HAW HAMBURG

BACHELORTHESIS
Daniel Osterholz

Implementierung einer ereignisbasierten Routenanpassung in MARS mithilfe von simulierten IoT-Sensordaten

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Daniel Osterholz

Implementierung einer ereignisbasierten Routenanpassung in MARS mithilfe von simulierten IoT-Sensordaten

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas Clemen
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 07. Januar 2022

Daniel Osterholz

Thema der Arbeit

Implementierung einer ereignisbasierten Routenanpassung in MARS mithilfe von simulierten IoT-Sensordaten

Stichworte

IoT, Smart City, Digitaler Zwilling

Kurzzusammenfassung

Die Arbeit befasst sich mit der Frage, ob es möglich ist, durch die Nutzung von IoT-Sensordaten eine Verminderung des innerstädtischen Straßenverkehrs zu erreichen. Diese Simulation ist georeferenziert. Im geografischen Gebiet im Zentrum von Hamburg (Deutschland) wird eine multimodale Verkehrssimulation implementiert. Parkräume werden innerhalb des Modells mit IoT-Sensoren ausgestattet. Agenten erhalten über diese Sensoren Ereignisse und müssen diese dann nutzen, um gegebenenfalls eine Routenanpassung durchzuführen, sollte dies der Verkürzung der Fahrzeit dienen. Mithilfe von geeigneten Szenarien wird anschließend eine Aussage über die Wirksamkeit von IoT-Sensoren im Parkraumsuchverkehr getroffen.

Daniel Osterholz

Title of Thesis

Implementation of event-based route adaptation in MARS using simulated IoT sensor data

Keywords

IoT, Smart City, Digital twin

Abstract

The work addresses the question of whether it is possible to achieve a reduction in inner-city road traffic by using IoT sensor data. This simulation is georeferenced. A

multimodal traffic simulation is implemented in the geographical area in the centre of Hamburg (Germany). Parking spaces are equipped with IoT sensors within the model. Agents receive events via these sensors and must then use them to adjust the route if necessary, should this serve to shorten the journey time. With the help of suitable scenarios, a statement is then made about the effectiveness of IoT sensors in parking space search traffic.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Es findet eine Urbanisierung statt. Die Menschen ziehen immer häufiger aus ländlichen Gegenden in die Metropolen der jeweiligen Länder. Auf dem ganzen Erdball drängen die Menschen vom Land in die Städte. Diese gewaltige Wanderung hat dramatische Veränderungen der globalen Siedlungslandschaft zur Folge.[17] Durch diese Zuwanderung der Menschen in die großen, ohnehin schon dicht besiedelten, Städte verschärft sich das Problem der Mobilität innerhalb dieser Metropolen. Durch die Bereitstellung von neuen Angeboten im Bereich des öffentlichen Nahverkehrs kann der Verkehrslast, welche durch die zugewanderten Menschen entsteht, nur bedingt begegnet werden. Eine Privatperson in den Industriestaaten hat in der Regel einen eigenen Personenkraftwagen (Pkw).„Wohlstand ohne Auto ist in unserer Welt der Gegenwart schlicht und ergreifend ausgeschlossen. [...] Bisher lebten wir in einer Welt, in der überwiegend der persönliche Besitz der vier Räder die Branche geprägt hat“[6] Diese Fahrzeuge sind bei der Stadtplanung mit einzubeziehen. Der Platzaufwand der immer größer werdenden Autos und die immer höheren Zulassungszahlen sind Gründe, um sich näher mit dem Parkraummanagement zu befassen.

Statistisch betrachtet wird ein Fahrzeug innerhalb von 24 Stunden nur 40 Minuten bewegt, während es den Rest der Zeit geparkt werden muss [10]. Die These dieser Arbeit ist, dass die Nutzung von Sensordaten deutlich messbaren Einfluss auf die gesamte Reisedauer in urbanem Gebieten hat. Außerdem sollte der Parkraumsuchverkehr und die hierdurch hohe Auslastung des Straßennetzes deutsch gesenkt oder entfernt werden können.

1.2 Ziel der Arbeit

Das SmartOpenHamburg-Modell wurde um eine ereignisbasierte Routenanpassung ergänzt. Das MARS-Framework ist in der Lage eventbasierte Ereignisse jeglicher Art zu behandeln. Das implementierte Modell kann die These beantworten, ob durch die Nutzung von IoT-Sensordaten im Bereich von Parkräumen eine Verkürzung der Reisedauer und die Entlastung der innerstädtischen Straßenkapazitäten erreicht werden kann.

1.3 Abgrenzung der Arbeit

Ein Modell, welches alle Einflussfaktoren innerhalb einer innerstädtischen Fahrt betrachtet ist, hochkomplex. Der Umfang dieser Arbeit ist hierfür nicht ausreichend. Es bedarf weiterer Simulationen und einer sehr genauen Datenlage, um eine exakte Aussage hierfür zu treffen. Weiter ist die Nutzung nur eines geografischen Gebietes nicht maßgebend für alle anderen denkbaren Orte. Die Wahl der Stadt Hamburg in Deutschland unterliegt außerdem bestimmten Gesetzgebungen, welche auch nicht repräsentativ für andere Länder sein muss.

1.4 Aufbau der Arbeit

Es wird eine genaue Analyse der geforderten Anforderungen unternommen. Innerhalb dieser Anforderungsanalyse werden Szenarien definiert, welche von dem zu implementierenden Modell erfüllt werden müssen. Sollten im Laufe der Bearbeitung der These Konzepte oder Technologien angesprochen oder verwendet werden, werden diese im Abschnitt Grundlagen erläutert. Auf diese Weise soll sichergestellt werden, dass alle unternommenen Schritte nachvollziehbar für den Leser bleiben. Sind die Anforderungen an das Modell klar, wird im nachfolgenden Abschnitt auf theoretischer Ebene das Modell entworfen. Hierfür werden Entwurfsprinzipien verwendet, welche für die Aufgabenerfüllung am sinnvollsten erscheinen. Alle Konzepte werden erläutert und gegebenenfalls begründet. Ist die theoretische Grundlage für die Implementierung geschaffen, werden diese dann in praktischer Form programmiert. Sollten im Laufe der Implementierung Änderungen an den Konzepten vorgenommen werden müssen, so wird dies in dem jeweiligen Abschnitt genau begründet. Anschließend werden Experimente durchgeführt und beschrieben und

im Schlussteil der Arbeit sowohl die Ergebnisse der Experimente diskutiert, als auch eine gesamtheitliche Betrachtung und Bewertung der Arbeit vorgenommen.

2 Grundlagen

In diesem Abschnitt werden alle notwendigen Grundlagen der verschiedenen in dieser Arbeit angesprochenen Themenbereiche erörtert und spezifiziert. Diese Erläuterungen sind Voraussetzung zum vollumfänglichen Verständnis der in dieser Arbeit verwendeten Technologien und Algorithmen.

2.1 Geoinformationssystem (GIS)

GIS ist ein System, das alle Arten von Daten erstellt, verwaltet, analysiert und kartiert. GIS verbindet Daten mit einer Karte und integriert reale Standortdaten mit allen Arten von beschreibenden Informationen. Auf diese Weise entsteht eine Grundlage für die Kartierung und Analyse, die in der Wissenschaft und in anderen Bereichen aktiv eingesetzt werden. Benutzer von GIS können Muster, Beziehungen und geografische Zusammenhänge extrahieren und für verschiedene Fragestellungen nutzbar machen. Die Effizienz und Verwaltung innerhalb dieser Daten wird mithilfe von hierarchisch strukturierten Features umgesetzt. Diese Features können grundsätzlich alle Arten von Informationen repräsentieren.

2.2 Notwendige GIS-Dateiformate

GIS-Formate lassen sich in 3 Formate gliedern, *vektor*-, *raster*- und *temporale* Formate. Diese Formate sind in der Lage dem **GIS** Daten in einer verarbeitbaren Art und Weise abzuspeichern bzw. bereitzustellen. Sie folgen immer einer wohldefinierten Form. Es gibt eine Vielzahl verschiedener Dateiformate und viele Unternehmen nutzen eine eigene Darstellung. Das bedeutet für ein GIS ist es notwendig verschiedene Formate ineinander überführen zu können. Es werden drei Vektorformate erläutert, welche in der Implementierung innerhalb des MARS-Frameworks genutzt werden.

- Geojson: Sehr weit verbreitet und ein de facto Standard im Bereich der GI-Systeme. Es ist wohl definiert und kann von vielen GIS verarbeitet werden. Im Rahmen dieser Arbeit ist die Verarbeitung dieses Datentyps in **QGIS** notwendig. Das Format basiert auf dem Json-Format und definiert die Art und Weise, in der geografische Merkmale und deren Eigenschaften vorliegen.[5]
- Geopkg: Ein GeoPackage ist ein plattformunabhängiger SQLite-Container und enthält neben Metadaten bestimmte Definitionen, Integritätsansprüche, Formatbeschränkungen und inhaltliche Einschränkungen. Der zulässige Inhalt eines GeoPackage ist vollständig im Open Geospatial Consortium (OGC) definiert, kann aber durch eine API vom Implementierer beliebig erweitert werden.[14]
- Graphml: Ein weniger weitverbreitetes Dateiformat, welches auf dem XML-Format basiert. Es wird beschrieben durch Knoten und Kanten. Diese Graphen können alle Einstellungen enthalten, welche Graphen besitzen (Bsp. gerichtet, ungerichtet, Attribute oä.). Es besitzt keine eigene Syntax und bedient sich in Gänze an der XML-spezifischen Syntax.[4]

2.3 OpenStreetMap (OSM)

OSM wurde 2004 mit dem Ziel geschaffen eine freie Weltkarte zu erstellen. Diese Weltkarte sollte kostenlos allen Menschen zur Verfügung stehen, welche mit GIS arbeiten. Die Erhebung dieser Daten geschieht nicht nur von Domänenexperten, sondern auch von Domänenfremden (z.B. unter Nutzung von GPS-Daten).[16] Diese Art der Datenerhebung bringt gewisse Ungenauigkeiten mit sich. Vorteil dieser Daten ist aber die hohe Anzahl von Menschen, die bestehende Informationen überprüfen. Auf diese Weise sind gerade in urbanen Gebieten sehr genau Karten entstanden.

2.4 Koordinatenbezugssystem (KBS)

Ein KBS ist ein Koordinatensystem, welches durch die Nutzung von Datum und Koordinaten eine Möglichkeit bietet einen Punkt auf der Erde zu beschreiben. Die genaue Definition wird in der ISO-Norm 19111 CRS beschrieben. [11] Es gibt verschiedene Koordinatenreferenzsysteme. In diesem Rahmen werden European Petroleum Survey Group

(EPSG)-Codes genutzt, um sicherzustellen, dass bei der Angabe von solchen Koordinaten dieselben Positionen gemeint sind (das gleiche Messverfahren genutzt wurde). [1]

2.5 Multiagentensystem

Multi-Agent-Systeme (MAS) sind ein leistungsfähiges verteiltes Computermodell, das es Agenten ermöglicht, miteinander zu kooperieren und sich zu ergänzen und sowohl semantische Inhalte als auch einen semantischen Kontext auszutauschen, um diese Inhalte automatisch und genau zu interpretieren.[15] Grundlegende Informationen, welche im Rahmen der Multiagentenmodellierung angesprochen werden müssen sind die Folgenden.

Kommunikationssprache

Eine Kommunikationssprache ist ein Protokoll, auf das sich innerhalb eines Systems geeinigt wird. Diese Einigung ist notwendig für eine möglichst effiziente Kommunikation zwischen Agenten (zwei oder mehreren) miteinander oder zwischen Agenten und dem System (der Umgebung). Es besteht aus folgenden notwendigen Bestandteilen und kann durch beliebig viele weitere Bestandteile erweitert werden (abhängig von den jeweiligen Anforderungen).

- Es gibt eine Menge von Regeln.
- Eine Nachricht hat eine klare Syntax.
- Es gibt eine wohl definierte Semantik.
- Die Synchronisationen innerhalb des Systems ist klaren Regeln unterworfen.

Agent Communication Language (ACL)

Für eine Kommunikation zwischen Agenten ist eine Sprache wichtig, welche für alle Individuen dasselbe bedeutet. Der Kontext, in dem eine Nachricht gesendet bzw. empfangen wird, ist ein weiterer wichtiger Parameter für die Definition einer nutzbaren ACL. Dieser Kontext beschreibt, mit welcher Art von Nachrichten (abstrakter auch Zuständen) ein

Agent umgehen kann und wie die austauschenden Individuen (in diesem Fall kann ein Individuum auch das System an sich sein) in Beziehung stehen.

Foundation for Intelligent Physical Agents (FIPA)

FIPA ist ein Gremium zur Entwicklung und Festlegung von Standards für Computersoftware für heterogene und interagierende Agenten und agentenbasierte Systeme.

2.6 MQTT

MQTT beschreibt ein Pushbasiertes Protokoll. Die wichtigsten Merkmale von MQTT ist seine Leichtigkeit und Bandbreiteneffizienz. Es ermöglicht IoT-Sensoren die Bi-direktionale Kommunikation und bietet eine sehr gute Skalierung von IoT-Geräten. Zusätzlich werden verschiedene Narchrichtenübermittlungstypen definiert.

- 0: Eine Nachricht wird höchstens einmal gesendet.
- 1: Eine Nachricht wird mindestens einmal gesendet.
- 2: Eine Nachricht wird genau einmal gesendet.

Die Unterteilung dieser Qualitätsstufen sollten alle IoT-Anwendungsfälle abdecken. Die genaue Spezifikation dem OASIS Standard.[12]

3 Anforderungsanalyse

Alle an die Simulation gestellten Anforderungen werden hier zusammengefasst. Eine Überprüfung des Simulationsgebietes wird durchgeführt. Durch eine Klassifizierung in funktionale und nichtfunktionale Anforderungen soll eine bestmögliche und dedizierte Betrachtung der einzelnen Anforderungen gewährleistet werden. Zusätzlich zu dieser Trennung wurde eine weitere Ebene der Analyse hinzugefügt, welche die Vorgaben aus der Sicht des Entwicklers aufzeigen. Auf diese Weise wird eine umfassende und vollständige Betrachtung der Anforderungen durchgeführt.

3.1 Beschreibung der Umgebung

Die Anforderung an das zu betrachtete Simulationsgebiet sind durch nachfolgende Punkte erläutert.

Die Simulation soll in einem **urbanen Bereich** stattfinden. Die Untersuchung der Einflussnahme von IoT-Sensordaten auf den Parkraumsuchverkehr und die damit direkt verbundene Einflussnahme auf die Fahrzeit stellt die Stadtplanung in dicht besiedelten Gebieten vor immense Herausforderungen. Um diese Herausforderungen mit der in dieser Arbeit beschriebenen Simulation abbilden zu können, muss das zu simulierende Gebiet geografisch ebenfalls im urbanen Bereich liegen.

Eine **realistische Auslastung des vorhandenen Verkehrsnetzes** muss gegeben sein. Außerdem müssen **Daten über die vorhandene Infrastruktur** vorliegen. Besondere Aufmerksamkeit gilt den vollständigen und korrekten Daten im Bereich der Parkräume. Die durch diese Daten garantierte Realitätsnähe kann für spätere Validierungen der gewonnenen Ergebnisse genutzt werden.

3.2 Anforderung an die Simulation

Um die Anforderungen an das zu implementierende Multiagentensystem (MAS) präzise formulieren zu können, werden verschiedene Anwendungsfälle aus unterschiedlichen Sichten beschrieben. Diese Anwendungsfälle stellen Situationen dar, welche im urbanen Straßenverkehr üblich sind. Außerdem werden Szenarien geschrieben, um die Beziehungen zwischen den Anwendungsfällen aufzuzeigen. Auf Grundlage dieser Anwendungsfallbeschreibungen können sowohl Testszenarien erarbeitet werden, als auch Validierungen stattfinden. Die Validierungen werden möglich durch einen Abgleich zwischen den Simulationsdaten und den Daten aus der Realität.

Das Ziel der Simulation soll sein, die Frage zu klären, ob es durch die Nutzung von IoT-Sensordaten möglich ist, eine messbare Verbesserung der Fahrzeit zu erreichen. Die verwendeten GIS-Daten und die für die Simulation genutzten Daten nähern die Simulation an die Realität an. Abstraktionen, welche für die Durchführbarkeit der Simulation nötig sind, werden auf ein erforderliches Mindestmaß eingeschränkt. Diese Abstraktionen werden unter dem Vorbehalt gemacht, dass sie die erhobenen Daten nicht oder nur minimal beeinflussen. Diese Beeinflussung sollten sich in einem zu vernachlässigen Bereich befinden und die Gesamtaussage der Daten nicht beeinträchtigen.

Im Rahmen dieser Anforderungen muss es möglich sein, dass ein Agent seine Route anpasst. Ein Parkplatz muss einen Sensor besitzen. Auf Nachrichten dieses Sensors muss dann vom Agenten reagiert werden können.

3.2.1 Fachliche Anforderungen

Tabelle 3.1: Agent geht zum Auto

Name	M01
Beschreibung	Ein Agent geht von seiner Startposition zu seinem Auto.
Akteure	Agent
Auslöser	Der Agent will von seiner Startposition zu einem Ziel.
Vorbedingungen	Keine
Nachbedingungen	Der Agent befindet sich im Auto
Standardablauf	1. Der Agent ermittelt die Position seines Autos. 2. Der Agent geht zu Fuß zu seinem Auto. 3. Der Agent steigt in sein Auto ein und wechselt somit die Modalität.

Tabelle 3.2: Agent fährt mit dem Auto

Name	M02
Beschreibung	Ein Agent fährt von seiner Position zu einem Ziel.
Akteure	Agent
Auslöser	Der Agent will von seiner Position mit dem Auto zu einem Ziel.
Vorbedingungen	Der Agent befindet sich in einem Auto.
Nachbedingungen	Der Agent hat das Auto in der Nähe der Zielposition geparkt.
Standardablauf	1. Der Agent wählt ein Zielort aus. 2. Der Agent sucht nach dem kürzesten Weg zu dem Parkplatz, der dem Ziel am nächsten ist. 3. Der Agent fährt mit dem Auto auf der ermittelten Route zu dem Parkplatz.

Tabelle 3.3: Agent geht vom Parkplatz zu dem endgültigen Ziel

Name	M03
Beschreibung	Ein Agent geht von seinem Auto zu seinem endgültigen Ziel.
Akteure	Agent
Auslöser	Der Agent muss zu Fuß gehen, um sein endgültiges Ziel zu erreichen.
Vorbedingungen	Der Agent hat das Auto geparkt und ist noch nicht am endgültigen Ziel angekommen.
Nachbedingungen	Der Agent erreicht das Ziel.
Standardablauf	1. Der Agent verlässt sein Auto. 2. Der Agent geht Fuß zur Zielposition.

Tabelle 3.4: Ein ParkingEvent tritt ein

Name	M04
Beschreibung	Ein ParkingEvent tritt ein
Akteure	Agent
Auslöser	Ein Parkplatz wird besetzt
Vorbedingungen	Parkplatz ist frei
Nachbedingungen	Parkplatz ist besetzt
Standardablauf	1. Der Parkplatz, den der Agent ausgewählt hat, wird besetzt. 2. Der Sensor des Parkplatzes registriert den neuen Zustand. 3. Der Parkplatz sendet den neuen Zustand an den EventHandler.

Tabelle 3.5: Agent muss Route anpassen

Name	M05
Beschreibung	Ein Agent muss seine Route anpassen.
Akteure	Agent
Auslöser	Der Parkplatz, zu dem der Agent unterwegs ist, ist nun belegt.
Vorbedingungen	Der Agent fährt im Auto zu einem Parkplatz.
Nachbedingungen	Der Agent hat eine neue Route zu einem anderen verfügbaren Parkplatz.
Standardablauf	1. Das Ereignis „Parkplatz belegt“ wird dem Agenten mitgeteilt. 2. Der Agent erfragt eine neue Route zu dem Parkplatz, welcher seinem Ziel am nächsten ist und noch frei ist. 3. Die Route des Agenten wird angepasst. 4. Die neue Route wird vom Agenten gefahren.

3.2.2 Anforderungen aus Sicht der Entwickler

Tabelle 3.6: Sammeln der GIS-Daten

Name	E01
Beschreibung	GIS-Daten werden mit der gewünschten geografischen Ausdehnung aus einer öffentlichen Quelle bezogen.
Akteure	Entwickler
Auslöser	Die erforderlichen Daten für die Simulation liegen nicht vor.
Vorbedingungen	Keine
Nachbedingungen	Geografische Informationen über das zu simulierende Gebiet liegen in Form von GIS-Daten bereit.
Standardablauf	1. Finden von öffentlich zu Verfügung stehenden GIS-Datenbank. 2. Auswahl eines geeigneten Datensatzes für die Simulation. 3. Herunterladen des richtigen Datensatzes.

Tabelle 3.7: GIS-Daten Vorbereitung

Name	E02
Beschreibung	Zuschneiden der gesammelten GIS-Daten und eventuelle Bereinigungen durchführen.
Akteure	Entwickler
Auslöser	GIS-Daten für den gewünschten geografischen Bereich liegen dem Entwickler lokal vor.
Vorbedingungen	GIS-Daten für ein gewünschtes geografisches Gebiet liegen vor.
Nachbedingungen	GIS-Daten sind für das Einladen in das MARS-Framework vorbereitet.
Standardablauf	1. Die GIS-Daten werden in QGIS eingeladen. 2. Die eingeladenen Daten werden auf Vollständigkeit überprüft. 3. Die eingeladenen Daten werden auf Plausibilität überprüft. 4. Die nun auf Vollständigkeit und Plausibilität überprüften Daten werden als Geojson-Datei exportiert.

