungs-Experten

Prozess:

1. Safety-Dokumentation vorbereiten
2. Safety-Analyse durchführen
3. Safety-Review-Termin planen
4. Safety-Architektur präsentieren
5. Diskussion und Feedback
6. Review-Protokoll erstellen
7. Maßnahmen ableiten

24.6.4. Change-Management

Change-Request-Prozess

Prozess:

1. Change-Request erstellen
2. Impact-Analyse durchführen
3. Change-Board-Review
4. Entscheidung treffen
5. Änderung implementieren
6. Verifikation durchführen
7. Change-Request abschließen

Change-Request-Template

- **Title**: Titel der Änderung
- **Description**: Beschreibung der Änderung
- **Rationale**: Begründung der Änderung
- **Impact**: Auswirkung der Änderung
- **Alternatives**: Alternative Lösungen
- **Cost**: Kosten der Änderung
- **Schedule**: Zeitplan der Änderung
- **Risk**: Risiken der Änderung

24.6.5. Entwicklungsphasen

Phase 1: Konzept (Monate 1-6)

- **Ziele**: Definition von Anforderungen, Architektur-Konzept, Technologie-Auswahl
- **Aktivitäten**:
  - Anforderungsanalyse
  - Architektur-Konzept-Entwicklung
  - Technologie-Evaluierung
  - Proof-of-Concept
- **Deliverables**: Anforderungsdokument, Architektur-Konzept, Technologie-Evaluierung
- **Meilensteine**: Konzept-Review, Technologie-Entscheidung

Phase 2: Design (Monate 7-18)

- **Ziele**: Detailliertes Architektur-Design, Komponenten-Spezifikation
- **Aktivitäten**:
  - Detailliertes Architektur-Design
  - Komponenten-Spezifikation
  - Interface-Definition
  - Sicherheits-Analyse

- **Deliverables**: Architektur-Spezifikation, Komponenten-Spezifikation, Interface-Dokumente
- **Meilensteine**: Design-Review, Sicherheits-Review

Phase 3: Implementierung (Monate 19-36)

- **Ziele**: Implementierung von Komponenten, Integration
- **Aktivitäten**:
  - Komponenten-Implementierung
  - Integration von Komponenten
  - Test-Implementierung
  - Dokumentation
- **Deliverables**: Implementierte Komponenten, Test-Suite, Dokumentation
- **Meilensteine**: Komponenten-Review, Integrations-Review

Phase 4: Validierung (Monate 37-42)

- **Ziele**: Validierung gegen Anforderungen, Verifikation der Korrektheit
- **Aktivitäten**:
  - System-Tests
  - Validierungs-Tests
  - Verifikations-Tests
  - Performance-Tests
- **Deliverables**: Test-Berichte, Validierungs-Berichte, Verifikations-Berichte
- **Meilensteine**: Validierungs-Review, Verifikations-Review

24.7. Qualitätssicherung und Testing

Dieser Abschnitt beschreibt Qualitätssicherung und Testing für E/E-Architekturen.

24.7.1. Test-Strategie

Test-Pyramide

Die Test-Pyramide für E/E-Architekturen gliedert sich in:

- **Unit-Tests** (Basis):
  - Tests für einzelne Komponenten
  - Hohe Test-Coverage ($> 80\%$)
  - Schnelle Ausführung
  - Automatisierte Ausführung
- **Integration-Tests** (Mitte):
  - Tests für integrierte Komponenten
  - Tests für Interfaces
  - Tests für Kommunikation
  - Automatisierte Ausführung
- **System-Tests** (Spitze):
  - Tests für gesamtes System
  - Tests für End-to-End-Funktionen
  - Tests für Performance
  - Manuelle und automatisierte Ausführung

Test-Typen

- **Funktionale Tests**: Tests für Funktionalität
  - Unit-Tests für einzelne Komponenten
  - Integration-Tests für integrierte Komponenten
  - System-Tests für gesamtes System
  - Acceptance-Tests für Abnahmetests
- **Performance-Tests**: Tests für Performance
  - Latenz-Tests für E2E-Latenzen
  - Throughput-Tests für Durchsatz
  - Load-Tests für Last-Tests

  - Stress-Tests für Stress-Tests

- **Sicherheits-Tests**: Tests für funktionale Sicherheit
  - Safety-Tests für ASIL-Compliance
  - Fault-Injection-Tests für Fehlerbehandlung
  - Redundancy-Tests für Redundanz-Mechanismen
  - Failover-Tests für Failover-Mechanismen

- **Cybersecurity-Tests**: Tests für Cybersecurity
  - Penetration-Tests für Sicherheits-Schwachstellen
  - Vulnerability-Scans für Sicherheits-Lücken
  - Encryption-Tests für Verschlüsselung
  - Authentication-Tests für Authentifizierung

- **Umgebungs-Tests**: Tests für Umgebungsbedingungen
  - Temperatur-Tests für Temperatur-Bereiche
  - Feuchtigkeit-Tests für Feuchtigkeit
  - Vibration-Tests für Vibrationen
  - EMC-Tests für elektromagnetische Verträglichkeit

- **Lifecycle-Tests**: Tests für Lebensdauer
  - Aging-Tests für Alterung
  - Endurance-Tests für Ausdauer
  - Reliability-Tests für Zuverlässigkeit
  - MTBF-Tests für Mean Time Between Failures

24.7.2. Test-Szenarien

Nominal-Szenarien

- **Standard-Betrieb**: Tests unter normalen Betriebsbedingungen
  - Stadtverkehr: 50 km/h, gemischter Verkehr
  - Autobahn: 120 km/h, hohe Geschwindigkeit
  - Landstraße: 80 km/h, Kurvenfahrten
  - Parkplatz: Manöver, Einparken

- **Wetter-Bedingungen**:
  - Sonnenschein: Optimale Sichtbedingungen
  - Regen: Reduzierte Sichtbarkeit
  - Nebel: Sehr reduzierte Sichtbarkeit
  - Schnee: Schlechte Sichtbarkeit, rutschige Straßen
- **Verkehrssituationen**:
  - Stau: Stop-and-Go-Verkehr
  - Überholmanöver: Spurwechsel, Beschleunigung
  - Kreuzungen: Abbiegen, Vorfahrt
  - Kreisverkehre: Einfahrt, Ausfahrt

Stress-Szenarien

- **Extreme Last**:
  - Hohe CPU-Last: 90%+ CPU-Auslastung
  - Hohe GPU-Last: 90%+ GPU-Auslastung
  - Hohe Netzwerk-Last: 80%+ Bandbreiten-Auslastung
  - Hohe Sensor-Daten-Rate: Maximale Sensor-Daten-Rate
- **Extreme Umgebungsbedingungen**:
  - Extreme Temperaturen: -40°C bis +85°C
  - Hohe Feuchtigkeit: 95%+ relative Feuchtigkeit
  - Starke Vibrationen: 10g+ Beschleunigung
  - Elektromagnetische Störungen: Hohe EMV-Belastung
- **Netzwerk-Stress**:
  - Netzwerk-Überlastung: 90%+ Bandbreiten-Auslastung
  - Paket-Verlust: 1%+ Paket-Verlust
  - Hohe Latenz: 100ms+ Netzwerk-Latenz
  - Jitter: Hohe Latenz-Schwankungen

Failure-Szenarien

- **ECU-Ausfälle**:
  - AD-DC-Ausfall: Ausfall der zentralen Rechenplattform
  - Zonen-Controller-Ausfall: Ausfall eines Zonen-Controllers
  - Sensor-ECU-Ausfall: Ausfall einer Sensor-ECU
  - Aktor-ECU-Ausfall: Ausfall einer Aktor-ECU
- **Netzwerk-Ausfälle**:
  - TSN-Switch-Ausfall: Ausfall eines TSN-Switches
  - Kabel-Bruch: Ausfall einer Netzwerk-Verbindung
  - Port-Ausfall: Ausfall eines Switch-Ports
  - Zeitsynchronisation-Ausfall: Ausfall von gPTP
- **Sensor-Ausfälle**:
  - Kamera-Ausfall: Ausfall einer Kamera
  - Radar-Ausfall: Ausfall eines Radars
  - LiDAR-Ausfall: Ausfall eines LiDAR
  - GNSS-Ausfall: Ausfall des GNSS-Empfängers
- **Aktor-Ausfälle**:
  - Lenkung-Ausfall: Ausfall der Lenkung
  - Bremse-Ausfall: Ausfall der Bremse
  - Antrieb-Ausfall: Ausfall des Antriebs
  - Getriebe-Ausfall: Ausfall des Getriebes

24.7.3. Test-Daten-Management

Test-Daten-Generierung

- **Synthetische Daten**:
  - Simulation: Generierung von synthetischen Sensordaten
  - Synthese: Generierung von synthetischen Szenarien
  - Variation: Generierung von variierten Test-Daten
  - Edge-Cases: Generierung von Edge-Case-Daten

- **Real-World-Daten**:
  - Datensammlung: Sammlung von Real-World-Daten
  - Aufzeichnung: Aufzeichnung von Fahrzeug-Daten
  - Annotation: Annotation von Daten
  - Validierung: Validierung von Daten
- **Test-Daten-Varianten**:
  - Wetter-Varianten: Regen, Nebel, Schnee, Sonne
  - Verkehrs-Varianten: Stau, Stadtverkehr, Autobahn
  - Szenario-Varianten: Überholen, Abbiegen, Einparken
  - Fehler-Varianten: Sensor-Ausfall, Netzwerk-Ausfall

Test-Daten-Verwaltung

- **Test-Daten-Repository**:
  - Zentrale Speicherung: Zentrale Verwaltung von Test-Daten
  - Versionierung: Versionierung von Test-Daten
  - Katalogisierung: Katalogisierung von Test-Daten
  - Suche: Suche nach Test-Daten
- **Test-Daten-Qualität**:
  - Validierung: Validierung von Test-Daten
  - Qualitäts-Metriken: Qualitäts-Metriken für Test-Daten
  - Daten-Profilierung: Profilierung von Test-Daten
  - Daten-Bereinigung: Bereinigung von Test-Daten
- **Test-Daten-Sicherheit**:
  - Anonymisierung: Anonymisierung von Test-Daten
  - Verschlüsselung: Verschlüsselung von Test-Daten
  - Zugriffskontrolle: Zugriffskontrolle auf Test-Daten
  - Backup: Backup von Test-Daten

24.7.4. Test-Environment-Management

Test-Umgebungen

- **Development-Environment**:
  - Entwicklungsumgebung: Entwicklungsumgebung für Entwickler
  - Lokale Tests: Lokale Tests auf Entwickler-Rechnern
  - Unit-Tests: Unit-Tests in Entwicklungsumgebung
  - Integration-Tests: Integration-Tests in Entwicklungsumgebung
- **Test-Environment**:
  - Test-Umgebung: Dedizierte Test-Umgebung
  - System-Tests: System-Tests in Test-Umgebung
  - Performance-Tests: Performance-Tests in Test-Umgebung
  - Safety-Tests: Safety-Tests in Test-Umgebung
- **Staging-Environment**:
  - Staging-Umgebung: Staging-Umgebung für Pre-Production-Tests
  - Acceptance-Tests: Acceptance-Tests in Staging-Umgebung
  - Integration-Tests: Integration-Tests in Staging-Umgebung
  - Performance-Tests: Performance-Tests in Staging-Umgebung
- **Production-Environment**:
  - Produktions-Umgebung: Produktions-Umgebung für Live-Tests
  - Monitoring: Monitoring in Produktions-Umgebung
  - Canary-Deployment: Canary-Deployment in Produktions-Umgebung
  - A/B-Testing: A/B-Testing in Produktions-Umgebung

Environment-Provisioning

- **Automated Provisioning**:
  - Infrastructure-as-Code: Infrastructure-as-Code für Umgebungen
  - Automated Setup: Automatisiertes Setup von Umgebungen
  - Configuration Management: Konfigurations-Management für Umgebungen
  - Environment Templates: Templates für Umgebungen

- **Containerization**:
  - Docker: Docker-Container für Umgebungen
  - Kubernetes: Kubernetes-Orchestrierung für Umgebungen
  - Container-Registry: Container-Registry für Umgebungen
  - Container-Monitoring: Monitoring von Containern
- **Orchestration**:
  - Kubernetes: Kubernetes-Orchestrierung
  - Docker Swarm: Docker Swarm-Orchestrierung
  - Ansible: Ansible-Orchestrierung
  - Terraform: Terraform-Orchestrierung
- **Monitoring**:
  - Health Monitoring: Health-Monitoring von Umgebungen
  - Performance Monitoring: Performance-Monitoring von Umgebungen
  - Resource Monitoring: Resource-Monitoring von Umgebungen
  - Alerting: Alerting bei Problemen

24.7.5. Test-Automatisierung

CI/CD-Pipeline

- **Continuous Integration**: Automatisierte Tests bei Code-Commit
- **Continuous Testing**: Automatisierte Tests in verschiedenen Umgebungen
- **Continuous Deployment**: Automatisierte Bereitstellung in Test-Umgebungen
- **Test-Reporting**: Automatisierte Test-Berichte

Test-Tools

- **Unit-Testing**: JUnit, pytest, Google Test
- **Integration-Testing**: Testcontainers, Mocking-Frameworks
- **System-Testing**: HiL (Hardware-in-the-Loop), SiL (Software-in-the-Loop)
- **Performance-Testing**: JMeter, Gatling, Locust
- **Security-Testing**: OWASP ZAP, Burp Suite

24.8. Zertifizierung und Compliance

Dieser Abschnitt beschreibt Zertifizierung und Compliance für E/E-Architekturen.

