



TECHNISCHE UNIVERSITÄT
ILMENAU

Fakultät für Informatik und Automatisierung
System- und Software-Engineering

Bachelorarbeit

Konzeptionierung und Implementierung einer intelligenten Ladestation in einer diskreten eventbasierten Simulationsumgebung

Anmeldedatum: 05. Oktober 2017

Abgabedatum: 05. Januar 2018

Hochschullehrer: Prof. Dr. Armin Zimmermann

Betreuer: M.Sc. Thomas Dietrich

Angefertigt von: Ludwig Breitsprecher

Matrikelnummer 54131

ludwig.breitsprecher@tu-ilmenau.de

Zusammenfassung

Die vorliegende Bachelorarbeit integriert eine intelligente Ladestation in eine Simulationsumgebung für Multicopter.

Dabei wird ein realistisches Ladeverhalten simuliert. Der Ladealgorithmus basiert auf empirischen Daten. Die gemessenen Werte zeigen Gemeinsamkeiten mit den von der Literatur präsentierten Daten. Im üblicherweise eingesetzten Ladeverfahren für Lithium-Ionen-Akkumulatoren, Constant Current Constant Voltage (CCCV), wird ein großer Anteil der Gesamtkapazität linear aufgeladen. Die Ladegeschwindigkeit für den kleineren abschließenden Teil sinkt im Vergleich stark.

Im Mittelpunkt der objektorientierten Konzeptionierung und Implementierung stehen die Aggregation, Aufbereitung und Kommunikation von Daten, die den Ladeprozess betreffen. Der Nachrichtenaustausch mit den Multicoptern dient der Verbesserung der Geschwindigkeit und Verlässlichkeit des Ladeprozesses. Die Ladestation kann unter anderem eine Vielzahl von Multicoptern verwalten und deren Ladeprozess organisieren, den Ladeverlauf prognostizieren und einen passenden Ersatzmulticopter zur Verfügung stellen, falls ein solcher in Verbindung mit der Ladestation steht. Darüber hinaus kann kurzfristig die geladene Gesamtkapazität erhöhen. Das wird durch die Priorisierung der Multicopter, die sich innerhalb der linearen Ladephase befinden, erreicht.

Inhaltsverzeichnis

1. Einleitung	1
2. Grundlagen	3
2.1. Unbemannte Luftfahrzeuge	3
2.1.1. Multicopter	3
2.2. Projektumgebung	4
2.2.1. C++ und Framework OMNeT++	4
2.2.2. Projekt	5
2.2.3. Implementierung	5
3. Ladeverhalten	9
3.1. Energiespeicher	9
3.1.1. Bleisäure Akkumulatoren	10
3.1.2. Nickel Akkumulatoren	10
3.1.3. Lithium-Ionen Akkumulatoren	11
3.2. Ladeverfahren	12
3.3. Beispielmessungen	13
3.4. Algorithmus Entwicklung	15
3.5. Fazit	18
4. Anforderungsanalyse	21
4.1. Funktionale Eigenschaften	21
4.2. Nicht funktionale Eigenschaften	23
4.3. Übersicht	24
5. Konzeptionierung	27
5.1. Grundlegende Designentscheidungen	27
5.2. Nachrichten	28

5.3. GenericNode	29
5.4. Lade- und Wartepplätze	30
5.5. Verbindung mit Multicopter	31
5.6. Realistisches Ladeverhalten	31
5.7. Reservierungen und Vorhersagen	33
5.8. Anfrage für Multicopter	36
5.9. Nutzungsstatistiken	36
5.10. Ressourcenverwaltung	37
5.11. Schnelle Teilaufladung priorisieren	38
5.12. Übersicht	39
6. Implementierung	43
6.1. Ankommende Multicopter	43
6.2. Aktualisierung der Ladestation	47
6.2.1. Ladeplätze besetzen	48
6.2.2. Ladeplätze aufladen	51
6.2.3. Ladeplätze leeren	53
6.2.4. Ladeplätze neu verteilen	54
6.3. Auswertung Anforderungen	56
7. Fazit und Ausblick	59
A. Anhang	I
CD Struktur	VI
Tabellenverzeichnis	VII
Abbildungsverzeichnis	VIII
List of Source Code	IX
Literaturverzeichnis	X

1. Einleitung

/todojede Art von intelligenter Aufladung - umformulieren

Ladegeräte sind allgegenwärtig. Jeder Laptop und jedes Smartphone müssen regelmäßig aufgeladen werden. Solange jedes Gerät sein eigenes Ladegerät hat, scheint jede Art von intelligenter Aufladung unnütz. Doch sobald man sich vorstellt, dass das Smartphoneladegerät mit zehn anderen Smartphonebesitzern geteilt werden soll, wird die Relevanz eines organisierten Ladeprozesses deutlich. Analog verhält es sich mit Ladestationen für Multicopter. Zusätzlich ist das Verhältnis von Nutzungsdauer zu Ladezeit bedeutend kleiner, verglichen mit einem modernen Smartphone. Der Ladeprozess ist dadurch ein größerer und wichtigerer Bestandteil des Multicopters. Darüber hinaus wird nicht nur ein einzelnes Ladegerät betrachtet, sondern unterschiedlich viele Ladestationen an unterschiedlichen geographischen Orten innerhalb der Simulation. Einfache Ladegeräte können nicht kommunizieren. Dadurch können Informationen die zur Verbesserung des Ladeprozesses beitragen nicht geteilt werden. Die aktuelle geschätzte Wartezeit oder gar eine Prognose für eine vollständige Aufladung könnten dem Multicopter bei der Auswahl der passenden Ladestation behilflich sein. Simulationsmodelle sind Abbilder der Realität. Sie werden erstellt um Rückschlüsse auf die modellierte Realität zu ermöglichen. Der Ladealgorithmus trägt zur Übertragbarkeit der gewonnen Informationen bei und sollte deshalb möglichst realistisch gestaltet sein.

Diese Bachelorarbeit hat das Ziel einen organisierten und intelligenten Ladeprozess in eine Simulation mit einer Vielzahl von Multicoptern und einer Vielzahl von Ladestationen zu integrieren. Kernproblem dabei ist die Aggregation, Aufbereitung und Weitergabe von Informationen die den Ladeprozess betreffen. Dadurch sollen Verbesserungspotentiale aufgedeckt werden, aus denen Rückschlüsse für Realweltprojekte gezogen werden können.

2. Grundlagen

Im folgenden Kapitel wird auf die nötigen Grundlagen und Hintergründe eingegangen. Die Ladestation wird in ein bestehendes Projekt eingepflegt. Dieses bestehende Projekt beschäftigt sich mit der Simulation von Missionen für unbemannte Luftfahrzeuge. Einleitend wird das angewendete Verständnis eben jener Luftfahrzeuge erläutert. Im Anschluss wird die Projektumgebung grob skizziert.

2.1. Unbemannte Luftfahrzeuge

Ein Unbemanntes Luftfahrzeug (engl. Unmanned aerial vehicle, UAV) ist ein Luftfahrzeug, welches ohne Besatzung an Bord betrieben wird. Für die Steuerung können Fernsteuersysteme oder an Bord befindliche Computer zum Einsatz kommen. [ICA11] Der von der ICAO definierte Begriff schränkt wenig ein. Abgesehen vom Ausschluss von Flugmodellen für Freizeit- und Luftsportaktivitäten werden keine Flugobjekte ausgeschlossen. Sowohl moralisch umstrittene militärische Drohnen, als auch die Prototypen der automatisierten Paketzustellung fallen unter die Definition. Im Rahmen des bestehenden Projektes wird ein UAV als Multicopter, der zu zivilen Zwecken eingesetzt wird, verstanden.

2.1.1. Multicopter

Multicopter sind Flugobjekte mit mehreren Rotoren, die für Auftrieb sorgen. Einer der bekannten Vertreter ist der Quadrocopter (mit vier Rotoren), darüber hinaus werden unter anderem auch Tricopter, Hexacopter und Octocopter eingesetzt. Die Anzahl der Rotoren beeinflussen in erster Linie die Stabilität in der Luft. Darüber hinaus können

Ausfälle einzelner Rotoren bei größerer Anzahl ausgeglichen werden. Eine höhere Rotorenanzahl birgt nicht nur Vorteile. Das Fluggewicht und die nötige Leistung steigen. Daraus folgt, dass entweder größere Energiespeicher nötig sind, welche das Gewicht weiter erhöhen würden oder die Flugdauer mit steigender Rotorenanzahl abnimmt.

Seit einigen Jahren finden UAVs auch in privaten Bereichen Einsatzmöglichkeiten. Sie werden als Spiel- und Spaßwerkzeug zum Beispiel in Verbindung mit Augmented Reality (AR) verwendet. Dafür wird unter anderem eine einfache Anwendung benötigt. Die Geräte müssen nach dem Auspacken, Zusammenbauen und Aufladen ohne Fachkenntnisse benutzbar sein. [BCVP11, S. 1477] Diese Vereinfachung für Endanwender wird durch Bordelektronik, die Steuerung und Stabilisierung ermöglicht.

Die Rotoren des Multicopters sind üblicherweise an einen Elektromotor angeschlossen. Die benötigte Energie wird von einem Energiespeicher geliefert. Hier kommen nahezu ausschließlich Lithium-Polymer-Akkumulatoren zum Einsatz, weitere Details dazu können im Kapitel 3 nachgelesen werden. [Dan13]

2.2. Projektumgebung

Dieses Kapitel stellt die Projektumgebung dar. Zu Beginn wird auf die verwendete Programmiersprache und das benutzte Framework eingegangen. Anschließend wird das bestehende Projekt abseits der zu implementierenden Ladestation vorgestellt.

2.2.1. C++ und Framework OMNeT++

C++ ist eine Erweiterung der Programmiersprache C. Hinzugefügt werden Eigenschaften, die das objektorientierte Paradigma ermöglichen. Durch den Sprachursprung bleibt C++ effizient und mächtig. Beispielsweise eine automatische Speicherbereinigung findet nicht statt. Der Anwender hat alles in der Hand. Dessen ungeachtet kann die Abstraktion der Objektorientierung eingesetzt werden, um übersichtliche und wartbare Software zu schreiben.

Frameworks werden eingesetzt, um den Entwicklungsaufwand zu reduzieren. Dies geschieht unter anderem durch die Bereitstellung von häufig genutzten Funktionen für den jeweiligen Anwendungsbereich. Das im Projekt eingesetzte Framework nennt sich OMNeT++, es basiert auf der Programmiersprache C++. Es beschreibt sich selbst als „extensible, modular, component-based C++ simulation library and framework primarily for building network simulators.“[Anda]. Die Nutzung ist für wissenschaftliche Arbeiten gebührenfrei [Andb]. Die angebotene Entwicklungsumgebung, eine Tutorialreihe und die online Dokumentation vereinfachen den Einstieg. Das Framework ermöglicht die Erstellung von Objekten, welche miteinander und der Umwelt interagieren können. Darüber hinaus bietet es eine optionale grafische Darstellung, welche den Ablauf insbesondere für Außenstehende veranschaulicht.

2.2.2. Projekt

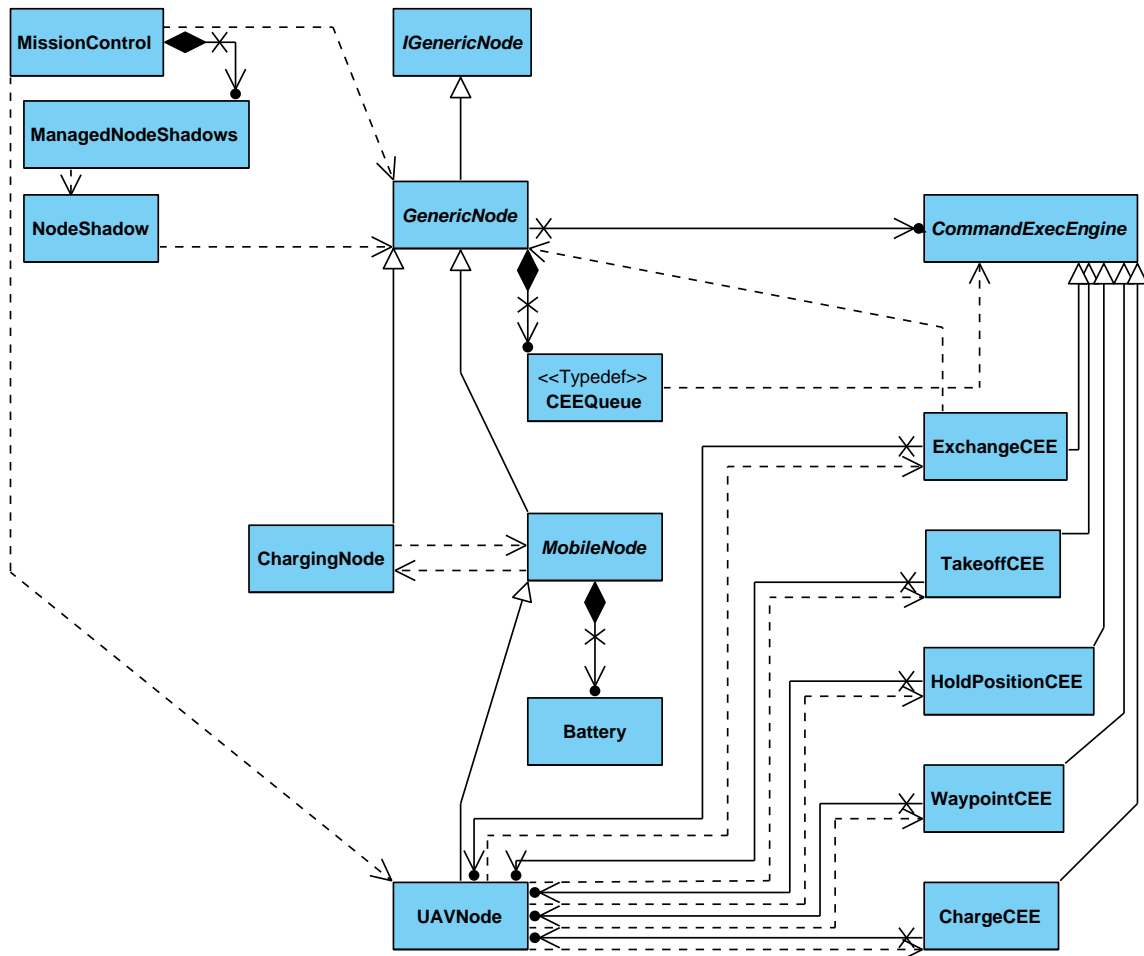
Die Arbeit baut auf ein bestehendes Projekt auf. In diesem Projekt werden Missionen für Multicopter simuliert. Die Missionen setzen sich aus einzelnen Kommandos zusammen, welche in der gleichen Form auch in der realen Welt genutzt werden können. Eine einfache Mission könnte aus einer Abfolge von mehreren Bewegungskommandos, die unendlich wiederholt werden sollen, bestehen. Im Laufe der Mission entleert sich der Akkumulator des eingesetzten Multicopters. Vor jedem Kommando führt der Multicopter eine Ressourcenkalkulation durch und prüft, ob die Energie noch für das nächste Kommando und einen anschließenden Flug zur nächstgelegenen Ladestation reicht. Falls dieser Test ein negatives Resultat liefert, wird das Flugobjekt ausgetauscht und das vorher im Einsatz befindliche fliegt zur nächsten Ladestation. Ein komplexeres Szenario könnte mehrere simultan genutzte UAVs enthalten und zusätzlich eine Begrenzung der vorhandenen Ersatz-UAVs.

2.2.3. Implementierung

Das Projekt nutzt die objektorientierten Möglichkeiten der Sprache C++. Im folgenden Klassendiagramm wird ein Überblick über die vorhandene Struktur zu Beginn der

Arbeit gezeigt. Wie in vielen Softwareprojekten wird nicht ein Teil nach dem anderen entwickelt, sondern unterschiedliche Personen arbeiten an unterschiedlichen Teilen des Projektes. Während der Arbeit hat sich auch die Struktur des restlichen Projektes stetig weiterentwickelt. In der Abbildung 2.1 wird das Klassendiagramm zum Projektstart aufgezeigt. Das gezeigte Klassendiagramm ist nicht vollständig, sondern soll den grundsätzlichen Aufbau verdeutlichen. Die zentrale Organisationseinheit der Simulation befindet sich innerhalb der MissionControl Klasse. Diese verwaltet über Abbilder NodeShadow's die Objekte innerhalb der Simulation. Die einzelnen Simulationsobjekte sind in einer deutlichen Hierarchie angeordnet. Die GenericNode implementiert das IGenericNode Interface und stellt die Grundlage für die Darstellung und Arbeit innerhalb der Simulation zur Verfügung. Außerdem wird hier die Kommandoliste CEEQueue an das Objekt gebunden. Die bestehende ChargingNode baut auf der GenericNode auf. Vor Projektstart war die ChargingNode nur eine leere Hülle ohne Implementierungen. Die MobileNode erbt ebenfalls von der GenericNode. Zusätzlich wird ein Energiespeicher Battery eingebunden. Die letzte Stufe der Hierarchie ist die UAVNode, welche von der MobileNode erbt. Diese repräsentiert einen Multicopter und wird für die Ausführung der Kommandos in den CommandExecEngine's benötigt.

Abbildung 2.1. – Projektstart UML-Klassendiagramm



3. Ladeverhalten

Im folgenden Kapitel wird das Ladeverhalten der Ladestation entworfen. Begonnen wird mit dem Vergleich mehrerer Energiespeicher. Dieser Vergleich soll herausarbeiten, aus welchen Gründen in mobilen Anwendungen vorzugsweise Lithium-Ionen-basierte Akkumulatoren eingesetzt werden. Darauf aufbauend werden übliche Ladeverfahren und deren Spezifika untersucht. Nach dieser von Literatur angetriebenen Untersuchung, werden Beispielwerte eines Ladegerätes und Akkumulators gemessen. Abschließend wird ein Fazit gezogen, welches die Grundlage für den zu implementierenden Ladealgorithmus liefert.

3.1. Energiespeicher

Im einleitenden Kapitel über Multicopter [2.1.1](#), wurde festgestellt, dass diese üblicherweise von Elektromotoren angetrieben werden. Elektromotoren benötigen eine elektrische Energiezufuhr, welche von Akkumulatoren bereit gestellt wird.