Tabelle 3.8: GIS-Daten Einladung in das Framework

Name	E03
Beschreibung	Die bereinigten und geprüften GIS-Daten werden in das MARS-Framework eingeladen und somit für eine Nutzung innerhalb des Modells bereitgestellt.
Akteure	Entwickler
Auslöser	Bereinigte und geprüfte GIS-Daten für den zu simulierenden Bereich stehen dem Entwickler lokal zur Verfügung.
Vorbedingungen	GIS-Daten sind für das Einladen in das MARS-Framwork vorbereitet.
Nachbedingungen	Alle erforderlichen geografischen Informationen sind innerhalb des MARS-Framworks bereitgestellt.
Standardablauf	1. Es wird eine Konfiguration für ein Modell innerhalb des MARS-Frameworks erstellt. 2. Innerhalb dieser Konfiguration wird dem Framework mitgeteilt, wo die GIS-Daten abgelegt sind. 3. Innerhalb dieser Konfiguration wird dem Framework mitgeteilt, auf welche Layer die Daten abgebildet werden sollen.

3.2.3 Qualitätsanforderungen

Für die Bestimmung der Güte einer Software sind nicht nur die fachlichen Anforderungen zu erfüllen. Vielmehr ist es notwendig, die weichen, nicht direkt an die Funktionalität der Software gebundenen Anforderungen klar zu umfassen. Optimalerweise sollen alle Anforderungen erfüllt werden. Das ist in der Realität aber nicht unbedingt möglich. Es muss dementsprechend eine Quantifizierung stattfinden, um die wichtigsten Qualitätsanforde-

rungen zu identifizieren und auszuwählen. Für diese Quantifizierung ist es notwendig, den genauen Anwendungsbereich und Zweck der zu schreibenden Software zu kennen. In dieser Arbeit wird sich auf Qualitätsanforderungen gemäß ISO 25010 bezogen.

Zeitverhalten Zur Simulation von geografisch weit ausgedehnten Gebieten ist es gegebenenfalls notwendig, eine Vielzahl von Agenten zu simulieren. Beispielsweise werden zur Simulation eines urbanen Gebietes sehr schnell sehr hohe Anzahlen an Agenten benötigt. Dies ist wichtig zur Vergleichbarkeit der Simulation zu seinem realen Pendant.

Modularer Aufbau Die Entwicklung in modularer Form wird für eine spätere Weiterverwendung benötigt. Mit dem MARS-Framework können unterschiedliche Fragestellungen beantwortet werden. Es ist sinnvoll und gewünscht, etwaige Entwicklungen auch anderen Entwicklern zur Verfügung zu stehen.

Wiederverwendbare Komponenten Wie beschrieben, sollen die hier implementierten und beschriebenen Entwicklungen auch zu einem späteren Zeitpunkt weiter zur Verfügung stehen. Auf diese Weise können zukünftige Fragestellungen ihren Fokus auf andere komplexe Fragestellungen richten.

Gute Analyse-Funktion Diese Arbeit wird nach wissenschaftlichen Standards durchgeführt. Es werden Messungen, Prognosen und Validierungen vorgenommen. Hierfür muss sichergestellt werden, dass Messungen richtig, zuverlässig und einfach durchführbar sind.

4 Simulations- und Modellumgebung

Für die Realisierung der Fragestellung soll ein MAS genutzt werden. Es gibt eine Vielzahl an Gründen für diese Entscheidung. Nachfolgend wird erläutert, wieso ein MAS sinnvoll ist. Es wurde sich vorab für das MARS-Framework (Multi-Agent Research and Simulation-Framework) entschieden; diese Entscheidung wird begründet. Für die Einschätzung des MAS wird auf die vorab erläuterten Anforderungen zurückgegriffen.

Die agentenbasierte Modellierung stammt aus dem Bereich der verteilten künstlichen Intelligenz. Dieses Simulationsparadigma umfasst Individuen (dargestellt als Agenten), die miteinander und mit ihrer Umgebung interagieren. Das Verhalten wird individuell so programmiert, dass es bestimmten Regeln folgt. Durch die Verlagerung der Komplexität von der Umgebung hin zu den einzelnen Agenten entsteht ein komplexes Verhalten des Gesamtsystems. Diese Eigenschaft macht die Multiagentenmodellierung besonders geeignet für die Modellierung von Systemen wie Städten.

Für die Beantwortung der hier gestellten Forschungsfrage sind nachfolgende Punkte von Relevanz:

- Frei wählbare und definierbare Agenten
- Auswahl von frei wählbaren Umgebungen (beliebiger geografischer Ausdehnung)
- Verarbeitung und Darstellung von GIS-Daten
- Performanz und Skalierbarkeit der Simulation
- Erweiterbarkeit des bestehenden MAS durch eigene Entwicklungen
- Komplexität und Verständlichkeit

4.1 MARS und Architektur

Das MARS-Framework wird vorgestellt und anschließend auf die vorab gestellten Anforderungen geprüft.

4.1.1 Agenten

Einer der zwei Hautkomponenten dieses Frameworks sind die Agenten. Um diese sinnvoll modellieren zu können, ist eine genaue Definition und die Vorstellung der Fähigkeiten der Agenten notwendig. Wie sollen diese mit ihrer Umwelt interagieren können, wie mit den anderen Agenten? Sie werden als ein Informationssystem mit drei Phasen betrachtet.

1. Zunächst gibt es eine Menge von Eingabedaten, die in das Modell integriert werden, entweder als Layer oder als Agent.
2. Die gewünschte Interaktion bzw. das gewünschte Verhalten wird simuliert.
3. Die gesammelten Informationen werden in eine gewünschte Ausgabeform gebracht und dem Entwickler zur Verfügung gestellt.

Eine genauere Betrachtung des Systems zeigt, dass die Eingabedateien über eine Simulationskonfiguration beschrieben werden. Die Eingabedaten werden verwendet, um bestimmte Eigenschaften des Agenten initial zu beschreiben. Während der Modellausführung werden die definierten Agenten in jedem Simulationsschritt (beschrieben in der Simulationskonfigurationsdatei) „an getickt“. Die Tick-Methode beinhaltet die gesamte Logik des jeweiligen Agenten. Die Ausgabeformate sowie die Filterkriterien (z.B. zur Festlegung, welche Attribute in die Ausgabe geschrieben werden sollen) müssen in der Simulationskonfiguration definiert werden.

4.1.2 Layer

Layer stellen die Umgebung dar, in der die Agenten leben. Sie bieten Eingabemöglichkeiten für verschiedene Datenformate. Die Layer sind das zweite zentrale Element in einem MARS-Modell und werden vom MARS-Laufzeitsystem automatisch generiert. Sie stellen den Agenten Informationen verschiedenster Art zur Verfügung und können vom Modellierer frei definiert werden. Auch können die Agenten die verschiedenen Layer nicht nur lesend (wahrnehmend), sondern auch schreibend (aktiv interagierend) nutzen. Auf diese

Weise ist es einem einzelnen Agenten auch möglich, Informationen global für alle anderen Agenten bereitzustellen. Bei einer Interaktion zwischen einzelnen Agenten ist dies nicht möglich.

4.1.3 Entitäten

Im Gegensatz zu Agenten sind Entitäten reine Objekte ohne eine Tick-Methode. Entitäten werden jedoch auch durch eine ID identifiziert und können in das Modell aufgenommen werden, um andere Daten einzugeben, die nicht direkt Teil der Agentenparametrisierung sind. Ein Beispiel für eine Entität ist das Auto eines Agenten. Diese Entität ist dem Agenten zugeordnet und für diesen nutzbar.

4.1.4 Environments

Die Umgebung kann verwendet werden, um Agenten und andere Entitäten im Raum zu positionieren und abzufragen. Sie hilft dabei, ein räumliches (ortsbezogenes) Objekt in die Simulation einzufügen und es nach den vorgegebenen Filterkriterien abzufragen.

4.1.5 Architektur

Eine komplette Darstellung der Architektur des MARS-Framworks erscheint in diesem Rahmen nicht sinnvoll, da es darum geht, wie das Framework für die in dieser Arbeit behandelte Fragestellung geeignet ist. Es folgt ein Diagramm, welches die Teile des Frameworks veranschaulicht, welche für die Arbeit relevant sind.

4.1.6 Erfüllung der Anforderungen

Agenten

Wie im Abschnitt 4.1.1 beschrieben, können Agenten völlig frei und mit allen benötigten Funktionalitäten realisiert werden. Grenzen in der Logik eines Agenten sind für alle Anwendungsfälle nicht gegeben. Selbst bei einer möglichen Fortsetzung und Vertiefung der Fragestellung können die Agenten dahingehend komplementiert werden.

Umgebung

Für die Umsetzung der Arbeit wird eine Möglichkeit benötigt, welche einen beliebig großen existierenden geografischen Raum zur Verfügung stellt. MARS bietet die Möglichkeit, genau diese Umgebung unkompliziert in Form von frei wählbaren Layern in das System einzubinden. Die Daten können auf unterschiedliche Layer gemappt werden und bieten so eine dedizierte Zugriffsmöglichkeit der Informationen für die Agenten an.

GIS Verarbeitung

Wie im obigen Absatz beschrieben, ist eine Einladung von variablen Umgebungen (mit geografischer Zuordnung) einfach umsetzbar. Die Daten werden in üblichen Dateiformaten akzeptiert (geojson-, shp- oder asc-Dateien). Durch die Nutzung von diesen Dateiformaten können viele Datengrundlagen (öffentlich zugängliche und freie Datenbanken) in MARS nutzbar gemacht werden.

Performanz und Skalierbarkeit

Die Skalierbarkeit und die Performanz wird im Rahmen der Anforderungen betrachtet. Für diese Arbeit bedeutet das, dass ein Raum abbildbar sein muss, welcher groß genug ist, um aussagekräftige und vergleichbare Ergebnisse zu liefern. Beispielsweise muss das Zentrum einer Stadt abbildbar sein. Diese Anforderungen bestimmen maßgeblich den Rahmen der Skalierbarkeit und die benötigte Performanz des Systems. Das MARS-Framework bietet in diesem Bereich einen ausreichenden Rahmen. Für deutlich größere Anwendungsfälle kann es durch die zentralisierte Berechnung von MARS zu deutlichen Einbußen kommen.

Erweiterbarkeit

Die quelloffene Art dieses Frameworks ermöglicht dem Programmierer eine komplett frei wählbare Erweiterung eines Modells. Die Anpassung des Frameworks kann gegebenenfalls auch vorgenommen werden, falls notwendig. Grundlegende Strukturen wie Agenten und Layer sind vorgegeben. Alle anderen Komponenten dieses Frameworks werden dem Programmierer überlassen und auch die Agenten und Layer können in Bezug auf enthaltene

Informationen, Struktur und Integrationsmöglichkeiten nach dem Willen des Anwenders angepasst werden.

Verständlichkeit

Die Verständlichkeit ist durch eine existierende und aktuelle Dokumentation gegeben. Bei Unklarheiten sind die Entwickler des Frameworks immer kurzfristig erreichbar und sehr hilfsbereit. Es stehen Vorlesungen und Tutorials zur Verfügung, um praxisbasierte Probleme, welche nicht sofort verstanden wurden, zu erläutern. Durch diese verschiedenen Ebenen der Unterstützung ist eine sehr gute Verständlichkeit des Systems und eine klare Vorstellung der Bearbeitung innerhalb des Frameworks gegeben.

4.2 Modellumgebung - SmartOpenHamburg

SmartOpenHamburg ist ein Projekt der MARS Gruppe (https://mars-group.org). Die Forschungsgruppe beschäftigt sich seit vielen Jahren mit großen komplexen Systemen. Das Modell beschreibt ein Modell, welches über verschiedene Arbeiten weiter entwickelt wurde. Es umfasst eine Vielzahl an Klassen und Komponenten. Es ist ein generisches, agentenbasiertes multimodales Modell. Ursprünglich als Automodell implementiert und im Bereich um Hamburg-Altona simuliert.[19] Es wurde um die Möglichkeit erweitert, dass Agenten sich multimodal fortbewegen können.[13] Dies umfasst das zu Fuß laufen, das Autofahren und das Fahrradfahren. Verkehrsregeln, wie sie in Deutschland gegeben sind, sind ebenfalls vorhanden. Es gibt die links vor rechts Regeln, Fahrspuren und Geschwindigkeitsbegrenzungen. Auf Ampelschaltungen und deren genauen Einfluss auf den Verkehr wurde verzichtet. Unter Multimodalität wird die Verfügbarkeit und geeignete Wahl von unterschiedlichen Verkehrsmitteln verstanden. Es bietet sich an, um die hier gestellt Frage zu untersuchen. Die Erweiterung dieses Modells um die eventbasierte Routenanpassung kann dem Modell weitere Möglichkeiten bieten, den innerstädtischen Verkehr noch besser darzustellen und zu untersuchen.

4.3 Beurteilung der betrachteten Umgebungen

Die Beurteilung muss nach nachvollziehbaren Kriterien erfolgen. Auch auf Grundlage der vorher angesprochenen Anforderungsanalyse (sowohl fachliche wie auch nicht funktionale Anforderungen).

5 Konzeption

Im Laufe der Anforderungsanalyse sind viele Anforderungen angesprochen und bewertet worden. Um all diese angesprochenen, im Besonderen die nichtfunktionalen, Anforderungen erfüllen zu können, muss ein geeignetes Konzept erarbeitet werden. Im Speziellen ist es notwendig, eine angemessene Architektur zu wählen und gegebenenfalls bestehende Architekturen mit Blick auf die gegebene Situation abzuändern, zu erweitern und/oder zu kombinieren.

Der Rahmen dieser Arbeit erlaubt die logische Teilung der verschiedenen Vorgaben. Dies entspricht auch dem Ansatz der Trennung von Aufgaben (Seperation of Concerns (SoC)). Die Konzepte unterscheiden sich je nach Teilaufgabe und zu erfüllender Anforderung. Um innerhalb dieses Abschnittes eine sinnvolle Gliederung herzustellen, werden im Folgenden die Konzepte je Teilaufgabengebiet beschrieben und begründet.

5.1 Erweiterung des Frameworks

Es wird ein bestehendes Framework erweitert. Dieses Framework enthält diverse Features, welche zur Implementierung der in dieser Arbeit benötigten Funktionalitäten verwendet werden.

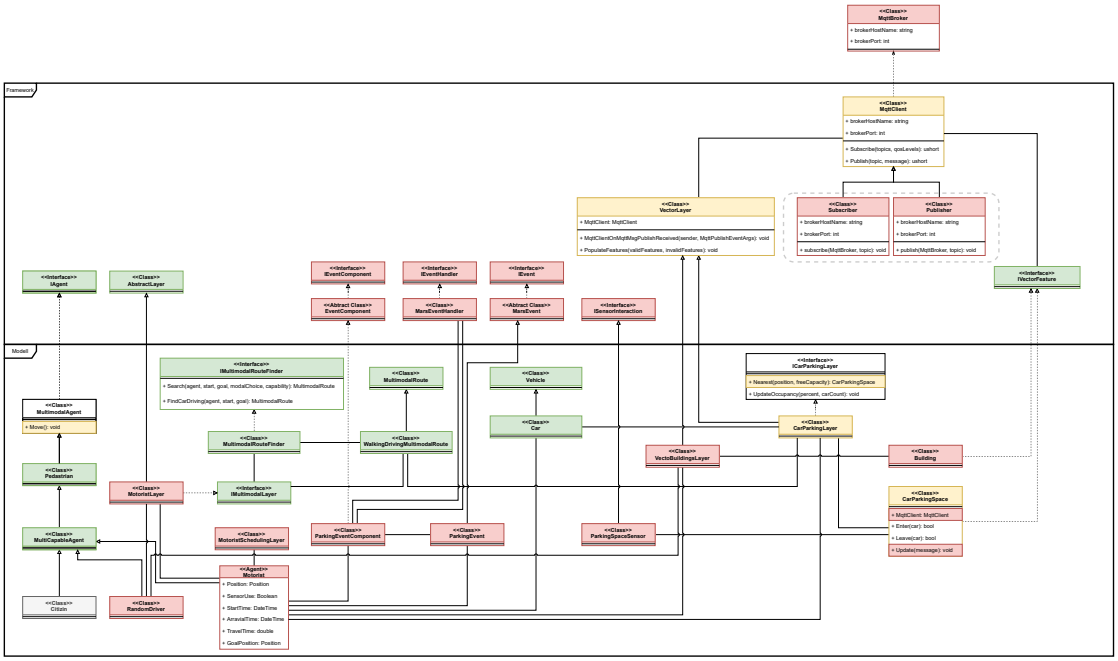

Abbildung 5.1: Klassendiagramm Ausschnitt MARS-Framework

Auf dem Diagramm ist ein Ausschnitt des MARS-Frameworks zu sehen. Das Diagramm ist in zwei Teile unterteilt. Im oberen Teil sind Komponenten abgebildet, welche zum Framework gehören. Im unteren Teil des Diagramms sind modellspezifische Komponenten abgebildet. Diese Trennung wurde gemacht, um auf klare und verständliche Weise die genaue Aufteilung zwischen Framework und Modell darzustellen. Da dieses Framework aktiv weiterentwickelt wird, ist es notwendig, sich darüber klar zu sein, welche Klassen nur in enger Zusammenarbeit mit den für die Weiterentwicklung des Frameworks zuständigen Programmierer umgesetzt werden können und welche Klassen zu dem Modell gehören. Diese Klassen können selbständig entwickelt werden und benötigen keine Änderungen des verwendeten Frameworks. Die farbliche Kennzeichnung gibt an, welche Klassen schon vorhanden sind, welche verändert werden müssen und welche komplett neu implementiert werden müssen, um das Ziel dieser Arbeit zu erreichen.

- Grüne Komponenten: Diese Klassen bestehen bereits und müssen nicht verändert werden. Sie und bestimmte Implementierungen innerhalb dieser Klassen werden für die Umsetzung dieses Projektes genutzt.
- Gelbe Komponenten: Diese Klassen sind bereits vorhanden, müssen aber ergänzt oder verändert werden (auf die Änderungen wird nachfolgend eingegangen).

- Rote Klassen: Diese Klassen sind noch nicht implementiert. Sie werden für die Durchführung dieses Projektes neu implementiert.

Klassen, die in Zusammenhang mit den simulierten IoT-Sensordaten stehen, sind im Diagramm oben rechts dargestellt. Im Laufe der Arbeit an diesem Projekt wurde der Fokus auf die ereignisbasierte Routenanpassung gelegt. Die Implementierung eines vollständigen IoT-Sensornetzes ist hierfür nicht notwendig. Agenten nutzen eine vereinfachte Darstellung der Sensordaten. Diese wird in dem Abschnitt 5.3 genau beschrieben. Der Broker ist weder im Framework noch in dem Modell abgebildet. Er steht außerhalb der Architektur, da er eine externe Komponente ist. Innerhalb dieses Brokers werden Sensordaten gesammelt und verwaltet (bspw. wie lange werden Daten gehalten und wie ist die Policy der Verbreitung an die Subscriber). Siehe hierfür auch siehe 2.6. Die Klassen **Subscriber** und **Publisher** werden auf andere Weise in dem hier beschrieben Modell berücksichtigt. Subscriber sind alle implementierten Agenten, welche über Änderungen des von ihnen angefahrenen Parkplatzes informiert werden wollen. Diese Funktionalität wird über einen Mechanismus realisiert, welcher im Abschnitt 5.4 näher beschrieben wird. Die grundsätzliche Idee ist es, dass sich Agenten (wenn gewollt) für eine bestimmte Art Event registrieren können. Diese Komponente kann alles Mögliche sein und ist unter dem Gesichtspunkt der Anforderungsanalyse und den nichtfunktionalen Anforderungen an das Modell einfach zu erweitern und auf alle erdenkliche Probleme erweiterbar (siehe Abschnitt 4.1.6).

5.2 Modellbeschreibung

Das Modell wird aus verschiedenen Layern bestehen. Ein Layer beschreibt die Umwelt, mit welcher ein Agent (aktiver Akteur) interagieren wird, um seine gegebenen Aufgaben zu erfüllen. Ein Layer beschreibt einen Teil der Umgebung und besteht aus einer Menge von Features. Die einzelnen Layer wiederum sind logisch voneinander getrennt. Die Trennung der Abhängigkeiten kann auf diese Weise umgesetzt werden.

Agenten sind der aktive Teil der Simulation und kommunizieren/interagieren mit ihrer Umwelt (den verschiedenen Layern). Durch Abfrage ihrer Umwelt sind sie in der Lage, komplexe Aufgaben zu lösen. Es ist ihnen außerdem möglich, die Layer zu beschreiben, um anderen Agenten über diese Schnittstelle Informationen mitzuteilen. Diese können dann wiederum die aktualisierten Informationen erfragen.

