



Bachelorarbeit

Bachelor Thesis

**Entwicklung eines Frameworks für die
automatisierte Architekturvalidierung**
Development of a framework for automated architecture
validation

Rushanth Rasaratnam
Matrikelnummer: 367117

Aachen, 31. Juli 2025

Gutachter:
Prof. Dr.-Ing. Stefan Kowalewski
Prof. Dr. rer. nat. Bernhard Rumpe

Betreuer:
David Klüner M.Sc.

Diese Arbeit wurde vorgelegt am
Lehrstuhl Informatik 11 – Embedded Software

This is to be done

Zusammenfassung

An abstract is basically a very short summary of the thesis topic. It describes short and precise the content of the thesis and what has been achieved. An abstract should not contain any discussion, quotation or reference to figures, chapters and so on. Please note that if you write your thesis in german, there should be a german and english version of the abstract. Although an abstract should generally not contain examples we provide here an example abstract for this document:

There exists a large variety of latex templates such as guidelines explaining or demonstrating the common structuring, dos and donts of bachelor and master theses. This document aims at providing an example for a typical structure of such theses containing useful knowledge about scientific writing and providing students with a correspondent latex template for bachelor or master theses at i11. Moreover, it gives some advice on very basic latex functionality for citation and figure handling.

Eidesstattliche Versicherung

Rasaratnam, Rushanth

367117

Name, Vorname

Matrikelnummer

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Bachelorarbeit mit dem Titel

Entwicklung eines Frameworks für die automatisierte Architekturvalidierung

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

- (1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
- (2) Strafflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum

Unterschrift

Inhaltsverzeichnis

1. Grundlagen	1
1.1. Automobile E/E-Architektur	1
1.2. Softwarearchitekturen in Fahrzeugen	3
1.2.1. Serviceorientierte Architektur	3
1.2.2. Automobile Middleware	4
1.3. Validierung von Architekturen	6
1.4. Webentwicklungs-Stack	7
2. Stand der Technik	9
2.1. Etablierte Tools und Plattformen	9
2.1.1. Vector PREEvision	9
2.1.2. Mathworks Simulink + System Composer	10
2.1.3. Eclipse Capella	11
2.2. Validierungsverfahren im Detail	12
3. Konzept & Design	15
3.1. Anforderungsanalyse	15
3.2. Systemarchitektur	17
3.3. API zur Algorithmenverwaltung	18
3.3.1. CREATE	19
3.3.2. READ	19
3.3.3. UPDATE	19
3.3.4. DELETE	20
3.4. Grafische Benutzeroberfläche	20
3.4.1. Algorithmenverwaltung	20
3.4.2. (Ergebnis-)Visualisierung + Verlauf	21
3.5. Validierungs-Engine	22
4. Implementierung	25
4.1. Backend	25
4.1.1. Umsetzung der API	25
4.1.2. Integrierung der Validierungs-Engine	27
4.2. Frontend	28
4.2.1. Algorithmenverwaltungs-Komponenten	28

Inhaltsverzeichnis

4.2.2. Visualisierung der Ergebnisse	30
4.3. Validierungsalgorithmus	31
Literaturverzeichnis	33
A. Appendix	37

Tabellenverzeichnis

2.2. Auflistung der Architekturvalidierungen nach Tool	13
3.2. Anforderungen an das Framework	16

Abbildungsverzeichnis

1.1. Vergleich zwischen funktionsorientierter (Domäne) und positionsorientierter (Zone) E/E-Architektur(Nachgezeichnet von [WGC ⁺ 24]) .	2
1.2. Schematische Darstellung einer serviceorientierten Middleware-Architektur (Nachzeichnung von [NDN06])	5
2.1. Schematische PREEvision Schichtenarchitektur. (Nachzeichnung von [Vec])	10
2.2. Verknüpfung einer System Composer Architektur mit einem Simulink Verhaltensmodell. (Entnommen aus [Cha20])	11
2.3. Das Arcadia-Vorgehensmodell in Capella als vier übereinander gestapelte Architekturebenen (Entnommen aus [let])	12
3.1. Blockdiagramm des ArchitekturTool's	17
A.1. Python-Codegerüst für die Validierungsalgorithmen	38

1. Grundlagen

Dieses Kapitel führt in die für das Framework relevanten Konzepte und Technologien ein. Zunächst wird der Begriff der elektrischen/elektronischen Architektur (E/E-Architektur) samt ihrer Evolution und ihres Aufbaus erläutert. Anschließend wird die Softwarearchitektur behandelt: Nach der Definition grundlegender Begriffe werden die Prinzipien der Serviceorientierten Architektur vorgestellt und darauf aufbauend die Rolle von Middleware in der Fahrzeugtechnik beleuchtet. Ein weiterer Abschnitt erläutert die Motivation, Ziele und Methoden zur Validierung von Architekturen. Den Abschluss bildet die Vorstellung des für die Implementierung verwendeten Webentwicklungs-Stacks.

1.1. Automobile E/E-Architektur

Für alle softwaregesteuerten Funktionen innerhalb eines Fahrzeuges bildet die E/E-Architektur das physikalische und logische Fundament. Sie definiert die grundlegende Organisation von Hardwarekomponenten wie Electronic Control Unit (ECU), Sensoren, Energieversorgungssystemen und Kabelbäumen und steuert deren Kommunikation und Interaktion, um die erwarteten Funktionen zu realisieren [Jia19]. Diese Architektur hat im Laufe der Zeit mehrere Varianten durchlaufen und wird stetig weiterentwickelt. Im Folgenden werden die evolutionären Stufen der E/E-Architektur genauer betrachtet.

Evolution der Architekturentwicklung

Ursprünglich waren Fahrzeuge hauptsächlich mechanische Produkte, was sich jedoch mit der Einführung von Elektronik, wie Motorsteuergeräte und Airbags, änderte und die Ära der Elektrifizierung innerhalb der Automobilindustrie begann [Jia19][ZZL⁺21]. Zunächst waren, aufgrund der geringen Anzahl, die Verbindungen zwischen den elektronischen Komponenten meist Punkt-zu-Punkt, sprich Komponenten wurden direkt miteinander verbunden. Dies führte jedoch durch die steigende Anzahl an ECUs, zu einem Anstieg der Komplexität und des Kabelbaums [WGC⁺24]. Um die Kommunikationseffizienz zu verbessern und die Komplexität zu reduzieren, wurden Controller Area Network (CAN)-Busse eingeführt. In der Folge wurde auch an Technologien wie Local Interconnect Network (LIN) und FlexRay entwickelt und eingesetzt, um

1. Grundlagen

die verschiedenen Kommunikationsanforderungen zu erfüllen, sodass Komponenten innerhalb einer Domäne Informationen gemeinsam nutzen [WGC⁺24][ZZL⁺21].

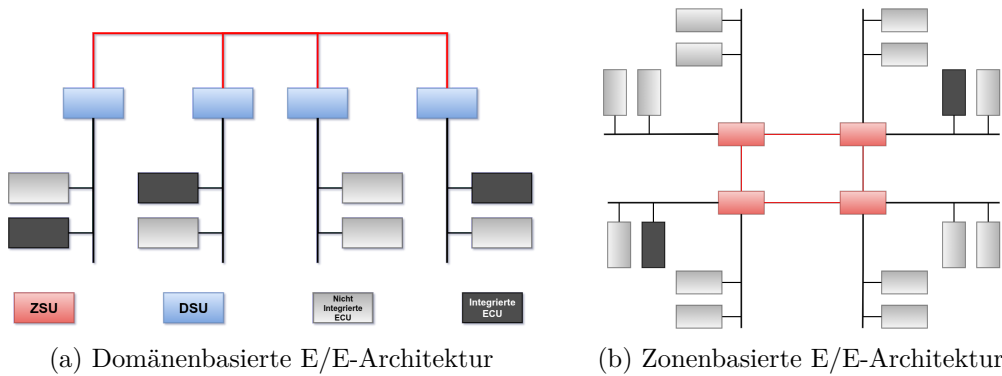


Abbildung 1.1.: Vergleich zwischen funktionsorientierter (Domäne) und positionsorientierter (Zone) E/E-Architektur (Nachgezeichnet von [WGC⁺24])

Das Konzept der Funktionsdomäne rückte, aufgrund der wachsenden Komplexität und Funktionen immer stärker in den Vordergrund. Wie Abbildung 1.1a veranschaulicht, ECUs und Funktionen wurden in Domänen wie Fahrwerk, Karosserie und Infotainment unterteilt [Jia19]. Durch die Nachfrage nach domänenübergreifender Kommunikation wurde die Gateway-Funktion, welche zuvor von einzelnen ECUs übernommen wurde, zunehmend in einem dedizierten, zentralen Gateway gebündelt [Jia19][ZZL⁺21]. Obwohl dieser Ansatz die Mainstream-Lösung war, führte die Zunahme der Funktionen und Rechenanforderungen zu einer starken Zunahme der ECUs. Dies hatte komplexe Kabelbäume mit erhöhtem Gewicht und Kosten zur Folge, außerdem konnte der Ausfall durch das Ausfallen des zentralen Gateways zu einem Zusammenbruch des gesamten Netzwerks führen.

Um diesem Problem entgegenzuwirken, wurden Domain Control Units (DCUs) eingeführt, welche Funktionen einiger ECUs innerhalb einer jeweiligen Funktionsdomäne integrieren und somit die Anzahl der ECUs und die Belastung der Gateways reduzieren [BSPL21][Jia19][WGC⁺24]. Um die bestmögliche Kommunikation zu garantieren, werden DCUs über das Automotive Ethernet mit dem zentralen Gateway verbunden, welches eine höhere Bandbreite bietet.

Ein weiterer Ansatz, der neben der domänenbasierten Architektur existiert, ist die zonenbasierte Architektur. Um Komplexität und das Gewicht der Verkabelung zu reduzieren, werden Komponenten anstatt nach Funktionen, basierend auf ihrer physischen Position innerhalb des Fahrzeuges unterteilt [WGC⁺24][Jia19]. Abbildung 1.1b zeigt diesen Ansatz, bei dem Zone Control Units (ZCUs) die Funktionen übernehmen, die in den entsprechenden physischen Zonen stattfinden. Auch hier wird die Kommunikation über das Automotive Ethernet durchgeführt. Dieser Ansatz

reduziert die Anzahl der ECUs und Kabelaufwand (Anzahl, Länge, Gewicht) und verringert so die Komplexität der E/E-Architektur deutlich.

Erst die Evolution der E/E-Architektur ermöglicht moderne Softwarekonzepte. Das folgende Unterkapitel beschreibt daher die Softwarearchitekturen, die diese neue Hardware-Basis nutzt.

1.2. Softwarearchitekturen in Fahrzeugen

Die allgemeine Definition einer Softwarearchitektur ist nach Bass et al. [BCK21] eine systematische Gliederung eines Softwaresystems in einzelne Bausteine, die Festlegung ihrer Schnittstelle sowie die Prinzipien, nach denen diese Komponenten zusammenwirken und organisiert sind. Seitdem sich der Schwerpunkt von Fahrzeugen weg von mechanikzentrierten Systemen bewegt hat und hin zu intelligenten, softwaredefinierten Plattformen, gibt es innerhalb der Automobilindustrie einen großen Fortschritt bezüglich der Softwarefunktionalität [RGKS20][Bro06]. Um bei diesem Fortschritt den Überblick zu behalten, ist eine fundierte Softwarearchitektur unerlässlich.