Akkumulatortypen lassen sich in Gruppen basierend auf ihren chemischen Bestandteilen einteilen. Drei häufig verwendete Typen basieren auf Lithium-Ionen, Nickel und Bleisäure. Durch die gleiche Basis haben Akkumulatoren innerhalb einer Gruppe ähnliche Merkmale. Für die Auswahl eines Energielieferanten sind neben den funktionalen Eigenschaften Energiedichte (Wh/kg), Kapazität (Ah), Spannung pro Zelle (V) auch weitere nicht funktionale Eigenschaften wie Langlebigkeit, Sicherheit und Ladegeschwindigkeit relevant. In der Anwendung sind die wirtschaftlichen Aspekte ebenfalls ein relevanter Faktor. Für die rein wissenschaftliche Betrachtung ergeben sich hier viele Probleme. Unterschiedliche Marktpreise basieren nicht zwingend auf der Wertigkeit der verwendeten Güter, sondern können auch durch Skaleneffekte der Massenproduktion, Wettbewerb oder Gesetze bezüglich der Technologie entstehen. Diese Effekte

wiederum basieren unter anderem auf der Nachfrage nach dem Produkt und damit auf der Bekanntheit. Dessen ungeachtet wird die Ladestation für aktuelle Multicopter konzeptioniert.

3.1.1. Bleisäure Akkumulatoren

In einem Bleisäure Akkumulator besteht die Elektrode aus Blei und Bleioxid, der dazugehörige Elektrolyt besteht aus verdünnter Schwefelsäure. Dieser Akkumulatortype gilt als der älteste Entwurf und reicht bis in das 19. Jahrhundert zurück. Die Nennspannung einer Zelle beträgt 2 V und ist damit ebenso wie die Energiedichte mit 20-50 Wh/kg vergleichsweise gering. Je nach Qualität kann eine Lebensdauer von 200-2000 Ladezyklen erwartet werden. [CS16, 1] Durch die lange Bekanntheit der Technologie, sind diese Akkumulatoren relativ günstig.

Für den Einsatz in Multicoptern sind diese Akkumulatoren aufgrund der geringen Energiedichte ungeeignet. Einsatzgebiete sind in erster Linie stationäre Anwendungen, für die Energiedichte und Spannung weniger relevant sind oder Anwendungen in denen Robustheit und Sicherheit einen hohen Stellenwert haben, beispielsweise Autobatterien.

3.1.2. Nickel Akkumulatoren

In dieser Gruppe gibt es vor allem zwei relevante Ausprägungen, Nickel-Cadmium und Nickel-Metalhydrid. Nickel-Cadmium Akkumulatoren sind die älteren und in der Energiedichte und Lebensdauer unterlegen. Darüber hinaus ist Cadmium ein giftiger Stoff und muss entsprechend sorgfältig behandelt werden. Der einzige Vorteil gegenüber Nickel-Metalhydrid ist die geringere Selbstentladung, welche für die Anwendung in Multicoptern von untergeordneter Relevanz ist. Nickel-Metalhydrid Akkumulatoren haben eine Energiedichte von 60-80 Wh/kg und Nennspannung von 1,2 V. Die Lebensdauer beträgt weniger als 3000 Ladezyklen. Nachteile sind vor allem die hohe Selbstentladung, die im Vergleich zu Bleiakkumulatoren höheren Kosten und der Memory-Effekt, welcher bei Teilentladungen zu einer kleineren nutzbaren Kapazität führt. [CS16, 1]

Um das Jahr 2000 waren diese Akkumulatoren die Wahl für mobile Anwendungen. Ältere Multicopter könnten daher auf Nickel Akkumulatoren basiert haben, inzwischen wurden diese jedoch durch Lithium-Ionen basierte Akkumulatoren ersetzt.

3.1.3. Lithium-Ionen Akkumulatoren

In der Regel kommen in mobilen Anwendungen aktuell Lithium-Ionen Akkumulatoren zum Einsatz. Elektroautos, Handys und auch Multicopter werden von ihnen mit Energie versorgt. Innerhalb dieser Akkumulatorgruppe gibt es eine Vielzahl von eingesetzten Verbindungen. Die für Elektromobilität genutzten Akkumulatoren sind meistens Lithium-Polymer-Akkumulatoren. Diese haben eine Energiedichte von 200 Wh/kg, eine Spannung von 3,8 V und sie können mehr als 1000 Ladezyklen durchleben. Darüber hinaus gibt es bei diesem Batterietyp keinen Memory-Effekt. Doch neben den offensichtlichen Vorteilen existieren auch einige Nachteile beim Einsatz von Lithium-Polymer-Akkumulatoren. Der Akkumulator reagiert sensibler während des Ladeprozesses, eine zu starke Auf- oder Entladung sind schädlich und verringern die Lebensdauer dramatisch. Für einen reibungslosen Ablauf muss der Ladezustand bekannt sein und die einzelnen Zellen (wenn mehrere Zellen zum Einsatz kommen) müssen gleichmäßig aufgeladen werden. Darüber hinaus sind diese Akkumulatoren die teuerste Variante, in erster Linie wegen der vergleichsweise neuen Technologie und der begrenzten zur Verfügung stehenden Menge von Lithium. [CS16, 2]

Trotz Betrachtung der Nachteile sind die hohe Energiedichte und die hohe Spannung ausschlaggebend für den Einsatz in mobilen Anwendungen. Im Rahmen von Elektromobilität stehen diese Akkumulatoren im Zentrum vieler Forschungen. Neben der Verbesserung der bestehenden Verbindungen wird auch mit neuen Batterietypen experimentiert. Ein Beispiel für diese Forschung sind Lithium-Sauerstoff Akkumulatoren, welche eine zehnfach höhere Energiedichte versprechen, aber wenigstens bisher durch ihre kurze Lebensdauer von 50 Zyklen nicht eingesetzt werden. [CS16, 2]

3.2. Ladeverfahren

Das Ladeverfahren mit dem ein Akkumulator aufgeladen wird beeinflusst diesen maßgebend. Im Bereich der Elektromobilität ist die Ladezeit essentiell. Elektroautos werden unnütz, wenn die tägliche Aufladung den Bedarf für beispielsweise den Arbeitsweg nicht abdecken kann. Im Fall von Multicoptern die eine Mission erfüllen sollen, ist die Aufladegeschwindigkeit ausschlaggebend für die Menge der benötigten Fluggeräte. Unter Annahme begrenzter zur Verfügung stehender Ressourcen (Multicopter), wird durch die Ladegeschwindigkeit die Maximalleistung des Schwarms beeinflusst.

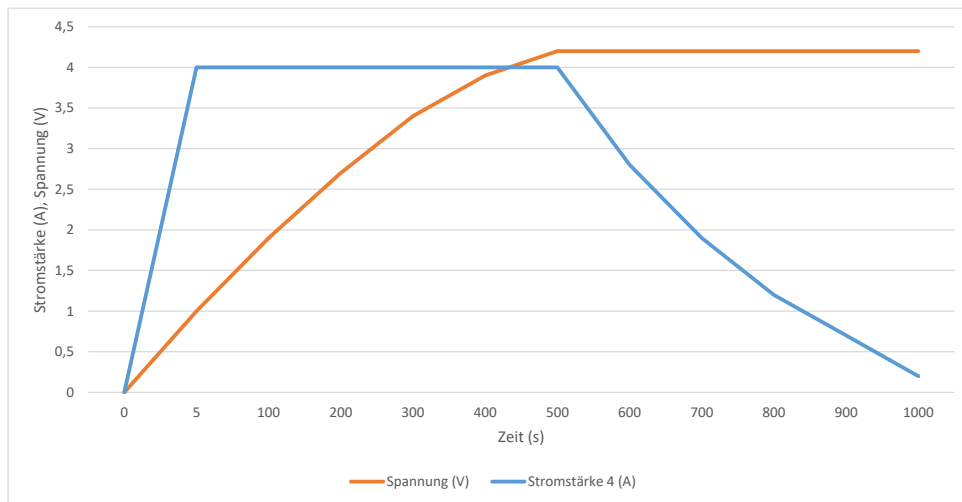
Lithium-Ionen Akkumulatoren reagieren sensibel auf Überladung oder zu hohe Spannung. Die chemischen Bestandteile werden beschädigt und die Leistung des Energiespeichers fällt ab. Ein einfacher Ladevorgang mit einer konstanten Stromstärke kommt deshalb nicht in Frage. Je näher der Füllstand der Kapazität des Akkumulators kommt, desto größer wird die Spannung. Lithium-Polymer Akkumulatoren haben üblicherweise eine Maximalspannung von 4,2 V, eine Überschreitung verursacht Schäden und reduziert die Lebensdauer dramatisch. [KPNC16, 381]

Das eingesetzte Ladeverfahren wird als Constant Current Constant Voltage (CCCV) bezeichnet. Es strebt eine möglichst hohe Ladegeschwindigkeit bei weiterhin hoher Lebensdauer des Akkumulators an. Dieses Verfahren besteht aus zwei Phasen. Der beispielhafte Verlauf von Stromstärke und Spannung über die Zeit, kann in der Abbildung 3.1 betrachtet werden. Die konkreten Werte sind erfunden, das Diagramm soll lediglich den üblichen Verlauf visualisieren. Konkrete Messungen der Kapazität über die Zeit können im folgenden Kapitel 3.3 gefunden werden. In der ersten Phase (CC) wird der leere Akkumulator mit einer konstanten Stromstärke geladen. Im Verlauf des Ladevorgangs steigt die Spannung an. Die Füllmenge des Akkumulators steigt linear über die Zeit an. Der Übergang zur zweiten Phase (CV) findet statt, sobald die Maximalspannung erreicht ist. Zu diesem Zeitpunkt ist der Akkumulator mit 75-80% seiner Maximalkapazität geladen. [HCH99, 1] [YXQX10, 1]

In der zweiten Phase bleibt die Spannung konstant beim Maximalwert und die Stromstärke nimmt ab. Daraus folgt, dass die Ladegeschwindigkeit nun nicht mehr linear verläuft, sondern immer weiter abfällt, je stärker der Akkumulator geladen ist. Phase zwei endet üblicherweise sobald eine festgelegte minimale Stromstärke erreicht wurde.

Der Akkumulator ist dann abhängig vom festgelegten Minimalwert gefüllt. [KPNC16, 381]

Abbildung 3.1. – Beispielverlauf CCCV Stromstärke und Spannung



3.3. Beispielmessungen

Die Ausführungen aus der Literatur geben einen Überblick über die Funktionsweise der Akkumulatoren und ihre Aufladung. Mit Hilfe der Beispielmessungen sollen Werte gefunden werden, die den Ladeverlauf realistisch abbilden. Für eine statistisch fundiertes Ladeverhalten müssten die Messungen erweitert werden. Die neun Testdatensätze weisen eine Reihe von Unsicherheiten auf. Es wurde nur ein Ladegerät und ein Akkumulator verwendet. Effekte die von einer verschiedenen Gesamtkapazität abhängen können deshalb nur vermutet und nicht nachgewiesen werden. Die geringe Anzahl der Versuche kann die Schwankungen der Testergebnisse nicht ausgleichen.

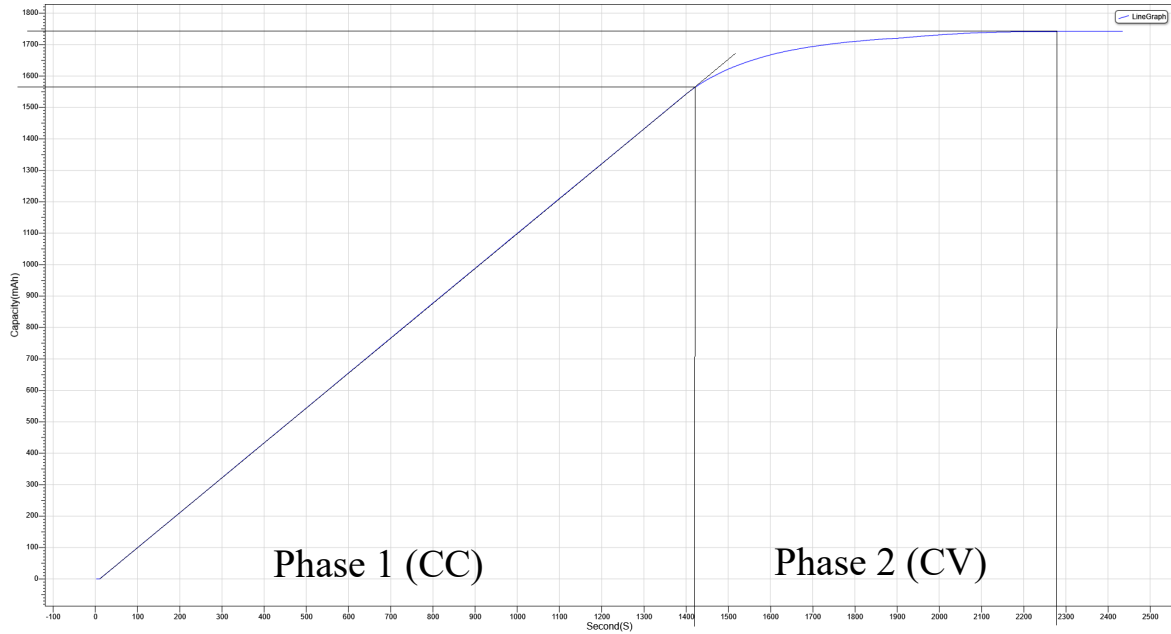
Für die Messungen wurde ein Turnigy Accucell 6 Ladegerät verwendet, welches das CCCV Verfahren benutzt, um unter anderem Lithium-Polymer-Akkumulatoren aufzuladen. Als Testakkumulator wird ein bereits für Multicopterflüge gebrauchter Akkumulator Zippy Compact mit einer Maximalkapazität von 1800 mAh verwendet. Der

Akkumulator wurde vor jedem Ladevorgang mit dem Ladegerät im Modus Discharge mit 1 A entladen. Für den Ladevorgang wurde der Akkumulator mit einer Stromstärke von 1 A, 2 A und 4 A geladen. Dieser Ablauf wurde dreimal wiederholt um die Wahrscheinlichkeit für zufällige Erscheinungen zu reduzieren. Das Ladegerät bietet eine Computerschnittstelle und kann den Ladeverlauf aufzeichnen. Aus den Diagrammen wurden der Endpunkt der ersten Phase und der Endpunkt des Ladevorgangs abgelesen, beispielhaft aufgezeigt in 3.2. Aufgrund der begrenzten Qualität der aufgezeichneten Diagramme sind die Kapazitätswerte auf Vielfache von 5 gerundet, die Zeit ist abhängig von der Skalierung gerundet. Neben den abgelesenen Werte können der folgenden Tabelle 3.1 die Steigung der ersten Phase, die Dauer der zweiten Phase und das Verhältnis von beiden Phasen bezüglich der Zeit und der Kapazität entnommen werden. Die Darstellung der Berechnungen sind gerundet, für die weitere Verwendung werden jedoch die exakten Ergebnisse verwendet. Die von der Ladegerätsoftware erstellten Diagramme können dem Anhang A entnommen werden. Die genauen Werte und die Diagramme der Ladevorgänge können dem Anhang entnommen werden. Darüber hinaus befinden sich die Diagramme mit den eingezeichneten Ablesegeraden und die Excel-Datei, in welcher die Berechnungen für die nachfolgenden Formeln angefertigt wurden, auf der beiliegenden CD.

Tabelle 3.1. – Ergebnisse Beispielmessungen

Ladestrom (A)	1	1	1	2	2	2	4	4	4
Ende Phase 1 (s)	5848	5882	5649	2862	2996	2833	1410	1421	1361
Ende Phase 1 (mAh)	1620	1620	1560	1580	1660	1560	1550	1565	1500
Ende Ladevorgang (s)	6330	6413	6031	3589	3723	3589	2276	2277	2175
Ende Ladevorgang (mAh)	1670	1685	1600	1700	1765	1660	1765	1740	1700
Steigung Phase 1 (mAh/s)	0,28	0,28	0,28	0,55	0,55	0,55	1,10	1,10	1,10
Dauer Phase 2 (s)	482	531	382	727	727	756	866	856	814
Phase 1 / Phase 2 (s/s)	0,92	0,92	0,94	0,80	0,80	0,79	0,62	0,62	0,63
Phase 1 / Phase 2 (mAh/mAh)	0,97	0,96	0,98	0,93	0,94	0,94	0,88	0,90	0,88

Abbildung 3.2. – Messung 4A Ladestrom mit Phasenkennzeichnung



3.4. Algorithmus Entwicklung

Für die Implementierung muss der Ladeverlauf modelliert werden. Relevant ist die Kapazität abgetragen über die Zeit. Im folgenden Abschnitt wird die Entwicklung des Algorithmus für den Ladevorgang präsentiert. Nachdem für beide Phasen eine geeignete Approximation gefunden wurde, werden die errechneten Werte mit den gemessenen Werten verglichen, um die Genauigkeit sichtbar zu machen.

