Dokumentation Integrationsprojekt

im Studiengang Information Engineering

von

Studiengruppe HFP424

Prüfer: Prof. Dr. Joel Greenyer

Hannover, 29. September 2025

Inhaltsverzeichnis

Kapitel 1

Motivation

Autonomes Fahren ist in der heutigen Zeit keine ferne Vision mehr, sondern bereits Realität. Unternehmen wie Waymo in San Francisco [Aut25], Baidu in Wuhan [Hou24] oder Volkswagen in Hamburg [Tie25] zeigen, dass fahrerlose Fahrzeuge am Straßenverkehr teilnehmen können. Hierdurch wachsen jedoch auch die Herausforderungen, die Sicherheit solcher Systeme zu gewährleisten und dementsprechend zu validieren. Die Vielzahl potenzieller Verkehrssituationen ist praktisch unbegrenzt, von einfachen Überholmanövern bis hin zu komplexen Interaktionen mit Fußgängern, Radfahrern, Ampeln und mehreren Fahrzeugen gleichzeitig. Hinzu kommen länderspezifische Verkehrsregeln und Regularien, die eine vollständige Erprobung im Realverkehr nahezu unmöglich machen.

Simulationen stellen daher einen unverzichtbaren Baustein in der Validierung autonomer Fahrfunktionen dar, da reale Testfälle nicht in der notwendigen Breite und Tiefe skalierbar sind. Mit Standards wie ASAM OpenScenario existieren bereits Ansätze, Szenarien in strukturierter Form zu beschreiben. Allerdings sind die XML-basierten Szenarien sehr aufwändig zu modellieren und müssen für jedes Testziel individuell erstellt werden. Neuere domänenspezifische Sprachen, wie die auf Constraints basierende OpenScenario DSL, versprechen zwar leichtere Handhabung, sind aber noch nicht weit verbreitet und erfordern ebenfalls manuelle Anpassungen. Industrielle Lösungen wie Fortellix Foretify nutzen Constraint-Solver und Planungsalgorithmen, erweisen sich in der Praxis jedoch häufig als starr und schwer automatisierbar.

Vor diesem Hintergrund ist die Entwicklung neuer, flexiblerer Ansätze zu automatisierten Szenariengenerierung von zentraler Bedeutung. Behavioral Programming bietet hierfür eine attraktive Grundlage. Es erlaubt eine modulare Modellierung von Szenarien und bildet konkurrierende Verhaltensweisen auf natürliche Weise ab. Ergänzend eröffnet Reinforcement Learning die Möglichkeit, aus abstrakten Szenariobeschreibungen, etwa ein Auto überholt das VUT und bremst anschließend, konkrete, ausführbare

Szenarien zu generieren, die den abstrakten Vorgaben genügen.

Dieses Projekt verfolgt das Ziel, die Kombination von Behavioral Programming und Reinforcement Learning als neuartigen Ansatz zur automatisierten Szenariengenerierung zu untersuchen. Dabei sollen nicht nur Methoden zur Modellierung abstrakter und konkreter Szenarien entwickelt werden, sondern auch Verfahren, wie die ausgeführten Szenarien geloggt und visuell aufbereitet werden können, um Test-Engineers eine transparente Analyse zu ermöglichen. Vor diesem Hintergrund ergibt sich die zentrale Forschungsfrage:

> **Wie kann BPpy zur Modellierung ausführbarer Testszenarien im autonomen Fahren eingesetzt werden?**

Zur Beantwortung dieser Frage werden folgende Unterfragen untersucht:

- Wie können Szenarien in BPpy modelliert werden und welche grundsätzlichen Unterschiede bestehen im Vergleich zu OpenSCENARIO?
- Welche Möglichkeiten der Komposition bietet BPpy für Szenarien, etwa sequenziell oder parallel?
- Wie kann Reinforcement Learning eingesetzt werden, um einen Agenten zu trainieren, der in der Lage ist Testszenarien so zu steuern, dass diese abstrakten Szenarien genügen?
- Wie können ausgeführte Szenarien geloggt und visualisiert werden, um eine transparente Analyse zu ermöglichen?

Das Vorhaben versteht sich als Machbarkeitsstudie. Es geht nicht darum, ein industrietaugliches Framework zu entwickeln, sondern vielmehr um die grundlegende Frage, ob sich durch die Kombination von Behavioral Programming und Reinforcement Learning ein flexibler, automatisierbarer Ansatz zur Generierung konkreter Testszenarien aus abstrakten Vorgaben realisieren lässt.

Kapitel 2

Verwandte Arbeiten

Es gibt im näheren Kontext dieses Projekts vier Veröffentlichungen, die im Folgenden vorgestellt und eingeordnet werden.

Das erste Paper stellt das Framework ViSTA zur Szenario-Generierung vor, welches sowohl automatische (auf Basis zufälligen Parametern) und manuelle Testszenarien zulässt [PCA+21]. Dieses Framework fokussiert sich auf eine Testausführung mit SVL und Apollo und ist deshalb für dieses Projekt nicht relevant.

Auch die zweite Veröffentlichung stellt unter dem Titel „ISS-Scenario“ ein eigenes Test-Framework vor, das die Ausführung der Testfälle wird mit CARLA realisiert [LQW24]. Das Paper stellt ein parametrisierten Szenario-Bibliothek vor und zeigt ein Diagramm dazu, dass mit der vorgestellten Methode Kollisionen um ein Vielfaches häufiger provoziert werden können als bei anderen Methoden. Diese Werte wurden allerdings nicht von Dritten bestätigt und das Paper liegt nur als Preprint vor. Daher und da die Simulation mit CARLA den Rahmen dieses Projekts überschreitet, hat dieses Paper keine weitere Bedeutung für das Projekt.

Die Publikation „Generating Critical Test Scenarios for Autonomous Driving Systems via Influential Behavior Patterns“ verfolgt ein ähnliches Ziel wie „ISS-Scenario“ mit „AVASTRA“ – einem Reinforcement Learning Ansatz, der Szenarien bildet, die das zu testende Fahrzeug in gefährliche Situation bringen [TWY+23]. Dieses Paper zeigt, dass RL als effektives Werkzeug bei der Entwicklung von Test-Szenarien eingesetzt werden kann.

In einer Studie mit dem Titel „Testing of autonomous driving systems: where are we and where should we go?“ wurde die Frage untersucht, welche Praktiken aktuell im Testen autonomer Fahrzeuge angewandt werden und welche Unterstützung sich Entwickler dieser Tests wünschen [LDZ+22]. Die Studie kommt zu dem Ergebnis, dass die Entwickler aktuell keine ausreichende Unterstützung bei der Analyse von Fahrzeugdaten durch geeignete Werkzeuge haben.

Kapitel 3

Grundlagen

3.1 Testumgebungen im autonomen Fahren

Um die Sicherheit und Zuverlässigkeit von autonomen Fahrsystemen zu gewährleisten, müssen diese umfassend getestet werden. Dabei werden verschiedene Testumgebungen eingesetzt, um die verschiedenen Anforderungen an die Systeme zu prüfen.

- Software-in-the-Loop (SIL)
  Die zu prüfende Software wird in einer Simulationsumgebung getestet, wobei die Hardware-Komponenten des Fahrzeugs simuliert werden. Ziel ist es Fehler in der Software durch verschiedene Testszenarien frühzeitig zu identifizieren und zu beheben. Auf dieser Ebene ist das wiederholte Testen aufgrund von benötigter Zeit und Rechenleistung am günstigsten, da zum Beispiel schneller als in Echtzeit getestet werden kann. Deshalb wird versucht möglichst viele Tests auf dieser Ebene durchzuführen.

- Hardware-in-the-Loop (HIL)
  Bei HIL-Tests werden reale Zielhardware der Fahrzeuge wie Steuergeräte und Sensoren in die Simulation eingebunden. So kann die Interaktion zwischen Software und Hardware unter kontrollierten Bedingungen relativ realitätsnah getestet werden.

- Vehicle-in-the-Loop (VIL)
  VIL-Tests testen reale Fahrzeuge durch Fahrzeugtests im Verkehr, auf Teststrecken oder im Teststand. Hier wird das komplette Zusammenspiel aller Software und Hardware-Komponenten im Fahrzeug geprüft. Diese Tests sind am aufwändigsten und teuersten, da sie reale Fahrzeuge sowie Testumgebungen benötigen.

3.2 Klassifikation von Testszenarien im autonomen Fahren

Testszenarien im Kontext des autonomen Fahrens können in zwei Kategorien unterteilt werden:

- **Abstrakte Szenarien:** Beschreiben die Verkehrssituation auf einem hohen Grad an Abstraktion. Sie definieren die grundlegenden Anforderungen und die Randbedingungen ohne spezifische numerische Werte. Beispiel Überholszenario: Ein Fahrzeug V1 überholt das VUT.

- **Konkrete Szenarien:** Stellen eine vollständige Instanziierung eines abstrakten Szenarios dar. Sie beschreiben die Anforderungen in präzise, quantifizierbaren Daten. Dazu gehören unter anderem die Positionen, Geschwindigkeiten und Zeitpunkte.
  Beispiel Überholszenario: Auf einer dreispurigen Autobahn startet das Fahrzeug V1 mit $v = 25\frac{\mathrm{m}}{\mathrm{s}}$ auf der linken Spur. Das VUT startet 15m davor auf der mittleren Spur mit $v = 20\frac{\mathrm{m}}{\mathrm{s}}$. Nach 5 Sekunden wechselt das Fahrzeug V1 auf die mittlere Spur vor das VUT.

Diese Unterscheidung ist entscheidend für die automatisierte Szenariengenerierung, da aus allgemeinen Vorgaben eine Vielzahl spezifischer Tests abgeleitet werden können.

3.3 OpenSCENARIO als Standard zur Beschreibung von Testszenarien

Ein Standard für die Beschreibung von Testszenarien für autonomes Fahren ist OpenSCENARIO, welches durch die ASAM (Association for Standardization of Automation and Measuring Systems) entwickelt wurde. Im Laufe der Zeit wurden verschiedene Versionen veröffentlicht, frühe Versionen basieren auf XML, was die Erstellung und Wartung von Szenarien aufgrund des Umfangs und dem Detailgrad komplex macht. Später wurde OpenSCENARIO DSL entwickelt, eine domänenspezifische Sprache, die eine einfachere und intuitivere Beschreibung von Szenarien ermöglicht.