Zu Beginn setzte man auf klassische, geschichtete Architekturen wie AUTOSAR Classic, jedoch merkte man mit dem Aufkommen des Konzepts eines Software-Defined Vehicle (SDV) sowie der steigenden Software-Komplexität, dass die klassische Softwarearchitektur an ihre Grenzen stieß [RGKS20]. Es gibt einige Gründe, wie die starre, monolithische Schichtenstruktur: Funktionen sind fest an Schichten gebunden, was Anpassungen einzelner Komponenten oder den Austausch von Modulen kaum umsetzbar machte.

In diesem Unterkapitel wird ein Einblick in die Service-orientierte Architekturen (SOAs) gegeben, die beschriebenen Schwächen der klassischen Softwarearchitektur überwinden. Anschließend wird die Middleware erläutert, die als technische Vermittlungsschicht dient.

1.2.1. Serviceorientierte Architektur

Die Notwendigkeit, Softwaremodule dynamisch bereitzustellen erfordert einen flexiblen Ansatz. Hier setzt die SOA, welche bereits in der Webentwicklung fester Bestandteil ist, an. Nach Rumez et al.[RGKS20] ein Architekturmuster, bei dem Anwendung als lose gekoppelte, wiederverwendbare Dienste organisiert werden. Jeder Dienst bietet eine klar definierte Funktionalität über gut beschriebene Schnittstellen und kann zur Laufzeit von beliebigen Clients zugegriffen werden.

Im Gegensatz zum klassischen signalorientierten Ansatz, bei dem die Konfiguration der Kommunikationspfade vollständig statisch zur Design-Zeit erfolgt, setzt der SOA-Ansatz auf eine dynamische Konfiguration während der Laufzeit [RGKS20]. Durch das dynamische Aufrufen von Diensten zur Laufzeit erhält die Architektur mehr Flexibilität. So lassen sich Funktionen einfacher hinzufügen, entfernen oder

bearbeiten, ohne dass eine komplette Neuzuweisung der Software erforderlich ist. Dank der Middleware-Schicht, auf die im nächsten Abschnitt näher eingegangen wird, sind die verschiedenen Anwendungen unabhängig von dem Steuergerät, auf dem ein Dienst läuft, sowie vom zugrunde liegenden Netzwerk aufgebaut [KOS21]. Diese Entkoppelung von der Plattform ermöglicht es der Software, wiederverwendbar und portierbar zu sein [CK20]. Ein weiteres Merkmal des SOA-Ansatzes sind Kommunikationsmuster, die Daten nur bei Bedarf übertragen [RGKS20]. Im Gegensatz dazu verschickt die klassische Softwarearchitektur in zyklischen Abständen Daten, selbst wenn kein Empfänger vorhanden ist. Dadurch kann die entstehende Netzwerklast auf das Nötigste reduziert werden, was die Effizienz steigert. Darüber hinaus kapselt jeder Dienst hinter einer Schnittstelle eine klar definierte Funktionalität. So lassen sich diese Dienste in unterschiedliche Kontexte und Projekten wiederverwenden, wodurch der Entwicklungs- und Testaufwand gesenkt wird [RGKS20][CK20]. Ein weiterer Vorteil ist die Integration externer und On-Demand-Dienste: SOA unterstützen die dynamische Anbindung von Backend- und Cloud-Diensten, welche weitgehend transparent ist [RGKS20][HSS⁺22]. Außerdem lassen sich SOA an wachsenden Anforderungen anpassen ohne dass das Gesamte System neu zu strukturieren. Aus den genannten Gründen erweist sich die SOA als flexiblere und anpassungsfähiger als die klassische Variante der Softwarearchitektur.

Um die Vorteile der SOA realisieren zu können benötigt es eine Zwischenschicht zwischen Anwendung und Netz. Im folgenden Abschnitt wird die Middleware-Schicht untersucht, welche die SOA-Prinzipien technisch umsetzt. Dabei wird zunächst auf Aufbau und Funktionen eingegangen.

1.2.2. Automobile Middleware

Innerhalb moderner, verteilter Softwaresysteme ist die Middleware-Schicht eine wichtige Vermittlungsebene. Neely et al. [NDN06] definieren die Middleware als Schicht zwischen Anwendungssoftware und Systemsoftware. In Systemen mit Netzwerkanbindung vermittelt sie zusätzlich zwischen Anwendungssoftware, Betriebssystem und Netzwerkkommunikationsschichten. Dabei liegt ihre Hauptaufgabe darin, die Komplexität des zugrunde liegenden Systems zu abstrahieren. Im Bereich der Fahrzeugtechnik wird die Middleware noch einmal in zwei Sub-Schichten unterteilt [KMK⁺24]:

- Kommunikations-Middlewares (untere Schicht), die den Datenaustausch zwischen ECUs organisieren und Quality-of-Service (QoS)-Funktionen bereitstellen.
- Architekturplattformen (obere Schicht), die umfassende Frameworks für die Entwicklung und Bereitstellung automobiler Softwaresysteme liefern.

Nun wird untersucht, wie die Kommunikations-Middlewares und Architekturplattformen konkret arbeiten.

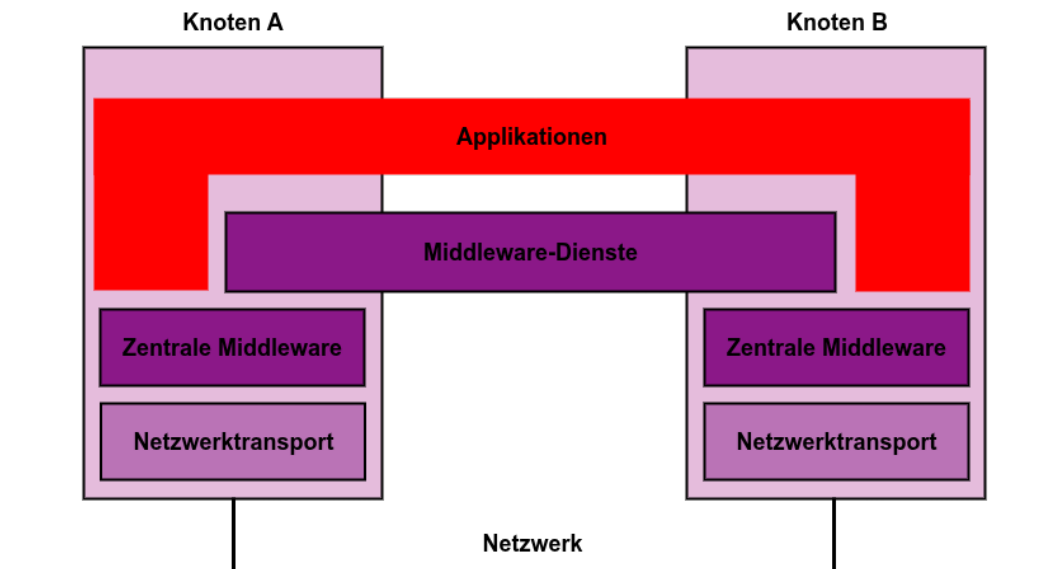


Abbildung 1.2.: Schematische Darstellung einer serviceorientierten Middleware-Architektur (Nachzeichnung von [NDN06])

Als untere Sub-Schicht sorgt die Kommunikations-Middleware die in Abbildung 1.2 aus der Zentrale-Middleware und Netzwerktransport besteht, dafür, dass sämtliche Details des Datenaustauschs im In-Vehicle Network (IVN) abstrahiert und Anwendungen von der zugrunde liegenden Netzwerktopologie entkoppelt sind. Sie sorgt zudem dafür, dass strukturierte Daten wie z.B. Sensormesswerte automatisch serialisiert und über Ethernet, CAN-Gateway oder Shared-Memory-Kanäle verteilt werden. Außerdem ermöglicht sie die Definition und Einhaltung von QoS-Parametern. Des Weiteren sorgt ein intelligentes Routing- und Topologie-Management auf der Netzwerk-Ebene dafür, dass Nachrichten den bestmöglichen Pfad für die Übertragung wählen, während ein Real-Time-Scheduler die Übertragungen koordiniert.

Mit der Kommunikations-Middleware als Basis, löst die Architekturplattform Herausforderungen in Bezug auf die Entwicklung von Automobil-Softwaresystemen. Sie stellt eine umfassende Laufzeitumgebung bereit und bietet Entwicklern standardisierte Dienste und APIs. Das Ziel ist es die bestehende Komplexität weiter zu verringern, sodass Entwickler den Fokus auf die Anwendungslogik setzen können, anstatt sich mit systemnahen Aufgaben zu beschäftigen. Abgesehen von der Kommunikation, gehört zunächst das umsetzen der Software-Designmustern, wie das zuvor angesprochene SOA-Prinzip, zu den Kernaufgaben. Ebenso Ressourcenmanagement und Orchestrierung, bei der die Ressourcen von inaktiven Diensten freigegeben werden. Das Ziel hierbei ist eine situationsabhängige optimale Ressourcenzuweisung. Als nächstes ist

die deterministische Ausführung sowie die Unterstützung bei der Einhaltung von Echtzeitanforderung zu nennen, um Verzögerungen im Softwaresystem zu vermeiden. Des Weiteren bietet sie oft Funktionen zur Cybersicherheit, wie Authentifizierung, Zugriffskontrolle, Verschlüsselung etc. an um Schwachstellen innerhalb des IVNs zu reduzieren. Darüber hinaus sorgt die obere Schicht dafür, dass Over-the-Air-Updates, Installation/Deinstallation von Software-Paketen, sowie die Konfiguration von Anwendungen während der Laufzeit durchgeführt werden und bietet dementsprechend Funktionen an. Eine weitere wichtige Aufgabe ist die Verwaltung des Systemzustands und die Überwachung des Lebenszyklus von Softwarekomponenten. Nicht zuletzt bietet die Architekturplattform Entwicklungstools an, welche für die Entwicklung und Bereitstellung verwendet werden dargestellt

1.3. Validierung von Architekturen

Wie bereits zu Beginn der Arbeit dargelegt, ist die frühzeitige Prüfung von Architekturen von entscheidender Bedeutung. Das technische Ziel der Architekturvalidierung ist daher, die Korrektheit und Konsistenz eines Systementwurfs anhand definierter Qualitätsmethoden, wie ISO 26262 für funktionale Sicherheit oder ISO 25010 was sind die Ziele für Softwarequalität, zu bewerten, noch bevor die Implementierung beginnt [VRST15]. Dieses Ziel lässt sich in die folgenden Teilziele unterteilen: Früherkennung von Fehlern, Kostenreduzierung, Sicherstellung von Qualität und Standardkonformität, Verifikation des Systemverhaltens und der Anforderungen, Beherrschung der Systemkomplexität und Verbesserung des Verständnisses [VRST15] [Kan19] [BKB19].

Zur Gewährleistung der Gültigkeit von automobilen Systemen existieren verschiedene Validierungsansätze. Dazu zählen unter anderem die Struktur- und Konsistenzprüfung, die Quantitative Analyse und Metrik, Sicherheits- und Zuverlässigkeitsanalysen, die verhaltensbasierte Validierung sowie die Versionsvalidierung. Im Rahmen dieser Arbeit wird der Fokus auf die Anforderungs- und Design-Validierung gelegt.

Anforderungs- und Design-Validierung

Diese Validierung befasst sich mit der Überprüfung von Anforderungen und der Qualität von Architektur-Entwürfen [AFPB13][KRS⁺22]. Sie stellt sicher, dass funktionale und nicht-funktionale Anforderungen, wie Skalierbarkeit, Erweiterbarkeit und Portabilität, in der Softwarearchitektur berücksichtigt werden. Dabei beginnt der Vorgang oft mit der Spezifikation von Anforderungen, anschließend erfolgt die Architekturmodellierung. Ein zentrales Verfahren hierbei ist die Traceability Analyse, welche überprüft, ob alle Anforderungen im Architekturmodell abgebildet sind und jedes Element in der Architektur auf mindestens eine Anforderung zurückgeführt werden kann [VKC⁺21].