Die erste Phase lässt sich durch eine lineare Funktion beschreiben. Die errechneten Werte für die Steigung schwanken minimal in der 3. Nachkommastelle. Die Kapazität C (in mAh) hängt von der Ladestromstärke I (in A) und der Ladezeit t (in s) ab.

$$C(t) = 0,2754 * I * t \quad (3.1)$$

Darüber hinaus muss der Endzeitpunkt t_{E1} der ersten Phase bestimmt werden. Dieser hängt von I und der Gesamtkapazität der Akkumulators C_{max} ab. Die Abhängigkeit

von der Stromstärke wird in 3.1 deutlich. Der Ladestrom fällt nach dem Wechsel in die zweite Phase ab. Ein niedrigerer Ladestrom führt dazu, dass die zweite Phase bei einer höheren relativen Kapazität erreicht wird, dessen ungeachtet verlängert sich der Ladevorgang.

$$\begin{aligned} C(t_{E1}) &= (-0,0291 * I + 1) * C_{max} \\ t_{E1} &= (-0,0291 * I + 1) * \frac{C_{max}}{0,2754 * I} \end{aligned} \quad (3.2)$$

Die zweite Phase wird unabhängig von der ersten Phase betrachtet, deshalb startet diese ebenfalls bei $t = 0$. Ihr Verlauf kann wie in 3.2 gezeigt durch eine nach oben begrenzte Wachstumsfunktion modelliert werden. Damit das Maximum erreicht werden kann, wird C_{max} mit dem selbst gewählten Wert a multipliziert. Der Wert für a wurde dabei über Testeinsetzungen ermittelt und so gewählt, dass die Ergebnisse für die Aufladung mit 2 A und 4 A möglichst genau sind.

$$\begin{aligned} a &= 0,0013 \\ C(t) &= C_{max} * (a + 1) - (C_{max} * (a + 1) - B_0) * e^{-kt} \end{aligned} \quad (3.3)$$

B_0 ist die Startkapazität, welche dem Ende von Phase 1 entspricht.

$$B_0 = C(t_{E1}) = (-0,0291 * I + 1) * C_{max} \quad (3.4)$$

k ist die Wachstumskonstante, welche von der linearen Steigung, dem Startzeitpunkt der zweiten Phase und a abhängt.

$$k = \frac{0,2754 * I}{C_{max} * (a + 0,0291 * I)} \quad (3.5)$$

Für die zweite Phase muss ebenfalls ein Endzeitpunkt bestimmt werden, welcher die Dauer der zweiten Phase angibt. Dieser Zeitpunkt ist erreicht wenn die Kapazität der Gesamtkapazität entspricht.

$$\begin{aligned} C(t_{E2}) &= C_{max} \\ t_{E2} &=? \end{aligned} \tag{3.6}$$

Abschließend folgt eine Tabelle 3.2 in der die gemessene Ladedauer, die berechnete Ladedauer und die Abweichung der beiden Phasen dargestellt wird. Zu sehen ist, dass die lineare Annäherung geringere Abweichungen aufweist. Zusätzlich ist die Dauer der ersten Phase länger, so dass die relativen Abweichungen noch geringer werden. Die maximale relative Abweichung tritt beim zweiten Versuch der 4 A Ladung auf und beträgt weniger als 2%. Die Abweichungen für Phase zwei sind bedeutend größer. Außerdem sieht man eine tendenziell abfallende Abweichung für wachsenden Ladestrom. Dies ist die Auswirkung des gewählten Parameters a . Dieser wurde unter Berücksichtigung einer abwärts Gewichtung des Ladestroms ausgewählt. Diese Gewichtung lässt sich auf die praktische Nutzung zurückführen. Der Ladestrom liegt üblicherweise zwischen 2 A und 4 A, weil die Ladedauer bei geringeren Strömen stark ansteigt und die Lebenszeit des Akkumulators nur geringfügig erweitert wird. Noch höhere Ladeströme führen zu einer deutlich verringerten Lebensdauer oder einem Totalschaden [Dan13, 50]. Dessen ungeachtet sind die relativen Abweichungen höher. Abgesehen von der 1 A Versuchsreihe weist der zweite Versuch einer Ladung mit 2 A eine relative Abweichung von 9% auf.

Tabelle 3.2. – Vergleich Messungen und Formel ($a = 0,0013$)

Ladestrom (A)	1	1	1	2	2	2	4	4	4
Dauer Phase 1 (s)	5848	5882	5649	2862	2996	2833	1410	1421	1361
Dauer Phase 1 Formel (s)	5887	5940	5641	2907	3018	2838	1416	1396	1364
Abweichung	-39	-58	8	-45	-22	-5	-6	25	-3
Dauer Phase 2 (s)	581	586	557	702	729	686	850	838	818
Dauer Phase 2 Formel (s)	482	531	382	727	727	756	866	856	814
Abweichung	-99	-55	-175	25	-2	70	16	18	-4
Gesamt (s)	6330	6413	6031	3589	3723	3589	2276	2277	2175
Gesamt Formel (s)	6469	6527	6197	3609	3747	3524	2265	2233	2182
Abweichung	-139	-114	-166	-20	-24	65	11	44	-7

3.5. Fazit

Die Literatur und Testmessungen bringen ähnliche Ergebnisse zum Vorschein. Es wird deutlich, dass der Ladevorgang in der ersten Phase linear verläuft und abhängig von der verwendeten Stromstärke ist. Die Dauer dieser Phase kann durch eine höhere Ladestromstärke proportional verringert werden. Die Kapazität des Akkumulators die am Ende der ersten Phase erreicht wurde sinkt mit steigender Stromstärke. In den Testreihen lag der Start zwischen 88% und 97%. In der Literatur werden Werte zwischen 75% und 80% für den Phasenwechsel angegeben [HCH99, 1] [YXQX10, 1]. Diese Werte würden nach dem angenommenen linearen Zusammenhang bei 8 A Ladestrom auftreten. Möglicherweise wurde bei diesen Angaben eine ähnliche Größenordnung verwendet.

Darüber hinaus könnten auch die Akkumulatorbauarten die genauen Werte beeinflussen. Wenn beispielsweise der mit Nennwert 1800 mAh verwendete Akkumulator eine größere Kapazität hat und diese nur nicht aufgeladen wird, um den Akkumulator zu schonen, könnte sich der relative Wert verschieben. Zusammengefasst sind die Werte für den Phasenwechsel ungenau, können aber im Zusammenhang mit den Literaturangaben erklärt werden.

Die eigenen Beispielwerte enthalten eine Reihe von Unsicherheiten. Für einen statistisch nachweisbaren Algorithmus müsste die Testreihe stark ausgebaut werden. Das Spektrum der verwendeten Akkumulatoren, Ladegeräte, Ladeströme und Nennkapazitäten sollte erweitert werden. Dessen ungeachtet werden die ausgearbeiteten Formeln in die Simulation integriert. Gegenüber einer strikt linearen Aufladung implementiert der vorgestellte Algorithmus einen realistischeren Ladeverlauf. In der zweiten Phase nimmt die Menge der pro Zeiteinheit geladenen Kapazitätseinheiten ab. Je nach angelegtem Ladestrom kann so für einen relativ kleinen Kapazitätsanteil trotzdem ein großer relativer Zeitanteil nötig sein.

In der Implementierung wird der oben vorgestellte Algorithmus verwendet. Dennoch sollte die Genauigkeit in Frage gestellt werden und eine Verbesserung ist erstrebenswert.

4. Anforderungsanalyse

Alle Anforderungen werden natürlich sprachlich ausgedrückt und erhalten eine hervorgehobene Bezeichnung. Diese Bezeichnung wird in der tabellarischen Darstellung am Ende des Kapitels aufgegriffen, welche eine Zusammenfassung der Anforderungen darstellt. Jede Anforderung wird gemäß der MoSCoW-Priorisierung gewichtet [Int12].

Tabelle 4.1. – MoSCoW Akronym

M	MUST	Diese Anforderung muss erfüllt werden.
S	SHOULD	Diese Anforderung sollte erfüllt werden.
C	COULD	Diese Anforderung könnte erfüllt werden, falls erstens die höher priorisierten Anforderungen abgearbeitet sind und zweitens sie diese nicht negativ beeinflusst.
W	WON'T	Diese Anforderung wird verschoben und nicht in dieser Iteration der Entwicklung betrachtet. Diese Priorisierung wertet nicht die Wichtigkeit der Anforderung für das System, sondern lediglich die Wichtigkeit für die aktuelle Entwicklung. Eine solche Priorisierung erkennt die Relevanz der Anforderung an, verschiebt diese jedoch auf eine spätere Entwicklungsiteration.

4.1. Funktionale Eigenschaften

Ladeplätze: Jede Ladestation hat eine begrenzte Anzahl von Ladeplätzen, diese bestimmen wie viele Ladevorgänge gleichzeitig parallel ablaufen können. *Priorität MUST*

Warteplätze: Jede Ladestation hat unbegrenzt viele Warteplätze. Hier können die Multicopter während sie auf ihre Aufladung warten verbleiben und bereits aufgeladene Multicopter können hier bis zu ihrem nächsten Einsatz bleiben. Die Warteplätze sollen begrenzt werden können. Eine Überschreitung der Kapazität soll eine Fehlerinformation erzeugen. *Priorität MUST*

Verbindung mit Multicopter: Ein Multicopter soll sich mit einer Ladestation verbinden können. Nach der Verbindung kann der Multicopter ruhen und sein Akkumulator wird ohne weitere erforderliche Aktivität aufgeladen. *Priorität MUST*

Realistisches Ladeverhalten: Die Ladestation soll einen realistischen Ladealgorithmus implementieren. *Priorität MUST*

Reservierungen: Ein Multicopter kann einen Ladeplatz einer Ladestation vorzeitig reservieren. Die Abarbeitung der Ladeaufträge soll basierend auf dem Reservierungszeitpunkt erfolgen. *Priorität MUST*

Vorhersagen: Die Ladestation kann eine Aussage über den Ladevorgang treffen und diese Aussage mit anderen Netzwerkteilnehmern kommunizieren. Für Vorhersageanfragen wird unterschieden in zwei Fälle. Erstens soll der Zeitpunkt für einen abgeschlossenen Aufladevorgang abgefragt werden können. Es müssen ein prozentualer Start- und Zielwert berücksichtigt werden. Zweitens soll die Kapazität des Akkumulators, die bis zu einem Zielzeitpunkt aufgeladen werden kann, ausgegeben werden können. Die Vorhersage soll die, zum Zeitpunkt der Anfrage bestehenden, Reservierungen mit einbeziehen. Dadurch ist das Ergebnis einer Vorhersage immer als pessimistische Angabe zu verstehen. Falls andere Reservierungen nicht wahrgenommen werden, ist ein besseres Ergebnis möglich. *Priorität MUST*

Anfrage für Multicopter: Die Ladestation soll auf Ersatzanfragen für Multicopter reagieren können und einen zur Anfrage passenden Multicopter auswählen. In der Anfrage ist ein Mindestladezustand enthalten. Dieser muss so wenig wie möglich überschritten werden. *Priorität MUST*

Nutzungsstatistiken: Im Laufe der Simulation sollen anfallende Daten aggregiert und zur Verfügung gestellt werden. Insbesondere soll die Anzahl der erfolgreichen Ladevorgänge und die aufgeladene Energie erfasst werden. *Priorität MUST*

Ressourcenverwaltung: Die Ladestationen sollen über einen Energiespeicher verfügen, welcher als begrenzt oder unbegrenzt konfigurierbar sein soll. Eine versuchte Aufladung an einer Ladestation mit leerem, begrenztem Energiespeicher soll fehlschlagen und eine Fehlerinformation ausgeben. *Priorität SHOULD*

Schnelle Teilaufladung Priorisieren: Es soll eine Möglichkeit geben den schnellen Aufladeteil zu priorisieren. Für das CCCV Verfahren bedeutet das, dass erst alle Wartenden Multicopter bis zum Ende der linearen Phase aufgeladen werden, bevor die ersten Multicopter vollständig aufgeladen werden. *Priorität SHOULD*

Umverteilung von Multicoptern: Alle Ladestationen sollen untereinander kommunizieren und die zur Verfügung stehenden Multicopter aufteilen. Die Zahlenwerte für die Abgabe und Aufnahme von Multicoptern anderer Ladestationen soll konfiguriert werden können. *Priorität COULD*

Kooperation zwischen Ladestationen: Alle Ladestationen sollen miteinander kooperieren, um Kapazitätsengpässe auszugleichen. *Priorität COULD*

Mobile Ladestation: Jede Ladestation soll über die Fähigkeit zur eigenen Fortbewegung verfügen. *Priorität WON'T*

4.2. Nicht funktionale Eigenschaften

Codesprache: Im Quelltext darf ausschließlich die englische Sprache verwendet werden. *Priorität MUST*

Coderichtlinien: Die bestehenden Coderichtlinien, welche in der automatischen Formatierung der Entwicklungsumgebung hinterlegt sind, sollen eingehalten werden. *Priorität MUST*

Ladeplatzorganisation Zeitverzögerung: Die Organisation der Ladeplätze soll ohne Zeitverzögerung erfolgen. Die benötigte Zeit, um einen Multicopter an den Lademechanismus an- und abzuschließen soll vernachlässigt werden. *Priorität MUST*

Graphische Darstellung: Die Ladestationen sollen in das bestehende Projekt eingepflegt werden und in der graphischen Simulationsansicht angemessen dargestellt werden. *Priorität SHOULD*

4.3. Übersicht

Tabelle 4.2. – Übersicht Anforderungen

Bezeichnung	Funktional	Priorität
Ladeplätze	Ja	MUST
Warteplätze	Ja	MUST
Verbindung mit Multicopter	Ja	MUST
Realistisches Ladeverhalten	Ja	MUST
Reservierungen	Ja	MUST
Vorhersagen	Ja	MUST
Anfrage für Multicopter	Ja	MUST
Nutzungsstatistiken	Ja	MUST
Ressourcenverwaltung	Ja	SHOULD
Schnelle Teilaufladung Priorisieren	Ja	SHOULD
Umverteilung von Multicoptern	Ja	COULD
Kooperation zwischen Ladestationen	Ja	COULD
Mobile Ladestation	Ja	WON'T
Codesprache	Nein	MUST
Coderichtlinien	Nein	MUST
Ladeplatzorganisation Zeitverzögerung	Nein	MUST
Graphische Darstellung	Nein	SHOULD

5. Konzeptionierung

Im folgenden Kapitel wird die Konzeptionierung der intelligenten Ladestation dargestellt. Grundlage dafür sind die funktionalen Eigenschaften, sowie das bestehende Projekt. Die Darstellung des Entwicklungsprozesses wird durch UML-Klassendiagramme unterstützt. Dort werden die neu hinzugefügten Eigenschaften und Methoden aufgeführt, deren Entstehung im Text erläutert wird. Die Diagramme geben einen Überblick über die für das behandelte Thema relevanten Aspekte der Klassen. Am Ende des Kapitels wird ein Klassendiagramm, welches alle MUST und SHOULD Anforderungen abdeckt, präsentiert.

5.1. Grundlegende Designentscheidungen

Einige grundlegende Designentscheidungen werden durch das bestehende Projekt vorgegeben. Die Entwicklung ist auf ein Objektorientiertes Modell in der Programmiersprache C++ und das Simulationsframework OMNeT++ festgelegt. Konfigurationen werden aus einer zentralen Datei eingelesen und können während der Initialisierung der Objekte in Eigenschaften überführt werden.


Das Netzwerk der Simulationsobjekte wird in (.ned) Dateien definiert. Neben den Informationen für die grafische Darstellung werden hier auch die Verbindungen der einzelnen Elemente festgelegt. Bevor Netzwerkteilnehmer miteinander kommunizieren können muss eine Verbindung definiert werden. Die Ladestationen müssen mit den Multicoptern und den anderen Ladestationen kommunizieren können.

Objekteigenschaften sind im Sinne der Datenkapselung `protected` und je nach Bedarf werden Getter- und Setter-Methoden für den äußeren Zugriff hinzugefügt. Diese Me-

thoden sind in den Klassendiagrammen, insofern sie für den beschriebenen Ablauf nicht zwingend notwendig sind, nicht enthalten.

Im Rahmen der diskreten eventbasierten Simulationsumgebung muss die Ladestation auf ein Event mit einem Aktualisierungsvorgang reagieren. Für die graphische Simulation wird zusätzlich eine zeitbasierte Aktualisierung implementiert, so dass Statusmitteilungen ausgegeben werden können. Die Aktualisierungen erfolgen nach einer initialen Nachricht, anschließend läuft die Simulation solange bis ein Fehler auftritt oder ein manuelles Ende herbeigeführt wird. Die Grundlage für den Aktualisierungsvorgang befindet sich in der `GenericNode` und wird in der [Sektion 5.3](#) erläutert.

5.2. Nachrichten

Für die Kommunikation zwischen verschiedenen Netzwerkteilnehmern stellt OMNeT++ Nachrichten zur Verfügung. Nachrichten sind Objekte vom Typ `cMessage`. Ihnen kann eine Bezeichnung zugewiesen werden, was eine Unterscheidung ermöglicht. Darüber hinaus können eigene Nachrichtentypen in speziellen (`.msg`) Dateien definiert werden. Diesen können neben einer Bezeichnung noch individuelle weitere Informationen in Form von Parametern enthalten. Diese vereinfachten Klassendefinitionen werden während der Kompilierung in vollwertige C++ Klassen übersetzt, welche alle nötigen Eigenschaften und Methoden für den Versand beinhalten. Parameter können über Setter-Methoden angehängt und über Getter-Methoden ausgelesen werden.

Damit die Nachricht erfolgreich versendet werden kann muss außerdem der richtige Empfänger bekannt sein. Die definierten Verbindungen ([5.1](#)) werden in der Simulation mit In- und Outputgates organisiert. Ein Gate verbindet dabei immer genau zwei Netzwerkteilnehmer. Durch die Unterteilung in In- und Output können so auch einseitige Kommunikationswege abgebildet werden. Da die Auswahl des richtigen Gates für den Nachrichtenversand ein für alle Objekte relevantes Problem ist, wird eine Hilfsmethode in der `GenericNode` geplant. Diese Methode soll das passende Gate zu einem übergebenen Objekt im Netzwerk zurückgeben.

Die Nachrichten werden in den Objekten in der `handleMessage()` Methode verarbeitet. Basierend auf der Bezeichnung kann der Nachrichtentyp identifiziert werden. Nach dem

das Objekt der Basisklasse `cMessage` in ein Objekt der abgeleiteten Klasse überführt wurde, kann auf die individuellen Parameter zugegriffen werden.

Im Verlauf der Konzeptionierung werden verschiedene Nachrichtentypen hinzugefügt, um die Kommunikation mit der Ladestation zu ermöglichen. Diese werden nur in ihrer vereinfachten Form modelliert, da die Entwicklungsumgebung die Übersetzung automatisch übernimmt.

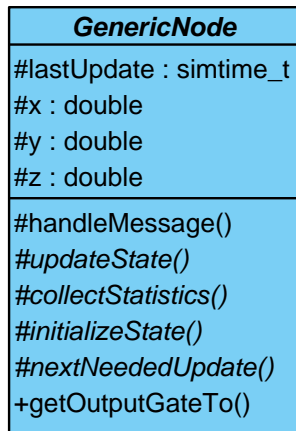
5.3. **GenericNode**

Die `GenericNode` stellt ein Grundgerüst für Objekte innerhalb der Simulation zur Verfügung. Die Klasse für die Ladestation `ChargingNode` wird von der `GenericNode` abgeleitet. Durch die Bereitstellung der im Diagramm 5.1 gezeigten Eigenschaften und Methoden, kann sich die Weiterentwicklung auf die relevanten Eigenschaften für die intelligente Ladestation fokussieren.

Nach der Instanziierung der Klasse wird die `initializeState()` Methode aufgerufen. Diese lädt die Konfigurationsdatei ein und übergibt die Werte für die Koordinaten `x`, `y`, `z` und die `timeStep` Eigenschaft. Anschließend wird das Objekt auf der Karte gemäß der angegebenen Koordinaten platziert. Darüber hinaus wird ein regelmäßiger Aufruf vorbereitet. Nach einer initialen Aktualisierungsnachricht wird die `updateState()` Methode aufgerufen. Hier werden die regelmäßigen Aufgaben definiert. Abschließend wird die Eigenschaft `lastUpdate` mit der aktuellen Zeit belegt und mit Hilfe der Methode `nextNeededUpdate()` eine neue Aktualisierungsnachricht generiert.

