Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Dennis Schröder

Agentenorientierte Softwareentwicklung* im Kontext der *Multi-Roboter-Wegplanung

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Dennis Schröder

Agentenorientierte Softwareentwicklung* im Kontext der *Multi-Roboter-Wegplanung

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Martin Becke
Zweitgutachter: Prof. Dr.-Ing. Andreas Meisel

Eingereicht am: 16. April 2019

Dennis Schröder

Thema der Arbeit

Agentenorientierte Softwareentwicklung im Kontext der *Multi-Roboter-Wegplanung*

Stichworte

Agentenorientierte Softwareentwicklung, AoSE, Agenten, Multi-Roboter-Wegplanung, JADE, CoDy-Algorithmus, Smart Chair, CaDS

Kurzzusammenfassung

Internet of Things (IoT), *Ubiquitous Computing* und *Industrie 4.0* sind Begriffe, die in letzter Zeit immer mehr an Bedeutung gewinnen. Gemeinsam haben diese Themengebiete vor allem ihren Fokus auf der Vernetzung von Rechnern. Die Anforderung an die Skalierbarkeit solcher Systeme wird mit der Anzahl der Geräte und dem Grad der Vernetzung steigen und somit zukünftig immer wichtiger werden, auch mit dem Blick auf das Jahr 2022, in dem 50 Milliarden IoT-Geräte aktiv sein sollen.

Im Rahmen dieser Bachelorarbeit wird ein solches skalierendes System entwickelt [1]. Eine Kernanforderung an das System ist es, mit unterschiedlichen Anzahlen von Geräten souverän umgehen zu können. Dabei reicht die Anzahl von wenigen Einzelnen bis vielen Hunderten. Um diesen Grad der Skalierbarkeit zu erreichen, wird das System mit Hilfe der *agentenorientierten Softwareenticklung* (AoSE) umgesetzt. Als Produkt entsteht ein Multi-Agenten-System, das durch das *Java Agent Development Framework* (JADE) implementiert wird. Zusammen mit dem *Cooperative Dynamic* (CoDy)-Algorithmus entsteht ein kooperatives System aus Agenten die sich in einer zweidimensionalen *Gridworld* bewegen können.

Als Beispiel für die gute Skalierbarkeit der AoSE, wird ein bestehendes Projekt erweitert. Dieses ist das „*Smart Chairs*“-Projekt der *Communication and Distributed System* (CaDS)-Arbeitsgruppe der *Hochschule für angewandte Wissenschaften Hamburg*. Dieses Projekt, wird im Rahmen dieser Arbeit, um die Fähigkeit an bestimmte Plätze fahren zu können erweitert.

In einer selbst entwickelten Simulationsumgebung werden dann Experimente aus [3] wiederholt. Anhand der Ergebnisse, den zuvor definierten qualitativen Merkmalen und

[1]Das resultierende *Java*-Projekt ist im *GitHub* Repository https://github.com/BeefMediumRare/AoSE_SmartChair öffentlich zur Verfügung gestellt.

Anforderungen, sowie den Kerneigenschaften des Multi-Agenten-Systems, Dezentralität und Autonomie, wird dann die Skalierbarkeit diskutiert und das Projekt bewertet.

Dennis Schröder

Title of the paper

Agent-oriented software engineering in the context of *mutli-robot path planning*

Keywords

Agent-oriented software engineering, AoSE, Agents, Mutli-robot path planning, JADE, CoDy Algorithm, Smart Chair, CaDS

Abstract

Internet of Things (IoT), *ubiquitous computing* and *Industry 4.0* are terms that are gaining importance lately. These topics share a focus on more complex network requirements. The scalability requirement of such systems will increase with the number of devices and the degree of networking and thus become more important in the future. Experts estimate, there will be 50 billion active IoT devices in 2022.

A system with such scaling capabilities will be developed within the scope of this bachelor thesis [2]. A core requirement of the system is to handle different numbers of devices confidently. The range varies from a few to many hundreds individuals. To achieve this level of scalability, the system is implemented using *agent-oriented software engineering* (AoSE). The product is a multi-agent system, which is implemented with the *Java Agent Development Framework* (JADE). Together with the *Cooperative Dynamic* (CoDy) algorithm, it creates a cooperative system of agents that can move in a two-dimensional *gridworld*.

The project, which demonstrates the scalability of the AoSE approach, addresses the extension of the *Smart Chairs* of the *Communication and Distributed System* (CaDS) team from the *Hamburg University of Applied Sciences* to be able to drive to certain places.

Experiments from [3] will be repeated in a self developed simulator. Based on the results, the previously defined qualitative characteristics and requirements as well as the core properties of the multi-agent system, decentralization and autonomy, the scalability is discussed and the project evaluated.

[2]The resulting *Java*-project is publicly available on the *GitHub* repository https://github.com/BeefMediumRare/AoSE_SmartChair.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation . . . 1
1.1.1 Relevanz . . . 1
1.1.2 Anwendungsfall . . . 1
1.2 Anforderungen . . . 2
1.3 Kapitelkurzzusammenfassungen . . . 3

2 Theoretischer Hintergrund **4**
2.1 Agentenorientierte Softwareentwicklung . . . 4
2.1.1 Agent . . . 4
2.2 JADE . . . 5
2.2.1 Grundlagen . . . 6
2.2.2 Behaviour . . . 7
2.2.3 Kommunikation . . . 8
2.3 CoDy-Algorithmus . . . 9
2.3.1 Grundlagen . . . 9
2.3.1.1 Das Weltmodell . . . 9
2.3.1.2 Die Entfernungskarte . . . 10
2.3.1.3 Die Umgebungskarte . . . 11
2.3.1.4 Die Erreichbarkeitskarte . . . 12
2.3.1.5 Der Raum-Zeit-Pfad . . . 12
2.3.2 Konfliktverarbeitung . . . 14
2.3.3 Kommunikation . . . 14
2.3.4 Prioritäten . . . 15
2.3.5 Algorithmus . . . 16
2.3.6 Abweichungen . . . 17
2.3.7 Zeitkomplexität . . . 18

