Schlussbericht
II. Eingehende Darstellung zum Forschungsvorhaben SelfAutoDOC: Self-healing Automotive Driving platform powered by Organic Computing

Schwerpunkt Goethe Universität (UGOE)

Prof. Dr. Uwe Brinkschulte

Goethe Universität Frankfurt am Main

[1. Juli 2025]

Zuwendungsempfänger: Goethe Universität Frankfurt am Main

Förderkennzeichen: 19A21044C

Laufzeit des Vorhabens: 01.12.2021 – 28.02.2025

Inhaltsverzeichnis

1 Eingehende Darstellung

1. **Erzieltes Ergebnis:**
   Die UGOE konnte folgende wesentliche Ergebnisse erzielen:

   - Erfolgreiche Portierung des AHS/ADNA-Systems auf die Zielhardware, was eine flexible Selbstkonfiguration und Selbstheilung der Steuergeräte ermöglicht.
   - Entwicklung und Implementierung von Algorithmen, die selbstheilende Maßnahmen bei Ausfällen von Steuergeräten initiieren. Hierzu wurden Simulationen und erste Hardwaretests durchgeführt. Dies geschah insbesondere durch die Bereitstellung einer Server-Infrastruktur für die Simulationsumgebung im Carla-Simulator samt deren Implementierung für das Gesamtprojekt. Dies ermöglichte Forschungsarbeiten im Bereich des autonomen Fahrens, der Redundanz und Selbsheilungseigenschaften des OC Ansatzes mittels geeigneter ADNA.
   - Integration von Diagnosemodellen, die auf Basis der synthetisierten ADNA konkrete Zustandsvariablen überwachen und im Fehlerfall automatisierte Reaktionsstrategien ableiten.
   - Erste formale Verifikationsansätze für die Middleware-Algorithmen, die den selbstorganisierenden Mechanismus unterstützen.
   - Erzeugung von Fehlern für das systematische Testen von den Selbst-x Eigenschaften.
   - Demonstrator: Bereitstellung einer Simulationsumgebung und durchführung von Reproduzierbaren Tests.

1.1 Organic Computing

In diesem Abschnitt wird das Orgenic Computing Konzept ausführlich dargestellt. Dabei werden die Methoden das ADNA-basierte Organic Computing detailliert beschrieben.

1.1.1 ADNA-basiertes Organic Computing

Das ADNA-basierte Organic Computing (OC) orientiert sich am endokrinen System von Säugetieren, welches Hormone als Botenstoffe verwendet, um biologische Prozesse zu steuern. Hormone wirken auf Zellen ein und lösen verschiedene Mechanismen aus, was zu einer Selbstregulation der biologischen Abläufe führt. Der Organismus kann sich selbst heilen, Zellen organisieren sich beim Wachstum eigenständig und konfigurieren sich gemäß ihrer Aufgaben. Zudem kann sich der Organismus gegen Angriffe schützen. Jede Zelle erfüllt dabei spezifische Funktionen. Die daraus abgeleiteten *Self-X*-Eigenschaften bilden die Grundlage des OC-Konzepts. Der Bauplan der Zellen ist in der DNA enthalten. ADNA und AHS stellen digitale Entsprechungen dieses biologischen Prinzips dar, die auf klassische Computerhardware übertragen werden.

1.1.2 Konzept von ADNA

Das **ADNA-basierte Organic Computing (OC)** ist schichtweise aufgebaut. Die OC-Middleware wird durch den sogenannten *DNA-Prozessor* realisiert, während die ADNA den Bauplan zur Definition von Funktionen innerhalb eingebetteter Systeme darstellt.

ADNA definiert den Funktionsumfang einer Embedded-Systems-Anwendung durch sogenannte *DNA-Bausteine*. Jeder dieser Bausteine entspricht einer spezifischen Aufgabe, wie z. B. das Überwachen von Sensorsignalen des Bremssystems oder das Steuern des Lenksystems durch das Setzen eines Zielwerts für den Lenkaktuator. Abbildung 1 zeigt ein einfaches Beispiel mit fünf DNA-Bausteinen.

Der **Basisbaustein - Basic Building Blocks (BBB)** fungiert als Vorlage (Klasse), aus der durch Parametrierung konkrete Aufgaben (Objekte) erzeugt werden. In Abbildung 1 steht beispielsweise in Zeile 1 die Aufgabe *1 = 70 (1:2.2) 100 25*, was bedeutet: Aufgabe 1 ist eine Konstante, die über Destination Link 1 an Source Link 2 von Aufgabe 2 alle 25 ms den Wert 100 sendet. Jeder BBB besitzt:

- Eine **Klassen-ID**, die den Baustein als Sensor (z. B. Klassen-ID = 500) oder Aktor (z. B. Klassen-ID = 600) identifiziert.
- **Source Link** n und **Destination Link** m, die zur Kommunikation der Blöcke untereinander verwendet werden (siehe Abbildung 2 und 1).
- Eine **Aufgaben-ID (Task ID)**, die den Task T in der DNA-Datei identifiziert und gleichartige Bausteine unterscheidbar macht (z. B. Zeile 4 in Abbildung 1 mit Klassen-ID = 600).
- Eine **Parameterliste**, welche spezifische Parameter des BBBs enthält (siehe Abbildung 2 und Tabelle 1), z. B. die Adresse einer Hardwarekomponente (Resource ID), Sendeperiode, Konstantwerte oder Operatoren. Beispiel: In Abbildung 3 greift der Sensor- und Aktorblock auf eine bestimmte Hardwareadresse zu (z. B. Resource ID = 501). Der Offsetblock addiert beispielsweise den Wert 2 auf seinen Eingang (Ausgabe von Task Resource ID = 502). In Task 5 in Tabelle 1 ist Resource ID = 2.

Komplexere Aufgaben lassen sich durch sogenannte **Compound Building Blocks (CBBs)** realisieren. Diese bestehen aus mehreren BBBs und entlasten die Kommunikationsstruktur im Hormonzyklus. Beispiel: Die Flugsteuerung einer Drohne kann aus dutzenden Tasks bestehen, lässt sich jedoch durch CBBs auf wenige Tasks reduzieren.

Die **DNA-Datei** fungiert als digitaler Bauplan für eine spezifische Anwendung in einem eingebetteten System. Sie besteht aus verknüpften DNA-Bausteinen und liegt jedem *DNA-Prozessor* lokal vor (siehe Abbildung 4). Ein DNA-Prozessor lässt sich im Allgemeinen als *Processing Element (PE)* oder *Processing Unit (PU)* verstehen.

1.1.3 OC-Architektur

Im Kontext des Organic Computing (OC) spielt der **DNA-Prozessor** eine zentrale Rolle und kann unterschiedliche Ausprägungen annehmen, darunter *General Purpose Processor* (GPP), *I/O-Prozessor* und *Lab-Prozessor* (LP). Ein DNA-Prozessor kann

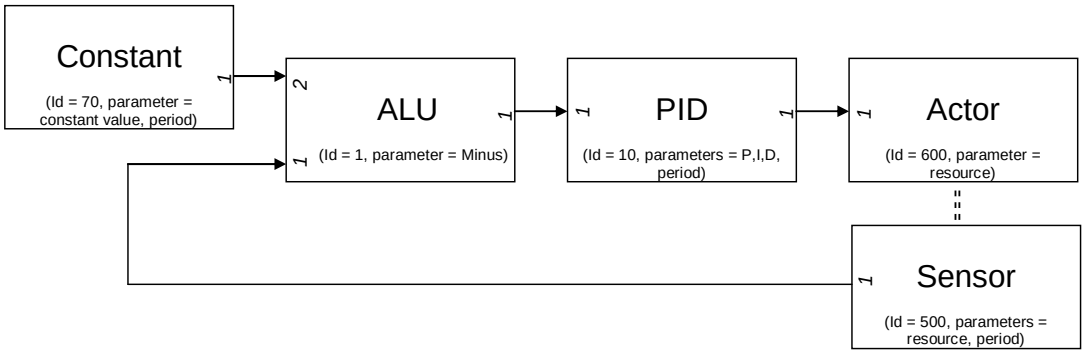

Abbildung 1: Ein einfacher Regelkreis bestehend aus fünf Basiselementen: einer ALU, einer Konstante, einem PID-Regler, einem Aktor und einem Sensor [3].

Tabelle 1: ADNA-Liste wie in einer DNA-Datei dargestellt [3]. Entspricht dem Regelkreis aus Abbildung 1. Die Tabelle enthält (von links nach rechts): Aufgaben-ID T, Klassen-ID, Kommunikationskanäle n, m, Parameterliste und Kommentare. Kommunikationsverbindungen sind in Klammern im Format $(n : T.m)$ kodiert.

Aufgaben-ID	**Klassen-ID**	**Kanäle**	**Parameterliste**	**Kommentar**
1	= 70	(1:2.2)	100 25	// Konstanter Sollwert, Zykluszeit 25 ms
2	= 1	(1:3.1)	–	// ALU, Regeldifferenz (Minus)
3	= 10	(1:4.1)	4 5 6 25	// PID-Werte ($P = 4, I = 5, D = 6$), Zykluszeit 25 ms
4	= 600		1	// Aktor, Resource ID = 1
5	= 500	(1:2.1)	2 25	// Sensor, Resource ID = 2, Zykluszeit 25 ms

entweder eine reale CPU darstellen oder auf die begrenzten Ressourcen eines einzelnen Kerns zurückgreifen. Bereits beim Entwurf der DNA-Prozessoren wird festgelegt, wie viele und welche Arten von Aufgaben sie ausführen können.

Ein GPP besitzt beispielsweise einen sogenannten *Eager-Wert* (eine Art Eignungs- oder Bereitschaftswert) von 14 für alle Basisbausteine (BBB) und einen Eager-Wert von 11 für unbekannte oder noch nicht definierte Bausteine wie Compound Building Blocks (CBBs). Der I/O-Prozessor hingegen ist speziell für Sensor- und Aktor-Bausteine optimiert und weist hierfür einen Eager-Wert von 20 auf. Dadurch werden diese Aufgaben ausschließlich und bevorzugt von ihm übernommen, bevor sie an den GPP übergeben würden.

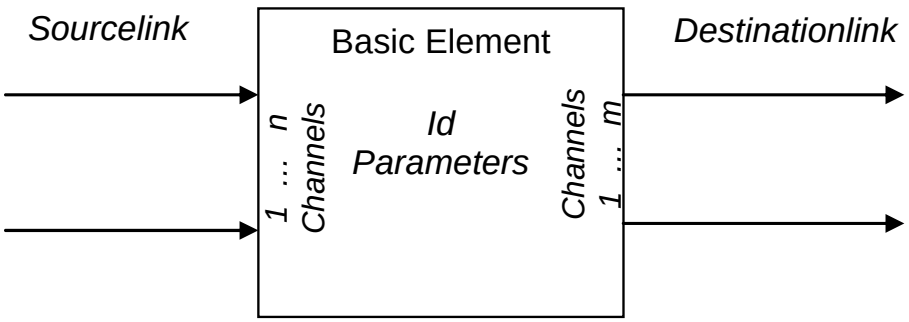

Abbildung 2: Basiselement mit Kommunikationsverbindungen, Klassen-ID und Parameterliste [3]. Die Abbildung zeigt 1 bis n Kanäle für den Source Link und 1 bis m Kanäle für den Destination Link. Nicht jeder Block muss sowohl Eingangs- als auch Ausgangsverbindungen besitzen.

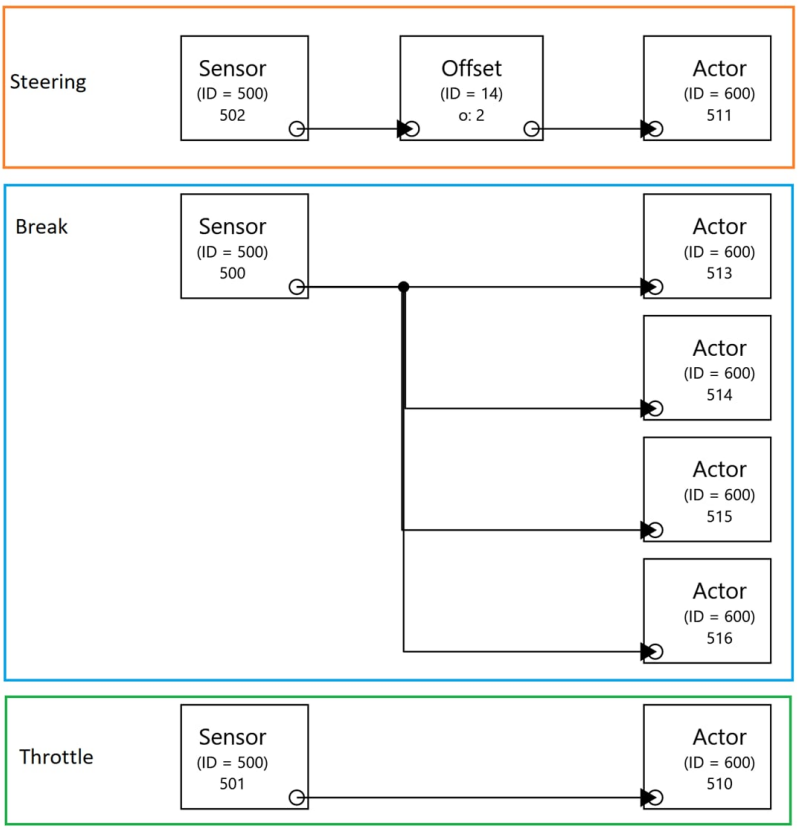

Abbildung 3: Übersicht der DNA: Einfache Automobilanwendung mit den drei Hauptkomponenten für manuelles Fahren: Lenken (orange), Bremsen (blau) und Gas geben (grün) [8]. Jede Hauptfunktion verfügt über einen Sensor- und einen Aktorblock, die Bremsen zusätzlich über vier Aktoren für jedes Rad. Die Parameterlisten geben die Resource IDs (Hardwareadressen) oder Konstantwerte an.

Der Lab-Prozessor ist anfangs so ausgelegt, dass er keine Aufgaben ausführt. Seine Aufgabe besteht vielmehr darin, über hormonelle Kommunikation die DNA-Prozessoren zu beeinflussen – vergleichbar mit einem Medikament, das über das endokrine System des Menschen (z. B. Jod zur Beeinflussung des Schilddrüsenhormons) wirkt. Die Kommunikation zwischen den DNA-Prozessoren erfolgt über **UDP-Broadcast** und dient dem Austausch von Hormonwerten.

Die Entscheidung, einen LP statt einer klassischen Laborsimulationsanwendung zu integrieren, ist bewusst getroffen worden. Der LP ist fester Bestandteil der OC-Middleware, basiert auf der GPP-Architektur und ist somit einfacher zu implementieren. Gleichzeitig unterscheidet er sich funktional deutlich vom GPP, verbraucht weniger Ressourcen als eine vollständige Simulation und ermöglicht dennoch eine garantierte Kommunikation im OC-System.

Das **AHS** (Autonomous Hormone System) fungiert als Middleware und realisiert die *Self-X*-Eigenschaften des OC. Es verwaltet Systemressourcen, initialisiert ein eigenes *AHS-Netzwerk*, lädt und repliziert die DNA-Datei und verteilt die darin gelisteten Aufgaben auf die vorhandenen DNA-Prozessoren. Darüber hinaus ermöglicht das AHS die hormonelle Kommunikation und nutzt den sogenannten **DNA-Builder**, um die Schnittstelle zwischen Software und Hardware bzw. Treibern zu bilden. Abbildung 5 veranschaulicht den Aufbau dieser Architektur.

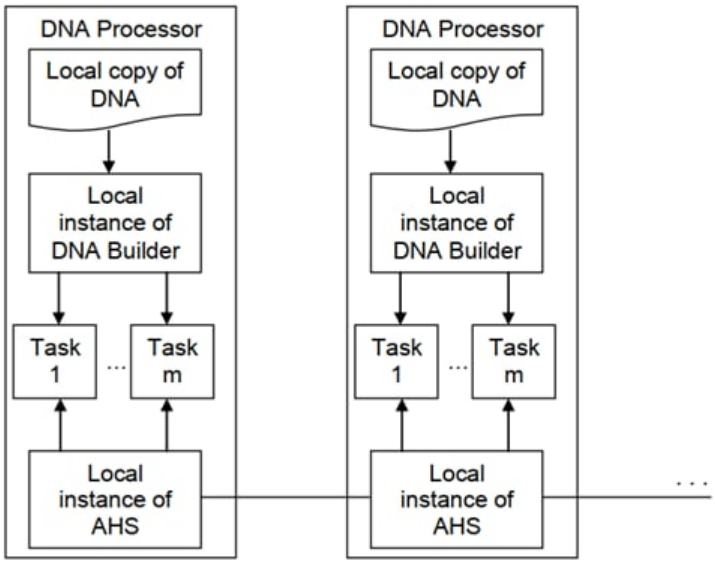

Abbildung 4: AHS-Netzwerkkonzept: DNA-Prozessor innerhalb des AHS und dessen lokale DNA-Kopien [**?**].

1.1.4 Hormone und der Hormonzyklus

Das AHS nutzt Hormone zur Kommunikation und steuert Aufgabenverteilungen über ein Auktionsverfahren, den sogenannten **Hormonzyklus** (siehe Abbildung 6). In diesem Zyklus bieten sogenannte **Processing Elements (PEs)**[1] auf Aufgaben. Ob ein PE eine Aufgabe ausführen kann, hängt von seinem aktuellen Hormonspiegel ab.

Drei zentrale Hormontypen steuern das Verhalten im System:

- **Suppressoren** senken den Eager-Wert eines PEs, sobald dieser eine Aufgabe ausführt. Dadurch wird verhindert, dass mehrere PEs dieselbe Aufgabe gleichzeitig übernehmen. Wird eine Aufgabe nicht ausgeführt, fehlen die entsprechenden Suppressoren, sodass andere PEs sie übernehmen können. Suppressoren werden systemweit über **UDP-Broadcast** verteilt.
- **Beschleuniger (Accelerators)** dienen der Optimierung der Kommunikationszeiten. Sie fördern die Ausführung zusammengehöriger Aufgaben (z. B. mehrere BBBs für Lenkung, Bremse und Gaspedal in Abbildung 3) auf demselben PE. Sie werden lokal (lokaler Broadcast) an benachbarte PEs gesendet, wodurch sich sogenannte *künstliche Organe* bilden. Dies erhöht die Wahrscheinlichkeit, dass verwandte Aufgaben vom gleichen PE übernommen werden.
- **Negatoren** sind Hormone, die eine bestimmte Aufgabe vollständig unterdrücken. Wird ein Negator für eine Aufgabe gesendet, darf diese nicht mehr ausgeführt werden. Das AHS bricht deren Ausführung dann systematisch ab.

Zusätzlich gibt es im OC-System verschiedene **Systemperioden**, die zur besseren Nachvollziehbarkeit erläutert werden:

[1] Ein PE bezeichnet hier einen DNA-Prozessor mit seinem modifizierten Eager-Wert (tatsächliche Eignung) unter Berücksichtigung aller aktiven Hormone. Dies schließt auch gestartete DNA-Prozessoren ein, die keine Aufgaben ausführen, aber mit der gleichen ADNA im selben AHS-Netzwerk laufen.

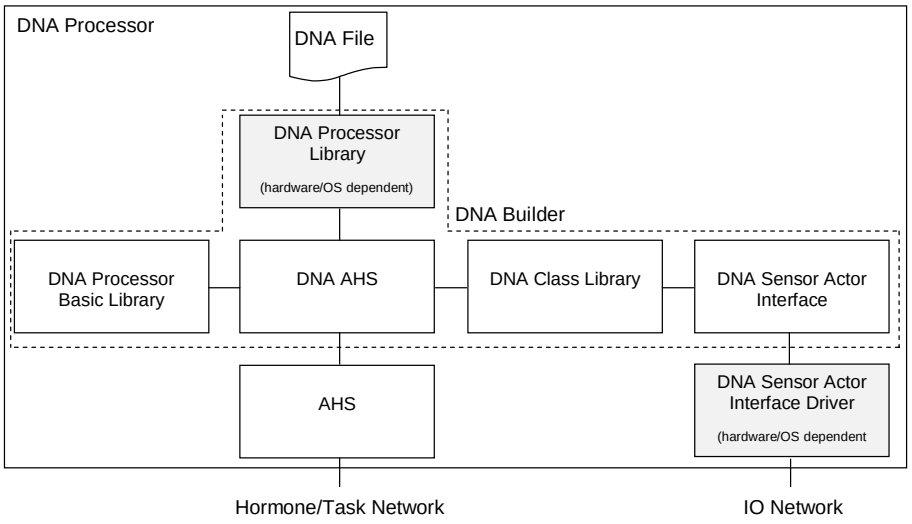

Abbildung 5: Detailansicht der OC-Middleware am Beispiel der DNA-Prozessorarchitektur [2], inklusive DNA-Datei, Hormon-/Tasknetzwerk und I/O-Treiberstruktur.

- Die **Sampling-Periode** beschreibt, in welchen Intervallen physikalische Sensoren ihre Daten liefern (z. B. ein GPS-Sensor mit 5 Hz).
- Die **Hormonperiode** definiert, in welchem Rhythmus ein DNA-Prozessor Hormone versendet. Der Standardwert beträgt 100 ms bzw. 10 Hz. Dabei handelt es sich um einen globalen Takt für das gesamte OC-System.
- Die **Task-Periode** gibt den Ausführungszyklus einzelner Aufgaben an. In manchen BBBs (z. B. Sensorblöcken) kann sie manuell festgelegt werden. Falls nicht, gilt die Hormonperiode als Standard.

Ein Vergleich der Perioden in Tabelle 1 zeigt beispielsweise, dass der Sensor in Zeile 5 seine Werte alle 25 ms, also mit 40 Hz sendet. Die Systemperioden sind somit grundsätzlich asynchron. In der Praxis ist es daher empfehlenswert, die Sampling-Periode des Sensors mit der Task-Periode abzugleichen, um Datenverluste oder verwaschene Messgenauigkeit zu vermeiden.

1.2 Selbstheilung

Ziel des vorliegenden Projekts war die Entwicklung und Evaluierung von Methoden zur Verbesserung der Selbstheilungsfähigkeit von ADNA-basierten Organic Computing (OC)-Systemen. Obwohl diese Methoden grundsätzlich domänenunabhängig einsetzbar sind, wurde ihre erste Validierung und experimentelle Erprobung exemplarisch im automobilen Umfeld vorgenommen – konkret in der Umgebung elektronischer Steuergeräte (ECUs). Durch den Einsatz von ADNA-basierter Selbstheilung in ECUs konnte die Abhängigkeit von spezifischen Halbleiterkomponenten reduziert werden. Dadurch ergeben sich Vorteile hinsichtlich der Resilienz gegenüber Lieferengpässen, der Kostensenkung sowie der ökologischen Nachhaltigkeit zukünftiger Fahrzeugsysteme.

Ein weiteres Ziel bestand darin, die Basis für fehlertolerante Fahrzeugsysteme zu schaffen, die ohne klassische, hardwareintensive, dreifache Redundanz auskommen. Diese Vision konnte durch eine adaptive, softwarebasierte Selbstheilungslogik im Rahmen des

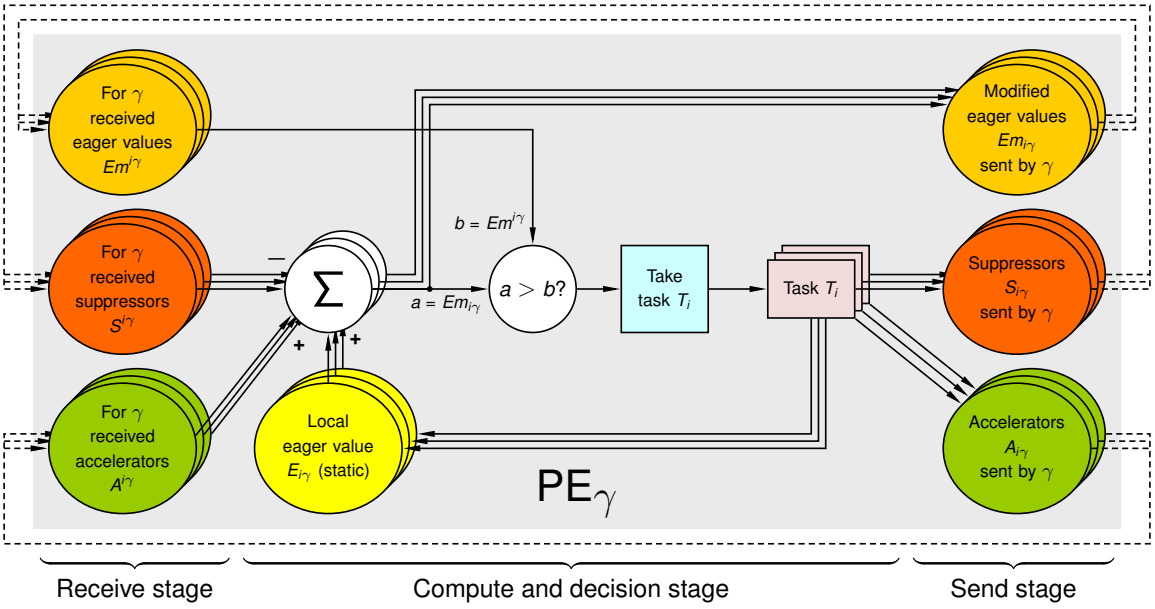

Abbildung 6: Grundprinzip des AHS dargestellt als Hormonzyklus [7].

OC-Ansatzes realisiert werden. Dafür ist die Erzeugung von Fehlern (Singularitäten) essenziell.

Im Rahmen des Forschungsprojektes wurde erkannt, dass Selbstheilung eine dynamische Form der Redundanz ist. Entsprechend wurde nachgewiesen, dass der OC-Ansatz mit der ISO26262 konform ist und alle Redundanzstufen abgebildet werden können. Dies ermöglicht es die bewährten klassischen Redundanzkonzepte nachzubilden und diese mit den Selbst-X Eigenschaften des OC zu verbinden.