Abbildung 3.1 zeigt einen beispielhaften Ausschnitt eines OpenSCENARIO XML Szenarios. Das Szenario ist dabei in „Acts“ gegliedert, welche jeweils eine Menge von Aktionen definieren. Diese werden durch Subtypen definiert und enthalten unter anderem Informationen über die beteiligten Akteure, deren Start- und Endzeitpunkt sowie die Art der Aktion. Damit ist ein genaues Beschreiben von Testszenarien möglich, allerdings ist die XML-Syntax sehr umfangreich und für Menschen unübersichtlich, dafür aber einfach maschinenlesbar.

```
<Act name="Behavior">
  <ManeuverGroup name="ManeuverSequence" maximumExecutionCount="1">
    <Actors selectTriggeringEntities="false">
      <EntityRef entityRef="adversary"/>
    </Actors>
    <Maneuver name="FollowLeadingVehicleManeuver">
      <Event name="LeadingVehicleKeepsVelocity" priority="overwrite">
        <Action name="LeadingVehicleKeepsVelocity">
          <PrivateAction>
            <LongitudinalAction>
              <SpeedAction>
                <SpeedActionDynamics dynamicsShape="step" value="20" dynamicsDimension="distanc
                <SpeedActionTarget>
                  <AbsoluteTargetSpeed value="$leadingSpeed"/>
                </SpeedActionTarget>
              </SpeedAction>
            </LongitudinalAction>
          </PrivateAction>
        </Action>
        <StartTrigger>
          <ConditionGroup>
            <Condition name="StartConditionLeadingVehicleKeepsVelocity" delay="0" conditionEdge
              <ByEntityCondition>
                <TriggeringEntities triggeringEntitiesRule="any">
                  <EntityRef entityRef="hero"/>
```

Abbildung 3.1: OpenSCENARIO XML Beispiel [Gre25]

In Abbildung 3.2 ist ein weiteres Szenario mit OpenSCENARIO DSL dargestellt. Die DSL-Syntax ist deutlich kompakter und einfacher lesbar als die XML-Syntax, was die Erstellung und Wartung von Szenarien erleichtert. Es werden zwei Fahrzeuge definiert, welche sich auf einer Straße befinden und eine Überholsituation durchführen sollen. Dabei sind im Sinne eines abstrakten Szenarios beispielsweise Intervalle definiert, in denen das Überholmanöver durchgeführt werden soll, sowie die Abstände, die zwischen den Fahrzeugen eingehalten werden sollen. Allerdings wird das Szenario nur deklarativ beschrieben, enthält also keine Anweisungen zur Ausführung, sondern dazu, welche Abläufe valide sind. Ein automatisiertes Testen ist damit dadurch nicht ohne weiteres möglich, da ein Parsen der Syntax und eine Interpretation der Szenariobeschreibung notwendig ist.

```
scenario overtake:
    v1: car # The first car
    v2: car # The second car
    p: path

    do parallel(duration: [3..20]s):
        v2.drive(p)
        serial:
            A: v1.drive(p) with:
                position([10..20]m, behind: v2, at: start)
                lane(same_as: v2, at: start)
                lane(left_of: v2, at: end)
            B: v1.drive(p) with:
                position([1..10]m, ahead_of: v2, at: end)
            C: v1.drive(p) with:
                lane(same_as: v2, at: end)
                position([5..10]m, ahead_of: v2, at: end)
```

Abbildung 3.2: OpenSCENARIO DSL Beispiel [Gre25]

3.4 Eingesetzte Simulations-Engines im Rahmen dieser Arbeit

Simulationsumgebungen sind ein zentrales Werkzeug im Bereich des autonomen Fahrens. Sie ermöglichen es, Verkehrssituationen unter kontrollierten Bedingungen nachzustellen und die entwickelten Verfahren zu testen. Abhängig vom jeweiligen Einsatzgebiet unterscheiden sich die Umgebungen vor allem im Grad der Abstraktion, in den bereitgestellten Funktionen und im benötigten Rechenaufwand. In den folgenden Abschnitten werden mit HighwayEnv sowie SUMO und dem dazugehörigen TraCI zwei Umgebungen vorgestellt, die in unserem Projekt verwendet werden.

3.4.1 HighwayEnv

HighwayEnv ist eine Open-Source-Simulationsumgebung für autonomes Fahren, welches auf der Basis von Gymnasium beziehungsweise OpenAI Gym entwickelt wurde [Leu18]. Der Fokus liegt auf einer schnellen abstrahierten Simulation auf mehrspurigen Straßen und Autobahnen. Das Ziel dieser Simulationsumgebung ist die Entwicklung, Training und Evaluierung von Reinforcement-Learning-Algorithmen und Planungsverfahren im Bereich des autonomen Fahrens.

Technisch ist es mit Python implementiert und folgt einem modularen Aufbau, welche sich an der klassischen Agent-Environment-Schnittstelle aus dem Bereich des Reinforcement-Learnings bedient [Leu18]. Die einzelnen Szenarien werden durch Environment-Klassen definiert, etwa highway-v0 (ein klassisches Autobahn-Szenario) oder roundabout-v0 (ein Kreisverkehr) [Leu18]. Damit eine effiziente Simulation stattfinden kann, nutzt es mittels eines kinematischen Modells eine vereinfachte Fahrzeugdynamik.

Bei der Fahrzeugmodellierung beschränkt sich HighwayEnv auf wesentliche Parameter, wie Position, Geschwindigkeit und Fahrtrichtung. Die Umgebungsmodellierung bietet mehrspurige Straßen, Fahrspurwechsel sowie unterschiedliche Verkehrsdichten [Leu18]. Ein Agent kann beschleunigen, bremsen, die Spur wechseln oder seine aktuelle Bewegung beibehalten. Die im Reinfocement-Learning benötigte Belohnungsfunktion kann je nach Zielstellung beliebig angepasst werden, z.B. auf Sicherheit oder Effizienz.

Der Fokus von HighwayEnv liegt vor allem auf abstrakten Szenarien und nicht hochrealistischen Simulationen. Es ist somit kein Ersatz für High-Fidelity-Simulatoren, wie CARLA. Zudem bietet es keine realistische Physik. Dadurch werden beispielsweise Fahrdynamiken oder Wetterbedingungen vernachlässigt. Der Vorteil von HighwayEnv liegt in den geringen Rechenkosten und ist somit geeignet für Experimente und Prototyping.

3.4.2 Sumo und TraCI

Die Simulation of Urban MObility, kurz SUMO, ist ein Open-Source, mikroskopischer, straßenverkehrsbasierter Simulator, welcher vom Deutschen Zentrum für Luft- und Raumfahrt (DLR) entwickelt wird [LBBW+18]. Das Ziel ist es Verkehrsflüsse und - dynamiken in realistischen Straßennetzwerken zu simulieren. Es unterstützt Einzel- sowie Massensimulationen von Fahrzeugbewegungen in Städten, Autobahnen und Mischverkehr.

Die Implementierungssprache ist im Kern C++. Zudem stehen Python-APIs zur Verfügung mit denen Erweiterungen möglich sind. Mikroskopisch bedeutet in diesem Fall, dass jedes Fahrzeug einzeln simuliert wird [fLuRD25b]. Die Straßennetze können aus realen Karten, wie OpenStreetMap importiert werden, sodass eine hohe Flexibilität geboten ist [fLuRD24b]. Des Weiteren werden verschiedene Fahrzeugtypen von LKWs und Bussen über den PKW bis hin zum Fahrrad oder Fußgänger bereitgestellt.

Für die Fahrdynamik werden Car-Following-Modelle wie das Krauß-Modell genutzt [fLuRD24a]. Darüber hinaus finden Lane-Change-Modelle Anwendung. Innerhalb der Verkehrssteuerung können Ampeln, Zuflussregelungen oder auch Stauszenarien simuliert werden. Des Weiteren gibt es eine eingebaute Routenplanung, sodass Fahrzeuge entweder feste Routen folgen oder auf Basis von dynamischen Routing-Algorithmen gesteuert werden können [fLuRD25a]. Im Bereich der Auswertungen werden verschiedene detaillierte Statistiken in Bezug auf Reisezeiten, Emissionen oder Staus geliefert.

Die Einsatzbereiche von SUMO reichen von der Verkehrsforschung, wobei Verkehrsflüsse, Engpässe oder Infrastrukturmaßnahmen untersucht werden bis hin zur Stadtplanung und Mobilitätskonzepten bei denen Szenarien, wie Car-Sharing oder ÖPNV, simuliert werden [LBBW+18]. Zudem kann es auch im Bereich des autonomen Fahrens als Testumgebung für Entscheidungs- und Koordinationsstrategien dienen.

Im Gegensatz zu HighwayEnv benötigt SUMO bei sehr großen Netzen eine hohe Rechenlast. Es bietet zudem auch keine High-Fidelity, da auch hier ein vereinfachtes Fahrzeugmodell verwendet wird.

Mit dem Traffic Control Interface (TraCI) bietet das DLR eine Client-Server-Schnittstelle zur Echtzeitsteuerung und Abfrage von SUMO-Simulationen [fLuRD25c]. Dies erlaubt die dynamische Interaktion mit einer laufenden SUMO-Simulation. Dabei kann ein lesender Zugriff zur Abfrage von Zuständen etwa der Position, Geschwindigkeit und Ampelphasen erfolgen. Darüber hinaus ist eine Manipulation der Simulation, indem beispielsweise Fahrzeuge hinzugefügt oder Routen verändert werden, möglich [fLuRD25c]. Dies erlaubt die Einbindung von externen Steuerungsalogrithmen z.B. Reinforcement-Learning oder Verkehrsmanagementsysteme.

3.5 Behavioral Programming Paradigma

Behavioral Programming (BP) ist ein sprachunabhängiges Programmierparadigma zur Entwicklung reaktiver Systeme [HMW12]. Dabei steht die inkrementelle Spezifikation von Verhalten im Vordergrund, während klassische Kontrollflussentscheidungen in den Hintergrund treten [HMW12]. BP wurde ursprünglich im Kontext der szenarienbasierten Programmierung entwickelt und später auch in Umgebungen wie Java (als *BPJ*-Framework) und Python (als *BPpy*) umgesetzt [HMW10]. Die zentralen Bausteine dieses Paradigmas sind sogenannte *Behavior Threads* (B-Threads), welche einzelne Verhaltensaspekte kapseln und nebenläufig ausgeführt werden [HMW10]. Alle B-Threads kommunizieren ausschließlich über *Ereignisse* und werden an definierten Synchronisationspunkten vom Laufzeitsystem koordiniert [HMW12].

3.5.1 Fokus auf Verhalten statt Kontrolle

In BP wird jeder Systemaspekt als separates Verhalten (in einem eigenen B-Thread) implementiert, anstatt einen zentralen Ablaufplan zu programmieren. Dadurch ergibt sich eine *indirekte Steuerung*: Anstatt dass ein Modul direkt ein anderes aufruft oder steuert, beeinflussen sich die Verhaltensmodule nur über das Anbieten oder Verhindern von Ereignissen [HMW12]. Dieses Prinzip erlaubt eine natürlichere Beschreibung von Anforderungen, das heißt es wird spezifiziert, was geschehen oder unterbleiben soll, ohne die Ausführung in Form von detailliertem Kontrollfluss explizit zu kodieren [HMW12]. Die Auswahl des nächsten Systemschritts (Ereignisses) erfolgt zur Laufzeit durch das BP-Laufzeitsystem, das aus den von allen B-Threads vorgeschlagenen und gegebenenfalls gesperrten Ereignissen ein zulässiges Ereignis bestimmt und ausführt [HMW12]. BP verlagert den Schwerpunkt der Programmierung somit von der Implementierung von Steuerungslogik hin zur Spezifikation von erwünschtem oder verbotenem Verhalten.