3 Software-Architektur **19**

4 Methodik der Evaluation **21**
4.1 Validierung . . . 21
4.2 Simulationsumgebung . . . 21

4.3 Metriken 22

5 Experimente **24**
5.1 Allgemeine Experimente 25
5.1.1 Kurze Engstelle 25
5.1.2 Umweg 26
5.1.3 Tunnel 27
5.1.4 Tunnel mit kleiner Ausweichbucht 27
5.1.5 Durchfahren einer stehenden Menge 28
5.1.6 Kreuzung 29
5.1.7 Beobachtungen 31
5.2 Experimente mit Messwerten 31
5.2.1 Sechs gegen sechs: Lockere Vorbeifahrt 31
5.2.2 Sechs gegen sechs: Enge Vorbeifahrt 32
5.2.3 Drei gegen drei: Enge Vorbeifahrt 33
5.2.4 Vier gegen vier: Enge Vorbeifahrt 33
5.2.5 Messergebnisse 34

6 Diskussion **37**

7 Fazit **40**
7.1 Ausblick 40

Tabellenverzeichnis

5.1 Messwerte der Experimente aus [3] 34
5.2 Messwerte der selbst durchgeführten Experimente 34

Abbildungsverzeichnis

2.1 *Entity-Relationship*-Diagramm aus [1] 6
2.2 Beispiel für eine kleine Entfernungskarte 10
2.3 Entfernungs- und Umgebungskarte 11
2.4 Beispiele für eine Erreichbarkeitskarte 12
2.5 Beispielhafte Wegplanung 13

5.1 Aufbau für das Durchfahren zweier Agenten durch eine kurze Engstelle . 25
5.2 Aufbau für ein Szenario, in dem ein Agent einen Umweg in Kauf nimmt . 26
5.3 Ausgangssituation für das Durchqueren eines Tunnels von zwei Gruppen, bestehend aus jeweils sechs Agenten 27
5.4 Aufbau für die Vorbeifahrt zweier Agenten in einem Tunnel mit einer kleinen Ausweichbucht 28
5.5 Aufbau für das Durchfahren eines Agenten durch ein Menge stehender Agenten 29
5.6 Aufbau für das Passieren einer Kreuzung von vier Gruppen, bestehend aus jeweils vier Agenten 30
5.7 Aufbau für die lockere Vorbeifahrt zweier Gruppen, bestehend aus jeweils sechs Agenten 32
5.8 Aufbau für die enge Vorbeifahrt zweier Gruppen, bestehend aus jeweils sechs Agenten 32
5.9 Aufbau für die enge Vorbeifahrt zweier Gruppen, bestehend aus jeweils drei Agenten 33
5.10 Aufbau für die enge Vorbeifahrt zweier Gruppen, bestehend aus jeweils vier Agenten 34
5.11 Beispielhafte Totalblockade für das Experiment „5.2.2 Sechs gegen sechs: Enge Vorbeifahrt„ 35
5.12 Prioritätsverlauf aller Agenten für das Experiment „5.2.4 Vier gegen vier: Enge Vorbeifahrt“ aus [3] 36
5.13 Prioritätsverlauf aller Agenten für das Experiment „5.2.4 Vier gegen vier: Enge Vorbeifahrt“ der selbst durchgeführten Experimente 36

1 Einleitung

1.1 Motivation

1.1.1 Relevanz

Internet of Things (IoT) ist ein, sich rasant entwickelndes, Themengebiet. Über 50 Milliarden IoT-Geräte werden bis zum Jahr 2022 erwartet [9]. Der damit verbundene Ausbau des 5G-Netzes wird auch immer wichtiger. In beides wird momentan und in naher Zukunft viel Geld investiert [11]. IoT-Geräte sind häufig ereignisgesteuert und proaktiv, da sie Daten analysieren und diese beim menschlichen Endbenutzer melden.
Proaktivität ist in Softwareparadigmen wie z.B. der *objektorientierten Programmierung* (OOP) nicht als Konzept mit inbegriffen [6]. *Agentenorientierte Softwareentwicklung* (AoSE) hingegen ist ein Softwareparadigma, das, wie der Name vermuten lässt, Agenten als Kernkonzept hat. Agenten werden als der nächste evolutionäre Schritt von Objekten bezeichnet [6] und grenzen sich zu Objekten durch ihre Proaktivität und Autonomie ab [[4], [8] in [2]]. Vor dem Hintergrund, dass Autonomie auf Ereignissen fußt, lässt sich vermuten, dass AoSE, obwohl es in der Industrie und im akademischen Umfeld zwar noch kaum zum Einsatz kommt [5], potentiell eine langfristige Lösung für die neuen Ansprüche von IoT-Netzen darstellt.

1.1.2 Anwendungsfall

Die *Communication and Distributed System* (CaDS)-Arbeitsgruppe der *Hochschule für angewandte Wissenschaften Hamburg* (HAW) präsentiert ihre *Smart Chairs*[1] auf Veranstaltungen, um Aufmerksamkeit für sich und die HAW zu generieren. In erster Linie werden die

[1]Ein Bürostuhl, der mit verschiedenen (Druck-, Abstands- und Temperatur-) Sensoren [17]) sowie einem Antrieb ausgestattet ist.

Smart Chairs als IoT-Geräte betrachtet und dienen der Erforschung dieses Themengebiets. Die CaDS-Arbeitsgruppe möchte, dass die *Smart Chairs* sich autonom bewegen, um eine Funktion für den kürzlich hinzugefügten Antrieb zu haben. Dabei sollen die *Smart Chairs* bestimmte Positionen in einem Raum anfahren, um einen realen Anwendungsfall darstellen zu können. Dies könnte zum Beispiel das Fahren an Schreibtische sein. Je näher man einem realem beziehungsweise alltäglichem Szenario ist, desto interessanter und greifbarer wirkt die Anwendung. Diese grobe Definition wird im Rahmen dieser Arbeit nicht weiter vertieft, da der Fokus im Entwickeln eines *Proof of Concepts* (PoC) liegt und hierfür ein detailliert ausformulierter Anwendungsfall nicht erforderlich ist. Lediglich die groben Rahmenbedingungen müssen bekannt sein, welche im folgenden Kapitel festgehalten werden.

1.2 Anforderungen

In diesem Kapitel werden die Anforderungen an den Anwendungsfall beziehungsweise an das Softwaresystem aufgezählt.

1. **Skalierbarkeit** ist eine wichtige Anforderung an das System, da der Anwendungsfall die Anzahl der Teilnehmer nicht klar vorgibt. Das System soll also skalieren, um mit einem einzigen, sowie auch mit mehreren Teilnehmern umgehen zu können.
2. Das System soll nach endlicher Zeit **terminieren**. Die Positionen, die die *Smart Chairs* anfahren, sollen, wenn sie erreicht werden können, in endlicher Zeit erreicht werden. Dabei müssen die Wege nicht optimal bezüglich ihrer Zeit oder Strecke gewählt werden.
3. Das System soll in **Echtzeit** agieren. Damit soll vermieden werden, dass lange Pausen oder Initialisierungsphasen eintreten.
4. Das System soll mit Hilfe von AoSE umgesetzt werden. Die *Smart Chairs* sollen jeweils als **Agent** abgebildet werden.
5. Das System soll Hindernisse erkennen und ihnen ausweichen. Es soll also **nicht** zu **Kollisionen** kommen.