Die im Nachrichtenkapitel (5.2) genannte Hilfsmethode für den Nachrichtenversand `getOutputGateTo()` ist ebenfalls enthalten und wird im Diagramm 5.1 präsentiert.

Abbildung 5.1. – GenericNode UML-Klassendiagramm



5.4. Lade- und Warteplätze

Für die Lade- und Warteplätze muss die Anzahl der jeweiligen Plätze gespeichert werden. Darüber hinaus müssen Containerobjekte für die Multicopter vorhanden sein. Abgelegt werden Verweise auf die Objekte. Die Multicopter werden in der Simulation als UAVNode dargestellt. Deren Basisklasse ist die MobileNode, welche bereits den Energiespeicher beinhaltet. Der Akkumulator ist der für die Ladestation relevante Aspekt ist. Daher liegt es Nahe jede MobileNode als Konsumenten zuzulassen.

An der Ladestation können die Objekte in drei unterschiedliche Phasen sein. Das Objekt wartet auf seine Aufladung, es wird gerade aufgeladen und es wurde fertig aufgeladen. Um diese drei Zustände klar voneinander zu trennen werden drei ähnliche Listen konzeptioniert. Die Listen müssen dabei eine variable Größe haben können und Objekte müssen an jeder Stelle des Container hinzugefügt und entfernt werden können. Aus der C++ Bibliothek erfüllt unter anderem der Datentyp deque diese Anforderungen. Im Klassendiagramm 5.2 können diese unter den Bezeichnungen objectsWaiting, objectsCharging und objectsFinished gefunden werden. Die Liste mit den fertig geladenen Multicoptern unterscheidet sich dabei von den anderen beiden. In dieser werden direkt MobileNode Objekte gespeichert. Die Warte- und Ladeliste hingegen speichern Containerelemente vom Typ ChargingNodeSpotElement, welche weitere Informationen enthal-

ten. Der Grund dafür wird im Abschnitt über Reservierungen und Vorhersagen 5.7 erläutert.

Während die Anzahl der Plätze von externen Objekten einsehbar sein soll, müssen die Warte- und Ladeliste exklusiv der Ladestation zur Verfügung stellen. Folglich sind die Listen `protected` und es existieren keine Getter- oder Setter-Methoden für diese Eigenschaften.

Abbildung 5.2. – ChargingNode UML-Klassendiagramm

ChargingNode
#spotsWaiting : unsigned int
#spotsCharging : unsigned int
-objectsWaiting : deque<ChargingNodeSpotElement*>
-objectsCharging : deque<ChargingNodeSpotElement*>
-objectsFinished : deque<MobileNode*>
+getSpotsCharging()
+getSpotsWaiting()

5.5. Verbindung mit Multicopter

Der Anschluss eines Multicopters an die Ladestation benötigt eine Kommunikationsschnittstelle. Als Anfang wird dafür eine Nachricht konzeptioniert, die vom Multicopter an die Ladestation versendet wird, sobald dieser bereit für den Ladeprozess ist. Die Ladestation muss nach dem Empfang einer solchen Nachricht prüfen, ob die Koordinaten von Multicopter und Ladestation übereinstimmen und kann ihn dann zu den wartenden Objekten hinzufügen.

5.6. Realistisches Ladeverhalten

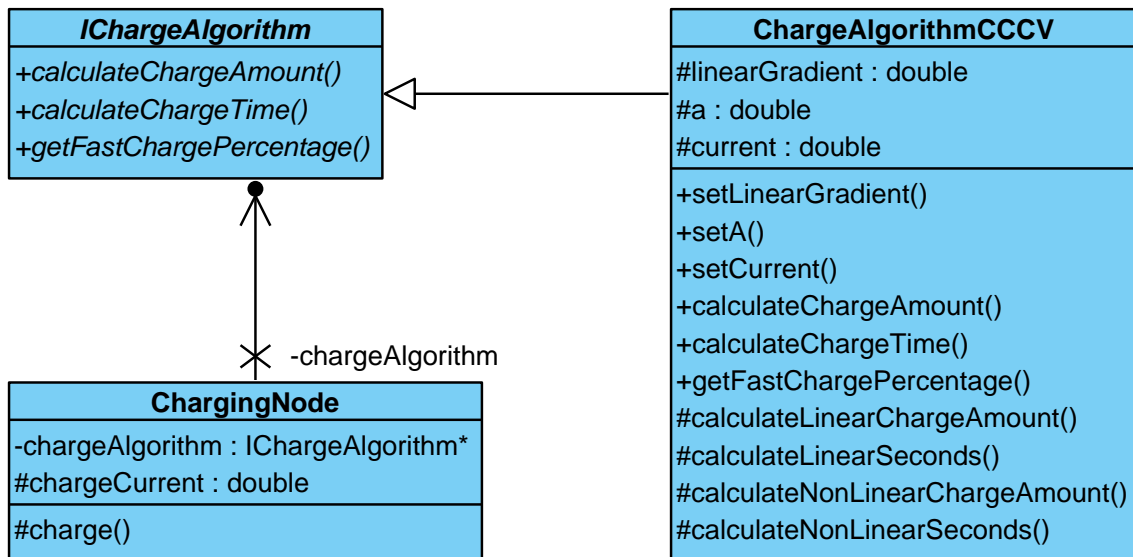
Zentrales Element einer Ladestation ist der zugehörige Ladealgorithmus. Im Kapitel über das Ladeverhalten 3 wurde bereits festgestellt, dass Akkumulatoren und die zugehörigen Ladealgorithmen zu aktuellen Forschungsthemen gehören. Während der

Konzeptionierung von Software ist es essentiell die Erweiterbarkeit und Austauschbarkeit der einzelnen Komponenten zu beachten. Ein Interface stellt eine Beschreibung für eine zukünftige Implementierung bereit. Es werden Methoden definiert, welche von den erbbenden Klassen zwingend implementiert werden müssen. Dadurch wird sichergestellt, dass die unterschiedlichen Implementierungen eines Interfaces untereinander kompatibel sind. Für den Ladealgorithmus werden ein Interface `IChargeAlgorithm` und eine Implementierung davon `ChargeAlgorithmCCCV` konzeptioniert. Der verwendete Ladealgorithmus wird während der Initialisierung spezifiziert und in eine Eigenschaft der Ladestation gespeichert. Für den Algorithmus relevante Eigenschaften müssen an dieser Stelle übergeben werden, dazu gehören die Ladestromstärke und die Steigung der linearen Ladungsphase. Anschließend kann über die Methode `calculateChargeAmount()` die zu ladende Kapazität für einen Zeitraum abgefragt werden. Die Methode `calculateChargeTime()` hingegen errechnet für einen Zielladezustand die benötigte Zeit.

Damit die Multicopter an der Ladestation aufgeladen werden können, müssen diese von den Warteplätzen auf die Ladeplätze verschoben werden. Dafür muss bei jedem Aktualisierungsvorgang die Platzvergabe geprüft und gegebenenfalls geändert werden. Direkt nach der Aufladung werden die bis zu ihrem Zielzustand aufgeladenen Multicopter von den Ladeplätzen entfernt und in die entsprechende Liste `objectsFinished` verschoben. Im Anschluss daran werden die leeren Ladeplätze mit wartenden Multicoptern gefüllt. Die Auswahl des nächsten zu ladenden Multicopters ist in eine andere Methode `getNextWaitingObjectIterator()` ausgelagert. Diese Auslagerung soll bei einem komplexen Auswahlalgorithmus die Gesamtkomplexität der Anwendung reduzieren. Im trivialsten Fall wird hier nur das erste Objekt aus der Warteliste ausgewählt.

Nachdem die Ladeplätze gefüllt wurden, wird die Methode `charge()` aufgerufen. Diese nutzt die `lastUpdate` Eigenschaft aus der `GenericNode`, um die vergangene Zeit seit dem letzten Ladevorgang zu errechnen. Basierend auf der vergangenen Zeit seit dem letzten Ladevorgang wird für jeden Multicopter auf einem Ladeplatz die aufzuladene Kapazität errechnet. Im Anschluss wird die Batterie des Multicopters um den entsprechenden Wert aufgeladen. Abschließend muss der Wert der letzten Aktualisierung auf den momentanen Zeitpunkt gesetzt werden und ein Aufladezyklus ist abgeschlossen.

Abbildung 5.3. – realistisches Ladeverhalten UML-Klassendiagramm



5.7. Reservierungen und Vorhersagen

Für andere Netzwerkteilnehmer ist die Information wie ein potenzieller Ladeprozess verlaufen würde notwendig, um entscheiden zu können, ob dieser wahrgenommen wird. Die Ladestation könnte Vorhersagen über die vermutliche Ladedauer abgeben und dabei alle in den Warte- und Ladecontainern befindlichen Multicopter einbeziehen. Neue Multicopter können allerdings ohne vorherige Information jederzeit eintreffen und der Warteliste hinzugefügt werden. Vorhersagen für Multicopter die noch nicht eingetroffen sind, werden dadurch entwertet und sind nicht verlässlich. Damit die Ladestation ihre Vorhersagen einhalten kann, muss den Multicoptern eine Möglichkeit gegeben werden, einen Platz auf der Warteliste zu reservieren, ohne dass diese bereits an der Ladestation angekommen sind. Reservierungen werden per Nachricht eingeleitet. Die Nachricht vom Typ `reserveSpotMsg` kann im Diagramm 5.4 gesehen werden. Sie enthält Informationen über die voraussichtliche Ankunftszeit, den aktuellen Ladezustand und den voraussichtlichen Verbrauch bis zur Ankunft an der Ladestation. Darüber hinaus ist ein prozentualer Zielzustand enthalten. Dadurch wird den Multicoptern die Möglichkeit geboten nur eine Teilaufladung zu erhalten. Diese Informationen, der Absen-

der und der Zeitpunkt der Reservierung müssen innerhalb der Ladestation gespeichert werden. Dadurch kann bereits zum Zeitpunkt der Reservierung eine Vorhersage über die Ladedauer getroffen werden. Da für jede Vorhersage die Fertigstellungszeitpunkte der vorherigen Multicopter benötigt werden, wird dieser Zeitpunkt mit im Container abgespeichert. Die individuelle Containerklasse `ChargingNodeSpotElement` benötigt eine Eigenschaft für jede zu speichernde Information und je eine Getter- und Setter-Methode für den Zugriff. Die Warte- und Ladeliste wird nicht von den Multicopter Objekten selbst gefüllt, sondern von den `ChargingNodeSpotElement` Objekten, die ein Verweis auf den entsprechenden Multicopter enthalten.

Die Reservierungen ermöglichen der Ladestation die Warteliste basierend auf der Zeit der Reservierung zu organisieren. Dafür muss der Auswahlalgorithmus das nächste Objekt basierend auf der geringsten Reservierungszeit wählen. Aufgrund der Reservierungen kann die Ladestation ihre Vorhersagen garantieren. Falls frühere Reservierungen nicht wahrgenommen werden, kann die Aufladezeit sich verringern, aber eine Verschlechterung ausgehend von der Vorhersage ist nicht möglich.

Die Vorhersagen sind in zwei unterschiedliche Formen vorgesehen. Für die interne Verwendung ist der Fertigstellungszeitpunkt relevant. Dieser wird ebenfalls externen Interessenten zur Verfügung gestellt. Darüber hinaus kann eine Vorhersage über den Ladezustand zu einem spezifizierten Zeitpunkt angefragt werden. Deshalb gibt es zwei verschiedene Nachrichten für eine Vorhersagenanfrage. Die Antwort hingegen hat immer den selben Nachrichtentyp, in dieser ist sowohl der erreichte Ladezustand, als auch der Zeitpunkt der Fertigstellung enthalten. Alle drei Nachrichtentypen können der Abbildung [5.4](#) entnommen werden.

Abbildung 5.4. – Nachrichten für Vorhersagen UML-Klassendiagramm

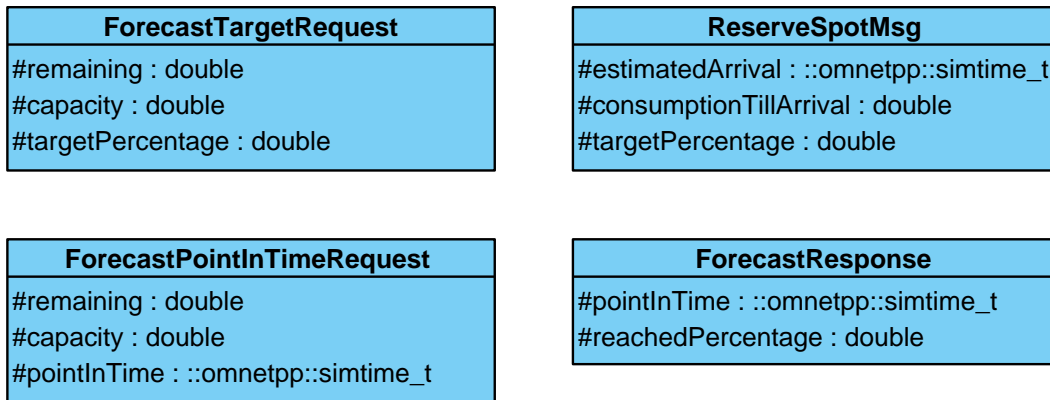
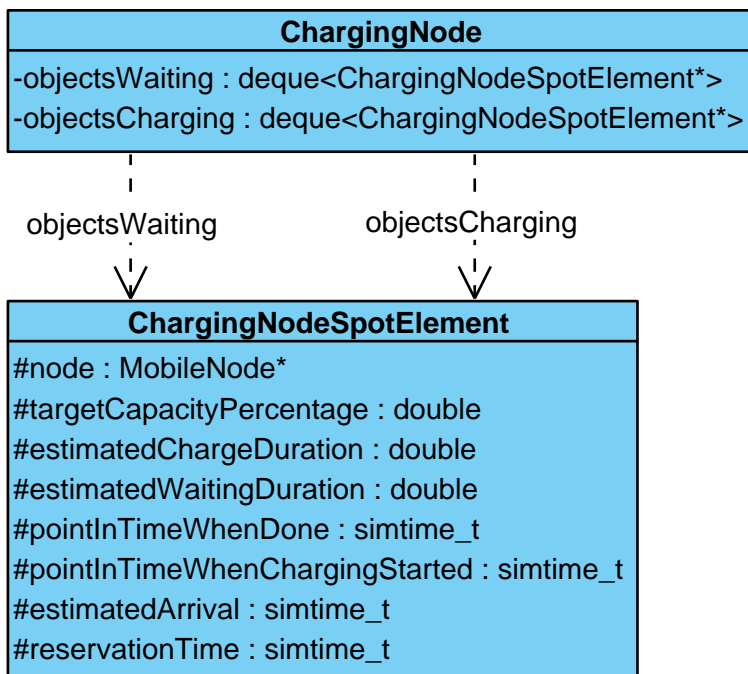


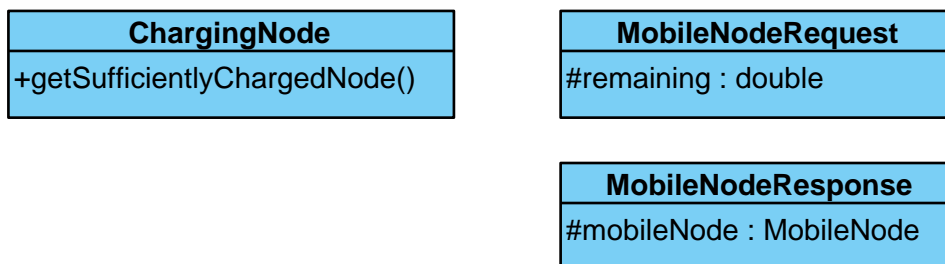
Abbildung 5.5. – Containerklasse UML-Klassendiagramm



5.8. Anfrage für Multicopter

Damit Anfragen für Ersatzmulticopter möglich sind, kann die Ladestation eine Nachricht vom Typ `MobileNodeRequest` empfangen und interpretieren. In dieser Nachricht ist eine Mindestkapazität enthalten. In der Methode `getSufficientlyChargedUAV()` werden die Listen mit wartenden, fertig geladenen und momentan ladenden Multicoptern ausgewertet. Ausgewählt wird der Multicopter mit der geringsten Kapazität, welche dennoch die Mindestanforderung erfüllt. Dadurch wird die kleinst mögliche Menge an Akkumulatorkapazität gebunden. Die Wahrscheinlichkeit den nächsten, mit unter größeren, Auftrag erfüllen zu können ist maximal. Anschließend wird eine Nachricht `MobileNodeResponse` mit der Information über den ausgewählten Multicopter an den Absender zurückgesendet. Der Anfrage- und Antwortnachrichtentyp kann im Diagramm 5.6 betrachtet werden.

Abbildung 5.6. – Anfrage für Multicopter UML-Klassendiagramm



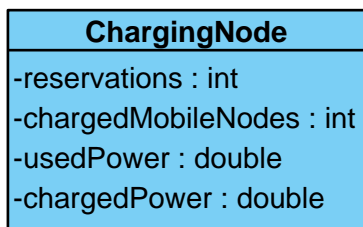
5.9. Nutzungsstatistiken

Das Simulationsframework unterstützt die Aufnahme und Verarbeitung von statistischen Daten. Für die Aufnahme müssen Variablen im Objekt definiert und initialisiert werden. Darüber hinaus müssen die Variablen in der `initialize ()` Methode des jeweiligen Objektes als statistische Daten gekennzeichnet werden. Durch die Nutzung der vom Framework zur Verfügung gestellten Mittel können die Daten zur Laufzeit aus den Objekten ausgelesen werden und stehen für die Nutzung in der graphischen Ober-

fläche zur Verfügung. Zusätzlich können ausführliche Berichte und Diagramme über die Werte im Verlauf der Simulation automatisch erstellt werden.

In der Ladestation werden die Anzahl der Reservierungen und der fertig geladenen Multicopter erfasst. Darüber hinaus wird die von der Ladestation aufgeladene und entladene Energie protokolliert. Die dazugehörigen Eigenschaften können im folgenden Diagramm 5.7 eingesehen werden.