3.5.2 B-Threads und ereignisgesteuerte Synchronisation

Ein B-Thread ist eine in sich geschlossene Verhaltenskomponente, typischerweise implementiert als unabhängiger Prozess (Thread, Coroutine o. Ä.), der wiederholt Ereignisse abgibt oder auf Ereignisse wartet. Die Koordination aller B-Threads erfolgt an expliziten *Synchronisationspunkten.* Jeder B-Thread ruft eine Synchronisationsoperation (etwa *sync*) auf, wobei er drei Kategorien von Ereignissen deklariert [HMW10]:

- **Gewünschte Ereignisse** (*requested events*): Ereignisse, die der Thread vorschlägt und bei deren Auftreten er fortgesetzt werden möchte.

- **Erwartete Ereignisse** (*waited-for events*): Ereignisse, auf die der Thread passiv wartet.
- **Blockierte Ereignisse** (*blocked events*): Ereignisse, die der Thread momentan verhindern möchte.

Sobald alle aktiven B-Threads ihren Synchronisationspunkt erreicht haben, sammelt das Laufzeitsystem alle vorgeschlagenen Ereignisse und wählt eines davon aus, das von mindestens einem B-Thread vorgeschlagen und von keinem B-Thread blockiert wurde [HMW12]. Das ausgewählte Ereignis wird anschließend ausgelöst; alle B-Threads, die dieses Ereignis vorgeschlagen oder darauf gewartet haben, werden aufgeweckt und setzen ihre Ausführung bis zum nächsten Synchronisationspunkt fort [HMW12]. B-Threads, die das Ereignis weder angefordert noch erwartet hatten, bleiben so lange pausiert [HMW12]. Falls mehrere vorgeschlagene Ereignisse gleichzeitig ausführbar sind, kann je nach Implementierung ein Prioritätenschema oder eine feste Rangfolge zur Anwendung kommen, um ein deterministisches Verhalten zu gewährleisten [HMW12]. Anschließend erfolgt die nächste Synchronisation aller Threads, und der Zyklus wiederholt sich. Dieses ereignisgesteuerte Synchronisationsverfahren stellt sicher, dass sich unabhängige Teilverhalten konsistent zu einem Gesamtablauf verweben, ohne dass eine zentrale Steuerlogik die einzelnen Schritte vorgibt.

3.5.3 Modularität und Erweiterbarkeit

Ein Hauptvorteil von BP ist die *hohe Modularität* der Verhaltensbeschreibung und die einfache Erweiterbarkeit von Systemfunktionalität. Neue Anforderungen können als zusätzliche B-Threads implementiert werden, was eine *inkrementelle Entwicklung* ermöglicht, bei der bestehender Code kaum oder gar nicht geändert werden muss [HMW12].
Das Hinzufügen oder Entfernen von Verhaltensmodulen entspricht dem Ein- oder Ausschalten von Szenarien, ohne unbeabsichtigte Seiteneffekte auf den übrigen Ablauf.

Alle Module interagieren ausschließlich über das gemeinsame Ereignis-Vokabular und kennen einander nicht direkt [HMW10]. So kann beispielsweise in einem einfachen Wassermisch-System ein zusätzliches Verhalten (Interleave, abwechselndes Heiß- und Kalt-Zapfen) durch einen neuen B-Thread realisiert werden, der abwechselnd auf das eine Ereignis wartet und das andere blockiert, ohne die bestehenden B-Threads zu verändern [HMW12].

BP unterstützt damit auch Konzepte wie die funktionsorientierten Entwicklung (Feature-Oriented Development) und Produktlinien: Unterschiedliche Ausprägungen eines Systems können durch Konfiguration

der Menge der aktiven B-Threads erzeugt werden [HMW12]. Im Vergleich zu objektorientierten oder aspektorientierten Ansätzen vermeidet BP die strikte Trennung zwischen Basis- und Erweiterungslogik; stattdessen wirken alle Verhaltensmodule gleichberechtigt zusammen [HMW12]. Darüber hinaus lässt sich BP mit anderen Programmierparadigmen kombinieren. B-Threads teilen keinen gemeinsamen Zustand, sondern koordinieren sich ausschließlich indirekt über Ereignisse, was eine lose Kopplung impliziert [HMW12].

Kapitel 4

Problemanalyse

4.1 Problemstellung

Das Projekt „Intelligentes Testen autonomer Fahrzeuge“ hat das Ziel, verifizier- und validierbare Tests für autonome Fahrsysteme (Automated Driving Systems, ADS) auf kostengünstige, reproduzierbare und skalierbare Weise bereitzustellen. Tests im realen Verkehr oder auf Teststrecken sind teuer, gefährlich und oft schlecht reproduzierbar. Deshalb müssen Tests zunehmend in virtuellen Umgebungen durchgeführt werden. Gleichzeitig sind Testfälle für ADS nicht nur einzelne, klar definierte Abläufe, sondern komplexe, parametrisierbare Szenarien mit dynamischen Akteuren (Fahrzeuge, Fußgänger, Wetter, Infrastruktur), was zu einer hohen Kombinatorik und schwer planbaren Ausführungen führt. Diese Kernprobleme gelten als Motivation für eine simulationsbasierte Testumgebung, um Ansätze für die Modellierung und Ausführung von Testszenarien zu erforschen. (vgl. [Gre25])

4.2 Herausforderungen

Szenario-Komplexität und hohe Kombinationenvielfalt Szenarien bestehen aus vielen Akteuren und parallelen Phasen. Die Anzahl möglicher Ausprägungen (Parameterkombinationen bzw. Reihenfolgen von Aktionen) ist sehr hoch, was eine systematische Abdeckung schwierig macht.(vgl. [Gre25])

Unvorhersehbares Verhalten des Vehicle-Under-Test (VUT). Selbst bei korrekt spezifizierten Szenarien kann das VUT anders reagieren als erwartet. Dadurch entstehen zwei verschiedene Situationen, die getrennt voneinander betrachtet werden müssen: (a) Das Testziel konnte in einer Ausführung nicht erreicht werden (Szenario ist nicht erfüllbar) oder (b) das VUT verhält sich falsch

(Testfehlverhalten). Diese Unterscheidung ist für die Auswertung und Weiterarbeit von zentraler Bedeutung.(vgl. [Gre25])

Semantische Lücken in Szenariensprachen Viele formale Szenariobeschreibungen (z. B. OSC DSL) sind deklarativ und liefern keine operationale Semantik, sodass unklar ist, wie eine Ausführung Schritt für Schritt erzeugt werden kann. Deshalb sind Mechanismen erforderlich, die aus abstrakten, constraint-basierten Beschreibungen ausführbare Steuerungsbefehle generieren.(vgl. [Gre25])

Notwendigkeit intelligenter Ausführungsstrategien Naive „Next-Step"-Heuristiken verletzen oft Constraints. Fortgeschrittene Verfahren wie Constraint-Solver, Reinforcement Learning (RL) oder evolutionäre Algorithmen können zwar bessere Ergebnisse liefern, sind aber meist rechenintensiv und benötigen viele Episoden bzw. Iterationen für das Training oder die Suche.(vgl. [Gre25])

Fehlende oder uneinheitliche Observability und Reporting Für eine aussagekräftige Evaluation sind strukturierte Observation-Schnittstellen (z. B. Helper-Funktionen zum Abruf von Abständen, Spurfreigaben etc.) sowie umfassende Log-/Reporting-Mechanismen erforderlich, um Testziele und VUT-Reaktionen transparent auswerten zu können.(vgl. [Gre25])

4.3 Use Case

Um aus den identifizierten Problemen und Herausforderungen konkrete Anforderungen an die Testumgebung abzuleiten, ist es sinnvoll, typische Anwendungsszenarien zu betrachten. Diese Use Cases verdeutlichen, welche Nutzergruppen mit dem System interagieren und welche Ziele sie dabei verfolgen. So wird deutlich, wie die Testumgebung im praktischen Einsatz genutzt werden soll und welche Funktionalitäten sie mindestens bereitstellen muss.

VUT-Tester Als Tester eines VUTs möchte ich konkrete Szenarien in der Testumgebung ausführen und anschließend prüfen lassen, ob die definierten Testziele erreicht wurden.

ML-Forscher Als Forscher möchte ich aus einem abstrakten Szenario konkrete Szenarien durch Verfahren wie RL oder evolutionäre KI erzeugen lassen.

Analyst Als Analyst möchte ich vorhandene Logs und Ergebnisse auswerten, Kennzahlen berechnen und einen Report mit Visualisierungen erstellen.

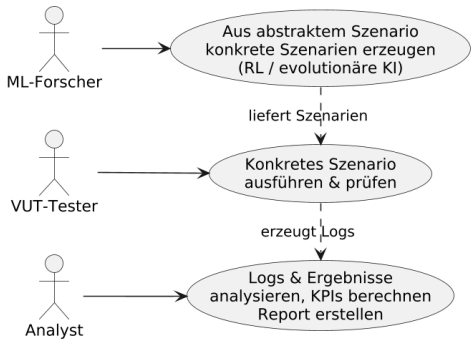

Abbildung 4.1: Use-Case-Diagramm für die Testumgebung

4.4 Technische Anforderungen

Aus der Problemstellung, Use-Cases und den Forschungsfragen aus dem Abschnitt Motivation 1 ergeben sich folgende Anforderungen:

4.4.1 Simulationsumgebung und Adapter

Die Testplattform muss eine simulationsbasierte Infrastruktur bieten, die sowohl schnelle Prototypdurchläufe als auch realistischere Simulationen unterstützt. Zudem muss sie den Wechsel zwischen leichteren Engines (z. B. HighwayEnv) und realistischeren Umgebungen (z. B. SUMO) ermöglichen. Dazu gehört eine Adapterschicht, die simulatorspezifische Details kapselt und eine einheitliche API für Szenarien und Fahrzeuge bereitstellt, um Portabilität und Reproduzierbarkeit zu gewährleisten.

4.4.2 Szenariomodellierung (Abstrakt – Konkret)

Die Plattform muss sowohl abstrakte als auch konkrete Repräsentationen von Szenarien unterstützen und den Übergang zwischen diesen Ebenen ermöglichen. Abstrakte Beschreibungen sollen Constraints, Regeln und Variationsräume formulieren können (zum Beispiel relative Positionen, Timing-Intervalle oder Geschwindigkeitsbereiche). Konkrete Szenarien sollen diese Vorgaben dann mit spezifischen Parameterwerten instanziieren und ausführbar machen. Wichtig ist eine Programmierschnittstelle, die die Erzeugung, Kombination und Wiederverwendung von Teilszenarien erlaubt – etwa in Form modularer Bausteine, die sequenziell oder parallel zusammengesetzt werden können. Diese Struktur erleichtert das systematische Erstellen komplexer Testszenarien und reduziert redundante Implementierungsarbeit.