Akteure

Ein Akteur ist der aktive Teil der Simulation. Er stellt den Agenten dar. Jeder Agent/Akteur besitzt eine Routine, die er in jedem Zeitschritt ausführt. Innerhalb dieser Methode (Tick-Methode genannt) werden alle logischen Schritte unternommen, welche zur Erfüllung der Aufgabe beitragen. Bei Start der Simulation wird jedem Agenten ein Start- und Zielort zugewiesen. Eine initiale Multimodalroute wird außerdem zugewiesen. Nun beginnt die Ausführung der Tick-Methode. Beispielsweise sitzt ein Agent bereits in seinem Auto und muss für einen Zeitschritt nun mit dem Auto eine bestimmte Strecke in Richtung seines Zielortes zurücklegen. Diese Routine wird dann in der Tick-Methode stehen und während der Programmausführung abgearbeitet werden (für diesen einen Zeitschritt). Bei der nächsten Ausführung würde dann entschieden werden, was nun im darauf folgenden Zeitschritt zu passieren hat und dementsprechend agiert werden.

Umwelt

Die Umwelt wird durch getrennte Schichten dargestellt. Ein Layer (Schicht) hat immer genau eine Zuständigkeit. In den nachfolgenden zwei Klassendiagrammen sind die für diese Arbeit notwendigen Layer genau aufgeführt. Das erste Diagramm zeigt Layer, mit denen interagiert wird (nachfolgend Ressourcenlayer genannt) und das zweite Diagramm zeigt Layer, auf denen sich ein Agent fortbewegen kann (Networklayer).

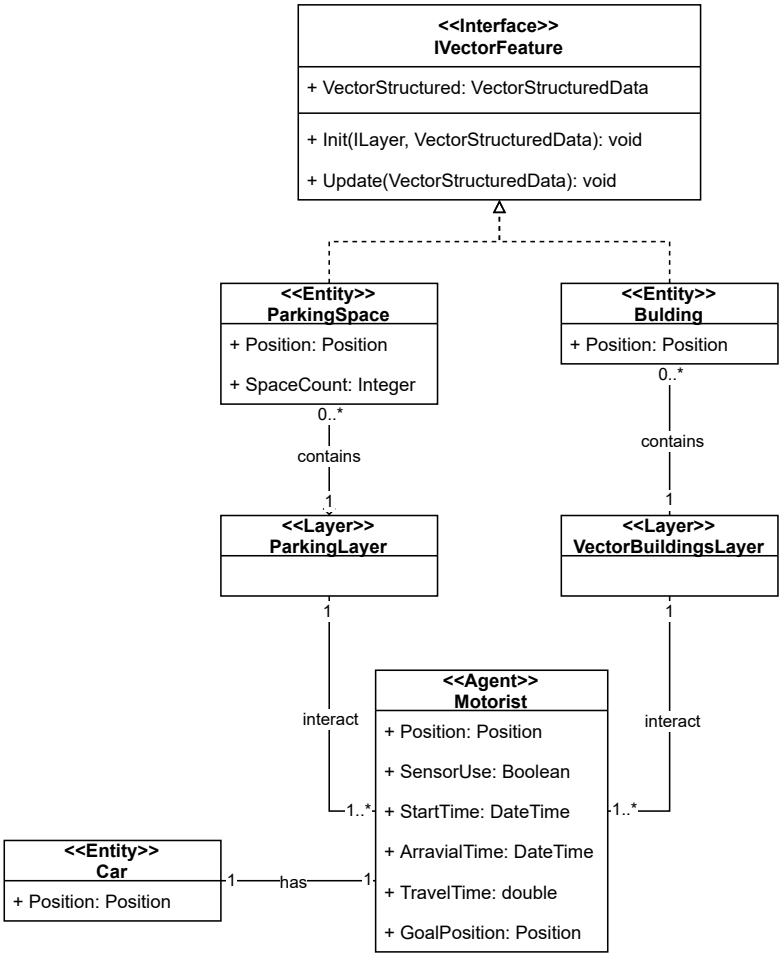

Abbildung 5.2: Klassendiagramm SoC durch Layer (Interaktion)

Ein Motorist (aktiver Agent) hat die Möglichkeit, mit einem ParkingLayer zu interagieren, um Informationen über ParkingSpaces zu erhalten. Diese Informationen beinhalten sowohl die Position (latitude und longitude), als auch die Information über seinen Wert (SpaceCount). Ein ParkingSpace ist ein in das MARS-Framework eingeladener Vektorlayer (Multipolygon-Layer) im Geojson-Format. Dieser Layer besteht aus einer Menge von Polygonen, welche Parkraum darstellen. Da ein Polygon sowohl ein einzelner Parkplatz, wie auch eine Parkfläche sein können, wird beim Einladen in das Modell eine Berechnung durchgeführt. Innerhalb dieser Rechnung wird die Fläche eines jeden Polygons hergenommen und überprüft, ob diese einem Einzelparkplatz entspricht. Falls nicht, wird berechnet, wie viele Autos auf dieser Fläche Platz finden müssten und diese er-

rechnete Zahl in dem Attribut SpaceCount abgespeichert (bei einem einfachen Parkplatz beträgt der SpaceCount 1).

Der VectorBuldingsLayer hält eine Anzahl an Building. Diese werden wie die Parkplätze auch als Vektorlayer (Multipolygon-Layer) im Geojson-Format dem System mitgegeben. Die Gebäude können eine Vielzahl an Informationen enthalten. In dieser Datei stehen u.a. Informationen über die Art des Gebäudes (Mehrfamilienhaus, Praxis, etc.). Da Gebäude in der Simulation genutzt werden, um einen Start- und Zielort zu erhalten, ist lediglich ihre Position notwendig und keine weiteren Informationen. Bei Multiagentensimulationen, welche mit realen GIS-Daten arbeiten, ist es wichtig, die Daten so weit wie möglich einzuschränken. GIS-Daten werden schnell schwer verarbeitbar, da sie häufig eine Vielzahl an (für die jeweilige Fragestellung) überflüssigen Informationen enthalten. Die Selektion dieser Features kann für eine deutliche Steigerung der Performanz sorgen.

Wie bei den beiden beschriebenen Layern dargestellt, werden Polygone aus den Geojson-Dateien in das Modell geladen. Die Attribute der Polygone werden über das Vector-Structured Attribut abgebildet. Auf diese Art können alle notwendigen und gewünschten Informationen aus den GIS-Daten in das Modell überführt werden.

Im Diagramm 5.2 wurde dargestellt, wie ein Agent, wenn notwendig und gewünscht, mit seiner Umwelt direkt interagieren kann. Im nachfolgenden Diagramm werden zwei weitere Aufgaben der Layer aufgezeigt.

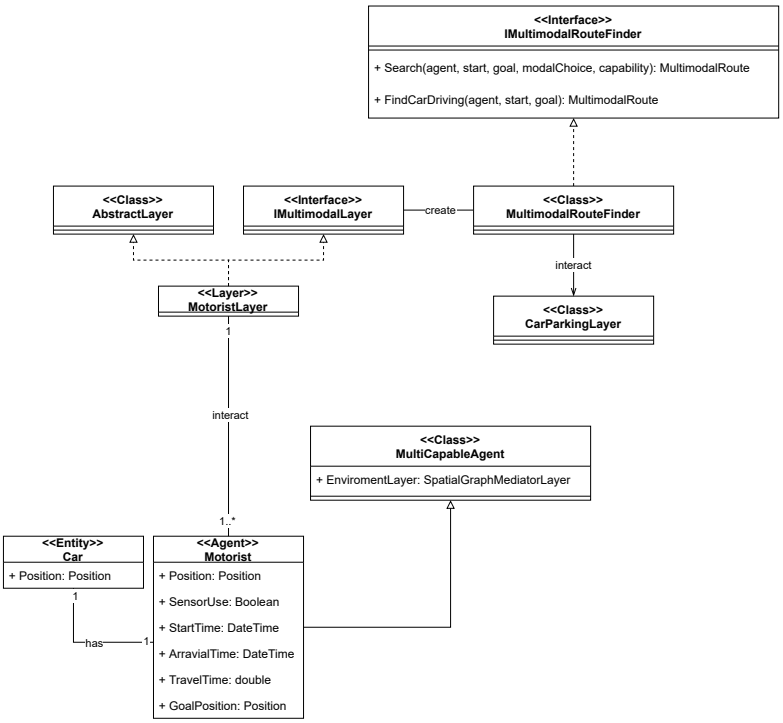

Abbildung 5.3: Klassendiagramm SoC durch Layer (Environment zur Bewegung der Agenten)

Zum einen wird gezeigt, wie mithilfe des IMultimodalLayers eine MultimodalRoute erstellt werden kann. Diese multimodale Route kann in dieser Simulation folgende Ausprägungen haben:

1. WalkingMultimodalRoute: Eine Route, die nur zu Fuß gegangen wird. Es werden keine anderen Modalitäten auf keiner der Teilstrecken verwendet. Diese Route wird auf dem eingeladenen Graphen ausgeführt, welcher nur für Fußgänger verwendet wird. Dieser Graph ist deutlich dichter als beispielsweise der Graph für Fahrradfahrer und Autofahrer. Die höhere Dichte dieses Graphen resultiert aus dem Fakt, dass viele Wege nur zu Fuß zugänglich sind.

2. WalkingDrivingMultimodalRoute: Eine Route, die Teilstrecken beinhaltet, welche zu Fuß und mit dem Auto bewältigt werden. Eine Teilstrecke kann beispielsweise der Weg vom Startort bis zum Parkplatz (auf welchem sich der Wagen des Agenten befindet) sein. Diese Strecke wird zu Fuß bewältigt. In diesem Beispiel wird anschließend die nächste Teilstrecke mit dem Auto gefahren werden. Die Reihenfolge ist nicht vorgegeben und muss auch nicht ausgewogen sein (zwischen zu fahrenden und

zu gehenden Teilstrecken). Diese Route sagt nur aus, dass diese beiden Modalitäten enthalten sind.

Mithilfe der Search-Funktion ermittelt das Framework die beste (kürzeste) Strecke vom Start zum Ziel, unter Berücksichtigung der für den Agenten verfügbaren Modalitäten. Diese werden durch das Attribut Capabilities definiert. In dieser Simulation ist es jedem Agenten möglich, zu Fuß zu gehen und mit seinem einem eigenen Auto zu fahren. Durch die Generalisierung-Spezialisierung (MultiCapableAgent - Motorist) ist es dem Motorist möglich, auf den SpatialGraphMediatorLayer zuzugreifen. Dieser beinhaltet alle über die config.json eingeladenen Environments (siehe Abschnitt 6.5). Ein Environment ist ein Graph, auf welchem sich ein Agent (mit einer bestimmten Modalität) bewegen kann. Wenn sich beispielsweise ein Agent zu einem Zeitschritt in seinem Auto befindet, so ist es ihm möglich, sich auf dem Graphen zu bewegen, welcher in der InputConfiguration die modality “CarDriving zugeordnet bekommen hat.

Interaktion innerhalb des Modells

Wie in den vorausgegangenen Abschnitten beschrieben, findet die Interaktion primär über die verwendeten Layer statt. Die Interaktion zwischen Agenten ist zwar ein elementares Konzept innerhalb von MAS, aber in diesem Anwendungsszenario vernachlässigbar. Die Vernachlässigung dieser Art der Interaktion zwischen Agenten liegt in dem Umstand begründet, dass sie im realen Straßenverkehr auch nur sehr bedingt (wenn überhaupt) stattfindet. Da diese Arbeit versucht, eine möglichst genaue Abbildung der Realität in Bezug auf Parkraumfindung herzustellen, müssen die hier implementierten Agenten nicht direkt miteinander kommunizieren. Diese Tatsache kann an einem einfachen Beispiel verdeutlicht werden. Im normalen Straßenverkehr wird ein Mensch immer mithilfe seiner Umwelt eine Entscheidung treffen. Diese Umwelt teilt ihm mit, ob er z.B. grün hat und fahren kann oder ob, wenn er fahren will, ein Auto seinen Weg kreuzt. In den seltensten Fällen ist ein Autofahrer darauf angewiesen, mit anderen Fahrern aktiv zu kommunizieren, um eine Entscheidung für sein Handeln zu treffen. Eine Ausnahme wäre der Spezialfall, dass an einer Kreuzung, an der die Links-vor-Rechtsregel zum Tragen kommt, alle Wege von mindestens einem Auto belegt sind. In diesem Fall muss ein Fahrer den Anfang machen und seine Vorfahrt aufgeben. Dieser Fall wird in dieser Simulation nicht eintreten. Eine Kommunikation zwischen Agenten ist also weder notwendig noch realitätsnah.

5.3 IoT-Sensordaten

Wo kommen die Daten her und wie sollen sie in das System integriert werden. den Unterschied erklären zwischen abgeschlossener Simulation und Nutzung von real daten.

Es gibt verschiedene Möglichkeiten eine Simulation mit Sensordaten interagieren zu lassen. Zwei grundlegend unterschiedliche Herangehensweise werden in diesem Abschnitt erläutert und begründet, für welche der Möglichkeiten sich entschieden wurde. Die unterschiedlichen Herangehensweisen können wie folgt eingeordnet werden:

- Eine Simulation ist während der gesamten Laufzeit in sich geschlossen und bekommt keine neuen Informationen der echten Welt hineingegeben (geschlossene Simulation).
- Eine Simulation bekommt während der Laufzeit neue Informationen des realen Gegenstücks und arbeitet sie in die laufende Simulation ein (digitaler Zwilling).

Beide Möglichkeiten haben bestimmte Vor- und Nachteile. Das Prinzip, nach dem IoT-Sensoren ihre Informationen verbreiten, kann aber in beiden Varianten auf die gleiche Art behandelt werden. Im nachfolgenden Designdiagramm ist das Prinzip der Verarbeitung von Sensordaten aufgezeigt. Es unterscheidet sich für die geschlossene Simulation und den digitalen Zwilling nur einem Punkt. Die Parking Spots und die dazugehörigen Sensoren sind bei einem digitalen Zwilling wie dargestellt in der realen Welt und bei einer geschlossenen Simulation innerhalb des Systems (simulierte IoT-Sensordaten).

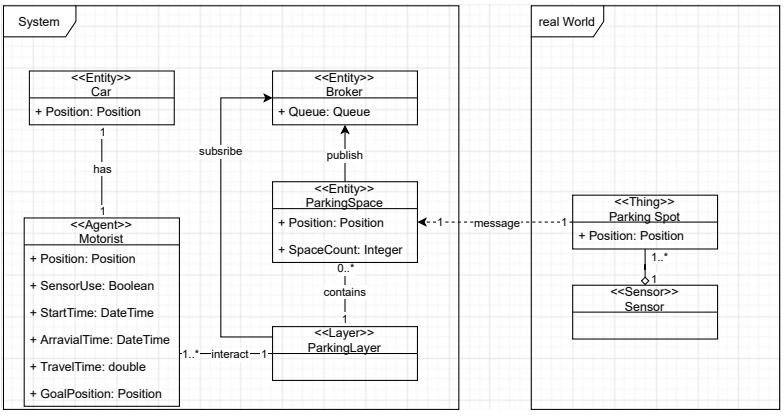

Abbildung 5.4: Designdiagramm Verarbeitung von IoT-Sensordaten

Geschlossene Simulation Das obige Diagramm 5.4 ist in diesem Rahmen ein wenig anders zu konzipieren. ParkingSpaces initialisieren direkt einen Sensor bei Erstellung. Dieser Sensor ist initial frei. Sollte nun das Auto eines Agenten den Parkplatz besetzen wird durch dieses Ereignis ein ParkingEvent getriggert (siehe Use Case 3.4). Dieses Event wird an einen Broker (in diesem Fall den MarsEventHandler) geschickt. Dieser Handler muss nun entscheiden, wem der dieses Event weiterleiten muss. Für diese Entscheidung haben sich zuvor Agenten über eine ParkingEventComponent bei dem Handler für diese Art von Events registriert.

Digitaler Zwilling Die Sensordaten kommen aus dem realen Zwilling der Simulation. Sie werden in Echtzeit übertragen und wirken sich dann direkt auf den Ablauf der Simulation aus. Bei dieser Art der Simulation gilt es, die realen Daten sinnvoll der Simulation hinzuzufügen. Beispielsweise ist innerhalb der Simulation ein Parkplatz belegt. Dieser belegte Parkplatzsensor bekommt nun aber von seinem realen Gegenstück die Meldung, dass er frei ist. Nun muss der Entwickler sich entscheiden, wie er verfahren will. Soll das simulierte Auto, welches auf besagtem Parkplatz steht, entfernt, versetzt oder bestehen bleiben? Eine ständige genaue Anpassung an die Realität führt dazu, dass die Simulation sehr nah an der Realität bleib. Es ist aber nicht möglich, die Simulation weit in die Zukunft laufen zu lassen, um Prognosen über das zukünftige Verhalten der realen Welt zu machen. Die nicht Anpassung der Simulation sorgt aber dafür, dass sich Fehler akkumulieren und die erhaltenen Daten ggf. nicht mehr präzise genug sind für eine aussagekräftige Prognose der Entwicklung. Ein Abwägen, wie oft reale Daten in die Simulation integriert werden sollen, muss getroffen werden.[18][9] Auf diese Weise kann eine Abstraktion der realen Welt und der aktuellen Situation getroffen werden, durch die eine ungefähre Prognose der Zukunft gegeben werden kann.

Von den angesprochenen Unterschieden abgesehen, verdeutlicht das Diagramm die Verteilung und das Konzept der IoT-Sensordaten. Jeder Agent besitzt ein Auto. Die Agenten interagieren, um einen freien Parkplatz zu finden, ausschließlich mit dem ParkingLayer. Der ParkingLayer enthält ParkingSpaces – diese existieren auch in der realen Welt und sind in die Simulation eingeladen worden. Ändert sich der Zustand eines Parkplatzes reagiert der Sensor, welcher zu dem jeweiligen Parkplatz gehört (der Zustand kann entweder Occupied gleich True oder False sein). Bei jeder Änderung des Zustandes teilt der Sensor dem Broker diese mit. Der Broker überprüft, ob es Agenten gibt, welche sich für diese Änderung interessieren (ein Agent oder mehrere fahren diesen Parkplatz an). Die

Änderung des Zustandes wird den Agenten mitgeteilt. Jeder Agent kann nun entscheiden, wie er auf das Ereignis reagieren will. Bei einem digitalen Zwilling kommt noch die Synchronisation der Realdaten hinzu.

Für diese Arbeit wird eine geschlossene Simulation implementiert. Der Fokus würde sich bei einem digitalen Zwilling stark in Richtung der angemessenen Synchronisationsmechanismen und -strategien verschieben. Die ereignisbasierte Routenanpassung würde so in den Hintergrund gerückt werden. Eine der Qualitätsanforerungen ist die einfache Erweiterbarkeit des Modells. Die implementierten Komponenten können dann auch für ein Modell genutzt werden, welches diese Art der Synchronisation im Fokus hat (siehe Abschnitte 3.2.3 und 5.4).

5.4 Erweiterbarkeit des Modells

Um eine der wichtigen Qualitätsanforderungen erfüllen zu können, werden Events (IoT-Sensor) und die Eventkomponenten (Verarbeitung der gegebenen Events) in dem Modell wie folgt dargestellt.

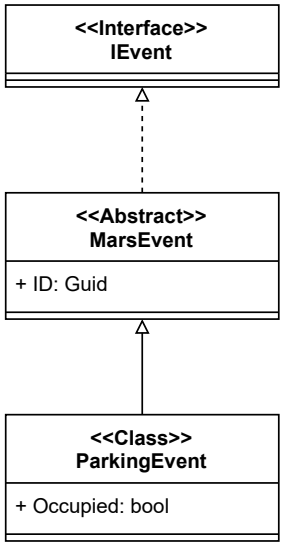

Abbildung 5.5: Klassendiagramm Erweiterbarkeit der Events

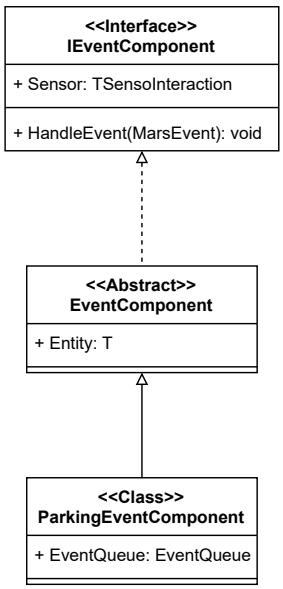

Abbildung 5.6: Klassendiagramm Erweiterung der Komponenten

Ein Event implementiert das Interface IEvent und erbt vom MarsEvent, welches vom EventHandler (dem Broker der IoT-Sensordaten) verarbeitet werden kann. Durch dieses Konzept (siehe interfaces hier 5.1) ist das Modell nicht darauf beschränkt nur ParkingEvents verarbeiten zu können. Event sind auf diese Weise völlig frei implementierbar. Vorgegeben ist lediglich das Vorhandensein einer eindeutigen ID (gegeben durch die Generalisierung-Spezialisierung zum MarsEvent). In dieser Arbeit werden ParkingEvents benötigt. Ein ParkingEvent hat den Zustand besetzt oder nicht besetzt. Eine Position ist nicht notwendig, da innerhalb der Simulation mit der ID des jeweiligen Parkplatzes gearbeitet wird. Ein Beispiel für die leichte Erweiterbarkeit ist ein RainingEvent. Angenommen ein Agent möchte über Regen an einer bestimmten Position informiert werden, und zwar nur dann, wenn er nicht im Auto sitzt. Um dieses Szenario aufzubauen, benötigt man nun nur drei Erweiterungen. In diesem Beispiel wird angenommen, dass Agenten nur an Regen innerhalb bestimmter Regionen interessiert sind. Eine Region kann ein Stadtteil sein.