Zwar sind alle Anforderungen wichtig, aber sie können im Rahmen dieser Arbeit, nicht alle mit gleichem Gewicht bearbeitet werden. Der Fokus liegt auf der Skalierbarkeit und dem AoSE, da sie Teile der Forschungsfrage sind.

1.3 Kapitelkurzzusammenfassungen

Diese Arbeit gliedert sich in folgende weitere Kapitel:

2 Theoretischer Hintergrund gibt einen Einblick in die Theorie verwendeter Methodiken, Konzepte und Algorithmen. Außerdem wird die Wahl der jeweiligen Methodiken, Konzepte und Algorithmen begründet.

3 Software Architektur beschreibt den Aufbau der eigenen Implementierung und erörtert wichtige Kernelemente.

4 Methodik stellt die Validierungsmethode vor und beschreibt den Simulator. Zusätzlich werden, die für die Experimente wichtigen, Metriken vorgestellt.

5 Experimente beschreibt den Aufbau der Experimente, stellt die erwarteten und eigenen Beobachtungen und Messwerte vor.

6 Diskussion diskutiert die Beobachtungen der Experimente im Hinblick auf die Fragestellung. Zusätzlich wird die Anwendung anhand der formulierten Anforderungen bewertet.

7 Fazit fasst den Kern der Arbeit zusammen und gibt einen Ausblick.

2 Theoretischer Hintergrund

2.1 Agentenorientierte Softwareentwicklung

Ein Agent ist für die AoSE, dass was ein Objekt für die OOP ist. AoSE ist, so wie die OOP auch, ein Programmierparadigma. Um AoSE zu verstehen, wird folgend der Begriff des Agenten definiert.

2.1.1 Agent

Es existiert keine universelle Definition für Agenten [2], jedoch gibt es Eigenschaften die einen Agenten beschreiben, die von diversen Definitionen aufgegriffen werden. Ein Agent ist:

- **zielorientiert.** Er hat ein oder mehrere Ziele. Sein Verhalten ist darauf ausgelegt, diese zu erfüllen.
- **autonom.** Ein Agent kontrolliert seinen internen Status und entscheidet selbständig, ob und welche Aktion er ausführt.
- **reaktiv.** Auf Änderungen der Umwelt kann reagiert werden.
- **proaktiv.** Der Agent ergreift Eigeninitiative. Es sind also keine Befehle von außen notwendig, um zu handeln.
- **sozial.** Er interagiert „menschenähnlich“ mit anderen Agenten. Das bedeutet, dass Agenten verhandeln, koordinieren, kooperieren und so weiter.

[2][1][6]

Eingangs wurde der Vergleich mit OOP gezogen. Objekte und Agenten teilen sich tatsächlich viele Merkmale. Objekte kapseln ihre Identität und ihren Status. Sie kontrollieren aber

nicht ihr Verhalten. Die Methoden der Objekte werden von außen und potenziell zu jeder Zeit aufgerufen. Agenten kapseln dies jedoch zusätzlich. Sie verfügen selbständig über ihre Methoden. Sie treten aber auch in komplexe Verhandlungen mit anderen Agenten. [6]
Ein weiterer Aspekt der sie von Objekten abgrenzt. Mit einer Nachricht an ein Objekt, bezweckt der Absender, das Ausführen einer bestimmten Methode. Im Gegensatz zu einer Nachricht an einen Agenten, muss die Nachricht an ein Objekt exakt einer Formatierung entsprechen. Objekte sind in ihrer Kommunikation also weniger flexibel als Agenten. Das Verhältnis von AoSE zur OOP kann man wie das Verhältnis zwischen der iterativen und der objektorientierten Programmierung beschreiben. Deshalb wird die AoSE auch als nächster evolutionärer Schritt bezeichnet. [7]

Über die Zeit ist es immer wichtiger geworden lose Kopplung und hohe Kapselung in Software zu erreichen. AoSE führt diesen Trend konsequent weiter, indem eine Entität seinen eigenen Kontrollfluss kontrolliert und damit seine Absichten kapselt. Diese Attribute gewinnen an Gewicht, je komplexer ein Softwaresystem ist. [6]

In dieser Arbeit wird mit Hilfe der AoSE ein Multi-Agenten-System entwickelt.

2.2 JADE

Die *Foundation for Intelligent Physical Agents* (FIPA) der *IEEE Computer Society* hat im Bereich des Multi-Agenten-Systems mehrere Standards erarbeitet. Diese Standards beschreiben alle grundlegenden Elemente und Funktionen, die für eine Multi-Agenten-Plattform benötigt werden. [6]

Die FIPA listet einige Projekte auf, die diese Standards implementieren [10]. Aktuell stehen lediglich das *JACK*- und *Java Agent Development* (JADE)-Framework zur Verfügung. *JACK* ist ein in *Java* implementiertes und gut dokumentiertes Framework. Es ist jedoch ein Produkt einer Firma und die Verwendung ist daher mit Lizenzen verbunden. [12] Auch JADE ist in *Java* implementiert und somit plattformunabhängig [13]. Die Benutzung ist nicht mit Lizenzkosten verbunden, da es open-source ist. Es kann also eigenständig verändert und erweitert werden. Zudem ist JADE in Form von [1] detailliert beschrieben. Aufgrund dieser Tatsachen, wird die in dieser Arbeit entstehende Anwendung mit dem JADE-Framework umgesetzt.

In den folgenden Kapiteln wird eine Auswahl wichtiger Grundkonzepte des JADE-Framworks vorgestellt.