Ausblick

Darüberhinaus waren neue Konzepte denkbar, was zur Entwicklung folgender Ansätze führte:

- Selbstheilung für den Fail-Operational Betrieb eines Fahrzeugs,
- Selbstheilung als Redundanz,
- Adaptive Redundanzen,
- Dreifach-ADNAs für Redundanzmodi,
- Neue ADNA Basis Bausteine für verschiedene Redundanzmodi,
- Algorithmen integration für Temperatur-, Leistung- und Frequenzüberwachung für die Proaktive Selbstheilung,
- Dynamisches Zu- und Abschalten von Redundanzebenen,
- Implementierung neuer DNA Prozessoren (Labor Prozessor, Netzwerk Prozessor, Temperatur Prozessor),
- Vorbereitung für Grundlagenforschung im Bereich der Selbst-X Eigenschaften.

Die Vertiefte Betrachtung der Redundanz mit dem OC-Ansatz kann als Grundlage für die Normungsgremien dienen, die geltenden Normen um die OC-Methoden zu erweitern, was künftige Entwicklungen begünstigt und Kosten bei der Entwicklung und bei der eingesetzten Hardware spart. Dies hat positive ökologische und ökonomische Effekte für den deutschen Automarkt und kann einen Wettbewerbsvorteil bedeuten.

1.2.1 Reaktive Selbstheilung

Reaktive Selbstheilung: Erzeugung von Fehlern

Mittels kontrollierter Testszenarien, darunter Fahrsimulationen mit gezielt eingeleiteten ECU-Fehlern, wurde demonstriert, wie ADNA-basierte OC-Systeme traditionelle Redundanzen durch effiziente Selbstheilungsmechanismen ersetzen können. Die gezielte Reproduzierbarkeit von Fehlern in einer kontrollierten Umgebung ermöglichte eine detaillierte Nachverfolgung von Ursachen und Auswirkungen, wodurch ein vertieftes Verständnis für generische Fehlerreaktionen innerhalb ADNA-basierter OC-Anwendungen gewonnen werden konnte.

Systemdesign

Das AHS fungiert als Middleware des OC-Systems. Es wird direkt auf einem Steuergerät (ECU) ausgeführt und nutzt *Processing Units* (PUs) als ausführende Recheneinheiten. Diese PUs repräsentieren virtuelle *Processing Elements* (PEs) bzw. DNA-Prozessoren[2] und erlauben den Zugriff auf Systembibliotheken des Betriebssystems.

Mehrere Steuergeräte (ECUs) können kooperativ zusammenarbeiten, um die im ADNA-Modell beschriebenen Fahrzeugfunktionen bereitzustellen. Sensoren und Aktoren des Fahrzeugs werden über Hard- oder Softwaretreiber angesprochen, welche im AHS integriert sind. Sie bilden die Schnittstelle zwischen der realen Fahrzeughardware und der softwarebasierten AHS-Logik.

Dieses Konzept ist in Abbildung 7 dargestellt, welche die klassische Architektur mit dreifacher Hardware-Redundanz und Mehrheitsentscheidung illustriert. Im Gegensatz dazu zeigt Abbildung 8 ein weiterentwickeltes Konzept für eine fehlertolerante Fahrzeugarchitektur auf Basis von ADNA und OC, das sowohl Redundanzen als auch funktionale Singularitäten integriert.

Diskussion

Um die Diagnose zu unterstützen erfolgt eine ausführliche Betrachtung der möglichen Fehlerquellen. Ausgehend von den Projektzielen ergeben sich die folgenden zentralen Forschungsfragen, die im Verlauf der Arbeiten adressiert und bearbeitet wurden:

(a) **Welche Anforderungen bestehen für die Durchführung reproduzierbarer Experimente?**
Um reproduzierbare Experimente durchzuführen, wurden etablierte wissenschaftliche Methoden angewendet. Ein Experiment gilt als reproduzierbar, wenn unter festgelegten Bedingungen das erwartete Ergebnis erneut eintritt oder innerhalb einer bestimmten Toleranz bestätigt wird. Diese Deterministik konnte in Simulationsumgebungen besonders gut realisiert werden, da externe Störungen weitgehend

[2]Hinweis: In der Literatur werden PU und PE häufig synonym verwendet, was zu Verwirrung führen kann. Während PE die softwareseitige Ausprägung beschreibt (z. B. die Eignung eines DNA-Prozessors zur Bearbeitung bestimmter Aufgaben – also seine momentane Ëagerness"), bezieht sich PU auf die hardwareseitige Einheit (z. B. ein Kern oder Rechenknoten einer CPU). Innerhalb von ADNA-basierten OC-Systemen können beide Begriffe jedoch als Ressourcen zur Ausführung von Aufgaben interpretiert werden.

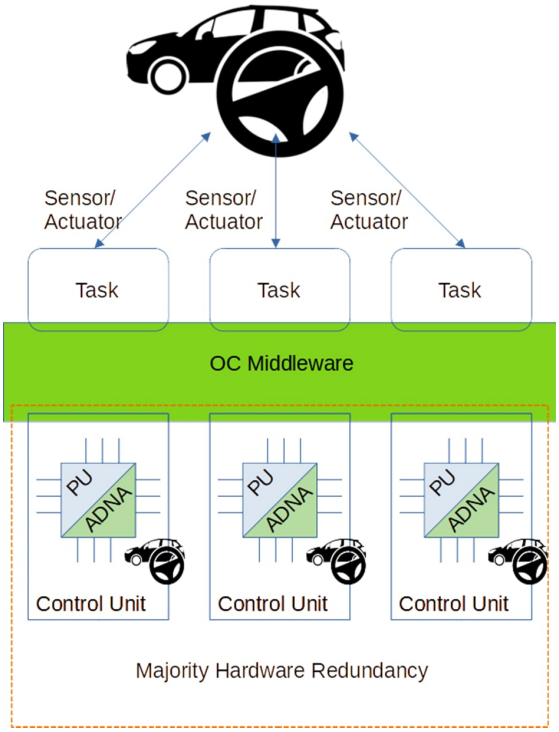

Abbildung 7: OC-basierte Illustration einer dreifachen Hardware-Redundanz mit Mehrheitsvoting gemäß ISO 26262-Anforderungen. [5]

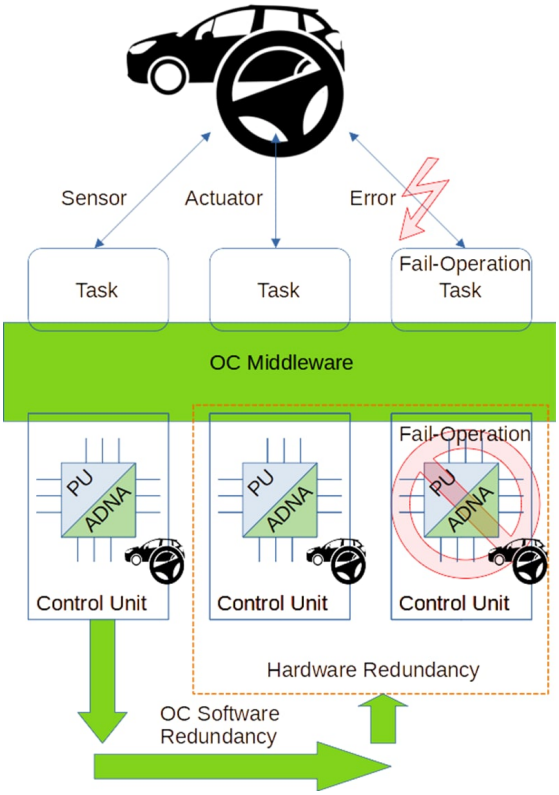

Abbildung 8: Konzept für ADNA-basiertes OC mit funktionalen Singularitäten und adaptiven Redundanzen. [5]

ausgeschlossen sind. Reale Tests wurden bevorzugt zunächst unter Laborbedingungen durchgeführt, um anschließend durch Simulation verifiziert zu werden. Hierfür wurden geeignete Versuchsaufbauten wie Prüfstandprototypen, Laborfahrzeuge oder offene Teststrecken konzipiert. Aufbauend darauf konnten dann Testfahrten in realitätsnahen Umgebungen erfolgen.

Die Durchführung reproduzierbarer Simulationen im Kontext von ADNA-basiertem OC ist unter anderem in [1] beschrieben. Die Analyse zeigt, dass geskriptete Tests den Anforderungen an Reproduzierbarkeit gerecht werden. Damit sind statistische Auswertungen, Erwartungsüberprüfungen und die Identifikation von Anomalien möglich.

(b) **Welche Fehlerquellen sind zu berücksichtigen?**
Fehlerursachen im Fahrzeugumfeld sind vielfältig und reichen von äußeren Einflüssen (Temperatur, Staub, Feuchtigkeit, Vibration) bis hin zu betriebsbedingten, materialtechnischen, produktionsbedingten oder systemisch-konstruktiven Faktoren. Auch komplexe Fehlerketten durch emergentes Verhalten oder Kaskadeneffekte wurden betrachtet.

Im Rahmen des Projekts wurden diese Ursachen auf verallgemeinerbare Fehlerkategorien reduziert: *Totalausfall* (z. B. Ausfall eines gesamten Steuergeräts oder Netzwerks), *Teilausfall* (z. B. Ausfall einzelner PEs) sowie *Funktionsausfall* (z. B. Lenkung, Bremse, Sensorik oder Aktuatorik sowie ganzer Assistenzsysteme).

(c) **Welche Ansätze bietet ADNA-basiertes OC zur Erzeugung von Singularitäten?**

Das AHS (Autonomous Hormone System) als dezentrale Middleware im Echtzeitbetrieb wurde so gestaltet, dass es durch seine Self-* Eigenschaften gegenüber Fehlern und Störungen robust ist. Nach dem Systemstart werden die jeweiligen ADNAs geladen und die Fahrzeugfunktionen (Lenkung, Bremse, Antrieb etc.) über PEs ausgeführt. Der Aufbau einer ADNA bietet dabei nur wenige Angriffsflächen. Solange Rechenressourcen verfügbar sind (gesteuert durch die *Eager*-Werte), werden Aufgaben ausgeführt. Nur durch den Ausfall einzelner oder aller PEs bzw. eines gesamten ECUs kann dies verhindert werden.

Störungen in verteilten Experimenten entstehen z. B. durch doppelt vergebene Task-IDs im Netzwerk oder widersprüchliche PE-IDs zwischen Simulationsinstanzen. Auch das gleichzeitige Laden inkompatibler ADNAs (z. B. Flugobjekt vs. Fahrzeugsteuerung) kann zu Fehlverhalten führen. Solche Störungen können durch die gezielte Vergabe von Netzwerk-IDs vermieden werden.

(d) i. **Wie können Singularitäten das Gesamtsystem bzw. eine ECU unterbrechen?**
Ein *Sleep-Signal* kann das gesamte System in einen Ruhemodus versetzen. Durch ein *Wake-Signal* wird es wieder aktiviert. Ebenso lassen sich PEs gezielt deaktivieren oder abschalten. Diese Zustände sind reversibel und ermöglichen reproduzierbare Totalausfälle für Untersuchungen.

ii. **Wie können Singularitäten einzelne Systemkomponenten bzw. PEs unterbrechen?**
Einzelne PEs lassen sich separat deaktivieren und reaktivieren. Die resultierende Last wird von den verbleibenden PEs übernommen – ein Beispiel für direkte Selbstheilung durch Task-Umverteilung. Dies erlaubt die gezielte Simulation und Untersuchung von Teilfunktionsstörungen.

iii. **Wie können Singularitäten gezielt einzelne Fahrzeugfunktionen stören?**
Die gezielte Störung einzelner Aufgaben ist aufwändiger:

A. Aufgaben können fest an bestimmte PEs gebunden werden. Wird der zuständige PE deaktiviert, übernimmt ein anderer PE die Aufgabe – Selbstheilung durch Umverteilung.

B. Durch *Negatoren* lassen sich Aufgaben gezielt unterdrücken. Solange diese gesendet werden, wird die Ausführung verhindert. So können spontan und zeitlich gesteuert Fehler simuliert werden. Die Selbstheilung erfolgt, sobald die Negatoren entfernt werden. Eine geeignete Implementierung befindet sich in Entwicklung.

C. Kommunikationsstörungen können durch fehlerhafte Adressierung simuliert werden, z. B. durch das Senden von Nachrichten an falsche IPs. Die Aufgaben laufen weiter, aber die Kommunikation ist gestört – reversibel simulierbar.

D. Mit einer *Conditional ADNA* lassen sich alternative Funktionen bei Ereignissen aktivieren, etwa wie beim Einlegen des Rückwärtsgangs die Rückfahrkamera aktiviert wird. Eine alternative ADNA überlagert dabei temporär die ursprüngliche Funktion.

E. Bei einer *Modified ADNA* wird zur Laufzeit die funktionierende ADNA durch eine fehlerhafte ersetzt. Dies geschieht durch kurzes Unterbrechen mit Negatoren und das Nachladen neuer Tasks. Diese Methode ist reversibel, aber nicht sofort wirksam.

Die genannten Ansätze wurden untersucht, implementiert oder prototypisch vorbereitet. Geeignete Experimente wurden konzipiert und in ersten Tests validiert.

(e) **Wie wirksam ist die Selbstheilung hinsichtlich der Aufrechterhaltung sicherer Fahrzeugfunktionen trotz Singularitäten?**
Nach Auftreten einer Singularität muss das System innerhalb kürzester Zeit wieder in einen sicheren Betriebszustand überführt werden. Dies erfordert die vollständige Wiederherstellung der ursprünglichen ADNA und die unverzögerte Reaktivierung sicherheitskritischer Aufgaben (z. B. Bremsfunktion). Erste Untersuchungen zeigen, dass dies innerhalb weniger Millisekunden realisierbar ist. Weitere Studien zur Bewertung der Effektivität und zur Substitution klassischer Redundanzmechanismen sind in Arbeit.

Implementierung eines Labor Prozessors

Zur Erfüllung der Forschungsziele sind reproduzierbare und gezielt steuerbare Tests erforderlich, welche durch einen Labor Prozessor durchgefürht werden können. Ein geeignetes Testszenario besteht in der Simulation des Fahrverhaltens bei Ausweichmanövern (z. B. ein Fahrzeug nähert sich einem parkenden Auto). Hierzu wird ein Simulator (z. B. CARLA) mit der OC-Middleware verbunden. Die Fahrzeugfunktionen werden über eine ADNA in einer DNA-Datei abgebildet. Abbildung 3 zeigt eine mögliche DNA für manuelles Fahren.

Zur Sicherstellung der Reproduzierbarkeit wird ein Skript verwendet, das im Hintergrund des Simulators läuft und eine identische Simulationsumgebung sowie Fahrzeugaktionen für jeden Testlauf definiert. Dadurch verlaufen die Tests unter gleichen Bedingungen und die Ergebnisse sind vergleichbar. Der Versuchsaufbau umfasst die Inbetriebnahme des Lab Processors (LP) sowie mehrerer General Purpose Processors (GPPs), siehe Abbildung 9. Die GPPs führen die ADNA aus, während der LP Singularitäten in Echtzeit oder skriptbasiert in das System einbringen kann.

Das System ist inzwischen funktional umgesetzt. Es ist möglich, einzelne oder mehrere Tasks zur Laufzeit sowohl lokal (auf einem bestimmten PE) als auch global (systemweit) zu deaktivieren und wieder zu aktivieren. Diese Steuerung erfolgt skriptgesteuert. Hierbei wird zunächst ein *Negator* gesendet, um den jeweiligen Task aus der Ausführungsliste zu entfernen. Anschließend werden in festgelegten Hormonperioden *Suppressoren* versendet, die das Nachladen des Tasks blockieren. Auf diese Weise lassen sich Tasks temporär deaktivieren und später wieder aktivieren.

Die Simulation startet, sobald die DNA auf die DNA-Prozessoren verteilt ist und das Fahrskript ausgeführt wird. In einem Beispiel fährt das Fahrzeug nach einem definierten Zeit- oder Geschwindigkeitswert los und führt ein Ausweichmanöver durch. Mithilfe des LP können währenddessen Tasks deaktiviert und ihre Auswirkungen auf die Fahrzeugtrajektorie analysiert werden.

Dieser Prozess wird für verschiedene DNAs und Szenarien wiederholt, um die Systemreaktionen zu untersuchen. Die Auswirkungen der Unterbrechungen und die Fähigkeit zur Selbstheilung lassen sich dabei gezielt beobachten. Verschiedene Lenk- und Geschwindigkeitswerte werden zusätzlich untersucht. Ebenso wird die Wirkung von Redundanzen im Systemverhalten betrachtet. Abbildung 8 zeigt beispielhafte Singularitäten in schematischer Darstellung.

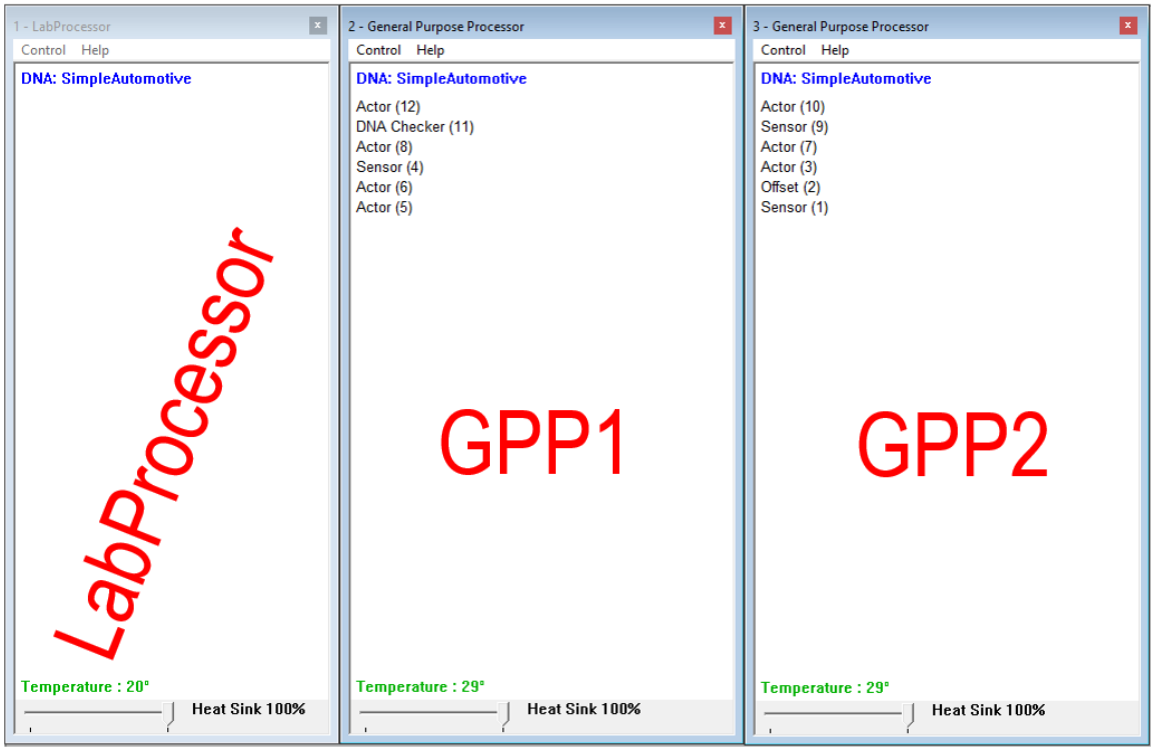

Abbildung 9: Anordnung der DNA-Prozessoren für ein mögliches Testszenario. Links ist der Lab Processor dargestellt, der über die DNA:SimpleAutomotive in die AHS-Kommunikationsschicht integriert ist. In der Mitte und rechts sind die General Purpose Processors dargestellt, die die in der DNA definierten Fahrzeugfunktionen ausführen.

Ausblick und Fazit

Die Reproduzierbarkeit von Experimenten im Kontext ADNA-basierter OC-Systeme wurde erfolgreich umgesetzt. Skriptbasierte Methoden ermöglichen die gezielte Erzeugung von Singularitäten auf Task-Ebene, sowohl für einzelne Komponenten als auch für das Gesamtsystem.

Die temporäre De- und Reaktivierung von Tasks über die Kombination aus Negator und Suppressoren ist realisiert. Damit können gezielt Funktionsunterbrechungen erzeugt und untersucht werden. Das System reagiert wie erwartet und ermöglicht eine gezielte Beobachtung des Systemverhaltens unter Störung.

Mit der vollständigen Integration des Lab Processors (LP) können weitere systematische Untersuchungen folgen. Fahrzeugtrajektorien und Systemlogs werden dabei zur Bewertung herangezogen. Zusätzlich erfolgen Tests mit verschiedenen Simulationsplattformen und Prototypen, um das Verhalten unter unterschiedlichen Bedingungen zu analysieren und weiterführende Fehlermodelle zu entwickeln.

1.2.2 Redundanzansätze nach ISO26262

Der OC-Ansatz integriert Selbstheilung als eingebauten Redundanzmechanismus. Herkömmliche OC-Architekturen verzichteten zunächst auf klassische Redundanz [12]. Inzwischen kombiniert die ADNA-basierte OC-Implementierung erfolgreich N-fache Redundanz mit Selbstheilung. N-fache Redundanz beseitigt auftretende Fehler unmittelbar und dauerhaft. Selbstheilungsprozesse hingegen benötigen eine gewisse Zeit zur Wiederherstellung der Funktionsfähigkeit.

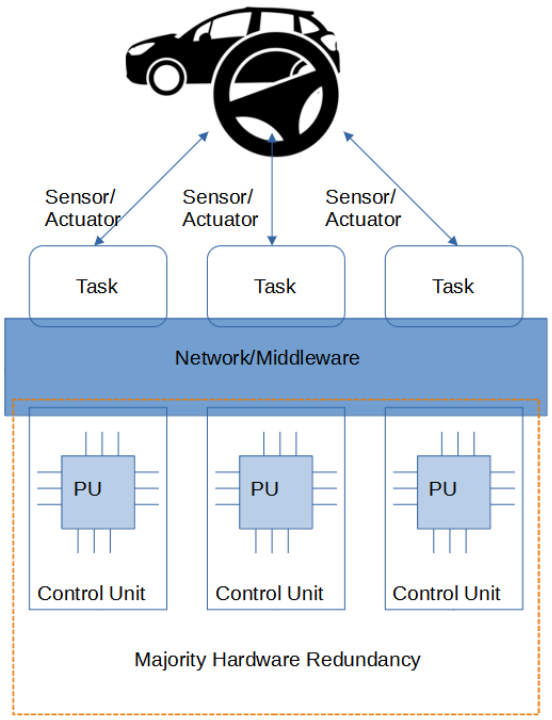

Abbildung 10: Klassischer Redundanzansatz nach ISO26262.

Die Kombination beider Ansätze gewährleistet eine hohe Ausfallsicherheit. Bei einem Fehler innerhalb eines Steuerpfads übernimmt zunächst die redundante Struktur die Funktion. Parallel dazu wird der fehlerhafte Pfad selbstständig repariert. Nach Abschluss der Selbstheilung steht die vollständige Funktionalität erneut zur Verfügung. Der fehlerfreie Pfad speichert Systemzustände über längere Zeiträume hinweg, während der selbstheilende Pfad neu initialisiert wird. Die Informationsbasis älterer Pfade ist in der Regel umfassender, kann jedoch fehlerbehaftet sein. Fehlerfortpflanzungseffekte – z. B. im Dead-Reckoning – verursachen potenziell sicherheitskritische Abweichungen. Daher ist eine Abwägung zwischen Informationsalter und Genauigkeit erforderlich.

Einfach Fahrzeug-ADNA als Redundanz ohne Voting – „ADNA Simple Automotive“

Abbildung 12 zeigt eine funktionale ADNA eines Fahrzeugs („ADNA Simple Automotive“). Die Hauptfunktionen des Fahrzeugs sind farblich hervorgehoben: Orange markiert die Lenkung, Blau die Bremsfunktion und Grün das Gaspedal. Sensor- und Aktor-ADNA-Blöcke bilden die direkten Schnittstellen zu den Ein- und Ausgängen des Simulators. Der Offset-ADNA-Block führt interne Berechnungen durch.

Abbildung 10 zeigt ein klassisches 3-fach redundantes System. Abbildung 11 veranschaulicht die Integration von AHS und ADNA in die OC-Middleware sowie das Design für das Fahrzeugumfeld.

Dreifache Redundanz ohne Voting – „ADNA 3 x Simple Automotive“

Abbildung 13 zeigt eine OC-Implementierung mit dreifacher Redundanz, jedoch ohne Mehrheitsentscheidung. Die Steuerungseinheit aus Abbildung 12 wird dreifach reali-

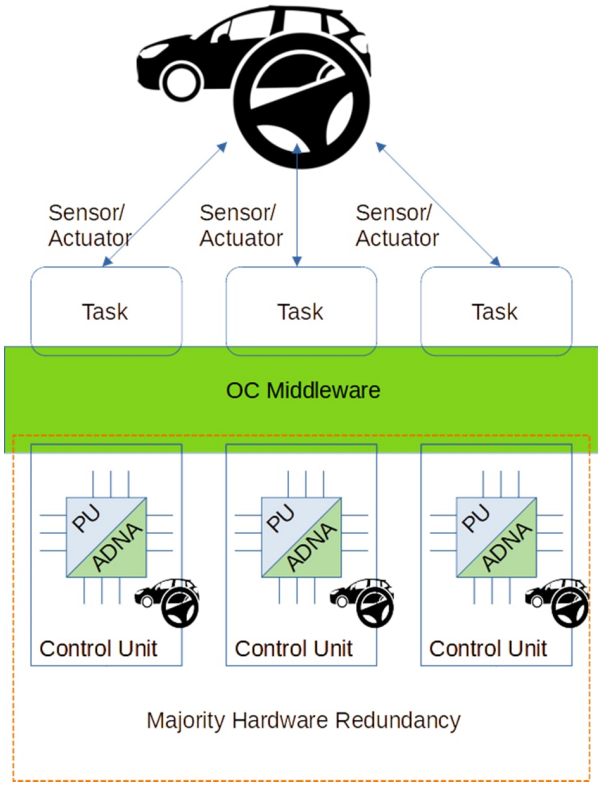

Abbildung 11: OC Redundanzansatz gemäß ISO26262.