1. Es werden Events erstellt, welche die hierarchische Aufteilung meiner MarseEvents definiert. Es kann z.B. eine AltonaRainingEvent Klasse erstellt werden. Hier werden Events zusammengefasst, welche geografisch in dem Stadtteil Hamburg-Altona stattfinden.

2. Regen-Sensoren innerhalb dieses Stadtteils publishen nun AltonaRainingEvents (z.B. bei anfangendem Regen).

3. Nun fehlt nur noch die RainingComponent, diese erbt von der EventComponent. Agenten können sich für verschiedene Komponenten anmelden. Die Komponente registriert sich selbständig bei dem Broker (für das gewünschte Event) und verarbeitet diese dann für ihn. Die Komponente würde überprüfen, ob der Agent in einem Auto sitzt und dementsprechend agieren.

Der Broker (MarsEventHandler) ist in der Lage alle Arten von Eine weitere Ausbaustufe dieses Scenarios ist die Unterteilung in verschiedene geografische Gebiete. Mit dem hier aufgezeigten Konzept kann eine frei wählbare hierarchische Struktur unter den MarsEvents erstellen werden. Ein Agent kann sich beispielsweise auf HamburgRainingEvent registrieren, AltonaRainingEvents usw. . Der MarsEventHandler würde den Agenten dann nur über die relevanten Events informieren. Das Konzept des MarsEventHandlers ist nachfolgend erläutert.

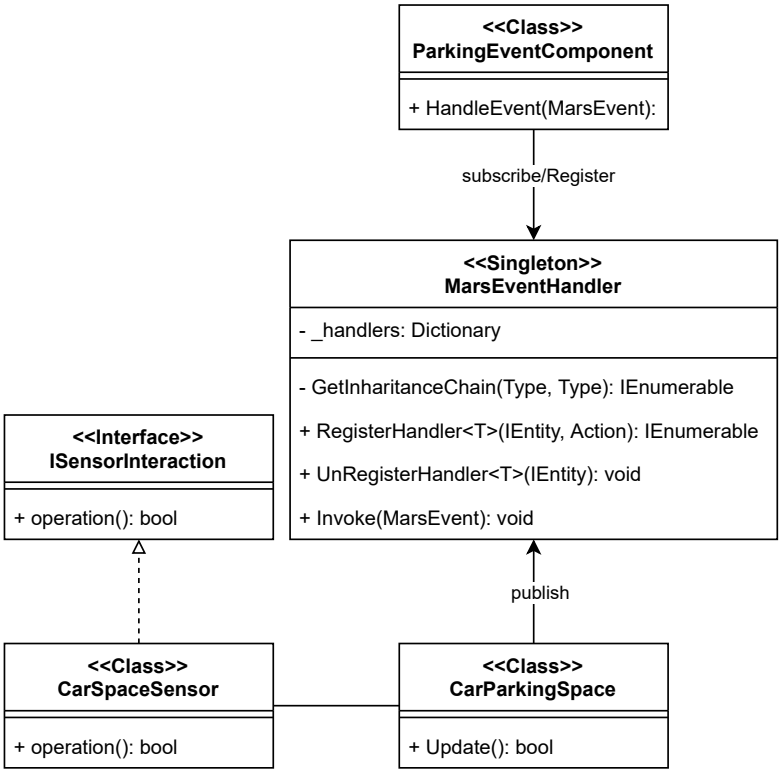

Abbildung 5.7: Klassendiagramm MarsEventHandler

Um sicherzustellen, dass es nicht mehr als einen EventHandler in der Simulation gibt (ein einziger ist in der Lage alle Events zu bearbeiten) wird diese Klasse als Singleton geplant.[7] Kern dieser Komponente ist die richtige Verarbeitung und Verbreitung von Events, an registrierte Agenten. Eine Komponente registriert einen Agenten bei dem Handler. Er kann alle Arten von MarsEvents verarbeiten. Die Registrierung bei dem Handler erfordert immer auch die Art des Events, bei dem sich registriert werden soll. Ist das Event noch unbekannt, wird ein neuer Eintrag erstellt und der Agent als Subscriber hinzugefügt. Bei einem UnRegister wird ein Agent als Subscriber (falls eingetragen) entfernt. Beim Auslösen eines Events finden nun folgende Schritte innerhalb des Handlers statt:

- Es wird überprüft, ob das Event Kindklassen besitzt.
- Es wird jedes Event an die Agenten verteilt, welche gleich dem eingehenden Event sind oder Kindklassen von diesem.

Die Logik welche ausgeführt werden soll bei eingehendem Event ist innerhalb der Event-Component (HandleEvent(MarsEvent)-Methode) geregelt. Durch die GetInharitanceChain-Methode können unkompliziert komplexe Strukturen und Sachverhalte abgedeckt werden. Sollte ein Agent für mehrere Events registriert sein (innerhalb einer Vererbungshierarchie), so wird ihm die spezialisierteste Variante des Events mitgeteilt. Diese Logik basiert auf der Annahme, dass ein sehr spezielles Event einem generellen Event vorzuziehen ist (es beinhaltet genauere Informationen ohne einen Informationsverlust). Die Methode wird ausnahmslos im Kontext der Invoke-Methode innerhalb des MarsEvent-Handlers ausgeführt.

Methoden des MarsEventHandlers

Es wird noch einmal genau auf die Konzeption innerhalb der wichtigsten Funktionen des MarsEventHandlers eingegangen. UnRegister ist nicht näher beschrieben, da diese Methode lediglich nach der zu entfernenden Entität sucht und diese aus dem `_handlers` Dictonary entfernt.

Abbildung 5.8: Aktivitätsdiagramm RegisterHandler

Eine ParkingEventComponent registriert seinen Agenten über diese Methode bei dem MarsEventHandler für das gewünschte Event. Es wird überprüft, ob das gewünschte Event bereits vorhanden ist. Ist es dem MarsEventHandler bereits bekannt, wird ein

Dupel mit der zu registrierenden Entität und der auszuführenden Methode (Systemintern wird die Speicheradresse der Methode übergeben) in der Liste hinzugefügt unter dem Key des Events (Beispielsweise Key = ParkingEvent). Ist das Event noch unbekannt, wird dem Dictionary ein Key mit dem neuen Eventtypen erstellt und als Value eine neue Liste mit einem Dupel hinzugefügt. Das Dupel entspricht demselben wie bei schon vorhandenem Event. Sollte der Agent bereits registriert sein, wird ein Fehler geschmissen, da davon auszugehen ist, dass der Implementierer einen logischen Fehler gemacht hat.

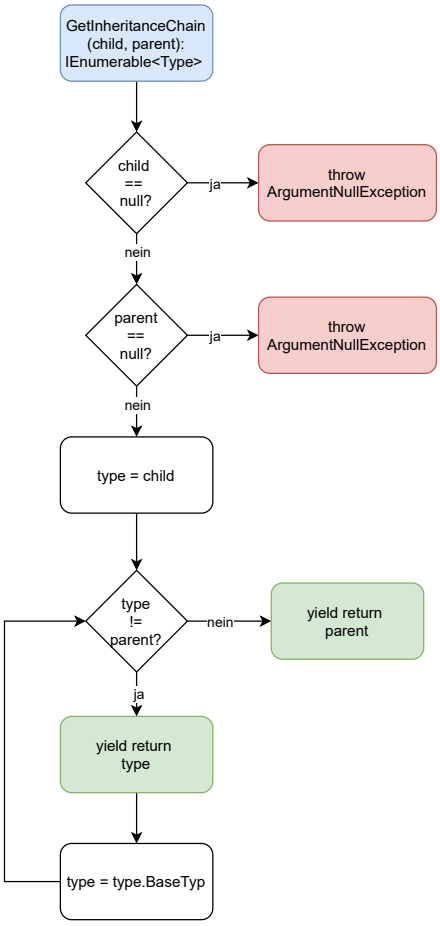

Abbildung 5.9: Aktivitätsdiagramm GetInharitanceChain

Diese Methode wird nur im Rahmen der Invoke-Methode genutzt und stellt sicher, dass auch alle hierarchisch übergeordneten Events weiter gegeben werden. Es findet zuerst eine Überprüfung statt, um sicherzustellen, dass weder der erste noch der zweite Parameter null sind. Nun wird der zurückgebende Parameter *type* mit dem *child* belegt. Dieser wird

zurückgegeben. Beim nächsten Aufruf wird durch das *yield*-Keyword der Rückgabewert auf den Basistypen von child gesetzt. Ist nun das child gleich dem parent so wird dieser Zurückgegeben und der Aufruf, welcher nur von invoke genutzt wird, ist beendet. Bei der nächsten Ausführung der Methode (ein neues Event wurde ausgelöst) startet die Methode wieder von vorne.

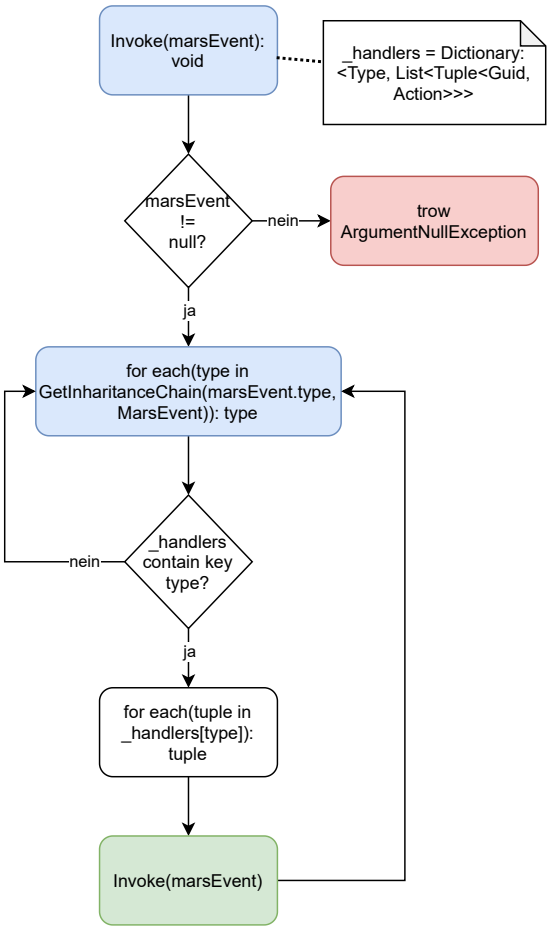

Abbildung 5.10: Aktivitätsdiagramm Invoke

Invoke wird von Sensoren ausgelöst. die Auslösung der Methode bewirkt die Verteilung der Events an alle registrierten EventComponents. Zuerst wird geprüft, ob das auszulösende Event null ist. Anschließend wird für alle MarsEvents und deren Elternklassen die C# Funktion *invoke* ausgeführt. Ist die Basisklasse MarsEvent erreicht, ist die Ausführung der Invoke-Methode und die damit verbundene Verteilung der Events (alle registrierten Methoden wurden ausgeführt) beendet.

5.4.1 Pluginarchitektur

Die Plugin-Architektur ist ein Entwurfsmuster im Bereich der Softwareentwicklung. [8] Mit der Hilfe dieser Architektur ist eine klare Teilung der Aufgabengebiete möglich. Einzelne Komponenten des Systems werden gekapselt und so austauschbar gemacht. Sie interagieren über klar definierte Schnittstellen mit dem Rest des Systems. Diese Schnittstellen müssen wohl definiert sein, um dem System zu ermöglichen, die unterschiedlichen Implementierungen innerhalb der Pluginkomponente richtig zu interpretieren. Dieses Verhaltensmuster unterstützt die Softwarearchitekturprinzipien der losen Kopplung und hohen Kohäsion. Die hohe Kohäsion ist gegeben durch die fachliche Aufteilung der Aufgaben (z.B. innerhalb einer Pluginkomponente). Durch die Nutzung der Schnittstellen wird die dahinterliegende Implementierung verborgen. Die Architektur empfiehlt sich bei Erweiterungen von bestehenden Frameworks (siehe Abschnitt 3.2.3).

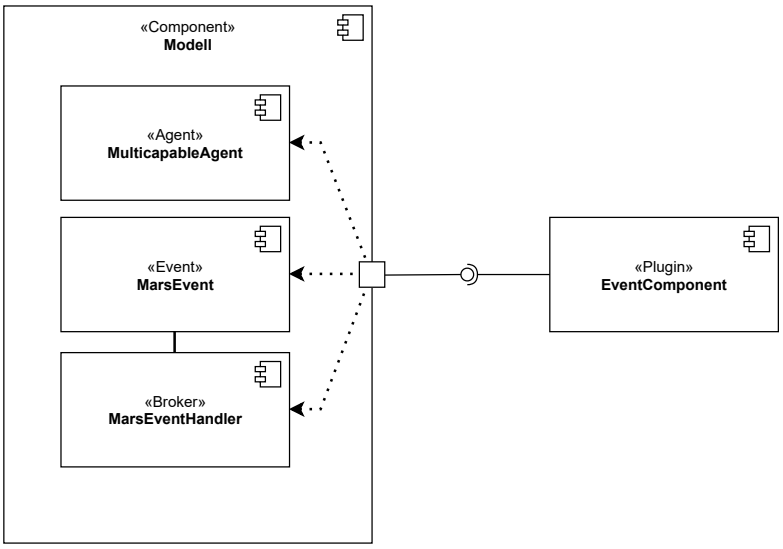

Abbildung 5.11: Komponentendiagramm Plugin-Architektur

Wie auf dem Diagramm zu sehen ist können EventComponents einfach ausgetauscht werden. Das definierte Interface ermöglicht es dem Modell, die unterschiedlichen Implementierungen innerhalb der Pluginkomponente zu ignorieren. Außerdem ist zu sehen, welche der im Modell befindlichen Bestandteile mit dem Plugin interagieren.

- Der MulticapableAgent muss in der Lage sein eine oder mehr beliebige Komponenten anzumelden, um so über alle gewollten Arten von Events informiert zu werden und auf diese reagieren zu können.

- Das MarsEvent ist die abstrakte Elternklasse aller Events. Die EventComponent muss diese verarbeiten können.
- Der MarsEventHandler teilt der Komponente eintretende Events mit.

Es wird deutlich, dass eine lose Kopplung der Komponenten erreicht werden kann.

5.5 Darstellung und Bearbeitung von Ereignissen

Ein eintretendes Ereignis wird als ein Event behandelt. Um die vorher angesprochenen Konzepte und Architekturen zusammenzubringen und einen gesamtheitlichen Überblick über das Kernkonzept dieser Arbeit zu verschaffen, wird das nachfolgende Diagramm verwendet.

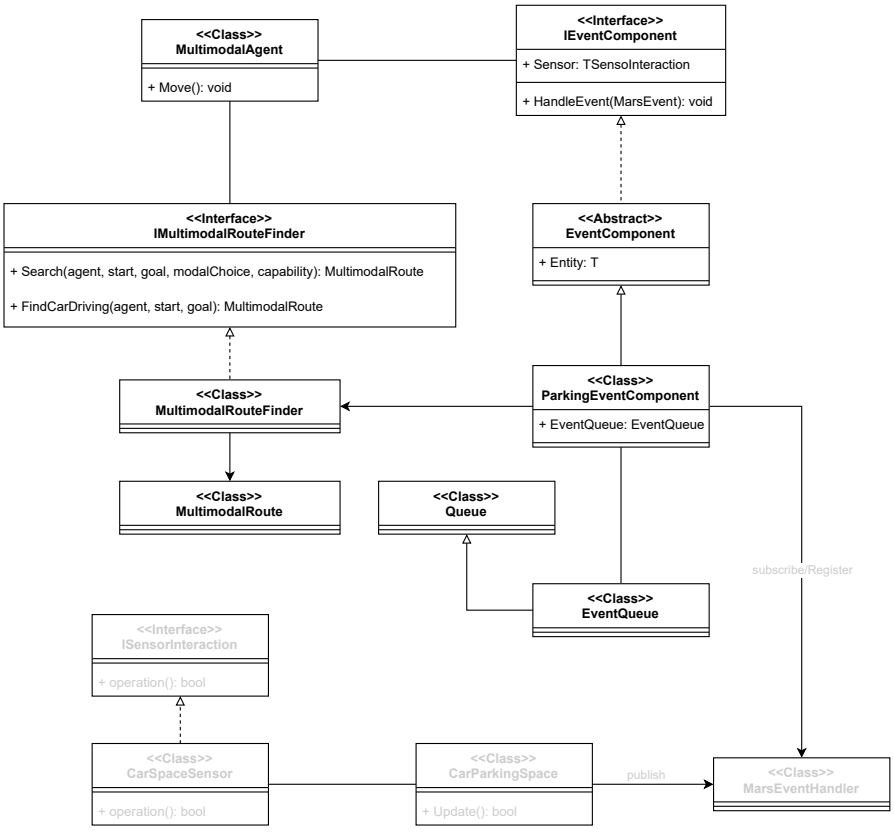

Abbildung 5.12: Klassendiagramm Adapter Pattern

Der MultimodalAgent ist in der Lage sich bei einem oder mehreren Komponenten anzumelden. Diese Anmeldung ist die Initialisierung einer bestimmten Komponente. In der weiteren Beschreibung wird es sich um eine bestimmte Ausprägung einer Komponente handeln. Diese Komponente ist die ParkingEventComponent. Nach der Initialisierung muss der Agent sich nicht weiter um eingehende Änderungen des Parkraums kümmern. Die ParkingComponent übernimmt das Handling aller eintreffenden ParkingEvents. Diese Bearbeitung ist entstanden durch folgende Gedanken.

- Man kann die Komponente als Smartphone betrachten, auf dem gerade eine Routenplanung zum eingegebenen Ziel stattfindet. In diesem Fall würde das Smartphone automatisch die Route anpassen, wenn das angefahrene (und bereits bestätigte Ziel) nicht mehr frei bzw. erreichbar ist. Es ist nicht notwendig, dass der Nutzer (der Agent) sich aktiv damit beschäftigt.
- Ausgehend vom vorausgehenden Punkt wäre die Routenanpassung keine aktive Handlung des Agenten und muss daher auch nicht aktiv in seiner Tick-Methode behandelt werden. Die Behandlung kann in der Komponente parallel bearbeitet werden.

Dem Diagramm weiter folgend ist zu erkennen, dass die genutzte Komponente eine Event-Queue besitzt. Diese Queue kann genutzt werden, um eingehende Events einer Priorität nachzuordnen. Diese Ordnung kann von Interesse sein, wenn entschieden werden muss, ob es ein wichtigeres Event gibt, auf das vorher reagiert werden soll. Eine andere Möglichkeit der Nutzung dieser Queue ist der Aufbau eines Gedächtnisses. Die Komponente kann bereits eingetretene Events speichern und gegebenenfalls mit diesen Informationen arbeiten. Da die Komponente in der Lage sein soll, die Route des Agenten während der Abarbeitung einer bereits bestehenden Route zu ändern, muss sie eine neue MultimodalRoute erstellen können. Es ist performanter eine neue Route zu konstruieren von der Position, an der der Agent sich befindet bei Eintreffen des Events (Parkplatz besetzt). Die alternative Möglichkeit ist die tatsächliche Anpassung der alten Route. Wählt man die Anpassung statt der neu Erstellung, so muss auf die Klassen MultimodalRoute zugegriffen werden. Hierbei ergeben sich Probleme. Ein ernst zu nehmendes Problem ist die schon vorher angesprochene aktive Entwicklung des MARS-Frameworks. Im Rahmen dieser Entwicklung ergeben sich kontinuierliche Änderungen am Framework. Während der Bearbeitung dieser Aufgabe wird aktiv die MultimodalRoute Klasse überarbeitet. In Absprache mit den zuständigen Entwicklern wurde dann über die Möglichkeiten der Routenanpassung diskutiert. Das Ergebnis ist die Entscheidung, nicht simultan an der-

selben Klasse zu arbeiten, um Inkonsistenzen und mögliche Fehler zu vermeiden. Durch die komplette Neuberechnung der Route (von einem neuen Startpunkt aus) kann über den MultiModalRouteFinder eine neue Route berechnet werden, ohne gleichzeitig an der gleichen Klasse zu arbeiten. Ein weiterer Vorteil ist das wegfallende Aufräumen nach der Neuberechnung. Gibt es eine neue Route, wird die alte Route überschrieben und es bleibt keine Referenz auf die alte Route zurück. Somit kann die automatische Speicherverwaltung sie aus dem Speicher entfernen.

Im Diagramm in grauer Schrift ist die Verteilung der simulierten IoT-Sensordaten zu sehen. Die ParkingEventComponent registriert den Agenten mit seiner HandleEvent-Methode bei dem MarsEventHandler und subscribt sich auf diese Weise für die für ihn relevanten Events. Fährt nun innerhalb der Simulation ein Auto auf einen Parkplatz für den Agenten (bzw. einer Komponente) registriert wurden, so kann die Komponente das eingehende Ereignis bearbeiten. Ein ParkingSpace startet hierfür bei der Initialisierung einen Sensor. dieser ist so lange frei, wie kein Fahrzeug den Parkplatz benutzt. Wurde der ParkingSpace nun angefahren und ein Auto steht auf ihm, wird der Sensor aktiviert und published ein Event. Dieses Event wird vom MarsEventHandler empfangen und verarbeitet (siehe Abschnitt 5.4 und 5.4).