1.4. Webentwicklungs-Stack

Um die Integration des erweiterbaren Validierungs-Frameworks in das bestehende System zu ermöglichen, wird im Folgenden der eingesetzte Technologie-Stack des webbasierten Architekturtools vorgestellt. Die Anwendung basiert vollständig auf JavaScript bzw. TypeScript und folgt einer klassischen Drei-Schichten-Architektur bestehend aus Frontend, Backend und Datenbank.

Frontend

Zur Entwicklung der Graphical User Interface (GUI) des ArchitekturTools wird das JavaScript-Framework *ReactJS*¹ genutzt. React ist ein populäres Frontend-Framework, dessen zentrales Konzept die komponentenbasierte Architektur ist. Dabei wird die GUI in kleine, wiederverwendbare und in sich geschlossene Bausteine, auch Components genannt, zerteilt. Für die visuelle Darstellung dieser Komponenten wird *CSS*² verwendet, eine Stylesheet-Sprache, die das Design vom strukturellen Aufbau trennt und dabei Farben, Schriftarten und das Layout der GUI definiert. Dabei nutzt React zur Aktualisierung der Ansicht ein virtuelles Document Object Model, wodurch bei Änderungen nur die betroffenen Bauteile der Webseite neu geladen werden.

Backend

Die Backend-Logik des Tools wird die JavaScript-Laufzeitumgebung *Node.js*³ verwendet. Node.js ermöglicht die serverseitige Ausführung von JavaScript und bildet damit das Fundament für die ereignisgesteuerte, asynchrone Verarbeitung von Anfragen. Darauf aufbauend wird das Webframework *Express.js*⁴ genutzt, welches eine strukturierte Definition von Routen und Middlewares ermöglicht. Es dient als Hauptkomponente zur Bereitstellung der Application Programming Interface (API)-Endpunkte, über die das Frontend mit dem Backend kommuniziert. Für einen einfacheren Zugriff auf die Datenbank wird das objekt-relationale Mapping-Tool *Prisma ORM*⁵ verwendet. Prisma übersetzt JavaScript-Anfragen in SQL-Befehle und ermöglicht somit eine typensichere und leistungsfähige Interaktion mit der relationalen Datenbank.

¹<https://react.dev/learn>

²<https://www.w3.org/Style/CSS/>

³<https://nodejs.org/en>

⁴<https://expressjs.com/>

⁵<https://www.prisma.io/>

Datenbank

*PostgreSQL*⁶ wird bei dem Tool als Datenbank genutzt, da es sich um ein objektrelationales Open-Source-Datenbanksystem handelt, das sich durch hohe Stabilität und umfangreiche Erweiterbarkeit auszeichnet. PostgreSQL dient als Speicherkomponente des ArchitekturTools und verwaltet alle strukturierten Daten, wie Informationen zu Fahrzeugkomponenten, Verbindungen oder Sichten. Die relationale Struktur ermöglicht die Realisierung von komplexen Anfragen und Verknüpfungen. In Verbindung mit Prisma wird ein typensicherer Zugriff auf die Datenbank gewährleistet.

⁶<https://www.postgresql.org/>

2. Stand der Technik

In diesem Kapitel werden die bestehenden Tools zum Spezifizieren von Architekturen im Bereich der Fahrzeugindustrie sowie die von ihnen verwendeten Verfahren zur Architekturvalidierung. Dies ermöglicht ein besseres Verständnis für den Beitrag, den die Abschlussarbeit leistet.

2.1. Etablierte Tools und Plattformen

Um die zunehmende Komplexität der E/E-Architekturen in modernen Fahrzeugen mit Blick auf autonome Fahrfunktionen und Fahrassistenzsystemen zu bewältigen, wird aktiv an Software gearbeitet, die es Entwicklern und Ingenieuren ermöglicht, funktionale, softwarebezogene und E/E-Architekturen zu modellieren bzw. zu spezifizieren [AHFK22] [Sch16]. Im folgenden werden die Tools PREEvision und Simulink + System Composer, welche als Industriestandard gelten, und Capella als kostenlose Alternative analysiert.

2.1.1. Vector PREEvision

PREEvision ist ein kommerzielles, modellbasiertes Tool zur Entwicklung und Optimierung verteilter, eingebetteter Systeme in der Automobilindustrie. Das Hauptziel des Tools ist es, die zunehmende Komplexität der E/E-Architekturen innerhalb der SDVs beherrschbar zu halten [AHFK22]. Das Tool richtet sich dabei an die weit akzeptierten Automobilstandards wie AUTomotive Open System ARchitecture (AUTOSAR) RIF/ReqIF¹, KBL² und VEC³ [Sch16]. Es werden außerdem die drei System-Engineering-Prinzipien unterstützt: Abstraktion (Implementationsaspekte auf eine konzeptionellere Ebene abstrahieren), Dekomposition (System kann in jeder Schicht hierarchisch zerlegt werden), Wiederverwendung (Komponenten und Modelle können von verschiedensten Produktlinien und Varianten verwendet werden) [Sch16].

Der Schwerpunkt von PREEvision ist jedoch, die auf Abbildung 2.1 grob dargestellte Schichtenarchitektur. Die Abbildung zeigt den Weg von der abstrakten Anforderungsebene zur konkreten Implementierung. Ausgehend von den Anforderungen wird eine Logische Funktionsarchitektur entworfen, welche dann in eine SOA und

¹Requirements Interchange Format: www.automotive-his.de/rif

²Kabelbaumliste: www.vda.de

³Vehicle Electric Container: www.vda.de

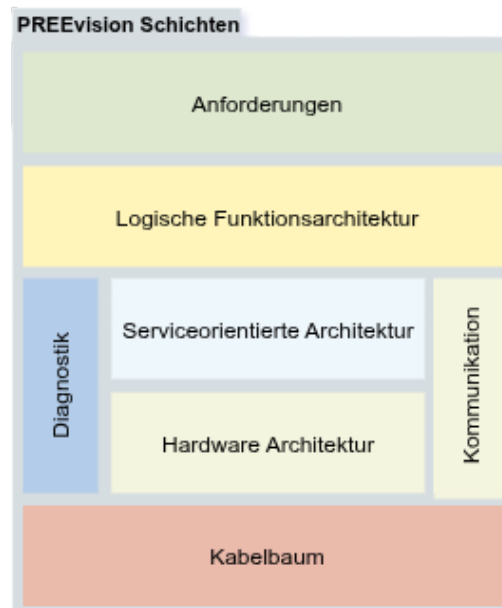


Abbildung 2.1.: Schematische PREEvision Schichtenarchitektur. (Nachzeichnung von [Vec])

eine Hardware Architektur überführt wird, bis hin zum physischen Kabelbaum. Dieser durchgängige Ansatz stellt sicher, dass alle Aspekte des Systems, inklusive Diagnostik und Kommunikation, miteinander verbunden sind (Single-Source-of-Truth) [Sch16]. Eine detaillierte Abbildung der Schichtenarchitektur ist im Appendix ?? zu finden.

2.1.2. Mathworks Simulink + System Composer

Simulink und System Composer sind kommerzielle Tools, die eng miteinander integriert sind, um modellbasierte Systementwicklung (MBSE) zu unterstützen [WVK⁺20]. Während System Composer für die Modellierung und die Analyse der statischen System- und Softwarearchitektur verwendet wird, liegt der Fokus von Simulink in der Simulation des dynamischen Verhaltens dieser Architekturen. Dieses Zusammenspiel ermöglicht einen Übergang von dem reinen Architekturentwurf zu dem konkreten, ausführbaren Designmodell [Cha20].

Wie diese enge Integration praktisch umgesetzt wird, ist in Abbildung 2.2 zu erkennen: Eine in System Composer entworfene Architekturkomponente wird direkt mit einem detaillierten Verhaltensmodell in Simulink verknüpft. Der diesem Vorgehen zugrundeliegende Arbeitsablauf folgt dem V-Modell, bei dem das Systemverhalten phasenweise modelliert und validiert wird [The25]. Um komplexe und Verhaltenswei-

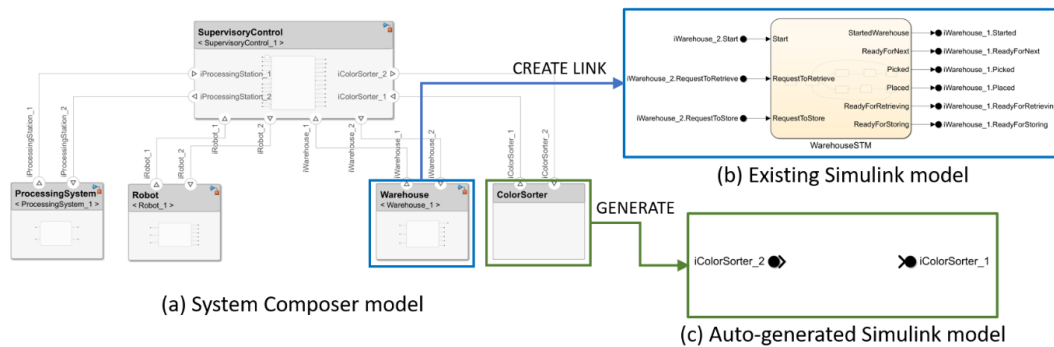


Abbildung 2.2.: Verknüpfung einer System Composer Architektur mit einem Simulink Verhaltensmodell. (Entnommen aus [Cha20])

sen modellieren zu können, werden spezielle Tools innerhalb von Simulink genutzt [Cha20]:

Stateflow dient zur Modellierung der Logik von Systemen mittels Zustandsautomaten, Flussdiagrammen oder Wahrheitstafeln. Es wird insbesondere für die übergeordnete Steuerung (Supervisory Control) und dem Fehlermanagement genutzt [Cha20]. Die Fähigkeit, ereignisdiskrete Systeme zu modellieren/ analysieren, wird durch SimEvents ermöglicht. Mithilfe von Nachrichten- und Entitätenbasierten Konzepten können Leistungsmerkmale wie der Durchsatz oder die Latenz optimiert werden [Cha20].

Wie auch bei PREEvision, unterstützt dieser gesamte Ansatz die zentralen System-Engineering Prinzipien: Abstraktion, Dekomposition und Wiederverwendung[The25].

2.1.3. Eclipse Capella

Capella ist ein nicht-kommerzielles Modellierungstool, das auf der Engineering-Methode ARCADIA⁴ basiert [Roq16]. Diese umfassende Methodik gibt einen klaren, schrittweisen Konstruktionsprozess vor, der den Anwender von der Bedarfsanalyse bis zum detaillierten Architektur-Design führt [let].

Die Abbildung 2.3 zeigt die vier Konstruktionsphasen von Arcadia. Der Prozess beginnt mit der Betriebsanalyse, die analysiert, was die Erwartungen, Anwendungsbedingungen, Integrations-, Verifikations- und Validierungsvoraussetzungen des Endnutzer sind. Darauf aufbauend folgt die Funktionale & Nicht-Funktionale Bedarfsanalyse, in der die Systemfunktionen und nicht-funktionalen Anforderungen abgeleitet werden.