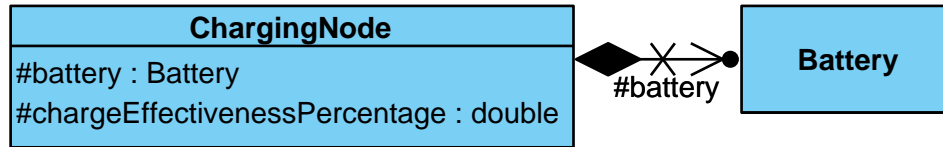
Abbildung 5.7. – Statistikeigenschaften UML-Klassendiagramm



5.10. Ressourcenverwaltung

Für das Hinzufügen eines Energiespeichers zur Ladestation, kann auf die bestehende Projektarchitektur zugegriffen werden. Die Ladestation wird mit einem Objekt der Klasse Battery assoziiert, zu sehen ist dies in der Abbildung 5.8. Während der Initialisierung wird die Maximalkapazität aus der Konfiguration ausgelesen. Darauf basierend wird das Akkumulatorobjekt erstellt. Dieses stellt die Funktionalität eines begrenzten oder unbegrenzten Energiespeichers zur Verfügung, je nach Konfiguration. Innerhalb der charge() Methode wird der Akkumulator des Multicopters geladen und der Akkumulator der Ladestation entladen. Die Entladung wird dabei durch einen zwischen 0 und 1 liegenden Effektivitätsparameter geteilt. Dieser Parameter kann konfiguriert werden. Während des Ladeprozesses wird geprüft, ob die Restladung der Station ausreichend ist und andernfalls eine Fehlermeldung erzeugt.

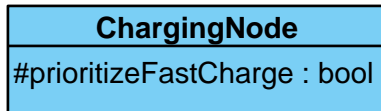
Abbildung 5.8. – Ressourcenverwaltung UML-Klassendiagramm



5.11. Schnelle Teilaufladung priorisieren

Das Interface für den Ladealgorithmus erfordert die Bereitstellung der Information, bis zu welcher relativen Kapazität der Ladevorgang effizient betrieben wird. Für einen durchgehend linearen Algorithmus wäre der Wert 100%. Im genutzten CCCV Verfahren wird der Akkumulator in der ersten Phase bedeutend schneller geladen. Damit insgesamt in der Simulation eine größere Kapazität in kürzerer Zeit aufgeladen werden kann, kann es sinnvoll sein, ausschließlich den linearen Teil aller wartenden Multicopter aufzuladen. Sobald alle Multicopter die lineare Phase abgeschlossen haben, wird die zweite Phase der Aufladung durchgeführt. Diese Funktionalität ist konfigurierbar mit Hilfe der in der Abbildung 5.9 gezeigten booleschen Eigenschaft. Gegebenenfalls werden vollständig geladene Multicopter für eine Mission gebraucht. Falls die Konfiguration dennoch aktiviert ist wird nach der Aufladung geprüft, ob einer der gerade geladenen Akkumulatoren die erste Phase überschritten hat und eine Alternative bereit zur Ladung ist. Sollte dieser Fall eintreten werden die Warte- und Ladeplätze der beiden zugehörigen Multicopter getauscht. Durch den unvorhersehbaren Tausch sind die getroffenen Vorhersagen nicht mehr genau und können unter Umständen nicht eingehalten werden. Wobei weiterhin zum zugesagten Zeitpunkt wenigstens die lineare Phase abgeschlossen ist und dem Multicopter je nach verwendeter Ladestromstärke ein Großteil seiner Gesamtkapazität zur Verfügung steht.

Abbildung 5.9. – Ressourcenverwaltung UML-Klassendiagramm



5.12. Übersicht

In der folgenden Sektion befinden sich die zusammengefassten Klassendiagramme der Konzeptionierung. Die Details werden in den einzelnen Sektionen erläutert. Das erste Diagramm [5.10](#) enthält die konzeptionierten Klassen. Darüber hinaus sind während der Implementierung eine Reihe von Hilfsmethoden innerhalb der jeweiligen Klasse entstanden. Diese sind im Diagramm nicht enthalten. Die zweite Abbildung [5.11](#) zeigt die unterschiedlichen Nachrichtenklassen. Dabei sind nur jene enthalten, welche nach dem Kapitel [5.2](#) eigene Nachrichtentypen darstellen.

Abbildung 5.10. – Überblick UML-Klassendiagramm

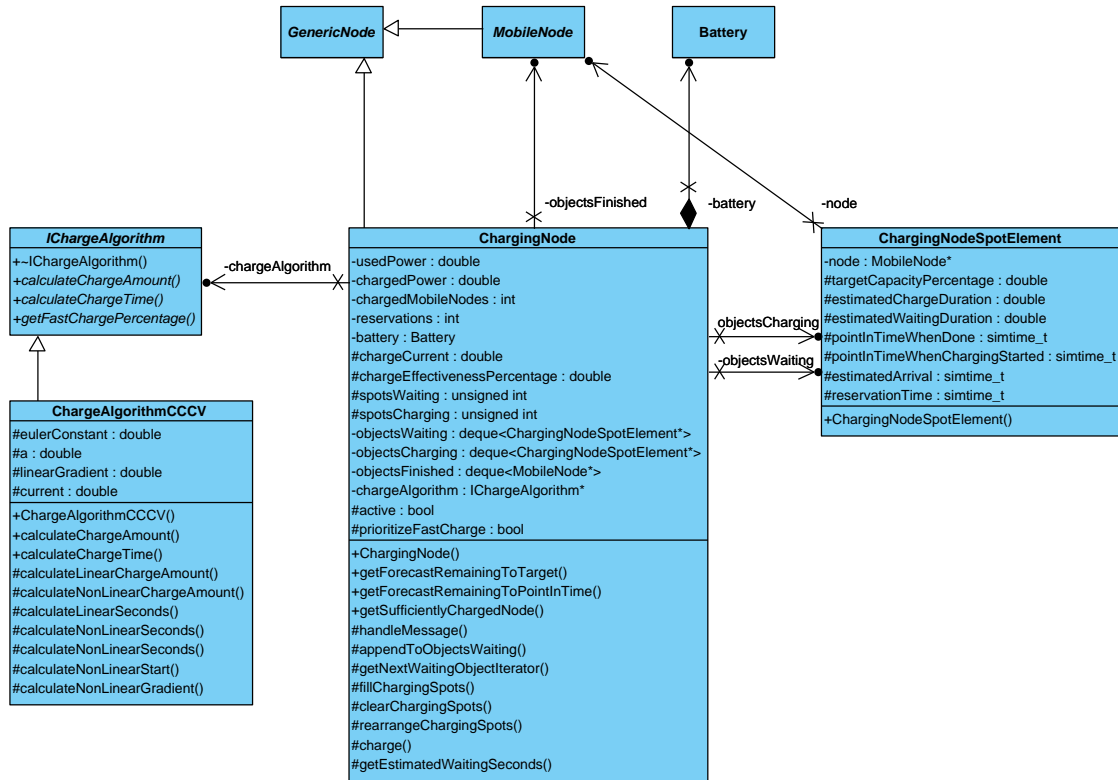
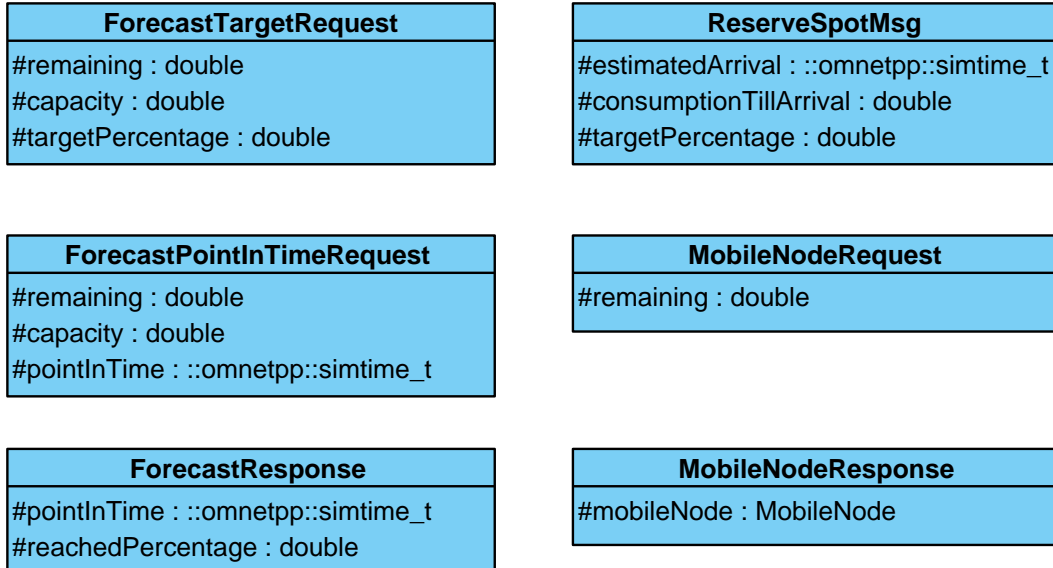


Abbildung 5.11. – Nachrichtentypen UML-Klassendiagramm



6. Implementierung

Im folgenden Kapitel werden einzelne Teile der Implementierung gezeigt und erläutert. Dabei wird ein möglicher Ablauf eines Multicopters mit einer Ladestation aufgezeigt. Eine detaillierte Auseinandersetzung mit den Implementierungen aller Fähigkeiten der Ladestation würde quantitativ den Rahmen sprengen. Die konkreten Implementierungen der anderen Fähigkeiten der Ladestation können dem Quellcode entnommen werden. Es wird mit der Verarbeitung ankommender Multicopter begonnen, dabei wird diesem ein Wartepplatz zugewiesen. In der darauffolgenden regelmäßigen Aktualisierung, werden die Ladeplätze verteilt. Darüber hinaus wird die gesamte Organisation der Ladeplätze vorgestellt. Die initiale Füllung, die Leerung nach erfolgreicher Aufladung und der Austausch insofern bestimmte Kriterien erfüllt werden.

Das Kapitel endet mit einer Übersicht über die erledigten Anforderungen. Dabei werden nicht nur die hier vorgestellten Implementierungen beachtet, sondern die gesamte erledigte Arbeit einbezogen.

6.1. Ankommende Multicopter

Es werden zwei Möglichkeiten angeboten, wie Multicopter einen Ladeplatz erhalten können. Erstens kann eine Nachricht mit dem Namen `startCharge` ohne weitere Parameter versendet werden, sobald die Position der Ladestation erreicht wurde. Das aufzuladende Objekt wird, mit der im zweiten Teil dieses Abschnitts erläuterten Methode `appendToObjectsWaiting()`, an die Warteliste angehängen. Als Objekt wird der Absender der Nachricht angenommen, als erwarteter Zielzustand werden 100% angenommen. Falls die Ladestation bis zu diesem Zeitpunkt nicht aktiv war, wird die regelmäßige Aktualisierungsroutine mit Hilfe einer ersten Nachricht mit dem Namen `update` aktiviert. Diese Nachricht wird zur aktuellen Simulationszeit verschickt und

kommt durch mangelnde Nachrichtenverzögerung zum selben Zeitpunkt an und löst den ersten Ladevorgang und die erste Platzverteilung aus.

Die zweite Möglichkeit um einen Ladeplatz zugewiesen zu bekommen ist eine Reservierung zu tätigen. Dafür muss eine Nachricht mit dem Namen `reserveSpot` an die Ladestation gesendet werden. Diese muss vom Typ `ReserveSpotMsg` sein und Informationen über den gewünschten Zielzustand, die erwartete Ankunft und den erwarteten Energieverbrauch bis zur Ankunft enthalten. Der aktuelle Stand des Akkumulators kann direkt vom Objekt abgefragt werden und muss deshalb nicht verschickt werden. Vorteil einer Reservierung ist, dass die Warteliste nach dem Zeitpunkt der Reservierung sortiert wird. Darüber hinaus wird der Ladestation eine bessere Vorhersage für andere Interessenten ermöglicht. Der Ablauf nach dem Empfang der Nachricht ist analog zur ersten Möglichkeit, mit dem Unterschied das die enthaltenen Informationen an die `appendToObjectsWaiting()` Methode übergeben werden.

```

1 void ChargingNode::handleMessage(cMessage* msg)
2 {
3     if (msg->isName("startCharge")) {
4         EV_INFO << "MobileNode is ready to get charged" << endl;
5         MobileNode *mn = check_and_cast<MobileNode*>(msg->
6             getSenderModule());
7         appendToObjectsWaiting(mn, 100.0);
8
9         if (not active) {
10             msg->setName("update");
11             scheduleAt(simTime(), msg);
12             active = true;
13         }
14     }
15     else if (msg->isName("reserveSpot")) {
16         ReserveSpotMsg *rsmg = check_and_cast<ReserveSpotMsg*>(msg);
17         MobileNode *mn = check_and_cast<MobileNode*>(msg->
18             getSenderModule());
19         appendToObjectsWaiting(mn, rsmg->getTargetPercentage(), simTime
20             (), rsmg->getEstimatedArrival(), rsmg->
21             getConsumptionTillArrival());
22         reservations++;
23         EV_INFO << "Mobile Node is on the way to CS. Spot reserved for:
24             " << rsmg->getEstimatedArrival() << endl;

```

```
20
21     if (not active) {
22         msg->setName("update");
23         scheduleAt(simTime(), msg);
24         active = true;
25     }
26 }
27 // ...
28 }
```

Source Code 6.1 – handleMessage() in ChargingNode.cc

Die zu Beginn des Kapitels mehrfach referenzierte Methode `appendToObjectsWaiting()` hat den Zweck ein Objekt der Warteliste hinzuzufügen. Dabei muss ein Zeiger auf ein Objekt vom Typ `MobileNode` und ein gewünschter Zielladezustand in Prozent übergeben werden. Darüber hinaus werden im Fall einer Reservierung die Zeit der Reservierung, die erwartete Ankunft und der erwartete Verbrauch bis zur Ankunft als Parameter übergeben. Zu Beginn der Methode wird geprüft, ob noch Wartplätze frei sind und ob das Objekt bereits in der Warteschlange ist. Sollte eine der beiden Bedingungen zutreffen wird eine entsprechende Fehlermeldung generiert und die Methode vorzeitig beendet. Anschließend wird ein Containerobjekt der Klasse `ChargingNodeSpotElement` erstellt. Dafür wird ein die Aufladezeit, die zu erwartende Wartezeit, der Zielzustand und Zeiger auf das `MobileNode` Objekt benötigt. Die Aufladezeit wird vom implementierten Ladealgorithmus berechnet, dafür wird der aktuelle Zustand und der zu erwartende Verbrauch berücksichtigt. Die Wartezeit wird in einer Hilfsmethode `getEstimatedWaitingSeconds()` berechnet. Das Ergebnis ist, falls eine schnelle Aufladung priorisiert wird, nicht bindend. Nach der Erstellung werden die Ankunftszeit und der Zeitpunkt der Reservierung mit je einer Setter-Methode gesetzt. Falls es sich um keine Reservierung handelt wird der Standardwert, welcher dem aktuellen Simulationszeitpunkt entspricht verwendet. Abschließend wird das erstellte Containerobjekt der Warteliste `objectsWaiting` hinzugefügt und eine Bestätigungsmitteilung ausgegeben.

```
1 /**
2  * Appends a MobileNode to the waiting queue.
3  * Estimated wait and charge duration get calculated and appended.
4  * Default values take place when there is no reservation and the object
   is already present (object appeared with startCharge message):
```

```

5  */
6  void ChargingNode::appendToObjectsWaiting(MobileNode* mobileNode, double
    targetPercentage, simtime_t reservationTime, simtime_t estimatedArrival
    , double consumption)
7  {
8      // check if the waiting queue size would be exceeded
9      if (objectsWaiting.size() >= spotsWaiting && spotsWaiting != 0) {
10         EV_INFO << "All spots for waiting (" << spotsWaiting << ") are
            already taken." << endl;
11         return;
12     }
13     // check if given object is already in waiting queue
14     if (isInWaitingQueue(mobileNode)) {
15         EV_INFO << "Mobile Node is already in waiting Queue." << endl;
16         return;
17     }
18
19     // generate a new waiting element with estimated charge and waiting
        times
20     // subtract consumption which will occur between reservation and
        the charging process
21     double chargeTime = chargeAlgorithm->calculateChargeTime(mobileNode
        ->getBattery()->getRemaining() - consumption, mobileNode->getBattery
        ()->getCapacity(), targetPercentage);
22     ChargingNodeSpotElement* element = new ChargingNodeSpotElement(
        mobileNode, chargeTime, getEstimatedWaitingSeconds(),
        targetPercentage);
23
24     // set estimatedArrival and reservationTime if not 0, otherwise
        simTime() will be used as default value
25     if (!estimatedArrival.isZero()) {
26         element->setEstimatedArrival(estimatedArrival);
27     }
28     if (!reservationTime.isZero()) {
29         element->setReservationTime(reservationTime);
30     }
31
32     objectsWaiting.push_back(element);
33     EV_INFO << "MobileNode got appended to a waiting spot." << endl;
34 }

```

Source Code 6.2 – appendToObjectsWaiting() in ChargingNode.cc

6.2. Aktualisierung der Ladestation

[caption=updateState() in ChargingNode.cc] Zu Beginn dieser Sektion wird die updateState() Methode vorgestellt, welche den Rahmen des Aktualisierungsvorgangs bereitstellt. Im Anschluss werden die nötigen Schritte und ihr Ablauf vorgestellt. Die Struktur orientiert sich an dem Durchlaufzyklus eines Multicopters und es wird mit dem Hinzufügen zu einem Ladeplatz begonnen.