Wahl der Eventverarbeitungsmethode

Im vorangehenden Abschnitt wurde die Wahl begründet, wie ein ParkingEvent die Route neu bauen soll. Es wurde beschrieben, dass die neue Route die alte überschreiben muss. In diesem Abschnitt ist mithilfe eines Sequenzdiagramms die genaue Kommunikation aufgezeigt. Vor der Erläuterung des Diagramms werden, um einen Kontext und ein gewisses Hintergrundwissen zu schaffen, alle unbekannten Klassen erläutert.

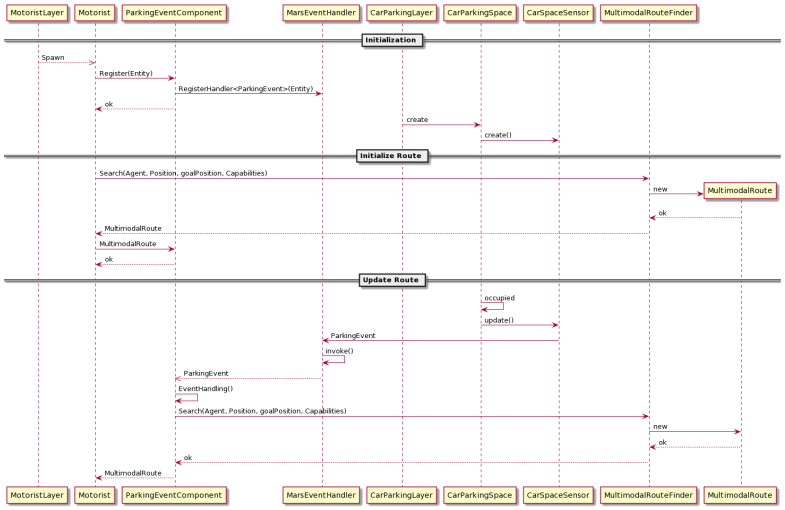

Abbildung 5.13: Sequenzdiagramm Kommunikation zur Routenanpassung

MotoristLayer Stellt den Layer dar, welcher für die Initialisierung von Agenten bestimmt ist. Der MotoristLayer ist ein Layer über den der Agent (der auf ihm lebt) mit den verschiedenen Environments interagieren kann. Er kapselt die Logik der verschiedenen Environments vom Agenten weg. Der MotoristLayer erbt vom MultimodalLayer.

Motorist Ist der Agent dieser Simulation. Er erbt von dem MultimodalAgent und dem CapableAgent. Durch diese Vererbungshierarchie ist es ihm möglich, die Funktionalitäten dieser zu nutzen. Zu diesen Fähigkeiten gehört beispielsweise die Möglichkeit der multimodalen Fortbewegung. Ein jeder Motorist besitzt ein eigenes Auto, einen Startort und einen Zielort. Die Tick-Methode dieses Agents beinhaltet im Wesentlichen die Aufgabe von diesem Startort zu seinem Zielort zu kommen. Bei dieser Routenabarbeitung verschiedene Attribute beschrieben, um spätere Experimente durchzuführen und Messungen erstellen zu können. Für die Beantwortung der Frage dieser Arbeit ist es ihm außerdem möglich ParkingEvents zu verarbeiten.

ParkingEventComponent Die genaue Aufgabenbeschreibung dieser Klasse bzw. ihrer Funktionsweise ist im Abschnitt 5.5 zu finden.

MarsEventHandler Agiert als Broker für die Publisher und die Subscriber. Die genaue Aufgabenbeschreibung dieser Klasse ist im Abschnitt 5.5 zu finden.

CarParkingLayer Ist ein Layer, mit dem der Agent interagieren kann. Er gehört nicht zu den Environmentlayern. Dieser Layer beinhaltet alle eingeladenen Parkplätze. Um die fachliche Logik klar getrennt zu haben und dem Prinzip der Verteilung von Aufgaben gerecht zu werden, interagieren Agenten nicht direkt mit den Parkplatzobjekten. Sie können stattdessen mit **einem** ParkingLayer interagieren und nicht mit einer Vielzahl verschiedener ParkingSpaces. Alle Informationen der Parkplätze können über ihn erfragt und gegebenenfalls geändert werden.

CarParkingSpace stellen einen Parkraum dar. Sie werden vom ParkingLayer gehalten. Ein ParkingSpace muss kein Einzelparkplatz sein. Er hat einen *Count*, welcher die Anzahl der Parkplätze innerhalb einer gewissen Ausdehnung beträgt. Ein Beispiel ist ein Parkhaus. Innerhalb der Simulation bekommen Parkplätze einen IoT-Sensor, welcher den Zustand frei oder belegt an einen Broker published.

CarSpaceSensor Wird von einem jeden ParkingSpace initialisiert. Diese Klasse repräsentiert den Sensor dieses Parkraumes. Seine Aufgabe ist es, wenn der Parkplatz voll ist, den Status an den Broker zu senden. Sollte der Parkplatz voll gewesen sein und wird nun wieder freigegeben, so published der Sensor den Status„nicht belegt“.

MultimodalRouteFinder Ist das Verbindungsglied zwischen der ParkingEventComponent und der MultimodalRoute. Um die direkte Anpassung der MultimmodalRoute zu umgehen (siehe Diagramm 5.13) wird diese Klasse zur Kommunikation genutzt. Sie wird im Rahmen dieser Arbeit nicht modifiziert.

Das Diagramm ist in drei Abschnitte aufgeteilt:

- **Initialization**: Innerhalb dieser Phase ist die erforderliche Kommunikation dargestellt, welche notwendig ist, um einen CarSpaceSensor (ein IoT-Sensor eines Parkplatzes) zu initialisieren. Ein MotoristLayer spawnt alle Agenten der Simulation. Außerdem werden bestimmte Attribute des Agenten einem Wert zugeordnet. Alle Motorist bekommen hier als Capability die Möglichkeit einen eigenen Wagen fahren zu können. Die Autos werden in unmittelbarer Nähe zu seinem Besitzer gespawnt – dies erfordert einen ParkingSpace in der Umgebung. Anschließend registrieren sich die Agenten bei einer ParkingEventComponent. Durch diese Registrierung sind die Agenten in der Lage auf ParkingEvents zu reagieren. Eine Registrierung ist auch

nur notwendig, wenn der Agent diese Möglichkeit bekommen soll. Sobald ein neues ParkingEventComponent-Objekt erstellt wurde, wird sich dieses bei dem Broker (MarsEventHandler registrieren). Mit diesem Schritt ist nun die erste der beiden voneinander logisch getrennten Schritte abgeschlossen und Events können erhalten werden. Um Event erstellen zu können werden nun die Sensoren benötigt.

Für die Initialisierung von Sensoren sind folgende Kommunikationen zwischen den Komponenten (Klassen) nötig. Ein CarParkingLayer muss mit den Informationen der Konfigurationsdatei (siehe Abschnitt A.1) für Parkraum initialisiert werden. Es werden die Kapazitäten und Positionen der Parkräume auf die ParkingSpaces übertragen und die Objekte erstellt.Bei Erstellung der ParkingSpaces werden von diesen nun die CarSpaceSensors erstellt. Diese sind initial nicht belegt. Mit diesen Schritten ist es nun auch möglich Events zu erstellen.

- **Initialize Route**: In diesem Abschnitt ist die erforderliche Kommunikation zu sehen, welche notwendig ist für die Erstellung einer Route. Dieser Vorgang wird bei der Simulation gemacht, um dem Agenten eine Route zu übergeben. Der Motorist benötigt eine erste Route und muss hierfür den MultimodalRouteFinder darüber informieren. Dieser erstellt dann eine Route für den Agenten. Um die richtige Route zu erstellen, benötigt er Informationen. Es wird eine `Position`übergeben, welche den Start der Route darstellt und eine `goalPosition` als Zielort. Außerdem werden die dem Agenten zu Verfügung stehenden Möglichkeiten der Fortbewegung mitgeteilt, seinen `Capabilities`. Diese liegen in Form einer Liste vor und beinhalten für dieses Modell immer die Fähigkeit zu Fuß zu gehen (immer gegeben) und mit dem eigenen Fahrzeug zu fahren. Weitere Fortbewegungsmittel stehen dem Agenten nur zur Verfügung, wenn es für bestimmte Szenarien gewünscht ist. Beispielsweise um die Fahrzeiten von Autos und Fahrrädern zu vergleichen.

- **Update Route**: Es wird aufgezeigt, wie der komplette Ablauf des Eintretens eines Events, bis hin zu der Aktualisierung der Route aussieht. Um dieses Szenario zu starten wird davon ausgegangen, dass ein Wagen einen Parkplatz besetzt, welcher von einem anderen Agenten angefahren wird. Der Prozess ist aus Sicht dieses Agenten. Ein Fahrzeug fährt auf den Parkplatz. Der Parkraum, welcher durch diesen ParkingSpace dargestellt wird, ist nun belegt (er hat keine weiteren Kapazitäten für andere Fahrzeuge). Der Sensor dieses ParkingSpaces wird über die Update-Methode über diese Änderung informiert. Nun published der CarSpaceSensor ein ParkingEvent an seinen Broker. Das Event wird über die invoke-Methode ausgelöst,

um alle Subscriber zu informieren. Der die ParkingEventComponent ist registriert auf diesen Parkplatz und erhält das Event. Die Komponente entscheidet innerhalb der EventHandling-Methode, ob und wenn welche Logik ausgeführt werden muss. In diesem Szenario wird entschieden, dass eine neue Route benötigt wird. Die neue Route wird über die Search-Methode beim MultimodalRouteFinder angefordert. Als Startposition der Route wird die aktuelle Position des Agenten genommen und als Zielort wieder das schon bekannte Ziel. Innerhalb des RouteFinders wird nun der nächste freie Parkplatz in nächster Nähe zu dem Zielort angefahren. Nach Erhalt der neuen Route wird die alte überschrieben und der Agent fährt die neu erstellte, weiter ab.

Nach Ablauf der Anpassung der Route muss sich die ParkingEventComponent nur noch für den nun besetzten Parkplatz abmelden. Diese Abmeldung findet über die Unregister-Methode statt. Ist die Abmeldung beendet, registriert sich die Komponente für den neuen Parkplatz.

publish/subsribe Pattern

Das Publish-Subscribe Pattern ist ein Nachrichtenmuster innerhalb der Softwarearchitektur. Bei diesem kennen der Absender (die Publisher), die Empfänger (die Subscriber) nicht. Die Nachrichten werden nicht direkt an diese gesendet.Die Nachrichten werden in hierarchisch angeordneten Klassen organisiert und an die Subscriber von einem Server (Broker) verteilt. Es wird gesendet ohne zu wissen, ob Subscriber vorhanden sind. In ähnlicher Weise interessieren sich die Abonnenten (Subscriber) für eine oder mehrere Klassen und erhalten nur die Nachrichten, die sie interessieren, ohne zu wissen, ob und welche Publisher es gibt. Publish-Subscribe ist ein Modell, das mit dem Paradigma der Nachrichtenwarteschlange verwandt ist. Dieses Muster bietet eine gute Skalierbarkeit.

Die Umsetzung dieses Musters kann auch mit dem Beobachter-Entwurfsmuster erledigt werden. In diesem kann sich ein Abonnent bei einem Anbieter registrieren und Benachrichtigungen von diesem empfangen. Dieses Modell eignet sich besonders für die Verarbeitung von Benachrichtigungen. Das Muster definiert einen Anbieter und Beobachter. Beobachter registrieren sich beim Anbieter, und sobald ein Ereignis eintritt, benachrichtigt der Anbieter automatisch alle Beobachter, indem er eine definierte Methode von diesem aufruft.

Das Klassendiagramm verdeutlicht das zu implementierende Konzept.

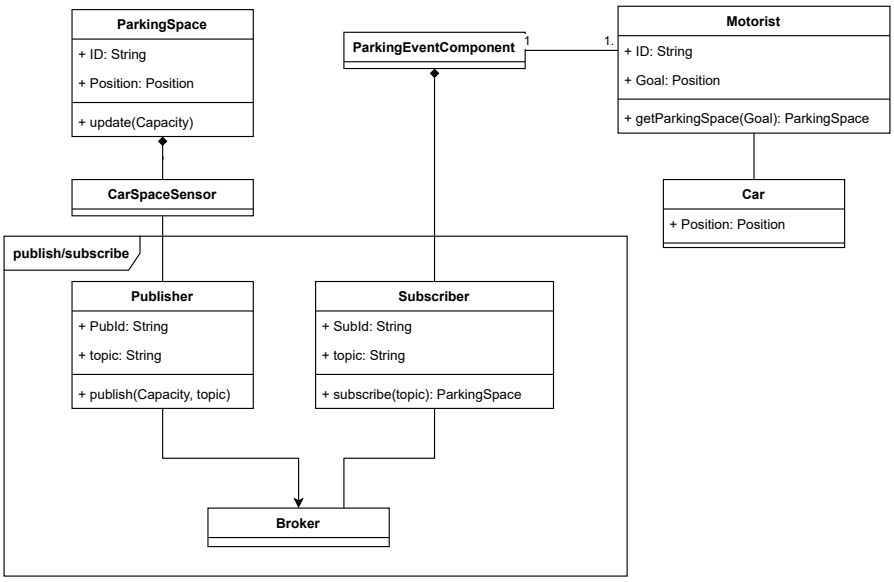

Abbildung 5.14: Klassendiagramm publish/subscribe Pattern

Der Fokus liegt innerhalb dieses Kapitels innerhalb des Rahmens **publish/subscribe**. Die Trennung von Zuständigkeiten soll im ganzen System erhalten bleiben und wird auch hier weitergeführt. Der Agent interagiert über seine Komponente mit den Sensoren. Diese haben sich als Subscriber beim Broker für den richtigen Parkplatz registriert. Die ParkingSpaces haben sind über ihre Sensoren als Publisher bei dem gleichen Broker. Durch diese Architektur ist eine lose Kopplung gegeben. Außerdem entstehen keine zyklischen Abhängigkeiten und Komponenten sind leicht austauschbar.

5.5.1 Priorisierung der Events

Die Möglichkeit Events zu priorisieren ist in vielen Anwendungsfällen notwendig. Beispielsweise kann die Zeit, zu der ein Event eingetreten ist, seine Priorität definieren. Es ist auch denkbar, dass es wichtigere und weniger wichtige Events innerhalb einer Simulation gibt. Die Priorität kann sich daraus ergeben, ob auf ein Event sofort regiert werden muss, später reagiert werden kann oder ob es sogar ignoriert werden kann. Innerhalb der Belegung von Parkplätzen scheint es eine untergeordnete Rolle zu spielen. Ist ein Parkplatz, für den der Agent registriert ist, belegt, so muss die Komponente eine Routenanpassung durchführen. Mit Blick auf die Qualitätsanforderungen (siehe Abschnitt

3.2.3) ist es wichtig sich über diese Anforderung klar zu sein, dass viele Fragestellungen eine Priorisierung der Events benötigen.

6 Realisierung

Dieser Abschnitt befasst sich mit der programmatischen Umsetzung aller in der Anforderungsanalyse angesprochenen Anforderungen. Diese Umsetzung wird unter Berücksichtigung der Konzeption durchgeführt. Etwaige Änderungen, welche aufgrund von Programmiersprachen abhängigen Eigenarten gemacht werden müssen, werden in den jeweiligen Abschnitten näher erläutert. Zusätzlich zu den Konzepten wird einleitend die Datenbeschaffung beschrieben. Hierfür wurde kein Konzept entworfen, da das Sammeln und verarbeiten der Daten fallspezifisch stattfindet. Sie wird händisch durch geführt und es wird der Situation (wie liegen Daten vor, in welchem Ausmaß liegen sie vor und welcher Qualität entsprechen sie) nach gehandelt.

Der Anhang zur Arbeit befindet sich auf CD und kann beim Erstgutachter eingesehen werden. Die .cs-Dateien sind im Rahmen des SOH-Modells mit dem MARS-Framework implementiert und können nicht ausgeführt werden ohne Zugriff auf das MARS Life-System. Soll die Simulation ausgeführt werden, muss dies in dem Mars Life Repository (urlhttps://git.haw-hamburg.de/mars/life) unter dem Branch *feature/ba-paring* passieren. Alle genutzten und implementierten Klassen sowie die genutzten Layer liegen hier zur Einsicht. Die für die Layer genutzten Dateien liegen unter Scenarios/ParkingBehaviorBox/resources/ Ordner. Die genutzte Konfigurationsdatei sowie die Program.cs liegen unter Scenarios/ParkingBehaviorBox/.

6.1 Systemrelevante Datenverarbeitung

Im Rahmen der Anforderungsanalyse wurde festgehalten, dass auch die Datenbeschaffung ein elementarer Bestandteil dieser Simulation ist (siehe Abschnitt 3.1 und die use-cases im Abschnitt 3.2.2). Um die Realitätsnähe der Simulation zu gewährleisten, werden zur räumlichen Darstellung des Simulationsgebietes GIS-Daten verwendet. Die Wahl des geografischen Raumes unterliegt verschiedenen Überlegungen und Abwägungen.

1. Der simulierte Raum muss ein Ballungsgebiet sein, welches eine gewisse Bevölkerungsdichte beinhaltet. Eine Großstadt eignet sich für die Fragestellung am besten.
2. Die Datenlage der Stadt muss eine gewisse Vollständigkeit besitzen und die Datenlage muss ausreichend sein.
3. Die Daten müssen frei verfügbar sein.

In dieser Arbeit wurde sich für die Stadt Hamburg entschieden. Dieses Ballungsgebiet bietet sich an, da alle angesprochenen Voraussetzungen erfüllt sind.

6.1.1 Datenakquirierung

Hamburg verfügt über ein Datenportal (Datenportal Hamburg), welches über eine gewisse Qualität verfügt. Hier sind frei zur Verfügung stehende GIS-Daten bereitgestellt. Über dieses Portal gelangt man zu weiteren relevanten Seiten, für GIS-Daten von Hamburg. Es kann also als eine Art Sammelstelle für GIS-Daten dieses Raumes gesehen werden. Über die Themenauswahl gelangt man zu verschiedenen Kategorien von Daten. Es wird ein übersichtliches und leicht zu verstehendes Dropdown-Menü geöffnet, über das man seine Auswahl treffen kann. Über dieses Portal werden die Parkraumdaten gesucht und gedownloadet (Verlinkung zu einer weiteren Seite mit dieser Art von Daten).

Als nächsten Schritt werden Gebäudeinformationen benötigt. Um diese Informationen zu erhalten wird sich der OpenStreetMap (siehe Abschnitt 2.3) bedient. Dieses ist ein Projekt mit dem Ziel eine freie verfügbare Weltkarte zu erschaffen. Es ist möglich über die Homepage kleinere geografische Karten herunterzuladen. Ab einer bestimmten Ausdehnung ist dies nicht mehr möglich. Es existiert eine Bibliothek (osmnx) in der Programmiersprache Python. Mit dieser Bibliothek kann man beliebig große Karten automatisiert in verschiedenen Formaten herunterladen. Es wurde ein Pythonskript geschrieben, um aufsetzend auf der osmnx-Bibliothek verschiedene Informationen zu akquirieren. Bei der Implementierung dieses Codes sind eine Reihe von Herausforderungen zu bewältigen gewesen.

- Das Installieren der benötigten Pakete auf Windows konnte nicht über `pip install` gelöst werden. Dies resultiert daraus, dass zwei der von osmnx genutzten Bibliotheken nicht ohne weiteres unter Windows installiert werden konnten.

- die Bibliothek `fiona` hat ein Versionsproblem. Fiona wird genutzt, um die gesammelten Daten, welche nun in einer Reihe von Variablen stehen, in Form von GIS-Dateien zu extrahieren. Das Problem tritt auf, sobald das zu extrahierende Dateiformat nicht dem .graphml Format entspricht (siehe Abschnitt 2.2). Dieses Format hat leider den Nachteil, dass ausschließlich Graphen dargestellt werden können. Bei Gebäuden benötigt die Simulation Polygone. Außerdem ist die Visualisierung dieses Formats nicht ohne weiteres zu erhalten.

Die Herausforderungen dieser Aufgaben wurden mit einem Pythonprogramm gelöst, welches in der Lage ist über die osmnx-API beliebige Daten herunterzuladen. Als Eingabe wird der geografische Raum, ein beliebiger valider OSM-Tag (Ein Tag kann beispielsweise buildings, railway, amenity u.a. sein) und der gewünschte Name der Ausgabedatei benötigt. Da die Bibliothek zu Extraktion von internen Daten in GIS-Dateien nicht funktionsfähig war, wurde außerdem ein Geojson-Datei-Exporter gebaut. Um eine richtige Darstellung der Features innerhalb der Datei zu erhalten, muss dem Programm mitgeteilt werden, um welche Art von zu exportierender Struktur es sich handelt. Es gibt drei mögliche Ausprägungen:

1. (Multi-)Poligone
2. (Multi-)Lines
3. (Multi-)Points

Das Ergebnis eines erfolgreichen Programmablaufs ist dann eine valide Geojsondatei mit dem gewünschten Inhalt. Auf diese Weise konnten die Gebäude innerhalb der Stadt Hamburg mit einer Reihe von zusätzlichen Informationen (welche für die Simulation nicht notwendig sind) in ein Format gebracht werden, welches zur weiteren Verarbeitung und späteren Einladung in das MARS-Framework geeignet sind.