Bei Arcadia haben Bedarfsanalyse und Modellierung, Architekturentwicklung und -validierung sowie Anforderungsanalyse die gleiche Wichtigkeit. Wobei die

⁴ARChitecture Analysis and Design Integrated Approach

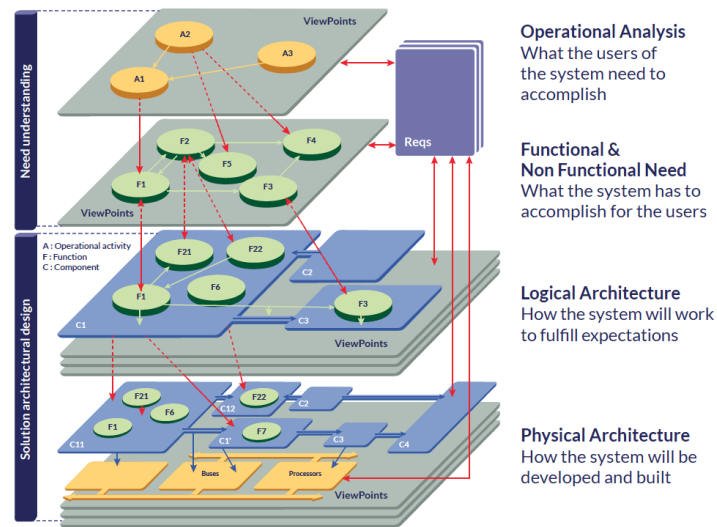


Abbildung 2.3.: Das Arcadia-Vorgehensmodell in Capella als vier übereinander gestapelte Architekturebenen (Entnommen aus [let])

Anforderungen gegen eine frühe Version des Architekturmodells auf Robustheit und Machbarkeit geprüft werden.

Nachdem System, Hardware und Software modelliert wurden, wird eine logische Architektur entwickelt. Dabei wird nach einem Kompromiss zwischen Entwurfsfaktoren, (nicht-funktionalen) Bedingungen und Ansichten gesucht. Jede Ansicht beschäftigt sich mit bestimmten Aspekten wie funktionale Konsistenz, Schnittstellen, Leistung, Echtzeit, Sicherheit, Integration, Wiederverwendung, Kosten und Risiko.

Abschließend beschreibt die physische Architektur die Umsetzung auf konkrete Hardware- und Softwarekomponenten und sichert durch die Trennung der Aspekte, Schichtenarchitekturen und standardisierte Interaktionsmuster eine effiziente, sichere Entwicklung und IVVQ⁵ [let].

Während Arcadia die komplette linke Seite des V-Modells abdeckt, deckt das Produktlebenszyklusmanagement (PLM) die rechte Seite des Modells ab und garantiert somit eine vollständige Umsetzung des V-Modells [20224].

2.2. Validierungsverfahren im Detail

Ein zentraler Aspekt moderner Architekturtools ist die Fähigkeit, entworfene Modelle zu validieren und deren Qualität sicherzustellen. Die in dieser Arbeit analysierten

⁵Integration, Verifizierung & Validierung, Qualifizierungsaktivitäten (Testing)

Tools verfolgen dabei unterschiedliche Ansätze und decken verschiedene Bereiche der Architekturvalidierung ab, die in Tabelle 2.2 vergleichend dargestellt werden.

Validierungsart	PREEvision	Capella	Simulink
<i>Modell-Integrität & Konsistenz</i>			
Inkonsistenz/Vollständigkeit	✓	✓	✓
Nachverfolgbarkeit	✓	✓	✓
Ebenenübergreifende Realisierung	✓	✓	
<i>Anforderungs- & Design-Validierung</i>			
Validierung gegen Anforderungen	✓	✓	✓
Analyse von Designalternativen	✓	✓	✓
<i>Quantitative Analysen & Metriken</i>			
Performance-Analyse	✓	✓	✓
Netzwerk- und Ressourcen-Metriken	✓		
Statische Analyse		✓	✓
<i>Sicherheits- & Zuverlässigkeitsanalysen</i>			
Sicherheitsanalysen	✓	✓	
Fehlerfortpflanzungsanalyse		✓	
<i>Verhaltensbasierte Validierung</i>			
Dynamische Simulation			✓
<i>Versions-Validierung</i>			
Prüfung von Modellversionen	✓	✓	✓

Tabelle 2.2.: Auflistung der Architekturvalidierungen nach Tool

Eine kurze Analyse der ausgewählten Tools zeigt, dass jedes von ihnen seine spezifischen Stärken in unterschiedlichen Bereichen der Architekturvalidierung besitzt. Während PREEvision bei E/E-spezifischen Metriken wie Netzwerk- und Ressourcenmetriken sowie Sicherheitsanalysen dominiert [Sch16], liegen die Stärken von Capella in der Sicherstellung der Architekturkonsistenz über mehrere Ebenen hinweg [Roq16]. Als einziges Tool bietet Simulink zudem die Möglichkeit zur dynamischen Validierung des Systemverhaltens mittels ausführbarer Simulation [The25].

In der Praxis ergibt sich durch diese hohe Spezialisierung eine zentrale Herausforderung: Keines der Tools deckt alle notwendigen Validierungsverfahren ab. Darüber hinaus sind diese Systeme meist geschlossene Plattformen. Die Anbindung neuer oder unternehmensspezifischer Validierungsalgorithmen, die vom Hersteller nicht vorgesehen sind, ist häufig gar nicht oder nur mit erheblichem Aufwand möglich. Es fehlt an flexiblen und offenen Schnittstellen zur Integration externer Algorithmen.

2. Stand der Technik

Genau an dieser Stelle setzt die Bachelorarbeit an. Ziel dieser Arbeit ist es, das generische, webbasierte *ArchitekturTool* um ein Framework zur automatisierten Architekturvalidierung zu erweitern. Durch die Implementierung einer offenen API-Schnittstelle soll die bestehende Lücke hinsichtlich mangelnder Flexibilität geschlossen und die einfache Anbindung beliebiger Validierungsalgorithmen ermöglicht werden.

3. Konzept & Design

Nachdem in den vorherigen Kapiteln die Grundlagen gelegt und die Lücken der bestehenden Systeme analysiert wurden, wird nun der Entwurf der Lösung thematisiert. In diesem Kapitel wird der Entwurf des Frameworks zur automatisierten Architekturvalidierung von Anforderung bis zum detaillierten Design nachvollziehbar dargelegt. Im ersten Abschnitt werden daher die Spezifikationen der funktionalen und nicht-funktionalen Anforderungen präsentiert. Auf dieser Basis wird anschließend der Architekturentwurf vorgestellt, welcher die Einbettung in das übergeordnete System erläutert. Die nachfolgenden Abschnitte fokussieren diesen Entwurf durch das detaillierte Konzept der API, der Validierungs-Engine, dem ersten Validierungsalgorithmus sowie dem Design der GUI.

3.1. Anforderungsanalyse

Eine präzise Definition der Anforderungen ist die zwingende Voraussetzung für eine zielgerichtete Entwicklung. Basierend auf der Analyse der bestehenden Systemlücke werden daher die funktionalen und nicht-funktionalen Anforderungen an das zu entwickelnde Framework in Tabelle 3.2 spezifiziert. Diese bilden die verbindliche Grundlage für alle nachfolgenden Design- und Implementierungsentscheidungen.

Im Folgenden wird die Notwendigkeit für jede dieser Anforderungen näher erläutert.

Algorithmen-API

Ein zentrales Element des Frameworks ist die Bereitstellung der API, um die Validierungsalgorithmen an das ArchitekturTool binden zu können (FA-1). Diese Funktion ist die Grundvoraussetzung, um das System, wie im Ziel der Arbeit gefordert, erweiterbar zu gestalten. Um die Integration weiterer bzw. Anpassung bestehender Funktionalitäten für Entwickler zu vereinfachen und eine hohe Kompatibilität zu gewährleisten, wird zudem die Einhaltung des REST-Standards gefordert (NFA-1). Dieser etablierte Standard fördert eine lose Kopplung zwischen Framework und den angebundenen Algorithmen und ermöglicht eine schnelle Einarbeitung für Entwickler, die nicht vertraut sind mit dem Programm.

ID	Anforderung	Beschreibung
<i>Funktional</i>		
FA-1	Algorithmen-API	Bereitstellung einer API, die es Nutzern ermöglicht, Validierungsalgorithmen zu verwalten (hochladen, bearbeiten, löschen) und auszuführen.
FA-2	Algorithmenverwaltung-GUI	Bereitstellung einer GUI zur Verwaltung von Validierungsalgorithmen. Diese muss Funktionen zum Hochladen, Bearbeiten, Löschen sowie zum Starten der Ausführung umfassen.
FA-3	Ergebniss-GUI	Bereitstellung einer GUI zur visuellen Darstellung der Ergebnisse von Validierungsläufen. Die GUI muss verschiedene Ausgabeformate (z. B. textuell, grafisch) unterstützen.
FA-4	Erster Validierungsalgorithmus	Bereitstellung eines ersten Validierungsalgorithmus, der eine grundlegende Architekturprüfung durchführt.
<i>Nicht-Funktional</i>		
NFA-1	Standardisierte API	Die zu entwickelnde API muss den Prinzipien des RESTfolgen.
NFA-2	Benutzerfreundlichkeit	Die GUI muss intuitiv bedienbar sein.

Tabelle 3.2.: Anforderungen an das Framework

Grafische Benutzeroberfläche

Für die Interaktion mit dem Nutzer ist eine GUI zur Verwaltung der Algorithmen essenziell (FA-2). Sie muss alle notwendigen Funktionen wie das Hochladen und Ausführen der Validierungsalgorithmen ermöglichen. Um einen reibungslosen Ablauf beim benutzen des Tools zu gewährleisten, muss die Bedienung intuitiv und selbsterklärend sein (NFA-2). Die Benutzerfreundlichkeit ist entscheidend für die Ingenieure und Entwickler im Arbeitsalltag. Des Weiteren müssen die Ergebnisse der Architekturvalidierungen visuell aufbereitet und ausgegeben werden (FA-3). Ohne eine Visualisierung der Ergebnisse ist es für den Nutzer schwer ersichtlich, was genau die Validierung ergeben hat. Die Unterstützung verschiedener Visualisierungsarten, wie textueller oder grafischer Ausgaben, ermöglichen es dem Nutzer, die Fehlerursache bzw. den Engpass effizient zu identifizieren.

Erster Validierungsalgorithmus

Das Anbinden eines ersten Validierungsalgorithmus (FA-4) ist aus mehreren Gründen wichtig. Zum einen lässt sich damit das Framework auf Korrektheit prüfen, sprich

es dient als Proof of Concept. Mit einem Validierungsalgorithmus lässt sich zum einen prüfen, ob das Hochladen, Bearbeiten und Ausführen der API wie gefordert funktioniert. Zum anderen können zukünftige Ingenieure und Entwickler den Algorithmus als Referenz nutzen um weitere Algorithmen anzubinden. Schließlich ermöglicht dieser erste Algorithmus die Durchführung von aussagekräftigen Tests und bildet die Grundlage für die Evaluation dieser Arbeit.

3.2. Systemarchitektur

Bei dem bestehenden ArchitekturTool handelt es sich um eine dockerisierte Webanwendung, deren Architektur auf einem React-Frontend, einem Node.js-Backend mit dem Express-Framework und einer PostgreSQL-Datenbank mit Prisma-ORM basiert. Um die geforderte Architekturvalidierung zu realisieren, wird diese Architektur, wie in Abbildung 3.1 dargestellt, um eine dedizierte Validierungs-API sowie um zugehörige Frontend-Module erweitert.