4.4.3 Observability und semantische Abstraktion

Die Umgebung muss nicht nur Beobachtungsdaten roh liefern, sondern auch eine semantische Abstraktion bereitstellen. Diese umfasst fachlich sinnvolle Metriken und Abfragen (z. B. Abstände, Lane-Clear-Checks, relative Positionen, Geschwindigkeitsdifferenzen). Durch diese semantische Schicht werden Observations für Orchestratoren, Prüfmodule und Lernalgorithmen direkt nutzbar und es wird verhindert, dass jede Komponente eigene, fehleranfällige Interpretationen der Rohdaten vornimmt. Eine konsistente Observations-API stärkt die Wiederverwendbarkeit von Prüflogiken und ermöglicht einheitliche Definitionen von Testzielen und Belohnungsfunktionen. Darüber hinaus vereinfacht sie das Debugging und die Analyse, da relevante Informationen in aussagekräftiger Form vorliegen.

4.4.4 Logging, Persistenz und Reporting

Die Plattform erfasst Ereignisse, Zustandsänderungen, Beobachtungen und Metadaten in strukturierten, maschinenlesbaren Logs (z. B. JSON-Zeilen mit Zeitstempel, Szenario-ID, Ereignistyp, Fahrzeugstatus, Seed und Simulatorversion). Die Logs sind exportierbar und können mithilfe von Such- und Filtermechanismen analysiert werden. Über Schnittstellen zu Visualisierungstools oder einem einfachen Dashboard sind KPI-Übersichten, Timeline-Ansichten und das Rendern von Trajektorien möglich. Langzeitpersistenz, Archivierung von Experiment-Metadaten und Export (z. B. ZIP mit Szenario, Logs und Konfiguration) unterstützen die Reproduzierbarkeit.

4.4.5 Schnittstellen für intelligente Erzeugung

Die Architektur muss das Einbinden und Testen „intelligenter“ Erzeugungs- und Steuerungsverfahren, wie etwa Reinforcement Learning, evolutionäre Algorithmen oder Constraint-Solver, ermöglichen. Wesentlich sind hierbei modulare Schnittstellen für Trainings- und Evaluationsläufe, Konfigurationsmöglichkeiten für Episoden, Mechanismen zur Übergabe und Aggregation von Reward-Signalen sowie die Erfassung von Trainings-Traces für Debugging und Analyse. Dadurch ist es möglich, Lernalgorithmen gezielt auf abstrakte Ziele hin zu optimieren und ihre Ergebnisse in konkrete Steuerfolgen zu überführen. Dadurch kann erforscht werden, ob und wie automatisierte Verfahren die Generierung valider, zielgerichteter Testszenarien verbessern.

Kapitel 5

Umsetzung

5.1 Gesamtarchitektur

Die Architektur besteht aus drei Hauptkomponenten: den Simulations-Engines, der Logging- und Visualisierungskomponente sowie dem BProgram, in dem Szenarien modelliert und ausgeführt werden können. Das BProgram kann zur Ausführung unabhängig zwischen den beiden bereitgestellten Simulations-Engines wählen. Innerhalb des Projekts wurden zunächst konkrete Szenarios manuell implementiert. Im weiteren Verlauf wurde ein Ansatz entwickelt, um konkrete Szenarien automatisiert zu erlernen, sodass diese einem abstrakt definierten Szenario folgen. In den konkreten Szenarien kommt zudem ein Logger zum Einsatz, der Daten für die Visualisierung bereitstellt. Eine zukünftige Vision der Gesamtarchitektur ist in Abbildung 5.1 dargestellt.

Dabei ist zu beachten, dass der in der Abbildung rot markierte Kasten keine bereits implementierte Komponente repräsentiert, sondern eine Vision darstellt, wie die Architektur künftig erweitert werden könnte. Die Idee besteht darin, eine unterstützende DrivingEngine einzuführen, die auf der abstrakten Klasse `gymnasium.Env` basiert und diese gezielt erweitert. So könnte beispielsweise eine zusätzliche Methode `build_vehicles` vorgesehen werden, die von den jeweiligen Umgebungen implementiert werden muss, um sicherzustellen, dass die Fahrzeuge innerhalb der Simulationsumgebung korrekt erzeugt und verfügbar sind. Ergänzend steht ein Vehicle-Interface zur Verfügung, das implementiert werden muss, um unabhängig von der gewählten Umgebung konsistent auf Fahrzeuginformationen zugreifen zu können. Auf dieser Basis ließe sich ein B-Thread definieren, der die DrivingEngine steuert und bei Bedarf flexibel durch eine andere Engine ersetzt werden kann, solange diese die entsprechenden Schnittstellen erfüllt.

Aktuell ist dieser Ansatz jedoch nicht umgesetzt. Stattdessen muss im Code manuell festgelegt werden, ob die Simulation mit SumoEnv oder HighwayEnv ausgeführt werden soll. Ebenso erfordert ein Wechsel

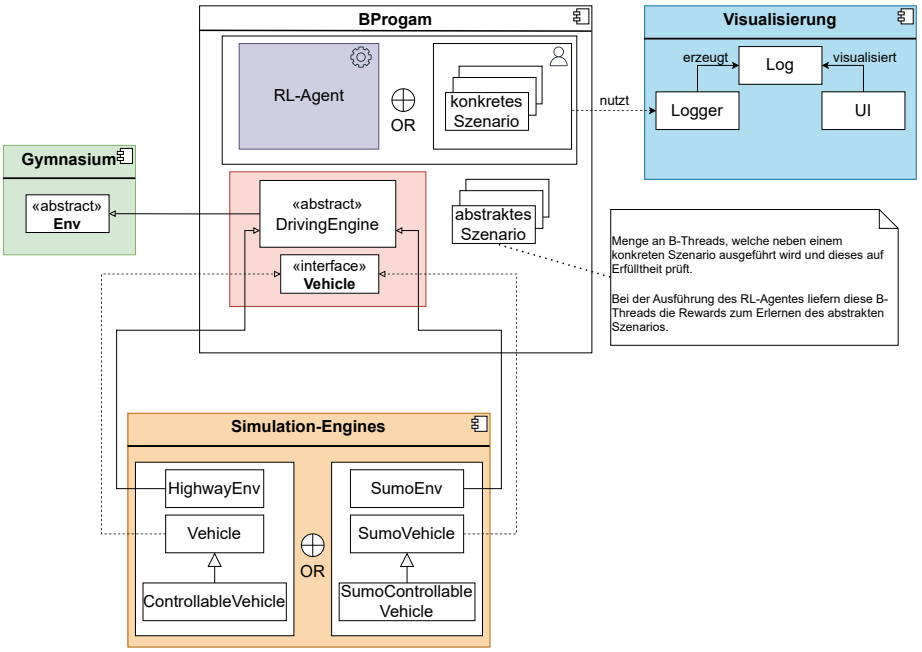

Abbildung 5.1: Vision der Gesamtarchitektur

der Simulationsumgebung derzeit auch Anpassungen an den zugehörigen Vehicle-Zusatzklassen, die in der Komponente Simulation-Engines abgebildet sind. Die vorgesehene DrivingEngine sowie das Vehicle-Interface soll diesen manuellen Aufwand künftig vermeiden und die Austauschbarkeit vereinfachen.

Im Folgenden werden zunächst die beiden Simulations-Engines sowie die Hilfsklassen vorgestellt, die den Zugriff auf Fahrzeuginformationen ermöglichen. Anschließend wird ein Ansatz für die Modellierung in BPpy erläutert, die verschiedene B-Threads zur Ausführung abstrakter und konkreter Szenarien nutzt. Darauf folgt eine Beschreibung der Logging- und Visualisierungskomponente. Abschließend wird der Ansatz zum automatisierten Erlernen konkreter Szenarien auf Basis eines abstrakten Szenarios vorgestellt.

5.2 Simulationsumgebungen

In diesem Abschnitt wird die Nutzung von HighwayEnv und SUMO sowie dessen Steuerungsschnittstelle TraCI beschrieben. Zunächst wird die Konfiguration von HighwayEnv beschrieben sowie zwei Ansätze zur Extraktion von Fahrzeuginformationen aus der Umgebung. Danach wird auf die Sumo-Verwendung innerhalb des Projektes eingegangen.

5.2.1 HighwayEnv

Konfiguration von HighwayEnv

In unserem Projekt kommt die von HighwayEnv bereitgestellte Umgebung `highway-v0` zum Einsatz. Diese simuliert eine mehrspurige Fahrbahn und kann konzeptionell als Autobahnumgebung verstanden werden.

Damit mehrere Fahrzeuge gleichzeitig steuerbar sind, muss die gewünschte Anzahl an kontrollierbaren Fahrzeugen über den Parameter `controlled_vehicles` spezifiziert werden. Zusätzlich lassen sich weitere, nicht kontrollierte Fahrzeuge über den Parameter `vehicles_count` in die Simulation integrieren. Ein spezielles VUT kann somit ebenfalls über diesen Mechanismus berücksichtigt werden.

Da unser Projekt in einem Multi-Agent-Setting ausgeführt wird, ist eine entsprechende Konfiguration notwendig. Hierfür müssen sowohl der Action-Typ (`MultiAgentAction`) als auch der Observation-Typ (`MultiAgentObservation`) gesetzt werden. Auf diese Weise wird sichergestellt, dass:

1. mehrere Agenten gleichzeitig Aktionen ausführen können und
2. die Beobachtungsdaten (Observation) auch für alle kontrollierten Fahrzeuge bereitgestellt werden.

Ansätze zum Zugriff auf Fahrzeuginformationen

Um in HighwayEnv Entscheidungen zu treffen, etwa hinsichtlich Abständen zwischen Fahrzeugen, Spurwahl oder Spurwechseln, ist eine geeignete Repräsentation von Fahrzeuginformationen erforderlich. In diesem Kapitel werden zwei unterschiedliche Ansätze vorgestellt. Zunächst der `ObservationWrapper`, anschließend ein Vehicle-basiertes Modell. Abschließend erfolgt ein Vergleich der beiden Methoden.

Der `ObservationWrapper` kapselt die von der Environment zurückgegebene Observation und interpretiert deren Werte fachlich. Zu Beginn einer Simulation wird dieser einmalig initialisiert und nach jedem Step mit den aktuellen Daten aktualisiert. Auf Basis dieser Struktur können Methoden, wie `is_right_lane_` oder `get_distance_to_leading_vehicle` genutzt werden, um Abstände und Spurverfügbarkeit zu prüfen.