Um die verschiedenen Environments für die Agenten zu erhalten, konnte dasselbe Programm genutzt werden. Hierfür war es nur notwendig der API mitzuteilen, dass man Netzwerkinformationen für Fahrzeuge und anschließend für Fußgänger benötigt. Als Ausgabeformat der gewünschten Dateien wurde wieder das Geojson-Format gewählt, da dieses mithilfe der geeigneten Software leicht zu visualisieren, zu verifizieren und zu validieren ist. Die weiteren Schritte werden im nächsten Abschnitt genau erläutert.

6.1.2 Datenvorbereitung

Im Anschluss an die Datenakquirierung kann mithilfe der QGIS-Software (siehe Abschnitt 2.1) die Güte der Daten überprüft werden. Außerdem können enthaltene Unstimmigkeiten oder Fehler innerhalb der Daten gegebenenfalls korrigiert oder entfernt werden.

- ParkraumLayer (Multipolygon Layer) und Datenquelle
- BuildingLayer und Datenquelle
- Walkgraph und Drivegraph und die Methode der Extraktion

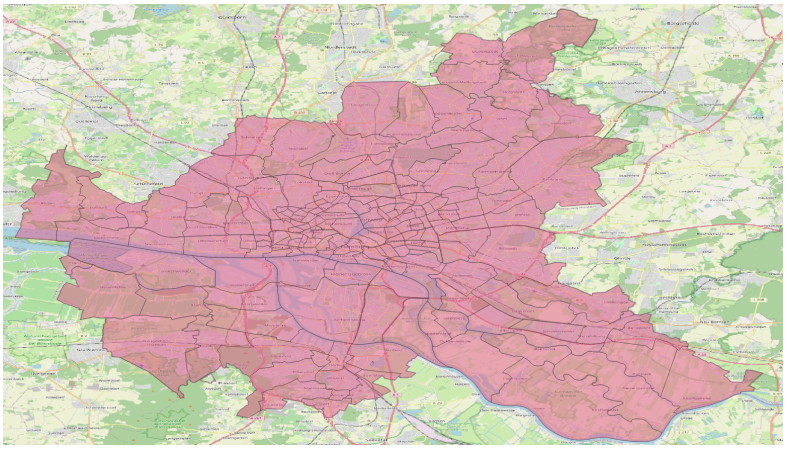

Abbildung 6.1: Stadt Hamburg (mit Stadtteilen und Bezirken)

Die Darstellung wird mit dem Koordinatenreferenzsystem EPSG:4326 - WGS 84 dargestellt (siehe Abschnitt 2.4). Auf dem Bild ist das Stadtgebiet von Hamburg mit hinterlegter OSM (siehe Abschnitt 2.3) zu sehen. Es dient der Veranschaulichung und Validierung. Außerdem kann mithilfe der über gelegten Schicht eine Schablone für die Simulation gebaut werden. Die Daten sind über das Esri Deutschland Open Data Portal akquiriert worden.

Abbildung 6.2: Gebäude im Stadtbereich Hamburg

Abbildung 6.3: Parkraum im Stadtbereich Hamburg

Der buildings-layer-hamburg besteht aus 307.561 Features (siehe links) und der parking-layer-hamburg aus 56.522 Features. Es handelt sich bei beiden Darstellungen um Geojson-Dateien (siehe Abschnitt 2.2). Ein Feature ist ein Polygon (somit sind beide Darstellungen Multipolgpon-Layer). Die Features werden in dem Schritt der Einladung in das MARS-Framework berücksichtigt. Es werden sowohl die Ausdehnung, als auch die Koordinaten in das Modell eingeladen.

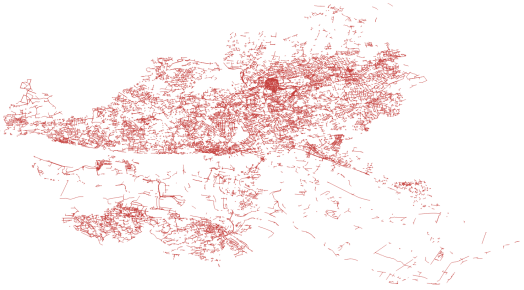

Abbildung 6.4: (fußläufiger) Graph im Stadtbereich Hamburg

Abbildung 6.5: (Straßennetz) Graph im Stadtbereich Hamburg

Bei diesen Graphen handelt es sich um jene, auf denen die Agenten sich bewegen werden (siehe Abbildung 5.3). Die Kanten dieser Graphen sind auf den Bildern dargestellt. Zur besseren Übersichtlichkeit wurden die Knoten entfernt. Ein Knoten stellt das Bindeglied zwischen zwei oder mehr Kanten dar (Kreuzungen). Außerdem beendet ein Knoten den Verlauf einer Straße. Die Anzahl der Knoten und Kanten:

- walk-graph-hamburg: 168.506 Knoten und 451.988 Kanten.
- drive-graph-hamburg: 21.727 Knoten und 52.338 Kanten.

Zeigt deutlich, mit was für einer Menge an Daten schon bei so kleinen geographischen Ausdehnungen zu arbeiten ist. Außerdem ist zu sehen, dass der Walk-Graph deutlich weniger dicht ist als der Drive-Graph. Das ergibt Sinn, da es mehr Möglichkeiten gibt zu Fuß zu gehen, als für Autos befahrbare Straßen existieren.

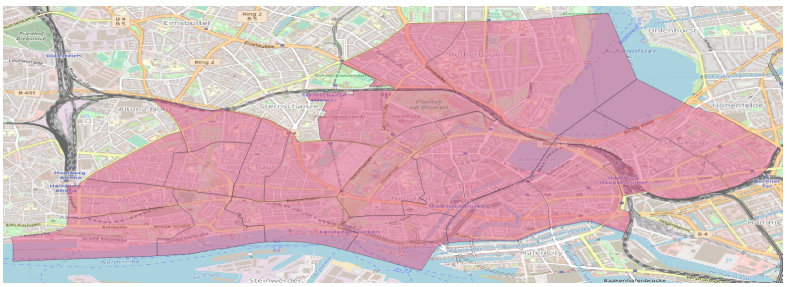

Abbildung 6.6: Schablone zur Bearbeitung der Fragestellung

Auf diesem Bild sind folgende Stadtteile von Hamburg ausgewählt worden (mithilfe von QGIS (ein Verarbeitungstool für GI-Daten) wird mit logischen Operatoren gearbeitet (dies wäre hier ein logisches OR)).

- Stadtteil:
  - St.Pauli, Altona-Altstadt, Neustadt, Hamburg-Altstadt, St.Georg und Rotherbaum

Diese Stadtteile werden gewählt, da sie im Zentrum von Hamburg liegen (hohe Bevölkerungsdichte) und weil die Parkräume vollständig genug sind.

Die Daten, welche für gesamt Hamburg vorliegen werden nun mithilfe der Schablone zugeschnitten. Der Algorithmus beschneidet einen Vektorlayer durch die Objekte auf einem zusätzlichen Polygonlayer. Nur die Teile der Objekte des Eingabelayers, die in die Polygone des Überlagerungslayer fallen, werden dem Ergebnislayer hinzugefügt. Die Attribute der Objekte werde nicht geändert, obwohl die Eigenschaften wie Flächen oder Länge der Objekte durch die Schnittoperation geändert werden. Wenn solche Eigenschaften als Attribute geführt werden, müssen sie manuell aktualisiert werden.

Abbildung 6.7: Gebäude im Simulationsgebiet

Abbildung 6.8: Parkraum im Simulationsgebiet

Dargestellt sind die eben genannten Stadtteile, welche für das Modell herangezogen wurden. Sie entsprechen demselben Typ wie schon die Darstellung von gesamt Hamburg. Es sind dementsprechend Multipoligon-Layer mit einer Anzahl von Features. Diese Features haben eine Ausdehnung und sind bestimmt durch ihre Koordinaten. Von links nach rechts die Beschreibung der Diagramme (zur Verwendung innerhalb der Simulation):

- buildings-layer-sim: Bestehend aus 8.446 Features. Ein jedes ist ein Gebäude einer bestimmten Größe.
- parking-layer-sim: Bestehend aus 24.389 Features. Ein jedes ist Parkraum einer bestimmten Größe.

Abbildung 6.9: (fußläufiger) Graph im Simulationsgebiet

Abbildung 6.10: (Straßennetz) Graph im Simulationsgebiet

Der obigen Beschreibung folgend von links nach rechts die Graphen zur Bewegung der Agenten (zur Verwendung innerhalb der Simulation):

- walk-graph-sim: Bestehend aus 8.477 Knoten und 24.554 Kanten. Hierauf können die Agenten wie im Use Case beschrieben (siehe 3.1 und 3.3) zum Auto gehen und vom Auto zum Ziel.
- drive-graph-sim: Bestehend aus 1.251 (Kreuzungen) und 2.761 Kanten (Straßen und Straßenabschnitten). Hierauf können die Agenten wie im Use Case beschrieben (siehe 3.2) von einem Parkplatz mit ihrem Auto zu einem anderen Parkplatz fahren.

6.2 Datenbehandlung innerhalb des Frameworks

6.2.1 Temporäre Daten

Daten, die nur während der Simulationszeit zur Verfügung stehen. Die Verwaltung dieser Daten wird von dem MARS-Framework übernommen. Daten, die nicht für die spätere Auswertung und/oder Visualisierung genutzt werden sollen, werden mit dem Zugriffsmodifikator ungleich `public` beschriftet. Diese umfassen alle `Properties (Eigenschaften)` und Variablen, die nur für die Logik innerhalb der Simulation nötig sind.

6.2.2 Zu persistierende Daten

Daten, welche aus der Simulation herausgeschrieben werden sollen. Es werden von dem Framework alle Properties heraus geschrieben, welche mit dem Zugriffsmodifikator `public` deklariert sind und einem bestimmten Datentyp entsprechen. Daten-Typen, die auf diese Weise in ein Ausgabemedium geschrieben werden können, sind folgende.

- Alle einfache Datentypen
- von den Referenzdatentypen der String-Type

Für dieses Projekt werden die `TravelTime` (vom Typ Integer) und die `ID` (vom Typ Guid) von allen Agenten benötigt. Außerdem wird für die grafische Darstellung eine Geojson-Datei (siehe Abschnitt 2.2) exportiert. Diese Datei ist notwendig, um am Ende eines Durchlaufes sowohl die Fahrroute, als auch die Anzahl der Agenten visuell nachvollziehen zu können. Sie enthält die unterschiedlichen Koordinaten zu einem bestimmten Zeitpunkt der Agenten, welche dann mit einem Geoverarbeitungstool (QGIS[3] oder kepler.gl[2](für die Visualisierung mit zeitlichem Verlauf)) dargestellt werden können. Die Datei wird unter der dem Namen „trips.geojson“ gespeichert.

Für das Ausgabeformat der Simulationsdaten (für die spätere Auswertung der Experimente) wurde das CSV-Format gewählt (der Name der Datei ist „Motorist.csv“). Eine Alternative wäre die Speicherung der Daten in einem üblichen Datenbankformat gewesen (z.B. einer relationalen Datenbank wie SQL). Es wurde aber davon ausgegangen, dass die Datenmenge nicht die Größenordnung erreicht, welche den Mehraufwand einer Datenbankeinrichtung gerechtfertigt hätte. Über einen Outputfilter kann innerhalb der Konfigurations-Datei (config.json) auf die benötigten Informationen gefiltert werden. Im

Laufe der Implementierung ist festgestellt worden, dass dieser mit der momentanen Version des Frameworks nicht funktionsfähig ist. Daraufhin wurde die Entscheidung noch einmal überdacht und, da die Ausgabe-Daten schon teilweise vorlagen, ein Pythonskript geschrieben worden. Das Skript war notwendig, da die einzelnen CSV-Dateien zwischen 2.5 und 3.2 GByte groß waren und übliche Applikationen (Excel oder Notepad++) diese nicht mehr öffnen konnten. Das Pythonskript filtert die Ergebnisse pro Agent bei berechneter `TravelTime` (diese wird immer bei Ankommen des Agenten am Zielort von 0 auf die Reisedauer gesetzt).

6.3 Wahl der Programmiersprache

Die Wahl der Programmiersprache ist davon abhängig gemacht worden, in welcher Sprache das MARS-Framework und das SOH-Modell geschrieben wurde. Das Ziel der Erweiterung des Frameworks hat mich dazu veranlasst, die Sprache C# zu nutzen. Außerdem wurde für die Datenbeschaffung, Datenaufbereitung und den Outputfilter Python genutzt. Die Entscheidung der zweiten Programmiersprache fiel aufgrund meiner Erfahrungen in dieser Sprache. Python bietet davon ab eine Vielzahl an frei verfügbaren Bibliotheken, im Speziellen für Datenaufbereitung und Datenverarbeitung. Eine große Community ist in diesen Bereichen sehr aktiv. Gerade im Bereich Datenanalyse (Data Analytics) ist Python ein de facto Standard.

6.4 Implementierung

Nachfolgend wird auf die wesentlichen Komponenten der Implementierung eingegangen. Da das Eventhandling und die Integration in das Modell eine zentrale Rolle in der Implementierung spielt, wird in diesem Kontext eine genaue Ausführung gemacht. Es gibt eine Vielzahl an verschiedenen Möglichkeiten, um eventbasierte Programmabläufe zu implementieren. Bevor aber die verschiedenen Möglichkeiten beschrieben und diskutiert werden, soll ein Rahmen geschaffen werden, in dem diese verarbeitet werden. Die zentrale Komponente des Eventhandlings ist der MarsEventHandler. Wie im Konzept beschrieben (siehe Diagramm 5.7) wird dieser als Singleton implementiert. Das MARS-Framework arbeitet in vielen Aspekten asynchron und mit einer Vielzahl an Threads. Unter dieser Bedingung muss die implementierte Singleton-Klasse Threadsicher sein. Es gibt verschiedene Möglichkeiten in C# das Singleton-Pattern umzusetzen. Grob unterteilt:

1. Eine nicht Thread sichere Version. Diese kommt im Rahmen dieses Modells nicht infrage.

2. Einfache Thread-Sicherheit mithilfe des `lock` Keywords von C#. Diese Implementierung ist threadsicher. Der Thread sperrt das gemeinsam genutztes Objekt und prüft, ob die Instanz bereits existiert, bevor die Instanz erneut erstellt wird. Auf diese Weise wird sichergestellt, dass nur ein Thread eine Instanz erstellt (da sich jeweils nur ein Thread in diesem Teil des Codes befinden kann. Der Nachteil dieser Methode ist die beeinträchtigte Leistung, da jedes Mal, wenn die Instanz angefordert wird, eine Sperre ausgelöst wird.

3. Threadsicherheit durch Double-Check-Locking. In dieser Version wird nicht jedes Mal eine Sperre gesetzt. Die Performanz ist aber noch nicht optimal.

4. Den Typ `System.Lazy<T>` verwenden. Es muss ein Delegat an den Konstruktor übergeben werden, der den Singleton-Konstruktor aufruft (was mit einem Lambda-Ausdruck möglich ist). Durch diese Implementierung ist es außerdem möglich, mit dem Property `IsValueCreated` zu prüfen, ob die Instanz bereits erstellt wurde.

Es gibt weitere mögliche Implementierungen, diese werden nicht weiter beschreiben. Sie sind verschiedene Abwandlungen der oben beschrieben.

6.4.1 Möglichkeit der ereignisbasierten Eventverarbeitung

Ein Event ist in .NET eine Benachrichtigung, die von einem Objekt gesendet wird, um das Eintreten einer Aktion darzustellen. Dieses Modell basiert auf dem Delegatmodell und folgt dem Beobachterentwurfsmuster. Dieses Muster ist in C# wie folgt umgesetzt:

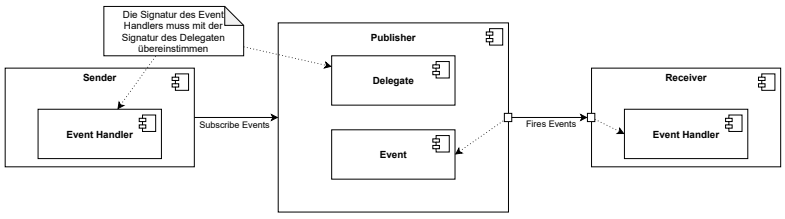

Abbildung 6.11: Delegatmodell von C#

Das Delegat wird über eine Signatur deklariert. Bestandteile dieser Signatur sind der Rückgabetyp und die Parameter. Dies entspricht damit einem typsicheren Funktions-

zeiger. Diese Möglichkeit der Eventverarbeitung von C# liegt sehr nahe an der Konzeption der in dieser Arbeit beschriebenen Verarbeitung und bietet sich daher für die Nutzung an. Es gibt verschiedene Arten von Delegaten. Das `Action`-Delegat besitzt keinen Rückgabewert. Es kann dementsprechend nur mit Methoden genutzt werden, welche den Rückgabewert void haben. Der alternative Typ ist das `func`-Delegat, welches einen Rückgabewert besitzen kann. Im Rahmen dieser Ausarbeitung und der hier gestellten Fragestellung ist dies aber nicht notwendig.

6.4.2 Framework

In diesem Abschnitt wird beschrieben, welche Komponenten im MARS-Framework hinzugefügt wurden. Außerdem werden Abweichungen zu der Konzeption angesprochen, falls gegeben. Komponenten, die hier implementiert worden sind:

1. MarsEvent: Ist eine abstrakte Klasse und implementiert das `IEvent`. Als zu instanziierende Klasse wird eine `ParkingEvent`-Klasse implementiert. diese wird auf Modellebene implementiert. Durch diese Hierarchie ist es dem Entwickler (auch für andere Fragestellungen) ermöglicht worden seine eigenen spezifischen Eventklassen zu erstellen. Dies geschieht unter dem Aspekt der Erweiterbarkeit des MARS-Frameworks.

2. EventComponent: Ist eine abstrakte Klasse und implementiert das `IEventComponent`. Diese abstrakte Klasse bekommt eine `Entity` mit. Hierdurch ist sichergestellt, dass eine Komponente immer eine möglichst abstrakte Darstellung eines Agenten nutzen kann. Außerdem gibt diese Klasse eine Methode vor, welche bei Nutzung der Komponente dem EventHandler übergeben werden muss.

3. MarsEventHandler: Es konnte sich in Gänze an das Konzept gehalten werden. Diese Klasse ist als threadsichere Singletonklasse implementiert (Version 4 der angesprochenen möglichen Singleton-Varianten) und nutzt das Interface `IEventHandler`. Sie bekommt bei einer Registrierung eine `IEntity` und eine `Action` mit. Die IEntity ist möglichst Abstract gewählt, um dem Entwickler möglichst viele Freiheiten in der Nutzung zu überlassen. Die einzig geforderte Property ist eine eindeutige `Guid` des Objektes. Die Action ist vom Typ `MarsEvent` und muss nur sicherstellen, dass auch die Action eine eindeutige `Guid` besitzt. Auf diese Weise kann der MarsEventHandler alle verschiedenen Arten von MarsEvents behandeln, ohne in

Typprobleme zu laufen. Bei einer Registrierung wird geschaut, ob das zu registrierende MarsEvent schon bei dem Handler bekannt ist. Falls nicht, wird ein neuer Eintrag in einem Dictionary angelegt und die Entität der Eventart hinzugefügt. So kann ein und dieselbe Entität auf eine Vielzahl von verschiedenen Events registrieren. Um die Möglichkeit der Identifizierung der angemeldeten Entitäten zu behalten, besteht ein Eintrag in einem Event immer aus einem Tupel aus Guid der Entität und der Action. Die Action ist die Speicheradresse der auszuführenden Methode. Ist das Event dem EventHandler bereits als Key bekannt, so wird eine Hinzufügung zu diesem gemacht (wieder als Tupel). Bei einem `Unregister-Handler` wird dem EventHandler mitgeteilt für welches Event sich eine Entität abmelden möchte und innerhalb dieses Keys dann nach der Guid gesucht, um diese dann gegebenenfalls zu entfernen. Soll ein Event von einem Sensor ausgelöst werden, muss auf dem Eventhandler die `Invoke`-Methode ausgeführt werden. In diesem Fall wird bei vorhanden sein des Events (Event muss dem EventHandler bekannt sein) die Methode `GetInheritanceChain` ausgeführt (siehe Abschnitt 5.4). Diese triggert dann für alle Basisklassen des auszuführenden Events und die Eventklasse selbst die registrierten Aktionen (Die Methoden innerhalb der verschiedenen `EventComponents`.

4. ISensorInteraction: Wird als Interface vorgegeben, um die simulierten Sensordaten (Sensor-Klassen auf Modellebene) zu definieren.

6.4.3 Modell

Auf Modellebene implementierte Klassen sind:

1. ParkingEvent: Ist eine Kindklasse der abstrakten `MarsEvent`-Klasse. Im Rahmen der Fragestellung und der Vorgabe der Mutterklasse wird nur eine eindeutige ID benötigt. Es können Modellabhängig alle gewünschten Informationen eines Events hier als Attribute mitgegeben werden.