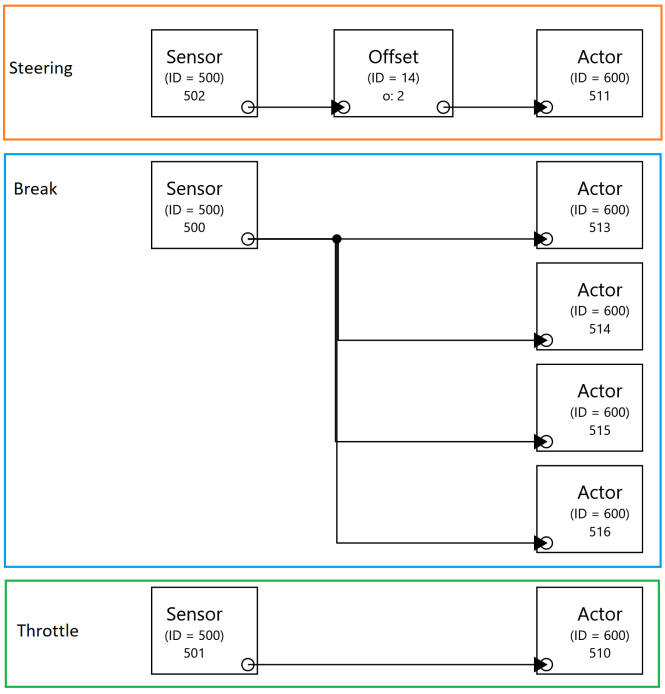

Abbildung 12: Einfache Fahrzeug-ADNA.

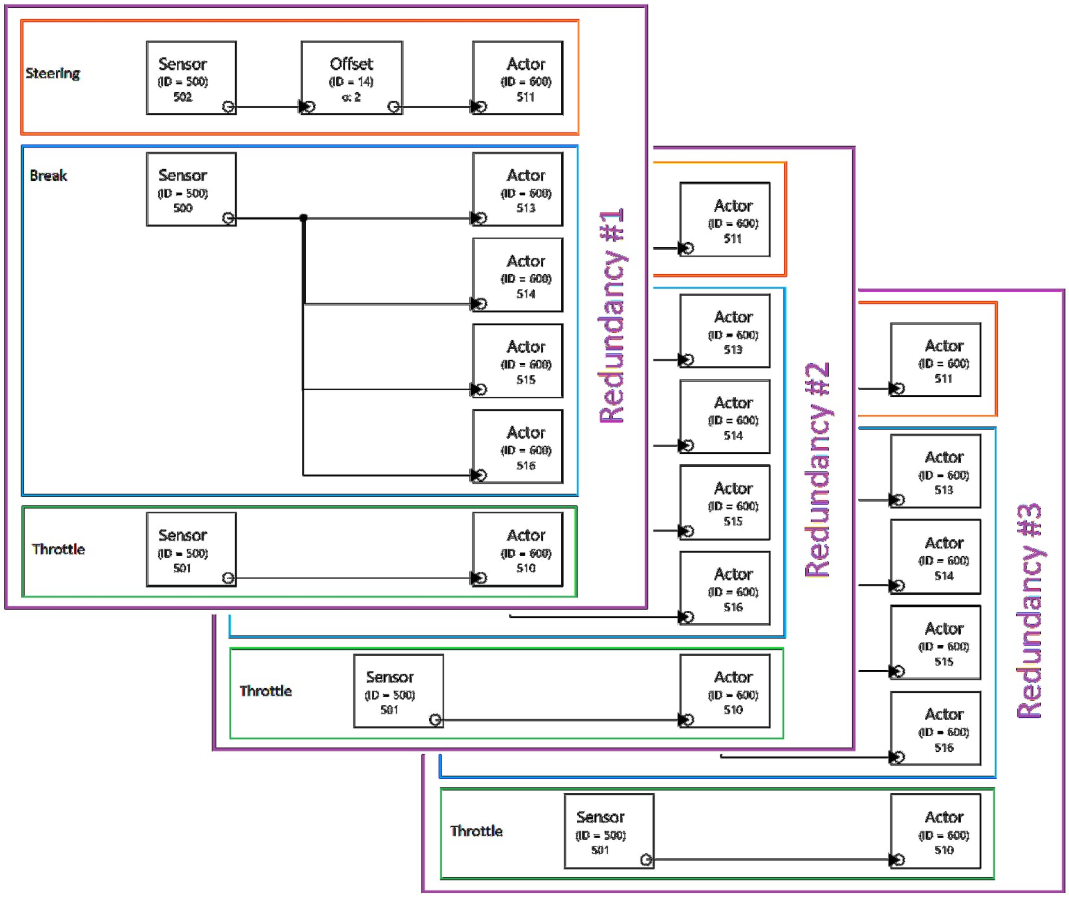

Abbildung 13: „ADNA 3 x Simple Automotive“ – Dreifach-Redundanz ohne Voting-Mechanismus.

siert. Alle drei Instanzen geben gleichzeitig Signale über das AHS aus. Das jeweils zuerst eintreffende Signal wird ausgeführt; ein neueres überschreibt das vorherige. Sensoren und Aktoren sind physikalisch jeweils nur einmal vorhanden, auch wenn die Logik dreifach vorliegt. Der Aufbau liefert eine einfache Form der Redundanz, ohne explizite Entscheidungslogik.

Dreifache Redundanz mit Mehrheitsentscheid – „ADNA 3 Red majority voting“

Abbildung 14 Zeigt den Signal weg. Wenn Signal 1 und Signal 2 ungleich sind, wird Signal 3 durchgeschaltet, ansonsten Signal 1. Analog dazu wurde die ADNA entwickelt. Abbildung 15 zeigt die ADNA für Triple-Modular-Redundanz. Drei Steuerungseinheiten erzeugen unabhängig voneinander Signale. Stimmt die Ausgabe von mindestens zwei Einheiten überein, wird das Mehrheits-Signal umgesetzt. Liefern alle drei unterschiedliche Signale, wird eines davon ausgeführt. Die Sensorsignale stammen stets aus derselben physischen Quelle. Dieses Verfahren erhöht die Systemsicherheit signifikant und minimiert Fehlentscheidungen.

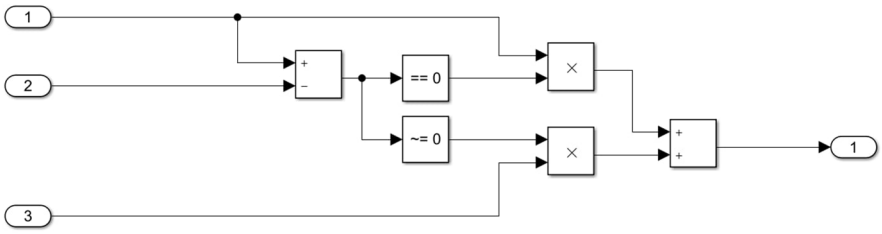

Abbildung 14: Signallogik der Triple-Modular-Redundanz mit Mehrheitsentscheid.

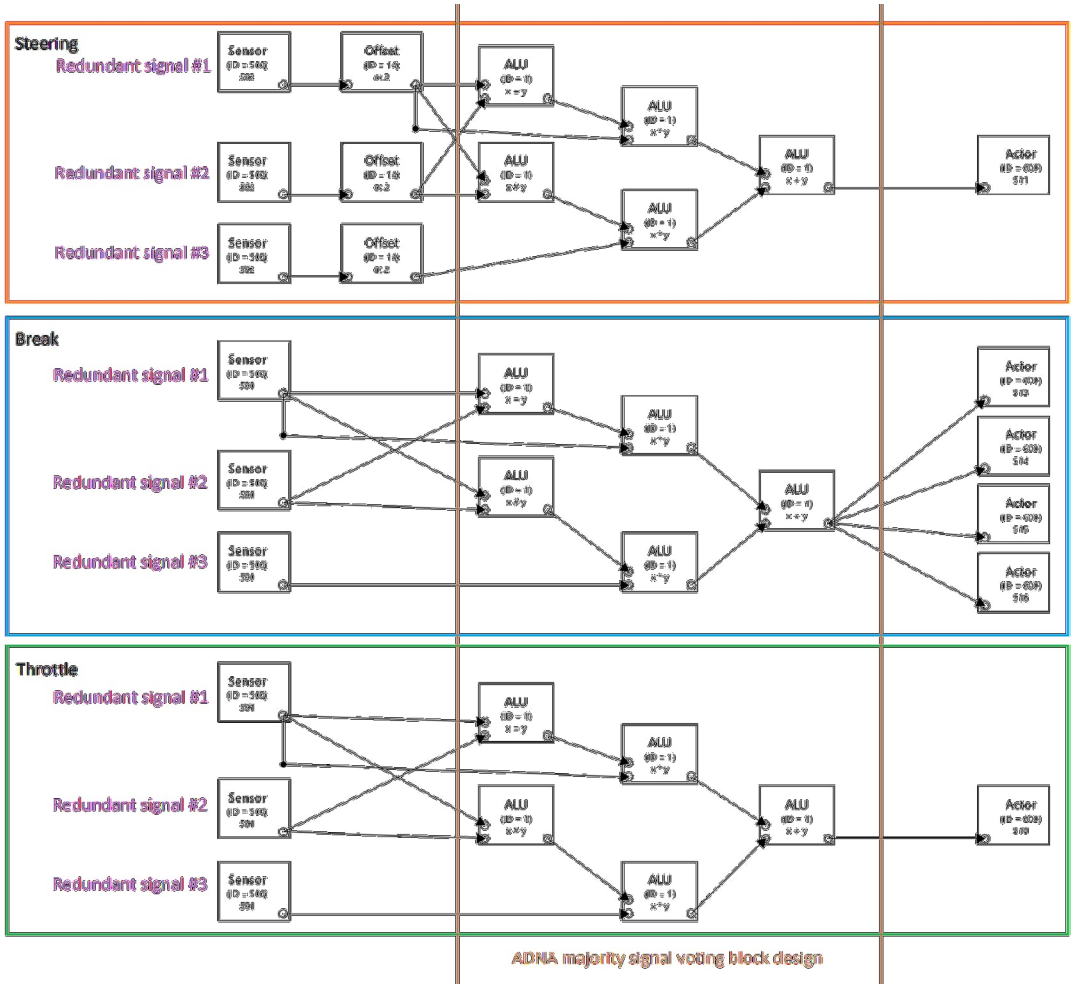

Abbildung 15: Triple-Modular-Redundanz mit Mehrheitsentscheid in einer einfachen Fahrzeug-ADNA.

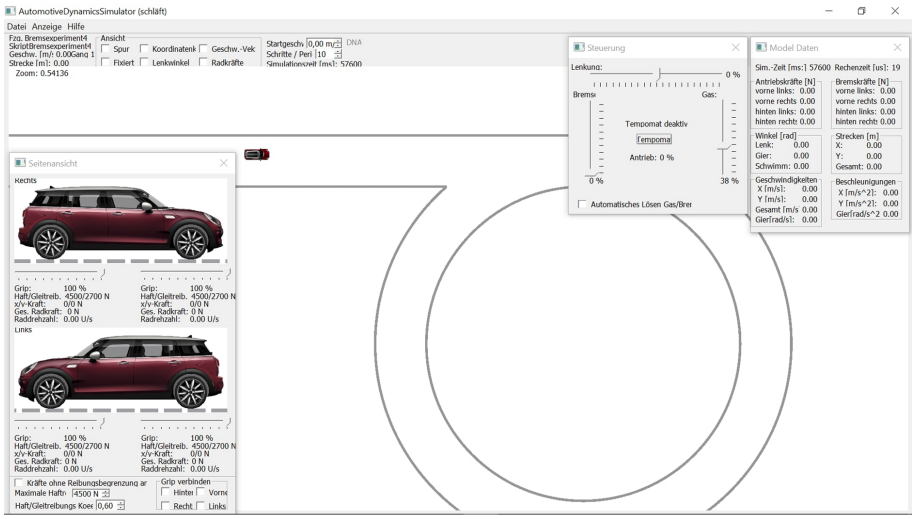

Abbildung 16: Testreihe im Automotive Dynamics Simulator.

Tests und Ergebnisse für Redundanzansätze nach ISO26262

Die Versuche wurden unter einer Linux-Umgebung mit dem von TTTech bereitgestellten RazorMotion-Entwicklungsboard[3] durchgeführt [10, 11]. Dieses Board enthält mehrere Renesas-Quadcore-Prozessoren mit unterschiedlichen Automotive Safety Integrity Levels (ASIL), darunter zwei ASIL-B-CPUs sowie eine ASIL-D-CPU, welche den höchsten Sicherheitsstandard darstellt [9]. Für die Testreihen kamen die beiden ASIL-B-CPUs zum Einsatz.

Das Board beherbergt sowohl die Artificial DNA (ADNA) als auch einen ausführbaren ADNA-Prozessor. Ein Windows-basierter Laborrechner fungiert als Client und betreibt einen Fahrzeugsimulator, der Sensordaten empfängt und Fahrzeugdynamiken auswertet. Das Artificial Hormone System (AHS) verarbeitet dabei Sensordaten zu Steuerfunktionen wie Lenkung, Bremsen und Beschleunigung. Die ADNA ist als Softwareschicht direkt mit dem Simulator verbunden und fungiert als virtuelle Steuereinheit. Dadurch wird eine vollständige Interaktion zwischen Sensorblöcken, Aktorblöcken und dem Simulator ermöglicht. Steuerbefehle der Aktorblöcke werden an den Simulator übermittelt und dort in Fahrmanöver umgesetzt. Es wurde der hauseigene Automotive Dynamics Simulator (Abbildung 16 für diese Tests verwendet, da zu diesem Zeitpunkt eine CARLA Implementierung noch nicht vorlag.

Die zentrale Herausforderung bestand in der Integration des AHS und der systemübergreifenden Kommunikation aller beteiligten Komponenten, Bussysteme und Subsysteme. Die ADNA musste spezifisch an die angebundene Sensorik und Aktorik angepasst werden. Für die Testdurchführung wurden bis zu sechs ADNA-Prozessoren initialisiert, welche unabhängig voneinander Aufgaben wie Sensor-, Aktor- und Offsetblöcke ausführen. Die Kommunikation erfolgte über das User Datagram Protocol (UDP) an dedizierte Empfänger. Der Labor-PC und das Entwicklungsboard waren über ein Ethernet-Kabel verbunden, was eine stabile bidirektionale UDP-Kommunikation zwischen Simulator und ADNA-Prozessor ermöglichte.

Die Experimente wurden reproduzierbar mithilfe von Skriptdateien des Automotive Dynamics Simulator durchgeführt. Dabei wurden die drei ADNA-Varianten aus den

[3]Unterstützt durch TTTech Auto Germany GmbH

Abbildungen 12, 13 und 15 getestet. Das virtuelle Fahrzeug beschleunigte zunächst auf 50 km/h und führte anschließend mit einem Lenkeinschlag von 22 % des maximalen Lenkwinkels eine konstante Rechtskurve aus.

Während der Tests wurden unter anderem Zeit, Trajektorien (Fahrzeugposition, Geschwindigkeit, Lenkwinkel, Driftwinkel, Gierwinkel) und Rechenzeit aufgezeichnet. Die Ergebnisse zur Trajektorie für alle Redundanzstufen ist in Abbildung 17 zur Rechenzeit sind in Abbildung 18 visualisiert.

Die initiale Rechenzeit lag je nach Prozessorverteilung zwischen 500ms und 1000ms. Mit Beginn der Kurvenfahrt bei etwa 7000ms stieg die Rechenzeit deutlich an. Bei 17500ms wurde eine Optimierung der Taskverteilung in der Variante „ADNA 3 x Simple Automotive“ vorgenommen. Unter Normalbedingungen zeigten alle Varianten ein gleichwertiges Fahrverhalten.

1.2.3 Entwicklung und Integration von ADNA-Voting-Komponenten

Für fehlertolerante eingebettete Systeme sind modulare und flexible Architekturen essenziell. Die Artificial DNA (ADNA) stellt ein Framework dar, das die Entwicklung und Integration solcher modularen Komponenten – sogenannter *Building Blocks* (BBs) – unterstützt. In diesem Kapitel wird die Erweiterung der ADNA um drei neue Voting-Komponenten beschrieben, die unterschiedliche Anforderungen an Verfügbarkeit, Fehlertoleranz und Echtzeitanforderungen abdecken.

Architektur und Integration

Neue Komponenten werden in ADNA integriert, indem ihre strukturellen Attribute definiert, ihre Berechnungslogiken implementiert und sie in das Framework registriert werden. Der allgemeine Prozess ist in Tabelle 2 zusammengefasst.

Modul	**Beschreibung**	**Zentrale Funktionen/Anforderungen**
Prozessor-Klassen	Definition verschiedener Prozessor-Klassen mit Hormonwerten	Neue Komponenten werden den passenden Klassen zugeordnet
DNA-Klassen-Definitionen	Spezifikation der DNA-Klassen mit ID, Namen, Nachrichtenlängen, Parametern und Funktionen	Erweiterung der Systemkompatibilität durch Ergänzung der Klassenstruktur
Funktionsdefinitionen für DNA-Klassen	Implementierung der Funktionslogik und Hilfsfunktionen	Registrierung der Komponenten über die Index-Arrays

Tabelle 2: Überblick über Entwicklungsbausteine zur Integration neuer ADNA-Komponenten

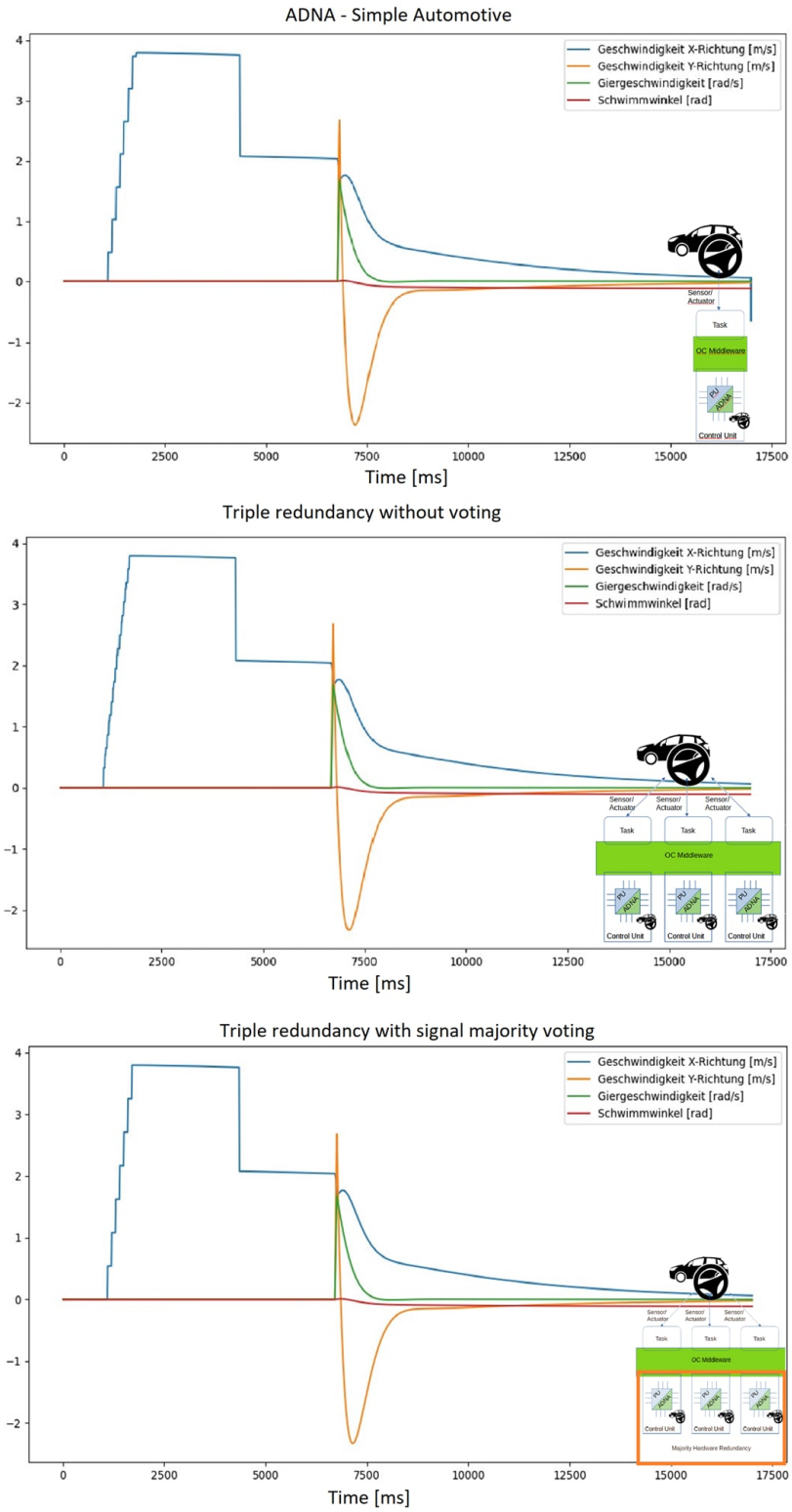

Abbildung 17: Vergleich der Trajektorien für die drei ADNA-Varianten: (oben) „ADNA Simple Automotive“, (mitte) „ADNA 3 x Simple Automotive“ und (unten) „ADNA 3 Red Majority Voting“.

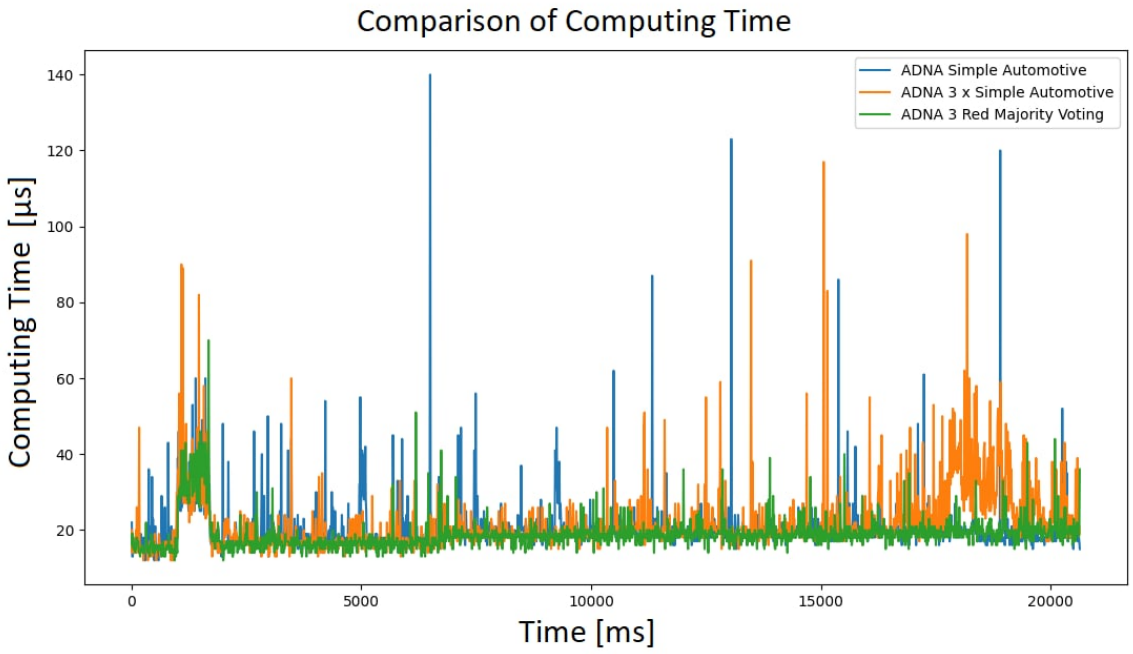

Abbildung 18: Vergleich der Rechenzeit für die drei ADNA-Varianten: „ADNA Simple Automotive“, „ADNA 3 x Simple Automotive“ und „ADNA 3 Red Majority Voting“.

Voting-Mechanismen in ADNA

Zur Erhöhung der Ausfallsicherheit wurden drei Voting-Komponenten integriert:

- **Majority Voter** – Mehrheitsentscheidung zur Fehlererkennung
- **Median Voter** – Medianwert zur Ausreißerresistenz
- **Smoothing Voter** – Glättung zur Signalstabilisierung

Die Voter wurden anhand von fünf Klassifikationskriterien gemäß [**?**] bewertet, siehe Tabelle 3.

Testergebnisse der Voting-Komponenten

Die drei Voting-Komponenten wurden auf Sensor-Aktor-Ebene in vier verschiedenen Szenarien getestet. Um sowohl gerade als auch ungerade Redundanzfälle abzudecken, erfolgten die Tests mit drei und vier Sensoren. Die Voting-Parameter $\epsilon = 0{,}5$ und $\beta = 1$ wurden aus der Literatur übernommen [**?**, **?**]. Die ADNA-Strukturen wurden für jede Konfiguration angepasst.

Neuer Datensatz pro Zyklus: Alle Voter arbeiten im vierfach redundanten System stabiler. Der `Majority Voter` ist ideal zur Fehlererkennung geeignet, versagt jedoch bei mehreren fehlerhaften Eingaben im Dreifach-System. Der `Median Voter` zeigt in beiden Konfigurationen hohe Verfügbarkeit, da er stets einen Wert ausgibt. Der `Smoothing Voter` ist im Dreifach-System am unzuverlässigsten, erreicht jedoch im Vierfach-System die höchste Verfügbarkeit.

Kriterium	Majority Voter	Median Voter	Smoothing Voter
Funktion	Häufigster Wert wird gewählt	Medianwert wird ausgegeben	Ausreißer werden geglättet
Übereinstimmung	Exakt/ unkonfiguriert (Schwelle)	Exakt/ unkonfiguriert (abhängig von Sensoranzahl)	Inexakt mit Akzeptanzbereich
Robustheit	Gut bei großen Eingangsmengen	Hohe Ausreißerresistenz	Glättung schwankender Signale
Implementierung	Modular, rein softwarebasiert	Robust, für Echtzeitsysteme geeignet	Erweiterung des Majority Voters mit Filterung
Anwendungsbereich	Fehlertoleranz, Fehlererkennung	Systeme mit hoher Verfügbarkeit	Regelung in Echtzeitumgebungen

Tabelle 3: Vergleich von ADNA-Voting-Mechanismen

Veraltete Sensorwerte: Ergebnisse sind mit dem ersten Test vergleichbar. Der `Smoothing Voter` kann nicht als Kompromisslösung überzeugen.