Ein zentrales Kriterium für Korrektheit ist dabei die Reihenfolge der Features innerhalb der Observation. In der aktuellen Implementierung müssen die Werte `["x", "y", "vx", "vy"]` genau in dieser Abfolge am Anfang des Arrays stehen. Nur dann können Distanz- und Geschwindigkeitsberechnungen valide durchgeführt werden. Darüber hinaus ist der Zugriff auf die Environment selbst notwendig, da bestimmte Informationen, etwa die Struktur des Straßennetzes, nicht allein aus der Observation extrahiert

werden können. So lassen sich etwa Lanes über die Road-Network-Struktur bestimmen, was zusätzliche Abfragen an die Environment erfordert.

Damit wird eine relativ direkte Möglichkeit geboten mit den von HighwayEnv bereitgestellten Daten zu arbeiten. Gleichzeitig macht ihn die starke Abhängigkeit von der Feature-Reihenfolge und der Environment-Konfiguration anfälliger für Fehler.

Der zweite Ansatz modelliert Fahrzeuge explizit als Objekte. Dazu stehen die Klassen `Vehicle` und `ControllableVehicle` zur Verfügung.

1. `Vehicle` repräsentiert ein nicht kontrolliertes Fahrzeug und kapselt Methoden, die Informationen direkt aus der Simulation extrahieren. Dazu gehören unter anderem `speed()`, `lane_index()` sowie Distanzwerte wie `delta_pos()` oder `delta_x_pos()`. Zusätzlich lassen sich Relationen wie `is_ahead_of()` oder `is_behind()` zwischen zwei Fahrzeugen prüfen.

2. `ControllableVehicle` erweitert die Funktionalität um konkrete Aktionsmöglichkeiten. Ein solches Fahrzeug kann über Aktionen wie LANE_LEFT, LANE_RIGHT, FASTER, SLOWER oder IDLE direkt gesteuert werden. Damit bildet es die Schnittstelle zwischen Entscheidungslogik und HighwayEnv.

Die Objekte werden beim Start der Simulation einmalig erzeugt und behalten anschließend den Zugriff auf die von ihnen zugeordneten Fahrzeuge. Eine explizite Aktualisierung interner Variablen ist nicht erforderlich, da die Methoden unmittelbar auf die jeweils aktuellen Daten der Environment zugreifen.

Im direkten Vergleich erweist sich der Vehicle-basierte Ansatz als robuster und flexibler, da die Objekte nur einmalig instanziiert werden und ihre Methoden jederzeit aktuelle Daten liefern, entfällt die Gefahr inkonsistenter Zustände. Der `ObservationWrapper` hingegen ist fehleranfälliger denn er ist auf die exakte Reihenfolge der Features in der Observation angewiesen und bei Änderungen der Konfiguration können ungültige Ergebnisse geliefert werden.

5.2.2 Sumo

Im Verlauf des Projektes zeigte sich, dass die Definition eines eigenen Straßennetzes in HighwayEnv über die Implementierung einer individuellen Environment erfolgen muss. Daher wurden alternative Lösungen untersucht, was uns zur Verkehrssimulation SUMO führte. In SUMO können individuelle Karten einfach mit SUMO-netedit erstellt werden. Zudem bietet die Software die Möglichkeit, Straßennetze auf Basis realer Straßen zu generieren.

SumoEnv

Im Gegensatz zu HigwhayEnv bietet SUMO keine direkte Implementierung einer `gym.Env`-Umgebung. Um dennoch Reinforcement-Learning-Experimente in SUMO durchführen zu können wurde eine eigene Umgebung in der Klasse `SumoEnv` entwickelt. Diese basiert auf dem Gymnasium-Framework und bildet die standardisierten Methoden `reset`, `step`, `close` sowie die Definition von Beobachtungs- und Aktionsräumen ab. Voraussetzung für die Nutzung ist die Installation von SUMO sowie das Python-Package TraCI, das eine bidirektionale Steuerung der SUMO-Simulation erlaubt.

Der Konstruktor übernimmt die Initialisierung der Umgebung. Neben der Referenz auf die SUMO-Konfigurationsdatei werden steuerbare Fahrzeuge als Parameter übergeben. Zudem werden die für das Reinforcement Learning relevanten Schnittstellen definiert:

- **Beobachtungsraum**: Als `spaces.Box` modelliert, repräsentiert er kontinuierliche Werte und ist in der aktuellen Version als Matrix der Dimension (20, 5) umgesetzt.
- **Aktionsraum**: Über `spaces.MultiDiscrete` werden diskrete Handlungsoptionen für jedes steuerbare Fahrzeug definiert. Dies ermöglicht eine flexible Steuerung mehrerer Agenten gleichzeitig.

Zur Verwaltung des Simulationsverlaufs werden Episoden- und Schrittzähler geführt (`episode`, `step_count`, `max_steps`).

Zum Starten der Simulation wird in `start_simulation` zunächst geprüft, ob die Umgebungsvariable `SUMO_HOME` gesetzt ist, um die ausführbare SUMO-Datei korrekt zu lokalisieren. Die Verbindung zu SUMO wird anschließend über TraCI hergestellt. Danach werden Parameter zur Zeitschrittlänge, Kollisionsbehandlung und Startoptionen gesetzt. In der GUI werden zudem Kameraeinstellungen angepasst, um die Simulation bestmöglich visuell nachvollziehen zu können.

Das Hinzufügen von Fahrzeugen übernimmt `build_vehicle`. Neben einem vordefinierten Vehicle under Test (vut) werden alle übergebenen steuerbaren Fahrzeuge erzeugt. Mit den individuellen Eigenschaften wie Startzeit, Fahrspur, Geschwindigkeit, Route oder Fahrverhalten können diese weiter angepasst werden. Besonders hervorzuheben ist, dass der Spurwechsel- und Geschwindigkeitsmodus der kontrollierbaren Fahrzeuge so gesetzt wurde, dass möglichst wenige Sicherheitsfeatures von SUMO aktiv sind. Auf diese Weise greifen SUMO-interne Automatikmechanismen, die Fahrmanöver einschränken oder korrigieren, nur minimal ein.

Ein Neustart der Umgebung erfolgt über `reset`. Falls bereits eine Simulation aktiv ist, wird diese zunächst beendet (`close`). Danach erfolgen ein Neustart der SUMO-Instanz sowie die Einbindung der Fahrzeuge. Eine kurze Verzögerung sorgt dafür, dass die Setup-Phase abgeschlossen ist,

bevor auf die Fahrzeuge zugegriffen wird, um Fehler bei der Ausführung zu vermeiden.

Innerhalb der `step`-Methode wird ein Simulations-Schritt, wie folgt ausgeführt:

1. Anwendung der Agentenaktionen
2. Fortsetzung der Simulation um einen Zeitschritt
3. Erhebung der Beobachtung
4. Berechnung der Belohnung
5. Prüfung des Episodenendes

Das Rückgabeformat entspricht dem Gymnasium-Standard: (obs, reward, terminated, truncated, info).

Die Aktionslogik wird in einer separaten Methode `_apply_action()` implementiert. Das Listing 5.1 zeigt die aktuelle Implementierung der Anwendung einer Aktion mittels der TraCI-Schnittstelle. Hierbei wird dieselbe Logik und Enumeration verwendet, wie auch in HighwayEnv.

Die Beobachtungslogik sammelt hierbei für jedes Fahrzeug relevante Zustandsgrößen wie Geschwindigkeit und Position. Fehlerfälle (etwa aus der Simulation entfernte Fahrzeuge) werden abgefangen, indem Default-Werte zurückgegeben werden.

Die Belohnungsfunktion ist aktuell als Platzhalter implementiert und gibt einen konstanten Wert zurück. Damit ist die Umgebung lauffähig, ohne dass bereits eine spezifische Optimierungslogik definiert werden muss. In unserem Anwendungsfall liegt der Fokus ohnehin nicht auf der Optimierung der Simulation, sondern auf der Steuerung der Fahrzeuge, die später einem abstrakten Szenario folgen sollen. Weitere Platzhalter-Methoden wie `_get_info`, `render` oder `_render_frame` sind vorgesehen, um die Funktionalität um detaillierte Diagnoseinformationen oder zusätzliche Visualisierungen zu erweitern. Innerhalb unserer Experimente war dies bisher nicht notwendig.

Die Methode (`close`) beendet die TraCI-Verbindung und sorgt für einen sauberen Abschluss der Simulation. Dies ist insbesondere für die korrekte Durchführung mehrerer Episoden im Training notwendig.

Abbildung der Fahrzeugeigenschaften

Wie bereits in Abschnitt 5.2.1 gezeigt, hat sich der Vehicle-basierte Ansatz als robust und flexibel herausgestellt. Daher wurde er auch im Kontext von SUMO übernommen.

Zu diesem Zweck wurden die Klassen `SumoVehicle` und `SumoControllableVehicle` implementiert. `SumoVehicle` repräsentiert

Listing 5.1: apply_action Implementierung

```
def _apply_action(self, action):
    for i, vehicle in enumerate(self.
    controllable_vehicles):
            vehicle_id = vehicle.vehicle_id
            speed = traci.vehicle.getSpeed(vehicle_id)
            if action[i] == 0:
                if 0 <= traci.vehicle.getLaneIndex(
    vehicle_id) + 1 < 3:
                    traci.vehicle.changeLane(
                        vehicle_id, traci.vehicle
                            .getLaneIndex(vehicle_id) +
    1, 1
                    )
            elif action[i] == 1:
                traci.vehicle.setSpeed(
                    vehicle_id, speed
                )  # Idle action, maintain current speed
            elif action[i] == 2:
                if 0 <= traci.vehicle.getLaneIndex(
    vehicle_id) - 1 < 3:
                    traci.vehicle.changeLane(
                        vehicle_id, traci.vehicle
                            .getLaneIndex(vehicle_id) -
    1, 1
                    )
            elif action[i] == 3:
                traci.vehicle.setSpeed(vehicle_id, speed
    + 1)
            elif action[i] == 4:
                traci.vehicle.setSpeed(vehicle_id, max(0,
     speed - 1))
```

dabei nicht kontrollierte Fahrzeuge und stellt Methoden zur Verfügung, um relevante Fahrzeuginformationen direkt aus der SUMO-Simulation zu extrahieren. `SumoControllableVehicle` erweitert diese Funktionalität um konkrete Aktionsmöglichkeiten, die analog zur `ControllableVehicle`-Klasse im Kontext von HighwayEnv implementiert sind.

Durch diese Struktur müssen bestehende Szenario-Implementierungen lediglich einmalig auf die neue Simulations-Engine und die entsprechenden Fahrzeugobjekte angepasst werden. Die eigentliche Steuerungslogik kann unverändert übernommen werden. Gleichzeitig ermöglicht dieser Ansatz einen schnellen Wechsel der Simulations-Engine, sodass das Experiment-Setup nicht an eine spezifische Plattform gebunden ist und flexibel zwischen HighwayEnv und SUMO oder anderen kompatiblen Umgebungen adaptiert werden kann.