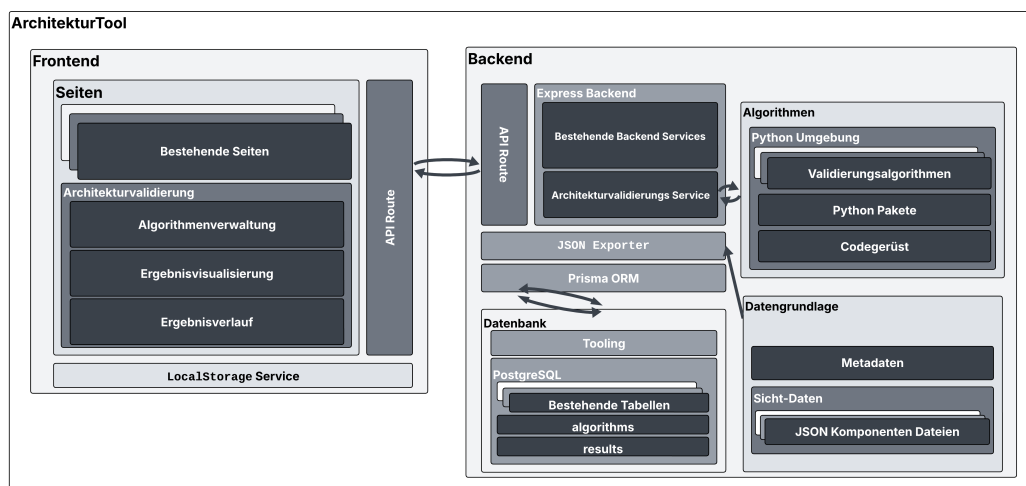


Abbildung 3.1.: Blockdiagramm des ArchitekturTool's

Wie im Blockdiagramm dargestellt, lässt sich die Erweiterung in vier Hauptbereiche unterteilen:

Algorithmenverwaltung (Frontend)

Diese Komponente bildet die zentrale Benutzeroberfläche zur Verwaltung der Validierungsalgorithmen. Sie ermöglicht es dem Nutzer, Algorithmen hochzuladen, zu bearbeiten und zu löschen. Darüber hinaus kann über die Oberfläche ein bereits

hochgeladener Algorithmus gemeinsam mit einer im Architekturtool definierten Sicht ausgewählt und ein Validierungslauf gestartet werden.

Ergebnisvisualisierung (Frontend)

Diese Komponente ist ausschließlich für die Darstellung der Ergebnisse eines Validierungslaufs verantwortlich. Die empfangenen Resultate werden entsprechend ihrer Struktur aufbereitet und dem Nutzer in geeigneter Form präsentiert, je nach Definition entweder als textuelle Ausgabe (z.B. Fehlerliste) oder als grafische Darstellung (z.B. Balken- oder Netzdiagramm). Zusätzlich wird ein Verlauf der letzten zehn Validierungsergebnisse angezeigt, um Entwicklungen oder wiederkehrende Auffälligkeiten im Zeitverlauf nachvollziehbar zu machen.

Validierungsalgorithmus-API (Backend)

Die API bildet die zentrale serverseitige Schnittstelle zur Verwaltung der Validierungsalgorithmen. Sie stellt REST-Endpunkte bereit, mit denen Algorithmen hinzugefügt, ausgelesen, aktualisiert und entfernt werden können (CRUD¹-Operationen). Dadurch wird eine flexible und standardisierte Integration neuer Validierungsverfahren ermöglicht.

Validierungs-Engine (Backend)

Dieser Teil ist für die eigentliche Durchführung bzw. Berechnung zuständig. Dabei wird ein zuvor ausgewählter Validierungsalgorithmus gemeinsam mit einer spezifischen Sicht der Architektur an die Funktion übergeben. Zur Ausführung wird eine isolierte Python-Umgebung genutzt, so erfolgt die Berechnung gekapselt und getrennt vom Hauptprozess des Webservers, was für Sicherheit und Stabilität sorgt.

Die beschriebene Systemarchitektur legt damit die technische Grundlage für die im Folgenden im detail betrachteten Komponenten und Algorithmen.

3.3. API zur Algorithmenverwaltung

Zur Verwaltung der Validierungsalgorithmen wird gemäß der Anforderung FA-1 + NFA-1 eine REST-konforme API zur Verfügung gestellt. Diese ermöglicht es, Algorithmen hochzuladen, zu bearbeiten, auszulesen und zu löschen. Die API bildet die technische Grundlage für die Interaktion zwischen Frontend und Backend und wird so gestaltet, dass sie flexibel erweiterbar und robust gegenüber Fehlern ist. Die

¹Create, Read, Update, Delete

Datenbankzugriffe erfolgen über Prisma ORM, während Datei-Uploads über die Middleware *Multer*² gehandhabt werden.

Die Funktionalität wird über die nachfolgend beschriebenen Endpunkte realisiert:

3.3.1. CREATE

Dieser Endpunkt ermöglicht das Anbinden eines neuen Validierungsalgorithmus. Dazu wird ein Formular aus dem Frontend an das Backend übermittelt, das neben Metadaten wie Name, Beschreibung, Autor, Tags und Parametern auch die eigentlich Datei, wie in aktuellen Fall eine Python-Datei, enthält. Die Datei wird über Multer verarbeitet, wobei ein zuvor konfigurierter Speicherpfad genutzt wird. Dabei wird die hochgeladene Datei mit einem Zeitstempel versehen und im Zielverzeichnis abgespeichert, um eventuelle Namenskonflikte zu vermeiden. Eine Begrenzung der Dateigröße von 10MB schützt vor übergroßen Uploads.

Vor dem Abspeichern der Datei wird geprüft, ob bereits ein Algorithmus mit dem angegebenen Namen existiert. Ist dies der Fall, wird der Vorgang mit einem Fehlerwurf und Statuscode abgebrochen. Anschließend wird anhand der Dateiendung automatisch der Dateityp bestimmt.

Die mitgesendeten Tags und Parameter werden zunächst als JSON geparkt. Abschließend wird die Datei auf dem Server abgespeichert und die Metainformationen inklusive Dateipfad über Prisma in der Datenbank festgehalten. Die *Response* enthält das neu angelegte Objekt und erfolgreichen Statuscode. Uploadfehler oder fehlerhafte Eingaben werden über entsprechende Statuscodes abgefangen und mit einer passenden Fehlermeldung versendet.

3.3.2. READ

Über diesen Endpunkt wird eine vollständige Liste aller, in der Datenbank abgelegten, Validierungsalgorithmen bereitgestellt. Die Daten werden mithilfe von Prisma abgefragt und als JSON an den Client(Frontend) gesendet. Dies ermöglicht die Anzeige aller verfügbaren Validierungsalgorithmen auf der Benutzeroberfläche. Im Fehlerfall wird ein entsprechender Statuscode und Fehlermeldung versendet und protokolliert.

3.3.3. UPDATE

Dieser Endpunkt dient zur Aktualisierung eines bereits bestehenden Algorithmus. Nach Erhalt der *id* aus der URL wird zunächst geprüft, ob ein entsprechender Eintrag in der Datenbank existiert. Sollte dies nicht der Fall sein, wird der Client mit einem Fehlercode informiert. Die zu aktualisierenden Daten beinhalten alle Felder, die auch bei der Erstellung der Algorithmen verfügbar waren.

²<https://www.npmjs.com/package/multer>

Falls beim Aktualisieren eine neue Datei mitgesendet wird, wird zunächst die alte Datei aus dem Dateisystem entfernt. Anschließend wird die neue Datei gespeichert, und sowohl der neue Dateipfad als auch der aktualisierte Dateityp werden im Datensatz aktualisiert. Die eigentliche Datenaktualisierung erfolgt über Prisma. Auch hier wird beim erfolgreichen Aktualisieren das Objekt als JSON zurückgegeben. Andernfalls werden auftretende Fehler, wie beispielsweise beim Dateizugriff, beim Parsen der Daten oder bei der Aktualisierung der Datenbank, abgefangen und protokolliert.

3.3.4. DELETE

Der letzte Endpunkt erlaubt das vollständige Entfernen eines Validierungsalgorithmus. Zunächst wird geprüft, ob zur übergebenen `id` ein Datensatz existiert. Ist dies nicht der Fall, wird eine Fehlermeldung zurückgegeben. Falls der Datensatz vorhanden ist, wird versucht, die zugehörige Datei aus dem Dateisystem zu löschen. Sollte die Datei nicht mehr vorhanden oder nicht zugänglich sein, wird dies in der Konsole als Warnung vermerkt und der Löschvorgang fortgesetzt.

Anschließend wird der entsprechende Eintrag in der Datenbank mithilfe von Prisma entfernt. Die *Response* enthält eine Bestätigung über die erfolgreiche Löschung.

3.4. Grafische Benutzeroberfläche

Um mit dem Framework zu interagieren, wird eine GUI entwickelt, die es dem Nutzer ermöglicht, Validierungsalgorithmen einfach zu verwalten, auszuführen und die Ergebnisse in passender Form einzusehen. Die GUI erweitert die zuvor beschriebene API zur Algorithmenverwaltung um eine bedienbare Oberfläche, die sowohl technische Konfigurationen als auch visuelle Rückmeldung beinhaltet. Der Fokus liegt auf einer klaren Trennung der Bedienbereiche: zum einen die Verwaltung der Algorithmen und zum anderen die Darstellung der Validierungsergebnisse.

Die GUI folgt, wie der Rest des ArchitekturTools einem modularen Aufbau. Jeder Teilbereich, vom Hochladen über das Ausführen eines Algorithmus bis hin zur Visualisierung der Ergebnisse, ist als eigenständige Komponente realisiert. Dadurch wird die Erweiterbarkeit des Tools gewährleistet und eine klare Trennung zwischen den Funktionen ermöglicht.

Im Folgenden werden die Konzepte der Algorithmenverwaltung und Ergebnisvisualisierung genauer beschrieben.

3.4.1. Algorithmenverwaltung

Das Ziel dieser Komponente ist es, dem Nutzer eine möglichst intuitive Möglichkeit anzubieten, neue Algorithmen hochzuladen, vorhandene auszuwählen, zu bearbeiten

oder zu löschen und schließlich die Validierung in Kombination mit einer ausgewählten Sicht zu starten.

Die Bedienung folgt einer funktional klar getrennten Struktur: Eine interaktive Hauptkomponente bietet die Auswahl und Steuerung, während ein Modalfenster zur Eingabe und Verwaltung der Algorithmen dient. Durch diese Aufteilung kann der Arbeitsablauf, d.h. das Verwalten und Ausführen der Algorithmen, klar gegliedert werden.

Hauptkomponente: Steuerung und Auswahl

Innerhalb der Verwaltungs-Komponente stehen zwei Auswahlfenster zur Verfügung, eines für die bestehenden Algorithmen, das andere für die im Tool definierten Architektursichten. Nachdem beide ausgewählt wurden, kann durch einen Button ein Validierungslauf ausgeführt werden. Die Ergebnisse werden anschließend angezeigt (siehe Abschnitt 3.4.2). Außerdem lässt sich über einen weiteren Button ein separates Modalfenster öffnen, in dem neue Algorithmen hochgeladen oder bestehende bearbeitet bzw. gelöscht werden können.

Das Konzept bezieht sich dabei auf typische Nutzungsszenarien: Im Ausgangszustand wird der Upload eines neuen Algorithmus vorgeschlagen. Sobald jedoch ein Algorithmus ausgewählt wird, wechselt der Button in den Bearbeitungsmodus. So bleibt die Bedienung dynamisch.

Modalfenster: Hochladen und Bearbeiten

Das modale Fenster übernimmt die Erstellung und Bearbeitung einzelner Algorithmen. Neben klassischen Eingabefeldern für Name, Beschreibung, Autor, Tags und Parameter enthält das Formular auch ein integriertes Editorfeld zur Eingabe von Python-Code. Dabei wird ein Codegerüst mit bereits geladenen Packages bereitgestellt, das als Einstiegspunkt für eigene Validierungsalgorithmen dient.