Störwerte und manuelle Fehler: Alle Voter reagieren erwartungsgemäß. Der `Median Voter` ist besonders robust gegenüber stark divergierenden Sensorwerten. Der `Majority Voter` bricht bei zu großen Abweichungen erwartungsgemäß ab. Der `Smoothing Voter` liefert vereinzelt noch ein Ergebnis, auch bei extremen Werten.

Fehlende oder fehlerhafte Parameter: Standardwerte sorgen dafür, dass die Voter auch ohne gültige Parametereingaben lauffähig bleiben.

Die modulare Struktur der ADNA erlaubt eine fehlerrobuste Integration. Zeitverzögerungen aufgrund asynchroner Sensordaten konnten durch interne Voter-Strategien kompensiert werden. Der `Majority Voter` eignet sich besonders für sicherheitskritische Anwendungen, während der `Median Voter` eine hohe Verfügbarkeit sicherstellt. Der `Smoothing Voter` erreicht zwar in bestimmten Szenarien gute Ergebnisse, kann jedoch nicht als universeller Kompromiss gelten.

Zufallsbasierte Tests bestätigen die Funktionsfähigkeit aller Voter unter variablen Eingabebedingungen, einschließlich großer Wertunterschiede und negativer Sensorwerte.

Fazit

Die Integration der Voting-Komponenten stärkt die Fähigkeit der ADNA zur Unterstützung fehlertoleranter, sicherer und adaptiver Systeme. Durch die modulare Architektur sind individuelle Anforderungen leicht umsetzbar – ein entscheidender Schritt in Richtung robuster, softwarebasierter Steuerung in sicherheitskritischen Echtzeitsystemen.

Die Integration klassischer Dreifach-Redundanz in eine ADNA-basierte OC-Architektur für automobile Steuergeräte ist gelungen. Ziel war es, bestehende Forschungslücken aufzuzeigen und erste Lösungsansätze zur Etablierung von Organic Computing (OC) in sicherheitskritischen automobilen Anwendungen zu liefern.

Die Tests auf der realen Hardwareplattform zeigten, dass alle drei ADNA-Varianten ein identisches Fahrzeugverhalten aufwiesen – unabhängig von der zugrundeliegenden Redundanzarchitektur. Der nächste Entwicklungsschritt besteht darin, die Eigenschaften eines selbstheilenden Systems systematisch zu untersuchen und diese mit klassischer Redundanz hinsichtlich Robustheit und Performanz zu vergleichen. Geplante Tests sollen das Verhalten bei gezielten Komponentenausfällen anhand automatisierter Skripte analysieren.

Vergleichende Untersuchungen mit gezielt ausgelösten Fehlfunktionen stehen noch aus und sind Bestandteil der weiteren Arbeiten. Dabei gilt es insbesondere die Frage zu klären, inwiefern autonome Selbstheilung als gleichwertige Redundanzform akzeptiert werden kann. Eine positive Bewertung würde neue Möglichkeiten zur Reduktion der Steuergeräteanzahl eröffnen, ohne auf funktionale Redundanz verzichten zu müssen. Durch softwareseitige ADNA-Lösungen und gezielte AHS-Kommunikation – etwa über OC-Netzwerkprozessoren oder spezialisierte Multi-/Demultiplexer – könnten unterschiedliche Redundanzlevel flexibel abgebildet werden. Auch die Einführung eines Voting-Blocks innerhalb der ADNA bietet Potenzial zur Vereinfachung der Fahrzeugarchitektur.

Zusammenfassend wurde ein funktionsfähiger Einstieg in einen ADNA-basierten OC-Ansatz geschaffen, der in der Lage ist, flexibel gewünschte Redundanzgrade bereitzustellen, Systemressourcen effizient zu nutzen und die Ausfallsicherheit zu erhöhen. Diese Lösung verspricht sowohl wirtschaftliche als auch ökologische Vorteile durch reduzierten Hardwareeinsatz und stellt einen wichtigen Schritt in Richtung einer selbstorganisierten, softwarezentrierten Fahrzeugarchitektur dar.

1.2.4 Proaktive Selbstheilung: Systemüberwachung

Problembeschreibung

Die Entwicklung proaktiver Prozessorsteuerungen stellt eine Herausforderung dar, da geeignete Systemparameter zunächst identifiziert, kontinuierlich überwacht und darauf basierend adäquate Reaktionsmechanismen implementiert werden müssen. Insbesondere Parameter wie Temperatur, Leistungsaufnahme sowie gegebenenfalls Taktfrequenz sind hierbei von zentraler Bedeutung, da sie die Stabilität und Leistungsfähigkeit des Systems maßgeblich beeinflussen.

Ziel ist die Konzeption adaptiver Regelmechanismen, die bei Abweichungen von definierten Betriebsgrenzen automatisiert Gegenmaßnahmen, beispielsweise durch eine Reduktion der Rechenlast, initiieren. Dabei wird angestrebt, auf eine zentrale Steuerung zu verzichten und stattdessen eine dezentrale Integration in Middleware-Architekturen wie das Adaptive Hybrid System (AHS) zu ermöglichen, um Skalierbarkeit und Fehlertoleranz sicherzustellen.

Zielsetzung

Im Rahmen dieser Arbeit wurden neue Regelstrategien zur proaktiven Prozessorkontrolle implementiert und deren Auswirkungen in einer Simulationsumgebung analysiert. Folgende Teilziele wurden dabei definiert:

- Analyse der Betriebsparameter: Identifikation relevanter Größen, beispielsweise Temperatur und Leistungsaufnahme, sowie deren Verarbeitung im Systemkontext.
- Entwicklung von Regelmechanismen: Entwurf und Implementierung adaptiver Strategien zur Überwachung und Steuerung des Systemverhaltens.
- Bewertung mittels geeigneter Metriken: Untersuchung von Reaktionszeit, Systemstabilität und Ressourcenmanagement zur Beurteilung der Effektivität.
- Simulation und Vergleichbarkeit: Entwicklung eines Simulationsmodells und Bewertung der Ergebnisse hinsichtlich ihrer Realitätsnähe.

Forschungsfrage

Die zentrale Forschungsfrage lautet:

Welche proaktiven Mechanismen können in DNA-Prozessoren implementiert werden, um durch geeignete Betriebsparameter eine zuverlässige und effiziente Verarbeitung im Grenzbereich zu gewährleisten?

Zur Beantwortung dieser Frage werden unter anderem folgende Aspekte betrachtet:

- Effiziente Nutzung relevanter Eingangsdaten.
- Einfluss interner Parameter auf das Systemverhalten und die Leistung.
- Definition und Anwendung geeigneter Bewertungsmetriken.
- Übertragbarkeit der Simulationsergebnisse auf reale Systeme.

Abgrenzung

Der Fokus dieser Arbeit liegt auf der betriebsparameterbasierten Optimierung innerhalb des Adaptive Hybrid Systems (AHS). Aspekte wie genetische Algorithmen, neuronale Netzwerke, hardwareseitige Realisierungen oder sicherheitsrelevante Zertifizierungen werden nicht berücksichtigt.

Die Umsetzung erfolgt ausschließlich auf softwareseitiger Ebene innerhalb einer bestehenden Simulationsumgebung. Eine praktische Umsetzung in realen Systemen wird konzeptionell diskutiert, jedoch nicht realisiert.

Optimierung der Temperaturregelung

Zur Optimierung der Temperaturregelung wurde ein konfigurierbarer Berechnungsmechanismus für die Suppressoren implementiert. Über einen im User Interface steuerbaren Switch können verschiedene Berechnungsmodi gewählt werden, die unterschiedliche mathematische Ansätze zur Bestimmung der Suppressorenzahl verwenden.

Berechnungsansätze

- **Polynomiale Methode:** Die Suppressorenzahl wird durch eine quadratische Funktion

$$f(t) = a(t - 60) - b(t - 60)^2, \quad t \in [60, 80] \tag{1}$$

bestimmt, wobei die Parameter $a = 1{,}4$ und $b = 0{,}035$ anhand der Bedingungen $f(60) = 0$, $f(80) = 14$ und $f'(80) = 0$ ermittelt wurden.

- **Exponentialfunktion auf Temperaturänderung:** Basierend auf der durchschnittlichen Temperatursteigung $\bar{m}$ wird die Suppressorenzahl durch

$$f(B) = A \cdot \left(1 - e^{-k \cdot \bar{m}}\right) \cdot C \tag{2}$$

berechnet, wobei $A = 14$ die Maximalanzahl der Suppressoren, $k = 0{,}2$ ein Skalierungsfaktor und $C = 0{,}5$ ein Gewichtungsfaktor darstellen.

- **Erweiterte Exponentialfunktion mit Temperaturdifferenz:** Zusätzlich zur Steigung wird die Differenz zur Schwellentemperatur $T_{\text{diff}} = T - 60$ berücksichtigt:

$$f(B, T_{\text{diff}}) = A \cdot \left(1 - e^{-k \cdot \bar{m} \cdot T_{\text{diff}}}\right) \tag{3}$$

Diese dynamischere Funktion ermöglicht eine stärkere Anpassung an die absolute Temperatur und deren Veränderung.

Verifikation

Die Implementierung wurde in einer Simulation unter variierenden Kühlbedingungen (0 %, 10 %, 20 %, 30 % Kühlleistung) getestet. Dabei wurde die Temperaturentwicklung im Bereich von 60°C bis 80°C sowie die Anzahl aktiver Suppressoren und verbleibender Tasks analysiert. Die Ergebnisse zeigen, dass die exponentielle Methode mit Temperaturdifferenz eine dynamischere Steuerung ermöglicht, während die polynomiale und einfache exponentielle Methode vergleichbare, aber konservativere Verläufe aufweisen.

Erweiterung: Regelung der Leistungsaufnahme

Ergänzend zur Temperaturregelung wurde ein Modell zur Simulation und Begrenzung der Leistungsaufnahme entwickelt. Die Leistungsaufnahme orientiert sich an der aktuellen Aufgabenlast und wird analog zur Temperatur in Intervallen berechnet. Die Simulation liegt mit maximal 33,6 W unterhalb realer Prozessorwerte (max. ca. 60 W), was die Plausibilität der Modellierung unterstreicht.

Zur Steuerung der Leistungsaufnahme wurde eine Schwellenwertregel implementiert, die die Anzahl der Suppressoren abhängig von der aktuellen Leistungsaufnahme anpasst:

- **Überlast:** Exponentielle Erhöhung der Suppressorenanzahl bei Überschreitung der Maximalleistung $P_{\max}$:

$$m_{\text{powerSuppressor}} = \text{maxSuppressor} \cdot \left(1 - e^{-k_2 \frac{P - P_{\max}}{P_{\max}}}\right) \tag{4}$$

- **Unterlast:** Lineare Reduktion der Suppressoren bei Unterschreitung:

$$m_{\text{powerSuppressor}} = m_{\text{powerSuppressor}} - k_1(P_{\max} - P) \tag{5}$$

- **Toleranzbereich:** Keine Änderung innerhalb eines definierten Grenzbereichs, um Systemstabilität zu gewährleisten.

Diese Regelung erlaubt eine flexible und stabile Anpassung der Systemauslastung und unterstützt so die Gesamtsystemstabilität.

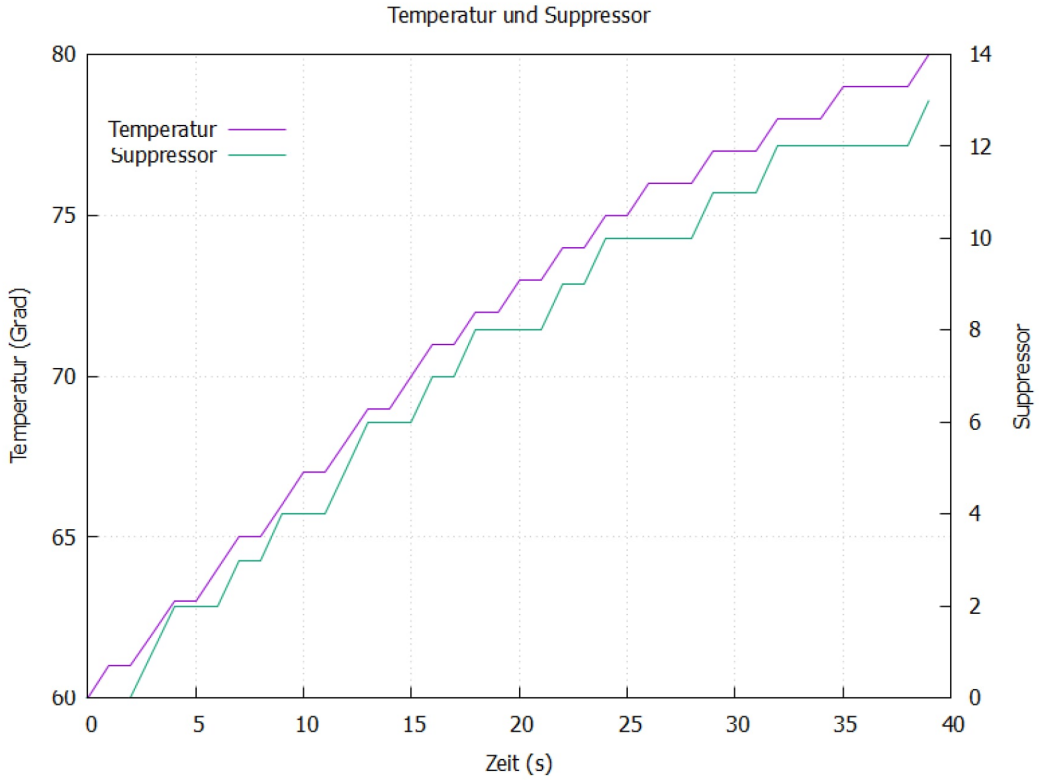

Abbildung 19: Temperaturverlauf bei 0 % Kühlleistung für die lineare Methode

Ergebnisse der Tests zur Leistungsregelung

Zur Bewertung der Methode zur Regelung der maximalen Leistungsaufnahme wurden zwei Testreihen durchgeführt, die sich ausschließlich im verwendeten Wachstumsfaktor k_2 der Exponentialfunktion unterscheiden: Einmal wurde $k_2 = 2$, einmal $k_2 = 4$ gewählt. Als maximal zulässige Leistungsaufnahme dienten jeweils 16 W, 20 W, 24 W und 28 W.

Da die Eingabe der Maximalleistung manuell erfolgte, stimmen die Reaktionszeitpunkte der Testreihen nicht exakt überein. Daher wurde als Vergleichskriterium der Zeitpunkt der ersten Änderung des Suppressorwertes herangezogen.

Die Ergebnisse zeigen signifikante Unterschiede im Regelverhalten: Mit $k_2 = 4$ schlägt die exponentielle Methode stark aus und erreicht schnell die maximale Anzahl von 14 Suppressoren (vgl. Abb. 21). Im Gegensatz dazu reagiert die Variante mit $k_2 = 2$ deutlich moderater und weist weniger extreme Ausschläge auf (vgl. Abb. 22).

Besonders bemerkenswert ist das Verhalten bei einer Maximalleistung von 24 W: Beide Varianten zeigen Schwingungen des Suppressorwertes, ohne eine stabile Endlage zu erreichen (vgl. Abb. 23, Abb. 24). Für $k_2 = 4$ ist das Regelverhalten instabil, da keine zufriedenstellende Regelposition erreicht wird.

Abgesehen von diesem Fall verbleiben alle anderen Konfigurationen im definierten Toleranzbereich stabil. Die Variante mit $k_2 = 2$ zeichnet sich durch ein insgesamt robusteres und gleichmäßigeres Regelverhalten aus. Einzig bei 28 W treten bei beiden Varianten unerwartet starke Ausschläge auf, die von der Berechnungsformel nicht prognostiziert werden.

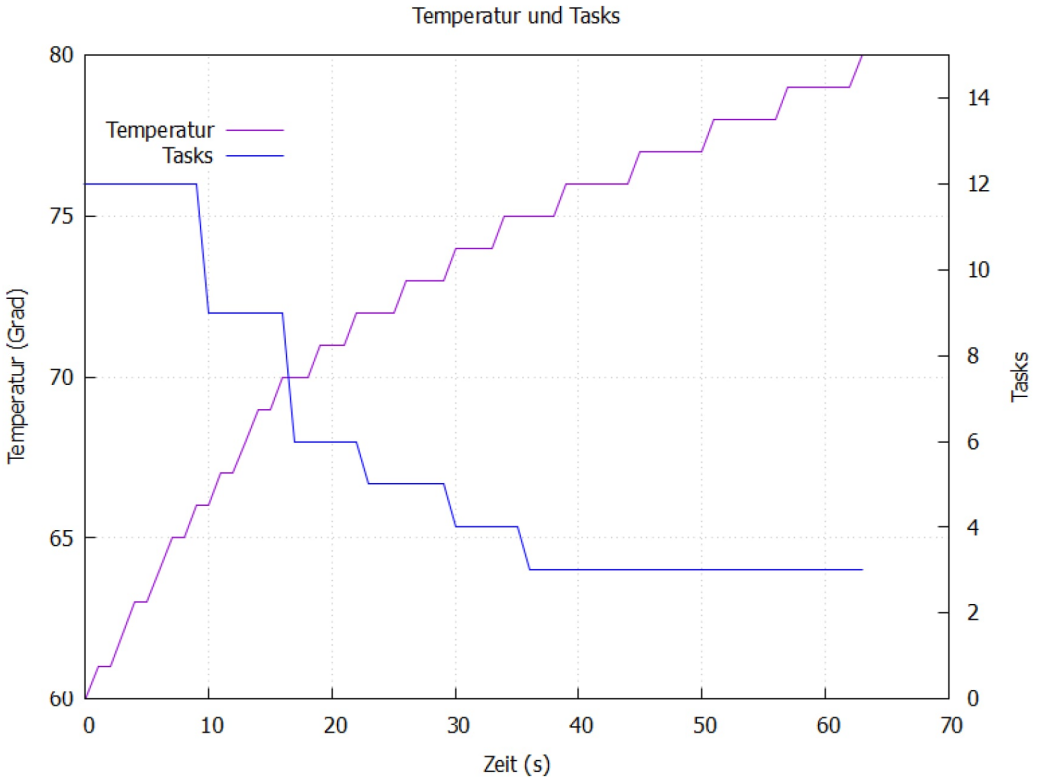

Abbildung 20: Anzahl aktiver Tasks bei 0 % Kühlleistung für die polynomiale Methode

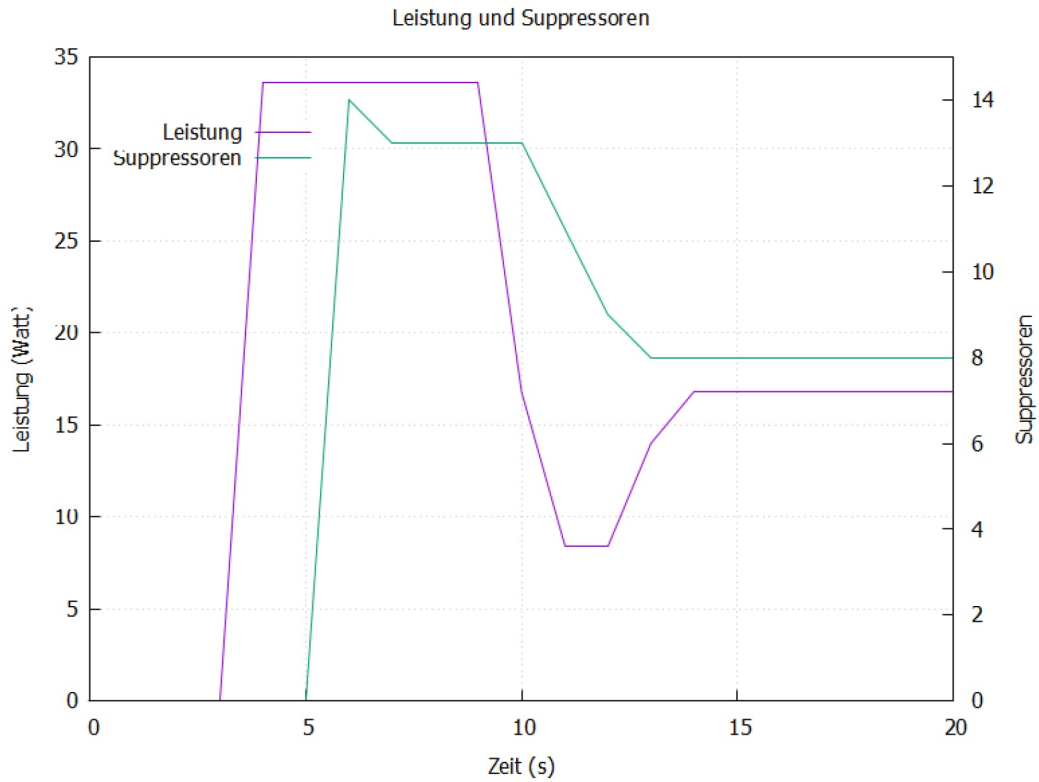

Abbildung 21: Verlauf der Suppressorenanzahl bei maximal 16 W mit Wachstumsfaktor 4.

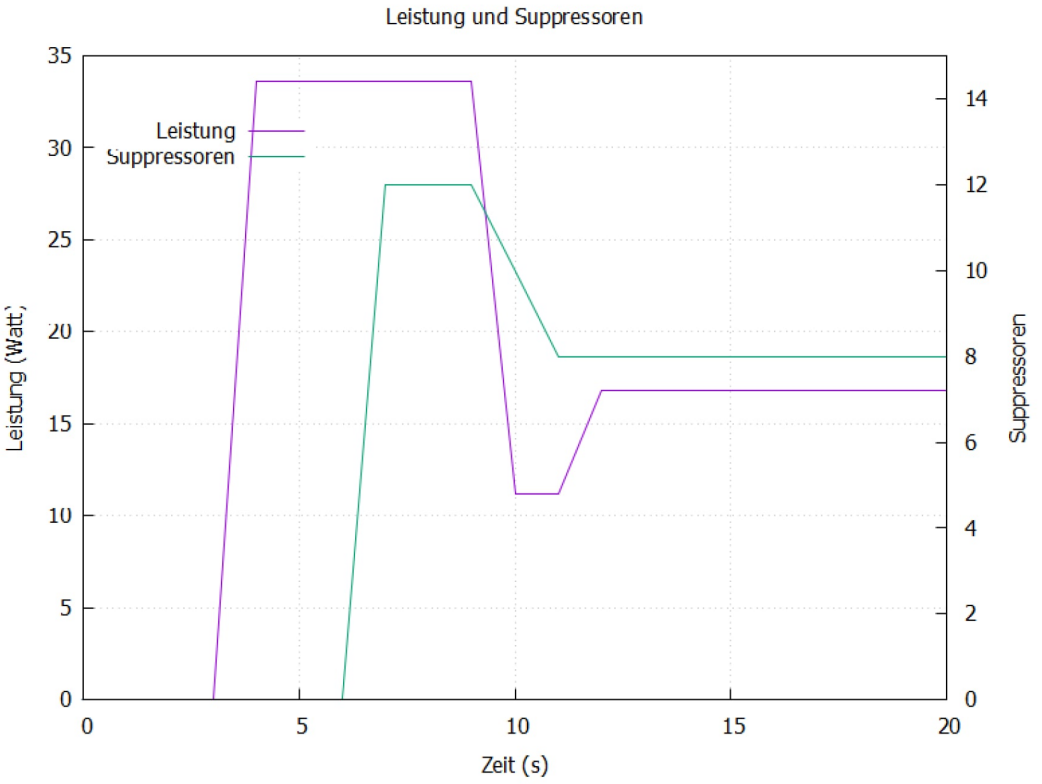

Abbildung 22: Verlauf der Suppressorenanzahl bei maximal 16 W mit Wachstumsfaktor 2.

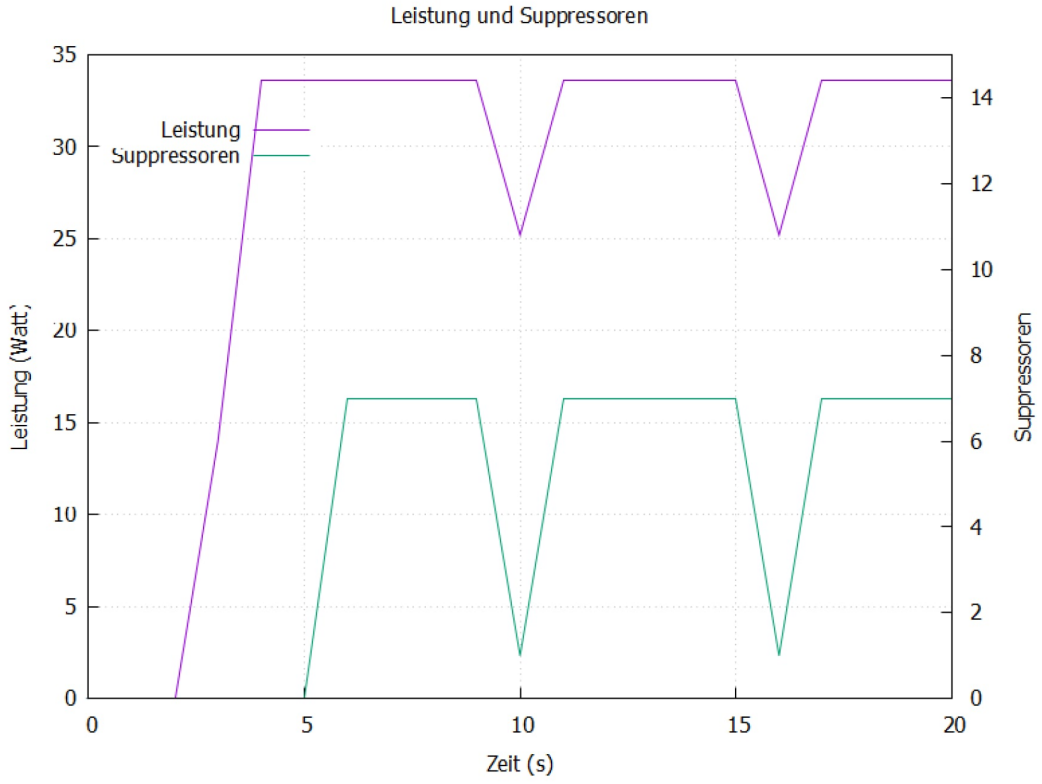

Abbildung 23: Schwingungsverhalten bei maximal 24 W mit Wachstumsfaktor 2.