Die Aktualisierung beginnt nach dem Empfang einer Nachricht mit dem Namen update. Die updateState() Methode, welche jede von der GenericNode erbbende Klasse implementieren muss wird aufgerufen. Zu Beginn wird der Ladezustand der Station überprüft, falls keine weitere Energie zur Verfügung steht wird der restliche Ablauf direkt abgebrochen. Wenn die Ladeplätze besetzt sind werden diese in der charge() Methode aufgeladen. Anschließend werden die bis zum Zielzustand geladenen Multicopter von den Ladeplätzen entfernt. Im Anschluss und unabhängig davon, ob in diesem Aktualisierungsvorgang bereits Objekte geladen wurden, werden die Ladeplätze gefüllt. Nachdem alle Ladeplätze belegt sind oder keine weiteren wartenden Multicopter zur Verfügung stehen, wird die Methode rearrangeChargingSpots() aufgerufen. In dieser Methode werden die Ladeplätze umverteilt. Wenn ein Objekt der Ladeliste später reserviert hat, als ein Objekt der Warteliste, werden diese getauscht. Falls die Priorisierung der schnellen Aufladung aktiviert ist, kann hier eine nachträgliche Änderung der Ladeplätze erfolgen.

```
1 void ChargingNode::updateState()
2 {
3     if (battery.isEmpty()) {
4         EV_WARN << "The battery of the Charging Station is exhausted!";
5         return;
6     }
7     if (not objectsCharging.empty()) {
8         charge();
9         clearChargingSpots();
```

```

10     }
11     fillChargingSpots();
12     rearrangeChargingSpots();
13 }

```

6.2.1. Ladeplätze besetzen

Damit die Multicopter aufgeladen werden können, müssen sie aus der Warteliste in die Ladeliste verschoben werden. Dies geschieht in der Methode `fillChargingSpots()`, welche während eines Aktualisierungsvorgangs nach dem Aufladen ausgelöst wird. Die Änderung der Plätze erfolgt erst nach der Aufladung, weil nur die Multicopter aufgeladen werden sollen, welche bereits bei der vorherigen Aktualisierung auf den Ladeplätzen stationiert waren. Als erstes wird geprüft, ob sich Multicopter in der Warteliste befinden, die bereits an der Ladestation angekommen sind. Reservierungen erscheinen auch in der Warteliste, bevor der entsprechende Multicopter an der Ladestation eingetroffen ist. Objekte können jedoch nur geladen werden wenn die Koordinaten des Objektes und der Ladestation übereinstimmen. Danach wird das nächste wartende Objekt ausgewählt. Die Priorisierung erfolgt in der Methode `getNextWaitingObjectIterator()`, welche im zweiten Teil dieser Sektion erläutert wird. Diese Methode gibt einen Iterator zurück, welcher auf das entsprechende Containerobjekt zeigt. Anschließend wird eine Schleife durchlaufen, solange es freie Ladeplätze gibt und für die Aufladung bereit Objekte zur Verfügung stehen. Innerhalb der Schleife wird der Startzeitpunkt des Ladevorgangs mit Hilfe einer Setter-Methode zu dem Containerobjekt hinzugefügt. Anschließend wird der Multicopter an die Ladeliste `objectsCharging` angehängen und von der Warteliste entfernt. Am Ende der Schleife wird die Anzahl der zur Verfügung stehenden wartenden Objekte und der Zeiger auf den nächsten Container aktualisiert.

```

1  /**
2   * Populates the charging nodes.
3   */
4  void ChargingNode::fillChargingSpots()
5  {
6      // when there are no waiting objects, the method does nothing
7      int availableNodes = numberWaitingAndPhysicallyPresent();

```

```
8   if (availableNodes == 0) {
9       return;
10  }
11
12  // get the next waiting object
13  std::deque<ChargingNodeSpotElement*>::iterator nextWaitingObject =
14      getNextWaitingObjectIterator(prioritizeFastCharge);
15
16  // loop through empty charging spots and fill them with waiting
17  // objects
18  while (spotsCharging > objectsCharging.size() && availableNodes > 0)
19  {
20      (*nextWaitingObject)->setPointInTimeWhenChargingStarted(simTime
21          ());
22      objectsCharging.push_back(*nextWaitingObject);
23      objectsWaiting.erase(nextWaitingObject);
24      nextWaitingObject = getNextWaitingObjectIterator(
25          prioritizeFastCharge);
26      availableNodes = numberWaitingAndPhysicallyPresent();
27  }
```

Source Code 6.3 – fillChargingSpots() in ChargingNode.cc

Die Auswahl des nächsten wartenden Objekts erfolgt innerhalb der Methode getNextWaitingObjectIterator(). Diese erwartet als Parameter einen booleschen Wert, der angibt, ob eine schnelle Teilaufladung priorisiert werden soll. Der Rückgabewert ist ein Iterator, welcher ein Containerelement aus der Warteliste referenziert. Der Iterator kann in Schleifen Verwendung finden, ebenso kann durch Dereferenzierung das entsprechende Containerobjekt angesprochen werden. Zu Beginn wird der Ergebnisiterator für das nächste wartende Objekt mit einem leeren Wert gefüllt. Da Iteratoren keinen NULL Wert kennen wird dafür objectsWaiting.end() verwendet. Dieser Ausdruck kann verstanden werden, als das Element, welches auf das letzte Element in der Warteliste folgt. Anschließend wird der Laufiterator mit dem ersten Wert der Warteliste initialisiert. Die darauffolgende Schleife durchläuft die gesamte Warteliste. Falls das Objekt sich nicht an der Ladestation befindet wird der Laufiterator inkrementiert und der Schleifendurchlauf abgebrochen. Das gleiche Ereignis tritt ein, falls eine Schnelle Aufladung priorisiert wird und das vom Laufiterator referenzierte Objekt über eine höhere Ladung verfügt,

als vom Ladealgorithmus vorgeschrieben wird. Falls in einem Schleifendurchlauf der nächste Schritt erreicht wird und der Ergebnisiterator noch mit seinem Initialwert belegt ist, wird dieser mit dem aktuellen Laufiterator überschrieben. Falls der Ergebnisiterator bereits belegt wurde, aber das referenzierte Objekt eine spätere Zeit der Reservierung aufweist, als das vom Laufiterator referenzierte Objekt, wird er ebenfalls überschrieben. Abschließend muss innerhalb der Schleife der Laufiterator inkrementiert werden, damit keine Endlosschleife entsteht.

Wenn die Schleife komplett durchlaufen wurde, eine schnelle Aufladung priorisiert wird und der Ergebnisiterator immernoch mit dem Initialwert belegt ist, erfolgt ein rekursiver Aufruf, in welchem der boolesche Parameter mit `false` übergeben wird. Dadurch werden Multicopter ausgewählt, welche die Grenzkapazität für eine schnelle Aufladung überschreiten. Dies tritt allerdings nur ein, wenn unter Berücksichtigung der Grenzkapazität kein wartendes Objekt gefunden wurde. Abschließend wird der Ergebnisiterator zurückgegeben. Solange sich ein Objekt in der Warteliste befindet, hat der Ergebnisiterator einen Inhalt, ansonsten wird der initiale NULL Wert zurückgegeben.

```

1  /**
2  * Elements in the waiting queue get prioritized by their reservationTime
3  *
4  * When fastCharge is enabled the top priority is that the object has
5  * less energy then the chargeAlgorithm is advertising as fastCharge.
6  * Furthermore they need to be physically at the ChargingNode.
7  * @return std::deque<ChargingNodeSpotElement*>::iterator to the next
8  * element in waiting queue which is physically present
9  */
10 std::deque<ChargingNodeSpotElement*>::iterator ChargingNode::
11     getNextWaitingObjectIterator(bool fastCharge)
12 {
13     std::deque<ChargingNodeSpotElement*>::iterator next = objectsWaiting
14         .end();
15     std::deque<ChargingNodeSpotElement*>::iterator objectWaitingIt =
16         objectsWaiting.begin();
17     while (objectWaitingIt != objectsWaiting.end()) {
18         if (not isPhysicallyPresent((*objectWaitingIt)->getNode())) {
19             objectWaitingIt++;
20             break;
21         }
22     }
23     if (fastCharge

```



```
17      && static_cast<double>((*objectWaitingIt)->getNode()->getBattery
18      ()->getRemainingPercentage()) > chargeAlgorithm->
19      getFastChargePercentage()) {
20          objectWaitingIt++;
21          break;
22      }
23      if (next == objectsWaiting.end()
24      || ((*objectWaitingIt)->getReservationTime() < (*next)->
25      getReservationTime() && (*objectWaitingIt)->getEstimatedArrival
26      () <= simTime())) {
27          next = objectWaitingIt;
28      }
29      objectWaitingIt++;
30  }
31  if (fastCharge && next == objectsWaiting.end()) {
32      return getNextWaitingObjectIterator(false);
33  }
34  return next;
35  }
```

Source Code 6.4 – getNextWaitingObjectIterator() in ChargingNode.cc

6.2.2. Ladeplätze aufladen

Nachdem die Multicopter einen Ladeplatz zugewiesen bekommen haben, kann die Aufladung starten. Zu Beginn der `charge()` Methode wird die aktuelle Zeit für den späteren Zugriff abgespeichert. Anschließend werden mit Hilfe einer `for` Schleife alle Ladeplätze durchlaufen. In jedem Schleifendurchlauf wird die aufzuladende Kapazität berechnet. Dafür wird die Methode `calculateChargeAmount()`, welche jeder Ladealgorithmus durch das Interface implementieren muss, aufgerufen. Der Funktion werden der aktuelle Ladezustand, die Maximalkapazität und die Anzahl der vergangenen Sekunden seit der letzten Aktualisierung übergeben. Für die Berechnung der vergangenen Zeit wird das Maximum aus der Eigenschaft `lastUpdate` und der im Containerobjekt gespeicherten Startzeit gebildet. Anschließend wird der entsprechende Wert von der aktuellen Zeit abgezogen. Das Ergebnis ist die Zeit in Sekunden, welche der Multicopter seit seiner

letzten Aufladung auf dem Ladeplatz verbracht hat. Die Auswahl aus der letzten Aktualisierungszeit und der Ladestartzeit des jeweiligen Multicopters stellt sicher, dass ein Multicopter nicht versehentlich für einen Zeitraum aufgeladen wird, in dem dieser nicht auf dem Ladeplatz war. Nachdem die Ladekapazität berechnet wurde, wird eine Statusmitteilung auf der Konsole ausgegeben und anschließend der Akkumulator des Multicopters um den Wert aufgeladen. Da die Ladestation gegebenenfalls ebenfalls begrenzte Ressourcen zur Verfügung hat, wird im Anschluss der Akkumulator um die Ladekapazität geteilt durch die Ladeeffektivität entladen. Zum Abschluss werden die beiden statistischen Werte `usedPower` und `chargedPower` aktualisiert.

```

1  /**
2  * Charges the nodes placed on the charging spots depending on the last
   update.
3  */
4  void ChargingNode::charge()
5  {
6      simtime_t currentTime = simTime();
7      for (unsigned int i = 0; i < objectsCharging.size(); i++) {
8          double chargeAmount = chargeAlgorithm->calculateChargeAmount(
9              objectsCharging[i]->getNode()->getBattery()->getRemaining(),
10             objectsCharging[i]->getNode()->getBattery()->getCapacity(),
11             (currentTime - std::max(lastUpdate, objectsCharging[i]->
12                 getPointInTimeWhenChargingStarted())) .dbl());
13             EV_INFO << "UAV in slot " << i << " is currently getting charged
14                 . Currently Remaining: " << objectsCharging[i]->getNode()->
15                 getBattery()->getRemaining() << " mAh. Amount: " << chargeAmount
16                 << " mAh" << endl;
17             objectsCharging[i]->getNode()->getBattery()->charge(chargeAmount
18                 );
19             battery.discharge(chargeAmount / this->
20                 chargeEffectivenessPercentage);
21             usedPower += chargeAmount / this->chargeEffectivenessPercentage;
22             chargedPower += chargeAmount;
23         }
24     }

```

Source Code 6.5 – `charge()` in `ChargingNode.cc`

6.2.3. Ladeplätze leeren

Die Ladestation hat nur eine begrenzte Anzahl Ladeplätze. Nach einem abgeschlossenen Ladevorgang wird der Ladeplatz wieder freigeräumt, damit weitere Multicopter aufgeladen werden können. In einer Iterator basierten Schleife werden alle Ladeplätze nacheinander abgearbeitet. In jedem Durchlauf wird geprüft, ob das Element auf dem aktuellen Ladeplatz seinen Zielladezustand erreicht hat oder vollständig aufgeladen ist. Falls diese Bedingung zutrifft wird eine Mitteilung auf der Konsole ausgegeben. Anschließend wird eine wait Nachricht an die entsprechende MobileNode versendet, gefolgt von einer nextCommand Nachricht. Diese Nachrichten bewirken, dass ein Wartekommando an die Kommandoliste der MobileNode angehängen wird und anschließend das aktuelle Ladekommando beendet wird. Im Wartemodus verbleibt eine MobileNode untätig, bis ihr eine neue Aufgabe zugewiesen wird. Nachdem der Multicopter informiert wurde, wird dieser der Liste mit den fertig geladenen Multicoptern hinzugefügt und von seinem aktuellen Ladeplatz entfernt. Im Anschluss wird der statistische Wert chargedUAVs, der die geladenen Objekte mitzählt, inkrementiert. Abschließend wird der Laufiterator inkrementiert, um den nächsten Schleifendurchlauf einzuleiten und damit den nächsten Ladeplatz zu überprüfen.

```
1  /*
2  * Remove nodes from charging spot when done
3  */
4  void ChargingNode::clearChargingSpots()
5  {
6      std::deque<ChargingNodeSpotElement*>::iterator objectChargingIt =
          objectsCharging.begin();
7
8      while (objectChargingIt != objectsCharging.end()) {
9          if ((*objectChargingIt)->getNode()->getBattery()->
              getRemainingPercentage() > (*objectChargingIt)->
              getTargetCapacityPercentage()
10             || (*objectChargingIt)->getNode()->getBattery()->isFull()) {
11              EV_INFO << "MobileNode (Id: " << (*objectChargingIt)->
                  getNode()->getId() << ") is charged to target: "
12              << (*objectChargingIt)->getNode()->getBattery()->
                  getRemainingPercentage() << "/" << (*objectChargingIt)->
                  getTargetCapacityPercentage()
13              << "%" << endl;
```

```

14      // Send wait message to node
15      MobileNode* mobileNode = (*objectChargingIt)->getNode();
16      send(new cMessage("wait"), getOutputGateTo(mobileNode));
17      // Send a message to the node which signalizes that the
18      // charge process is finished
19      send(new cMessage("nextCommand"), getOutputGateTo(mobileNode));
20      // Push fully charged nodes to the corresponding list
21      objectsFinished.push_back((*objectChargingIt)->getNode());
22      objectsCharging.erase(objectChargingIt);
23      // increment the statistics value
24      chargedMobileNodes++;
25  }
26      objectChargingIt++;
27  }

```

Source Code 6.6 – clearChargingSpots() in ChargingNode.cc

6.2.4. Ladeplätze neu verteilen

Für ein reibungslosen Ladeprozess, unter Berücksichtigung einer Priorisierung, müssen die Ladeplätze neu verteilt werden können, bevor der entsprechende Multicopter vollständig geladen ist. Zu Beginn der Methode rearrangeChargingSpots() wird geprüft ob die Ladeplätze vollständig belegt sind und wenigstens ein zur Aufladung bereiter Multicopter in der Warteliste ist. Ein Austausch ist in beiden Fällen nicht nötig beziehungsweise unmöglich. Demzufolge wird die Funktion abgebrochen. Sollten beide Bedingungen nicht zutreffen, folgt der weitere Methodenablauf. Ein Iterator der auf das nächste wartende Objekt zeigt wird gemäß der in 6.2.1 vorgestellten Methode getNextWaitingObjectIterator() ausgewählt. Anschließend werden die Ladeplätze analog zu 6.2.3 nacheinander bearbeitet. Im Schleifenrumpf folgen die mit einer Adjunktion verknüpften Austauschbedingungen. Wenn die schnelle Aufladung deaktiviert ist oder der nächste wartende Multicopter die effiziente Grenzkapazität nicht überschreitet und zusätzlich die Reservierungszeit des aktuell geladenen Multicopters unterbietet, wird der Ladeplatz neu verteilt. Die zweite Möglichkeit um eine Neuverteilung auszulösen ist, dass die schnelle Aufladung priorisiert wird und der aktuell geladene Multicopter

die effizienten Grenzkapazität überschreitet, während der nächste wartende Multicopter diese unterschreitet. Für den Austausch wird das wartende Objekt zwischengespeichert. Anschließend wird das Zeigerziel des wartenden Objekts mit dem des aktuell geladenen Objekts überschrieben. Danach wird der Zeiger, der auf den Ladeplatz zeigt mit dem temporär gespeicherten nächsten wartenden Objekt überschrieben. Anschließend wird für den neuen geladenen Multicopter die Ladestartzeit gesetzt. Nachdem der Austausch abgeschlossen ist, wird eine Mitteilung in der Konsole ausgegeben und der Zeiger auf das nächste wartende Objekt neu generiert. Zum Abschluss der Schleife wird der Laufiterator inkrementiert, damit der nächste Ladeplatz abgearbeitet werden kann.

```
1  /*
2  * Exchange waiting spots when needed due to earlier reservation or the
   fast charge mechanism
3  */
4  void ChargingNode::rearrangeChargingSpots()
5  {
6      // this method does nothing when either there is no object charged
       currently or there is no available waiting object
7      if (objectsCharging.size() < spotsCharging ||
       numberWaitingAndPhysicallyPresent() == 0) {
8          return;
9      }
10
11     // get the next waiting object
12     std::deque<ChargingNodeSpotElement*>::iterator nextWaitingObject =
       getNextWaitingObjectIterator(prioritizeFastCharge);
13
14     // loop through currently used spots and check for earlier
       reservations
15     // when an earlier reservation time occurs, throw out the currently
       charged node and push it back to the waiting objects
16     std::deque<ChargingNodeSpotElement*>::iterator objectChargingIt =
       objectsCharging.begin();
17     while (objectChargingIt != objectsCharging.end()) {
18         if (((*objectChargingIt)->getReservationTime() > (*
       nextWaitingObject)->getReservationTime()
19         && (not prioritizeFastCharge
```

```

20     || static_cast<double> ((*nextWaitingObject)->getNode()->
    getBattery()->getRemainingPercentage())
21 < getChargeAlgorithm()->getFastChargePercentage ((*
    nextWaitingObject)->getNode()->getBattery()->getCapacity()))
22 || (prioritizeFastCharge
23 && static_cast<double> ((*nextWaitingObject)->getNode()->
    getBattery()->getRemainingPercentage())
24 < getChargeAlgorithm()->getFastChargePercentage ((*
    nextWaitingObject)->getNode()->getBattery()->getCapacity())
25 && (static_cast<double> ((*objectChargingIt)->getNode()->
    getBattery()->getRemainingPercentage())
26 >= getChargeAlgorithm()->getFastChargePercentage ((*
    objectChargingIt)->getNode()->getBattery()->getCapacity())))) {
27     ChargingNodeSpotElement* temp = *nextWaitingObject;
28     *nextWaitingObject = *objectChargingIt;
29     *objectChargingIt = temp;
30     (*objectChargingIt)->setPointInTimeWhenChargingStarted(
        simTime());
31
32     EV_INFO << "MobileNode ID(" << (*nextWaitingObject)->getNode
        (->getId() << ") charge spot exchanged with ID("
33     << (*objectChargingIt)->getNode()->getId() << ") waiting
        spot." << endl;
34
35     nextWaitingObject = getNextWaitingObjectIterator(
        prioritizeFastCharge);
36 }
37 objectChargingIt++;
38 }
39 }

```

Source Code 6.7 – rearrangeChargingSpots() in ChargingNode.cc

6.3. Auswertung Anforderungen

Im folgenden Kapitel wird die Implementierung im Hinblick auf die in 4 festgestellten Anforderungen ausgewertet. In der Tabelle 6.1 ist zu sehen, dass alle nicht funktionalen Anforderungen erledigt wurden. Darüber hinaus sind im Bereich der funktionalen

Anforderungen die beiden wichtigsten Kategorien MUST und SHOULD vollständig abgearbeitet und erfüllt. Lediglich die zwei Anforderungen aus der Kategorie COULD und die eine Anforderung aus der Kategorie WON'T bleiben unerledigt. Gemäß der MoSCoW-Methode war die Implementierung erfolgreich. Die noch offen gebliebenen Anforderungen können eine Anregung für weiterführende Arbeiten darstellen.