5.3 Modellierung von Szenarien mit BPpy

Für die Modellierung von Szenarien wurde BPpy genutzt. Hierbei wurden verschiedene BThreads erstellt, die unterschiedliche Aspekte der Szenarien abbilden. Einzelne BThreads bilden bestimmte Funktionalitäten und Teilszenarien ab, die dann in Kombination ein vollständiges, konkretes Szenario ergeben. So können die Funktionalitäten insgesamt besser strukturiert und wiederverwendet werden. Die BThreads werden aber außer für einzelne Funktionalitäten auch für die Simulation und Abstraktion von Szenarien genutzt, sowie gebündelt, um ein konkretes Szenario wie ein Überholmanöver abzubilden. So gibt es BThreads, die ein abstraktes Szenario modellieren, also die grundlegenden Anforderungen an das Szenario definieren, ohne sich auf eine konkrete Implementierung festzulegen. Diese abstrakten Szenarien werden dann im BProgram mit ausgeführt um zu überprüfen, dass die Anforderungen an das Szenario erfüllt werden. Dabei werden konkrete Angaben überprüft, wie z.B. das sich bestimmte Fahrzeugpositionen verändert haben oder in welcher Fahrbahn sich die Fahrzeuge im Vergleich zueinander befinden. Dazu kann dann geloggt werden, ob die Anforderungen erfüllt wurden beziehungsweise welche Anforderungen nicht erfüllt wurden.

Der Simulations-Thread hat die Aufgabe, die Simulation zu steuern und sicherzustellen, dass die Szenarien in der Simulationsumgebung korrekt ausgeführt werden. Er sorgt dafür, dass die Fahrzeuge entsprechend den definierten Szenarien agieren und interagieren. Dabei erhält der Simulations-Thread Informationen von den anderen BThreads, um die Simulation entsprechend anzupassen und zu steuern. Z.B. werden in diesem Thread Aktionen der Fahrzeuge ausgeführt, die von anderen BThreads definiert wurden. Dieser Thread dient insbesondere der korrekten Übersetzung der Aktionen für die jeweilige Simulationsumgebung, die in diesem Integrationsprojekt entweder HighwayEnv oder Sumo sein kann. In diesem Thread wird auch auf Kollisionen und andere terminierende Ereignisse geprüft, um die Simulation entsprechend zu beenden oder anzupassen.

Das konkrete Szenario wird durch die Kombination von verschiedenen BThreads modelliert, die zusammen die vollständige Logik und Anforderungen des Szenarios abbilden. Dabei wird spezifische Logik implementiert, die nur für dieses Szenario relevant ist und meist in weitere Methoden ausgelagert wird. Die entsprechende Logik wird dann im relevanten BThread aufgerufen, um einen Teil des Szenarios zu steuern. Dabei können mehrere BThreads auch zu einem konkreten Szenario gehören, um verschiedene Aspekte abzudecken. Diese Threads können über ein Parallelitätskonstrukt parallel ausgeführt werden, um die gleichzeitige Ausführung verschiedener Aspekte des Szenarios zu ermöglichen. Spezifische Logik wird in den einzelnen BThreads per `yield from` aufgerufen, um diese anzuwenden.

Die Aufteilung und Aufgaben der einzelnen BThreads wird jetzt noch einmal an einem konkreten Beispiel, dem Follow-Behind-Szenario, erläutert.

5.3.1 Follow-Behind-Szenario

Das Follow-Behind-Szenario modelliert eine Situation, in der ein oder mehrere Fahrzeuge einem anderen Fahrzeug (VUT) folgt. Dabei wird überprüft, ob das folgende Fahrzeug einen sicheren Abstand zum vorausfahrenden Fahrzeug einhält. Dabei wird auch geprüft, ob dieses Szenario in einer bestimmten Zeit abgeschlossen wird, sodass alle Nicht-VUT-Fahrzeuge sich hinter dem VUT und in derselben Fahrbahn befinden. Diese Anforderungen stellen die grundlegenden Anforderungen an das Szenario dar, die in einem abstrakten BThread modelliert werden. Das entsprechende abstrakte Szenario wird mit geringen Anforderungen (nur die Positionierung in einer vorgegebenen Zeit) in der Methode `abstract_scenario_two_vehicles_follow_vut` modelliert. Das abstrakte Szenario mit den zusätzlichen Anforderungen an die Fahrbahnpositionierung wird in der Methode `abstract_scenario_2` modelliert.

Listing 5.2: Abstraktes Szenario: Zwei Fahrzeuge folgen dem VUT

```
@thread
def abstract_scenario_two_vehicles_follow_vut():
    def condition():
        return v1.is_behind_by_x(vut) and v2.
   is_behind_by_x(v1)

    satisfied = yield from await_condition(condition, 10)
    if satisfied:
        print("################ SAT")
    else:
        print("################ UNSAT")
```

Der Simulations-Thread sorgt dafür, dass die Fahrzeuge entsprechend den definierten Szenarien agieren und interagieren. Dabei erhält der Simulations-Thread Informationen von den anderen BThreads, um die Simulation entsprechend anzupassen und zu steuern. In diesem Beispiel interagiert der Simulations-Thread mit Sumo, um die Aktionen korrekt für die Umgebung zu übersetzen und auszuführen. In früheren Beispielen wurde dies auch für HighwayEnv umgesetzt.5.4 Der Simulations-Thread prüft auch auf Kollisionen und andere terminierende Ereignisse, um die Simulation entsprechend zu beenden oder anzupassen. Zudem wird für alle Fahrzeuge der Simulation geprüft, ob gültige Aktionen vorliegen, die dann las Tupel an `step`-Methode übergeben werden.5.3 Die Simulation wird über die `step`-Methode der Umgebung ausgeführt, die die Aktionen der Fahrzeuge verarbeitet und den Zustand der Simulation aktualisiert. Zuletzt wird die Anzahl an Schritten der Simulation erhöht.

Das konkrete Szenario ist in zwei BThreads gekapselt, die zusammen die vollständigen Anforderungen des Szenarios abbilden. Der erste Teil der Szenario-Logik in dient dazu, dass ein spezifisches Fahrzeug

Listing 5.3: Simulations-Thread Beispiel SumoEnv

```
@thread
def sumo_env_bthread():
    global step_count
    while True:
        # set vehicle_id to one of in this scenario
        # to let it the gui follow that vehicle
        # vut, veh_manual_1, veh_manual_2
        traci.gui.trackVehicle("View #0", "veh_manual_1")
        collisions = traci.simulation.getCollisions()
        if collisions:
            print("Collision detected! Exiting simulation
    ...")
            traci.close()
            raise SystemExit()

        e = yield sync(waitFor=true)

        actions = []
        for vehicle in controllable_vehicles:
            action_vehicle = e.eval(vehicle.
    vehicle_smt_var)
            if action_vehicle in action_map:
                actions.append(action_map[action_vehicle
    ])
            else:
                actions.append(4)  # default is IDLE
        actions_tuple = tuple(actions)

        obs, reward, truncated, terminated, _ = env.step(
    actions_tuple)
        print(f"OBSERVATION in step {step_count}: {obs}")
        step_count += 1
```

einem anderen Fahrzeug folgt. Die Methode nimmt auch eine optionale Verzögerung in Sekunden an. Die konkrete Logik, die für diesen Teil des Szenarios benötigt wird, wird über `yield from` aufgerufen, um die Logik für `follow_behind` anzuwenden. In der Methode `get_behind` wird zuerst die Methode `get_behind` aufgerufen, die eine Wrapper-Methode für einzelne Logik-Bausteine darstellt. Daraus wird Logik aufgerufen, die dafür sorgt, dass das Fahrzeug hinter ein anderes Fahrzeug zurückfällt und in die gleiche Fahrbahn wechselt. Dann wird die dadurch entstehende Distanz zwischen den Fahrzeugen minimiert, indem die Geschwindigkeit des folgenden Fahrzeugs distanzabhängig angepasst wird. Zuletzt um das Szenario mit zwei folgenden Fahrzeugen hinter einem VUT abzuschließen, werden in der Methode `concrete_scenario_two_vehicles_follow_vut`

Listing 5.4: Simulations-Thread Beispiel HighwayEnv

```
@thread
def highway_env_bthread():
    global step_count
    while True:
        evt = yield sync(waitFor=true)
        logger.debug(
            f"highway_env_bthread: step_count: {
   step_count}, action: {evt.eval(v1.vehicle_smt_var)}"
        )
        action_val = evt.eval(v1.vehicle_smt_var)
        if action_val == LANE_LEFT:
            action_index = 0
        elif action_val == LANE_RIGHT:
            action_index = 2
        elif action_val == FASTER:
            action_index = 3
        elif action_val == SLOWER:
            action_index = 4
        else:
            action_index = 1  # IDLE
        env.step(action_index)
        step_count += 1
        env.render()
```

zwei BThreads parallel ausgeführt, die jeweils ein Fahrzeug hinter dem VUT folgen lassen 5.5. Das Ergebnis ist, dass sich zunächst ein Fahrzeug hinter dem VUT positioniert, und, sobald dies erreicht ist, das zweite Fahrzeug sich hinter dem ersten Fahrzeug positioniert. Danach wird für die Fahrzeuge auch der Abstand und die angepasste Geschwindigkeit gehalten.

Listing 5.5: Konkretes Szenario: Zwei Fahrzeuge folgen dem VUT - Relevante BThreads

```
@thread
def follow_behind(
    behind_vehicle: SumoControllableVehicle,
    in_front_vehicle: SumoVehicle,
    delay_seconds: float = 0.0,
):
    # " serial: "
    # yield from wait_seconds(delay_seconds)
    yield from get_behind(behind_vehicle,
    in_front_vehicle)
    yield from stay_behind(behind_vehicle,
    in_front_vehicle)


@thread
def two_vehicles_follow_vut():
    yield from parallel(follow_behind(v1, vut),
    follow_behind(v2, v1))
```

5.4 Visualisierung von ausgeführten konkreten Szeanrien

In diesem Abschnitt wird das Logging-Format und die Visualisierung mit ihren Funktionen zur Analyseunterstützung vorgestellt. Dabei sind sowohl die Beispiel-Logs als auch die Screenshots aus dem Visualisierungstool zur besseren Lesbarkeit in je zwei Teile unterteilt und die für die Abbildungen visualisierten Logs nur in Teilen eingefügt.

Die Grundidee der entwickelten Visualisierung ist die Darstellung der Ereignisse in Form von Balken, die mehrere zusammenhängende Events zusammenfassen, dazustellen.