2.2.1 Grundlagen

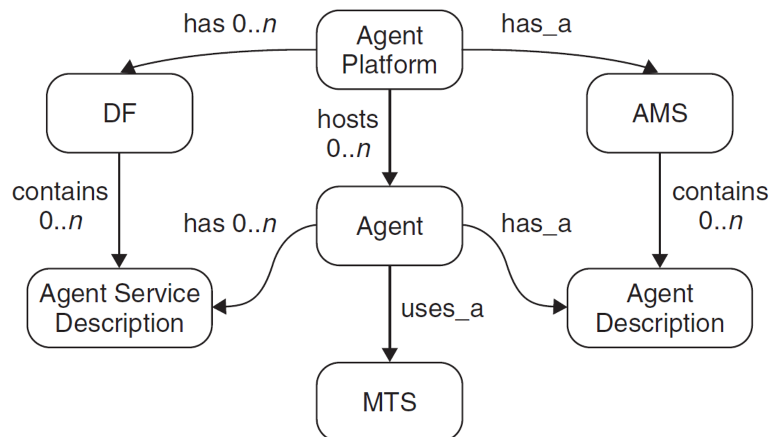

Abbildung 2.1: *Entity-Relationship*-Diagramm aus [1]

***Agent Platform* (AP)** Die AP beschreibt die physikalische Infrastruktur, in der die Agenten ausgeführt werden. Hier sind die Rechner, die Netzwerke, die Betriebssysteme, das *Agent Management System*, die Agenten selbst sowie zusätzliche Software mit inbegriffen. [1]

Agent Ein Agent existiert in der AP und bietet ein oder mehrere Services, die mit Hilfe einer *Service Description* veröffentlicht werden. Ein Agent hat ein eindeutigen *FIPA Agent Identifier* (AID). [1]

Diretory Faciliator (DF) Der DF ist eine optionale Komponente. Der DF stellt den Agenten einen „Gelbe Seiten“-Service zur Verfügung und hält eine komplette Liste aller Agenten. Der „Gelbe Seiten“-Service wird von den Agenten genutzt, um ihre Services zu registrieren und somit den anderen Agenten verfügbar zu machen. Die AP kann beliebig viele DF starten, die ihre Daten untereinander synchronisieren. [1]

Agent Management System (AMS) Das AMS ist für das Erstellen und Löschen der Agenten verantwortlich. Jeder Agent muss sich beim AMS registrieren. Bei diesem Schritt vergibt das AMS dann die AID für die Agenten. Ein Agent beendet sich, wenn er sich vom AMS abmeldet. Das AMS ist als eine Entität zu verstehen und erstreckt sich auch über mehrere Rechner. [1]

Message Transport Service (MTS) Dieser Service wird von der AP bereitgestellt. Über ihn können die Agenten Nachrichten austauschen. [1]

2.2.2 Behaviour

Ein *Behaviour* ist eine Aufgabe, die ein Agent ausführen kann und als erbende Klasse von `jade.core.behaviours.Behaviour` implementiert ist. Für jeden Agenten startet die JADE Umgebung einen Thread. Ein *Behaviour* wird über einen Scheduler auf diesem Thread ausgeführt. Der Agent fügt ein *Behaviour* mit `addBehaviour()` der Scheduling-Queue hinzu. Dies kann der Agent während der Initialisierungsphase in der `setup()` Methode tun oder innerhalb eines *Behaviours*. Jede *Behaviour*-Subklasse muss die `action()` und `done()` Methode implementieren. Die `action()` Methode enthält die Logik, also den auszuführenden Code, des *Behaviours*. Die `done()` Methode gibt einen `boolean` zurück, der darüber informiert, ob das *Behaviour* seine Aufgabe abgeschlossen hat und damit aus Scheduling-Queue entfernt werden soll. Ein Agent kann mehrere *Behaviour* ausführen. Diese sind jedoch nicht unterbrechbar. Wenn die `action()` Methode vom Scheduler aufgerufen wird, kann diese nicht unterbrochen werden. Das *Behaviour* muss also selbständig diese Ressource wieder freigeben. [1]

Das JADE-Framwork stellt jedoch nicht nur den Basistypen zur Verfügung, sondern implementiert weitere abstrakte *Behaviour*. Folgend wird eine Auswahl aus [1] vorgestellt.

One-Shot Behaviour Für das `jade.core.behaviours.OneShotBehaviour` muss eine erbende Klasse nur die `action()` Methode implementieren. `done()` liefert standardmäßig `true` zurück. Ein *One-Shot Berhaviour* wird also nur einmal ausgeführt.

Cyclic Behaviour Das `jade.core.behaviours.CyclicBehaviour` ist dem *One-Shot Behaviour* recht ähnlich. Die `done()` Methode liefert jedoch standardmäßig `false` zurück. Ein *Cyclic Behaviour* beendet sich also nicht automatisch.

Ticker Behaviour Das `jade.core.behaviours.TickerBehaviour` implementiert sowohl `action()` als auch `done()` Methoden. Die `done()` Methode liefert immer `false` zurück. Die `action()` Methode führt die `onTick()` Methode periodisch aus. Das Zeitintervall wird über den Konstruktor definiert. Das *Ticker Behavior* ist selbst eine abstrakte Klasse. Erbende Klassen implementieren die Methode `onTick()`.

2.2.3 Kommunikation

Agenten interagieren miteinander. Dies geschieht indirekt über das Verändern der Umwelt oder durch direkte Kommunikation. Die direkte Kommunikation ist wahrscheinlich eines der wichtigsten Funktionen des JADE-Frameworks. Die Kommunikation zwischen Agenten erfolgt durch den asynchronen Nachrichtenaustausch. Jeder Agent besitzt eine Queue, in der alle Nachrichten eines Agenten empfangen werden. Eine Nachricht enthält dabei mehrere Felder:

- Die ID des **Senders**
- Eine Liste, die alle **Empfänger** enthält.
- Die **Absicht** der Nachricht. Die FIPA definiert hier eine Liste mit Möglichkeiten. Zum Beispiel „Inform„. Der Sender teilt den Empfänger einen Fakt mit.
- Den **Inhalt**, den der Sender mitteilen möchte.
- Die **Kodierung** der Nachricht, so, dass die Empfänger wissen, wie die Nachricht zu lesen ist.

[1]

2.3 CoDy-Algorithmus

Ein Wegplanungs-Algorithmus kann entweder verteilt oder zentral arbeiten. Zentrale Algorithmen skalieren in der Regel aber schlecht, da sie abhängig von der Anzahl der Teilnehmer sind [3]. Es ist für diese Arbeit also nicht interessant, da ein potentiell stark skalierendes Agenten-System zum Einsatz kommt. Ein echtzeitfähiger, verteilter und skalierender Algorithmus ist der CoDy-Algorithmus. Dieser berechnet die Wege für jeden Agenten aus Sicht der jeweiligen Agenten und löst mit Hilfe von heuristischer Prioritätsanpassung auftretende Konflikte. Der CoDy-Algorithmus hat bestimmte Voraussetzungen an das implementierende System. Es soll verteilt sein, muss den Agenten die Möglichkeit bieten, untereinander zu kommunizieren und homogen sein. Zusätzlich muss ein Agent in der Lage sein seine Umgebung zu erfassen, entweder durch die eigene Sensorik oder durch Kommunikation mit anderen Agenten. [3] Alle Voraussetzungen können erfüllt werden.