Tabelle 6.1. – Auswertung Anforderungen

Bezeichnung	Funktional	Priorität	Implementiert
Ladeplätze	Ja	MUST	Ja
Warteplätze	Ja	MUST	Ja
Verbindung mit Multicopter	Ja	MUST	Ja
Realistisches Ladeverhalten	Ja	MUST	Ja
Reservierungen	Ja	MUST	Ja
Vorhersagen	Ja	MUST	Ja
Anfrage für Multicopter	Ja	MUST	Ja
Nutzungsstatistiken	Ja	MUST	Ja
Ressourcenverwaltung	Ja	SHOULD	Ja
Schnelle Teilaufladung Priorisieren	Ja	SHOULD	Ja
Umverteilung von Multicoptern	Ja	COULD	Nein
Kooperation zwischen Ladestationen	Ja	COULD	Nein
Mobile Ladestation	Ja	WON'T	Nein
Codesprache	Nein	MUST	Ja
Coderichtlinien	Nein	MUST	Ja
Ladeplatzorganisation Zeitverzögerung	Nein	MUST	Ja
Graphische Darstellung	Nein	SHOULD	Ja

7. Fazit und Ausblick

Die Bachelorarbeit hatte das Ziel eine intelligente Ladestation in eine Multicopter Simulation zu integrieren. Dafür wurden in Kapitel 4 Anforderungen aufgenommen und kategorisiert. Wie in der Auswertung der Anforderungen 6.3 festgestellt wurde, konnten die wichtigsten Anforderungen erfüllt werden. Die Ladestation verfügt über eine definierbare, variable Anzahl an Warte- und Ladeplätzen. Sie kann mit den anderen Netzwerkteilnehmern, in erster Linie Multicopter, kommunizieren und Informationen über den Ladeverlauf vorhersagen. Darüber hinaus können Reservierungen der Ladeplätze vorgenommen werden, die einen vorhersehbaren Simulationsverlauf ermöglichen. Der Ladeverlauf wird realitätsnah abgebildet, dennoch sind wie in Kapitel 3.5 festgestellt die Beispielmessungen nicht repräsentativ. Durch die Ähnlichkeit mit den in der Literatur festgestellten Angaben, können die Werte ungeachtet der Mängel verwendet werden. Die Ladestation ist dazu fähig den nicht durchgehend linearen Ladeverlauf zu berücksichtigen und kann, wenn gewünscht, den effizienteren Ladeanteil priorisieren. Alle mit der Ladestation in Kontakt stehenden Multicopter werden von dieser verwaltet und können auf Anfrage für neue Aufgaben zur Verfügung gestellt werden. Dabei wird ein möglichst wenig geladener und dennoch ausreichender Multicopter von der Ladestation ausgewählt, um möglichst wenig Ressourcen zu binden.

Für zukünftige Arbeiten bieten sich vor allem zwei sehr verschiedene Gebiete an. Erstens könnten die Testmessungen stark erweitert werden. Dafür wäre eine breitere Auswahl der Ladegeräte und Akkumulatoren mit unterschiedlichen Maximalkapazitäten sinnvoll. Gestützt durch eine ausführlichere Literaturrecherche spezialisiert auf den Themenbereich Ladeprozess für Lithium-Ionen-Akkumulatoren könnte dadurch der Ladealgorithmus verbessert werden und realitätsnähere Ergebnisse liefern. Zweitens beschränkt sich die Kommunikation der Ladestation nahezu ausschließlich auf Multicopter. Damit der globale Ladeprozess aller in der Simulation befindlicher Multicopter verbessert werden könnte, könnten die Ladestationen untereinander kommunizieren.

Ansätze dafür bieten die nicht erfüllten funktionalen Anforderungen aus 4. Durch den Austausch von Multicoptern während und nach dem Ladeprozess könnte dieser beschleunigt werden und außerdem der Bedarf von Ersatzmulticoptern besser bedient werden.