2. ParkingEventComponent: Ist eine Kindklasse der abstrakten `EventComponent<T>`-Klasse. `T` steht in dieser Arbeit für eine Entität vom Typ `Motorist`. Die Komponente speichert diese ab, um eine mögliche Anpassung der Route des Agenten vorzunehmen. Bei Initialisierung wird außerdem der angefahrene Parkplatz abgespeichert, um dessen ID dann bei eintreffenden Events abzugleichen. Nun wird die `HandleEvent`-Methode bei EventHandler registriert. Kommt ein Event rein, für

das sich die Komponente registriert hat, wird die ID des ParkingEvents mit der gespeicherten ID abgeglichen. Ist diese identisch, kann eine Routenanpassung vorgenommen werden. Eine Routenanpassung wird durch den Aufruf `ReRouteToGoal` ausgeführt. Diese Methode befindet sich innerhalb des `MultiModalAgents`. Hierdurch kann eine neue multimodale Route für den Agenten gesucht werden. Diese neue Route überschreibt die bisherige Route des Agenten. Auf diese Weise werden auch alle zugehörigen Parameter der Route erneuert (Dies ist beispielsweise das Attribut `GoalReached`). Der Motorist bekommt von dieser Ersetzung nichts mit und fährt seine Route wie gehabt weiter ab. Abschließend wird der neu angefahrene CarParkingSpace für einen Abgleich der IDs in der Komponente abgespeichert.

3. ParkingSpaceSensor: Ist eine Kindklasse der abstrakten `ISensorInteraction`-Klasse. Jeder `CarParkingSpace` initialisiert bei Erstellung eine Instanz dieser Klasse. Sie besitzt eine eindeutige ID und die Methode `TriggerEvent`. Diese Methode wird immer dann ausgeführt, wenn der CarParkingSpace `Occupied` ist. Dieses Attribut eines Parkplatzes ist bei vollständiger Belegung `True`. In diesem Fall wird die TriggerEvent-Methode des zugehörigen Sensors ausgeführt. Innerhalb der Methode wird die `Invoke`-Methode des EventHandlers aufgerufen und alle registrierten Agenten über ihre `ParkingSpaceEventComponent` informiert. Das Verhalten der Agenten wird dann durch die `HandleEvent`-Methoden bestimmt.

Abweichung vom Konzept gibt es bei der ParkingEventComponent und der Art der Routenanpassung. Das eigentliche Konzept sah keine Änderungen in der MultimodalRoute vor (siehe Diagramm 5.13). Im Laufe der Implementierung ist aber klar geworden, dass dies nicht zu umgehen ist. Die Änderung der Route außerhalb des vorgesehenen Kontextes führte zu einer Vielzahl an Fehlern. Diese Fehler wurden durch bestimmte Faktoren ausgelöst. Um diese zu verstehen, muss der Aufbau einer MultimodalRoute im SOH-Modell erläutert werden.

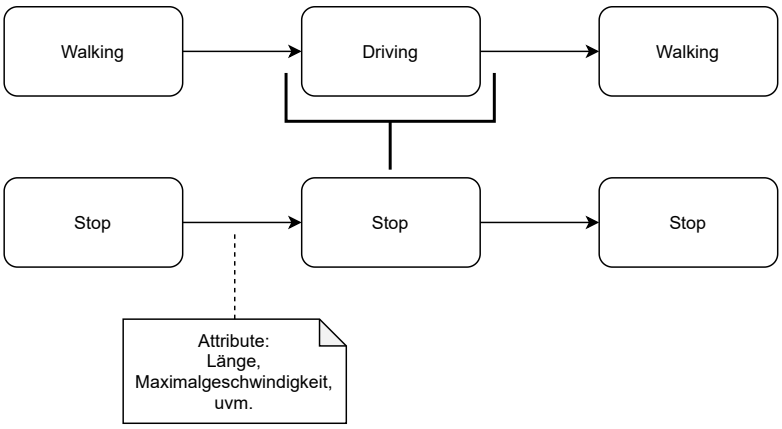

Abbildung 6.12: MultimodalRoute in MARS

Eine solche Route besteht beispielsweise aus einem Walking, dann einer Driving und dann wieder einem Walking Anteil (vom Gebäude geht der Agent zum Auto, von dort fährt er zum nächsten Parkplatz, um dann das letzte Stück der Strecke zu Fuß zu beenden). Dieser Ablauf entspricht den Use Cases 3.1, 3.2 und anschließend 3.3. Ursprünglich konnte ein Agent diese Route nur umplanen, wenn eines dieser Streckenteile beendet war. Mit der hier beschriebenen eventbasierten Routenanpassung muss die Routenanpassung innerhalb des Drivingparts der Strecke funktionieren. Hierfür wurde die `ReRouteToGoal`-Methode angepasst. Diese ruft die Methode `AppendAndDeleteTail` auf. Innerhalb dieser wird eine neue Route von der derzeitigen Position zum Zielort ermittelt und gebaut. Anschließend wird die alte Route ab der derzeitigen Position gelöscht und die neue Route der alten hinzugefügt. Daraus resultiert dann eine neue Gesamtroute, welche aus der zusammengesetzten alten Route in Verbindung, mit der neu berechneten Route entsteht. In der Konzeption wurde nicht bedacht, dass es nicht möglich war eine Route innerhalb eines dieser Teilabschnitte zu splitten. Also wurde abweichend die `AppendAndDeleteTail` verändert. Nun ist es möglich, die multimodale Route auch innerhalb eines solchen Teilabschnittes zu teilen und neu zusammenzubauen. Für die fehlerfreie Ausführung musste dafür außerdem eine neue Länge der Gesamtroute berechnet und zugewiesen werden, wie auch die neue Anzahl der `Stops` angepasst werden. Diese Änderungen wurden durch die Aufrufe der Methoden:

- ReRouteToGoal: Berechnung und Erstellung einer neuen Route.
- AppendAndDeleteTail: Einkürzen der aktuellen Route um die nicht benötigten Teile (z.B. wird der letzte Walking-Abschnitt nicht benötigt).

- DeleteTail: Einkürzung innerhalb eines Streckenabschnittes und Neuberechnung der Länge der Strecke und dessen Stops. Beispielsweise wird nun die Driving-Route verändert.

Implementiert.

6.5 Konfiguration und Ausführung des Modells

Ein Modell wird im MARS-Framework über eine Konfigurationsdatei gesteuert. Hier werden die Agenten, Entitäten, Layer, Simulationshyperparameter (z.B. Startzeit der Simulation, Ende der Simulation, Zeiteinheit u.ä.) und Ausgabeeinstellungen getroffen (eventuelle Outputfilter). Die Einstellungen der Agenten umfasst auch deren Anzahl. Bei den Layern kann mitgegeben werden, wie hoch beispielsweise die Parkplatzbelegung des CarParkingLayers ist. Das Format ist eine wohldefinierte Json-Datei mit dem Namen `config.json` (siehe Anhang A.1). Die config.json wird abhängig von den gemachten Experimenten angepasst. Es wurde die Parkplatzbelegung des CarParkingLayers variiert und in anderen Experimenten wurde ein SchedulingLayer genutzt. Durch dessen Nutzung ist die in der Konfiguration gemachte Angabe der Agentenanzahl nichtig. Die Agentenanzahl orientiert sich dann an der Scheduling-Datei (siehe Anhang A.1). Für die Experimente wurde unter anderem eingestellt, dass jede Minute 10 Agenten spawnen sollen. Unabhängig von der Art der Initialisierung der Agenten starten diese immer am gleichen Ort und fahren zum selben Ziel. Abhängig von der gewünschten Art der Agentenregistrierung muss in der `Program.cs` angegeben werden, welchen Layer das auszuführende Modell nutzen soll. Entweder wird der `MotoristSchedulingLayer` genutzt (dann wird die initiale Agentenanzahl von 1000 ignoriert und minütlich 10 Agenten gespawnt, bis zu einer Anzahl von abermals 1000) oder es werden direkt bei Simulationsstart 1000 Motorist-Agenten gespawnt. Zum Starten des Modells wird die Program.cs-Datei ausgeführt, welche dann die Simulation startet. In dieser Datei wird über die `ModelDescribtion` jeder Layer eingeladen, welcher in der config.json referenziert wurde.

Anmerkung: Im Laufe der Ausführung kam es selten zu einem Fehler, der darauf zurückzuführen ist, dass es in dem Simulationsgebiet (siehe Abbildung 6.4 und 6.5) isolierte Teilgrafen besitzt.

6.6 Tests

Zur Sicherstellung der Richtigkeit der Implementierung wurden verschiedene Unittests implementiert. Es ist nicht möglich alle möglichen Fälle zu testen. Es wird sich darauf beschränkt, die Hauptfunktionalitäten der verschiedenen Komponenten abzudecken und das nach Möglichkeit an denkbaren Grenzfällen.

Aufbau

Testumgebungen:

- Es wird ein Graph mit 4 Knoten erstellt. Auf diesem Graphen ist es möglich zu Fuß zu gehen und Auto zu fahren. Der Agent muss von Knoten 1 zu Knoten 4.
- Es wird ein `SpatialGraphEnvironment` gebaut. Dieser Graph wird aus der Datei *WalkGraphAltonaAltstadt* für die Fußgänger und *DriveGraphAltonaAltstadt* für die Autofahrer erstellt. Der Graph besteht aus einem kleinen Teil von Altona in Hamburg und wird auch für andere Tests innerhalb des SOH-Modells genutzt. Er ist zu finden unter den *ResourcesConstants*-Dateien. Die Konfiguration lässt keine Einbahnstraßen zu. Außerdem wird ein CarParkingLayer geografisch passend zu dem Testgebiet eingeladen.

Die Start- und Zielkoordinaten sind festgesetzt und es gibt nur einen Agenten. Startkoordinate ist 9.9337397,53.5446853 und Zielkoordinate ist 9.9566304, 53.5570752. Der Agent ist ein *TestMultiCapableCarDriver*. Dieser Agententyp verfügt nur über eingeschränkte Fähigkeiten, um sicherzustellen, dass so wenig Komplexität wie möglich innerhalb des Agenten vorhanden ist. Dieser Agententyp kann zu Fuß gehen, mit seinem Auto fahren und sich eine multimodale Route bauen. Die Tests werden innerhalb der*EventDrivenReRouteTest*-Klasse gemacht. Es wird sowohl mit dem einfachen Graphen (4 Knoten), als auch mit dem komplexeren SpatialGraphEnvironment getestet.

Testfall 1: ReRouteEventHandleTest

Getestet wird zuerst, ob die Initialisierung am eingegebenen Start- und Zielpunkt ist. Anschließend werden die zur Verfügung stehenden Modalitäten des Agenten getestet, um sicherzustellen, dass dieser sowohl laufen, als auch mit seinem eigenen Auto fahren

kann. Nachdem sicher ist, dass die Initialisierung wie erwartet funktioniert hat, wird das eigentliche Ziel dieses Tests durchgeführt. Kann der Agent Events empfangen und reagiert er auf eine zu erwartende Weise?

Testfall 2: ReRouteEventGoalReachedTest

Dieser Test wurde geschrieben, um sicherzustellen, dass die Anpassung der Route richtig funktioniert. Nach dem ersten erfolgreichen Test kann davon ausgegangen werden, dass ein Agent ein Event empfängt. Nun wird geprüft, ob die Route wie gewünscht geteilt und richtig wieder zusammengefügt wird. Der Test wird erst auf dem einfachen Graphen und schließlich noch auf dem deutlich komplexeren Hamburg-Altona-Graphen ausgeführt.

7 Experimente und Ergebnisse

Nach abgeschlossener Implementierung und Prüfung der Plausibilität des Modells werden in diesem Abschnitt eine Reihe von Experimenten durchgeführt. Es wird der Aufbau und der Ablauf eines jeden Experiments erläutert. Außerdem werden die gewonnenen Ergebnisse beschrieben. Die Interpretation der jeweiligen Ergebnisse wird im nächsten Abschnitt vorgenommen.

7.0.1 Experiment 1

In diesem Experiment geht es darum, zu einem bestimmten Zeitpunkt eine große Menge an Agenten spawnen zu lassen. Diese müssen dann zu demselben Zielort. Dieses Experiment wird sowohl mit, als auch ohne Sensoren durchgeführt, um auf diese Weise Unterschiede in den Fahrzeiten herausstellen zu können. Außerdem wird der Parkraum in dem simulierten Gebiet immer weiter eingeschränkt.

Aufbau

Es werden für jeden Durchlauf 1000 Agenten gestartet. Diese starten bei den Koordinaten: 9.955990, 53.553548 und fahren zu den Zielkoordinaten 10.020904, 53.556362 (die entspricht einer Entfernung von fast genau 6 km). Die ersten 10 Durchläufe werden mit Sensornutzung durchgeführt. Begonnen wird mit 0% initialer Parkraumbelegung, um dann für jeden folgenden Durchlauf die Parkraumbelegung um 10% zu erhöhen bis zu einer maximalen initialen Parkraumbelegung von 90%. Anschließend wird das gleiche Prozedere mit 1000 Agenten ohne Sensornutzung durchgeführt.

Ablauf

Ablauf der ersten 10 Durchläufe mit Sensornutzung. Die Startposition ist ein Gebäude innerhalb des Simulationsgebietes (Die Koordinaten sind 9.955990, 53.553548). Jeder Agent registriert sich bei seiner Initialisierung für einen Parkplatz, welcher frei und möglichst nahe am Zielort liegt. Der Zielort ist ein Gebäude im Simulationsgebiet (10.020904, 53.556362). Bei Registrierung wird der gewählte Parkplatz für alle anderen Agenten blockiert. Es ist anderen Agenten in diesem Experiment nicht möglich einen von einem anderen Agenten gewählten Parkplatz zu nehmen. Jeder Agent fährt auf schnellstem Weg zu seinem Parkplatz, dieser ist immer der am nächsten gelegene noch verfügbare zum Zielgebäude. Die Zeitmessung beginnt bei Simulationsstart und endet, sobald der Agent in dem Zielgebäude angekommen ist. Begonnen wird mit einer initialen Parkraumbelegung von 0% und mit jedem weiteren Durchlauf wird diese um 10% erhöht bis zu einer maximalen Belegung von 90%.

Nun wird dieses Experiment noch einmal durchgeführt, mit der einzigen Änderung, dass die Agenten in den nachfolgenden 10 Durchläufen keine Sensorunterstützung zur Parkraumsuche nutzen. Alle Agenten fahren nun direkt zum Zielort, um dann dort nach einem freien Parkplatz zu suchen. Ist ein Agent am Zielort angekommen, wird in einem 10 Meter Radius nach einem freien Parkplatz gesucht. Wird keiner gefunden, fährt der Agent eine Zeiteinheit (Sekunde) in eine zufällige Richtung. Diese Routine wird so lange wiederholt, bis der Agent einen freien Parkplatz in seinem Radius gefunden hat. Anschließend wird dieser Parkplatz gewählt und der Agent beendet seine Reise mit dem Erreichen des Zielortes. Die Staffelung der initialen Parkraumbelegung, die Anzahl der Agenten, der Start- und Zielort und die Zeiterfassung sind identisch zu den Durchläufen mit Sensornutzung.

7.0.2 Ergebnisse zu Experiment 1

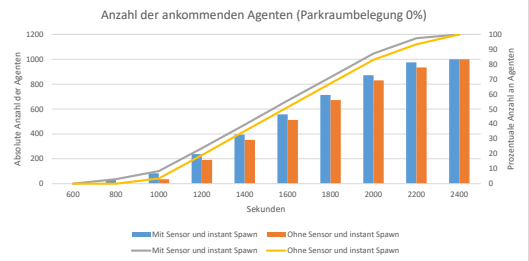

Abbildung 7.1:
Sofortiger Spawn und 0%
Parkplatzbelegung

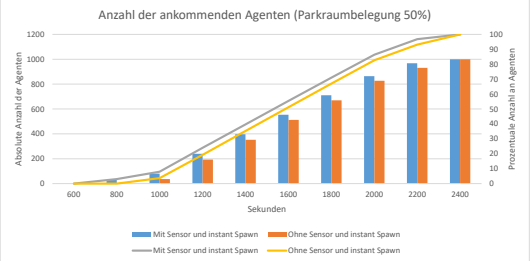

Abbildung 7.2:
Sofortiger Spawn und 50%
Parkplatzbelegung

Auf den Abbildungen 7.1 und 7.2 ist zu sehen, dass bei steigender Zeit immer mehr Agenten ankommen. Bei einer Zeit von 2400 Sekunden (40 Minuten) sind 100% der gespawnten Agenten an ihrem Ziel angekommen, das betrifft sowohl die Agenten mit Sensornutzung, als auch ohne Sensornutzung. Die ersten Agenten kommen bei einer Zeit von 800 Sekunden (ca. 13 Minuten) an und fahren mit Sensorunterstützung. Grundsätzlich sieht man wie die Agenten mit Sensorunterstützung eine kürzere Gesamtreisedauer haben. Die Unterschiede zwischen einer Parkraumbelegung von 0% und 50% ist nicht klar zu identifizieren.

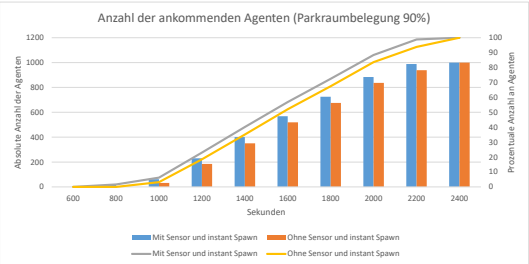

Abbildung 7.3:
Sofortiger Spawn und 90%
Parkplatzbelegung

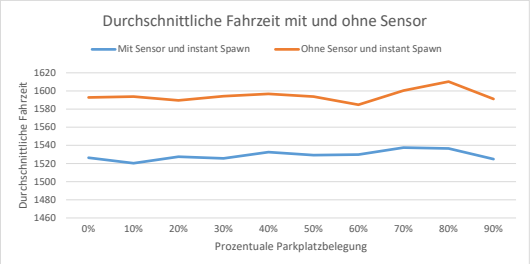

Abbildung 7.4:
Sofortiger Spawn mit durchschnittlicher
Fahrzeit

Auf der Abbildung 7.3 kann man erkennen, dass nur noch vereinzelnd Agenten bei 800 Sekunden (ca. 13 Minuten) ankommen, deutlich weniger als bei einer Parkraumbelegung von 0% und 50%. Die Entwicklung der ankommenden Agenten über die Zeit entspricht in etwa der Entwicklung bei niedrigeren Parkraumbelegungen. Zu erkennen ist, dass der Unterschied zwischen Agenten mit Sensornutzung zu den Agenten ohne diese erkennbar höher ist (im Vergleich zu niedrigeren Parkraumbelegungen). Auf Abbildung 7.4 sieht man, dass die durchschnittliche Fahrzeit bei Sensornutzung deutlich unter der durchschnittlichen Fahrzeit liegt, wenn keine Sensoren zur Unterstützung genommen werden. Dies gilt für alle Durchläufe, unabhängig von der initialen Parkraumbelegung.

7.0.3 Experiment 2

Dieses Experiment simuliert Verhalten, welches bei einer hohen Auslastung der Straßenkapazitäten entsteht. Agenten werden über einen gewissen Zeitraum mit dem gleichen Start- und Zielgebiet initialisiert. Es soll durch dieses Experiment außerdem simuliert werden, wie die Routenanpassung während der Fahrt eine Verbesserung der Reisedauer bewirkt.

Aufbau

Es werden im Minutentakt 10 Agenten initialisiert. Dieser Takt wird ausgeführt bis insgesamt 1000 Agenten gespawnt wurden. Die Start- und Zielkoordinaten (Start: 9.955990, 53.553548 und Ziel: 10.020904, 53.556362) sind bei allen Agenten identisch. Es wird wieder mit einer Initialen-Parkraumbelegung von 0% begonnen, um diese dann in 10er Schritten zu erhöhen (bis maximal zu einer Belegung von 90%). All diese Durchläufe werden einmal mit und einmal ohne Sensorunterstützung durchgeführt.

Ablauf

Die ersten 10 Durchläufe wurden wieder mit Sensorunterstützung gemacht. Der Ablauf ist nun aber etwas anders, da die Parkplätze nicht mehr blockiert werden. Beispielsweise starten die ersten 10 Agenten und planen ihren Weg zum nächsten freien Parkplatz in direkter Nähe zum Zielort. Während diese Agenten unterwegs sind, starten weitere Agenten, die dasselbe Ziel ansteuern. Sobald nun der erste Agent diesen Parkplatz erreicht,

werden die anderen Agenten (die sich auf diesen Parkplatz registriert haben) benachrichtigt, dass sie eine neue Route planen müssen (nun zum nächsten noch freien Parkplatz in direkter Nähe zum Zielort). Jeder der 10 Durchläufe entspricht diesem Verfahren, mit dem Unterschied der (in 10er Schritten) erhöhten initialen Parkraumbelegung (maximale Belegung von 90%). Die Start- und Zielorte bleiben identisch. Die Zeitmessung beginnt bei Start der Simulation und endet bei Erreichen des Zielgebäudes.