Ein zentrales Konzept des Formulars ist die Wiederverwendbarkeit: Das Modal erkennt anhand übergebener Information (ausgewählter Algorithmus), ob es sich um eine Erstellung oder eine Bearbeitung handelt, und passt den Inhalt des Formulars entsprechend an. So lassen sich bestehende Algorithmen problemlos aktualisieren bzw. vollständig entfernen. Nach Abschluss einer Aktion wird die Auswahl in der Hauptkomponente automatisch aktualisiert, sodass der Nutzer immer mit dem neuesten Datenstand arbeitet.

3.4.2. (Ergebnis-)Visualisierung + Verlauf

Nach dem Start eines Validierungslaufs besteht die zentrale Aufgabe des Tools darin, die erhaltenen Ergebnisse transparent und nutzerfreundlich darzustellen. Die grafische Visualisierungskomponente wird so entwickelt, dass verschiedene Ausgabeformate

unterstützt werden und sich automatisch an den vom Algorithmus angegebenen Ergebnisdatentyp anpasst. Ergänzt wird diese Komponente durch einen Verlauf bzw. History, welcher die letzten zehn Validierungsdurchläufe archiviert und so dem Nutzer eine nachträgliche Einsicht sowie den Vergleich verschiedener Läufe ermöglicht. Zusammen bilden diese Komponenten die Grundlage für eine nachvollziehbare und stetig überprüfbare Architekturvalidierung.

Ergebnisvisualisierung

Die Ergebnisvisualisierung wird so entwickelt, dass sie auf verschiedene Ergebnistypen reagieren kann und diese übersichtlich darstellt. Je nach übermitteltem Typ wird das Ergebnis entweder als formatierter Text oder als Diagramm visualisiert. Die Auswahl der Darstellungsform erfolgt automatisch und wird innerhalb der Komponente entschieden.

Für die Visualisierung metrischer Daten wird eine eigene Hilfskomponente integriert, die verschiedene Darstellungsformen wie Linien-, Balken- oder Flächendiagramme unterstützt. Falls vom Algorithmus ein Grenzwert mitgeliefert wird, erscheint dieser als Referenzlinie im Diagramm. Datenpunkte, die diesen Grenzwert überschreiten, werden farblich markiert. Dies ermöglicht eine schnelle Interpretation und erleichtert die Bewertung der Ergebnisse.

Wie auch beim dem Modalfenster, folgt diese Komponente einem modularen und erweiterbaren Ansatz, sodass neue Ergebnisformate bei Bedarf ergänzt werden können, ohne das gesamte Framework zu verändern.

Ergebnisverlauf

Ergänzend zu der Darstellung aktueller Ergebnisse wird eine Verlaufskomponente implementiert, welche die letzten zehn Validierungsläufe in chronologischer Reihenfolge aufführt. Jeder Eintrag enthält Eckdaten zum verwendeten Algorithmus, zur gewählten Sicht sowie den Zeitpunkt, an dem die Validierung durchgeführt wurde. Die entsprechenden Ergebnisse können aufgeklappt werden und in der selben Form wie die Hauptanzeige analysiert werden.

Das Design dieser Komponente folgt dem aufklappbaren Listenprinzip. Dadurch bleibt die Übersichtlichkeit auch bei mehreren Einträgen bestehen. Durch die Wiederverwendung des bereits beschriebenen Visualisierungskonzept, bleibt die Nutzererfahrung konsistent.

3.5. Validierungs-Engine

Das Ziel dieser Komponente ist es, einen ausgewählten Validierungsalgorithmus mit einer konkreten, im ArchitekturTool bereits spezifizierten, Sicht als Eingabe

auszuführen und das Ergebnis danach nutzerfreundlich bereitzustellen. Dafür wird diese Komponente so realisiert, dass sowohl die vorbereitende Datenzusammenführung als auch die Ausführung des Algorithmus sowie die Verarbeitung des Ergebnisses übernimmt.

Der Aufbau der Komponente ist so, dass sie losgelöst vom Hauptprozess des Webservers arbeitet. Dazu wird die Ausführung des Algorithmus in einem separaten Python-Prozess gestartet. Dies ermöglicht neben der Nutzung von etablierter Bibliotheken in Bezug auf die Architekturvalidierung auch verbesserte Stabilität und Sicherheit des Gesamtsystems, da potentielle Laufzeitfehler innerhalb des Skripts isoliert behandelt werden.

Zu Beginn jeder Ausführung wird ein Algorithmus und einer Sicht, über das Frontend ausgewählt und an das Backend versendet. Ausgehend von der gewählten Sicht müssen die zugehörigen Architekturkomponenten vor der Ausführung als strukturierte JSON-Dateien vorliegen. Falls dem nicht so ist, wird der Export automatisch angestoßen.

Für diesen Zweck wird eine bestehende Export-Komponente (JSON-Exporter) verwendet, die im Rahmen dieser Arbeit angepasst wurde. Diese ist dafür verantwortlich die in der Datenbank gespeicherten Komponenten, Verbindungen und Schnittstellen zu extrahieren und in ein komponentenbasiertes JSON-Format zu parsen. Dafür wird jede Komponente gemeinsam mit ihren Anforderungen, Garantien und internen Verbindungen exportiert und serverseitig in einer strukturierten Ordnerstruktur abgespeichert. Dieser wird so lange wiederverwendet, bis die zugrunde liegende Architektur verändert wird.

Nachdem alle notwendigen Dateien vorliegen, wird ein JSON-Objekt erzeugt, welches alle Komponenten der gewählten Sicht enthält. Diese Architektur wird als Eingabe an den Python-Prozess gesendet. Während der Ausführung werden sowohl Ausgabe als auch Fehlermeldungen überwacht. Nach erfolgreichem Ausführen des Skripts wird das Ergebnis interpretiert und entweder der textuelle Ausgabe oder als metrische Ausgabe zugewiesen.

Das Ergebnis wird danach gemeinsam mit weiteren Metadaten, wie Ausführungsdauer, Status und eventueller Fehlermeldung in der Datenbank abgespeichert. Diese Informationen stehen danach sowohl für die Ergebnisvisualisierung als auch für den Verlauf zur Verfügung.

4. Implementierung

In diesem Kapitel wird die Umsetzung der in Kapitel 3 definierten Konzepte und Designentscheidungen für das Framework im Detail beschrieben. Die Implementierung erfolgt als Erweiterung des bestehenden ArchitekturTools und nutzt dessen etablierten Technologie-Stack.

Zunächst wird die Backend-Implementierung vorgestellt, die sowohl die API zur Verwaltung der Algorithmen als auch die Engine zur Ausführung der Validierungsläufe enthält. Danach wird die Implementierung der GUI erläutert, welche die Komponenten zur Algorithmenverwaltung und zur Visualisierung der Ergebnisse beinhaltet. Abschließend wird die Implementierung des ersten Validierungsalgorithmus erläutert, dessen Durchführung im Rahmen der Evaluation als Proof of Concept für die Funktionsfähigkeit des Frameworks dient.

4.1. Backend

Das Backend bildet den serverseitigen Kern des implementierten Frameworks. Es ist dabei für die zentrale Logik verantwortlich und setzt die zuvor entworfenen Konzepte für Verwaltung und Steuerung um. Die Implementierung basiert auf der Node.js-Laufzeitumgebung in Verbindung mit dem Express.js-Framework und dem Prisma ORM. Die folgenden Unterabschnitte erläutern im Detail, wie sowohl zuvor konzipierte API zur Algorithmenverwaltung als auch die Validierungsengine umgesetzt wurden.

4.1.1. Umsetzung der API

Um dem Frontend die Interaktion und Verwaltung der Validierungsalgorithmen zu ermöglichen, wurde serverseitig eine REST-API implementiert, die Endpunkte für alle erforderlichen CRUD-Operationen bereitstellt. Die neu implementierten Endpunkte integrieren sich in die bestehende Routing-Struktur des ArchitekturTools und sind unter dem Präfix `/api/v2/` erreichbar.

CREATE - POST /api/v2/algorithm

Zum Anbinden neuer Validierungsalgorithmen dient der **POST**-Endpunkt. Zunächst wird eine ankommende Anfrage von der *Multer*¹-Middleware verarbeitet, welche die hochgeladene Datei aus der **multipart/form-data**-Anfrage extrahiert und in dem vordefinierten Verzeichnis (**/uploads/algorithm**) serverseitig speichert. Die Eindeutigkeit der Algorithmen wird gewährleistet, in dem vor dem Speichern mit der von Prisma zur Verfügung gestellten Methode **prisma_db.algorithm.findFirst()** geprüft, ob ein Algorithmus mit dem übergebenen Namen bereits in der **algorithm**-Datenbanktabelle existiert. Falls dies der Fall ist, wird die Anfrage mit dem Statuscode 409 Conflict abgelehnt. Andernfalls werden die Metadaten zusammen mit dem Dateipfad durch **prisma_db.algorithm.create()** in der Datenbanktabelle abgelegt und der neu erstellte Datensatz mit dem Created-Statuscode als *Response* zurückgegeben.

READ - GET /api/v2/algorithms

Um alle verfügbaren Algorithmen im Frontend darzustellen, stellt der **GET**-Endpunkt eine Liste aller Einträge innerhalb der **algorithm**-Tabelle bereit. Dies wird durch den Aufruf der Prisma-Methode **prisma_db.algorithm.findMany()** umgesetzt, die alle Datensätze abrufen und als JSON-Array an das Frontend übermittelt.

UPDATE - PATCH /api/v2/algorithm/:id

Zur Aktualisierung eines bestehenden Validierungsalgorithmus wird der **PATCH**-Endpunkt genutzt. Anhand der in der *URL*² übergebenen **:id** wird der zugehörige Datensatz mittels **prisma_db.algorithm.findunique()** aus der **algorithm**-Tabelle abgerufen. Wird im Rahmen der Anfrage eine neue Datei hochgeladen, wird die alte, zugehörige Datei zunächst mit der Node.js-Funktion **fs.unlinkSync** vom Server gelöscht, um Dateien ohne Zugehörigkeit zu vermeiden. Anschließend werden die aktualisierten Metadaten sowie der neue Dateipfad mithilfe von **prisma_db.algorithm.update()** in der **algorithm**-Tabelle gespeichert.

DELETE - DELETE /api/v2/algorithm/:id

Der **DELETE**-Endpunkt dient dem vollständigen Löschen eines Algorithmus. Um die Konsistenz des Systems sicherzustellen, folgt die Implementierung dieses Endpunkts einem zweistufigen Prozess: Zu Beginn wird der Datensatz anhand der übergebenen **:id** aus der Datenbank gelesen, um den Pfad zur entsprechenden Datei zu ermitteln. Diese Datei wird im Anschluss über **fs.promises.unlink** vom Dateisystem des Servers entfernt. Danach wird im zweiten Schritt der Datenbankeintrag selbst über

¹<https://www.npmjs.com/package/multer>

²Uniform Resource Locator

`prisma_db.algorithm.delete()` aus der Datenbanktabelle gelöscht. Mit diesem Ablauf wird sichergestellt, dass, wie beim **UPDATE**-Endpunkt, keine verwaisten Dateien auf dem Server zurückbleiben

4.1.2. Integrierung der Validierungs-Engine

Die Durchführung der Validierungsläufe bildet die Schlüsselfunktionalität des Backends. Sie ist dafür zuständig die Daten aufzubereiten, die kontrollierte Ausführung eines isolierten Python-Prozesses und die sowie die Verarbeitung und Abspeicherung der Ergebnisse. Dieser Prozess wird durch einen dedizierten API-Endpunkt gestartet.

Datenaufbereitung und Export

Nachdem die Anfrage mit den IDs für den Algorithmus und der Architektursicht validiert wurde, stellt die Engine die erforderlichen Komponentendaten bereit. Hierfür wird auf eine angepasste Version des bestehenden *JSON-Exporter* des ArchitekturTool zurückgegriffen.