24.8.1. ISO 26262

Funktionale Sicherheit

- **ASIL-Klassifizierung**: Klassifizierung von Funktionen nach ASIL-Level
- **Safety-Analyse**: Hazard Analysis, Risk Assessment
- **Safety-Architektur**: Safety-Architektur-Design
- **Safety-Tests**: Safety-Tests für ASIL-Compliance

ISO 26262-Prozess

- **Phase 1: Konzept**: Safety-Konzept, Hazard Analysis
- **Phase 2: Design**: Safety-Architektur, Safety-Mechanismen
- **Phase 3: Implementierung**: Safety-Implementierung, Safety-Tests
- **Phase 4: Validierung**: Safety-Validierung, Safety-Verifikation

24.8.2. UN ECE

Homologation

- **UN ECE R157**: Automatisiertes Fahren (ALKS)
  - Anforderungen an automatisiertes Fahren
  - Sicherheits-Anforderungen
  - Test-Anforderungen
  - Zertifizierungs-Anforderungen
- **UN ECE R152**: Notbremsassistenz (AEBS)
  - Anforderungen an Notbremsassistenz
  - Sicherheits-Anforderungen
  - Test-Anforderungen
  - Zertifizierungs-Anforderungen
- **UN ECE R130**: Spurhalteassistenz (LDWS)

  - Anforderungen an Spurhalteassistenz
  - Sicherheits-Anforderungen
  - Test-Anforderungen
  - Zertifizierungs-Anforderungen
- **UN ECE R131**: Notbremsassistenz für Fußgänger (PAEB)
  - Anforderungen an Notbremsassistenz für Fußgänger
  - Sicherheits-Anforderungen
  - Test-Anforderungen
  - Zertifizierungs-Anforderungen

Homologations-Prozess

- **Vorbereitung**: Dokumentation, Tests, Validierung
  - Erstellung von Homologations-Dokumentation
  - Durchführung von Homologations-Tests
  - Validierung von Homologations-Anforderungen
  - Vorbereitung von Homologations-Antrag
- **Antrag**: Antrag bei homologierender Behörde
  - Einreichung von Homologations-Antrag
  - Bereitstellung von Dokumentation
  - Bereitstellung von Test-Ergebnissen
  - Zahlung von Gebühren
- **Prüfung**: Prüfung durch homologierende Behörde
  - Prüfung von Dokumentation
  - Prüfung von Test-Ergebnissen
  - Prüfung von Fahrzeugen
  - Prüfung von Systemen
- **Zertifizierung**: Zertifizierung bei erfolgreicher Prüfung
  - Ausstellung von Homologations-Zertifikat
  - Registrierung von Homologation
  - Bereitstellung von Homologations-Markierung
  - Validierung von Homologation

24.8.3. Regionale Compliance

USA (FMVSS)

- **FMVSS 126**: Electronic Stability Control (ESC)
- **FMVSS 136**: Electronic Stability Control for Heavy Vehicles
- **FMVSS 141**: Minimum Sound Requirements for Hybrid and Electric Vehicles
- **FMVSS 150**: Vehicle-to-Vehicle (V2V) Communication

China (GB Standards)

- **GB 14166**: Safety belts and restraint systems
- **GB 14167**: Safety belts and restraint systems for buses
- **GB 17675**: Steering systems
- **GB 21670**: Electronic stability control systems

Europa (ECE/EC)

- **ECE R13**: Braking systems
- **ECE R79**: Steering systems
- **ECE R152**: Advanced emergency braking systems
- **ECE R157**: Automated Lane Keeping Systems (ALKS)

24.9. Migration und Legacy-Integration

Dieser Abschnitt beschreibt Migration und Legacy-Integration für E/E-Architekturen.

24.9.1. Migrations-Strategie

Big-Bang-Migration

- **Beschreibung**: Komplette Migration auf einmal
- **Vorteile**: Schnelle Migration, keine Parallel-Betrieb
- **Nachteile**: Hohes Risiko, keine schrittweise Validierung
- **Anwendung**: Nur bei kleinen Architekturen oder bei kompletter Neuentwicklung

Inkrementelle Migration

- **Beschreibung**: Schrittweise Migration von Komponenten
- **Vorteile**: Niedriges Risiko, schrittweise Validierung
- **Nachteile**: Längere Migrationszeit, Parallel-Betrieb erforderlich
- **Anwendung**: Standard-Ansatz für große Architekturen

24.9.2. Legacy-Integration

Gateway-Ansatz

- **Beschreibung**: Gateway für Integration von Legacy-Systemen
- **Vorteile**: Einfache Integration, keine Änderungen an Legacy-Systemen
- **Nachteile**: Zusätzliche Latenz, Single Point of Failure
- **Anwendung**: Für Legacy-Systeme, die nicht migriert werden können

Adapter-Ansatz

- **Beschreibung**: Adapter für Integration von Legacy-Systemen
- **Vorteile**: Flexible Integration, Anpassung an Legacy-Systeme
- **Nachteile**: Höhere Komplexität, Wartungsaufwand
- **Anwendung**: Für Legacy-Systeme mit speziellen Anforderungen

24.10. Toolchain und Werkzeuge

Dieser Abschnitt beschreibt die Toolchain und Werkzeuge für E/E-Architekturen.

24.10.1. Entwicklungs-Toolchain

Architektur-Modellierung

- **PREEvision**: Architektur-Modellierung und -Design
- **Enterprise Architect**: UML-Modellierung
- **Vector CANoe**: Bus-Simulation und -Testing
- **MathWorks Simulink**: Modellbasierte Entwicklung

Software-Entwicklung

- **IDE**: Visual Studio Code, Eclipse, IntelliJ IDEA
- **Versionierung**: Git, GitLab, GitHub
- **Build-Tools**: CMake, Gradle, Maven
- **Testing**: JUnit, pytest, Google Test

24.10.2. CI/CD-Pipeline

Continuous Integration

- **Source Control**: Git, GitLab, GitHub
- **Build-Server**: Jenkins, GitLab CI, GitHub Actions
- **Artifact-Repository**: Nexus, Artifactory
- **Testing**: Automatisierte Tests in CI-Pipeline

Continuous Deployment

- **Deployment-Tools**: Kubernetes, Docker, Ansible
- **Monitoring**: Prometheus, Grafana, ELK Stack
- **Logging**: ELK Stack, Splunk
- **Alerting**: PagerDuty, Opsgenie

24.11. Deployment und Betrieb

Dieser Abschnitt beschreibt Deployment und Betrieb für E/E-Architekturen.

24.11.1. OTA-Updates

Update-Strategien

- **Full-Update**: Komplettes Update des Systems
- **Delta-Update**: Nur Änderungen werden übertragen
- **Incremental-Update**: Schrittweises Update von Komponenten
- **Rolling-Update**: Update von Fahrzeugen in Gruppen

Update-Prozess

- **Vorbereitung**: Update-Paket-Erstellung, Validierung
- **Verteilung**: Übertragung an Fahrzeuge
- **Installation**: Installation auf Fahrzeugen
- **Verifikation**: Verifikation der Installation
- **Rollback**: Rollback bei Problemen

24.11.2. Monitoring

System-Monitoring

- **Performance-Monitoring**: CPU, GPU, Memory, Network
- **Health-Monitoring**: System-Health, Komponenten-Health
- **Error-Monitoring**: Fehler-Erkennung, Fehler-Reporting
- **Security-Monitoring**: Security-Events, Intrusion-Detection

Monitoring-Tools

- **Prometheus**: Metriken-Sammlung
- **Grafana**: Visualisierung
- **ELK Stack**: Logging und Analyse
- **Splunk**: Log-Analyse

24.12. Kostenmodell und Wirtschaftlichkeit

Dieser Abschnitt beschreibt Kostenmodell und Wirtschaftlichkeit für E/E-Architekturen.

24.12.1. Kosten-Modell

Hardware-Kosten

- **Central Compute**: NVIDIA DRIVE Thor, Infotainment-DC, Body-DC
- **Zonen-Controller**: Front, Side, Rear Zone-Controller
- **Sensoren**: Kameras, Radar, LiDAR, Ultraschall

- **Aktoren**: Lenkung, Bremse, Antrieb
- **Kommunikation**: TSN-Switches, Kabel, Stecker

Software-Kosten

- **Entwicklung**:
  - Entwicklungskosten: Entwicklungskosten für Software-Komponenten
  - Entwickler-Kosten: Kosten für Entwickler (Personalkosten)
  - Tool-Kosten: Kosten für Entwicklungstools
  - Infrastruktur-Kosten: Kosten für Entwicklungs-Infrastruktur
- **Licensing**:
  - Betriebssystem-Lizenzen: Lizenzen für Betriebssysteme (MB.OS)
  - Middleware-Lizenzen: Lizenzen für Middleware (AUTOSAR, DDS)
  - Tool-Lizenzen: Lizenzen für Entwicklungstools
  - Third-Party-Lizenzen: Lizenzen für Third-Party-Software
- **Wartung**:
  - Wartungskosten: Wartungskosten für Software
  - Bug-Fixing: Kosten für Bug-Fixes
  - Updates: Kosten für Software-Updates
  - Support: Kosten für Software-Support
- **Updates**:
  - OTA-Updates: Kosten für OTA-Updates
  - Software-Updates: Kosten für Software-Updates
  - Firmware-Updates: Kosten für Firmware-Updates
  - Security-Updates: Kosten für Security-Updates

Integration-Kosten

- **Hardware-Integration**:
  - Integration: Kosten für Hardware-Integration
  - Testing: Kosten für Integration-Tests
  - Validierung: Kosten für Integration-Validierung

  - Zertifizierung: Kosten für Integration-Zertifizierung

- **Software-Integration**:
  - Integration: Kosten für Software-Integration
  - Testing: Kosten für Software-Integration-Tests
  - Validierung: Kosten für Software-Integration-Validierung
  - Zertifizierung: Kosten für Software-Integration-Zertifizierung

- **System-Integration**:
  - Integration: Kosten für System-Integration
  - Testing: Kosten für System-Integration-Tests
  - Validierung: Kosten für System-Integration-Validierung
  - Zertifizierung: Kosten für System-Integration-Zertifizierung

Betriebs-Kosten

- **Wartung**:
  - Preventive Maintenance: Kosten für vorbeugende Wartung
  - Corrective Maintenance: Kosten für korrektive Wartung
  - Predictive Maintenance: Kosten für prädiktive Wartung
  - Spare Parts: Kosten für Ersatzteile

- **Support**:
  - Technical Support: Kosten für technischen Support
  - Customer Support: Kosten für Kunden-Support
  - Training: Kosten für Training
  - Documentation: Kosten für Dokumentation

- **Monitoring**:
  - Monitoring-Infrastruktur: Kosten für Monitoring-Infrastruktur
  - Monitoring-Tools: Kosten für Monitoring-Tools
  - Alerting: Kosten für Alerting
  - Reporting: Kosten für Reporting

24.12.2. Wirtschaftlichkeit

ROI-Analyse

- **Investitionen**: Hardware, Software, Entwicklung
- **Einsparungen**: Reduzierte Kabel-Kosten, vereinfachte Wartung
- **Mehrwert**: Neue Funktionen, bessere Performance
- **ROI-Berechnung**: Return on Investment über Lebensdauer

ROI-Berechnung

- **ROI-Formel**:

$$ROI = \frac{Ertrge - Investition}{Investition} \times 100\% \tag{24.1}$$

- **Payback-Period**:

$$Payback - Period = \frac{Investition}{JhrlicheErtrge} \tag{24.2}$$

- **NPV (Net Present Value)**:

$$NPV = \sum_{t=0}^{n} \frac{Ertrge_t - Kosten_t}{(1+r)^t} \tag{24.3}$$

  wobei r der Diskontierungs-Satz und n die Anzahl der Jahre ist.

- **IRR (Internal Rate of Return)**:

$$NPV = \sum_{t=0}^{n} \frac{Ertrge_t - Kosten_t}{(1+IRR)^t} = 0 \tag{24.4}$$

ROI-Szenarien

- **Konservatives Szenario**:
  - ROI: 15-20% nach 5 Jahren
  - Payback-Period: 3-4 Jahre
  - NPV: Positiv nach 5 Jahren
  - IRR: 10-15%
- **Realistisches Szenario**:
  - ROI: 25-30% nach 5 Jahren

  - Payback-Period: 2-3 Jahre
  - NPV: Positiv nach 3 Jahren
  - IRR: 15-20%
- **Optimistisches Szenario**:
  - ROI: 35-40% nach 5 Jahren
  - Payback-Period: 1-2 Jahre
  - NPV: Positiv nach 2 Jahren
  - IRR: 20-25%

Kosten-Optimierung

- **Hardware-Optimierung**: Optimierung von Hardware-Kosten
  - Komponenten-Optimierung: Optimierung von Hardware-Komponenten
  - Redundanz-Optimierung: Optimierung von Redundanz-Mechanismen
  - Skalierung: Optimierung durch Skalierung
  - Bulk-Purchasing: Optimierung durch Bulk-Purchasing
- **Software-Optimierung**: Optimierung von Software-Kosten
  - Code-Optimierung: Optimierung von Software-Code
  - Lizenzen-Optimierung: Optimierung von Software-Lizenzen
  - Wartungs-Optimierung: Optimierung von Software-Wartung
  - Open-Source: Verwendung von Open-Source-Software
- **Entwicklungs-Optimierung**: Optimierung von Entwicklungs-Kosten
  - Prozess-Optimierung: Optimierung von Entwicklungs-Prozessen
  - Tool-Optimierung: Optimierung von Entwicklungs-Tools
  - Automatisierung: Automatisierung von Entwicklungs-Prozessen
  - Reusability: Wiederverwendung von Komponenten
- **Wartungs-Optimierung**: Optimierung von Wartungs-Kosten
  - Predictive Maintenance: Optimierung durch Predictive Maintenance
  - Automatisierung: Automatisierung von Wartungs-Prozessen
  - Remote-Maintenance: Optimierung durch Remote-Maintenance
  - Self-Healing: Self-Healing-Systeme

24.13. Troubleshooting und Problembehandlung

Dieser Abschnitt beschreibt Troubleshooting und Problembehandlung für E/E-Architekturen.