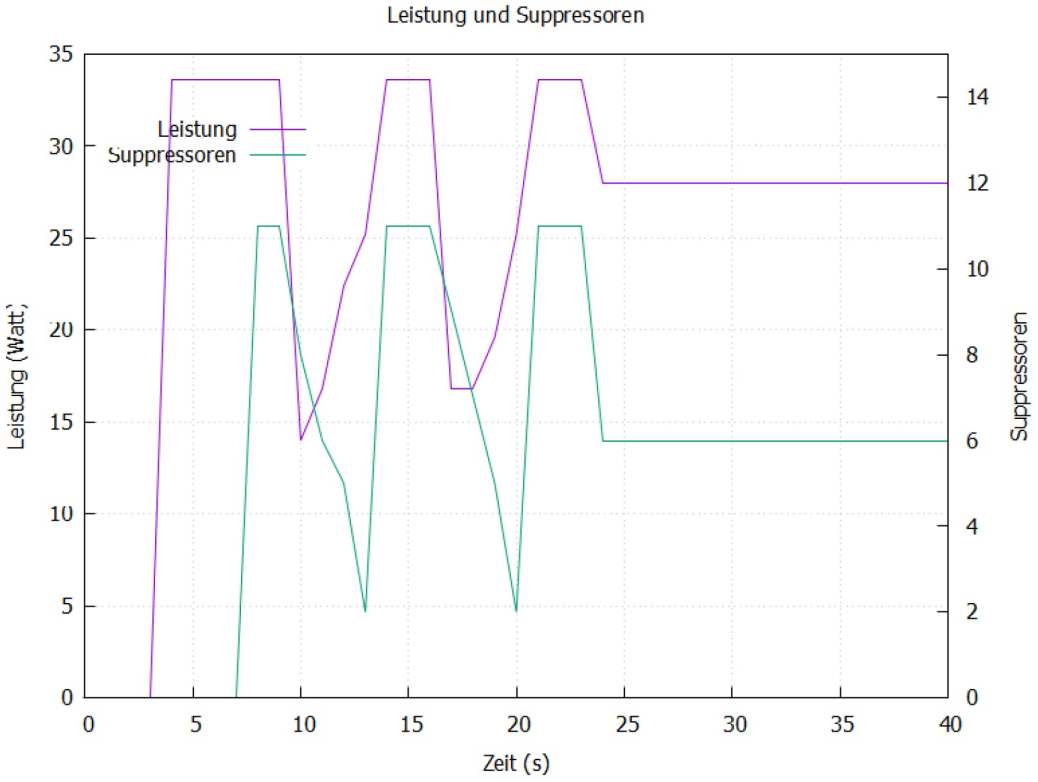

Abbildung 24: Instabiles Schwingungsverhalten bei maximal 24 W mit Wachstumsfaktor 4.)

Implementierung klassischer Regler: P- und PI-Regler

Zur Überprüfung der Wirksamkeit und Stabilität der entwickelten Regelansätze wurden zwei klassische Regler implementiert: ein Proportionalregler (P-Regler) und ein Proportional-Integral-Regler (PI-Regler). Diese dienen als Referenz für die exponentielle und lineare Suppressorenberechnung und ermöglichen eine vergleichende Analyse des Regelverhaltens bei abweichender Leistungsaufnahme.

Grundprinzip: Der P-Regler reagiert proportional auf den aktuellen Fehler, definiert als Differenz zwischen der tatsächlichen Leistungsaufnahme $m_{\text{processingElementPower}}$ und der maximal erlaubten Leistung m_{maxPower}. Der PI-Regler ergänzt diesen Ansatz durch einen Integralanteil, der die Summe vergangener Fehler berücksichtigt und so eine präzisere Sollwertverfolgung erlaubt.

P-Regler Die Implementierung des P-Reglers erfolgt über den Verstärkungsfaktor K_p. Überschreitet die Leistungsaufnahme die maximal zulässige Grenze und ist keine Temperaturregelung aktiv, so wird der Suppressorwert proportional zum Fehler angepasst:

$$m_{\text{powerSuppressor}} = K_p \cdot (m_{\text{processingElementPower}} - m_{\text{maxPower}}) \tag{6}$$

Der Wert wird anschließend auf den zulässigen Bereich $[0, 14]$ begrenzt.

Analyse der Regelverläufe von P- und PI-Regler

Das Regelverhalten der beiden klassischen Regler wurde bei Maximalleistungen von 16 W, 20 W, 24 W und 28 W untersucht.

P-Regler: Charakteristisch sind deutliche und langanhaltende Schwingungen um den Sollwert, insbesondere bei 16 W . Für 20 W und 24 W stabilisiert sich der Regelwert, bleibt jedoch signifikant oberhalb des Sollwertes. Bei 28 W reduziert sich die Abweichung, bleibt aber bestehen).

PI-Regler: Insgesamt zeigt der PI-Regler ein präziseres Verhalten. Nach anfänglichen Schwingungen erreicht der Sollwert bei 16 W eine stabile Lage. Bei 20 W treten kurze Überschwinger auf, das System pendelt sich jedoch schnell ein. Bei 24 W bleibt die Regelung größtenteils stabil, zeigt gegen Ende leichte Schwingungen. Für 28 W sind längere Schwingungen zu beobachten, jedoch bleibt der Sollwert innerhalb eines Toleranzbereiches stabil.

Diese Ergebnisse bestätigen die typischen Eigenschaften: Der P-Regler reagiert schnell, aber mit bleibenden Abweichungen, während der PI-Regler durch den Integrationsanteil Fehler langfristig kompensiert und präzisere Regelwerte erzielt.

Vergleich der entwickelten Regelung mit klassischen Reglern

Die entwickelte Methode kombiniert eine exponentielle Zunahme der Suppressorenanzahl bei Überlast mit einem linearen Abbau bei Unterlast sowie einem definierten Toleranzbereich ohne Reaktion. Dadurch entsteht ein nichtlinearer, dennoch einfacher Regelmechanismus, der gezielt auf Schwellenwerte reagiert und instabile Schwingungen bei kleinen Schwankungen vermeidet.

Im Vergleich zeigt der P-Regler schnelle Reaktionen mit bleibender Regelabweichung und anhaltenden Schwingungen bei niedrigen Grenzwerten. Der PI-Regler erzielt stabilere Endwerte, weist aber bei höheren Grenzwerten gelegentlich Überschwinger auf.

Die eigens entwickelte Methode ist robuster gegenüber kleinen Schwankungen und vermeidet übermäßige Reaktionen durch den integrierten Toleranzbereich. Der exponentielle Anstieg ermöglicht schnelles Eingreifen bei starker Überlast, der lineare Abbau sorgt für sanfte Rückführung bei Entlastung. Insgesamt bietet der Ansatz eine ausgewogene Balance zwischen Einfachheit, Stabilität und praktischer Reaktionsfähigkeit und eignet sich besonders für Systeme mit diskontinuierlicher, schwellenwertbasierter Regelung.

Die klassische Reglerimplementierung erweitert die Bewertung und bestätigt, dass abgestufte, nichtlineare Regelstrategien unter Berücksichtigung von Schwellenwerten eine sinnvolle Alternative zu kontinuierlichen Reglern darstellen.

Temperatursimulation

Die Simulation der Temperatur erfolgt mit einer Basis- oder Umgebungstemperatur von 20°C (definiert als `BASE_TEMPERATURE`). Jede Sekunde wird ein neuer Temperaturwert berechnet, basierend auf der aktuellen Wärmeentwicklung und Kühlleistung.

Die Temperaturänderung ΔT wird als Differenz zwischen der durch die aktuelle Aufgabenlast (`m_taskTemperatureLoad`) erzeugten Wärme und der vom Kühlkörper abgeführten Wärme berechnet. Zur Bestimmung der abgeführten Wärme wird zunächst die Differenz zwischen der aktuellen Prozessortemperatur (`m_processingElementTemperature`) und der Basistemperatur berechnet. Diese Differenz wird mit der Kühlkörpereffizienz (`m_heatSinkEfficiency`) multipliziert und anschließend durch 10 geteilt. Die resultierende Temperaturänderung ΔT wird durch den Temperaturgradienten (`TEMPERATURE_GRADIENT`) geteilt, wobei dieser einen festen Wert von 500 hat.

Die berechnete Temperaturänderung ΔT wird anschließend zur aktuellen Prozessortemperatur addiert (vgl. Listing 1).

```
int DNAProcessorWindowsTemplate::CalculateTemperature()
{
    float deltaT;

    // Berechnung der Temperaturdifferenz
    deltaT = (float)m_taskTemperatureLoad -
             (((m_processingElementTemperature - BASE_TEMPERATURE) *
             (float)m_heatSinkEfficiency) / 10);
    deltaT /= TEMPERATURE_GRADIENT;

    // Hinzuf"ugen zur aktuellen Temperatur
    m_processingElementTemperature += deltaT;

    return (int)m_processingElementTemperature;
}
```

Listing 1: Simulation Temperatur

Die Aufgabenlast `m_taskTemperatureLoad` wird aktualisiert, sobald neue Aufgaben hinzugefügt oder bestehende entfernt werden.

Bestimmung der Suppressoren

Ab einer Temperatur von 60 °C werden Suppressoren in Abhängigkeit von der aktuellen Temperatur linear ausgeschüttet. Die Berechnung des Suppressor-Werts basiert auf der Differenz zwischen der aktuellen Temperatur (`temp`) und der festgelegten Referenztemperatur (`GOOD_TEMPERATURE`).

Diese Temperaturdifferenz wird mit einem festen Skalierungsfaktor (`TEMPERATURE_MONITORING_SUPPR`) multipliziert, der auf 20 gesetzt ist. Um eine angemessene Skalierung des Suppressor-Werts sicherzustellen, erfolgt eine Normierung durch den Faktor `TEMPERATURE_MONITORING_SUPPRESS`. Dieser ergibt sich aus der Differenz zwischen der maximal tolerierbaren Temperatur (`FAIL_TEMPERATURE` = 90 °C) und der Referenztemperatur (`GOOD_TEMPERATURE` = 60 °C), wodurch ein Wert von 30 resultiert. Abbildung 25 zeigt die Benutzeroberfläche mit den eingefärbten Grenzwerten (grün, gelb, rot).

```
tempSuppressor = ((temp - GOOD_TEMPERATURE) *
                  TEMPERATURE_MONITORING_SUPPRESSOR_NOM) /
                  TEMPERATURE_MONITORING_SUPPRESSOR_DENOM;
```

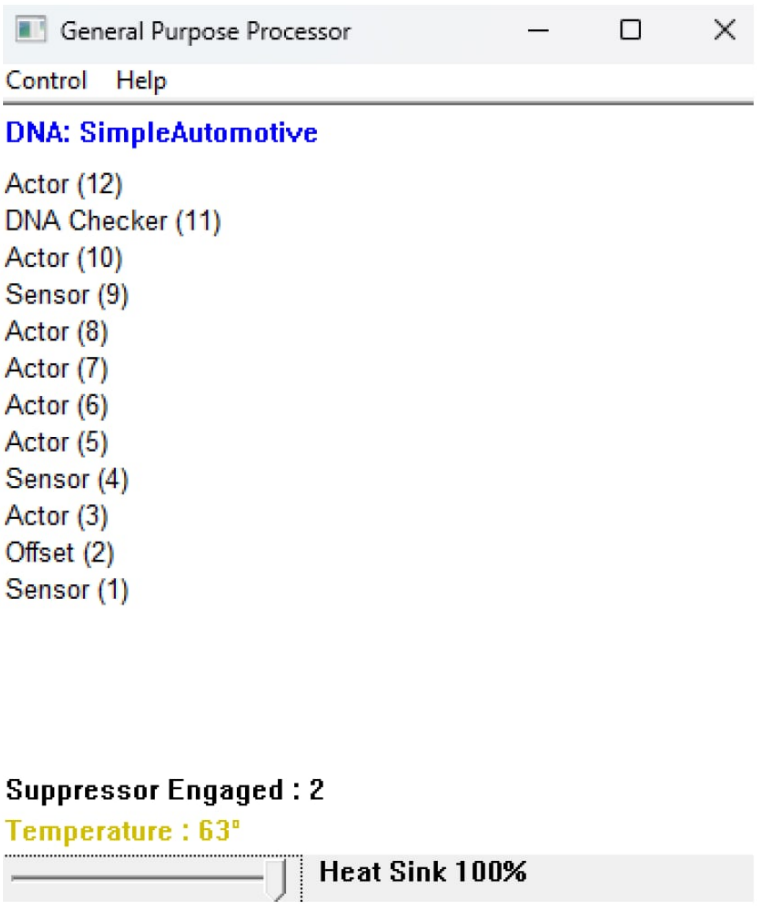

Abbildung 25: Die Benutzeroberfläche verfügt über eine Suppressoranzeige in Ampelfarben.

Listing 2: Berechnung Suppressoren

Durch diese Berechnung wird sichergestellt, dass die Suppressor-Werte in einem definierten Bereich steigen und die Temperatur kontrolliert bleibt. Der Suppressor-Wert erhöht sich proportional zur Temperaturdifferenz und trägt dazu bei, ein Überschreiten kritischer Temperaturgrenzen zu vermeiden.

Temperaturquellen (Proof of Concept)

In diesem Abschnitt wurde neben der simulierten Temperaturentwicklung auch untersucht, ob externe Temperaturquellen zur Steuerung der Suppressorenregelung eingebunden werden können. Ziel war es, alternative Datenpfade als Proof of Concept zu implementieren und damit die Flexibilität des Systems zu demonstrieren.

Zwei exemplarische Ansätze wurden dabei realisiert:

- **TXT-Datei-Einbindung:** In einer ersten Variante wird eine vorbereitete `.txt`-Datei ausgelesen, die eine Reihe von Temperaturwerten enthält. Diese Werte sind kommasepariert in einer Zeile gespeichert und werden der Reihe nach eingelesen und verarbeitet. Diese Methode eignet sich insbesondere für Testszenarien mit festgelegten Temperaturverläufen und ermöglicht reproduzierbare Abläufe ohne Einfluss externer Faktoren.
- **JSON-basierte Echtzeittemperatur:** In einer zweiten Variante wird die aktuelle CPU-Temperatur über das Open-Source-Tool *Open Hardware Monitor* erfasst und in Form einer `.json`-Datei bereitgestellt. Dabei agiert Open Hardware Monitor als lokaler Webserver, der die Sensorinformationen über eine HTTP-Schnittstelle bereitstellt. Die JSON-Datei kann dann über einen konfigurierbaren Port (standardmäßig Port 8085) vom Programm abgefragt und eingelesen werden. Der Port muss dabei explizit im Quellcode berücksichtigt und angepasst werden, da die Anbindung direkt über einen HTTP-Request erfolgt.

Aufgrund der individuellen Pfad- und Portkonfiguration sowie der lokal installierten Software ist diese Variante aktuell auf den Entwicklungsrechner beschränkt. Eine Übertragung auf andere Systeme wäre prinzipiell möglich, erfordert jedoch eine entsprechende Anpassung der Systemumgebung.

Abbildung 26 veranschaulicht die Temperaturanzeige im Tool Open Hardware Monitor sowie deren Visualisierung im Proactive Processor. Die Verknüpfung erfolgt derzeit über die Prozessor-ID. Da sich in der Praxis jedoch nicht eindeutig feststellen lässt, welcher physische Core tatsächlich den Prozess bearbeitet, wurde auf eine tiefere Einbindung in die logische Steuerung verzichtet.

Beide Ansätze wurden erfolgreich implementiert und getestet. Aus Gründen der Vergleichbarkeit und Kontrolle wurde jedoch ausschließlich auf die intern simulierte Temperatur zurückgegriffen. Die externen Quellen hätten durch Schwankungen und Hardwareabhängigkeiten zu inkonsistenten Ergebnissen geführt. Dennoch belegen die implementierten Alternativen die Erweiterbarkeit des Systems und zeigen, dass eine Echtzeitkopplung mit physikalischen Sensorwerten grundsätzlich möglich ist.

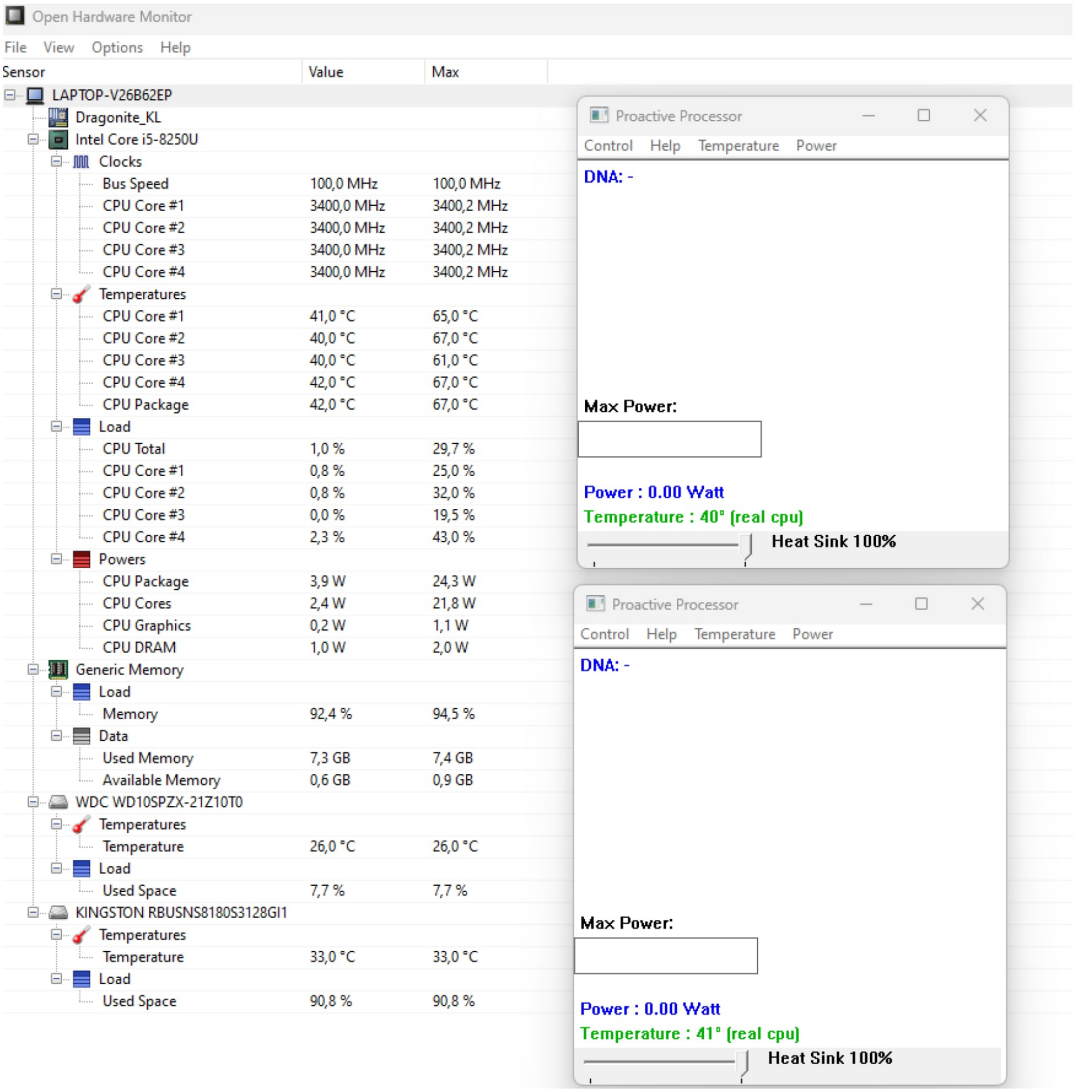

Abbildung 26: Open Hardware Monitor Anbindung.

Fazit

Es wurde die Temperatur und die Leistungsaufnahme als Betriebsparameter eingeführt. Wie beschrieben sind für die Temperatur mehrere Ansetze implementiert. Ein Schwellenwertmechanismus regelt die Anzahl der Suppressoren abhängig von der Differenz zur maximal zulässigen Leistung. Dabei wurde exponentielles Wachstum bei Überlast und linearer Abbau bei Entlastung kombiniert. Vergleichende Tests mit P- und PI-Reglern zeigten, dass die entwickelte Methode hinsichtlich Stabilität und Reaktionsfähigkeit konkurrenzfähig ist und teilweise robuster gegenüber Schwingungen reagiert.

Beantwortung der Forschungsfrage

Die zentrale Forschungsfrage lautete:

> Welche proaktiven Mechanismen lassen sich in DNA-Prozessoren integrieren, um durch geeignete Betriebsparameter eine zuverlässige und effiziente Verarbeitung im Grenzbereich zu gewährleisten?

Antwort:

> Temperatur und Leistungaufnahme des Systems als Simulation für die DNA.Prozessoren. Bedingt über eine Schnittstelle zum Systemüberwachungstool OpenHardwareMonitor für echte Prozessoren.

Diese konnte umfassend beantwortet werden: Es wurden adaptive, simulationsnahe Mechanismen entwickelt, die Temperatur- und Leistungsdaten auswerten und dadurch den Prozessorbetrieb steuern. Die Evaluation zeigte klare Vorteile gegenüber statischen oder klassischen Regelansätzen.

Ausblick

Neben Temperatur und Leistungsaufnahme bieten weitere Parameter Potenzial zur Optimierung:

- **Spannung:** Adaptive Überwachung kann Leistung und Stabilität verbessern, insbesondere in mobilen Systemen.
- **Taktfrequenz:** Dynamische Frequenzanpassung ermöglicht eine aktivere und feinere Steuerung der Rechenleistung.
- **Mehrgrößenregelung:** Ein kombinierter Regelansatz für Temperatur, Leistung, Spannung und Frequenz stellt ein zukunftsweisendes Forschungsfeld dar, insbesondere für Organic Computing und autonome Systeme.

Ein kombinierter Ansatz (`combinedSuppressor`) zur gewichteten Überlagerung von Temperatur- und Leistungsregelung ermöglicht eine feinere Anpassung in komplexen Zuständen und bildet die Grundlage für ein umfassendes Ressourcenmanagement.

Zusammenfassend leisten die entwickelten proaktiven Steuerungsansätze einen wichtigen Beitrag zur autonomen Regelung in selbstorganisierenden Prozessorarchitekturen

und schaffen eine solide Basis für zukünftige Erweiterungen. Eine modulare und priorisierbare Steuerung der Betriebsparameter ist nicht nur praktikabel, sondern auch ein vielversprechender Ansatz für robuste und adaptive Prozessorarchitekturen der Zukunft.

1.3 Formale Verifikation

1.3.1 Bewertung und Erkenntnisse aus der formalen Verifikation

Die formale Verifikation mit UPPAAL ermöglichte eine systematische Analyse der ADNA-Bausteine unter realistischen Annahmen sicherheitskritischer eingebetteter Systeme. Durch die Modellierung der Kontrollflüsse als zeitbehaftete Automaten konnten zentrale Anforderungen wie Deadlock-Freiheit, deterministisches Verhalten und korrekte Reaktion auf ungültige Eingaben überprüft werden.

Ein wesentliches Ergebnis war, dass sich die Modularität der ADNA-Architektur als vorteilhaft für die Verifikation erwiesen hat. Die isolierte Modellierung einzelner Bausteine – etwa der `ALU`, Voting-Komponenten oder der Diagnose-Units – erleichterte sowohl die Fehleranalyse als auch die iterative Verfeinerung der Modelle. Für jede dieser Komponenten konnten spezifische Sicherheitseigenschaften erfolgreich verifiziert werden, unter anderem:

- Zustandskonsistenz und deterministischer Kontrollfluss,
- Deadlock-Freiheit in allen Ausführungsmodi,
- Erreichbarkeit aller gewünschten Betriebszustände bei gültigen Eingaben,
- Robuste Fehlerbehandlung bei fehlerhaften oder fehlenden Eingaben,
- Einhaltung definierter Kontrollstrukturen (z. B. Schleifenbegrenzung).

Zudem zeigten die Ergebnisse, dass bereits einfache formale Modelle eine hohe Aussagekraft hinsichtlich der strukturellen Korrektheit liefern können – selbst ohne konkrete Zeitanteile oder exakte numerische Berechnungen. Dies erlaubt eine frühe Absicherung in der Softwareentwicklung, noch bevor eine vollständige Implementierung vorliegt.

Als Herausforderung stellte sich hingegen die begrenzte Ausdrucksstärke des UPPAAL-Modellieransatzes bei komplexer Interprozess-Kommunikation sowie Gleitkommazahloperationen heraus. Hier mussten geeignete Abstraktionen und Approximationen eingesetzt werden, um dennoch sicherheitsrelevante Aussagen treffen zu können. Besonders bei komponentenübergreifender Kommunikation (z. B. zwischen Diagnose- und Selbstheilungsmodulen) wurde die Bedeutung wohldefinierter Schnittstellen deutlich.

Insgesamt belegt die durchgeführte formale Verifikation die strukturelle Korrektheit der ADNA-Kernkomponenten und deren Fähigkeit zur robusten Fehlerverarbeitung. Die entwickelten Modelle und Methodiken können in zukünftigen Projektphasen erweitert werden, etwa durch:

- Integration zeitlicher Einschränkungen zur Analyse von Echtzeiteigenschaften,
- Erweiterung auf vollständige Systemmodelle mit mehreren parallel agierenden Bausteinen,

- Verifikation konkreter Anwendungsfälle im Fahrzeugkontext (z. B. fail-operational Lenkfunktionen).

Die durchgeführte Arbeit stellt damit eine solide Grundlage für den sicheren Einsatz von Organic-Computing-Architekturen in sicherheitskritischen Automobilanwendungen dar.