Die Engine prüft zunächst, ob die Komponenten der angeforderten Sicht bereits als Datei im JSON-Format auf dem Server liegen. Falls nicht, wird der Exportprozess automatisch gestartet, welcher die Komponentendaten aus der Datenbank extrahiert und in einem sichtenspezifischen Ordner ablegt. Anschließend werden diese JSON-Dateien eingelesen und zu einem einzigen Architektur-Objekt zusammengefügt, das als Eingabe für den Validierungsalgorithmus dient.

Durchführung des Python-Prozess

Die eigentliche Validierung wird in einem isolierten Python-Prozess durchgeführt, um die Stabilität des Servers zugewährleisten. Hierfür wird die `spawn`-Funktion aus dem `child_process`-Modul von Node.js genutzt. Diese startet das Python-Skript, das zum ausgewählten Algorithmus gehört. Das zuvor erstellte Architektur-Objekt wird als JSON-String in den Standard-Input des Python-Prozesses geschrieben. Eventuelle Probleme beim Starten des Prozesses werden vom Error-Listener abgefangen.

Ergebnisverarbeitung

Gegen Ende der Durchführung nimmt das Backend die Standardausgabe und die Fehlerausgabe des Python-Skripts entgegen. Wenn ein Fehler auftritt, wird der Validierungslauf als fehlgeschlagen gewertet und ein entsprechender Eintrag mit der entsprechenden Fehlermeldung in der `result`-Tabelle der Datenbank gespeichert. Anhand der Definierung der Struktur des Objekts wird der Ergebnistyp dynamisch als `TEXT` oder `METRIC` bestimmt. Schließlich wird das Ergebnis gemeinsam mit den Metadaten wie der Ausführungsdauer und dem Erfolgsstatus mithilfe von Prisma

auch in der `result`-Tabelle abgespeichert und zur Visualisierung an das Frontend gesendet.

4.2. Frontend

Die GUI wurde mit React entwickelt und folgt einer modularen, komponentenbasierten Architektur, was eine klare Trennung der Zuständigkeit ermöglicht. Ein übergeordneter `ValidationContainer` verwaltet den gesamten Anwendungszustand wie Ladeindikatoren, Fehler, Ergebnisse und Validierungsverlauf. Dieser Zustandshalter gibt Daten sowie Callback-Funktion über Props an die untergeordneten Komponenten weiter. Zur Speicherung des Verlaufs über die Browsersitzung hinweg nutzt der Zustandshalter den `localStorage` des Browser.

4.2.1. Algorithmenverwaltungs-Komponenten

Die Verwaltung der Algorithmen ist die zentrale Interaktionsstelle für den Nutzer. Diese besteht aus einer Hauptkomponente zur Steuerung und einem modalen Fenster zum Erstellen, Bearbeiten und Löschen der Algorithmen.

Hauptkomponente

Die zentrale Steuereinheit ruft bei ihrer Initialisierung mithilfe der Lebenszyklus-Methode `componentDidMount` über *axios*³ asynchron jeweils die `GET`-Anfrage für die Algorithmen und die Architektursichten auf. Die erhaltenen Daten werden lokal im Zustand der Komponente gespeichert und dazu genutzt, die beiden Auswahllisten zu füllen. Die `onChange`-Handler dieser Listen aktualisieren den Zustand, sobald ein Algorithmus bzw. eine Sicht ausgewählt wurde.

Ein Klick auf den deidierten "Validieren"-Button startet den in Abschnitt 4.1.2 beschriebenen Validierungslauf. Dabei wird eine `POST`-Anfrage mit der `algorithmId` und `viewId` als Parameter an den entsprechenden Endpunkt gesendet. Mittels einer Callback-Funktion wird der Fortschritt und das Ergebnis dieses Durchlaufs an die übergeordnete Container-Komponente gesendet.

Modalfenster

Für das Hochladen, Bearbeiten und Entfernen der Algorithmen wird die Komponente `AlgorithmModal` genutzt die ein separates Fenster öffnet. Dieses modale Fenster unterscheidet zwischen einem Hochladungs- und einem Bearbeitungsmodus, je nachdem, ob eine `selectedAlgorithm`-Prop übergeben wird. Im Bearbeitungsmodus wird das Formular bereits mit den jeweiligen Daten vorbefüllt. Dem Nutzer wird es mithilfe

³<https://axios-http.com/docs/intro>

des integrierten *Monaco-Editors*⁴ ermöglicht, Python-Code direkt im Browser zu schreiben, einzufügen und zu bearbeiten. Um dem Nutzer dabei den Einstieg zu erleichtern und die Konsistenz der Algorithmen zu gewährleisten, wird ein Code-Gerüst als Vorlage bereitgestellt, wie in Abbildung A.1 dargestellt.

Die Ziele beim Anbinden neuer Algorithmen sind das vereinfachen und das standardisieren:

- **Verringern von Boilerplate-Code:** Das Gerüst enthält eine `main`-Funktion, welche die gesamte Interaktion mit der Engine zusammenfasst. Sie übernimmt das einlesen der Architektur über den Input, die Fehlerbehandlung bei ungültigem JSON sowie das Schreiben des Ergebnis-Objekts auf den Output. Nutzer müssen sich somit nicht mit dem wiederkehrenden Teil des Codes befassen.
- **Klar definierter Bereich:** Durch die vorgegebene Struktur, ist die einzige Aufgabe des Nutzers, die Funktion `run_validation_algorithm` mit der eigentlich Validierung zu füllen. Somit wird sichergestellt, dass der Fokus nur auf das Prüfen der Logik liegt.
- **Bereitstellung gängiger Bibliotheken:** Wie in Abbildung A.1 zu sehen ist, importiert das Gerüst bereits einige in der Architekturvalidierung etablierten Bibliotheken, darunter `pandas`, `numpy`, `matplotlib` und `cantools`. Damit wird eine einheitliche und leistungsstarke Entwicklungsumgebung geschaffen. Natürlich kann man manuell weitere Bibliotheken hinzufügen, indem man diese in der `py-libraries.txt` hinein schreibt.
- **Definierte Ergebnisstruktur:** Das Gerüst gibt durch die Dokumentation eine klare Struktur für das Ergebnis-Dictionary vor. Der Nutzer legt durch die Wahl der Schlüsselbegriffe im zurückgegebenen Dictionary fest, wie das Ergebnis zu interpretieren ist. Diese Struktur ist entscheidend, da sie vom Backend ausgelesen wird, um den Typen des Ergebnisses (`TEXT` oder `METRIC`) zu bestimmen. Damit wird es dem Frontend ermöglicht, ohne weitere Konfiguration die passende Visualisierungskomponente zu wählen.

Beim Speichern wird der Code aus dem Editor in ein `Blob`- und anschließend in eine `File`-Objekt umgewandelt. Dieses wird zusammen mit den Metadaten aus dem Formular über eine `POST`- oder `PATCH`-Anfrage an das Backend gesendet. Zusätzlich ist eine Löschfunktion implementiert, welche mit einer `DELETE`-Anfrage die entsprechenden Einträge entfernt.

⁴<https://microsoft.github.io/monaco-editor/>

4.2.2. Visualisierung der Ergebnisse

Die Darstellung der Ergebnisse ist, ebenso wie die Algorithmenverwaltung, modular aufgebaut, um verschiedene Ergebnistypen und einen Ergebnisverlauf abzubilden. Sie unterteilt sich in die dynamische Darstellung des aktuellen Ergebnisses, mit der zugrundeliegenden Graphenvisualisierung und dem Verlauf der Validierungsdurchläufe.

Mithilfe dieser Komponente wird das Validierungsergebnis der aktuellen Ausführung visualisiert. Ihr gerendeter Inhalt wird durch die vom Container übergebenen Props gesteuert.

Das Kernstück dieser Komponente ist das dynamische Rendern der Ergebnisse, das auf den `resultType`-Prop basiert. Hier wird mittels `switch`-Anweisung, je nach Ergebnistyp, die passende Visualisierung ausgewählt:

- **Textuelle Ausgabe(TEXT):** Standardmäßig werden die Ergebnisse als JSON in einem `<pre>`-Element dargestellt. Wie dieser formatiert sein soll bestimmt der Nutzer in seinem Python-Code.
- **Metrische Ausgabe(METRIC):** Bei metrischen Ergebnisdaten wird die Komponente Diagramm-Visualisierung instanziiert. Ihr werden die Messwerte und Grenzwerte als Props übergeben, woraufhin die Komponente die Darstellung als Linien-, Balken- oder Flächendiagramm übernimmt.

Der Ansatz des dynamischen Rendern stellt sicher, dass das Frontend flexibel auf unterschiedliche Ausgabeformate der Algorithmen reagieren kann, ohne dass Änderungen an der Komponente vorgenommen werden müssen.

Diagramm-Visualisierung

Die Visualisierung der metrischen Daten baut auf der populären Bibliothek *Recharts*⁵, welche mit den gelieferten Daten in eine interaktive und auswertbare grafische Form überführt.

Der Ablauf innerhalb dieser Komponente ist wie folgt:

1. **Normalisierung der Daten:** Zunächst werden die eingehenden Daten durch eine Funktion aufbereitet und normalisiert. Diese Funktion extrahiert aus den Datenpunkten eine einheitliche Bezeichnung für die X-Achse, welche entweder aus einem `timestamp` oder `name` Feld entnommen wird. Außerdem stellt sie sicher, dass die restlichen numerischen Werte korrekt als verwertbare Datenserie für das Diagramm zur Verfügung stehen.

⁵<https://recharts.org/en-US>

2. **Dynamische Diagramm-Auswahl:** Basierend auf dem `chartType`-Prop, die vom Validierungsalgorithmus mit dem Ergebnis-Dictionary mitgegeben wird, wählt die Komponente dynamisch die passende Visualisierung aus Recharts aus (`LineChart`, `BarChart` oder `AreaChart`). Dies ermöglicht dem Ersteller des Algorithmus, die am besten geeignete Darstellungsform für die Metrik vorzugeben.
3. **Visualisierung von Grenzwerten:** Ein wichtiges Feature ist es Grenzwerte visuell darzustellen. Wird ein `threshold` übergeben, rendert die Komponente eine horizontale, gestrichelte Linie in das Diagramm. Dies macht es sofort ersichtlich, ob die Messwerte innerhalb des erwarteten Bereichs liegen. Zudem wurde eine Funktion zur Prüfung jedes einzelnen Datenpunkt eines Liniendiagramms implementiert. Wird der Wert des definierten Grenzwerts überschritten, wird der zugehörige Punkt innerhalb des Diagramms farblich markiert.

Des Weiteren werden UI-Elemente, wie eine Legende, Tooltips beim überfahren mit der Maus sowie beschriftete X-,Y-Achsen, gerendert um eine hohe Benutzerfreundlichkeit und eine einfache Interpretation zu gewährleisten.

Ergebnisverlauf

Damit die Verfolgbarkeit der Validierungsergebnisse zu gewährleisten, wird die Komponente für die Darstellung der letzten Zehn Validierungsläufe implementiert. Hierzu wird die `history`-Prop vom übergeordneten Container empfangen, der diesen Zustand über den `localStorage`-Service des Webbrowsers auch nach einem Neuladen der Seite erhält.

Die Ergebnisse werden hierbei in Form einer aufklappbaren Liste präsentiert, um die Benutzeroberfläche übersichtlich zu halten. Listeneinträge zeigen zunächst die wichtigsten Eckdaten bzgl. des Validierungslauf: Name des Algorithmus, die gewählte Architektursicht und dem Zeitstempel der Durchführung. Mit einem Klick auf den jeweiligen Eintrag öffnet sich dieser und visualisiert die vergangenen Ergebnisse mit der selben Darstellung wie bei der Visualisierung der aktuellen Ergebnisse. Dieser Ansatz stellt eine konsistente Nutzererfahrung sicher.