24.13.1. Häufige Probleme

Kommunikations-Probleme

- **Netzwerk-Latenz**:
  - Symptome: Hohe E2E-Latenzen, Jitter, Deadline-Misses
  - Ursachen: Netzwerk-Überlastung, TSN-Konfiguration, Switch-Konfiguration
  - Lösungen: TSN-Optimierung, Bandbreiten-Erhöhung, Priority-Configuration
  - Prävention: Netzwerk-Monitoring, Proaktive Optimierung
- **Paket-Verlust**:
  - Symptome: Fehlende Daten, Timeouts, Fehler
  - Ursachen: Netzwerk-Überlastung, Kabel-Probleme, Switch-Probleme
  - Lösungen: Netzwerk-Diagnose, Kabel-Ersatz, Switch-Reparatur
  - Prävention: Netzwerk-Monitoring, Redundanz
- **Zeitsynchronisation**:
  - Symptome: Zeit-Offset, Unsynchronisierte Daten, Fehler
  - Ursachen: gPTP-Ausfall, Netzwerk-Probleme, Clock-Drift
  - Lösungen: gPTP-Konfiguration, Netzwerk-Reparatur, Clock-Synchronisation
  - Prävention: Redundante gPTP-Master, Clock-Monitoring

Performance-Probleme

- **CPU-Überlastung**:
  - Symptome: Hohe CPU-Auslastung, Deadline-Misses, Langsame Reaktion
  - Ursachen: Zu viele Tasks, Ineffiziente Algorithmen, Hohe Last
  - Lösungen: Task-Optimierung, Algorithmus-Optimierung, Last-Reduzierung
  - Prävention: CPU-Monitoring, Proaktive Optimierung
- **GPU-Überlastung**:
  - Symptome: Hohe GPU-Auslastung, Langsame Inferenz, Frame-Drops
  - Ursachen: Zu viele KI-Modelle, Große Modelle, Hohe Auflösung

  - Lösungen: Modell-Optimierung, Quantisierung, Auflösungs-Reduzierung
  - Prävention: GPU-Monitoring, Modell-Optimierung

- **Speicher-Probleme**:
  - Symptome: Speicher-Überlauf, Out-of-Memory, System-Absturz
  - Ursachen: Memory-Leaks, Zu große Datenstrukturen, Hohe Last
  - Lösungen: Memory-Leak-Fixes, Datenstruktur-Optimierung, Last-Reduzierung
  - Prävention: Memory-Monitoring, Proaktive Optimierung

Sicherheits-Probleme

- **Cybersecurity-Angriffe**:
  - Symptome: Unautorisierte Zugriffe, Daten-Leaks, System-Kompromittierung
  - Ursachen: Schwachstellen, Fehlkonfiguration, Social Engineering
  - Lösungen: Patch-Installation, Konfiguration-Korrektur, Intrusion-Detection
  - Prävention: Security-Monitoring, Regelmäßige Updates, Security-Training

- **Safety-Verletzungen**:
  - Symptome: Safety-Verletzungen, System-Ausfälle, Fehler
  - Ursachen: Fehlerhafte Safety-Mechanismen, Redundanz-Ausfälle, Design-Fehler
  - Lösungen: Safety-Mechanismen-Reparatur, Redundanz-Wiederherstellung, Design-Korrektur
  - Prävention: Safety-Monitoring, Regelmäßige Tests, Safety-Reviews

24.13.2. Debugging-Strategien

Logging und Monitoring

- **Strukturiertes Logging**:
  - Log-Level: DEBUG, INFO, WARNING, ERROR, CRITICAL
  - Log-Format: Strukturiertes Format (JSON, XML)
  - Log-Rotation: Automatische Log-Rotation
  - Log-Analyse: Automatische Log-Analyse

- **Monitoring**:

  - Metriken: CPU, GPU, Memory, Network, Disk
  - Alerts: Automatische Alerts bei Problemen
  - Dashboards: Visualisierung von Metriken
  - Trends: Langfristige Trend-Analyse

Fehler-Analyse

- **Fehler-Klassifizierung**:
  - Kritisch: System-Ausfall, Safety-Verletzung
  - Hoch: Performance-Probleme, Funktionalitäts-Probleme
  - Mittel: Warnings, Informations-Meldungen
  - Niedrig: Verbesserungs-Vorschläge
- **Root-Cause-Analyse**:
  - Problem-Identifikation: Identifikation des Problems
  - Ursachen-Analyse: Analyse der Ursachen
  - Lösung-Entwicklung: Entwicklung von Lösungen
  - Verifikation: Verifikation der Lösung

24.14. Lessons Learned und Best Practices

Dieser Abschnitt beschreibt Lessons Learned und Best Practices aus der Praxis.

24.14.1. Lessons Learned

Architektur-Design

- **Frühe Entscheidungen**:
  - Wichtig: Frühe Entscheidungen für Architektur-Struktur
  - Begründung: Späte Änderungen sind teuer und zeitaufwändig
  - Best Practice: ADRs für wichtige Entscheidungen dokumentieren
  - Beispiel: Entscheidung für zonale Architektur früh treffen
- **Skalierbarkeit**:
  - Wichtig: Skalierbarkeit von Anfang an berücksichtigen
  - Begründung: Nachträgliche Skalierung ist schwierig

  - Best Practice: Modulare Architektur, lose Kopplung
  - Beispiel: Verwendung von Microservices-Architektur

- **Redundanz**:
  - Wichtig: Redundanz für sicherheitskritische Funktionen
  - Begründung: Redundanz erhöht Verfügbarkeit und Sicherheit
  - Best Practice: Hot-Standby-Redundanz, Voting-Mechanismen
  - Beispiel: Redundante AD-DC für ASIL-D-Funktionen

Entwicklung

- **Test-Driven Development**:
  - Wichtig: Tests früh und häufig schreiben
  - Begründung: Tests finden Fehler früh und reduzieren Kosten
  - Best Practice: Test-Driven Development, Continuous Testing
  - Beispiel: Unit-Tests für alle Komponenten

- **Code-Reviews**:
  - Wichtig: Code-Reviews für alle Änderungen
  - Begründung: Code-Reviews finden Fehler und verbessern Qualität
  - Best Practice: Peer-Reviews, Checklisten, Automatisierung
  - Beispiel: Automatische Code-Reviews mit Tools

- **Dokumentation**:
  - Wichtig: Dokumentation während der Entwicklung
  - Begründung: Dokumentation erleichtert Wartung und Verständnis
  - Best Practice: Living Documentation, Automatische Dokumentation
  - Beispiel: API-Dokumentation automatisch generieren

Testing

- **Test-Automatisierung**:
  - Wichtig: Automatisierung von Tests
  - Begründung: Automatisierung reduziert Zeit und Kosten
  - Best Practice: CI/CD-Pipeline, Automatische Tests

  - Beispiel: Automatische Tests bei jedem Commit

- **Test-Coverage**:
  - Wichtig: Hohe Test-Coverage
  - Begründung: Hohe Test-Coverage findet mehr Fehler
  - Best Practice: $>$ 80% Code-Coverage, Branch-Coverage
  - Beispiel: Automatische Test-Coverage-Analyse

- **Real-World-Tests**:
  - Wichtig: Tests mit Real-World-Daten
  - Begründung: Real-World-Tests finden Probleme, die Simulation nicht findet
  - Best Practice: Real-World-Daten-Sammlung, Edge-Case-Tests
  - Beispiel: Tests mit aufgezeichneten Fahrzeug-Daten

24.14.2. Best Practices

Architektur-Best-Practices

- **Modularität**:
  - Prinzip: Modulare Architektur mit klaren Interfaces
  - Vorteile: Wartbarkeit, Skalierbarkeit, Testbarkeit
  - Implementierung: Microservices, Service-orientierte Architektur
  - Beispiel: VAN.APPVERSE mit Microservices

- **Lose Kopplung**:
  - Prinzip: Lose Kopplung zwischen Komponenten
  - Vorteile: Unabhängige Entwicklung, einfache Wartung
  - Implementierung: Message-Bus, Event-driven Architektur
  - Beispiel: DDS für lose gekoppelte Kommunikation

- **Hohe Kohäsion**:
  - Prinzip: Hohe Kohäsion innerhalb von Komponenten
  - Vorteile: Klare Verantwortlichkeiten, einfaches Verständnis
  - Implementierung: Single Responsibility Principle
  - Beispiel: Jede SWC hat eine klare Verantwortlichkeit

Entwicklungs-Best-Practices

- **Versionierung**:
  - Prinzip: Semantische Versionierung (Major.Minor.Patch)
  - Vorteile: Klare Versions-Verwaltung, Backward-Kompatibilität
  - Implementierung: Git-Tags, Version-Management
  - Beispiel: API-Versionierung für Backward-Kompatibilität
- **Continuous Integration**:
  - Prinzip: Häufige Integration von Code-Änderungen
  - Vorteile: Frühe Fehler-Erkennung, kontinuierliche Validierung
  - Implementierung: CI/CD-Pipeline, Automatische Tests
  - Beispiel: Automatische Tests bei jedem Commit
- **Code-Qualität**:
  - Prinzip: Hohe Code-Qualität durch Standards und Reviews
  - Vorteile: Wartbarkeit, Lesbarkeit, Fehler-Reduzierung
  - Implementierung: Coding-Standards, Code-Reviews, Linting
  - Beispiel: Automatische Code-Qualitäts-Checks

24.15. Dokumentationsrichtlinien

Dieser Abschnitt beschreibt Dokumentationsrichtlinien für E/E-Architekturen.

24.15.1. Dokumentations-Standards

Architektur-Dokumentation

- **Architektur-Übersicht**: High-Level-Architektur-Übersicht
- **Architektur-Diagramme**: UML-Diagramme, Architektur-Diagramme
- **Komponenten-Dokumentation**: Dokumentation von Komponenten
- **Interface-Dokumentation**: Dokumentation von Interfaces

Entwicklungs-Dokumentation

- **Anforderungen**: Anforderungsdokumente
- **Design**: Design-Dokumente
- **Implementierung**: Code-Dokumentation
- **Testing**: Test-Dokumentation

24.15.2. Dokumentations-Templates

Architektur-Dokumentations-Template

- **Übersicht**: Architektur-Übersicht
- **Komponenten**: Komponenten-Beschreibung
- **Interfaces**: Interface-Beschreibung
- **Deployment**: Deployment-Beschreibung

ADR-Template

- **Title**: Titel der Entscheidung
- **Status**: Status der Entscheidung
- **Context**: Kontext und Problemstellung
- **Decision**: Getroffene Entscheidung
- **Consequences**: Konsequenzen
- **Alternatives**: Alternativen
- **Rationale**: Begründung

24.16. Zusammenfassung

Dieses Kapitel hat ein umfassendes E/E-Architektur-Regelwerk für MB Vans bereitgestellt. Das Regelwerk umfasst:

- **Architecture Decision Records (ADRs)**: Systematische Dokumentation von Architektur-Entscheidungen

- **Design Patterns und Best Practices**: Bewährte Patterns für E/E-Architekturen
- **MB.OS-Integrationsrichtlinien**: Detaillierte Richtlinien für MB.OS-Integration
- **VAN.EA-Spezifikation**: Vollständige VAN.EA-Architektur-Spezifikation
- **Entwicklungsprozess und Methoden**: V-Modell, Entwicklungsphasen, Meilensteine
- **Qualitätssicherung und Testing**: Teststrategien, Testautomatisierung, Testpyramide
- **Zertifizierung und Compliance**: ISO 26262, UN ECE, Homologation
- **Migration und Legacy-Integration**: Migration von bestehenden Architekturen
- **Toolchain und Werkzeuge**: Entwicklungs-Toolchain, CI/CD, Monitoring
- **Deployment und Betrieb**: OTA-Updates, Monitoring, Wartung
- **Kostenmodell und Wirtschaftlichkeit**: Kostenanalyse, ROI, Budget-Planung
- **Dokumentationsrichtlinien**: Standardisierte Dokumentation, Templates

Das Regelwerk bietet eine vollständige Referenz für die Entwicklung neuer E/E-Architekturen für MB Vans und stellt sicher, dass alle Aspekte von der Konzeption bis zum Betrieb abgedeckt sind.

25. Schlussfolgerungen und Ausblick

25.1. Zusammenfassung der Beiträge

Die Arbeit etabliert eine reproduzierbare, validierte Kette von der Architektur bis zur Simulation und zeigt, wie frühe, datengetriebene Architekturentscheidungen möglich werden. Die entwickelten Methoden und Werkzeuge ermöglichen es, komplexe E/E-Architekturen für moderne Fahrzeuge [?, ?] frühzeitig zu evaluieren und zu optimieren, bevor kostspielige Hardware-Prototypen erstellt werden.

Die zentralen Beiträge dieser Arbeit umfassen:

- Eine domänenspezifische Komponententaxonomie und ein erweiterbares Metamodell für moderne E/E-Architekturen, das die Anforderungen autonomer Fahrzeuge [?] und moderner Kommunikationstechnologien [?, ?] berücksichtigt.
- Eine methodische Erweiterung der Synthese-Metrik zur quantitativen Ableitung von Ressourcen-, Timing- und Verfügbarkeitsparametern, die auf etablierten Methoden der Echtzeit-Systeme [?] und funktionalen Sicherheit [?] aufbaut.
- Ein regelbasiertes Transformationsframework, das moderne Simulationsplattformen [?] und modellbasierte Entwicklungsansätze [?, ?] unterstützt.
- Eine systematische Evaluationsmethodik, die moderne Validierungsansätze [?, ?] für Fahrzeugsysteme nutzt.

25.2. Ausblick und zukünftige Arbeiten

Künftige Arbeiten adressieren verschiedene Erweiterungen und Verbesserungen:

25.2.1. Erweiterte Modellierung

- **Physikalische Sensor-/Aktormodelle**: Integration detaillierter physikalischer Modelle für Sensoren [?] und Aktoren, um realistischere Simulationen zu ermöglichen.

- **Adaptive Scheduling-Strategien**: Entwicklung adaptiver Scheduling-Algorithmen, die sich dynamisch an Laständerungen anpassen [?].
- **KI/ML-Integration**: Integration von KI-Modellen [?] in die Simulation, um realistischere Perzeptions- und Entscheidungsprozesse zu modellieren.

25.2.2. Erweiterte Validierung

- **Closed-Loop-Verifikation**: Integration von HIL/SiL-Co-Simulation [?] für Closed-Loop-Validierung.
- **Formale Verifikation**: Kombination von Simulation mit formalen Verifikationsmethoden [?] für sicherheitskritische Systeme.
- **Continuous Validation**: Integration der Validierung in CI/CD-Pipelines für kontinuierliche Qualitätssicherung.

25.2.3. Skalierung und Performance

- **Cloud-basierte Simulationen**: Nutzung von Cloud-Computing-Ressourcen für große Simulationsläufe.
- **Distributed Simulation**: Entwicklung verteilter Simulationsansätze für sehr große Architekturen.
- **Real-time Simulation**: Optimierung für Echtzeit-Simulationen für Hardware-in-the-Loop-Tests.