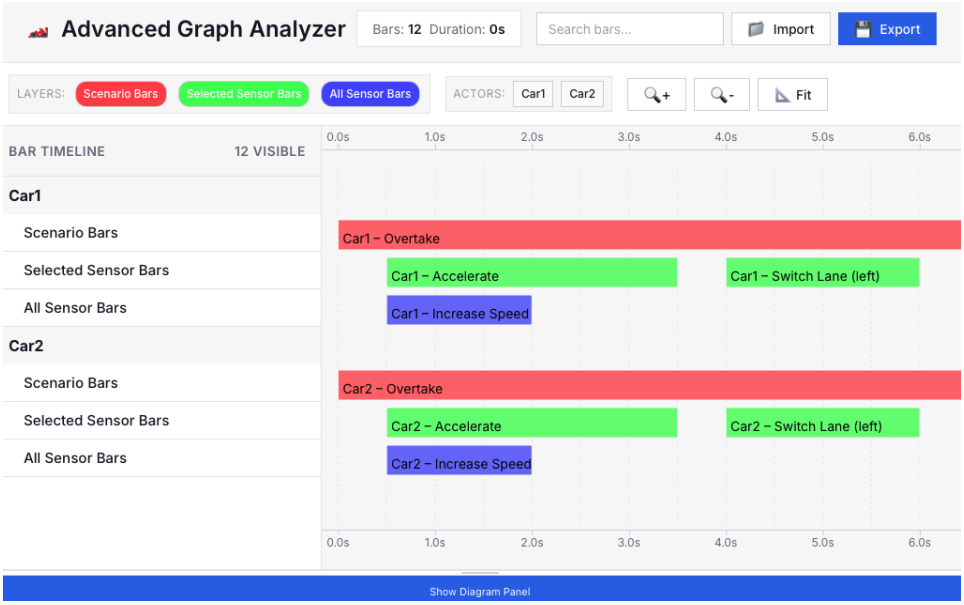

Abbildung 5.2: Visualisierung der Logs im Graph Analyzer

Das Logging folgt grundsätzlich dem Schema, das beispielhaft in den Listings 5.6 und 5.7 zu sehen ist.

Listing 5.6: Beispiel-Log Schichten

```
{
  "layers": [
    { "id": "scenario", "display_name": "Scenario Bars",
    "color": "#FF4444" },
    { "id": "selection", "display_name": "Selected
    Sensor Bars", "color": "#44FF44" },
    { "id": "simulation", "display_name": "All Sensor
    Bars", "color": "#4444FF" }
  ],
  "bars": "Hier stehen Logs"
```

```
}
```

Dabei kann über das `layers`-Attribut eine Liste der vorgesehenen Schichten mitgegeben werden, in denen die gelogten Event angezeigt werden sollen. Darüber hinaus kann der Nutzer hier definieren, welche Namen die Schichten tragen und in welchen Farben die Balken der jeweiligen Schicht dargestellt werden. Im Beispiel also von oben nach unten die Schichten „Scenario Bars“, „Selected Sensor Bars“ und „All Sensor Bars“ wie in Abbildung 5.2 zu sehen ist.

Listing 5.7: Beispiel-Log Einträge

```
{
  "layers": "Hier werden Schichten definiert",
  "bars": [
    { "timestamp": "0.0", "bar_id": "car1-overtake",
    "display_name": "Car1 - Overtake", "actor": "Car1",
    "layer": "scenario" },
    { "timestamp": "3.5", "bar_id": "car1-accelerate",
    "display_name": "Car1 - Accelerate", "actor":
    "Car1", "layer": "selection" },

    { "timestamp": "4.0", "bar_id":
    "car1-switch-lane-left", "display_name": "Car1 -
    Switch Lane (left)", "actor": "Car1", "layer":
    "selection" },
    { "timestamp": "20.0", "bar_id":
    "car1-ride-into-sunset", "display_name": "Car1 -
    Ride into sunset", "actor": "Car1", "layer":
    "selection" },

    { "timestamp": "0.5", "bar_id":
    "car1-increase-speed", "display_name": "Car1 -
    Increase Speed", "actor": "Car1", "layer":
    "simulation", "distance_to_vut": 2.5 },
    { "timestamp": "2.0", "bar_id":
    "car1-increase-speed", "display_name": "Car1 -
    Increase Speed", "actor": "Car1", "layer":
    "simulation", "distance_to_vut": 12.5 },
    { "timestamp": "2.5", "bar_id":
    "car1-increase-speed", "display_name": "Car1 -
    Increase Speed", "actor": "Car1", "layer":
    "simulation", "distance_to_vut": 17.5 }
  ]
}
```

Im darauf folgenden `bars`-Attribut gibt der Nutzer die anzuzeigenden Events an (siehe Listing 5.7). Hierbei wird der jeweilige Zeitpunkt des Events als `timestamp` und die Identifikation des Balken, dem es zugeordnet wird, als `bar_id` mitgegeben. Auf Basis der `bar_id` ermitteltet die Visualisierung

das erste und letzte Event eines Balken und stellt einen entsprechenden Balken dar. Innerhalb dieses Balkens wird dann zur leichteren Identifikation während der Analyse der als `display_name` mitgegebene Name angezeigt (siehe Abbildung 5.2).

Des Weiteren steht in jedem Eintrag ein `actor`-Attribut, welches einen Akteur benennt, dem der Balken zugeordnet wird. Hier im Beispiel sind die Akteure die beteiligten Fahrzeuge „Car1“ und „Car2“.

Im Beispiel zuletzt wird zu jedem Eintrag ein `layer` gelogt, welches die Zuordnung zu den oben erläuterten Schichten herstellt. Werden allerdings keine Schichten als `layers`-Attribut gelogt, werden die hier in den einzelnen Einträgen benannten Schichten genutzt, um Schichten zu identifizieren, in denen die Balken angezeigt werden. Dabei wird auch eine Hierachie inferiert, die jedoch von den gewünschten Hierachie abweichen kann, weshalb empfahlen wird, die Schichten explizit zu definieren.

Die beiden Attribute `actor` und `layer` werden aggregiert und in der Visualisierung genutzt, um dem Nutzer eine Filterung nach Schichten und Akteuren zu ermöglichen. Die daraus generierten Filter-Buttons sind im oberen Bereich von Abbildung 5.2 zu sehen. Darüber hinaus ist auch eine Freitextsuche über das Eingabefeld oben rechts möglich. Diese Möglichkeiten können beliebig kombiniert werden, um die Analyse optimal zu unterstützen.

Neben den zwingend notwendigen Attributen der Log-Einträge können frei weitere Informationen gelogt werden. Im Listing ist zu den letzten Einträgen zusätzlich `distance_to_vut` gelogt. Diese Daten werden genutzt, um Diagramme wie in Abbildung 5.3 zu erstellen, das im „Diagram Panel“ angezeigt wird. Dabei kann der Nutzer die jeweiligen Attribute beliebig benennen. Zu einem Attribut (hier `distance_to_vut`) werden dann alle gelogten Werte in einem Diagramm nach Akteuren unterteilt eingetragen.

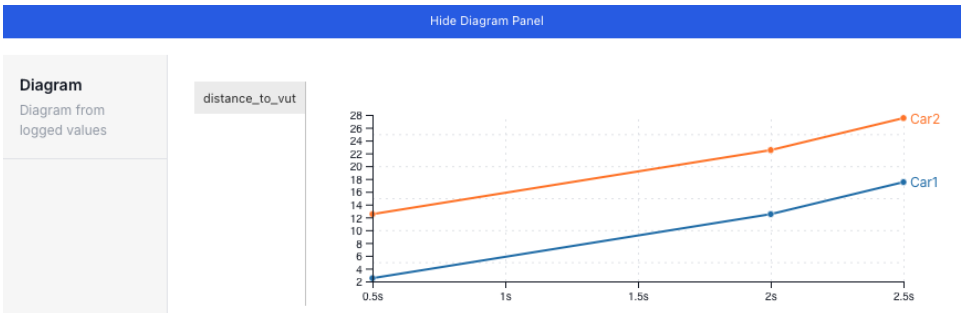

Abbildung 5.3: Diagram Panel im Graph Analyzer

Ein besonderer Vorteil des entwickelten Analysetools ist, dass es durch die Entwicklung ausschließlich mit HTML, CSS und JavaScript jederzeit auf dem Computer des Entwicklers mit Hilfe eines beliebigen Browsers eingesetzt werden kann. Da es als Teil des Projekts abgelegt wurde, ist es während der Arbeit am Projekt stets verfügbar.

5.5 Reinforcement Learning

Hier wird die Struktur und Funktionalität des Projekts, das ein Reinforcement-Learning-Modell (RL) für Überholszenarien trainiert, speichert und nutzt beschrieben. Die Hauptkomponenten des Projekts sind in mehreren Python-Dateien organisiert.

`src/envs/overtake_env.py` Diese Datei definiert die Umgebung für das Überholszenario, bei dem das VUT zum Überholen des vom Modell gesteuerten Fahrzeugs angeregt werden soll. Sie basiert auf einer Kernumgebung und erweitert diese um spezifische Logik für das Training eines RL-Agenten. Analog existiert die Datei `src/envs/intersection_env.py`, bei dem eine Umgebung für ein Kreuzungsszenario definiert wird.

- `__init__(render_mode, **config_overrides)`: Initialisiert die Umgebung mit dem übergebenen Rendermodus. Zusätzlich kann die Konfiguration der Umgebung über den `config_overrides` Parameter überschrieben werden.
- `reset(**kwargs)`: Setzt die Umgebung zurück, initialisiert die Position und Geschwindigkeit des Agenten sowie des zu überholenden Fahrzeugs (VUT).
- `step(action)`: Führt einen Schritt in der Umgebung aus, basierend auf der Aktion des Agenten. Berechnet Belohnungen anhand des Zustandes der Simulation, unter anderem der Position der Fahrzeuge im Vergleich zueinander. Abschließen wird überprüft, ob die Episode beendet ist.
- `render()`: Rendert die Umgebung.

`src/main.py` Diese Datei enthält den Einstiegspunkt für die Ausführung des Projekts und die Konfiguration der Umgebung.

- `create_env(config: Dict[str, Any])`: Erstellt eine neue Umgebung basierend auf der übergebenen Konfiguration.
- `set_config()`: Definiert die Konfiguration der Umgebung, einschließlich Fahrspuren, Fahrzeugpositionen und Beobachtungs-/Aktionsräume.
- `main()`: Führt die Hauptlogik aus, einschließlich der Initialisierung der Umgebung und der Simulation von Aktionen.

`src/training/train_overtake_agent.py` Diese Datei ist für das Training des RL-Modells verantwortlich.

- `make_env(rank: int)`: Erstellt eine Instanz der Überholumgebung für das Training.
- `RenderCallback`: Eine Callback-Klasse, der die Umgebung während des Trainings in regelmäßigen Abständen rendert. Ihre `_on_step()` Methode wird vom Modell während des Trainings aufgerufen um zu bestimmen, ob die aktuelle Episode gerendert werden soll.
- `main()`: Führt das Training des RL-Modells durch. Es initialisiert das Modell und definiert die Trainingsumgebung, trainiert es mit der `DQN`-Methode und speichert das trainierte Modell.