Die Funktionsweise des CoDy-Algorithmus wird in den folgenden Unterkapiteln genauer beschrieben. Es soll aber nicht Sinn dieser Arbeit sein, die Dissertation [3] wiederzugeben. Die grundlegenden Elemente und Funktionen werden nicht nur zum Grundverständnis wiedergegeben. Vor allem soll aber eine Basis geschaffen werden, auf deren Grund dann im Kapitel 3 beschrieben werden kann, wie der Algorithmus umgesetzt wird und wo es zu Abweichungen kommt.

2.3.1 Grundlagen

In diesem Kapitel werden die grundlegenden Mechanismen und Elemente des CoDy-Algorithmus vorgestellt.

2.3.1.1 Das Weltmodell

Das Weltmodell ist zeitlich und räumlich diskret. Der CoDy-Algorithmus kann mit jedem Weltmodell arbeiten, das sich als ungerichteter Graph darstellen lässt. Um die Berechnungen und Anschauungen aber nicht unnötig komplex zu gestalten, beschränkt sich das Weltmodell auf ein klassisches zweidimensionales Gitter. Die Zellen des Gitters sind quadratisch und die Nachbarschaftsbeziehungen der Zellen beschränken sich auf vier Himmelsrichtungen. Bedeutet also, dass Zellen, die diagonal zu einander liegen nicht als benachbart

gelten. Eine Zelle kann verschiedene Zustände einnehmen. Die Zustände variieren jedoch zwischen Kartentypen. Sie werden deshalb in den folgenden Kapiteln erörtert. Ein weiteres Attribut der Zellen ist, dass sie eine bestimmte räumliche Größe haben. [3] Für diese Arbeit wird jedoch angenommen, dass die Zellen ein wenig größer als die Agenten sind. Dies ist das einfachste anzunehmende Modell [3].

2.3.1.2 Die Entfernungskarte

Jeder Agent hält eine individuelle Entfernungskarte. Die grundlegende Aufgabe dieser Karte ist es, die Entfernung zum Ziel des Agenten wiederzugeben. Sie ist als Tabelle zu verstehen, in der für jede Zelle notiert ist, wie viele Bewegungsschritte nötig wären, um das Ziel von dieser Zelle aus zu erreichen. Diese Entfernungen werden durch eine einfache Breitensuche ermittelt. Die aktuelle Position des Agenten ist dabei nicht interessant. Der CoDy-Algorithmus ist auch in der Lage mit unbekannten Umgebungen umzugehen. [3] Zum einen ist das für den Anwendungsfall in erster Linie nicht interessant und zum anderen würde dies die Komplexität unnötig erweitern. Deshalb ist eine Entfernungskarte, im Kontext dieser Arbeit, eine Karte, die die gesamte Umgebung erfasst.

8	9			
9				
8		Z	1	2
7			2	3
6	5	4	3	4

Abbildung 2.2: Beispiel für eine kleine Entfernungskarte

Abbildung 2.2 zeigt beispielhaft eine Entfernungskarte. Die grauen Zellen stellen statische Hindernisse dar. Das „Z" ist das Ziel des Agenten. Die Zahlen sind die Entfernungen und weiße Zellen ohne Ziffern sind nicht erreichbar.

2.3.1.3 Die Umgebungskarte

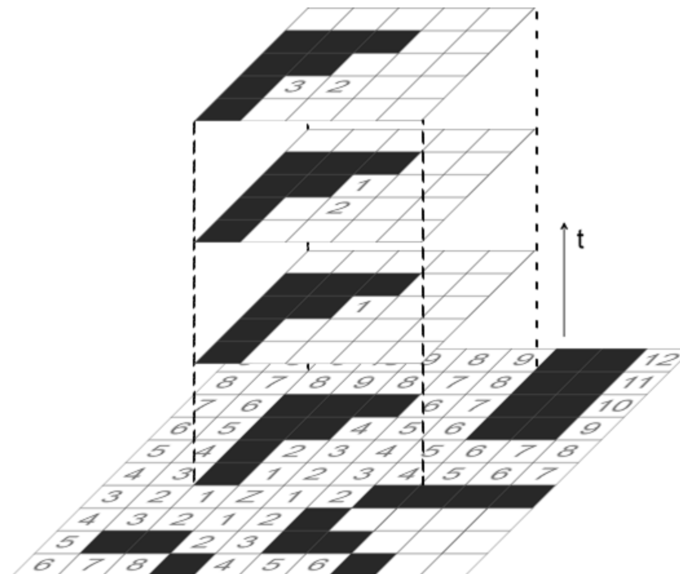

Abbildung 2.3: Entfernungs- und Umgebungskarte

Die Umgebungskarte stellt einen Ausschnitt aus der Entfernungskarte dar. Im Gegensatz zur Entfernungskarte besitzt die Umgebungskarte eine Zeitdimension. Damit ist es möglich dynamische Objekte, also andere Agenten, in die Wegplanung mit einzubeziehen. Dies beschreibt auch die Hauptfunktion dieser Karte. In die Umgebungskarte werden die, über Nachrichten von den anderen Agenten übermittelten, geplanten Wege eingetragen und für die eigene Wegplanung bereitgestellt. Die Dimensionen der Umgebungskarte sind immer ungerade, da die mittlere Zelle die aktuelle Position des Agenten spiegelt. Die Zeitdimension t zeigt von dem aktuellen Zeitpunkt aus in die Zukunft, wird aber durch die zeitlichen Berechnungstiefe t_{max} begrenzt. Abbildung 2.3 zeigt, dass die statischen Hindernisse von der Entfernungskarte übernommen werden, die Entfernungen jedoch nicht. Die Zellen halten andere Daten. Eine Zelle der Umgebungskarte ist entweder frei, von einem statischen Hindernis belegt oder von einem anderen Agenten reserviert. [3]

2.3.1.4 Die Erreichbarkeitskarte

Die Erreichbarkeitskarte ist eine Erweiterung der Umgebungskarte und wird bei der Planung des eigenen Weges erstellt. Sie gibt für jeden Zeitpunkt Auskunft, welche Zellen mit wie vielen Wegschritten von der aktuellen Position aus erreichbar sind. Da nicht davon auszugehen ist, dass ein Agent eine Zelle in einem Zeitschritt vollständig verlässt und die benachbarte Zelle vollständig betritt, muss der planende Agent seine Bewegungsschritte für t und $t+1$ reservieren. Das hat zur Folge, dass Agenten nicht direkt hintereinander fahren können und ist einer der beiden Mechanismen, wie Kollisionen zwischen den Agenten verhindert werden. [3]