25.2.4. Integration moderner Technologien

- **Edge-Cloud-Hybrid**: Unterstützung für Edge-Cloud-Hybrid-Architekturen [?].
- **5G/V2X**: Integration von 5G und V2X-Kommunikation [?] in die Simulation.
- **Digital Twins**: Entwicklung von Digital-Twin-Ansätzen für kontinuierliche Validierung während des Fahrzeugbetriebs.

Die kontinuierliche Weiterentwicklung dieser Methoden wird entscheidend sein, um mit der wachsenden Komplexität moderner Fahrzeuge [?, ?] Schritt zu halten und die Entwicklung sicherer, effizienter und zuverlässiger E/E-Architekturen zu unterstützen.

25.3. Lessons Learned und Best Practices

Basierend auf der Entwicklung und Evaluierung des Transformations-Frameworks können mehrere wichtige Erkenntnisse und Best Practices abgeleitet werden, die für zukünftige Projekte von Bedeutung sind.

25.3.1. Architektur-Design

- **Intermediate Model als Abstraktionsebene**: Die Verwendung eines Intermediate Models als Abstraktionsebene zwischen Quell- und Zielformaten hat sich als sehr wertvoll erwiesen. Es ermöglicht es, verschiedene Quellformate (nicht nur PREEvision) und Zielplattformen zu unterstützen, ohne die Transformationslogik zu duplizieren.
- **Schema-basierte Validierung**: Die Verwendung formaler Schemas (JSON Schema, XML Schema) für die Validierung von Modellen hat dazu beigetragen, Fehler frühzeitig zu erkennen und die Konsistenz der Modelle zu gewährleisten.
- **Erweiterbarkeit durch Stereotypen**: Der Stereotyp-Mechanismus ermöglicht es, domänenspezifische Erweiterungen vorzunehmen, ohne das Basis-Metamodell zu ändern. Dies erleichtert die Wartung und ermöglicht es, verschiedene Domänen zu unterstützen.

25.3.2. Transformations-Strategie

- **Regelbasierte Transformation**: Die Verwendung regelbasierter Transformationen hat sich als flexibel und wartbar erwiesen. Regeln können unabhängig voneinander entwickelt, getestet und gewartet werden.
- **Template-basierte Code-Generierung**: Die Verwendung von Templates (z. B. Jinja2) für die Code-Generierung ermöglicht es, die Generierungslogik von der Transformationslogik zu trennen und erleichtert die Wartung.
- **Inkrementelles Vorgehen**: Der Minimalstart-Ansatz hat sich als sehr wertvoll erwiesen. Durch die frühzeitige Implementierung einer einfachen, aber vollständigen Architektur konnten Probleme frühzeitig erkannt und behoben werden.

25.3.3. Validierung und Qualitätssicherung

- **Mehrschichtige Validierung**: Die Validierung auf mehreren Ebenen (Syntax, Semantik, Constraints) hat dazu beigetragen, Fehler in verschiedenen Phasen zu erkennen.

- **Analytische Modelle als Referenz**: Der Vergleich mit analytischen Modellen hat dazu beigetragen, die Korrektheit der Simulationsergebnisse zu validieren.
- **Automatisierte Tests**: Die Verwendung automatisierter Tests (Unit-Tests, Integration-Tests) hat die Qualität des Codes erheblich verbessert und Regressionen verhindert.

25.3.4. Projekt-Management

- **Frühe Stakeholder-Einbindung**: Die frühzeitige Einbindung von Stakeholdern (Domänenexperten, Tool-Experten) hat dazu beigetragen, Anforderungen zu klären und Fehlentwicklungen zu vermeiden.
- **Iterative Entwicklung**: Die iterative Entwicklung mit regelmäßigen Reviews und Anpassungen hat die Qualität der Ergebnisse verbessert und Risiken reduziert.
- **Dokumentation als First-Class Citizen**: Die frühzeitige und kontinuierliche Dokumentation hat dazu beigetragen, Wissen zu bewahren und die Wartbarkeit zu verbessern.

25.4. Beitrag zur Wissenschaft und Praxis

Diese Arbeit leistet mehrere wichtige Beiträge zur Wissenschaft und Praxis der E/E-Architektur-Entwicklung:

25.4.1. Wissenschaftliche Beiträge

- **Metamodell für E/E-Architekturen**: Entwicklung eines erweiterbaren Metamodells, das die Komplexität moderner E/E-Architekturen abbildet und für verschiedene Anwendungsfälle verwendet werden kann.
- **Synthese-Metrik**: Erweiterung der Synthese-Metrik zur automatisierten Ableitung von Simulationsparametern aus Architekturmerkmalen, was die Effizienz der Architektur-Evaluierung verbessert.
- **Transformations-Framework**: Entwicklung eines regelbasierten Transformations-Frameworks, das die Transformation von Architekturmodellen in Simulationsmodelle automatisiert und reproduzierbar macht.

- **Validierungsmethodik**: Entwicklung einer systematischen Methodik zur Validierung von Simulationsergebnissen durch Vergleich mit analytischen Modellen und Sensitivitätsanalysen.

25.4.2. Praktische Beiträge

- **Open-Source-Framework**: Bereitstellung eines Open-Source-Frameworks, das von anderen Entwicklern verwendet und erweitert werden kann.
- **Best Practices**: Dokumentation von Best Practices und Lessons Learned, die für zukünftige Projekte von Nutzen sind.
- **Beispiel-Architekturen**: Bereitstellung von Beispiel-Architekturen und Simulations-Szenarien, die als Referenz für andere Entwickler dienen können.
- **Integration in Entwicklungsworkflow**: Demonstration der Integration von Simulation in den Architektur-Entwicklungsworkflow, was die Effizienz der Entwicklung verbessert.

25.5. Ausblick auf zukünftige Entwicklungen

Die Entwicklung von E/E-Architekturen wird in den kommenden Jahren weiterhin von mehreren Trends geprägt sein, die neue Anforderungen an Simulations- und Validierungsmethoden stellen:

25.5.1. Technologische Trends

- **Zunehmende Komplexität**: Die Komplexität von E/E-Architekturen wird weiter zunehmen, was leistungsfähigere Simulations- und Validierungsmethoden erfordert.
- **KI/ML-Integration**: Die Integration von KI/ML-Modellen in E/E-Architekturen erfordert neue Simulationsansätze, die diese Modelle berücksichtigen.
- **Edge-Cloud-Hybrid**: Die Entwicklung von Edge-Cloud-Hybrid-Architekturen erfordert neue Simulationsansätze, die sowohl Edge- als auch Cloud-Komponenten modellieren.
- **5G/V2X**: Die Integration von 5G und V2X-Kommunikation erfordert neue Simulationsansätze für drahtlose Kommunikation.

25.5.2. Methodische Trends

- **Digital Twins**: Die Entwicklung von Digital-Twin-Ansätzen ermöglicht kontinuierliche Validierung während des Fahrzeugbetriebs.
- **Continuous Validation**: Die Integration von Validierung in CI/CD-Pipelines ermöglicht kontinuierliche Qualitätssicherung.
- **Formale Verifikation**: Die Kombination von Simulation mit formaler Verifikation ermöglicht höhere Sicherheitsgarantien.
- **AI-basierte Optimierung**: Die Verwendung von KI für die automatische Optimierung von Architekturen wird zunehmend relevant.

Diese Trends werden die Entwicklung von E/E-Architekturen in den kommenden Jahren prägen und neue Anforderungen an Simulations- und Validierungsmethoden stellen. Die in dieser Arbeit entwickelten Methoden und Werkzeuge bilden eine solide Grundlage für die Bewältigung dieser Herausforderungen.

25.6. Zusammenfassung der Beiträge

Diese Arbeit leistet mehrere wichtige Beiträge zur Forschung und Praxis der E/E-Architektur-Entwicklung:

25.6.1. Methodische Beiträge

- **Erweiterbares Metamodell**: Entwicklung eines erweiterbaren Metamodells für moderne E/E-Architekturen, das zonale Architekturen, TSN-Netzwerke, KI-Integration und Cybersecurity-Aspekte abdeckt.
- **Synthese-Metrik**: Erweiterte Synthese-Metrik zur automatisierten Ableitung von Simulationsparametern aus Architekturmerkmalen, einschließlich Multi-Core-Scheduling, TSN-Latenz-Berechnung und Energie-Modellierung.
- **Transformations-Framework**: Regelbasiertes Transformations-Framework für die automatische Transformation von Architekturmodellen in Simulationsmodelle, unterstützt durch ein Intermediate Model.
- **Validierungsmethodik**: Systematische Validierungsmethodik mit analytischen Modellen, Sensitivitätsanalysen und Benchmarking.

25.6.2. Praktische Beiträge

- **Open-Source-Framework**: Bereitstellung eines Open-Source-Frameworks für die Transformation und Simulation von E/E-Architekturen.
- **Best Practices**: Dokumentation von Best Practices und Lessons Learned für die praktische Anwendung.
- **Fallstudien**: Umfassende Fallstudien, die die praktische Anwendbarkeit demonstrieren.
- **Benchmarking**: Benchmarking-Ergebnisse, die die Performance und Genauigkeit validieren.

25.7. Fazit

Diese Arbeit hat eine umfassende Methodik zur Transformation von E/E-Architekturmodellen in Simulationsmodelle entwickelt und validiert. Die Methodik ermöglicht es, Architekturentscheidungen frühzeitig zu validieren und Optimierungspotenziale zu identifizieren, bevor kostspielige Hardware-Prototypen erstellt werden.

Die entwickelten Methoden und Werkzeuge bilden eine solide Grundlage für die Bewältigung der Herausforderungen moderner E/E-Architekturen und können einen wichtigen Beitrag zur Effizienz und Qualität der Fahrzeugentwicklung leisten.

25.8. Erweiterte Lessons Learned

Dieser Abschnitt beschreibt erweiterte Lessons Learned aus der Entwicklung und Anwendung des Frameworks.

25.8.1. Technische Lessons Learned

- **Metamodell-Design**: Ein gut durchdachtes Metamodell ist entscheidend für die Erweiterbarkeit
- **Performance**: Streaming-Parsing ist essentiell für große Modelle
- **Validierung**: Frühe und kontinuierliche Validierung spart Zeit
- **Modularität**: Modulares Design ermöglicht einfache Erweiterung

25.8.2. Methodische Lessons Learned

- **Inkrementeller Ansatz**: Inkrementeller Ansatz reduziert Risiken
- **Feedback-Loops**: Kurze Feedback-Loops verbessern die Qualität
- **Dokumentation**: Umfassende Dokumentation ist essentiell
- **Beispiele**: Praktische Beispiele erleichtern das Verständnis

25.8.3. Organisatorische Lessons Learned

- **Stakeholder-Einbindung**: Frühe Einbindung von Stakeholdern ist wichtig
- **Kommunikation**: Regelmäßige Kommunikation verhindert Missverständnisse
- **Risikomanagement**: Proaktives Risikomanagement ist essentiell
- **Qualitätssicherung**: Kontinuierliche Qualitätssicherung spart Zeit

A. Anhang

A.1. Attributkataloge (Auszug)

Tabelle A.1.: Attributkatalog ECU (Auszug)

Attribut	Beschreibung
CPU_Cores	Anzahl/Architektur
GPU_TFLOPS	Rechenleistung GPU
Power_States	Energiezustände/Duty-Cycle
ASIL	Safety-Level

A.2. Export-/IM-Schemata

A.3. Transformationsregeln (Beispiel)

A.4. Szenarien- und Variablenkatalog

Dieser Abschnitt enthält einen umfassenden Katalog von Szenarien und Variablen, die für die Simulation verwendet werden können. Der Katalog dient als Referenz für die Konfiguration von Simulationsläufen und ermöglicht es, verschiedene Betriebsbedingungen systematisch zu testen.

Tabelle A.2.: Stadtverkehr-Szenario: Parameter

Parameter	Wert	Beschreibung
Dauer	30 min	Simulationsdauer
Geschwindigkeit	0-50 km/h	Variable Geschwindigkeit
Objektdichte	20-40	Anzahl Objekte in Szene
Spurwechsel	Häufig	Viele Spurwechsel
Bremsvorgänge	Häufig	Viele Brems- und Beschleunigungsvorgänge
CPU-Last (erwartet)	55-65%	Erwartete CPU-Auslastung
Netzwerk-Last (erwartet)	40-50%	Erwartete Netzwerk-Auslastung

Tabelle A.3.: Autobahn-Szenario: Parameter

Parameter	Wert	Beschreibung
Dauer	60 min	Simulationsdauer
Geschwindigkeit	100-130 km/h	Hohe konstante Geschwindigkeit
Objektdichte	5-15	Weniger Objekte, aber höhere Geschwindigkeit
Spurwechsel	Selten	Wenige Spurwechsel
Bremsvorgänge	Selten	Wenige Bremsvorgänge
CPU-Last (erwartet)	45-55%	Erwartete CPU-Auslastung
Netzwerk-Last (erwartet)	30-40%	Erwartete Netzwerk-Auslastung

A.4.1. Nominal-Szenarien

Stadtverkehr-Szenario

Autobahn-Szenario

A.4.2. Stress-Szenarien

Stau-Szenario

A.4.3. Fehler-Szenarien

ECU-Ausfall-Szenario

A.5. Erweiterte Beispiele

A.5.1. Beispiel: Komplexe Funktionskette

Dieses Beispiel zeigt eine komplexe Funktionskette mit mehreren Sensoren, Verarbeitungsschritten und Aktoren:

- **Chain-ID**: `MultiSensor_Fusion_to_Brake`
- **Startpunkte**:

Tabelle A.4.: Stau-Szenario: Parameter

Parameter	Wert	Beschreibung
Dauer	20 min	Simulationsdauer
Geschwindigkeit	0-20 km/h	Sehr niedrige Geschwindigkeit
Objektdichte	50-100	Sehr viele Objekte in Szene
CPU-Last (erwartet)	70-85%	Sehr hohe CPU-Auslastung
Netzwerk-Last (erwartet)	60-75%	Sehr hohe Netzwerk-Auslastung
Ziel	Deadline-Prüfung	Prüfung unter extremer Last

Tabelle A.5.: ECU-Ausfall-Szenario: Parameter

Parameter	Wert	Beschreibung
Ausfallzeitpunkt	Zufällig	Zufälliger Zeitpunkt während Simulation
Ausfallmodus	Sofortig	Sofortiger Ausfall (keine Degradation)
Betroffene ECU	ZC_Front	Front-Zonen-Controller
Redundanz	Ja	Backup-ECU vorhanden
Switchover-Zeit	< 100 ms	Zeit bis Backup aktiv ist
Ziel	Fehlertoleranz	Prüfung der Fehlertoleranz

  - Front-Kamera (1920x1080, 30 fps)
  - Front-Radar (77 GHz, 20 Hz)
  - LiDAR (64-Layer, 10 Hz)

- **Verarbeitung**:
  - Bildverarbeitung (Kamera) auf ZC_Front
  - Radar-Signalverarbeitung auf ZC_Front
  - LiDAR-Punktwolken-Verarbeitung auf ZC_Front
  - Sensorfusion auf AD-DC
  - Objekterkennung und Tracking auf AD-DC
  - Kollisionsprädiktion auf AD-DC
  - Notbrems-Entscheidung auf AD-DC
- **Endpunkt**: Bremse (EHB - Electro-Hydraulic Brake)
- **E2E-Deadline**: 50 ms (kritisch für Notbremsung)
- **ASIL-Level**: D

A.5.2. Beispiel: Redundante Architektur

Dieses Beispiel zeigt eine redundante Architektur für sicherheitskritische Funktionen:

- **Primärer Pfad**:
  - Sensor → ECU_A → Aktor
- **Backup-Pfad**:
  - Sensor → ECU_B → Aktor
- **Redundanz-Mechanismus**:
  - Voting: Vergleich der Ausgaben beider Pfade
  - Switchover: Automatischer Wechsel bei Ausfall
  - Switchover-Zeit: < 10 ms
- **Verfügbarkeit**: 99.99% (4 Nines)

A.6. Erweiterte Transformations-Beispiele

Dieser Abschnitt enthält detaillierte Beispiele für die Transformation verschiedener Architektur-Elemente.