1.3.2 Verifikationsergebnisse

Die in Abschnitt 1.3.1 beschriebenen Modellierungsansätze wurden auf die zehn ADNA-Komponenten (Basis Baustein – Building Blocks) angewendet, um deren Kontrollfluss mittels UPPAAL zu verifizieren. Die resultierenden Modelle umfassten jeweils zwischen 6 und 42 Zuständen pro Komponente und enthielten eine Vielzahl an Übergängen mit Bedingungen, Synchronisationen und Verzweigungen. Für jede Komponente wurden spezifische Eigenschaften definiert und geprüft.

Einige exemplarische Verifikationsziele umfassten:

- **Deadlock-Freiheit:** Sicherstellung, dass kein Zustand erreicht werden kann, aus dem keine weiteren Aktionen möglich sind.
- **Gültigkeit von Eingaben:** Überprüfung, ob bei der Verarbeitung von Nachrichten bestimmte Vorbedingungen erfüllt sind, bevor Operationen durchgeführt werden.
- **Fehlersichere Schleifenstruktur:** Verifikation, dass bei unerwarteten oder fehlerhaften Eingaben ein sicherer Rücksprung zum Schleifenkopf erfolgt.
- **Funktionsaufruf-Reihenfolge:** Sicherstellung, dass Funktionsaufrufe korrekt synchronisiert und in der vorgesehenen Reihenfolge ausgeführt werden.
- **Zustandsinvarianten:** Gewährleistung, dass während der Ausführung bestimmte Bedingungen immer erfüllt bleiben (z. B. bestimmte Moduswerte, gesetzte Flags, erlaubte Übergänge).

Die Verifikation wurde für jede Komponente separat durchgeführt. Die Dauer der Zustandsraumanalyse betrug – abhängig von Modellgröße und Komplexität – zwischen wenigen Sekunden und mehreren Minuten. In Fällen, in denen eine Verifikation fehlschlug, konnte durch die von UPPAAL erzeugte Gegenbeispiel-Simulation die Ursache gezielt analysiert und das Modell entsprechend angepasst werden.

Trotz der modellierungsbedingten Einschränkungen (z. B. fehlende Gleitkomma-Unterstützung) konnten alle wesentlichen Kontrollflüsse, Zweigentscheidungen und Sicherheitsüberprüfungen formal verifiziert werden.

1.3.3 Diskussion

Die durchgeführte Modellierung und Verifikation der ADNA-Bausteine mit UPPAAL zeigt, dass der Ansatz einer strukturierten, zustandsbasierten Modellierung auch für eingebettete Komponenten mit komplexem Kontrollfluss praktikabel und zielführend ist. Insbesondere die strikte Trennung von Zuständen und Übergängen sowie die explizite Modellierung von Schleifen, Bedingungen und Funktionsaufrufen ermöglicht eine detaillierte Analyse des Systemverhaltens.

Vorteile des Ansatzes

- **Früherkennung von Fehlern:** Die symbolische Verifikation identifiziert logische Fehler, die bei reiner Simulation oder Tests eventuell unentdeckt bleiben würden.
- **Modularität:** Die Modellierung jeder Funktion als separater Automat fördert die Nachvollziehbarkeit und Wiederverwendbarkeit der Struktur.
- **Exakte Kontrolle:** Durch die Verwendung von Zustandsinvarianten und synchronisierten Übergängen lässt sich die Einhaltung von Regeln präzise überprüfen.

Herausforderungen

- **Abstraktion numerischer Werte:** Aufgrund der fehlenden Gleitkomma-Unterstützung in UPPAAL müssen bestimmte Berechnungen (z. B. im `PID`-Regler) abstrahiert werden. Dies limitiert die Verifikation auf strukturelle Aspekte und Sicherheitsprüfungen, nicht jedoch auf exakte Rechenergebnisse.
- **Komplexität bei tief verschachtelten Kontrollflüssen:** Komponenten mit vielen verschachtelten Bedingungen oder iterativen Schleifen erzeugen große Zustandsräume. Dies kann zu längeren Verifikationszeiten und Speicherproblemen führen.
- **Initialer Modellierungsaufwand:** Die manuelle Umsetzung der Programmstruktur in ein UPPAAL-Modell ist zeitintensiv und erfordert eine gute Kenntnis sowohl der Zielkomponente als auch der Modellierungssprache.

Potenziale für zukünftige Arbeiten

- **Automatisierte Modellgenerierung:** Durch Parsing von ADNA-Sourcecode und Transformation in UPPAAL-kompatible Modelle ließe sich der Modellierungsaufwand erheblich reduzieren.
- **Integration in CI/CD-Prozesse:** Eine automatisierte Verifikation im Rahmen von Continuous Integration würde eine laufende Qualitätssicherung ermöglichen.
- **Erweiterung auf datenbasierte Verifikation:** Ergänzend zur Kontrollflussanalyse könnte durch Integration zusätzlicher symbolischer Auswertungen auch die Gültigkeit von Rechenoperationen überprüft werden – beispielsweise durch Intervallarithmetik oder abstrahierte Zustandsmodelle.

Insgesamt zeigt sich, dass UPPAAL ein effektives Werkzeug zur Verifikation von Kontrollflüssen in eingebetteten Komponenten darstellt, auch wenn es in Bezug auf Datenverarbeitung nur eingeschränkt nutzbar ist. Die entwickelten Modelle liefern einen belastbaren Nachweis über die Korrektheit des Ablaufs der ADNA-Bausteine und stellen eine wertvolle Ergänzung zu klassischen Testverfahren dar.

Evaluation

Nachfolgend sind exemplarisch die Ergebnisse der Verifikation für ausgewählte ADNA-Komponenten dargestellt:

Tabelle 4: Ergebnisse der formalen Verifikation ausgewählter ADNA-Komponenten

Komponente	Zustände	Übergänge	Verifikationszeit	Geprüfte Eigenschaften
StateMachine	14	22	2,3 s	Deadlock-Freiheit, gültige Modusumschaltung, kein illegaler Rücksprung
StartProcedure	9	13	1,1 s	Korrekte Initialisierung, kein unerlaubter Reset, sichere Aktivierung
PIDController	18	26	3,6 s	Strukturkorrektheit, Fehlervermeidung bei Grenzwertprüfung, Moduswechsel
SafetyCheck	11	17	1,9 s	Prüfen von Invarianten, korrektes Verhalten bei fehlerhaften Eingaben
LoopHandler	24	37	4,8 s	Schleifenstruktur, Rücksprünge bei Fehlern, Deadlock-Freiheit

Fazit

Die durchgeführte formale Verifikation mit UPPAAL hat gezeigt, dass sich sicherheitskritische Kontrollflüsse eingebetteter Softwarekomponenten bereits auf Modellbasis zuverlässig analysieren lassen. Auch ohne Berücksichtigung konkreter Rechenwerte ist die Korrektheit von Zustandsübergängen, Verzweigungen und

1.4 Simulator

Für erste Tests wurden selbstentwickelte Simulatoren genutzt, wie der Automotive Dynamics Simulator der Goethe Universität (Abbildung 16). Da dieser wenig geeignet ist für die Entwicklung von autonomen Fahrzeugen, wurde nach einem State of the Art Simulator gesucht. Es soll mit ihm ein realistisches Umfeld generiert werden können und reproduzierbar Tests durchgeführt werden, damit Systemfehler oder Unfälle jederzeit genaustens untersucht werden können. Er muss auf Windows- und Linux-Systemen laufen.

1.4.1 Vergleich und Auswahl von Simulatoren

Zur Entwicklung eines robusten und flexiblen Simulationsframeworks für autonome Fahrsysteme wurden verschiedene Simulationsplattformen untersucht. Im Mittelpunkt stehen drei etablierte Simulatoren: AirSim [**?**], beamNG.drive [**?**] und CARLA [**?**]. Tabelle 5 bietet eine Gegenüberstellung der jeweiligen Eigenschaften, Vorteile und Einschränkungen. Aufgrund seiner starken Ausrichtung auf den Automobilbereich, der umfassenden Sensorunterstützung sowie der hohen Anpassbarkeit fiel die Wahl auf CARLA.

Tabelle 5: Vergleich von Simulationsplattformen

Merkmal	AirSim	beamNG.drive	CARLA
Open Source	Ja	Nein	Ja
Automobil-Fokus	Eingeschränkt	Mittel	Hoch
Sensorunterstützung	Umfassend	Eingeschränkt	Umfassend
Anpassbare Umgebung	Mittel	Eingeschränkt	Hoch
Community-Support	Mittel	Gering	Hoch

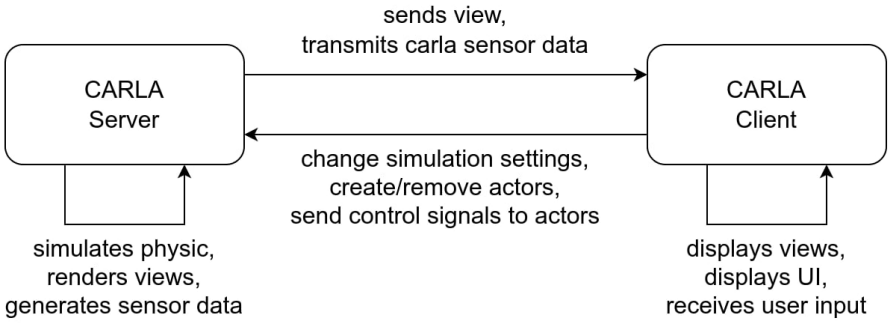

Abbildung 27: Architektur der pyDNAAHS-Schnittstelle.

1.4.2 CARLA Automotive Simulator

CARLA ist ein Open-Source-Simulator zur Entwicklung, dem Training und der Validierung autonomer Fahrfunktionen im urbanen Kontext (vgl. Abbildung 41). Der Simulator basiert auf der Unreal Engine 4[4] und verwendet frei verfügbare 3D-Assets wie Fahrzeugmodelle, Gebäude und Straßennetze [6].

Die Architektur folgt einem Server-Client-Prinzip. Der Server simuliert Umgebung, Fahrzeuge und Sensoren; der Client steuert das Verhalten der Akteure und verarbeitet Sensordaten. Über das Python- bzw. C-Modul `carla` kann der Nutzer Fahrzeuge und Sensoren konfigurieren sowie Steuersignale senden und Daten empfangen.

Sensoren wie Kameras, LIDAR, RADAR, GNSS und IMU werden vom Server simuliert und liefern Daten über Callback-Funktionen an den Client. Dabei kann zwischen periodischen und ereignisgesteuerten Messungen unterschieden werden. Alle im Simulator agierenden Einheiten werden als *Actors* bezeichnet.

1.4.3 pyDNAAHS: Organic Computing Interface für CARLA

Um für Linux-Systeme CARLA (in Python geschrieben) und die AHS-/OC-Middleware (in C geschrieben) zu verbinden, wurde eine Schnittstelle entwickelt, das pyDNAAHS Interface. Windows-Systeme können ohne größere Hindernisse das C-Build von CARLA nutzen und mit der OC-Middleware in C kommunizieren.

Zur Umsetzung ADNA-basierter Steuerung in CARLA müssen mehrere Komponenten parallel ausgeführt werden (siehe Abbildung 27):

CARLA-Server: Originalserver mit ggf. angepassten Startparametern.

[4] `https://www.unrealengine.com/`

CARLA-Client: Python-basierter Client, erweitert um eine C-Schnittstelle zur Kommunikation mit dem AHS-System. Die Integration erfolgt mittels *Cython* und der Python-C-API. Zusätzlich wurden Funktionen zur Datenerfassung und Steuerung per Tastatur implementiert.

pyDNAAHS-Schnittstelle: Die in C entwickelte Python-Schnittstelle vereint die AHS-Kommunikation und Datenverarbeitung. Sie ermöglicht Zugriff auf Sensor- und Aktuatordaten, verwaltet asynchrone Prozesse durch Mutex-geschützte Caches und stellt diese Daten dem ADNA-Netzwerk zur Verfügung. Aktuatordaten werden zurück an den Client und schließlich an den Server übermittelt. Die Implementierung erfolgte unter Linux. Erste Tests zeigen eine stabile Integration, jedoch besteht noch Optimierungspotenzial im Umgang mit Ausnahmen und Fehlermeldungen auf C-Ebene.

Fazit

Mit pyDNAAHS wurde eine funktionale Brücke zwischen dem AHS-System und dem CARLA-Simulator geschaffen. Dadurch ist der Einsatz des ADNA-Modells zur autonomen Fahrzeugsteuerung in einer realitätsnahen Umgebung möglich. Erste manuelle und autonome Fahrexperimente verliefen erfolgreich. Zur Sicherstellung der Reproduzierbarkeit wurden geeignete Testszenarien entworfen. Zukünftige Arbeiten fokussieren sich auf die Erweiterung um spezifische Teststrecken sowie die Analyse von ADNA-gesteuertem Verhalten. Hierzu werden die Sensordaten von pyDNAAHS mit CARLAs framebasierter Trajektorienauswertung verglichen, um umfassende Bewertungen zu ermöglichen.

1.4.4 Nachbildung einer benutzerdefinierten Karte

Ein spezifischer Bereich des Campus Bockenheim der Goethe-Universität Frankfurt wurde mithilfe von OpenStreetMap [**?**] in CARLA rekonstruiert (siehe Abbildung 28). Die Karte wurde vollständig in die Simulationsumgebung integriert, um reproduzierbare Experimente zu ermöglichen. Abbildung 29 vergleicht die Simulationsumgebung der Unreal Engine 4 mit der realen Welt in Abbildung 30 im Detail.

1.4.5 Skriptbasierte Steuerung und Trajektorienreproduktion

Das Fahrverhalten des Fahrzeugs wird durch Skripte gesteuert, die Testeingaben wie Beschleunigung, Bremsen, Lenken und Rückwärtsfahren auslösen. Die wichtigsten Anforderungen an skriptbasierte Experimente sind:

- **Präzise Kontrolle:** Jede Eingabe wird durch boolesche Flags und Eingabelimiter gesteuert.
- **Zeitgesteuerte Ausführung:** Aktionen werden zu bestimmten Zeitpunkten oder bei definierten Geschwindigkeiten ausgelöst.
- **Reproduzierbarkeit:** Die Skripte basieren auf einer modifizierten Version der CARLA-Komponenten `manual_control.py` und `world.py`.

Implementierungsdetails:

Abbildung 28: Integration der benutzerdefinierten Karte in CARLA. Das Straßennetz basiert auf OpenStreetMap-Daten vom Campus Bockenheim der Goethe-Universität Frankfurt rund um die Robert-Mayer-Straße.

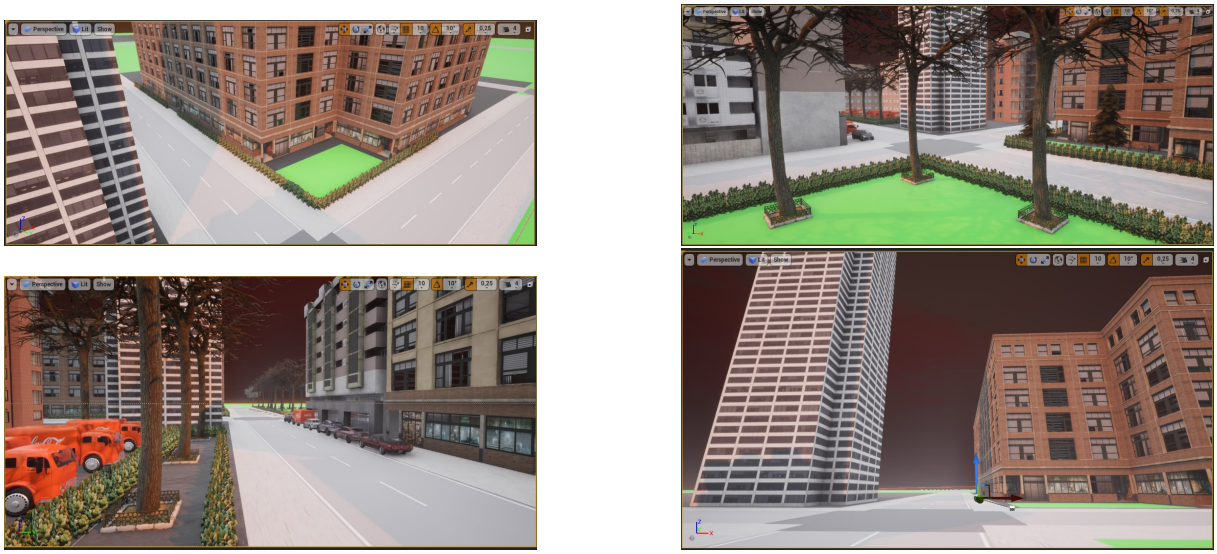

Abbildung 29: Straßenkreuzung der Robert-Mayer-Straße aus verschiedenen Blickwinkeln mit Matheturm und Informatikgebäude, visualisiert in der Unreal Engine 4.

Abbildung 30: Straßenkreuzung der Robert-Mayer-Straße aus verschiedenen Blickwinkeln mit Matheturm und Informatikgebäude, visualisiert in Google Street View.

- Erweiterung des CARLA-Clients um Flags wie `script_active` und `throttle_active`.
- Limitierung unrealistischer Eingabewerte (z. B. maximal 60% Gas).
- Hotkeys zur Aktivierung/Deaktivierung des Skripts (`CTRL+K` bzw. `CTRL+J`).
- Zeitsteuerung mit Python`datetime`-Modul.

Ergebnisse

Zur Evaluierung der Skriptsteuerung und der Reproduzierbarkeit der Fahrzeugbewegung wurden vier Testszenarien durchgeführt. Jedes Testszenario überprüft bestimmte Aspekte der Steuerung (Beschleunigung, Bremsung, Lenkung) in der CARLA-Simulation. Die nachfolgenden Abschnitte beschreiben die jeweiligen Versuchsaufbauten, implementierten Skripte sowie die resultierenden Fahrzeugtrajektorien.

Testfall 1: Verifikation aller Eingaben

Ziel dieses Testfalls ist die Validierung aller steuerbaren Eingaben (Rückwärtsfahrt, Beschleunigung, Bremsung, Lenkeinschläge). Das Fahrzeug wird an der Position `Location (x=110, y=15, z=0.2)` mit einer Orientierung von `Rotation (pitch=0, yaw=180, roll=0)` in der Simulationsumgebung platziert. Das verwendete Steuerungsskript ist in Listing 3 dargestellt.

```
# Test all inputs
if curr_time < start_time + timedelta(seconds=2):
    World.set_reverse_active(self.world)
    World.set_thr_active(self.world)
if start_time + timedelta(seconds=2) < curr_time < start_time +
    ↪ timedelta(seconds=5):
    World.set_thr_active(self.world)
```

```
7  if start_time + timedelta(seconds=5) < curr_time < start_time +
   ↪ timedelta(seconds=7):
8      World.set_bra_active(self.world)
9  if start_time + timedelta(seconds=7) < curr_time < start_time +
   ↪ timedelta(seconds=9):
10     World.set_thr_lim_active(self.world)
11     World.set_thr_lim(self.world, 0.5)
12     World.set_right_turn_active(self.world)
13 if start_time + timedelta(seconds=9) < curr_time < start_time +
   ↪ timedelta(seconds=11):
14     World.set_bra_active(self.world)
15 if start_time + timedelta(seconds=11) < curr_time < start_time +
   ↪ timedelta(seconds=13):
16     World.set_thr_lim_active(self.world)
17     World.set_thr_lim(self.world, 0.5)
18     World.set_left_turn_active(self.world)
```

Listing 3: Skript für Testfall 1: Überprüfung aller Eingaben

Der Ablauf umfasst folgende Phasen:

(a) Rückwärtsfahrt mit 100 % Gaspedal für 2 Sekunden.

(b) Vorwärtsbeschleunigung (ebenfalls 100 %) von Sekunde 2 bis 5.

(c) Bremsen von Sekunde 5 bis 7.

(d) Begrenzte Beschleunigung (50 %) mit rechtem Lenkeinschlag zwischen Sekunde 7 und 9.

(e) Erneutes Bremsen zwischen Sekunde 9 und 11.

(f) Begrenzte Beschleunigung (50 %) mit linkem Lenkeinschlag zwischen Sekunde 11 und 13.

Die aufgezeichnete Trajektorie aus den Sensorlogs ist in Abbildung 31 dargestellt. Die Abbildung bestätigt den erwarteten Verlauf der Eingaben, insbesondere die korrekte Umsetzung der Richtungswechsel und Lenkmanöver.

Testfall 2: Beschleunigung mit maximalem Gaspedalwert

In diesem Szenario wird das Verhalten des Fahrzeugs bei konstanter Beschleunigung über neun Sekunden untersucht. Ziel ist es, zu verifizieren, ob der Gaspedalwert korrekt auf 100 % gesetzt und über die gesamte Zeitspanne gehalten wird.

```
1 # Test accelerate for 9 seconds and hit car
2 if curr_time < start_time + timedelta(seconds=9):
3     World.set_thr_active(self.world)
```

Listing 4: Skript für Testfall 2: Maximale Beschleunigung

Das Fahrzeug beschleunigt schnell und konstant, bis es nach etwa sieben Sekunden mit einem stehenden Fahrzeug kollidiert. Die Sensordaten zeigen eine abrupte Geschwindigkeitsabnahme und eine Richtungsänderung infolge des elastischen Aufpralls. Die resultierende Trajektorie ist in Abbildung 32 zu sehen.

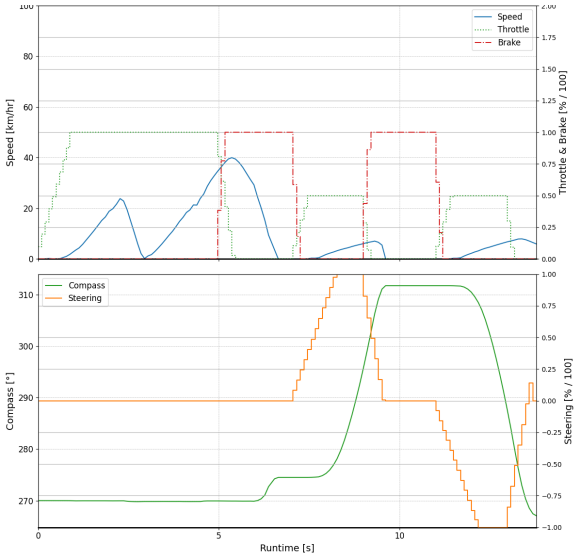

Abbildung 31: Trajektorie aus Sensordaten für Testfall 1: Überprüfung aller Eingaben.

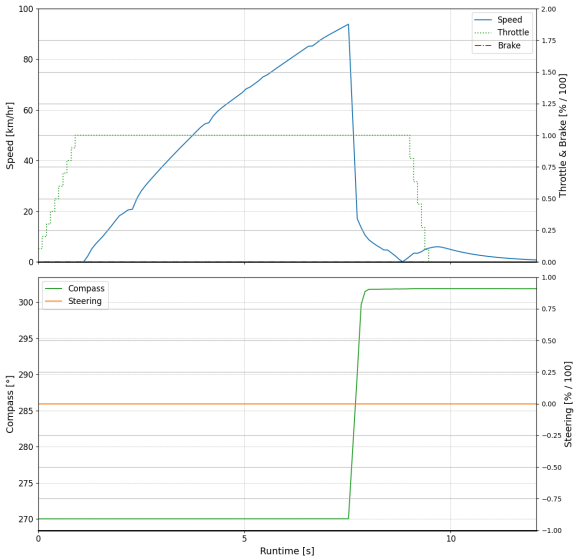

Abbildung 32: Trajektorie aus Sensordaten für Testfall 2: Maximale Beschleunigung.

Testfall 3: Bremsvorgang

Dieser Testfall basiert auf dem vorherigen, ergänzt jedoch um einen Bremsvorgang. Ziel ist es, das Fahrzeug durch gezielte Bremsung rechtzeitig vor dem Hindernis zum Stillstand zu bringen.

```
# Accelerate for 5.5 sec then brake for 2 sec
if curr_time < start_time + timedelta(seconds=5.5):
    World.set_thr_active(self.world)
if start_time + timedelta(seconds=5.5) < curr_time < start_time +
    ↪ timedelta(seconds=9):
    World.set_bra_active(self.world)
```

Listing 5: Skript für Testfall 3: Bremsverhalten

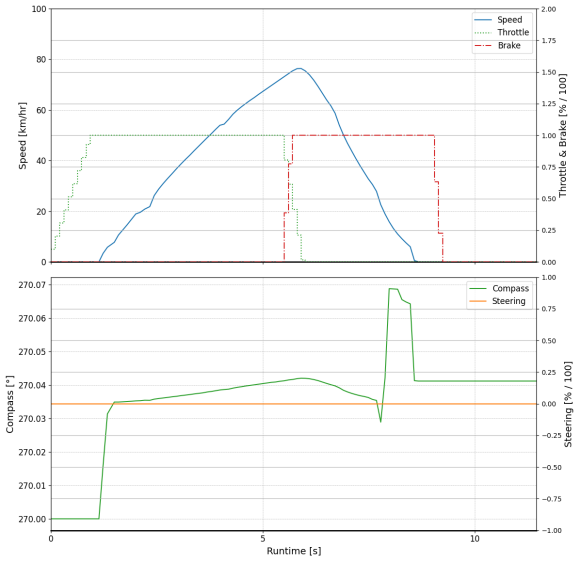

Abbildung 33: Trajektorie aus Sensordaten für Testfall 3: Bremsverhalten.

Die Sensortrajektorie in Abbildung 33 zeigt eine rasche Reduktion der Geschwindigkeit ab Sekunde 5,5. Das Fahrzeug kommt kontrolliert zum Stillstand. Die Kompassrichtung bleibt dabei nahezu konstant mit einer maximalen Abweichung von unter 0,1°, was auf eine geradlinige Fahrt hindeutet.