4.3. Validierungsalgorithmus

Literaturverzeichnis

- [20224] Arcadia (Ingenieurwesen). In: Wikipedia (2024), Dezember
- [AFPB13] AHRENS, Dirk ; FREY, Andreas ; PFEIFFER, Andreas ; BERTRAM, Tors-
ten: Objective Evaluation of Software Architectures in Driver Assi-
stance Systems. In: Computer Science - Research and Development
28 (2013), Februar, Nr. 1, S. 23–43. <http://dx.doi.org/10.1007/s00450-011-0185-x>. – DOI 10.1007/s00450-011-0185-x. – ISSN 1865–2042
- [AHFK22] ASKARIPOOR, Hadi ; HASHEMI FARZANEH, Morteza ; KNOLL, Alois: E/E
Architecture Synthesis: Challenges and Technologies. In: Electronics
11 (2022), Januar, Nr. 4, S. 518. <http://dx.doi.org/10.3390/electronics11040518>. – DOI 10.3390/electronics11040518. – ISSN 2079–9292
- [BCK21] BASS, Len ; CLEMENTS, Paul ; KAZMAN, Rick: Software Architecture
in Practice, 4th Edition. 1st edition. Erscheinungsort nicht ermittelbar :
Addison-Wesley Professional, 2021. – ISBN 978–0–13–688597–9
- [BKB19] BUCHER, Harald ; KAMM, Simon ; BECKER, Jürgen: Cross-Layer Beha-
vioral Modeling and Simulation of E/E-Architectures Using PREEvision
and Ptolemy II. In: SNE Simulation Notes Europe 29 (2019), Juni, Nr.
2, S. 73–78. <http://dx.doi.org/10.11128/sne.29.tn.10472>. – DOI 10.11128/sne.29.tn.10472. – ISSN 23059974, 23060271
- [Bro06] BROY, Manfred: Challenges in Automotive Software Engineering.
In: Proceedings of the 28th International Conference on Software
Engineering. New York, NY, USA : Association for Computing Ma-
chinery, Mai 2006 (ICSE '06). – ISBN 978–1–59593–375–1, S. 33–42
- [BSPL21] BANDUR, Victor ; SELIM, Gehan ; PANTELIC, Vera ; LAWFORDE, Mark:
Making the Case for Centralized Automotive E/E Architectures. In:
IEEE Transactions on Vehicular Technology 70 (2021), Februar, Nr. 2,
S. 1230–1245. <http://dx.doi.org/10.1109/TVT.2021.3054934>. – DOI 10.1109/TVT.2021.3054934. – ISSN 1939–9359

- [Cha20] CHATTERJEE, Joyeeta: Applications of MathWorks Tools in Model-Based Systems Engineering. Eindhoven, Eindhoven University of Technology, Diplomarbeit, November 2020
- [CK20] CEBOTARI, Vadim ; KUGELE, Stefan: Playground for Early Automotive Service Architecture Design and Evaluation. In: 2020 IEEE Intelligent Vehicles Symposium (IV), 2020. – ISSN 2642–7214, S. 1349–1356
- [HSS⁺22] HENLE, Jacqueline ; STOFFEL, Martin ; SCHINDEWOLF, Marc ; NÄGELE, Ann-Therese ; SAX, Eric: Architecture Platforms for Future Vehicles: A Comparison of ROS2 and Adaptive AUTOSAR. In: 2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC), 2022, S. 3095–3102
- [Jia19] JIANG, Shugang: Vehicle E/E Architecture and Its Adaptation to New Technical Trends. In: WCX SAE World Congress Experience, 2019, S. 2019–01–0862
- [Kan19] KANG, Eun-Young: A Formal Verification Technique for Architecture-based Embedded Systems in EAST-ADL
- [KMK⁺24] KLÜNER, David P. ; MOLZ, Marius ; KAMPMANN, Alexandru ; KOWALEWSKI, Stefan ; ALRIFAEI, Bassam: Modern Middlewares for Automated Vehicles: A Tutorial. <https://arxiv.org/abs/2412.07817v1>, Dezember 2024
- [KOS21] KUGELE, Stefan ; OBERGFELL, Philipp ; SAX, Eric: Model-Based Resource Analysis and Synthesis of Service-Oriented Automotive Software Architectures. In: Software and Systems Modeling 20 (2021), Dezember, Nr. 6, S. 1945–1975. <http://dx.doi.org/10.1007/s10270-021-00896-9>. – DOI 10.1007/s10270-021-00896-9. – ISSN 1619–1366, 1619–1374
- [KRS⁺22] KNIEKE, Christoph ; RAUSCH, Andreas ; SCHINDLER, Mirco ; STRASSER, Arthur ; VOGEL, Martin: Managed Evolution of Automotive Software Product Line Architectures: A Systematic Literature Study. In: Electronics 11 (2022), Juni, Nr. 12, S. 1860. <http://dx.doi.org/10.3390/electronics11121860>. – DOI 10.3390/electronics11121860. – ISSN 2079–9292
- [let] Let Yourself Be Guided with Arcadia: A Comprehensive Methodological and Tool-Supported Model-Based Engineering Guidance. <https://mbse-capella.org/arcadia.html>,

- [NDN06] NEELY, Steve ; DOBSON, Simon ; NIXON, Paddy: Adaptive Middleware for Autonomic Systems. In: annals of telecommunications - annales des télécommunications 61 (2006), Oktober, Nr. 9-10, S. 1099–1118. <http://dx.doi.org/10.1007/BF03219883>. – DOI 10.1007/BF03219883. – ISSN 0003–4347, 1958–9395
- [RGKS20] RUMEZ, Marcel ; GRIMM, Daniel ; KRIESTEN, Reiner ; SAX, Eric: An Overview of Automotive Service-Oriented Architectures and Implications for Security Countermeasures. In: IEEE Access 8 (2020), S. 221852–221870. <http://dx.doi.org/10.1109/ACCESS.2020.3043070>. – DOI 10.1109/ACCESS.2020.3043070. – ISSN 2169–3536
- [Roq16] ROQUES, Pascal: MBSE with the ARCADIA Method and the Capella Tool. In: 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016). Toulouse, France, Januar 2016
- [Sch16] SCHÄUFFELE, Jörg: E/E Architectural Design and Optimization Using PREEvision. In: SAE 2016 World Congress and Exhibition, 2016, S. 2016–01–0016
- [The25] THE MATHWORKS, INC.: System Composer™ Getting Started Guide (R2025a) / The MathWorks, Inc. 2025. – Forschungsbericht. – 90 S.
- [Vec] VECTOR INFORMATIK GMBH: PREEvision – Model-Based E/E Development from Architecture Design to Series Production (Fact Sheet, Version 1.1) / Vector Informatik GmbH. – Forschungsbericht. – 2 S.
- [VKC⁺21] VOGEL, Martin ; KNAPIK, Peter ; COHRS, Moritz ; SZYPERREK, Bernd ; PUESCHEL, Winfried ; ETZEL, Haiko ; FIEBIG, Daniel ; RAUSCH, Andreas ; KUHRMANN, Marco: Metrics in Automotive Software Development: A Systematic Literature Review. In: Journal of Software: Evolution and Process 33 (2021), Februar, Nr. 2. <http://dx.doi.org/10.1002/smr.2296>. – DOI 10.1002/smr.2296. – ISSN 2047–7473, 2047–7481
- [VRST15] VENKITACHALAM, Hariharan ; RICHENHAGEN, Johannes ; SCHLOSSER, Axel ; TASKY, Thomas: Metrics for Verification and Validation of Architecture in Powertrain Software Development. In: Proceedings of the First International Workshop on Automotive Software Architecture. New York, NY, USA : Association for Computing Machinery, Mai 2015 (WASA '15). – ISBN 978–1–4503–3444–0, S. 27–33
- [WGC⁺24] WANG, Wenwei ; GUO, Kaidi ; CAO, Wanke ; ZHU, Hailong ; NAN, Jinrui ; YU, Lei: Review of Electrical and Electronic Architectures for Autonomous Vehicles: Topologies, Networking and Simulators. In:

- Automotive Innovation 7 (2024), Februar, Nr. 1, S. 82–101. <http://dx.doi.org/10.1007/s42154-023-00266-9>. – DOI 10.1007/s42154-023-00266-9. – ISSN 2096-4250, 2522-8765
- [WVK⁺20] WATKINS, Christopher B. ; VARGHESE, Jerry ; KNIGHT, Michael ; PETTEYS, Becky ; ROSS, Jordan: System Architecture Modeling for Electronic Systems Using MathWorks System Composer and Simulink. In: 2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC), 2020. – ISSN 2155-7209, S. 1–10
- [ZZL⁺21] ZHU, Hailong ; ZHOU, Wei ; LI, Zhiheng ; LI, Li ; HUANG, Tao: Requirements-Driven Automotive Electrical/Electronic Architecture: A Survey and Prospective Trends. In: IEEE Access 9 (2021), S. 100096–100112. <http://dx.doi.org/10.1109/ACCESS.2021.3093077>. – DOI 10.1109/ACCESS.2021.3093077. – ISSN 2169-3536

A. Appendix

A. Appendix

```
1 Python-Gerüst für Validierungs-Algorithmen
2 - Liest ein JSON-Objekt von stdin
3 - Führt "run_validation_algorithm(architecture)" aus
4 - Schreibt das Ergebnis als JSON auf stdout
5
6 Bibliotheken sind importiert, implementiere Validierungsalgorithmus in "run_validation_algorithm".
7 """
8 #Ein-/Ausgabe und Textverarbeitung
9 import sys
10 import json
11 import re
12 import io
13 import base64
14
15 #Libraries für Validierung
16 import numpy as np
17 import scipy
18 import pandas as pd
19 import matplotlib.pyplot as plt
20 import can
21 import cantools
22 import lxml
23 import xmlschema
24 import networkx as nx
25 import asammdf
26 import control
27
28
29 def run_validation_algorithm(architecture: dict) -> dict:
30     """
31     Hier den Validierungsalgorithmus implementieren.
32
33     Args:
34         architecture (dict): Eingabe mit 'viewId' und 'components'.
35
36     Returns:
37         dict: Resultat-Dictionary, das serialisierbar ist.
38             Schlüssel: metric|text (threshold) line|bar|area.
39     """
40
41     raise NotImplementedError("run_validation_algorithm ist nicht implementiert.")
42
43
44 def main():
45     """
46     Main-Routine:
47     - Liest JSON von stdin
48     - Ruft run_validation_algorithm auf
49     - Gibt Ergebnis als JSON auf stdout aus
50     """
51     try:
52         raw = sys.stdin.read()
53         if not raw:
54             raise ValueError("Keine Daten von stdin erhalten.")
55         architecture = json.loads(raw)
56     except Exception as e:
57         error_obj = {"error": "Ungültiges JSON von stdin", "details": str(e)}
58         print(json.dumps(error_obj))
59         sys.exit(1)
60
61     try:
62         result = run_validation_algorithm(architecture)
63     except Exception as e:
64         error_obj = {"error": "Fehler im Algorithmus", "details": str(e)}
65         print(json.dumps(error_obj))
66         sys.exit(1)
67
68     try:
69         print(json.dumps(result))
70     except Exception as e:
71         error_obj = {"error": "Kann Ergebnis nicht serialisieren", "details": str(e)}
72         print(json.dumps(error_obj))
73         sys.exit(1)
74
75
76 if __name__ == "__main__":
77     main()
```

Abbildung A.1.: Python-Codegerüst für die Validierungsalgorithmen