A.6.1. Beispiel: Transformation einer komplexen Funktionskette

Dieses Beispiel zeigt die vollständige Transformation einer komplexen Funktionskette von der Perzeption bis zur Aktorik.

Architektur

Die Funktionskette umfasst:

- **Sensoren**: 3x Kameras (Front, Left, Right), 2x Radar, 1x LiDAR
- **Verarbeitung**:
  - Bildverarbeitung auf ZC_Front (3x Kamera-Streams)
  - Radar-Signalverarbeitung auf ZC_Front
  - LiDAR-Verarbeitung auf ZC_Front
  - Sensorfusion auf AD-DC
  - Objekterkennung (YOLOv8) auf AD-DC (NPU)

  - Tracking auf AD-DC (CPU)
  - Trajektorien-Prädiktion auf AD-DC (CPU)
  - Pfadplanung auf AD-DC (CPU)
  - Regelung auf AD-DC (CPU)
- **Aktoren**: EPS (Lenkung), EHB (Bremse)

Transformation

Die Transformation erfolgt schrittweise:

1. **Sensoren**: Transformation in OMNeT++ Sensor-Nodes
2. **Zonen-Controller**: Transformation in OMNeT++ StandardHost mit Gateway-Funktionalität
3. **AD-DC**: Transformation in OMNeT++ StandardHost mit CPU/GPU/NPU-Ressourcen
4. **Tasks**: Transformation in OMNeT++ Applications mit Periodicity
5. **Frames**: Transformation in OMNeT++ EthernetFrames mit TSN-Konfiguration
6. **Chains**: Transformation in OMNeT++ CompoundApplications

Ergebnisse

Die Transformation ergab:

- **OMNeT++-Modell**: 150+ Nodes, 500+ Connections
- **Simulationszeit**: 2 Stunden für 1 Stunde Fahrzeit
- **Ergebnisse**: Alle KPIs innerhalb der Ziele

A.6.2. Beispiel: Bosch 8MP Kamera mit NVIDIA DRIVE Thor

Dieses Beispiel zeigt die Transformation einer Bosch 8MP Multifunktionskamera mit NVIDIA DRIVE Thor:

Architektur

Die Architektur umfasst:

- **Bosch 8MP Multifunktionskamera**:
  - Auflösung: 3840x2160 @ 30 fps
  - Interface: Ethernet 2.5G
  - Datenrate: 50 MB/s (komprimiert)
- **NVIDIA DRIVE Thor**:
  - GPU: 2000 TOPS für KI-Inferenz
  - CPU: 12 ARM-Kerne @ 3.0 GHz
  - RAM: 512 GB LPDDR5X
  - Interface: Ethernet 2.5G für Kamera
- **KI-Modell**: YOLOv8 für Objekterkennung

Transformation

Die Transformation erfolgt schrittweise:

1. **Bosch 8MP Kamera**: Transformation in OMNeT++ Sensor-Node mit Ethernet 2.5G Interface
2. **Ethernet-Link**: Transformation in OMNeT++ Ethernet-Link mit TSN-Konfiguration
3. **NVIDIA DRIVE Thor**: Transformation in OMNeT++ StandardHost mit:
   - GPU-Ressourcen (2000 TOPS)
   - CPU-Ressourcen (12 Kerne)
   - Memory-Ressourcen (512 GB)
4. **YOLOv8 Task**: Transformation in OMNeT++ Application mit:
   - Periodicity: 33.3 ms (30 fps)
   - WCET: 15 ms (basierend auf Benchmarks)
   - GPU-Zuweisung: DRIVE Thor GPU

Ergebnisse

Die Transformation ergab:

- **OMNeT++-Modell**: 3 Nodes (Kamera, DRIVE Thor, EPS), 2 Links
- **Simulationszeit**: 5 Minuten für 1 Minute Fahrzeit
- **Ergebnisse**:
  - E2E-Latenz: 21 ms (Ziel: < 100 ms) ✓
  - GPU-Last: 0.975% (Ziel: $< 80\%$) ✓
  - Inferenz-Zeit: 15 ms (Ziel: < 33 ms) ✓

A.7. Relevante Literatur zur Fahrzeugtechnik

Dieser Abschnitt listet relevante Literatur aus dem Bereich der Fahrzeugtechnik auf, die für die Entwicklung und Validierung von E/E-Architekturen von Bedeutung ist. Die Literatur wurde aus dem Katalog von beck-shop.de [?] sowie aus KIT und TUM ausgewählt und deckt verschiedene Aspekte der modernen Fahrzeugentwicklung ab.

A.7.1. E/E-Architekturen und Bussysteme

- **E/E-Architekturen**: [?, ?] - Grundlagen und aktuelle Entwicklungen zu E/E-Architekturen im Fahrzeug
- **Bussysteme**: [?] - Protokolle, Standards und Softwarearchitekturen für Fahrzeugbussysteme
- **TSN in Fahrzeugen**: [?] - Time-Sensitive Networking für Automotive-Anwendungen
- **Car-to-X Kommunikation**: [?] - Moderne Kommunikationstechnologien in Fahrzeugen

A.7.2. Automatisiertes Fahren und Fahrerassistenz

- **Fahrerassistenzsysteme**: [?] - Umfassendes Handbuch zu automatisierten Fahrzeugsystemen
- **KI für autonomes Fahren**: [?] - Deep Learning Techniken für autonomes Fahren
- **Fahrzeugdynamik**: [?] - Regelung und Kontrolle von Fahrzeugsystemen
- **Sensoren**: [?] - Sensortechnologien im Fahrzeug

A.7.3. Software-Engineering und Modellbasierte Entwicklung

- **Automotive Software**: [?] - Software-Engineering für Fahrzeuge
- **Modellbasierte Entwicklung**: [?, ?] - Methoden und Werkzeuge für modellbasierte Entwicklung
- **Cyber-Physical Systems**: [?] - Grundlagen zu eingebetteten Systemen und CPS

A.7.4. Validierung, Verifikation und Simulation

- **Validierung von ADAS**: [?] - Validierung und Testmethoden für Fahrerassistenzsysteme
- **Modellbasiertes Testen**: [?] - Testmethoden für reaktive Systeme
- **Netzwerksimulation**: [?] - Modellierung und Werkzeuge für Netzwerksimulation
- **Echtzeit-Systeme**: [?] - Scheduling-Algorithmen für sicherheitskritische Systeme

A.7.5. Funktionale Sicherheit

- **ISO 26262**: [?, ?] - Standards und praktische Leitfäden zur funktionalen Sicherheit

A.7.6. Allgemeine Fahrzeugtechnik

- **Handbuch Kraftfahrzeugtechnik**: [?] - Umfassendes Standardwerk zur Fahrzeugtechnik

Tabelle A.6.: Übersicht: Typische Hardware-Komponenten in E/E-Architekturen

Komponente	Typ	Performance	Energie	Anwendung
CPU (ARM Cortex-A78)	8-Core	2.5 GHz	5-15 W	General Purpose
GPU (NVIDIA Orin)	2048 CUDA Cores	20 TFLOPS	15-60 W	KI-Inferenz
NPU (NVIDIA Orin)	2048 TOPS	200 TOPS	10-30 W	KI-Inferenz
TSN Switch	8-Port	10 Gbps	2-5 W	Netzwerk
Zonen-Controller	4-Core ARM	1.5 GHz	2-8 W	Gateway

Tabelle A.7.: Übersicht: Kommunikations-Protokolle

Protokoll	Bandbreite	Latenz	Topologie	Anwendung
CAN	1 Mbps	1-10 ms	Bus	Aktoren
CAN-FD	8 Mbps	0.5-5 ms	Bus	Aktoren
LIN	20 kbps	10-100 ms	Bus	Sensoren
FlexRay	10 Mbps	0.1-1 ms	Bus/Star	Safety
Ethernet (100BASE-T1)	100 Mbps	0.1-1 ms	Star	Infotainment
TSN (1 Gbps)	1 Gbps	0.01-0.1 ms	Star	AD/ADAS
TSN (10 Gbps)	10 Gbps	0.001-0.01 ms	Star	AD/ADAS

A.8. Erweiterte Tabellen und Übersichten

A.8.1. Übersicht: Hardware-Komponenten

A.8.2. Übersicht: Kommunikations-Protokolle

A.8.3. Übersicht: Scheduling-Algorithmen

A.9. Erweiterte Code-Beispiele

A.9.1. Beispiel: Python-Transformations-Skript

```
#!/usr/bin/env python3
"""
Beispiel: Transformation von PREEvision-Export zu OMNeT++
"""

import json
import yaml
from jinja2 import Template

def load_preevision_export(file_path):
```

Tabelle A.8.: Übersicht: Scheduling-Algorithmen

Algorithmus	Priorität	Echtzeit	Komplexität	Anwendung
Fixed Priority	Statisch	Ja	Niedrig	Standard
Rate Monotonic	Statisch	Ja	Niedrig	Periodische Tasks
Earliest Deadline First	Dynamisch	Ja	Mittel	Echtzeit
Time-Division Multiplexing	Statisch	Ja	Niedrig	Deterministisch
Round Robin	Dynamisch	Nein	Niedrig	Fairness

```
    """Lädt PREEvision-Export (JSON)"""
    with open(file_path, 'r') as f:
        return json.load(f)

def load_mapping_rules(file_path):
    """Lädt Mapping-Regeln (YAML)"""
    with open(file_path, 'r') as f:
        return yaml.safe_load(f)

def transform_ecu(ecu_data, rules):
    """Transformiert ECU-Daten"""
    omnet_node = {
        'name': ecu_data['name'],
        'type': 'StandardHost',
        'interfaces': []
    }

    # Transformiere Interfaces
    for interface in ecu_data['interfaces']:
        if interface['type'] == 'Ethernet':
            omnet_node['interfaces'].append({
                'type': 'EthernetInterface',
                'bandwidth': interface['bandwidth'],
                'tsn_config': transform_tsn(interface, rules)
            })

    return omnet_node

def transform_tsn(interface, rules):
    """Transformiert TSN-Konfiguration"""
```

```
    return {
        'gate_schedule': interface.get('gate_schedule', []),
        'priority': interface.get('priority', 0),
        'traffic_shaping': interface.get('traffic_shaping', {})
    }

def generate_omnet_config(architecture, output_path):
    """Generiert OMNeT++ Konfiguration"""
    template = Template("""
network {{ network_name }}
{
    submodules:
    {% for node in nodes %}
        {{ node.name }}: StandardHost {
            @display("p={{ node.x }},{{ node.y }}");
        }
    {% endfor %}

    connections:
    {% for link in links %}
        {{ link.source }}.ethg++ <--> {{ link.bandwidth }} <--> {{ link.target }}.et
    {% endfor %}
}
""")

    with open(output_path, 'w') as f:
        f.write(template.render(
            network_name=architecture['name'],
            nodes=architecture['nodes'],
            links=architecture['links']
        ))

# Hauptprogramm
if __name__ == '__main__':
    # Lade Daten
    preevision_data = load_preevision_export('architecture.json')
    mapping_rules = load_mapping_rules('mapping_rules.yaml')
```

```
    # Transformiere
    omnet_architecture = {
        'name': preevision_data['name'],
        'nodes': [transform_ecu(ecu, mapping_rules)
                  for ecu in preevision_data['ecus']],
        'links': preevision_data['links']
    }

    # Generiere OMNeT++ Konfiguration
    generate_omnet_config(omnet_architecture, 'omnet_config.ini')
```

A.9.2. Beispiel: YAML-Mapping-Regeln

```
# Mapping-Regeln für Transformation
mappings:
  ecu:
    source: "PREEvision::ECU"
    target: "OMNeT++::StandardHost"
    attributes:
      name: "name"
      type: "StandardHost"
      interfaces: "transform_interfaces"

  interface:
    source: "PREEvision::Interface"
    target: "OMNeT++::EthernetInterface"
    attributes:
      bandwidth: "bandwidth"
      tsn_config: "transform_tsn_config"

  task:
    source: "PREEvision::Task"
    target: "OMNeT++::Application"
    attributes:
      name: "name"
      period: "period"
      wcet: "wcet"
      priority: "priority"
```

```
frame:
  source: "PREEvision::Frame"
  target: "OMNeT++::EthernetFrame"
  attributes:
    size: "size"
    period: "period"
    priority: "priority"
    route: "transform_route"
```