Testfall 4: Ausweichmanöver mit Lenkung

Der vierte Testfall erweitert Testfall 3 um Lenkbewegungen. Ziel ist es, durch gezielte Lenkmanöver eine Kollision zu vermeiden. Nach einer Beschleunigungsphase erfolgt zunächst ein kontrollierter Linkseinschlag, anschließend ein schneller Rechtseinschlag bei gleichzeitigem Bremsen.

```
# test to avoid car by steering: accelerate, start steering left, steer right and brake
if curr_time < start_time + timedelta(seconds=6.7):
    World.set_thr_active(self.world)
if start_time + timedelta(seconds=6.7) < curr_time < start_time + timedelta(seconds=7):
    World.set_left_turn_lim_active(self.world)
    World.set_left_turn_lim(self.world, -0.4)
    World.set_bra_active(self.world)
if start_time + timedelta(seconds=7) < curr_time < start_time + timedelta(seconds=9) and curr_spd > 0.00000:
    World.set_right_turn_lim_active(self.world)
    World.set_right_turn_lim(self.world, 0.5)
    World.set_bra_active(self.world)
```

Listing 6: Skript für Testfall 4: Lenkmanöver zur Kollisionsvermeidung

Die resultierende Trajektorie (siehe Abbildung 34) zeigt, dass das Fahrzeug durch das Lenkmanöver erfolgreich einem Hindernis ausweicht. Der Richtungsverlauf spiegelt die

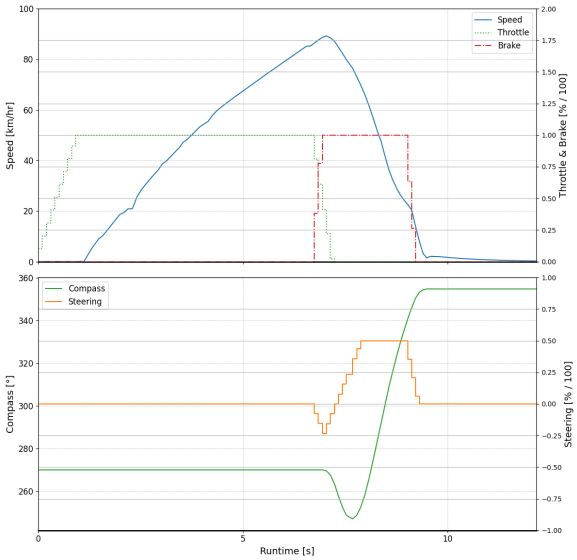

Abbildung 34: Trajektorie aus Sensordaten für Testfall 4: Lenkmanöver zur Kollisionsvermeidung.

Lenkimpulse deutlich wider. Zudem ist ein leichter Schlenker sichtbar, verursacht durch das Überfahren des Bordsteins nach dem Ausweichvorgang.

Fazit

Im Rahmen dieser Arbeit wurde ein reproduzierbares Simulationsframework für autonomes Fahren entwickelt. Nach der Analyse mehrerer Simulatoren wurde CARLA aufgrund seiner automobilen Ausrichtung ausgewählt. Eine reale Karte wurde integriert, skriptbasierte Testszenarien implementiert und vier Testfälle erfolgreich evaluiert. Die Ergebnisse zeigen, dass reproduzierbare Testszenarien mit kontrollierten Steuerimpulsen zu konsistenten Fahrzeugreaktionen führen. Damit bildet das Framework eine solide Basis für die Erforschung selbstadaptiver (ÖC") Systeme in sicherheitskritischen Anwendungsfällen des autonomen Fahrens.

1.5 Demonstrator

1.5.1 Integration des DNA-Prozessors in CARLA und automatisierte Testsystemeinrichtung

Die Integration der DNA-Prozessoren in den CARLA-Simulator ermöglicht eine flexible und effiziente Testumgebung für autonome Fahrfunktionen. Nach der erfolgreichen Verbindung des CARLA-Clients mit den DNA-Prozessoren (siehe Abbildung 35) wird der Status der Verbindung im *AHS DNA Checker* visualisiert. Die DNA-Prozessoren können während der Laufzeit des Clients gestoppt und neu gestartet werden, wodurch verschiedene DNA-Konfigurationen getestet werden können, ohne den gesamten Simulator neu initialisieren zu müssen.

Zur Validierung der Systemkompatibilität wurden die DNA-Prozessoren auf unterschiedlichen Hardwareplattformen getestet: der praxisnahen RazorMotion-Plattform,

Abbildung 35: CARLA Client im Betrieb mit aktivem AHS DNA Checker.

kostengünstigen Raspberry Pi Clustern, einem Linux-Server mit gehostetem CARLA-Simulator sowie einer Windows-Workstation zur Entwicklungsunterstützung. Diese heterogene Testlandschaft gewährleistet eine breite Abdeckung realer Anwendungsszenarien, insbesondere denen im Lastenheft geforderten Vorgaben der Projektpartner in Bezug auf die Systemkompatibelität. Einschränkungen ergaben sich durch Inkompatibilitäten zwischen für amd64 kompilierter Linux-Software und anderen Plattformvarianten, die auf Schnittstelleninkonsistenzen zurückzuführen sind und in zukünftiger Arbeit behoben werden sollen.

Um den zeitintensiven manuellen Aufbau von Testumgebungen zu reduzieren, wurde ein automatisiertes Provisionierungssystem mittels *Ansible* implementiert. Das Playbook automatisiert die Installation und Konfiguration aller notwendigen Komponenten, darunter OpenVPN, TurboVNC sowie CARLA inklusive Abhängigkeiten, und ermöglicht so eine schnelle und reproduzierbare Bereitstellung neuer Testserver. Die gesamte Einrichtung benötigt etwa 40 bis 60 Minuten und wurde erfolgreich auf Ubuntu 24.04.1 getestet.

Die automatisierte Lösung unterstützt eine mehrstufige Benutzerverwaltung und berücksichtigt sicherheitsrelevante Aspekte wie SSH-Zugang und VPN-Konfiguration. Für den Betrieb des CARLA-Simulators sind moderne Hardware-Ressourcen erforderlich, darunter mindestens 24 GB RAM, 100 GB Festplattenspeicher, eine Mehrkern-CPU sowie eine leistungsfähige GPU.

Abschließend lässt sich festhalten, dass durch die Integration der DNA-Prozessoren in CARLA in Kombination mit einer automatisierten Testsystembereitstellung eine effiziente und skalierbare Entwicklungsumgebung für autonome Fahrsysteme geschaffen wurde, die sowohl Flexibilität im Testbetrieb als auch hohe Reproduzierbarkeit bietet.

Sicherheitsmaßnahmen

Die durchgeführten Sicherheitsmaßnahmen gewährleisten einen effektiven Schutz des Testservers gegen unautorisierte Zugriffe bei gleichzeitiger Erhaltung der Nutzerfreundlichkeit. Die Umstellung von passwortbasierter SSH-Authentifizierung auf RSA-Schlüssel reduziert signifikant das Risiko von Brute-Force-Angriffen. Die Beschränkung des VNC-Zugangs durch eine Firewall in Kombination mit SSH-Tunneling sorgt für eine sichere Remote-Desktop-Verbindung, ohne den direkten Zugriff über unsichere Kanäle zu erlauben. Die Einführung dedizierter Benutzerkonten und differenzierter Rechteverwaltung ermöglicht eine feingranulare Zugriffskontrolle und verbessert die Administration des Systems. Diese Maßnahmen zusammen bilden eine robuste Sicherheitsarchitektur, die insbesondere bei der Nutzung durch universitäre und industrielle Partner eine verlässliche und sichere Betriebsumgebung garantiert.

1.5.2 Labor Demonstrator

Anforderungen an die Praxisumsetzung

Für die erfolgreiche Durchführung der Tests im Rahmen des Projekts ist ein Internetzugang mit Remote-Zugriffsmöglichkeiten unabdingbar, da mehrere Partner von unterschiedlichen Standorten auf den Server zugreifen müssen.

Die Rechnerbetriebsgruppe Informatik (RBI) der Goethe Universität Frankfurt stellt hierfür eine virtuelle Maschine mit leistungsfähiger Hardware bereit, insbesondere mit ausreichender Grafikleistung zur Nutzung des CARLA-Simulators.

Die Ausführung der DNA-Prozessoren ist nicht auf den Server beschränkt, sondern umfasst auch externe Clients. Die Kommunikation zwischen den verteilten Komponenten erfolgt über das UDP-Protokoll und wird mittels OpenVPN an alle verbundenen Teilnehmer weitergeleitet, wodurch eine verteilte Zusammenarbeit ermöglicht wird.

Für den Zugriff auf die grafische Benutzeroberfläche ist ein performanter Remote-Zugriff erforderlich. Die klassische X11-Weiterleitung via SSH erwies sich aufgrund der hohen Latenzzeiten und der Rechenintensität des CARLA-Simulators als nicht praktikabel. Stattdessen wurde ein VNC-Server eingesetzt, der mit der Unreal Engine kompatibel ist, auf der CARLA basiert. Insbesondere wurde TurboVNC [**?**] als optimale Lösung identifiziert, da es die 3D-Beschleunigung mittels VirtualGL unterstützt und im Vergleich zum Standard-VNC-Viewer von macOS [**?**] eine deutlich höhere Performance (ca. 15–20 fps Mehrleistung) erzielt.

Alle Projektpartner können mit ihren eigenen Systemen auf den RBI Rechner zugreifen und ihre Experimente durchführen. Dazu kann entweder direkt der hauseigene UDP-Stream der DNA-Prozessoren der Projektpartner zum RBI System getunnelt werden oder eigene Instanzen der zu testenden Software auf dem RBI System gestartet werden.

Portierung des AHS/ADNA-Systems auf die Zielhardware des Demonstrators

Die Portierung des Systems erfolgte auf das RazorMotion-Entwicklungsboard von TT-Tech in einer Linux-Umgebung. Das Board bietet eine moderne Plattform mit mehreren Renesas-Quadcore-Prozessoren unterschiedlicher ASIL-Stufen (zwei ASIL-B, eine

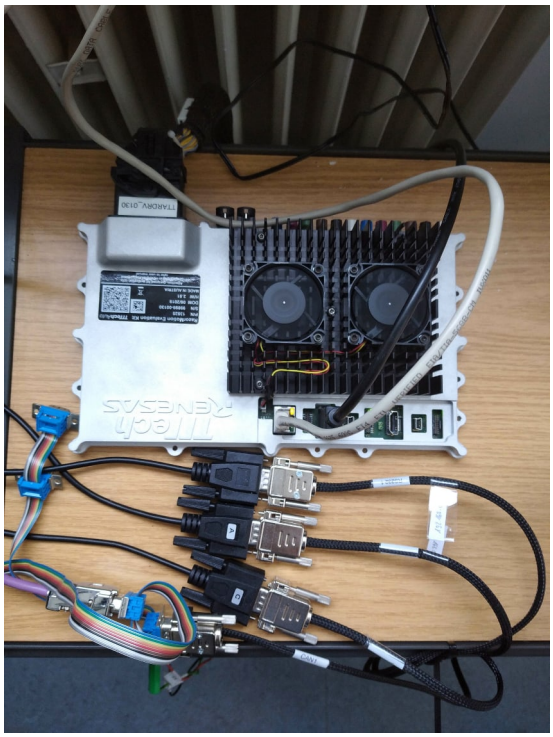

Abbildung 36: Experimenteller Aufbau mit dem RazorMotion-Entwicklungsboard von TTTech.

ASIL-D) [10, 11]. Die Experimente nutzten die beiden ASIL-B-CPUs. Auf dem Board sind sowohl die ADNA als auch ein ausführbarer DNA-Prozessor hinterlegt. Abbildung 36 zeigt den Aufbau.

Das RBI System übernimmt die Simulation der Fahrzeugumgebung sowohl von Server als auch Client, wobei das AHS Sensordaten aus Bereichen wie Lenkung, Bremse und Beschleunigung verarbeitet. Die ADNA ist als virtuelle Steuerungsschicht direkt mit dem Simulator verbunden. Dies ermöglicht eine geschlossene Interaktion zwischen Sensor- und Aktorblöcken sowie dem Simulator. Steuerbefehle aus den Aktorblöcken werden direkt an den Simulator übertragen und dort in Fahrmanöver umgesetzt. Für eine intuitive Bedienung wurde ein Force Feedback Steering Wheel eingebunden.

Die größte Herausforderung lag in der Konfiguration der ADNA-Komponenten sowie der stabilen Kommunikation zwischen allen Subsystemen. Für die Tests wurden bis zu sechs DNA-Prozessoren parallel betrieben, die jeweils eigenständig Sensor-, Aktor- und Offsetblöcke ausführten. Die Kommunikation erfolgte über UDP-Nachrichten an festgelegte Empfänger. Die Verbindung zwischen Simulator und DNA-Prozessor war über ein Ethernet-Kabel realisiert.

Die Stabilität und Performance des portierten Systems erwiesen sich als hoch zuverlässig. Die Initialisierung und Selbstkonfiguration der ADNA-Prozessoren verlief fehlerfrei, auch bei längeren Laufzeiten traten weder Systemabbrüche noch Kommunikationsverluste auf. Die Verarbeitung der Sensordaten sowie die Steuerung des Fahrzeugs erfüllten die Anforderungen an Echtzeitfähigkeit bei niedriger Latenz. Die Fähigkeit zur autonomen Reinitialisierung fehlerhafter Module konnte erfolgreich demonstriert werden, womit das System grundsätzlich für sicherheitskritische Anwendungen geeignet erscheint.

Die CAN-Bus-Kommunikation konnte erfolgreich initialisiert werden. Aufgrund von Bandbreiteneinschränkungen war der Betrieb im Fast-CAN-Modus jedoch nicht möglich. Daher wurde für die weitere Entwicklung Ethernet als primäres Kommunikationsmedium eingesetzt. Die UDP-basierte Kommunikation erwies sich dabei als effizient und stabil.

1.5.3 Live Demonstrator

Das Projekt *SelfAutoDOC* wurde erfolgreich live vorgeführt und demonstriert, wobei die definierten Forschungsziele erreicht und das Potenzial des Organic-Computing-Ansatzes für sicherheitskritische Systeme überzeugend demonstriert werden konnte. Im Rahmen der Vorbereitungen zu den praktischen Tests wurde darüber hinaus weiteres Optimierungspotenzial identifiziert sowie zusätzliche Anwendungsbereiche erschlossen.

Es konnte gezeigt werden, dass das entwickelte AHS in der Lage ist, hohe Datenraten zuverlässig zu verarbeiten und auf verschiedenen Hardwareplattformen stabil betrieben werden kann. Die enge Zusammenarbeit zwischen universitären Forschungspartnern und Industrieunternehmen ermöglichte die Bündelung spezifischer Expertisen und die Entwicklung leistungsfähiger Testsysteme, deren Praxistauglichkeit durch Demonstrationen validiert wurde.

Die praktische Demonstration umfasste mehrere Szenarien, die zentrale *Self-X*-Eigenschaften des Systems hervorhoben. So wurde mittels einer lokal eingerichteten Testserverinstanz und des CARLA-Simulators die Steuerung von Fahrzeugen über DNA-Prozessoren auf unterschiedlichen Plattformen realisiert. Die Selbstheilungsfähigkeit wurde durch das gezielte Trennen von Netzwerkverbindungen erprobt, wobei das System eine kontinuierliche Betriebsfähigkeit aufrechterhielt, solange mindestens ein DNA-Prozessor verfügbar war. Nach vollständigem Ausfall erfolgte eine automatische Reorganisation und Wiederherstellung des Systems.

Des Weiteren demonstrierte das Modul *HormoneGuard* die Fähigkeit, fehlerhafte oder schädliche Hormonmeldungen innerhalb des AHS zu erkennen und zu isolieren. In Versuchen mit mehreren DNA-Prozessoren wurde gezeigt, dass durch den Einsatz von HormoneGuard eine korrekte Lastverteilung trotz aggressivem Fehlverhalten einzelner Prozessoren gewährleistet wird.

Ein weiterer Demonstrationsaufbau beinhaltete eine autonome Fahrfunktion, die auf Sensordaten des CARLA-Simulators basiert und grundlegende Fahrmanöver wie Spurhalten ermöglicht. Auch hier zeigten sich robuste *Self-X*-Mechanismen, die den Betrieb bei simulierten Hardwareausfällen stabilisierten.

Die Industriepartner konnten durch die Nutzung eines eigens entwickelten `ansible-playbook` eigene Testserver effizient aufsetzen und das Organic-Computing-System auf der realen Hardware von PLC2 betreiben. Die praxisnahe Integration von Hard- und Softwarekomponenten wurde durch die Bereitstellung der RazorMotion-Plattform sowie spezieller Analysetools zur Überwachung des Datendurchsatzes ergänzt. Die erfolgreiche Echtzeitverarbeitung großer Sensordatenmengen bestätigt die Anwendbarkeit des Organic-Computing-Ansatzes in hochdynamischen und sicherheitskritischen Umgebungen.

Insgesamt belegt das Projekt die Effektivität und Flexibilität des Organic-Computing-Systems für den Einsatz in sicherheitsrelevanten Anwendungen und eröffnet vielver-

Abbildung 37: Simulation des manuellen Fahrens mit Lenkrad und Pedalen, gesteuert über DNA-Prozessoren auf der RazorMotion-Plattform.

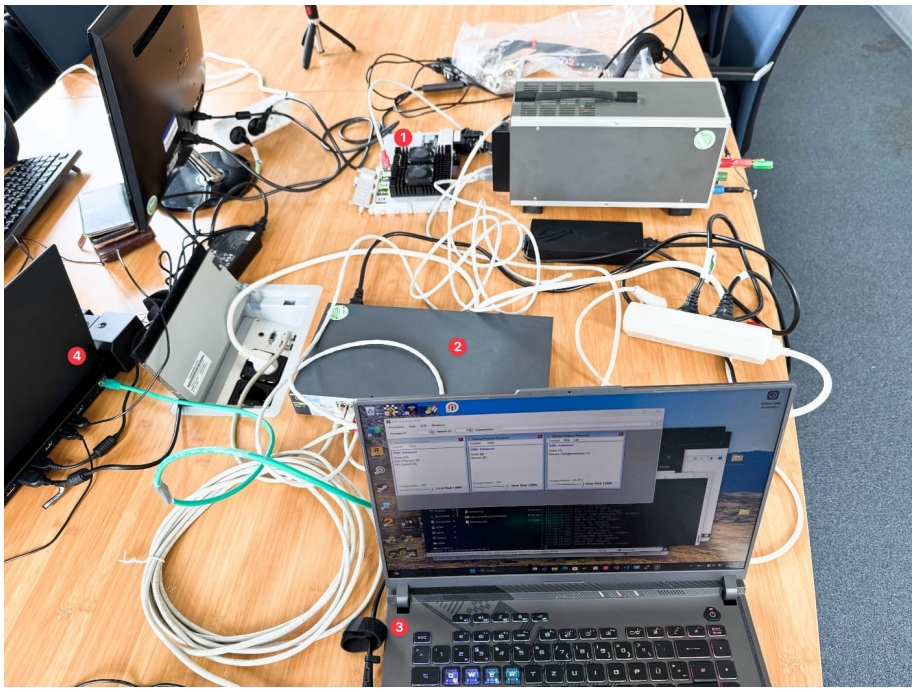

Abbildung 38: Projektaufbau mit: 1) RazorMotion-Plattform, 2) Standard-Switch zur UDP-Kommunikation, 3) Windows-Rechner mit CARLA-Simulator, 4) Steuerrechner für DNA-Prozessoren.

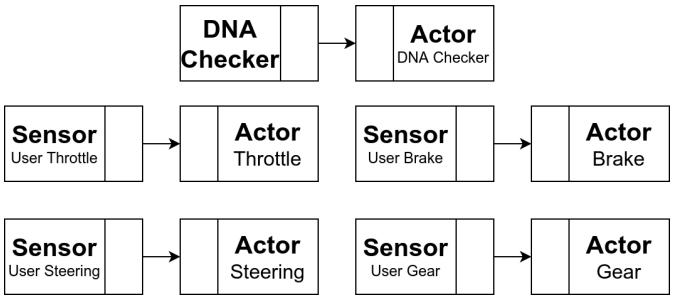

Abbildung 39: Grafische Darstellung der ADNA zur manuellen Steuerung.

sprechende Perspektiven für weiterführende Forschungen und Anwendungen.

1.6 Weitere Entwicklungen

Im Rahmen des Forschungsprojektes gab es weitere Entwicklungen insbesondere auf ADNA-Ebene, aus der sich neue Anwendungen wie das assistive oder autonome Fahren und ein Battery Management System (BMS) für Elektroautos ableiten ließen.

1.6.1 ADNA: Manuelle Steuerung

Tabelle 6: Fahrzeugsteuerungselemente und deren Wertebereiche.

Steuerelement	**Wertebereich**	**Beschreibung**
Gas	$x_{\text{throttle}} \in [0, 1]$	Keine Beschleunigung (0), volle Beschleunigung (1)
Bremse	$x_{\text{brake}} \in [0, 1]$	Kein Bremsen (0), volle Bremskraft (1)
Lenkung	$x_{\text{steering}} \in [-1, 1]$	Maximal links (-1), neutral (0), maximal rechts (1)
Gang	$x_{\text{gear}} \in \{-1, 1, \ldots, 6\}$	Rückwärtsgang (-1), Vorwärtsgänge (1–6)

Ziel der manuellen Steuerung ist es, das Fahrzeug über die ADNA-basierte OC-Middleware benutzerfreundlich steuerbar zu machen. Dazu werden Aktoren benötigt, die Steuerbefehle aus dem AHS empfangen. Diese Aktoren (sowie Sensoren) werden im CARLA-Client mittels Device-IDs definiert und konfiguriert. Um Benutzereingaben in Aktorbefehle umzusetzen, kommen spezielle AHS-Sensoren zum Einsatz, die Eingaben wie Gas, Bremse, Lenkung und Gang erfassen und weitergeben (siehe Tabelle 6).

Die eingesetzte ADNA überträgt die vom Nutzer erzeugten Sensordaten direkt an die zugehörigen Aktoren (vgl. Abbildung 39). Ein *DNA Checker*-Block ist jedem Aktor zugeordnet, um den Status des AHS an den CARLA-Client weiterzugeben (siehe auch Abbildung 41). Der DNA Checker sollte in jeder verwendeten ADNA im CARLA-Client vorhanden sein. Künftige Entwicklungen werden komplexere ADNAs hervorbringen [1, 2] und eröffnen Potenzial zur Erforschung selbst-* Eigenschaften.

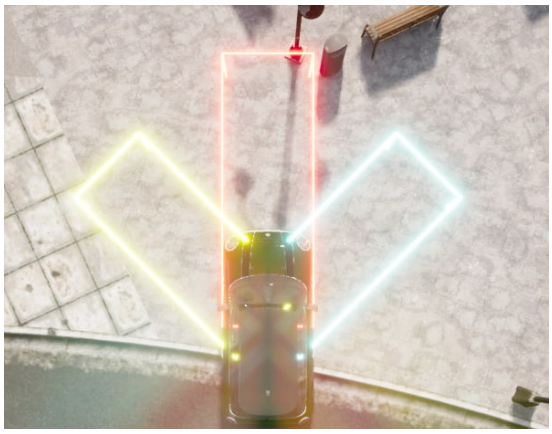

Abbildung 40: Top-Down-Ansicht des CARLA-Clients mit hervorgehobenen Hindernissensoren.

Abbildung 41: Laufender CARLA-Client mit ADNA-basierter OC-Steuerung.

1.6.2 ADNA: Autonome Steuerung

Für autonomes Fahren benötigt das Fahrzeug Informationen über seinen aktuellen Zustand. Entsprechende Sensoren erfassen Geschwindigkeit, Lenkwinkel, Bremskraft und eingelegten Gang. Diese liefern jedoch keine Informationen über die Umgebung. Ein einfacher PID-Regler ermöglicht die Umsetzung eines Tempomaten mit Sollgeschwindigkeit 10 km/h. Zur Hinderniserkennung dienen drei frontseitig ausgerichtete Sensoren (geradeaus, 45° links und rechts), die über Kollisionen mit unsichtbaren Objekten Hindernisse erkennen (siehe Abbildung 40). Die jeweilige Anordnung erlaubt eine grobe Richtungsbestimmung.

Die autonomen Fahrregeln lauten:

1.) Bei freier Strecke (kein vorderes Hindernis): Geschwindigkeit halten (10 km/h).

2.) Bei Hindernis links: Rechtslenkung einleiten.

3.) Bei Hindernis rechts: Linkslenkung einleiten.

4.) Bei Hindernis vorne: Bremsen und Beschleunigung deaktivieren.

Tabelle 7: Verwendete CARLA-Sensoren im CARLA-Client.

Sensor	CARLA-Name	Bemerkung
Kollisionssensor	sensor.other.collision	Reagiert auf Objektkollisionen
IMU	sensor.other.imu	Beschleunigung, Gyroskop, Kompass
Spurwechsel-Sensor	sensor.other.lane_invasion	Erfasst Überfahren von Linien/Markierungen
Front-Hindernissensor	sensor.other.obstacle	6 m Reichweite
Rechter Hindernissensor	sensor.other.obstacle	5 m Reichweite, 45° rechts
Linker Hindernissensor	sensor.other.obstacle	5 m Reichweite, 45° links

Die ADNA zur autonomen Steuerung besteht aus zwei Komponenten:

Fahrregelung – hält die Geschwindigkeit oder reduziert sie bei Hindernissen. Die Differenz zwischen Zielwert und Istwert (vom Geschwindigkeitssensor) wird im ADNA-Block ALU berechnet und an den PID-Regler übergeben. Die Parameter P, I, D wurden empirisch bestimmt. Zur Sicherheit wird die Gasgabe auf den Wertebereich $[0, 1]$

begrenzt. Die obere Hälfte von Abbildung 42 zeigt den Aufbau. Gate-Blöcke aktivieren oder deaktivieren die Steuerung je nach Sensorzustand. Das Bremsmodul (untere Hälfte) sendet bei aktivem Frontsensor eine Bremskraft von 1, während gleichzeitig die Geschwindigkeitsregelung deaktiviert wird (invertiertes Gatesignal).

Lenklogik – reagiert auf seitliche Hindernisse. Erkennt der linke Sensor ein Hindernis, wird ein Lenkbefehl von 0,8 nach rechts ausgegeben; bei rechtem Sensor −0,8 (links). Ohne Erkennung ist der Wert 0. Beide Signale werden summiert und als Sollwert an einen PID-Regler übergeben, der den Lenkaktor steuert. Bei gleichzeitiger Auslösung heben sich die Befehle gegenseitig auf.