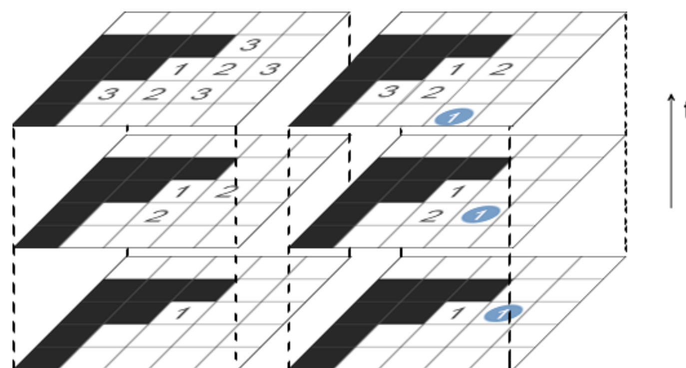

Abbildung 2.4: Beispiele für eine Erreichbarkeitskarte

Abbildung 2.4 zeigt zwei Erreichbarkeitskarten. Für die linke Karte sind keine Agenten eingetragen. In der rechten Karte ist der geplante Weg für „Agent 1“ eingetragen (in der Abbildung blau). Im Vergleich zeigt sich, dass wegen des Sicherheitsabstands nach zwei Zeitschritten auf der rechten Karte weniger Zellen erreicht werden können.

2.3.1.5 Der Raum-Zeit-Pfad

Der Raum-Zeit-Pfad ist das Datenmodell, das die geplanten Wege der Agenten beschreibt [3]. Dieses fußt zwar auf dem Weltmodell, stellt sich aber nicht als Gitter dar. Die Positionen werden in einem assoziativen Datenfeld, mit dem Schlüssel t, gespeichert. Zusätzlich zu der

Definition des Raum-Zeit-Pfades wird in diesem Kapitel kurz erläutert, wie dieser bestimmt wird.

Zuerst wird die Umgebungskarte zeitlich um $t + 1$ und räumlich so verschoben, dass die aktuelle Position des Agenten im Zentrum der Umgebungskarte liegt. Dann werden die alten Raum-Zeit-Pfade der anderen Agenten durch die Neuen ersetzt. Jetzt kann die Erreichbarkeitskarte entwickelt werden. Um den eigentlichen Weg zu planen, wird nun die letzte Zeitebene der Erreichbarkeitskarte betrachtet. Von den Zellen, die erreichbar sind, also einen Wert halten, wird diejenige ausgesucht, die die kleinste Entfernung in der Entfernungskarte hat. Von der gewählten Zelle aus kann man dann den Weg rückwärts durch die Zeit entwickeln. Wenn beim Entwickeln des Weges mehrere Zellen für einen Schritt in Frage kommen, wird die Zelle mit der niedrigsten Priorität gewählt. Wenn immer noch eine Auswahl besteht, dann gilt „Rechts vor Links“. Der Agent bestimmt die grobe Richtung zum lokalen Ziel und danach die relative Lage der Zellen. Die Zelle wird dann nach der folgenden Reihenfolge ausgesucht: rechts, vorne, links, hinten. [3]
Nun kann es aber passieren, dass keine Zelle frei ist, oder dass es zu Konflikten mit anderen Agenten kommt. Das wird im Kapitel 2.3.2 vertieft.

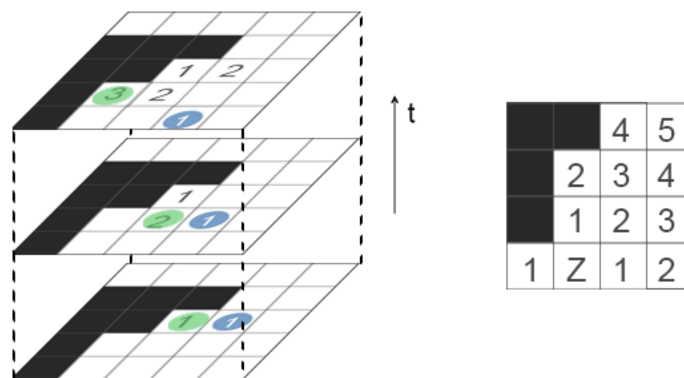

Abbildung 2.5: Beispielhafte Wegplanung

In Abbildung 2.5 ist die Wegplanung eines Agenten beispielhaft dargestellt. Die Abbildung besteht aus drei Teilen. Zum einen links, die Erreichbarkeitskarte aus Abbildung 2.4, in grün markiert der geplante Weg und rechts ein relevanter Ausschnitt aus der zugehörigen Entfernungskarte aus Abbildung 2.3. Die Zelle, die in der Erreichbarkeitskarte in drei

Schritten erreichbar ist, besitzt die geringste Entfernung zum globalen Ziel. Diese wird als lokales Ziel markiert. Von hier kann nun in der Erreichbarkeitskarte der Weg rückwärts durch die Zeit hin zur aktuellen Position des Agenten entwickelt werden.

2.3.2 Konfliktverarbeitung

Die Agenten des *CoDy-Algorithmus* sind kooperativ. Sie sind also in der Lage die eigenen „Bedürfnisse" zurückzustellen. Man kann zwischen passiver und aktiver Kooperation unterscheiden. Bei der aktiven Kooperation kann ein Agent die Planung übernehmen und bei anderen Agenten nach Zustimmung oder Ablehnung fragen. Bei der passiven Kooperation müssen alle in einem Konflikt beteiligten Agenten die gesamte Situation berechnen und dann anschließend in einer Verhandlung ihre Lösungen diskutieren. Die passive Kooperation hat gegenüber der aktiven Kooperation vor allem den Vorteil, dass der Kommunikationsaufwand deutlich geringer ist. Sie ist aber auch robuster und der Berechnungsaufwand, so wie die passive Kooperation im CoDy-Algorithmus umgesetzt ist, ist geringer. Die echte passive Kooperation kann sehr ineffizient sein. Deshalb benutzt der CoDy-Algorithmus eine abgewandelte Form. Agenten, die potenziell in einen Konflikt geraten können, kommunizieren in einer festen Reihenfolge (siehe Kapitel 2.3.3) und planen nur ihre eigenen Wege. [3]
Folgende Beispielsituation: „Agent 0" überschreibt einen Teil des geplanten Weges von „Agent 1". Dann schickt „Agent 0" seinen neuen Plan an alle Agenten. Wenn „Agent 1" wieder an der Reihe ist, plant dieser mit den neuen Daten. Wann welcher Agent, welche Wege überschreiben kann, wird in Kapitel 2.3.4 beschrieben.