A.10. Erweiterte Diagramme

A.10.1. Beispiel: Komplexe Architektur-Übersicht

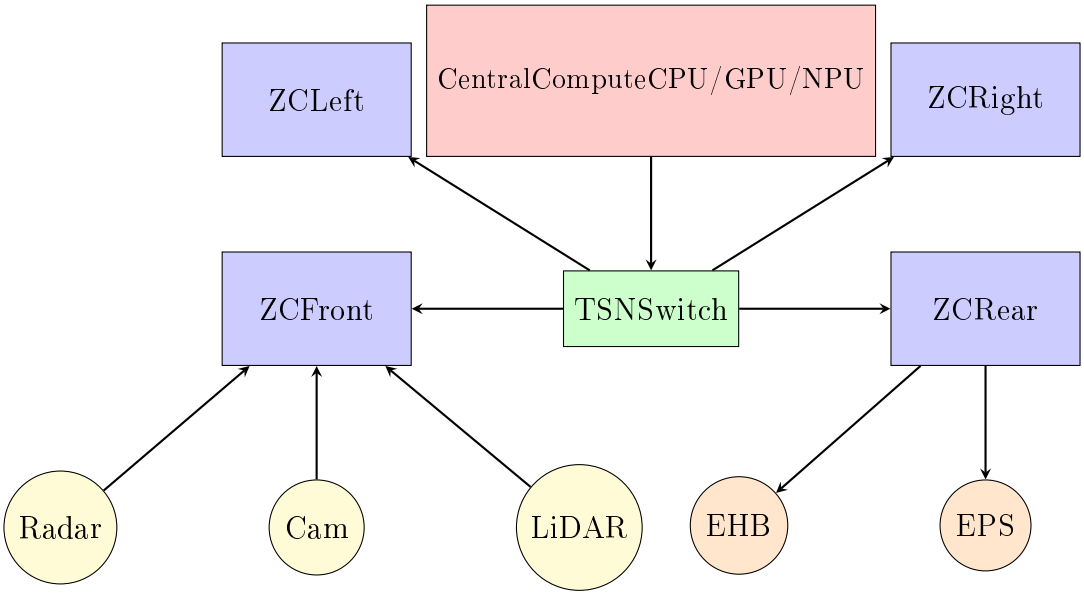

Abbildung A.1.: Beispiel: Komplexe zonale E/E-Architektur

A.10.2. Beispiel: Timing-Diagramm

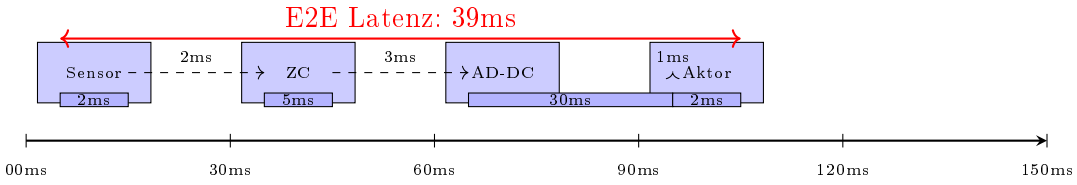

Abbildung A.2.: Beispiel: Timing-Diagramm für Funktionskette

A.11. Erweiterte Konfigurationsbeispiele

A.11.1. Beispiel: OMNeT++ Konfigurationsdatei

```
[Config General]
network = ZonalArchitecture
sim-time-limit = 3600s
**.vector-recording = true
**.scalar-recording = true

[Config Nominal]
extends = General
**.cpu.clock = 2.5GHz
**.network.bandwidth = 10Gbps
**.sensor.frameRate = 30fps

[Config Stress]
extends = General
**.cpu.clock = 2.0GHz
**.network.bandwidth = 5Gbps
**.sensor.frameRate = 60fps
**.objectDensity = 100

[Config Failure]
extends = General
**.ecu[1].failureRate = 0.01
**.link[0].failureRate = 0.005
```

A.11.2. Beispiel: TSN-Konfiguration

```
{
  "tsn_config": {
    "gate_schedules": [
      {
        "port": "port0",
        "schedule": [
          {"time": 0, "state": "open", "duration": 100},
          {"time": 100, "state": "closed", "duration": 900}
        ]
```

```
      }
    ],
    "traffic_classes": [
      {
        "priority": 7,
        "bandwidth": "100Mbps",
        "latency": "1ms",
        "jitter": "0.1ms"
      }
    ],
    "time_sync": {
      "protocol": "gPTP",
      "accuracy": "1us"
    }
  }
}
```

A.12. Erweiterte Tabellen und Referenzdaten

Dieser Abschnitt enthält umfassende Tabellen und Referenzdaten für die Entwicklung von E/E-Architekturen.

A.12.1. Referenz: Typische WCET-Werte

Tabelle A.9.: Referenz: Typische WCET-Werte für verschiedene Tasks

Task-Typ	Hardware	WCET (typisch)	BCET (typisch)
Bildverarbeitung (1920x1080)	CPU	10-20 ms	8-15 ms
Objekterkennung (YOLOv8)	NPU	8-12 ms	6-10 ms
Tracking	CPU	2-5 ms	1-3 ms
Trajektorien-Prädiktion	CPU	5-10 ms	3-7 ms
Pfadplanung	CPU	8-15 ms	5-10 ms
Regelung	CPU	0.5-2 ms	0.3-1 ms
Sensorfusion	CPU	3-8 ms	2-5 ms

Tabelle A.10.: Referenz: Typische Netzwerk-Latenzen

Protokoll	Frame-Größe	Latenz (typisch)	Jitter (typisch)
CAN	8 B	1-5 ms	0.1-0.5 ms
CAN-FD	64 B	0.5-2 ms	0.05-0.2 ms
Ethernet (100 Mbps)	1500 B	0.1-1 ms	0.01-0.1 ms
TSN (1 Gbps)	1500 B	0.01-0.1 ms	0.001-0.01 ms
TSN (10 Gbps)	1500 B	0.001-0.01 ms	0.0001-0.001 ms

Tabelle A.11.: Referenz: Typische Energieverbräuche

Komponente	Zustand	Leistung (typisch)	Energie/Stunde
CPU (8-Core, 2.5 GHz)	Active	10-15 W	10-15 Wh
CPU (8-Core, 2.5 GHz)	Idle	2-5 W	2-5 Wh
GPU (20 TFLOPS)	Active	30-60 W	30-60 Wh
GPU (20 TFLOPS)	Idle	5-10 W	5-10 Wh
NPU (200 TOPS)	Active	15-30 W	15-30 Wh
NPU (200 TOPS)	Idle	2-5 W	2-5 Wh
TSN Switch (8-Port)	Active	3-5 W	3-5 Wh
Kamera (1920x1080)	Active	2-4 W	2-4 Wh
Radar	Active	1-3 W	1-3 Wh
LiDAR (64-Layer)	Active	5-10 W	5-10 Wh

A.12.2. Referenz: Typische Netzwerk-Latenzen

A.12.3. Referenz: Typische Energieverbräuche

A.12.4. Referenz: ASIL-Level-Anforderungen

Tabelle A.12.: Referenz: ASIL-Level-Anforderungen

ASIL-Level	Verfügbarkeit	MTBF	Redundanz
ASIL A	99%	10.000 h	Optional
ASIL B	99.9%	100.000 h	Empfohlen
ASIL C	99.99%	1.000.000 h	Erforderlich
ASIL D	99.999%	10.000.000 h	Erforderlich (2-fach)

A.13. Erweiterte Berechnungsbeispiele

A.13.1. Beispiel: WCRT-Berechnung

Gegeben:

- Task 1: $C_1 = 5$ ms, $T_1 = 20$ ms, Priorität 3
- Task 2: $C_2 = 3$ ms, $T_2 = 15$ ms, Priorität 2
- Task 3: $C_3 = 2$ ms, $T_3 = 10$ ms, Priorität 1 (höchste)

Berechnung für Task 1 (niedrigste Priorität):

$$R_1^0 = C_1 = 5 \text{ ms} \tag{A.1}$$

$$R_1^1 = C_1 + \sum_{j \in hp(1)} \left\lceil \frac{R_1^0}{T_j} \right\rceil C_j \tag{A.2}$$

$$= 5 + \left\lceil \frac{5}{15} \right\rceil \times 3 + \left\lceil \frac{5}{10} \right\rceil \times 2 \tag{A.3}$$

$$= 5 + 1 \times 3 + 1 \times 2 = 10 \text{ ms} \tag{A.4}$$

$$R_1^2 = C_1 + \sum_{j \in hp(1)} \left\lceil \frac{R_1^1}{T_j} \right\rceil C_j \tag{A.5}$$

$$= 5 + \left\lceil \frac{10}{15} \right\rceil \times 3 + \left\lceil \frac{10}{10} \right\rceil \times 2 \tag{A.6}$$

$$= 5 + 1 \times 3 + 1 \times 2 = 10 \text{ ms} \tag{A.7}$$

$$R_1 = 10 \text{ ms} \quad \text{(konvergiert)} \tag{A.8}$$

A.13.2. Beispiel: TSN-Latenz-Berechnung

Gegeben:

- Frame-Größe: 1500 B
- Bandbreite: 1 Gbps
- Gate-Schedule: Port öffnet alle 1 ms für 0.1 ms
- Warteschlangenlatenz: 0.05 ms

Berechnung:

$$L_{tx} = \frac{1500 \times 8}{1 \times 10^9} = 0.012 \text{ ms} \tag{A.9}$$

$$L_{gate} = \max(0, \text{next\_gate\_open} - \text{arrival\_time}) \tag{A.10}$$

$$L_{queue} = 0.05 \text{ ms} \tag{A.11}$$

$$L_{total} = L_{tx} + L_{gate} + L_{queue} \approx 0.1 - 1 \text{ ms} \tag{A.12}$$

A.14. Erweiterte Architektur-Beispiele

A.14.1. Beispiel: Vollständige Van-Architektur

Dieses Beispiel zeigt eine vollständige Architektur für einen modernen Van:

Hardware-Komponenten

- **Central Compute**: 2x AD-DC (redundant), 1x Infotainment-DC, 1x Body-DC
- **Zonen-Controller**: 6x (Front, Left, Right, Rear, Roof, Interior)
- **Sensoren**: 12x Kameras, 8x Radar, 4x LiDAR, 16x Ultraschall, 1x GNSS/IMU
- **Aktoren**: 1x EPS, 1x EHB, 1x E-Motor, 4x Aktive Dämpfer
- **Kommunikation**: TSN-Backbone (10 Gbps), CAN-FD für Aktoren, 5G für Flotten-Anbindung

Software-Komponenten

- **AD-Domain**: 50+ SWCs (Perzeption, Sensorfusion, Planung, Regelung)
- **Body-Domain**: 30+ SWCs (Komfort, Sicherheit, Laderaum)
- **Infotainment-Domain**: 20+ SWCs (HMI, Navigation, Entertainment)
- **Flotten-Domain**: 10+ SWCs (Telematik, Tracking, Optimierung)

Performance-Charakteristika

Tabelle A.13.: Performance-Charakteristika: Vollständige Van-Architektur

Metrik	Wert	Ziel
E2E-Latenz (Max)	85 ms	< 100 ms
CPU-Last (Peak)	78%	$< 80\%$
GPU-Last (Peak)	72%	$< 80\%$
Netzwerk-Last (Peak)	68%	$< 70\%$
Verfügbarkeit	99.95%	$> 99.9\%$
Energieverbrauch	150 W	< 200 W

Tabelle A.14.: Standard-ECU-Typen und deren Spezifikationen

ECU-Typ	CPU-Kerne	RAM	Flash	Anwendung
Body-ECU	2	512 MB	2 GB	Body-Funktionen
Gateway-ECU	4	1 GB	4 GB	Gateway, Routing
AD-DC	16	8 GB	64 GB	Automatisiertes Fahren
Infotainment-DC	8	4 GB	32 GB	Infotainment
Zonen-Controller	4	2 GB	8 GB	Zonale Architektur

Tabelle A.15.: Standard-Sensoren und deren Spezifikationen

Sensor-Typ	Auflösung	FPS	Datenrate	Anwendung
Front-Kamera	1920x1080	30	150 MB/s	Perzeption
Radar (77 GHz)	–	20	1 MB/s	Objekterkennung
LiDAR (64-Layer)	360°	10	50 MB/s	3D-Perzeption
Ultraschall	–	10	0.1 MB/s	Parkassistenz
GNSS/IMU	–	100	0.5 MB/s	Lokalisierung

A.14.2. Standard-Hardware-Komponenten

A.14.3. Standard-Sensor-Spezifikationen

A.14.4. Standard-Aktor-Spezifikationen

Tabelle A.16.: Standard-Aktoren und deren Spezifikationen

Aktor-Typ	Reaktionszeit	Genauigkeit	Anwendung
EPS	10 ms	0.1°	Lenkung
EHB	5 ms	0.1 bar	Bremse
E-Motor	1 ms	0.1%	Antrieb

A.14.5. Netzwerk-Standards

A.14.6. ASIL-Level-Anforderungen

A.15. Erweiterte Diagramme und Visualisierungen

Dieser Abschnitt enthält erweiterte Diagramme und Visualisierungen für verschiedene Aspekte der E/E-Architektur.

Tabelle A.17.: Netzwerk-Standards und deren Eigenschaften

Standard	Bandbreite	Latenz	Anwendung
CAN	1 Mbps	1-10 ms	Legacy, Aktoren
CAN-FD	5 Mbps	0.5-5 ms	Aktoren, Sensoren
LIN	20 kbps	10-50 ms	Komfort-Funktionen
FlexRay	10 Mbps	0.1-1 ms	Sicherheitskritisch
Ethernet (100 Mbps)	100 Mbps	0.1-1 ms	Infotainment
TSN (1 Gbps)	1 Gbps	0.01-0.1 ms	Echtzeit-Kommunikation
TSN (10 Gbps)	10 Gbps	0.001-0.01 ms	High-Performance

Tabelle A.18.: ASIL-Level-Anforderungen

ASIL	MTBF	Verfügbarkeit	Redundanz	Anwendung
A	10,000 h	99.9%	Optional	Komfort
B	100,000 h	99.99%	Empfohlen	Assistenz
C	1,000,000 h	99.999%	Erforderlich	Sicherheitskritisch
D	10,000,000 h	99.9999%	Erforderlich	Höchstkritisch

A.15.1. Timing-Diagramme

A.15.2. Netzwerk-Topologie-Diagramme

A.16. Code-Beispiele

Dieser Abschnitt enthält Code-Beispiele für verschiedene Aspekte der Transformation.