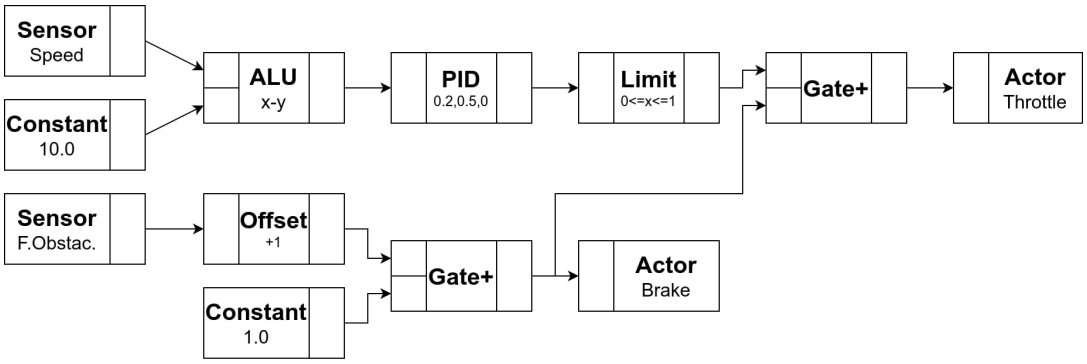

Abbildung 42: ADNA für Geschwindigkeitsregelung im autonomen Fahrmodus.

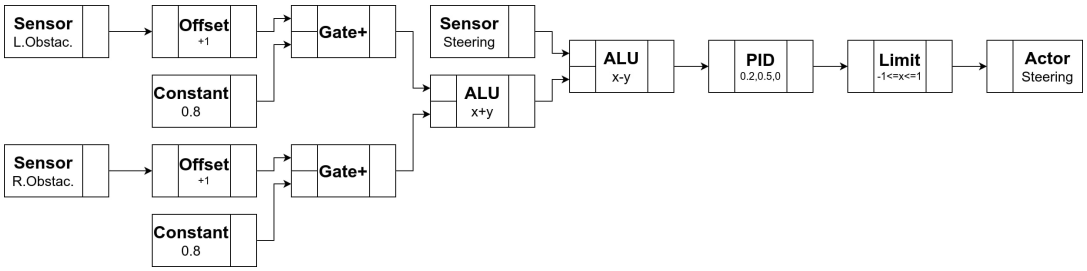

Abbildung 43: ADNA zur Lenklogik im autonomen Fahrmodus.

1.6.3 Assistenzsysteme

Zwei grundlegende Module wurden implementiert:

(a) **Abstandsregeltempomat (Cruise Control):** Dieses Modul hält eine Zielgeschwindigkeit ein und passt sich dynamisch an Hindernisse an. Es nutzt Geschwindigkeits- sowie Frontsensoren zur Erkennung von Objekten und reguliert Gaspedal und Bremse entsprechend.

(b) **Spurhalteassistent (Lane-Keeping Assist):** Zur Erkennung von Fahrbahnmarkierungen wird Kamerabildmaterial mithilfe von OpenCV verarbeitet. Die Lenkung wird in Abhängigkeit von der relativen Fahrzeugposition innerhalb der Fahrspur angepasst.

Ergebnisse

Die Module wurden im CARLA-Simulator unter verschiedenen Testszenarien evaluiert:

- **Abstandsregeltempomat:** Das System konnte Zielgeschwindigkeiten im Bereich von 10–50 km/h zuverlässig einhalten und reagierte effektiv auf dynamische Hindernisse.
- **Spurhalteassistent:** Das Modul zeigte eine hohe Genauigkeit bei der Erkennung von Fahrbahnbegrenzungen und der entsprechenden Lenkungsanpassung, selbst unter variierenden Lichtverhältnissen.

Diskussion

Die Ergebnisse verdeutlichen das Potenzial eines künstlichen Hormonsystems (AHS) zur Erhöhung der Robustheit und Adaptivität autonomer Fahrsysteme. Durch die dezentrale Struktur des AHS werden Systemabhängigkeiten reduziert und die Fehlertoleranz verbessert. Herausforderungen bestehen jedoch weiterhin bei der Verarbeitung datenintensiver Sensorik wie Kamerabildern, die optimierte Kommunikationsprotokolle erfordert.

1.6.4 Batterie Management System

Motivation

Lithium-Metall-Batterien (LiMeBs), d. h. Batterien mit einer festen Elektrolyt-Grenzfläche, bieten eine höhere spezifische Energie als herkömmliche *Lithium-Ionen-Batterien* (LiIonBs) [**?**]. Allerdings sind sie für Anwendungen mit häufigem Ladebedarf bislang ungeeignet, da ihre effektive Kapazität mit jedem Ladezyklus abnimmt. Neue Studien zeigen jedoch, dass sich dieser Kapazitätsverlust durch ein spezifisches Ladeverfahren vermeiden lässt: Die Batterie muss vollständig entladen werden, anschließend eine Ruhephase einhalten und dann ohne Unterbrechung vollständig geladen werden [**?**]. Dies erfordert ein intelligentes *Batteriemanagementsystem* (BMS).

Ein vielversprechender Ansatz zur Umsetzung eines solchen BMS ist der Einsatz eines *Artificial Hormone System* (AHS), eines dezentralen Echtzeitsystems für verteilte eingebettete Systeme [4]. Dieses System besteht aus vielen kleinen, hormonähnlich kommunizierenden Komponenten. Aufgrund seiner Selbst-x-Eigenschaften (z. B. Selbstorganisation, Selbstkonfiguration, Selbstheilung) ist es besonders fehlertolerant und somit prädestiniert für sicherheitskritische Anwendungen wie Elektrofahrzeuge, in denen LiMeBs durch ihre hohe Energiedichte besonders vorteilhaft wären. Ziel dieser Arbeit ist es, die Leistungsfähigkeit eines AHS in einem BMS mit LiMeBs und LiIonBs zu evaluieren.

Simulator

Als Grundlage wurde ein Simulator entwickelt, der das Verhalten einer definierten Anzahl von LiIonBs und LiMeBs nachbildet. Er verarbeitet eine Sequenz von Betriebsmodi (Laden, Entladen, Leerlauf) mit zugehörigem Stromfluss und Zeitintervall. So kann z. B. ein Beschleunigungsvorgang über fünf Sekunden mit einem definierten Entladestrom simuliert werden. Der Simulator ermöglicht den Betrieb mit einem klassischen, codebasierten BMS oder einem AHS und visualisiert den aktuellen Zustand der Batterien sowie die Betriebsparameter (vgl. Abbildung 44).

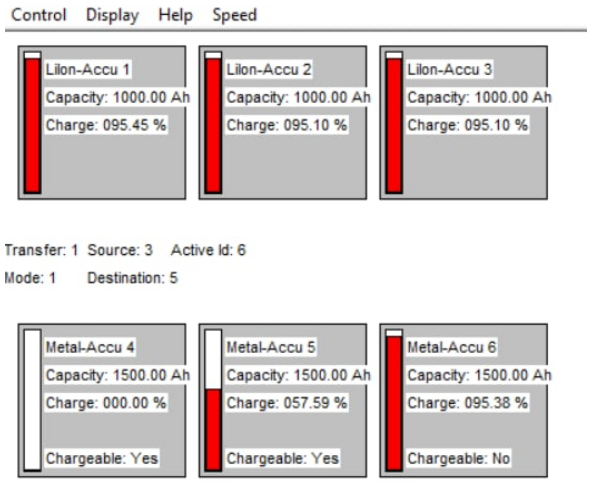

Abbildung 44: Graphische Benutzeroberfläche des Simulators.

Obwohl die Simulationszeiten stark reduziert wurden, liefern die Ergebnisse dennoch einen aussagekräftigen Vergleich zwischen dem klassischen BMS und dem ADNA-basierten System. Für die Simulation wurde eine Abfolge von drei Betriebsmodi (Entladen, Leerlauf, Laden) mit jeweils 20 Sekunden Dauer gewählt.

Batteriesimulation

Zur Modellierung des Batterieverhaltens wurde ein Algorithmus zur Berechnung des *State of Charge* (SoC) bei definiertem Lade- oder Entladestrom entwickelt.
Beim Laden kommt das *Constant Current–Constant Voltage* (CC-CV)-Verfahren zum Einsatz: Zunächst erfolgt das Laden mit konstantem Strom, bis die Maximalspannung erreicht ist. Anschließend wird mit konstanter Spannung und abnehmendem Strom weitergeladen (Abbildung 45).
Beim Entladen bleibt die Spannung weitgehend konstant und fällt bei niedriger Restladung schlagartig ab, während der Entladestrom konstant bleibt (Abbildung 46).

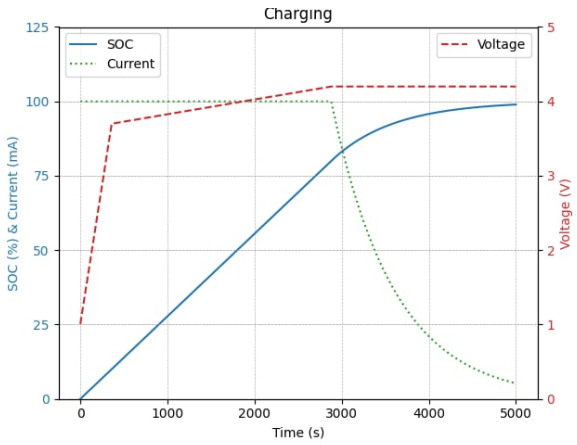

Abbildung 45: Verhalten einer Batterie beim Laden.

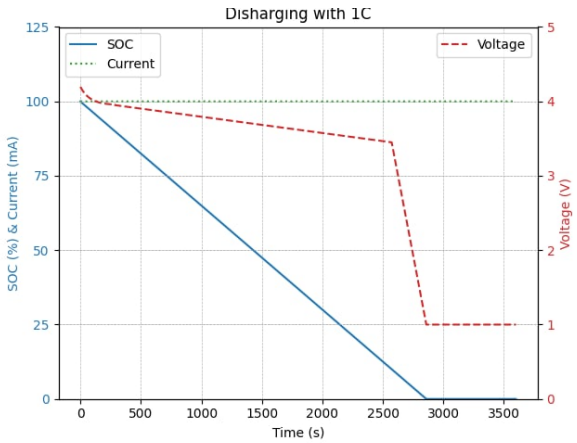

Abbildung 46: Verhalten einer Batterie beim Entladen.

Algorithmus

Ein codebasierter *Battery Management Algorithm* (BMA) wurde implementiert, um ihn mit dem ADNA-basierten System vergleichen zu können. Ziel ist es, die Ladeanforderungen der LiMeBs einzuhalten. Das Konzept basiert auf folgendem Ablauf:
Entladevorgänge bevorzugen die Nutzung von LiMeBs. Ladevorgänge betreffen ausschließlich LiIonBs. Erst wenn genügend Kapazität vorhanden ist und mindestens eine LiMeB bereit zum Laden ist, erfolgt eine Energieübertragung zur LiMeB.

Der Algorithmus läuft kontinuierlich: Nach dem Abruf der aktuellen Zustände wird für jede LiMeB geprüft, ob sie nach Ruhezeit bereit zum Laden ist. Anschließend wird je nach Betriebsmodus die aktive Batterie (activeId) bestimmt. Beim Laden ist dies die LiIonB mit dem geringsten SoC, beim Entladen eine geeignete LiMeB oder – falls keine verfügbar ist – die LiIonB mit dem höchsten SoC. Anschließend wird ein mögliches Quell-Ziel-Paar für die Energieübertragung identifiziert und geprüft, ob die Übertragung zulässig ist. Die Ergebnisse werden an den Simulator zurückgegeben.

ADNA

Ziel ist es zu überprüfen, ob ein ADNA mit vergleichbarer Funktionalität wie der BMA realisiert werden kann. Aufgrund der verfügbaren Basiselemente (ALUs, Sensoren, Aktoren, Gatter) ergeben sich strukturelle Unterschiede. Während der BMA mit Listen von ID-Wert-Paaren arbeitet, berechnet das ADNA simultan Eigenschaften aller Batterien und führt darauf basierende Vergleiche durch. Die resultierenden Funktionsblöcke sind in Abbildung 47 dargestellt. Eine detaillierte Zuordnung der Blöcke zu den jeweiligen Algorithmenzeilen erfolgt in Tabelle 8.
Die vollständige Implementierung umfasst rund 200 Komponenten und erfordert mindestens 15 AHS-Prozessoren.

Ergebnisse

Die zuvor definierte Operationsabfolge wurde sowohl mit dem codebasierten BMA als auch mit dem ADNA durchgeführt. Die Abbildungen 49 und 50 zeigen die aggregierten

Algorithm 1: Codebasierter Battery Management Algorithmus (BMA)

```
1  while true do
2      aktuelle Zustände vom Simulator abrufen;
       // Ladebereitschaft der LiMeBs bestimmen
3      foreach MeBattery do
4          if voll then
5              ReadyForCharge = False;
6          else if leer und Ruhezeit erfüllt then
7              ReadyForCharge = True;
       // activeId bestimmen
8      if Modus == Laden then
9          activeId = LiBattery mit niedrigstem SoC;
10     else if Modus == Entladen then
11         if LiMeB verfügbar then
12             activeId = LiMeB mit niedrigstem SoC;
13         else
14             activeId = LiBattery mit höchstem SoC;
       // Quell- und Zielbatterien für Transfer bestimmen
15     transferDestination = LiMeB mit höchstem SoC und ReadyForCharge;
16     transferSource = LiBattery ≠ activeId mit höchstem SoC;
       // Energieübertragung prüfen
17     verfügbare Kapazität = Summe aller SoCs der LiIonBs (ohne activeId);
18     transfer = möglich, falls Kapazität ausreicht;
19     Ergebnisse an Simulator senden;
```

Block	Funktion	Zeile in Alg. 1
1	Benötigte Kapazität für Transfer berechnen	20
2	LiMeB mit höchstem/tiefstem SoC identifizieren	11,12,15
3	Priorisierung von Entladung aus LiMeBs oder LiIonBs	11
4	Transferfreigabe prüfen	20
5	Ladebereitschaft bestimmen	3–7
6	Auswahl der LiIonB mit höchstem/niedrigstem SoC	9,14,18
7	Verfügbare LiIonB-Kapazität berechnen	19
8	activeId entsprechend Modus bestimmen	8–14

Tabelle 8: Funktionale Blöcke des ADNA

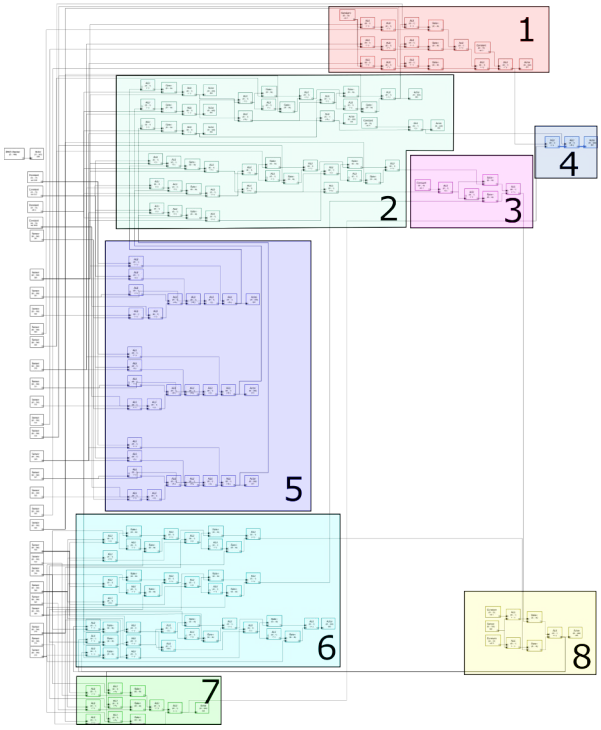

Abbildung 47: Funktionale Struktur des ADNA.

SoCs der LiIonBs und LiMeBs.
Beide Systeme zeigen im Entlade- und Leerlaufmodus ein vergleichbares Verhalten: LiMeBs werden primär entladen, während durch die Energieübertragung aus LiIonBs der SoC der LiMeBs insgesamt steigt. Im Leerlauf wird ausschließlich übertragen, was eine steilere SoC-Zunahme bei den LiMeBs erwarten lässt – wie in Abbildung 50 deutlich zu sehen ist.
Im Lademodus sinkt der SoC der LiIonBs weniger stark, da eine LiIonB geladen wird. Die SoC-Zunahme der LiMeBs verlangsamt sich, da eine Batterie beinahe voll ist und keine weitere geladen werden kann.
Leichte Unterschiede lassen sich durch Implementierungsdetails oder durch Verzögerungen im ADNA erklären. Insgesamt belegen die Ergebnisse die vergleichbare Leistungsfähigkeit beider Systeme.

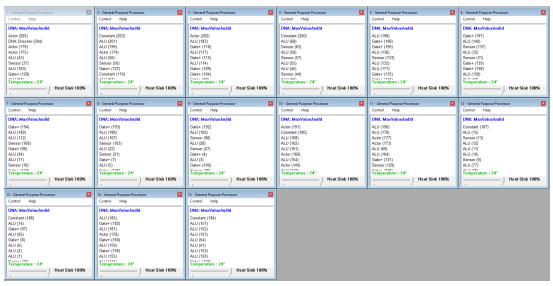

Abbildung 48: Auslastung der Prozessoren des AHS für das ADNA.

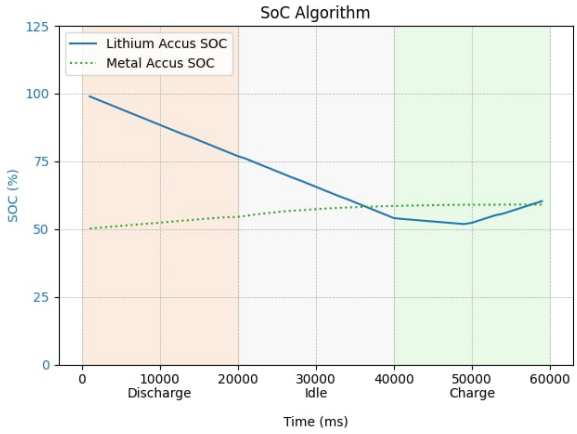

Abbildung 49: SoCs während der Simulation mit dem BMA.

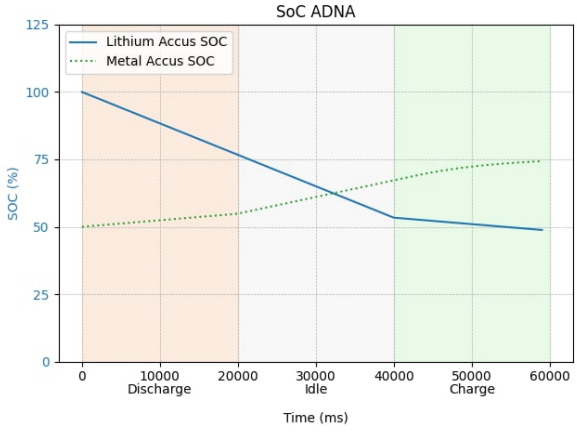

Abbildung 50: SoCs während der Simulation mit dem ADNA.

Zusammenfassung

Ein Simulator zur Modellierung von Lade-/Entladezyklen wurde erfolgreich implementiert. Anschließend wurden zwei Varianten eines Batteriemanagementsystems – ein codebasierter Algorithmus und ein ADNA – entwickelt und miteinander verglichen.
Die Ergebnisse zeigen für beide Systeme ein erwartungskonformes Verhalten. Die Ähnlichkeit der Resultate deutet darauf hin, dass ein ADNA prinzipiell in der Lage ist, die Funktion eines klassischen BMS zu übernehmen.

Ausblick

Zukünftige Arbeiten könnten komplexere Simulationsmuster und längere Sequenzen untersuchen. Zudem wäre die Entwicklung eines eigenen ADNA-Komponentenblocks zur effizienteren Ermittlung der Batterien mit extremen SoCs sinnvoll. Dies könnte die Skalierung auf größere Batteriesysteme erleichtern.

Wissenschaftliche Veröffentlichungen

Wissenschaftliche Vorträge

Tabelle 9: Wissenschaftliche Veröffentlichungen

Under Review	• **PAPER: ADNA-based Organic Computing for Fail-Safe Systems Providing High Dependability: Implementation and Verification of ADNA Building Blocks**
Im Druck	• **PAPER: Generating Singularities for Organic Computing Applications** in Intelligent Embedded Systems - Organic Computing: Doctoral Dissertation Colloquium 2024, Kassel University Press.
2025	• **POSTER: Organic Computing-Driven Vehicles in a Reproducible Test Setup for CARLA Simulator** • **POSTER: ADNA-based Organic Computing Battery Management System**
2024	• **PAPER: The Adaptation Mechanism of Chameleon - A Comprehensive Adaptive Middleware for Mixed-Critical Cyber-Physical Networks** in IEEE 27th International Symposium on Real-Time Distributed Computing (ISORC), Tunis, Tunisia, 2024. DOI:10.1109/ISORC61049.2024.10551345 • **PAPER: An Organic Computing Approach for CARLA Simulator** in Architecture of Computing Systems 2024, Springer International Publishing. DOI:10.1007/978-3-031-66146-4_10 • **PAPER: Redundancy and Self-Healing - Towards an Organic Computing Automotive Environment** in Intelligent Embedded Systems - Organic Computing: Doctoral Dissertation Colloquium 2023, Kassel University Press. DOI:10.17170/kobra-202402269661
2023	• **Evaluating the Comprehensive Adaptive Chameleon Middleware for Mixed-Critical Cyber-Physical Networks** in Architecture of Computing Systems. ARCS 2023. Lecture Notes in Computer Science, vol 13949. Springer, Cham. DOI:10.1007/978-3-031-42785-5_14 • **DISSERTATION: Comprehensive adaptive middleware for mixed-critical cyber-physical networks** in Deutsche Nationalbibliothek Frankfurt 2023. DOI:10.21248/gups.79293 • **PAPER: Organic Computing to Improve the Dependability of an Automotive Environment** in Architecture of Computing Systems 2022, Springer International Publishing. DOI:10.1007/978-3-031-21867-5_14 • **PAPER: Organic Computing in a Vehicle Environment** in Intelligent Embedded Systems - Organic Computing: Doctoral Dissertation Colloquium 2022, Kassel University Press. DOI:10.17170/kobra-202302107484

Tabelle 10: Wissenschaftliche Vorträge

2025	• **Embedded Systems Workshop 2025 - Aktuelle Entwicklungen und Ergebnisse in SelfAutoDOC** Workshop in Galltür
	• **ISORC 2025 - International Symposium on Real-Time Distributed Computing** Fachkonferenz in Toulouse
2024	• **ISORC 2024 - International Symposium on Real-Time Distributed Computing** Fachkonferenz in Tunis
	• **Sensybel 2024 - Rhein-Main-Hochschulverbund** Kolloquium in Hetschbach
	• **OC-DDC 2024 - Organic Computing: Doctoral Dissertation Colloquium 2024** Fachkonferenz in Frankfurt am Main
	• **ARCS 2024 - Architecture of Computing Systems 2024** Fachkonferenz in Potsdam
	• **GRADE - Goethe Research Academy for Early Career Researchers - CompuMath 2024** Kolloquium des GRADE Center in Hirschegg
2023	• **ARCS 2024 - Architecture of Computing Systems 2024** Fachkonferenz in Athen
	• **OC-DDC 2023 - Organic Computing: Doctoral Dissertation Colloquium 2023** Fachkonferenz in Hannover
	• **GRADE - Goethe Research Academy for Early Career Researchers - CompuMath 2023** Kolloquium des GRADE Center in Hirschegg
2022	• **ARCS 2022 - Architecture of Computing Systems 2022** Fachkonferenz in Heilbronn
	• **OC-DDC 2022 - Organic Computing: Doctoral Dissertation Colloquium 2022** Fachkonferenz in Hohenheim

Literatur

[1] Melanie Brinkschulte. Development of a vehicle simulator for the evaluation of a novel organic control unit concept. In R. H. Reussner, A. Koziolek, and R. Heinrich, editors, *INFORMATIK 2020*, pages 883–890. Gesellschaft für Informatik, Bonn, 2021.

[2] Uwe Brinkschulte. Prototypic implementation and evaluation of an artificial dna for self-descripting and self-building embedded systems. *J Embedded Systems*, 23, 2017.

[3] Uwe Brinkschulte and Mathias Pacher. Semantic description of artificial dna for an organic computing middleware architecture. In *Proceedings of the 1st International Workshop on Middleware for Lightweight, Spontaneous Environments*, MISE '19, page 1–6, New York, NY, USA, 2019. Association for Computing Machinery.

[4] Uwe Brinkschulte, Mathias Pacher, and Alexander von Renteln. *An Artificial Hormone System for Self-Organizing Real-Time Task Allocation in Organic Middleware*, pages 261–283. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[5] Butterflytronics. Auto, und, lenkung symbol. copyright. License: CC Version 4.0. `https://icon-icons.com/de/symbol/Auto-und-Lenkung-Rad-transport-Fahrzeug/123460`, N/A.

[6] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[7] E. Hutter and U. Brinkschulte. Towards a priority-based task distribution strategy for an artificial hormone system. In *Architecture of Computing Systems – ARCS 2020*, volume 12155 of *Lecture Notes in Computer Science*. Springer, Cham, 2020.

[8] T. Kisselbach. Redundancy and self-healing – towards an organic computing automotive environment. In S. Tomforde and C. Krupitzer, editors, *Proceedings of the Organic Computing Doctoral Dissertation Colloquium 2023*. Kassel University Press, Kassel, 2024.

[9] H. Ross. *Functional Safety for Road Vehicles. New Challenges and Solutions for E-mobility and Automated Driving*. Springer, Cham, 2016.

[10] TTTech. TTTech Auto AG. Last accessed 26 Jul 2022.

[11] TTTech, Vienna, Austria. *TTTech RazorMotion – Highly automated driving platform for development and evaluation.*

[12] VDE/ITG (Hrsg.). Vde/itg/gi-positionspapier organic computing: Computer und systemarchitektur im jahr 2010. Technical report, GI, ITG, VDE, 2003.

[Frankfurt am Main 1. Juli 2025] [Prof. Dr. Uwe Brinkschulte]