2.3.3 Kommunikation

Wie bereits erwähnt, kommunizieren die Agenten in einer festen Reihenfolge. In diesem Kapitel wird beschrieben, wie sich dieser verkette Ablauf ergibt. Jeder Agent merkt sich mit einem Zeitstempel, wann er mit seinem Berechnungsschritt begonnen hat. Nachdem ein Agent seinen Weg geplant hat, verschickt er den geplanten Weg und den Zeitstempel an alle Agenten und wartet auf Nachrichten der anderen Agenten. Wenn ein Agent eine Nachricht erhält, wird zunächst geprüft, ob sich die Umgebungskarten überschneiden. Wenn dies nicht der Fall ist, wird die Nachricht verworfen, da die Agenten sich nicht

beeinflussen. Wenn aber eine Überschneidung festgestellt wird, werden die Zeitstempel miteinander verglichen. Hat der andere Agent früher mit seiner Wegplanung begonnen, muss auf diesen gewartet werden, bevor mit dem Planen der nächsten Wegschritte begonnen wird. Sind beide Zeitstempel gleich, entscheidet die numerische Identifikationsnummer der Agenten. Kleine Nummern sind besser. Somit ergibt sich ein zufälliger, aber fester verketteter Ablauf. Agenten reihen sich also in diesen Ablauf ein, wenn sich die Umgebungskarten überschneiden. Wenn sich die Agenten aber soweit voneinander entfernen, dass keine Überschneidung mehr vorliegt, werden sie aus der Kette gelöscht. Durch dieses Vorgehen ist die Entwicklung der Prioritäten, das Hauptwerkzeug der Agenten, um Konflikte zu lösen, vorhersehbar. Dieser Vorteil wiegt mehr, als der Nachteil für diese Bereiche die parallele Berechnung aufzugeben. [3]

2.3.4 Prioritäten

Agenten sind in der Lage, geplante Wege anderer Agenten zu überschreiben. Dies wird durch die Prioritätswerte der Agenten ermöglicht. Wenn die Erreichbarkeitskarte erstellt wird, werden Zellen, die von Agenten mit niedrigerer Priorität als die eigene reserviert sind, als frei betrachtet. Beim Planen des Weges wird aber darauf geachtet, dass möglichst wenig Wege überschrieben werden. Wenn zum Beispiel zwei Zellen die Entfernung zum Ziel gleichermaßen verringern und eine von diesen Zellen tatsächlich frei ist, dann wird diese gewählt. [3]

Die dynamische Anpassung der Prioritäten passiert genau dann, wenn ein Konflikt beziehungsweise eine Blockade erkannt wird. Die Konflikterkennung ist dabei recht simpel. Um einen Konflikt zu erkennen, betrachtet man den vom Agenten geplanten Weg. Wenn jeder geplante Schritt den Abstand zum Ziel verringert, liegt **keine Blockade** vor. Der Prioritätswert wird um den Wert $PrioNoBlock$ dekrementiert. Der Prioritätswert eines Agenten kann dabei den Wert $BasePrio$ nicht unterschreiten. Das Dekrementieren des Prioritätswert geschieht, da der Agent eventuell andere Agenten blockiert. Wenn der geplante Weg Schritte enthält, die die Entfernung zum Ziel nicht verbessern, der Agent aber bis zum Schluss in Bewegung bleibt, handelt es sich um einen **Umweg**. In diesem Fall wird die Priorität nicht verändert, da die Agenten ja kooperativ arbeiten und Umwege in Kauf genommen werden sollen. Wenn der geplante Weg in einer Wartephase endet, handelt es sich um eine **Blockade**. Da es Agenten aber möglich sein soll, andere Agenten

verdrängen zu können, wird eine Blockade nur als solche erkannt, wenn in zwei aufeinander folgenden Berechnungsschritten die gleiche Blockade erkannt wird. In diesem Fall wird die Priorität des Agenten um $PrioBlock$ erhöht. Mit dieser neuen Priorität plant der Agent erneut seinen Weg. Wichtig ist, dass die Priorität sich nur einmal erhöht. Diese erneute Planung ist für den Berechnungsschritt also die finale Planung. Wenn kein Weg gefunden wird, also nicht einmal das Verharren auf der eigenen Position möglich ist, dann liegt eine **Totalblockade** vor. In diesem Fall wird die Priorität um $PrioFullBlock$ erhöht und die Wegplanung erneut durchgeführt. Wenn immer noch kein gültiger Weg berechnet werden konnte, wird ein Notweg erzwungen. Der Notweg ist das Verharren auf der eigenen Position. Zusätzlich wird die Priorität auf $PrioMax$ erhöht, damit der Notweg nicht direkt überschrieben werden kann. Für das Überschreiben der Wege ist noch wichtig, dass die ersten drei Zeitschritte geschützt sind. In diesen ersten Zeitschritten dürfen keine Wege anderer Agenten überschrieben werden. Dies stellt die zweite Methode dar, um Konflikte zu vermeiden. Diese Regel gilt jedoch nicht für das Erzwingen eines Notweges. Zu dem Inkrementieren der Prioritäten ist noch anzumerken, dass der Prioritätswert durch $PrioMax$ begrenzt ist. Wenn dieser Wert überschritten wird, kommt es zu einem Prioritätsüberlauf und der Prioritätswert setzt sich zurück. [3]

Für die Parameter der Prioritäten sollte gelten: $PrioFullBlock >> PrioBlock >> PrioNoBlock$ [3].

2.3.5 Algorithmus

In diesem Kapitel werden die zuvor erläuterten Elemente in Kontext gesetzt.