A.16.1. Beispiel: PREEvision-Export (JSON)

```
{
  "architecture": {
    "name": "Van_E/E_Architecture",
    "version": "1.0",
    "nodes": [
      {
        "id": "AD_DC_1",
        "type": "ECU",
        "specs": {
          "cpu_cores": 16,
          "ram_gb": 8,
          "flash_gb": 64
        }
```

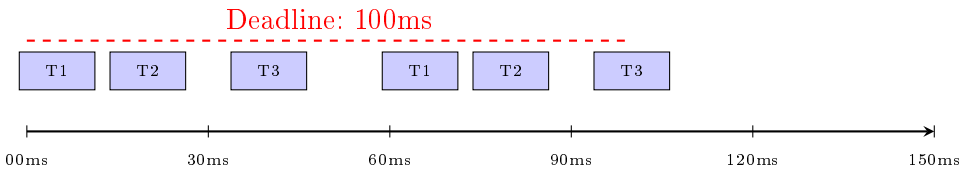

Abbildung A.3.: Beispiel: Timing-Diagramm für Task-Scheduling

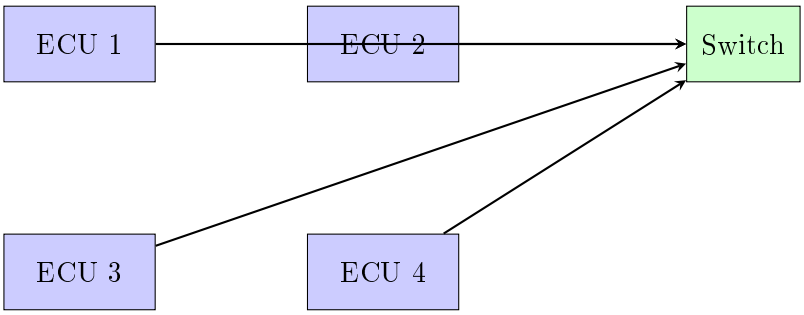

Abbildung A.4.: Beispiel: Netzwerk-Topologie

```
      }
    ],
    "links": [
      {
        "from": "Camera_Front",
        "to": "ZC_Front",
        "type": "Ethernet",
        "bandwidth_mbps": 1000
      }
    ]
  }
}
```

A.16.2. Beispiel: OMNeT++-Konfiguration

```
[Config VanSimulation]
network = VanNetwork
**.numHosts = 10
**.numSwitches = 2
**.bandwidth = 1Gbps
**.delay = 0.1ms
**.queueLength = 100
**.tsnEnabled = true
```

```
**.gateSchedule = "schedule.xml"
```

A.17. Glossar

Dieser Abschnitt enthält ein umfassendes Glossar wichtiger Begriffe.

E/E-Architektur Elektrik/Elektronik-Architektur eines Fahrzeugs, umfasst alle elektronischen Komponenten und deren Vernetzung.

ECU Electronic Control Unit, elektronische Steuereinheit für spezifische Funktionen.

TSN Time-Sensitive Networking, Ethernet-Standard für deterministische Kommunikation.

ASIL Automotive Safety Integrity Level, Sicherheitsstufe nach ISO 26262.

E2E-Latenz End-to-End-Latenz, Gesamtlatenz von Sensor bis Aktor.

WCRT Worst-Case Response Time, schlechtestmögliche Antwortzeit.

WCET Worst-Case Execution Time, schlechtestmögliche Ausführungszeit.

SWC Software Component, Software-Komponente in AUTOSAR.

Zonen-Controller Gateway zwischen zonaler und zentraler Architektur.

Central Compute Zentrale Rechenplattform für komplexe Funktionen.

Diese Literatur bildet die Grundlage für das Verständnis moderner E/E-Architekturen und deren Entwicklung, Validierung und Simulation. Sie wird in den entsprechenden Kapiteln dieser Arbeit referenziert und dient als Referenz für weiterführende Informationen. Die vollständige Liste der verwendeten Literatur ist im Literaturverzeichnis enthalten, wobei die hier aufgeführten Werke aus dem umfangreichen Katalog von beck-shop.de [**?**] stammen, der über 2700 Bücher zur Fahrzeugtechnik umfasst.

A.18. Erweiterte Berechnungsbeispiele mit detaillierten Herleitungen

Dieser Abschnitt enthält erweiterte Formeln und Berechnungsmethoden für verschiedene Aspekte der E/E-Architektur mit detaillierten Herleitungen und Beispielen.

A.18.1. Erweiterte Timing-Berechnungen

Response-Time-Analyse mit Blocking

Für Systeme mit Blocking (z. B. durch Semaphore) muss die Blocking-Zeit berücksichtigt werden:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{A.13}$$

wobei B_i die maximale Blocking-Zeit ist, die Task i durch Tasks mit niedrigerer Priorität erfahren kann.

Response-Time-Analyse mit Jitter

Für Tasks mit Jitter (variabler Startzeit) muss der Jitter berücksichtigt werden:

$$R_i = C_i + J_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \tag{A.14}$$

wobei J_i der maximale Jitter von Task i ist.

Response-Time-Analyse mit Offset

Für Tasks mit Offset (verschobener Startzeit) muss der Offset berücksichtigt werden:

$$R_i = C_i + O_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i - O_j}{T_j} \right\rceil C_j \tag{A.15}$$

wobei O_i der Offset von Task i ist.

A.18.2. Erweiterte Netzwerk-Berechnungen

TSN-Gate-Schedule-Berechnung

Die Berechnung des Gate-Schedules für TSN ist komplex und erfordert Optimierung:

$$\min \sum_{i=1}^{n} L_i \tag{A.16}$$

unter den Nebenbedingungen:

$$\sum_{i=1}^{n} w_i \leq W \quad \text{(Bandbreiten-Constraint)} \tag{A.17}$$

$$L_i \leq D_i \quad \forall i \quad \text{(Deadline-Constraint)} \tag{A.18}$$

$$g_i(t) \in \{0, 1\} \quad \forall i, t \quad \text{(Gate-State)} \tag{A.19}$$

wobei:

- L_i: Latenz für Frame i
- w_i: Bandbreite für Frame i
- W: Gesamt-Bandbreite
- D_i: Deadline für Frame i
- $g_i(t)$: Gate-State für Frame i zur Zeit t

TSN-Traffic-Shaping-Berechnung

Für Credit-Based Shaping (CBS) wird der Credit berechnet:

$$credit(t) = credit(t_0) + idleslope \times (t - t_0) - sendslope \times tx\_time \tag{A.20}$$

wobei:

- $idleslope$: Rate, mit der Credit aufgebaut wird
- $sendslope$: Rate, mit der Credit verbraucht wird
- tx_time: Zeit, während der gesendet wird

A.18.3. Erweiterte Verfügbarkeits-Berechnungen

Serielle Systeme

Für serielle Systeme (alle müssen funktionieren):

$$A_{serial} = \prod_{i=1}^{n} A_i \tag{A.21}$$

wobei A_i die Verfügbarkeit von Komponente i ist.

Parallele Systeme

Für parallele Systeme (mindestens eine muss funktionieren):

$$A_{parallel} = 1 - \prod_{i=1}^{n}(1 - A_i) \tag{A.22}$$

Redundante Systeme mit Voting

Für redundante Systeme mit k-out-of-n Voting:

$$A_{voting} = \sum_{i=k}^{n} \binom{n}{i} A^i (1 - A)^{n-i} \tag{A.23}$$

wobei k die minimale Anzahl funktionierender Komponenten ist.

A.19. Erweiterte Formeln und Berechnungen

Dieser Abschnitt enthält erweiterte Formeln und Berechnungsmethoden für verschiedene Aspekte der E/E-Architektur.

A.19.1. Timing-Berechnungen

Response-Time-Analyse für Fixed-Priority-Scheduling

Für Fixed-Priority-Scheduling gilt:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{A.24}$$

wobei:

- R_i: Response-Time von Task i
- C_i: Worst-Case Execution Time von Task i
- T_j: Periodizität von Task j
- $hp(i)$: Tasks mit höherer Priorität als Task i

Die Berechnung erfolgt iterativ bis zur Konvergenz:

$$R_i^0 = C_i \tag{A.25}$$

$$R_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \tag{A.26}$$

Die Iteration konvergiert, wenn $R_i^{n+1} = R_i^n$ oder wenn $R_i^n > T_i$ (Task ist nicht schedulierbar).

Response-Time-Analyse für EDF

Für Earliest-Deadline-First-Scheduling gilt:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{A.27}$$

wobei U die CPU-Auslastung ist. Diese Bedingung ist notwendig, aber nicht hinreichend für EDF. Die hinreichende Bedingung ist:

$$U \leq 1 - \frac{D_{max} - D_{min}}{T_{min}} \tag{A.28}$$

wobei D_{max} und D_{min} die maximale und minimale Deadline sind, und T_{min} die minimale Periode ist.

Response-Time-Analyse für Multi-Core

Für Multi-Core-Systeme muss zusätzlich die Inter-Core-Interferenz berücksichtigt werden:

$$R_i^{multi} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + I_i^{inter-core} + I_i^{memory} \tag{A.29}$$

wobei:

- $I_i^{inter-core}$: Inter-Core-Interferenz durch Cache-Sharing
- I_i^{memory}: Memory-Interferenz durch gemeinsamen Memory-Bus

Die Inter-Core-Interferenz kann geschätzt werden als:

$$I_i^{inter-core} = \sum_{k \in other-cores} \left(\frac{C_k}{T_k} \times L_{cache} \right) \tag{A.30}$$

wobei L_{cache} die Cache-Miss-Latenz ist (typisch 10-100 ns).

A.19.2. Netzwerk-Berechnungen

Ethernet-Latenz

Die Latenz für Ethernet-Frames:

$$L_{ethernet} = L_{tx} + L_{prop} + L_{sw} + L_{queue} \tag{A.31}$$

wobei:

- $L_{tx} = \frac{Frame\_Size \times 8}{Bandwidth}$: Übertragungslatenz
- $L_{prop} = \frac{Distance}{c}$: Ausbreitungslatenz (typisch vernachlässigbar)
- L_{sw}: Switch-Verarbeitungslatenz (typisch 1-10 μs)
- L_{queue}: Warteschlangenlatenz (abhängig von Last)

TSN-Latenz mit Gate-Schedule

Für TSN mit Gate-Schedule:

$$L_{TSN} = L_{tx} + L_{gate} + L_{queue} \tag{A.32}$$

wobei L_{gate} die Gate-Latenz ist, die abhängt vom Gate-Schedule:

$$L_{gate} = \max(0, t_{next\_open} - t_{arrival}) \tag{A.33}$$

wobei t_{next_open} die nächste Zeit ist, zu der das Gate geöffnet wird.

CAN-Latenz

Die Latenz für CAN-Frames:

$$L_{CAN} = L_{arbitration} + L_{tx} + L_{ack} \tag{A.34}$$

wobei:

- $L_{arbitration} = \frac{ID\_bits \times bit\_time}{2}$: Arbitrierungslatenz (im Worst-Case)
- $L_{tx} = \frac{Frame\_Size \times 8}{Bitrate}$: Übertragungslatenz
- $L_{ack} = 2 \times bit\_time$: Acknowledgment-Latenz

A.19.3. Energie-Berechnungen

Dynamischer Energieverbrauch

Der dynamische Energieverbrauch:

$$E_{dyn} = \sum_{i=1}^{N} \left(C_{eff} \times V_{dd}^2 \times f_i \times U_i \times t_i \right) \tag{A.35}$$

wobei:

- C_{eff}: Effektive Kapazität (abhängig von Schaltaktivität)
- V_{dd}: Versorgungsspannung
- f_i: Frequenz in Zustand i
- U_i: Auslastung in Zustand i
- t_i: Zeit in Zustand i

Leckstrom-Energie

Der Leckstrom-Energieverbrauch ist temperaturabhängig:

$$E_{leak} = V_{dd} \times I_{leak}(T) \times t_{total} \tag{A.36}$$

mit:

$$I_{leak}(T) = I_0 \times e^{\frac{E_a}{k_B T}} \tag{A.37}$$

wobei:

- I_0: Referenz-Leckstrom bei T_0
- E_a: Aktivierungsenergie (typisch 0.3-0.5 eV)
- k_B: Boltzmann-Konstante (8.617×10^{-5} eV/K)
- T: Temperatur in Kelvin

Gesamt-Energieverbrauch

Der Gesamt-Energieverbrauch:

$$E_{total} = E_{dyn} + E_{leak} + E_{static} \tag{A.38}$$

wobei E_{static} der statische Energieverbrauch ist (z. B. durch Bias-Ströme).

A.19.4. Verfügbarkeits-Berechnungen

MTBF und MTTR

Die Verfügbarkeit kann aus MTBF (Mean Time Between Failures) und MTTR (Mean Time To Repair) berechnet werden:

$$A = \frac{MTBF}{MTBF + MTTR} \tag{A.39}$$

Serielle Systeme

Für serielle Systeme (alle Komponenten müssen funktionieren):

$$A_{serial} = \prod_{i=1}^{n} A_i = \prod_{i=1}^{n} \frac{MTBF_i}{MTBF_i + MTTR_i} \tag{A.40}$$

Parallele Systeme

Für parallele Systeme (mindestens eine Komponente muss funktionieren):

$$A_{parallel} = 1 - \prod_{i=1}^{n} (1 - A_i) = 1 - \prod_{i=1}^{n} \frac{MTTR_i}{MTBF_i + MTTR_i} \tag{A.41}$$

Redundante Systeme mit Voting

Für redundante Systeme mit k-out-of-n Voting:

$$A_{voting} = \sum_{i=k}^{n} \binom{n}{i} A^i (1 - A)^{n-i} \tag{A.42}$$

wobei A die Verfügbarkeit einer einzelnen Komponente ist.