A. Anhang

Abbildung A.1. – Lademessung 1A Versuch 1

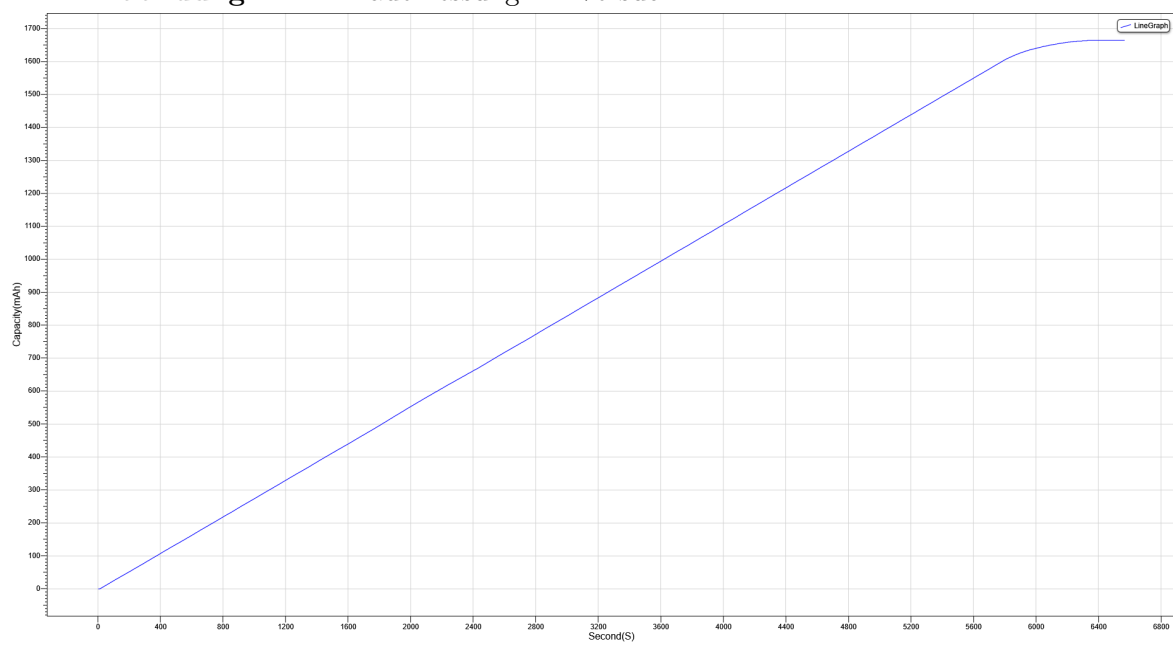


Abbildung A.2. – Lademessung 1A Versuch 2

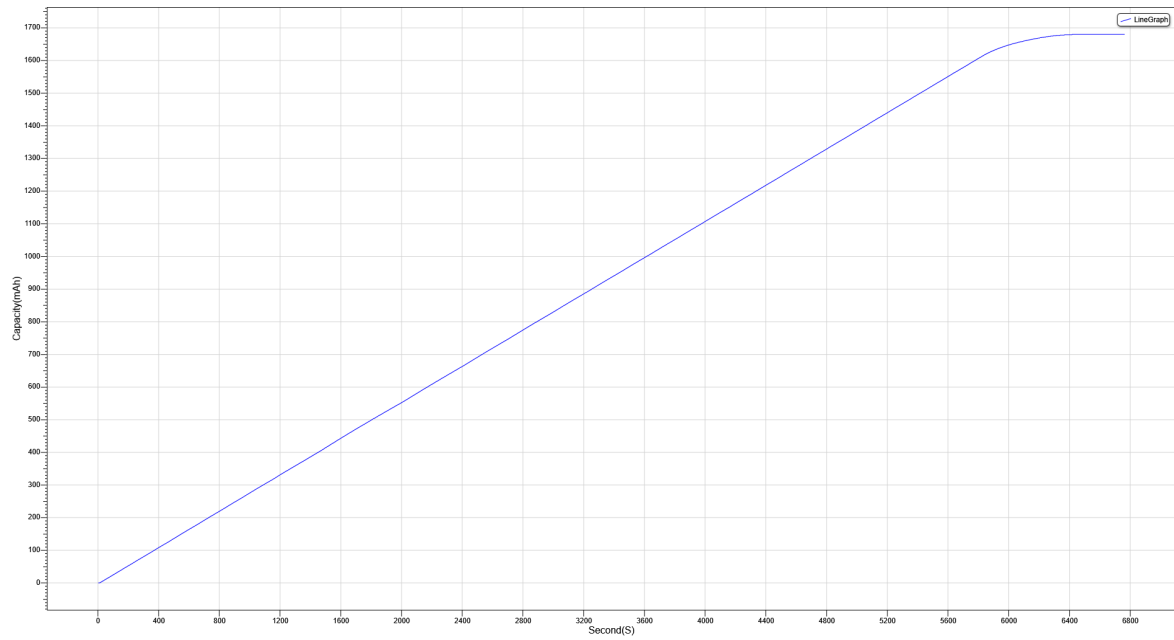


Abbildung A.3. – Lademessung 1A Versuch 3

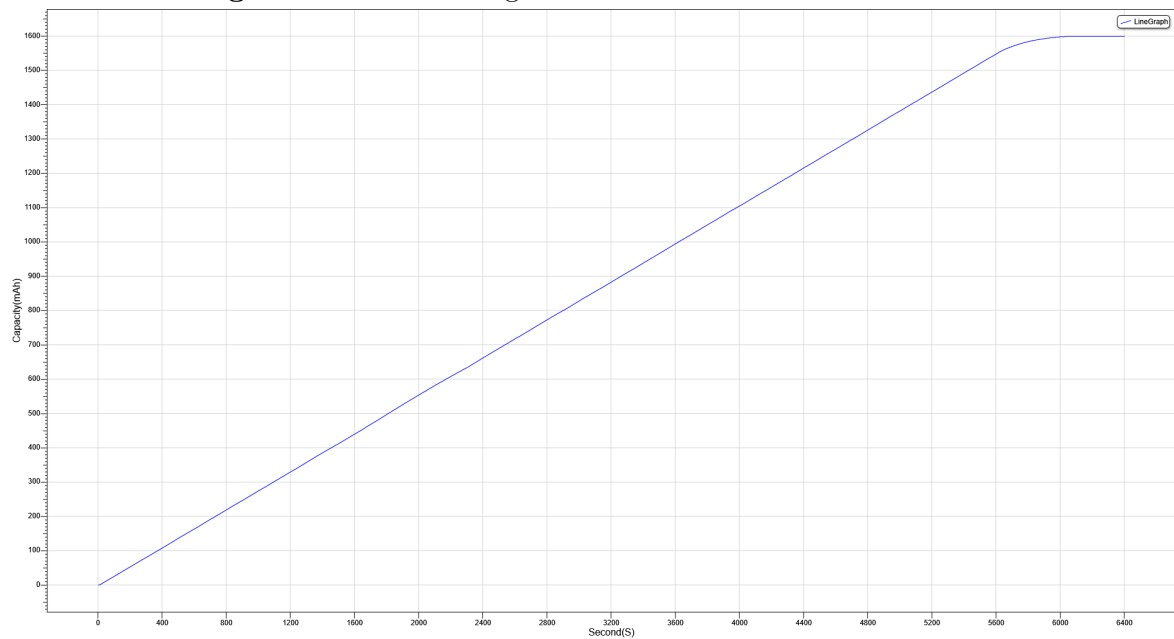


Abbildung A.4. – Lademessung 2A Versuch 1

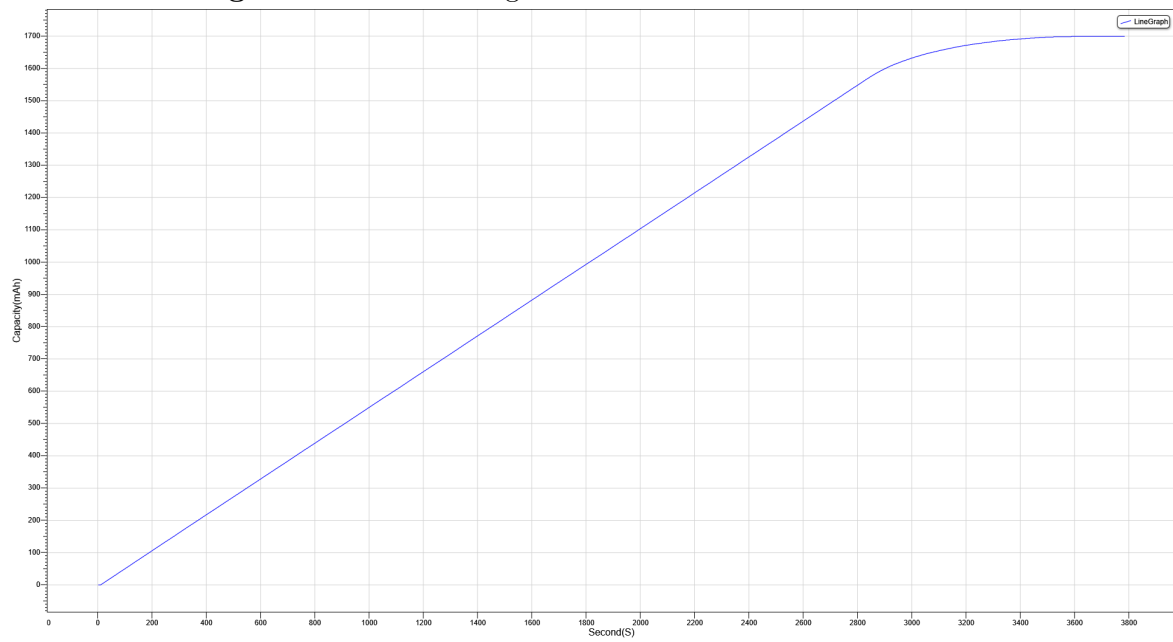


Abbildung A.5. – Lademessung 2A Versuch 2

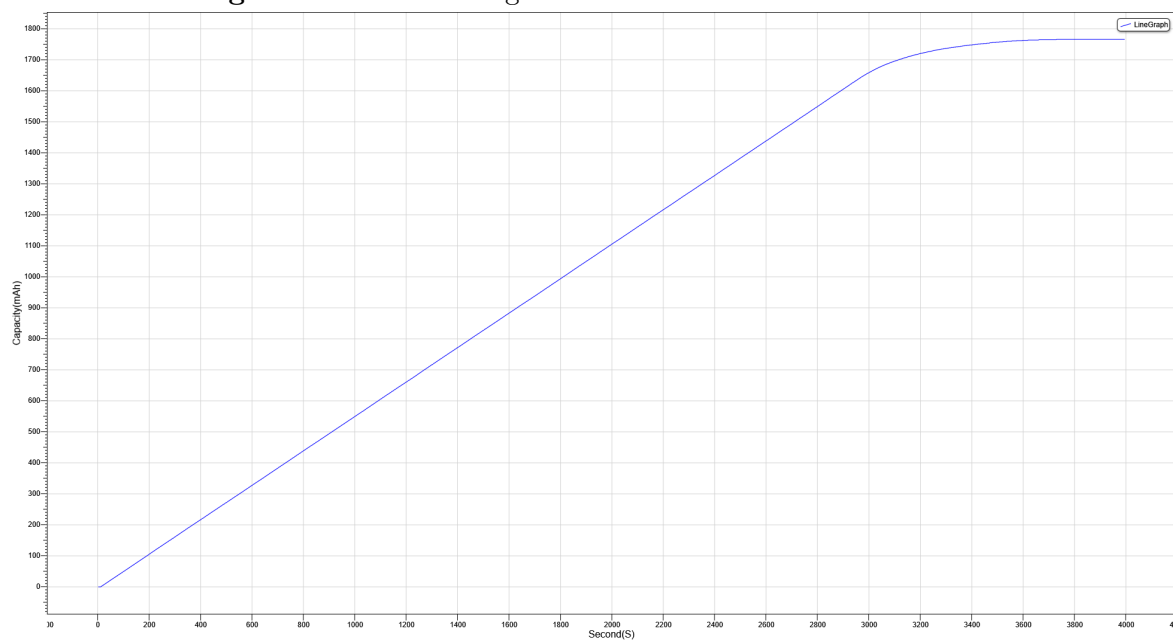


Abbildung A.6. – Lademessung 2A Versuch 3

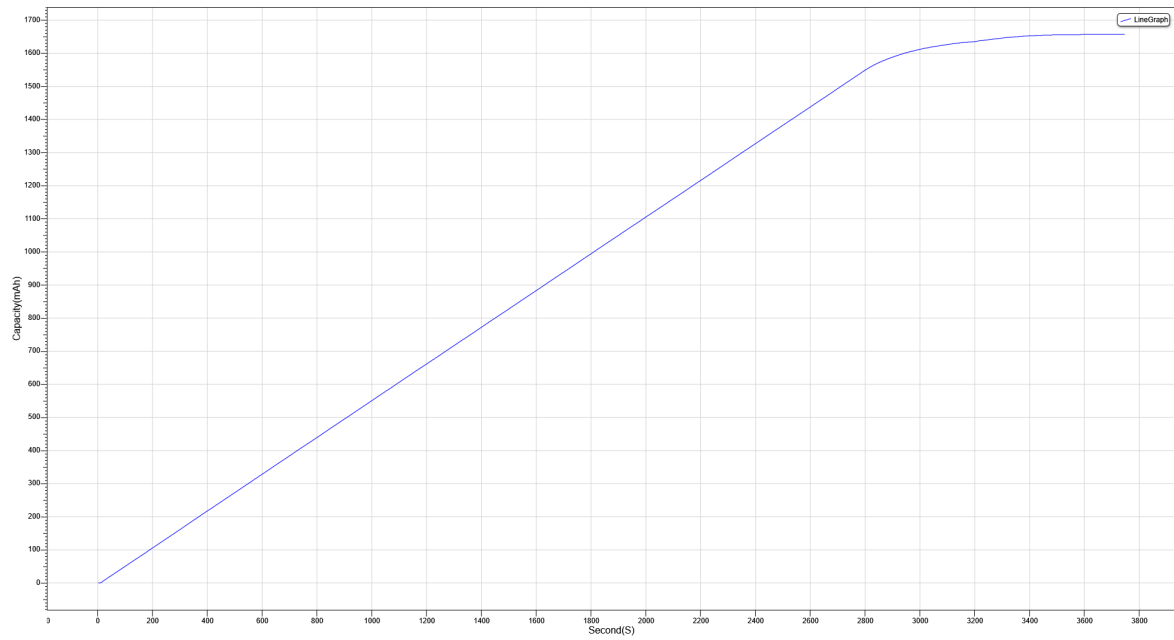


Abbildung A.7. – Lademessung 4A Versuch 1

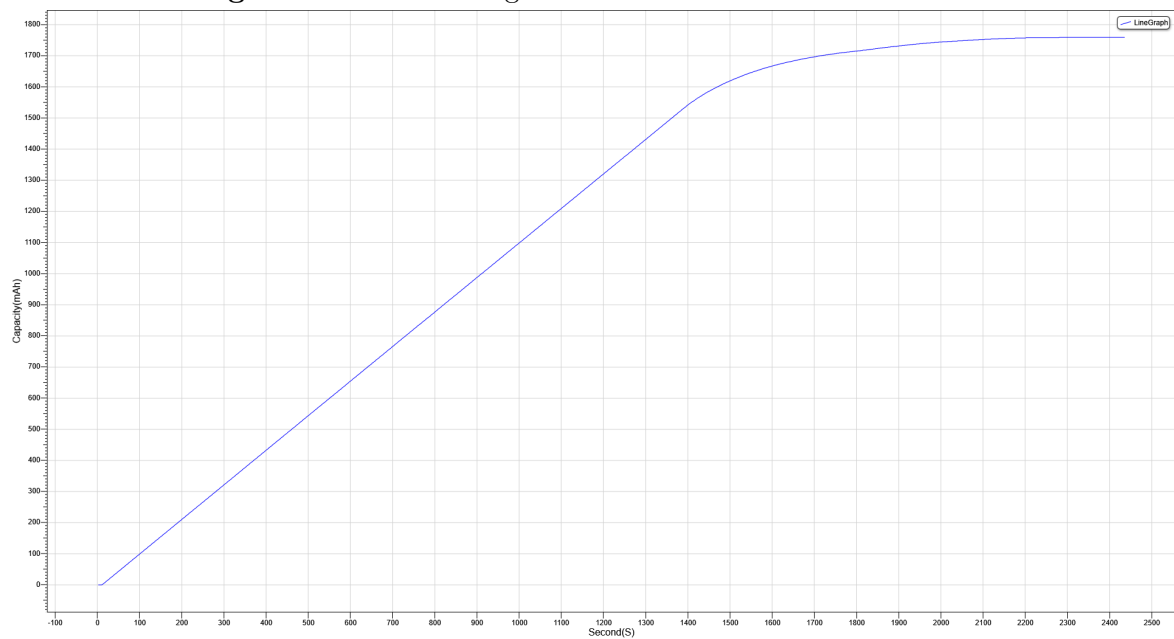


Abbildung A.8. – Lademessung 4A Versuch 2

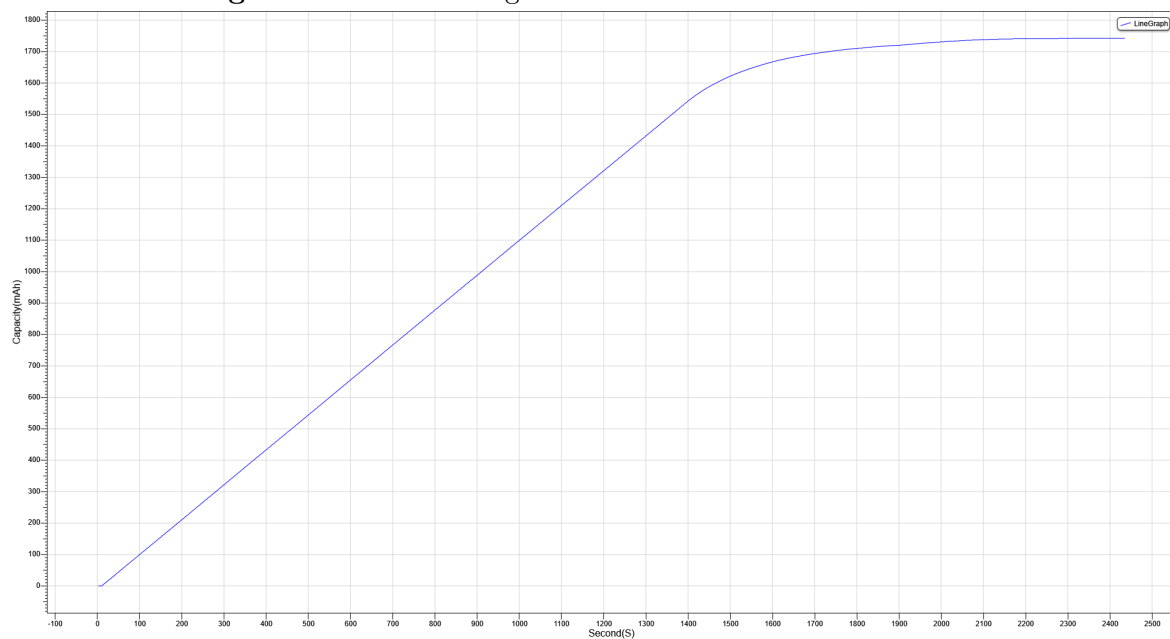
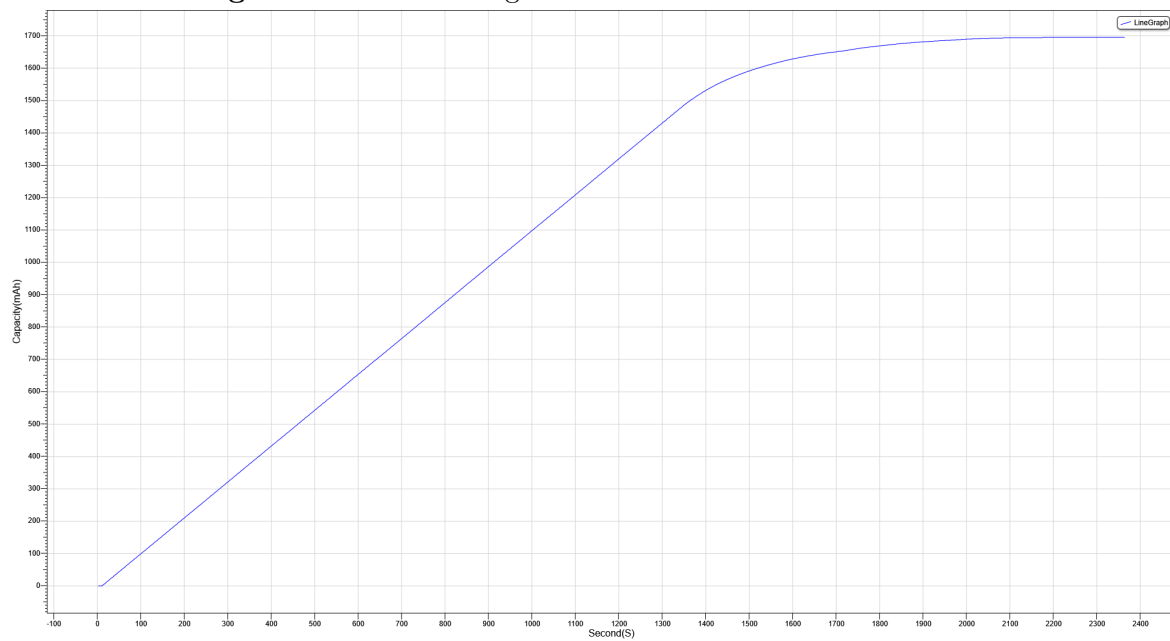
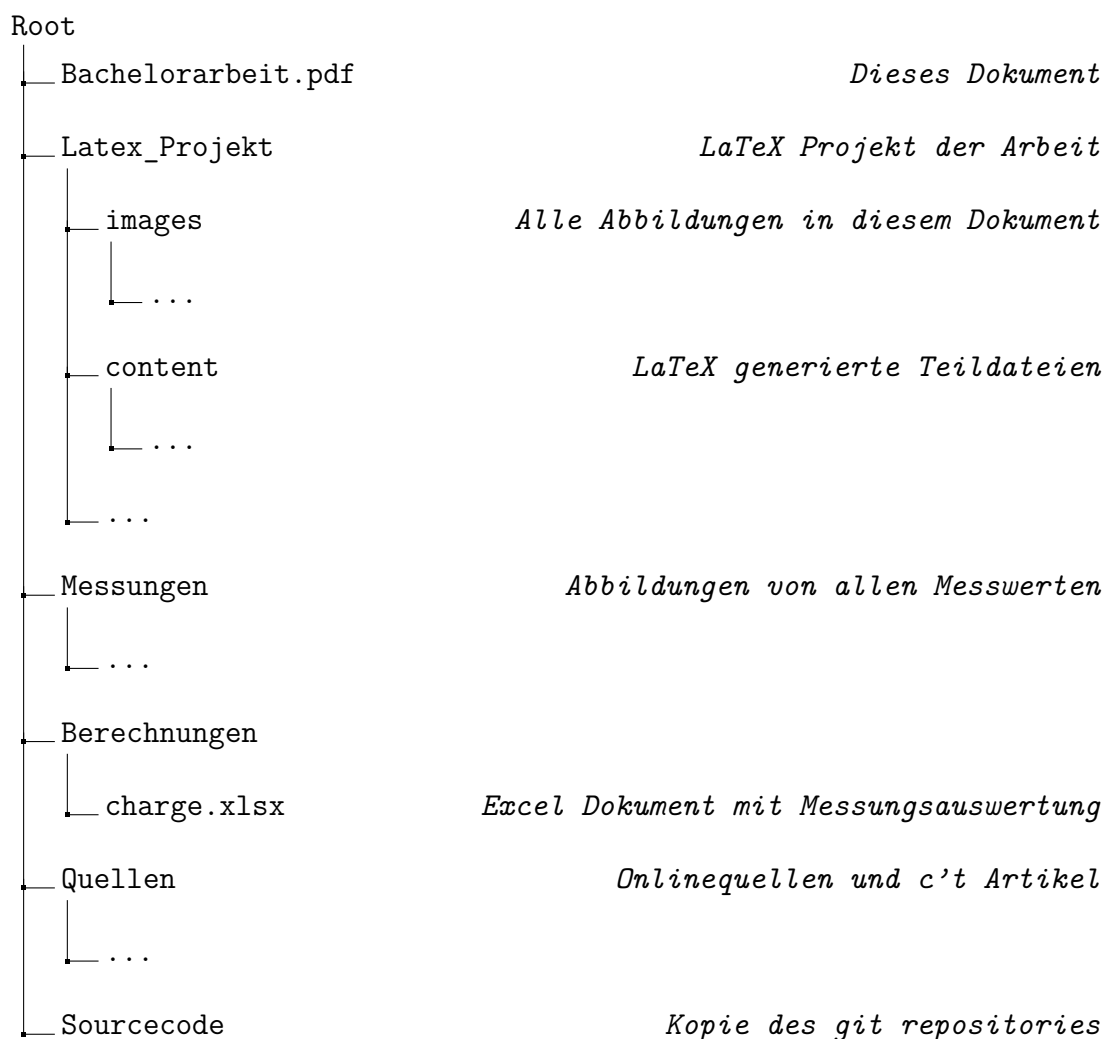


Abbildung A.9. – Lademessung 4A Versuch 3



CD Struktur

Im folgenden Teil wird die Ordnerstruktur der beiliegenden CD präsentiert.



Tabellenverzeichnis

3.1. Ergebnisse Beispielmessungen	14
3.2. Vergleich Messungen und Formel ($a = 0,0013$)	18
4.1. MoSCoW Akronym	21
4.2. Übersicht Anforderungen	25
6.1. Auswertung Anforderungen	58

Abbildungsverzeichnis

2.1. Projektstart UML-Klassendiagramm	7
3.1. Beispielverlauf CCCV Stromstärke und Spannung	13
3.2. Messung 4A Ladestrom mit Phasenkennzeichnung	15
5.1. GenericNode UML-Klassendiagramm	30
5.2. ChargingNode UML-Klassendiagramm	31
5.3. realistisches Ladeverhalten UML-Klassendiagramm	33
5.4. Nachrichten für Vorhersagen UML-Klassendiagramm	35
5.5. Containerklasse UML-Klassendiagramm	35
5.6. Anfrage für Multicopter UML-Klassendiagramm	36
5.7. Statistikeigenschaften UML-Klassendiagramm	37
5.8. Ressourcenverwaltung UML-Klassendiagramm	38
5.9. Ressourcenverwaltung UML-Klassendiagramm	39
5.10. Überblick UML-Klassendiagramm	40
5.11. Nachrichtentypen UML-Klassendiagramm	41
A.1. Lademessung 1A Versuch 1	I
A.2. Lademessung 1A Versuch 2	II
A.3. Lademessung 1A Versuch 3	II
A.4. Lademessung 2A Versuch 1	III
A.5. Lademessung 2A Versuch 2	III
A.6. Lademessung 2A Versuch 3	IV
A.7. Lademessung 4A Versuch 1	IV
A.8. Lademessung 4A Versuch 2	V
A.9. Lademessung 4A Versuch 3	V

List of Source Code

6.1. handleMessage() in ChargingNode.cc	44
6.2. appendToObjectsWaiting() in ChargingNode.cc	45
6.3. fillChargingSpots() in ChargingNode.cc	48
6.4. getNextWaitingObjectIterator() in ChargingNode.cc	50
6.5. charge() in ChargingNode.cc	52
6.6. clearChargingSpots() in ChargingNode.cc	53
6.7. rearrangeChargingSpots() in ChargingNode.cc	55

Literaturverzeichnis

- [Anda] ANDRÁS VARGA: *OMNeT++ Introduction*. <https://www.omnetpp.org/intro 2.2.1>
- [Andb] ANDRÁS VARGA: *OMNeT++ License*. <https://www.omnetpp.org/intro/license 2.2.1>
- [BCVP11] BRISTEAU, Pierre-Jean ; CALLOU, François ; VISSIÈRE, David ; PETIT, Nicolas: The Navigation and Control technology inside the AR.Drone micro UAV. In: *IFAC Proceedings Volumes* 44 (2011), Nr. 1, S. 1477–1484. <http://dx.doi.org/10.3182/20110828-6-IT-1002.02327>. – DOI 10.3182/20110828-6-IT-1002.02327. – ISSN 14746670 2.1.1
- [CS16] CHEN, Aoxia ; SEN, Pankaj K.: Advancement in battery technology: A state-of-the-art review. In: IEEE (Hrsg.): *2016 IEEE Industry Applications Society Annual Meeting*, IEEE, 2016. – ISBN 978-1-4799-8397-1, S. 1–10 3.1.1, 3.1.2, 3.1.3
- [Dan13] DANIEL BACHFELD: Quadrokooper-Know-how: Quadrokooper sind faszinierende Fluggeräte, in denen modernste Technik eingebaut ist. Wir erklären Komponente für Komponente, wie ein Quadrokooper funktioniert und was ihn in der Luft hält. In: *c't* (2013), Nr. 3, S. 42–53 2.1.1, 3.4
- [HCH99] HSIEH, Guan-Chyun ; CHEN, Liang-Rui ; HUANG, Kuo-Shun: Fuzzy-controlled active state-of-charge controller for fastening the charging behavior of Li-ion battery. In: IECON (Hrsg.): *IECON'99. Conference Proceedings. 25th Annual Conference of the IEEE Industrial Electronics Society (Cat. No.99CH37029)*, IEEE, 1999. – ISBN 0-7803-5735-3, S. 400–405 3.2, 3.5

- [ICA11] ICAO: *ICAO circular*. Bd. 328: *Unmanned aircraft systems: (UAS)*. Montréal : International Civil Aviation Organization, 2011. – ISBN 978–92–9231–751–5 2.1
- [IEC99] IECON (Hrsg.): *IECON'99. Conference Proceedings. 25th Annual Conference of the IEEE Industrial Electronics Society (Cat. No.99CH37029)*. IEEE, 1999 . – ISBN 0–7803–5735–3
- [IEE10] IEEE (Hrsg.): *2010 IEEE 72nd Vehicular Technology Conference - Fall*. IEEE, 2010 . – ISBN 978–1–4244–3573–9
- [IEE16a] IEEE (Hrsg.): *2016 IEEE Industry Applications Society Annual Meeting*. IEEE, 2016 . – ISBN 978–1–4799–8397–1
- [IEE16b] IEEE (Hrsg.): *2016 IEEE Transportation Electrification Conference and Expo, Asia-Pacific (ITEC Asia-Pacific)*. IEEE, 2016 . – ISBN 978–1–5090–1272–5
- [Int12] INTERNATIONAL INSTITUTE OF BUSINESS ANALYSIS: *Leitfaden zur Business Analyse: IIBA BABOk Guide 2.0 = Business analysis body of knowledge*. Version 2.0. Gießen : Schmidt, 2012. – ISBN 9783921313817 4
- [KPNC16] KHAN, Abdul B. ; PHAM, Van-Long ; NGUYEN, Thanh-Tung ; CHOI, Woojin: Multistage constant-current charging method for Li-Ion batteries. In: IEEE (Hrsg.): *2016 IEEE Transportation Electrification Conference and Expo, Asia-Pacific (ITEC Asia-Pacific)*, IEEE, 2016. – ISBN 978–1–5090–1272–5, S. 381–385 3.2
- [VLG⁺10] VAZQUEZ, Sergio ; LUKIC, Srdjan M. ; GALVAN, Eduardo ; FRANQUELO, Leopoldo G. ; CARRASCO, Juan M.: Energy Storage Systems for Transport and Grid Applications. In: *IEEE Transactions on Industrial Electronics* 57 (2010), Nr. 12, S. 3881–3895. <http://dx.doi.org/10.1109/TIE.2010.2076414>. – DOI 10.1109/TIE.2010.2076414. – ISSN 0278–0046
- [YXQX10] YAN, Jingyu ; XU, Guoqing ; QIAN, Huihuan ; XU, Yangsheng: Battery Fast Charging Strategy Based on Model Predictive Control. In: IEEE (Hrsg.): *2010 IEEE 72nd Vehicular Technology Conference - Fall*, IEEE, 2010. – ISBN 978–1–4244–3573–9, S. 1–8 3.2, 3.5

Selbstständigkeitserklärung

Ich, Ludwig Breitsprecher, Matrikelnummer 54131, erkläre hiermit , dass ich die vorliegende Bachelorarbeit mit dem Titel

Konzeptionierung und Implementierung einer intelligenten Ladestation in einer diskreten eventbasierten Simulationsumgebung

selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Gedanken, die aus fremden Quellen direkt oder indirekt übernommen wurden, sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ilmenau, 05.01.2018

LUDWIG BREITSPRECHER