5.5.1 Trainieren des Modells

Das Training erfolgt in der Datei `train_overtake_agent.py`. Die Hauptschritte sind:

1. Initialisierung der Umgebung mit `DummyVecEnv`, welchem das zu lernende Szenario übergeben wird.
2. Definition des RL-Modells mit `DQN`. Dabei werden diverse Parameter wie der Diskontierungsfaktor für das Lernen des Modells festgelegt.
3. Start des Trainings mit `model.learn()`.

5.5.2 Speichern des Modells

Das trainierte Modell wird mit einem Zeitstempel versehen und im Verzeichnis `models/` gespeichert:

```
model.save("models/overtake_dqn.zip" + current_time)
```

5.5.3 Nutzen des Modells

Das gespeicherte Modell kann später geladen und für Inferenz oder weitere Trainingsschritte verwendet werden:

```
from stable_baselines3 import DQN
model = DQN.load("models/overtake_dqn.zip")
```

Kapitel 6

Evaluierung

Kapitel 7

Fazit

Dieses Projekt hat innovative Methoden zur Simulation und zum Testen autonomer Fahrsysteme erfolgreich erforscht. Die Kombination von Behavioral Programming mit BPpy und den Simulationsumgebungen SUMO und HighwayEnv ermöglichte sowohl abstrakte als auch konkrete Szenarien modellieren und evaluieren zu können.

Modulare Szenarienmodellierung BPpy zeigte sich als sehr geeignet, um komplexe, konkurrierende Verhaltensaspekte modular zu spezifizieren. Die Nutzung von B-Threads ermöglicht eine flexible, inkrementelle Erstellung von Testszenarien mit hoher Wiederverwendbarkeit.

Stärken der Simulationsplattformen

- **SUMO** überzeugte durch realistische Karten und Verkehrsabläufe, insbesondere für komplexe Praxisszenarien.
- **HighwayEnv** ist als effiziente Prototyping-Plattform ideal für erstes RL-Training, geriet aber bei hoher Komplexität an Grenzen.

Erfahrungen mit Reinforcement Learning Das Zusammenspiel von RL-Agenten und BPpy bietet ein großes Potenzial, stellt aber hohe Anforderungen an Belohnungsdesign und Schnittstellenkonsistenz. Erste Implementierungen zeigten Wege zur episodischen Szenariosteuerung, machten aber auch den Forschungsbedarf deutlich.

Logging und Reporting Die systematische Erfassung und Auswertung von Simulationsdaten führte zu transparenten, nutzerfreundlichen Visualisierungen und Dashboards. Dies unterstützt eine zielgerichtete Analyse und Fehleridentifikation im Testprozess.

Herausforderungen und offene Punkte

- Heterogene Simulationsschnittstellen und mangelhafte Dokumentation forderten viel methodische Flexibilität.
- Die Skalierbarkeit komplexer Szenarien und das Belohnungsengineering sind weiterhin anspruchsvolle Aufgaben.
- Erweiterungen bezüglich Umwelteinflüssen und fortgeschrittener RL-Anbindung sind dringend erforderlich.

Zusammenfassend zeigt dieses Projekt, dass die Verbindung moderner KI-Methoden, modularer Softwarearchitektur und realitätsnaher Simulation eine solide Basis für zukünftige Testverfahren autonomer Fahrfunktionen bietet. Die gewonnenen Erkenntnisse können wertvolle Impulse für weiterführende Forschung und praktische Umsetzungen liefern.

Der Weg zur vollständigen Automatisierung und Stabilisierung der Szenariengeneration sowie die Verbesserung der Nutzerfreundlichkeit bleiben zentrale Aufgaben künftiger Entwicklungsarbeit.

Kapitel 8

Ausblick

Die im Projektverlauf gewonnenen Erkenntnisse zeigen eindrucksvoll, welches Potenzial die Kombination aus Behavioral Programming, Reinforcement Learning und flexibler Simulation für die Entwicklung und Überprüfung autonomer Fahrsysteme bietet. Auch wenn dieses Vorhaben im Rahmen eines Masterstudiums stattfand und nicht auf eine industrielle Anwendung abzielt, konnten zentrale Methoden auf Praxistauglichkeit überprüft und wichtige Grundlagen vermittelt werden.

Rückblickend eröffnen sich aus unserer studentischen Perspektive verschiedene Möglichkeiten für zukünftige Arbeiten und Vertiefungen:

- **Komplexere Szenarien und Umgebungen:** In folgenden Projekten könnten noch anspruchsvollere Szenarientypen, etwa mit Kreuzungen, wetterspezifischen Einflüssen oder besonderen Fahrmanövern, modelliert werden. Die Ausgestaltung realistischer Verkehrsabläufe bleibt ein spannendes und lehrreiches Forschungsfeld.

- **Automatisierung der Szenarienerstellung:** Es wäre erstrebenswert, die gesamte Generierung von Testszenarien weiter zu automatisieren und deren Konsistenz zu verbessern. Dies würde nicht nur den Projektaufwand verringern, sondern auch die Aussagekraft von Simulationen deutlich steigern.

- **Weiterentwicklung von KI-Steuerung und RL:** Die Verbindung aus Reinforcement Learning und modularen Szenario-Architekturen sollte weiter vertieft werden, um robuste und vielseitige KI-Agenten zu entwickeln. Insbesondere das Belohnungsdesign und die Kopplung an geeignete Simulationsdaten bieten hier viel Potenzial für weitere Experimente und studentische Forschung.

- **Vertiefung von Visualisierung und Auswertung:** Die bereits umgesetzten Dashboards und Analysewerkzeuge können in kommenden Arbeiten noch ausgebaut und benutzerfreundlicher gestaltet werden.

> Eine verständliche und transparente Darstellung der Ergebnisse fördert das Verständnis – nicht nur für uns, sondern auch für andere Studierende.

Das Projekt hat uns gezeigt, wie wichtig eine interdisziplinäre Herangehensweise aus Informatik, Simulation und methodischer Auswertung für die Lösungsentwicklung ist. Für die weitere persönliche und wissenschaftliche Entwicklung bieten die erarbeiteten Grundlagen vielfältige Ansatzpunkte zur Vertiefung — sowohl im Rahmen von Abschlussarbeiten als auch im Team oder in praktischen Workshops.

Literaturverzeichnis

[Aut25] AutomotiveIT. Waymo erreicht mit Robotaxis den nächsten Meilenstein. https://www.automotiveit.eu/technology/autonomes-fahren/so-will-volkswagen-es-mit-waymo-und-tesla-aufnehmen-435.html, Apr. 2025. Unter Verwendung von Material der dpa. Zuletzt abgerufen am 09.09.2025.

[fLuRD24a] D. Z. für Luft- und Raumfahrt (DLR). Car-Following-Models. https://sumo.dlr.de/docs/Car-Following-Models.html, Oct. 2024. Zuletzt abgerufen am 23.09.2025.

[fLuRD24b] D. Z. für Luft- und Raumfahrt (DLR). Open Street Map. https://sumo.dlr.de/docs/Networks/Import/OpenStreetMap.html, Oct. 2024. Zuletzt abgerufen am 23.09.2025.

[fLuRD25a] D. Z. für Luft- und Raumfahrt (DLR). Definition of Vehicles, Vehicle Types, and Routes. https://sumo.dlr.de/docs/Definition_of_Vehicles%2C_Vehicle_Types%2C_and_Routes.html, July 2025. Zuletzt abgerufen am 23.09.2025.

[fLuRD25b] D. Z. für Luft- und Raumfahrt (DLR). SUMO at a Glance. https://sumo.dlr.de/docs/SUMO_at_a_Glance.html#usage_examples, Mar. 2025. Zuletzt abgerufen am 23.09.2025.

[fLuRD25c] D. Z. für Luft- und Raumfahrt (DLR). TraCI. https://sumo.dlr.de/docs/TraCI.html, Apr. 2025. Zuletzt abgerufen am 23.09.2025.

[Gre25] J. Greenyer. Integrationsprojekt - Intelligentes Testen autonomer Fahrzeuge, 2025. Einführungspräsentation zum Integrationsprojekt.

[HMW10] D. Harel, A. Marron, and G. Weiss. Programming coordinated behavior in Java. In *ECOOP 2010 – Object-Oriented*

Programming, volume 6183 of *Lecture Notes in Computer Science*, pages 250–274. Springer, 2010. doi:10.1007/978-3-642-14107-2_12.

[HMW12] D. Harel, A. Marron, and G. Weiss. Behavioral programming. *Communications of the ACM*, 55(7):90–100, 2012. doi:10.1145/2209249.2209270.

[Hou24] L. Houben. Wie China Robotaxis zum Erfolg verhilft. `https://www.zdfheute.de/politik/ausland/china-autonomes-fahren-robotaxi-100.html`, July 2024. Zuletzt abgerufen am 09.09.2025.

[LBBW+18] P. A. Lopez, M. Behrisch, L. Bieker-Walz, J. Erdmann, Y.-P. Flötteröd, R. Hilbrich, L. Lücken, J. Rummel, P. Wagner, and E. Wießner. Microscopic traffic simulation using sumo. In *The 21st IEEE International Conference on Intelligent Transportation Systems*. IEEE, 2018. URL `https://elib.dlr.de/124092/`.

[LDZ+22] G. Lou, Y. Deng, X. Zheng, M. Zhang, and T. Zhang. Testing of autonomous driving systems: where are we and where should we go? In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 31–43, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3540250.3549111.

[Leu18] E. Leurent. An environment for autonomous driving decision-making. `https://github.com/eleurent/highway-env`, 2018.

[LQW24] R. Li, T. Qin, and C. Widdershoven. Iss-scenario: Scenario-based testing in carla, 2024, 2406.15777. URL `https://arxiv.org/abs/2406.15777`.

[PCA+21] A. Piazzoni, J. Cherian, M. Azhar, J. Y. Yap, J. L. W. Shung, and R. Vijay. ViSTA: a Framework for Virtual Scenario-based Testing of Autonomous Vehicles . In *2021 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 143–150, Los Alamitos, CA, USA, Aug. 2021. IEEE Computer Society. doi:10.1109/AITEST52744.2021.00035.

[Tie25] Y. Tiedemann. So will Volkswagen es mit Waymo und Tesla aufnehmen. `https://www.automotiveit.eu/technology/autonomes-fahren/so-will-volkswagen-es-mit-waymo-und-tesla-aufnehmen-435.html`, June 2025. Zuletzt abgerufen am 09.09.2025.

[TWY+23] H. Tian, G. Wu, J. Yan, Y. Jiang, J. Wei, W. Chen, S. Li, and D. Ye. Generating critical test scenarios for autonomous driving systems via influential behavior patterns. pages 1–12, 01 2023. doi:10.1145/3551349.3560430.