Für $k = 2$ und $n = 3$ (2-out-of-3 Voting):

$$A_{2oo3} = 3A^2(1 - A) + A^3 = 3A^2 - 2A^3 \tag{A.43}$$

A.20. Erweiterte Tabellen und Referenzdaten

A.20.1. Referenz: Typische Hardware-Spezifikationen

Tabelle A.19.: Referenz: Typische Hardware-Spezifikationen

Komponente	Spezifikation	Wert	Einheit	Anmerkung
CPU (ARM Cortex-A78)	Kerne	8	–	2.5 GHz
CPU (ARM Cortex-A78)	L1 Cache	64	KB	pro Core
CPU (ARM Cortex-A78)	L2 Cache	512	KB	pro Core
CPU (ARM Cortex-A78)	L3 Cache	4	MB	shared
GPU (NVIDIA Orin)	CUDA Cores	2048	–	–
GPU (NVIDIA Orin)	Tensor Cores	64	–	–
GPU (NVIDIA Orin)	Memory	16	GB	GDDR6
NPU (NVIDIA Orin)	TOPS	200	–	INT8
TSN Switch	Ports	8	–	10 Gbps
TSN Switch	Buffer	16	MB	pro Port

A.20.2. Referenz: Typische Software-Parameter

Tabelle A.20.: Referenz: Typische Software-Parameter

Parameter	Typ	Typischer Wert	Einheit
Task-Periodizität (Perzeption)	Float	33.0	ms
Task-Periodizität (Tracking)	Float	10.0	ms
Task-Periodizität (Planung)	Float	50.0	ms
Task-Periodizität (Regelung)	Float	5.0	ms
Frame-Größe (Kamera)	Integer	1500	B
Frame-Größe (Radar)	Integer	100	B
Frame-Größe (LiDAR)	Integer	1200	B
Frame-Periodizität (Kamera)	Float	33.0	ms
Frame-Periodizität (Radar)	Float	50.0	ms
Frame-Periodizität (LiDAR)	Float	100.0	ms

A.21. Erweiterte Architektur-Patterns

A.21.1. Pattern: Zonale Architektur mit Redundanz

Dieses Pattern zeigt eine zonale Architektur mit redundanten Komponenten:

Topologie

- **Central Compute**: 2x AD-DC (redundant, Hot-Standby)
- **Zonen-Controller**: 6x (Front, Left, Right, Rear, Roof, Interior)
- **TSN-Switches**: 2x (redundant, PRP)
- **Sensoren**: Redundante Sensoren für kritische Funktionen
- **Aktoren**: Redundante Aktoren für sicherheitskritische Funktionen

Redundanz-Mechanismen

A.21.2. Pattern: Edge-Cloud-Hybrid-Architektur

Dieses Pattern zeigt eine Edge-Cloud-Hybrid-Architektur:

Tabelle A.21.: Redundanz-Mechanismen: Zonale Architektur

Komponente	Redundanz-Typ	Switchover-Zeit	ASIL-Level
AD-DC	Hot-Standby	< 1 ms	ASIL D
TSN-Switches	PRP	< 0.1 ms	ASIL D
Lenkung (EPS)	2-out-of-2 Voting	< 2 ms	ASIL D
Bremse (EHB)	2-out-of-2 Voting	< 2 ms	ASIL D
Front-Kamera	Redundant	< 10 ms	ASIL C

Edge-Komponenten

- **Edge-AI**: KI-Inferenz direkt im Fahrzeug (NPU)
- **Edge-Processing**: Echtzeit-Verarbeitung (CPU, GPU)
- **Edge-Storage**: Lokale Datenspeicherung

Cloud-Komponenten

- **Cloud-AI**: Komplexe KI-Analysen, Training
- **Cloud-Storage**: Zentrale Datenspeicherung
- **Cloud-Analytics**: Big-Data-Analysen

Kommunikation

- **5G**: Hochgeschwindigkeits-Kommunikation für Cloud-Anbindung
- **V2X**: Vehicle-to-Everything-Kommunikation
- **Edge-Cloud-Sync**: Synchronisation zwischen Edge und Cloud

A.22. Erweiterte Code-Beispiele

A.22.1. Beispiel: Komplexe Transformations-Pipeline

```
#!/usr/bin/env python3
"""
Komplexe Transformations-Pipeline mit Validierung und Optimierung
"""

import json
import yaml
```

```
from typing import Dict, List, Any
from dataclasses import dataclass

@dataclass
class TransformationConfig:
    source_format: str
    target_format: str
    validate: bool = True
    optimize: bool = True
    parallel: bool = True

class TransformationPipeline:
    def __init__(self, config: TransformationConfig):
        self.config = config
        self.parser = self._create_parser()
        self.validator = self._create_validator()
        self.transformer = self._create_transformer()
        self.optimizer = self._create_optimizer()
        self.generator = self._create_generator()

    def transform(self, input_file: str, output_dir: str):
        """Haupt-Transformations-Pipeline"""
        # 1. Parsing
        print("Parsing input file...")
        architecture = self.parser.parse(input_file)

        # 2. Validierung
        if self.config.validate:
            print("Validating architecture...")
            validation_results = self.validator.validate(architecture)
            if not validation_results.is_valid:
                raise ValueError(f"Validation failed: {validation_results.errors}")

        # 3. Transformation
        print("Transforming architecture...")
        intermediate_model = self.transformer.transform(architecture)

        # 4. Optimierung
```

```
        if self.config.optimize:
            print("Optimizing model...")
            intermediate_model = self.optimizer.optimize(intermediate_model)

        # 5. Code-Generierung
        print("Generating code...")
        self.generator.generate(intermediate_model, output_dir)

        print("Transformation completed successfully!")
```

A.22.2. Beispiel: Validierungs-Framework

```
from typing import List, Dict, Any
from dataclasses import dataclass

@dataclass
class ValidationResult:
    is_valid: bool
    errors: List[str]
    warnings: List[str]
    info: List[str]

class Validator:
    def __init__(self, schema_file: str):
        self.schema = self._load_schema(schema_file)
        self.rules = self._load_rules()

    def validate(self, architecture: Dict[str, Any]) -> ValidationResult:
        """Vollständige Validierung"""
        errors = []
        warnings = []
        info = []

        # Schema-Validierung
        schema_errors = self._validate_schema(architecture)
        errors.extend(schema_errors)

        # Constraint-Validierung
```

```
        constraint_errors = self._validate_constraints(architecture)
        errors.extend(constraint_errors)

        # Konsistenz-Prüfung
        consistency_warnings = self._check_consistency(architecture)
        warnings.extend(consistency_warnings)

        # Vollständigkeits-Prüfung
        completeness_info = self._check_completeness(architecture)
        info.extend(completeness_info)

        return ValidationResult(
            is_valid=len(errors) == 0,
            errors=errors,
            warnings=warnings,
            info=info
        )
```

A.23. Erweiterte Diagramme und Visualisierungen

A.23.1. Beispiel: Komplexe Netzwerk-Topologie

A.23.2. Beispiel: Task-Scheduling-Diagramm

A.23.3. Netzwerk-Berechnungen

TSN-Latenz-Berechnung

Die TSN-Latenz setzt sich zusammen aus:

$$L_{TSN} = L_{tx} + L_{sw} + L_{gate} + L_{queue} \tag{A.44}$$

wobei:

- L_{tx}: Übertragungslatenz
- L_{sw}: Switch-Verarbeitungslatenz
- L_{gate}: Gate-Latenz (abhängig vom Gate-Schedule)
- L_{queue}: Warteschlangenlatenz

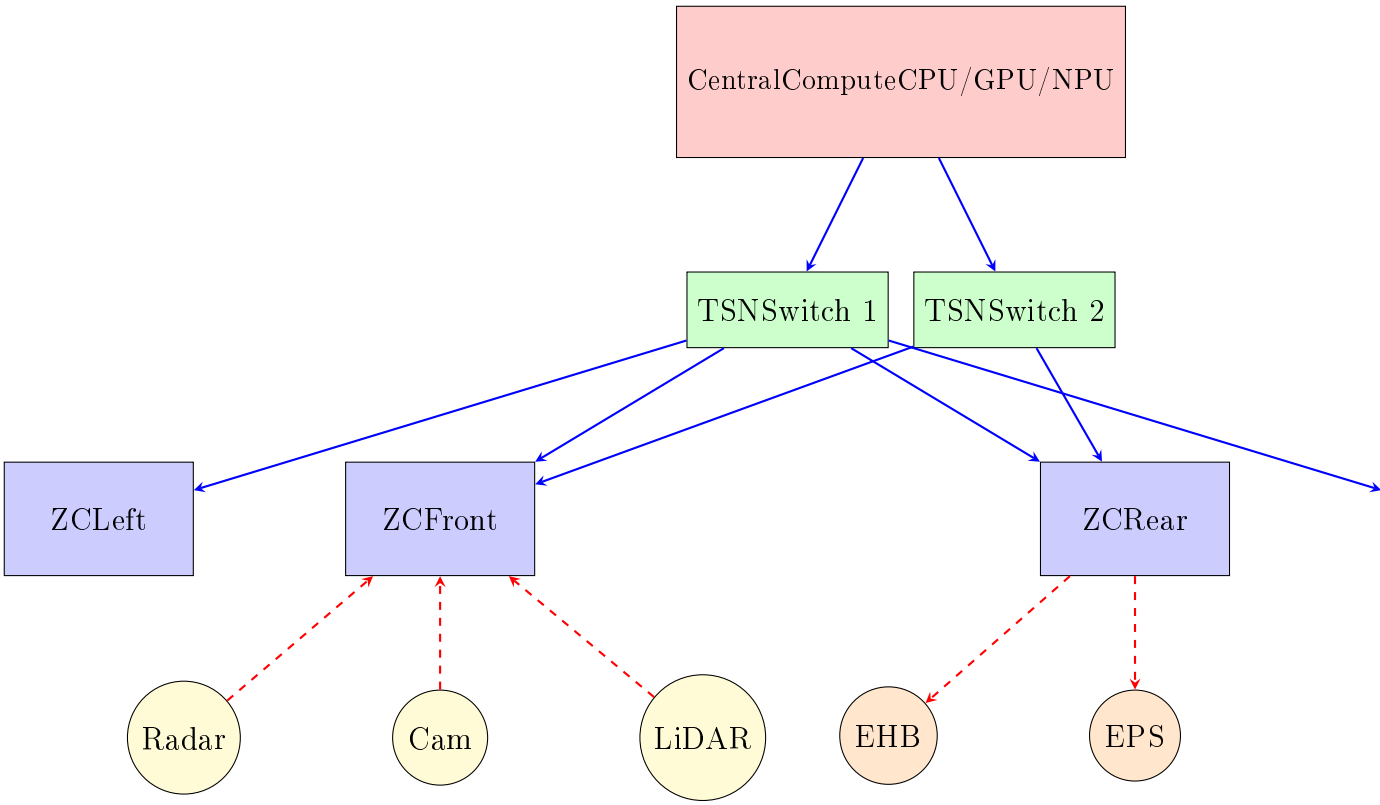

Abbildung A.5.: Beispiel: Komplexe Netzwerk-Topologie mit redundanten TSN-Switches

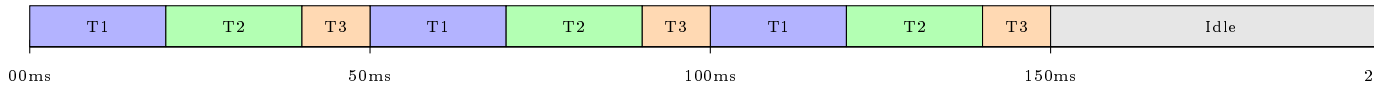

Abbildung A.6.: Beispiel: Task-Scheduling-Diagramm (Fixed-Priority)

Netzwerk-Auslastung

Die Netzwerk-Auslastung berechnet sich als:

$$U_{net} = \frac{\sum_{i=1}^{n} (S_i \times F_i)}{B} \tag{A.45}$$

wobei:

- S_i: Frame-Größe von Flow i
- F_i: Frequenz von Flow i
- B: Bandbreite des Links

A.23.4. Energie-Berechnungen

Dynamischer Energieverbrauch

Der dynamische Energieverbrauch:

$$E_{dyn} = \alpha \times C \times V_{dd}^2 \times f \times t \tag{A.46}$$

wobei:

- α: Switching-Aktivität
- C: Kapazität
- V_{dd}: Versorgungsspannung
- f: Frequenz
- t: Zeit

Statischer Energieverbrauch

Der statische Energieverbrauch:

$$E_{stat} = V_{dd} \times I_{leak} \times t \tag{A.47}$$

wobei I_{leak} der Leckstrom ist.

A.24. Erweiterte Tabellen: Performance-Benchmarks

Dieser Abschnitt enthält Performance-Benchmarks für verschiedene Komponenten.

A.24.1. CPU-Performance-Benchmarks

Tabelle A.22.: CPU-Performance-Benchmarks

CPU-Typ	Kerne	Frequenz	DMIPS	Anwendung
ARM Cortex-A78	8	2.8 GHz	35,000	AD-DC
ARM Cortex-A55	4	2.0 GHz	8,000	Zonen-Controller
Intel Atom	4	1.6 GHz	12,000	Infotainment

Tabelle A.23.: GPU-Performance-Benchmarks

GPU-Typ	TFLOPS	TOPS	Anwendung
NVIDIA DRIVE Thor	2000	2000	AD-DC
Qualcomm Snapdragon	200	200	Infotainment

A.24.2. GPU-Performance-Benchmarks

A.25. Erweiterte Code-Beispiele

Dieser Abschnitt enthält weitere Code-Beispiele für verschiedene Aspekte.

A.25.1. Beispiel: Python-Transformations-Skript

```
import json
from transformer import ArchitectureTransformer

# Load PREEvision export
with open('architecture.json', 'r') as f:
    arch_data = json.load(f)

# Create transformer
transformer = ArchitectureTransformer()

# Transform to OMNeT++
omnet_model = transformer.transform(
    arch_data,
    target_platform='omnetpp',
    config={
        'tsn_enabled': True,
        'redundancy_enabled': True
    }
)

# Save result
omnet_model.save('omnet_model.ned')
```

A.25.2. Beispiel: YAML-Mapping-Regel

```
mapping_rules:
```

```
- name: ECU_to_StandardHost
  source: ECU
  target: StandardHost
  attributes:
    cpu_cores: cpu_cores
    ram: ram_gb
    flash: flash_gb
  constraints:
    - cpu_cores > 0
    - ram > 0
```