Anschließend wurden die gleichen 10 Durchläufe ohne Sensorunterstützung gemacht. Alle Agenten fahren nun direkt zum Zielort, um dann dort nach einem freien Parkplatz zu suchen. Ist ein Agent am Zielort angekommen, wird in einem 10 Meter Radius nach einem freien Parkplatz gesucht. Wird keiner gefunden, fährt der Agent eine Zeiteinheit (Sekunde) in eine zufällige Richtung. Diese Routine wird so lange wiederholt, bis der Agent einen freien Parkplatz in seinem Radius gefunden hat. Anschließend wird dieser Parkplatz gewählt und der Agent beendet seine Reise mit dem Erreichen des Zielortes. Start, Ziel, Staffelung der initialen Parkraumbelegung und Zeiterfassung sind identisch zu den ersten 10 Durchläufen mit Sensorunterstützung.

7.0.4 Ergebnisse zu Experiment 2

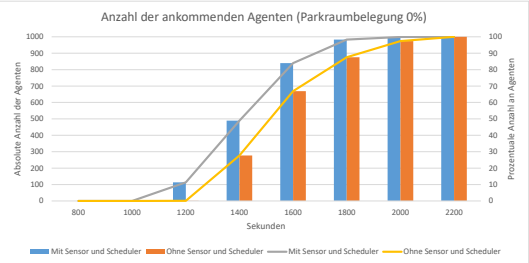

Abbildung 7.5:
Kontinuierlicher Spawn und 0% Parkplatzbelegung

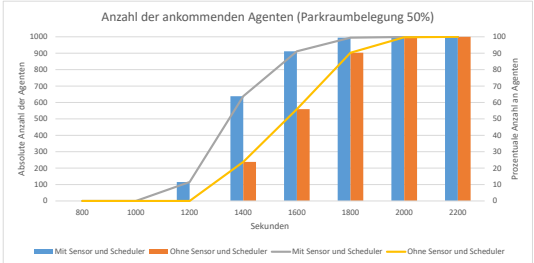

Abbildung 7.6:
Kontinuierlicher Spawn und 50% Parkplatzbelegung

Auf den Abbildungen 7.5 und 7.6 ist zu sehen, dass die ersten Agenten ca. bei 1000 Sekunden (ca. 16 Minuten) ankommen und mit Sensoren interagieren. Alle Agenten sind

spätestens bei 2200 Sekunden (ca. 33 Minuten) am Ziel angekommen. Agenten mit Sensornutzung haben eine deutlich kürzere Gesamtfahrzeit. Der Unterschied in der Reisedauer nimmt deutlich zu bei steigender Parkraumbelegung.

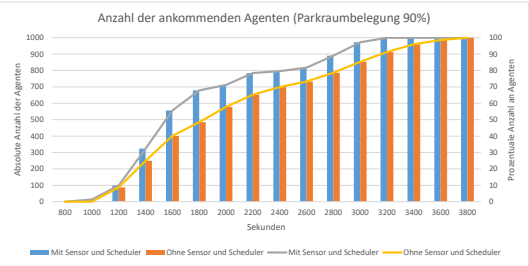

Abbildung 7.7:
Kontinuierlicher Spawn und 90% Parkplatzbelegung

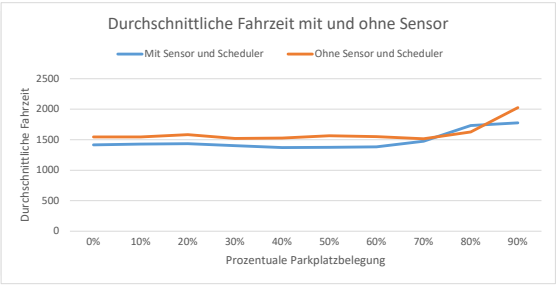

Abbildung 7.8:
Kontinuierlicher Spawn mit durchschnittlicher Fahrzeit

Auf der Abbildung 7.7 wird eine deutlich höhere Gesamtreisedauer benötigt. Die letzten Agenten mit Sensornutzung sind bei 3600 Sekunden (60 Minuten) am Ziel und die Agenten ohne Sensornutzung sind spätestens bei 3800 Sekunden (ca. 63 Minuten) dort. Die kürzere Fahrzeit zieht sich über den gesamten Verlauf der Simulation. Die Abbildung 7.8 zeigt, dass die durchschnittliche Fahrzeit aller Agenten (unabhängig, ob mit oder ohne Sensorunterstützung) sehr nahe bei einander liegt. Bei hoher Parkraumbelegung steigt die Fahrzeit für Agenten ohne Sensornutzung deutlich an.

8 Diskussion und Ausblick

8.1 Diskussion

Die Ergebnisse der Experimente lassen den Schluss zu, dass die Nutzung von IoT-Sensordaten im Parkraumsuchverkehr eine Verminderung des innerstädtischen Bereichs bewirkt. Diese Verminderung entspannt die Gesamtsituation in diesem Bereich, indem der Parkraumsuchverkehr, welcher die Kapazitäten der Straßenverkehrsnetze stark belastet, entfernt. Bei einer Nutzung von solchen Sensordaten können auf diese Weise nicht nur die Fahrzeiten der einzelnen Teilnehmer gemindert werden, sondern auch die der anderen. Die Verlängerung der Fahrzeit durch andere Teilnehmer, welche auf der Suche nach einem freien Parkplatz sind, entfällt gänzlich. Die Experimente haben außerdem gezeigt, dass das Reservieren von Parkraum und die Sperrung dieses Parkplatzes für andere Teilnehmer die größten Zeitersparnisse mit sich bringt. Bei dieser Art der Sperrung ist aber darauf hinzuweisen, dass es von der idealisierten und unrealistischen Annahme ausgeht, dass jeder Straßenverkehrsteilnehmer mit der Sensorunterstützung fährt. Fährt ein Teilnehmer ohne Sensorunterstützung und parkt dann auf einem freien Parkplatz (wie momentan üblich) wird dieses Optimum an Fahrzeitersparnis beeinträchtigt. Der registrierte Fahrer (registriert auf den nun besetzten Parkplatz) muss nun eine neue Route suchen. Selbst bei der Annahme, dass alle Fahrer mit IoT-Sensorunterstützung fahren würden, so fehlt es momentan noch an der Hardware. Es gibt im Moment noch keine Sensoren für öffentlichen Parkraum. Die einzigen Parkräume, bei denen momentan mit einer ähnlichen Herangehensweise gearbeitet wird, sind Parkhäuser (diese zählen häufig einfahrende und ausfahrende Autos). Auf diese Weise kann im Rahmen des Parkhauses eine Aussage über das vorhanden sein oder auch nicht vorhanden sein von Parkplätzen gemacht werden. Die Experimente zeigen aber auch schon eine Verbesserung der Fahrzeit bei teilweiser Nutzung von Sensoren. Mit Blick auf die Kapazitäten der Straßennetze und die Umweltvermutung durch Autoabgase kann auch diese schon als lohnenswert betrachtet werden.

Weitere Fragen sind im Laufe der Bearbeitung dieser Fragestellung aufgetreten. Mit der Annahme, dass die Städte ihre Parkräume mit Sensoren ausstatten und diese dann von den Straßenverkehrsteilnehmern stark genutzt werden, kommt es zu weiteren nicht trivialen Problemstellungen. Eine Stadt besitzt eine große Anzahl an Parkflächen und es gibt eine noch größere Anzahl an Autos. Sind diese in einer großen Anzahl für bestimmte Parkplätze registriert, entsteht eine riesige Flut an Daten. um diese Daten effizient verarbeiten zu können benötigt man Lösungsstrategien. Eine Strategie könnte sein, dass sich Autofahrer nicht mehr auf bestimmte Parkplätze registrieren, sondern stattdessen auf Stadtteile. Es müsste dann für verschiedene geografische Bereiche ein Server bereitgestellt werden. Diese Server könnten dann die Flut der einkommenden und ausgehenden Informationen bewältigen. Hierbei muss festgestellt werden, welche geografischen Flächen bewältigbar sind, da die Bevölkerungsdichte sehr unterschiedlich sein kann. Zudem müssen die Sensoren und Server gewartet werden. Im Rahmen des Datenaufkommens hat sich außerdem eine weitere wichtige Frage ergeben. Zu welchen Zeitpunkten muss ein Sensor seine Informationen weitergeben (publishen) ohne dass Informationen verloren gehen oder zu viele Daten weitergereicht werden. Ein möglicher Lösungsansatz wird durch MQTT aufgezeigt (siehe Abschnitt 2.6)

Die Aussagekraft, der hier erlangen, Ergebnisse müssen zudem etwas abgeschwächt werden. Diese Abschwächung hat verschiedene Gründe:

1. Die Anzahl der Agenten entspricht nicht der tatsächlichen Anzahl der in dem Bereich der Simulation stattfindenden Straßenverkehrs. Obwohl eine Verkehrslast erzeugt wurde, welche die Simulation an die Realität annähern sollte, ist dies auch nur eine Annäherung. Eine genaue Anzahl der Teilnehmer ist kaum zu erlangen, da diese Anzahl stark abhängig ist von verschiedenen Faktoren. Einige der Faktoren sind z.B. Uhrzeit, Baustellen im Umfeld und der Wochentag.

2. In den Experimenten wurden ausschließlich Autofahrer betrachtet. Wie sich die anderen Straßenverkehrsteilnehmer auf die Ergebnisse auswirken, kann mit dieser Arbeit und den gemachten Experimenten nicht gesagt werden. Zum Straßenverkehr gehören neben den Autofahrern alle Menschen, die sich auf der Straße oder dem Fußgängerweg befinden.

3. Das gewählte geografische Gebiet und die dort geltenden Gesetze. In der hier gemachten Simulation wurde ein bestimmtes Gebiet gewählt. Es können nur bedingt Aussagen über andere geografische Gebiete gemacht werden. Die Bevölkerungsdichte und das dort vorhandene Straßennetz sind wichtige Faktoren für Fahrzeiten

zwischen zwei Punkten. Aussagen über die benötigten Fahrzeiten und die Sinnhaftigkeit der Nutzung von Sensordaten können außerdem von bestehenden Gesetzen stark beeinflusst werden. Wenn nun zudem in einem anderen Gebiet keine Parkplatzknappheit besteht, kann es möglich sein, dass es keine oder nur kaum messbare Unterschiede bei den Fahrzeiten gibt.

Um eine genauere Einschätzung der Einflussnahme verschiedener Faktoren in Bezug auf Fahrzeiten im innerstädtischen Verkehr zu erhalten, müssen weitere Simulationen gemacht werden. Unter anderem sollten mehr Verkehrsteilnehmer mit einbezogen werden, wie auch genaue Zahlen aus der Realität zur Validierung betrachtet werden.

Bei der Implementierung der für die These notwendigen Simulation sind außerdem einige Verbesserungen des MARS-Frameworks aufgefallen. Diese wurden zu großen Teilen verbessert, wie im Abschnitt 6.4.2 beschrieben. Es ist aber nicht auszuschließen, dass weitere Ungenauigkeiten oder noch nicht gefundene Probleme im Framework vorhanden sind. Diese Problematik war bekannt und wurde mit eingeplant, da das Framework einer stetigen Verbesserung unterliegt. Die aktive Weiterentwicklung des Frameworks ist somit sowohl positiv, da bei Problemen und Fehlern eine Lösung gefunden werden kann, als auch sehr zeitintensiv. Der hier beschriebene Entwurf wurde immer mit dem Fokus der leichten Erweiterbarkeit gemacht und der möglichen Nutzung in vielen Anwendungsfällen. Es ist deutlich geworden, dass dieser Fokus nicht mit dem übereinstimmte, mit dem die Routensuche implementiert wurde. Die Routensuche und Bewegung wurde ursprünglich mit dem Ziel der Beantwortung einer bestimmten Frage implementiert. Das führte zu einer Vielzahl von Problemen und einer Änderung des vorher angedachten Konzeptes.

8.2 Zusammenfassung

Im Rahmen der Experimente und im Laufe der Arbeit an dieser Fragestellung konnten verschiedene Fragen offengelegt, wie auch einige Erkenntnisse gewonnen werden. Zusammenfassend kann man sagen, dass es trotz der angesprochenen Einschränkung der Aussagekraft der Experimente und der Simulation an sich eine Verbesserung der Fahrzeiten bestätigt werden konnte. Diese wurde zwar nur im Stadtgebiet von Hamburg aufgezeigt, kann aber leicht abgeschwächt auch für andere Großstädte Deutschlands zutreffen. Das ist möglich, da es eine grundsätzliche Parkplatzknappheit in deutschen Metropolen gibt. Die im Rahmen der Beantwortung dieser These implementierte Erweiterung für

das SmartOpenHamburg-Modell kann innerhalb des MARS-Frameworks vielseitig eingesetzt werden. Die Funktionalitäten des ereignisbasierten Eventhandlings kann auf andere Fragestellungen mit wenig Aufwand angewendet werden. Durch die implementierte Erweiterung des Frameworks konnte auf diese Weise eine Möglichkeit geschaffen werden, um das Framework noch vielseitiger einsetzbar zu machen.

8.3 Ausblick

Kurzfristig kann die vorhandene Simulation um die verschiedenen Verkehrsteilnehmer erweitert werden und eine genauere Betrachtung der Ist-Werte aus der Realität gemacht werden. Eine große Herausforderung stellt neben dem komplexen Verhalten vieler Agenten miteinander die Datenbeschaffung dar. Längerfristig kann aus diesem Grund sinnvoll sein, neben weiteren Experimenten und der Verfeinerung gleichartiger Simulationen, die Datenbeschaffung zu erleichtern. Straßennetze liegen in quelloffener Form bereits über Portale wie OSM (siehe Abschnitt 2.3) vor. Komplette Parkraumdaten sind nach meiner Recherche nicht oder nur sehr unvollständig vorhanden. Eine Erfassung von Parkraum und anderen notwendigen Daten für diese Art von Fragestellungen ist sinnvoll. Eine Möglichkeit wäre hier die Erfassung über Satellitendaten. Auf diese Weise könnte Entscheidungsträgern im Bereich der Stadtplanung und im Speziellen der Verkehrslenkung eine effiziente Hilfe zur Verfügung gestellt werden. Die Erfassung über Satellitendaten macht es außerdem möglich, auch Daten bereitzustellen in Ländern, die nicht über eine so gute Datenlage wie Hamburg verfügen.

Literaturverzeichnis

[1] *epsg*. – URL https://epsg.org/home.html. – Zugriffsdatum: 05/01/2022

[2] *Large-scale WebGL-powered Geospatial Data Visualization Tool*. – URL https://kepler.gl/. – Zugriffsdatum: 05/01/2022

[3] *QGIS*. – URL https://www.qgis.org/de/site/. – Zugriffsdatum: 05/01/2022

[4] Brandes, Ulrik ; Eiglsperger, Markus ; Herman, Ivan ; Himsolt, Michael ; Marshall, M S.: GraphML progress report structural layer proposal. In: *International symposium on graph drawing* Springer (Veranst.), 2001, S. 501–512

[5] Butler, Howard ; Daly, Martin ; Doyle, Allan ; Gillies, Sean ; Hagen, Stefan ; Schaub, Tim u. a.: The geojson format. In: *Internet Engineering Task Force (IETF)* (2016)

[6] Dudenhoeffer, Ferdinand: *Wer kriegt die Kurve?: Zeitenwende in der Autoindustrie*. campus, 2016. – ISBN 9783593434612

[7] Freeman, Adam: *The Singleton Pattern*. S. 113–136. In: *Pro Design Patterns in Swift*. Berkeley, CA : Apress, 2015. – URL https://doi.org/10.1007/978-1-4842-0394-1_6. – ISBN 978-1-4842-0394-1

[8] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John M.: *Design patterns entwurfsmuster als elemente wiederverwendbarer objektorientierter software*. mitp, 2015

[9] Glake, Daniel ; Ritter, Norbert ; Clemen, et a.: Utilizing Spatio-Temporal Data in Multi-Agent Simulation. In: *2020 Winter Simulation Conference (WSC)* Bd. 2020-Decem, IEEE, 2020, S. 242–253. – URL https://ieeexplore.ieee.org/document/9384124/. – ISBN 978-1-7281-9499-8

[10] Huppertz, Bernd ; Stollenwerk, Detlef: *Halten-Parken-Abschleppen: Praxishandbuch mit Rechtsprechungsübersichten sowie Verwarnungs-und Bußgeldtabellen.* Richard Boorberg Verlag, 2019

[11] Geographic information — Referencing by coordinates / International Organization for Standardization. 01 2019. – Standard

[12] MQTT Version 5.0 / OASIS Open. 03 2019. – OASIS Standard

[13] Ocker, Florian J.: *Entwicklung eines agentenbasierten multimodalen Verkehrsmodells mit MARS*, Hochschule für Angewandte Wissenschaften Hamburg, Diplomarbeit, 2020. – URL https://smartopenhamburg.de/wp-content/uploads/sites/9/2021/09/ocker-ma-thesis.pdf

[14] Open Geospatial Consortium (Veranst.): *GeoPackage Encoding Standard.* 11 2021. – OGC 12-128r18

[15] Poslad, Stefan: Specifying protocols for multi-agent systems interaction. In: *ACM Transactions on Autonomous and Adaptive Systems* 2 (2007), November, Nr. 4, S. 15. – URL https://doi.org/10.1145/1293731.1293735

[16] Ramm, Frederik ; Topf, Jochen: *OpenStreetMap: Die freie Weltkarte nutzen und mitgestalten.* Lehmanns Media, 2010

[17] Taubenboeck, Hannes ; Wurm, Michael: *Globale Urbanisierung – Markenzeichen des 21. Jahrhunderts.* Berlin, Heidelberg : Springer Berlin Heidelberg, 2015

[18] Ulfia A. Lenfers, et a.: Improving Model Predictions—Integration of Real-Time Sensor Data into a Running Simulation of an Agent-Based Model. In: *Sustainability* 13 (2021), Nr. 13. – URL https://doi.org/10.3390/su13137000

[19] Weyl, Julius: *Developing a generic multi-agent car model to simulate road traffic with MARS*, Hamburg University of Applied Sciences, Dissertation, 2019. – 1–83 S

A Anhang

A.1 Konfiguration

config.json


```
{
 "id": "Parking Behavior",
 "globals": {
  "deltaT": 1,
  "startPoint": "2021-01-01T00:00:00",
  "endPoint": "2021-01-01T04:00:00",
  "deltaTUnit": "seconds",
  "console": true,
  "debug": true
 },
 "agents": [
  {
   "name": "Motorist",
   "count": 10,
   "file": "resources/motoristInit1k.csv",
   "outputs": [
    {
     "kind": "csv",
     "options": {
      "delimiter": ";"
     }
    },
    {
     "kind": "trips",
     "outputConfiguration": {
      "tripsFields": [
       "StableId"
      ]
     }
    }
   ],
   "individual": [
    {
     "value": true,
     "parameter": "ResultTrajectoryEnabled"
    },
    {
     "value": true,
     "parameter": "CapabilityDriving"
    }
   ]
  },
  {
   "name": "RandomDriver",
   "count": 0,
   "file": "resources/motoristInit1k.csv",
   "outputs": [
    {
     "kind": "csv",
     "options": {
      "delimiter": ";"
     }
    },
    {
     "kind": "trips",
     "outputConfiguration": {
      "tripsFields": [
       "StableId"
      ]
     }
    }
   ],
   "individual": [
    {
     "value": true,
     "parameter": "ResultTrajectoryEnabled"
    },
    {
     "value": true,
     "parameter": "CapabilityDriving"
```


Abbildung A.1: config.json

Program Start

```
{
   class Program
   {
      static void Main(string[] args)
      {
         Thread.CurrentThread.CurrentCulture = new
CultureInfo("EN-US");
         LoggerFactory.SetLogLevel(LogLevel.Off);
         var description = new ModelDescription();

description.AddLayer<SpatialGraphMediatorLayer>
(new[] { typeof(ISpatialGraphLayer) });

description.AddLayer<CarParkingLayer>(new[] {
typeof(ICarParkingLayer) });
         description.AddLayer<MotoristLayer>();

description.AddLayer<VectorBuildingsLayer>();

description.AddLayer<MotoristSchedulingLayer>();


        description.AddAgent<Motorist,
MotoristLayer>();
         description.AddEntity<Car>();
         ISimulationContainer application;
         if (args != null && args.Any())
         {
            var container =
CommandParser.ParseAndEvaluateArguments(des
cription, args);
            var config = container.SimulationConfig;
            config.SimulationRunIteration = 1;
            application =
SimulationStarter.BuildApplication(description,
config);
         }else
         {
            var file = File.ReadAllText("config.json");
            var simConfig =
SimulationConfig.Deserialize(file);
            application =
SimulationStarter.BuildApplication(description,
simConfig);
         }
         var simulation =
application.Resolve<ISimulation>();
         var watch = Stopwatch.StartNew();
         var state = simulation.StartSimulation();
         watch.Stop();
         Console.WriteLine($"Executed iterations
{state.Iterations} lasted {watch.Elapsed}");
         application.Dispose();
      }
   }
}
```

Abbildung A.2: Program.cs

Scheduling-Datei

startTime	endTime	spawningIntervalInMinutes	spawningAmount	CapabilityDrivingOwnCar	source	destination
0:00	1:40	1	10	true	POINT (9.954967 53.551885)	POINT (10.013230 53.555722)

Abbildung A.3: MotoristScheduler

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original