Wenn die Agenten initialisiert werden, sind ihnen die statischen Hindernisse und die eigene Position bereits bekannt. Alle Agenten werden mit den gleichen Parametern erzeugt. Ein Agent führt periodisch die nächste geplante Bewegung aus. Weil für den ersten Berechnungsschritt die Agenten die Positionen der anderen nicht kennen, planen alle Agenten stehen zu bleiben. Zwischen den Bewegungen finden die Berechnungsschritte statt. Für diese Arbeit findet, aus Gründen der Übersicht, nur ein Berechnungsschritt pro Bewegungsschritt statt. Parallel zu den Bewegungs- und Berechnungsschritten, empfängt ein Agent die Nachrichten der anderen Agenten. [3]

Ein Berechnungsschritt läuft folgendermaßen ab:
Damit ein Berechnungsschritt beginnen kann, müssen von allen Agenten, auf die gewartet wird, Nachrichten empfangen werden. Wenn das erfüllt ist, wird die Umgebungskarte zeitlich und räumlich zentriert. Darauffolgend werden die neuen geplanten Wege in die Umgebungskarte eingetragen. Jetzt plant der Agent mithilfe einer Erreichbarkeitskarte den eigenen Weg. Prioritätsanpassungen und eventuelle Neuberechnungen werden gemäß 2.3.4 durchgeführt. Der geplante Weg wird an alle Agenten per Nachricht übermittelt. [3]

2.3.6 Abweichungen

An dieser Stelle werden Abweichungen in der Implementierung beschrieben und begründet.

In erster Iteration findet die Anwendung nur in einer Simulationsumgebung statt. Deshalb ist davon auszugehen, dass die Nachrichten der Agenten nicht verloren gehen. Daher wurde an den Stellen an denen ein „Timeout“ in [3] vorgeschlagen wird, darauf verzichtet.
Da die Anwendung auch ein PoC ist, werden einige Funktionen nicht umgesetzt. Zum einen das Verhalten wenn die Lokalisation oder die Kommunikation ausfällt und zum anderen das allgemeine Notfallverhalten.

Der CoDy-Algorithmus sieht vor, dass ein Agent zu Beginn seines Berechnungsschrittes, seinen eigenen Wegplan aus der Umgebungskarte löscht. Er setzt aber auch voraus, dass der Agent nach der Planung seines Weges, diesen Plan in der eigenen Umgebungskarte speichert. [3] Die eigene Implementation spart diesen Schritt, da die Nachricht die an die anderen Agenten verschickt wird, nur den Plan und nicht die gesamte Umgebungskarte enthält. Dies ist eine explizite Option in [3].

Die Prioritätswerte entwickeln sich in der eigenen Implementierung anders als in [3] vorgesehen. Wenn ein Agent einen neuen Berechnungsschritt beginnt und seinen Weg plant, soll laut [3], zuerst der alte Prioritätswert verwendet werden. Wenn dann erkannt wird, dass kein Konflikt vorliegt, wird die Priorität um $PrioNoBlock$ dekrementiert. Dabei ergibt sich folgendes Problem: Angenommen „Agent 1“ hat einen Prioritätswert von zehn. In seiner Umgebungskarte ist der Weg von „Agent 0“ mit einer Priorität von neun hinterlegt. „Agent 1“ plant seinen Weg mit der alten Priorität von zehn und überschreibt dabei einen Teil des Weges von „Agent 0“. Jeder Schritt in seiner Planung verringert die Entfernung zum Ziel. Es liegt also keine Blockade vor und der Prioritätswert wird um $PrioNoBlock = 1$

dekrementiert. Betrachtet man nun die Umgebungskarte, fällt auf, dass „Agent 1“ und „Agent 0“ die gleiche Priorität besitzen. „Agent 1“ hätte den Weg von „Agent 0“ also nicht überschreiben dürfen.
Um dieses Problem zu lösen, wird daher zu Beginn des Berechnungsschrittes die Priorität um $PrioNoBlock$ verringert. Das hat zur Folge, dass wenn der Verdacht auf eine Blockade besteht der Prioritätswert wieder um $PrioNoBlock$ inkrementiert wird, statt das sich die Priorität nicht verändert.

Zusätzlich ist bei der Prioritätsanpassung für den Fall, dass ein Notweg erzwungen wird nicht klar wie sich die Priorität entwickeln soll. An einer Stelle in [3] heißt es: „Dabei verwendet der Roboter die maximal möglichen Prioritätswerte, so dass dieser Notweg nicht sofort wieder überschrieben werden kann“. An anderer Stelle: „[...] es kommt dabei relativ häufig zur totalen Blockade, bei der ein Roboter keinen Weg mehr findet und einen Notweg mit *PrioFullBlock* erzwingen muss“ [3]. In der eigenen Implementation setzt sich die Priorität auf $PrioMax$.

2.3.7 Zeitkomplexität

In diesem Kapitel wird die Zeitkomplexität für einen Berechnungsschritt aus [3] wiedergegeben.

Beim Betrachten der Komplexität sind die Parameter Anzahl der Agenten Z_i; die zeitliche Berechnungstiefe t_{max} und die räumliche Dimension d der Umgebungskarte relevant. Da t_{max} ein festes Verhältnis zu d hat, kann es mit d substituiert werden. Da für die Planung des Weges nur die Agenten relevant sind, deren Umgebungskarten sich überlappen, kann Z_i durch den Ausdruck $(\frac{d}{2} - 1)^2$ beschrieben werden. Die Zeitkomplexität eines Berechnungsschrittes beträgt gesamt $O(d^5)$ und ist damit nur von der Dimension der Umgebungskarte abhängig. [3]

3 Software-Architektur

In diesem Kapitel wird erörtert, wie und aus welchen Elementen, sich das einstehende Softwareprojekt gliedert. Dabei wird jedoch nur die grobe Struktur des Projekts und die für den Anwendungsfall relevanten Aspekte beleuchtet.

Das Projekt ist in *Java* implementiert. Um grafische Benutzeroberflächen zu ermöglichen, wird *JavaFX* eingesetzt. Die Struktur des Projekts orientiert sich am bekannten *Model-View-Controller* (MVC)-Muster [16].

Kernaspekt des Projekts ist der Agent. Dieser ist in Form der Klasse `codyAgent.CoDyAgent` implementiert, die die Klasse `jade.core.Agent` erweitert. Ein CoDy-Agent implementiert, abgesehen von der Wegplanung, die Logik des CoDy-Algorithmus. Die Wegplanung ist in Objekten gekapselt, die den verschiedenen Kartentypen des CoDy-Algorithmus entsprechen. Diese werden jedoch direkt von dem Agenten gesteuert. Wie in [1] empfohlen, implementiert der CoDy Agent sein Verhalten als innere Klassen.

PlanningBehaviour Das *PlanningBehaviour* übernimmt mehrere Aufgaben. Zum einen führt es die Wegplanung aus, zum anderen wandelt es den geplanten Weg als *JSON* und dann als `String` um, übernimmt also das *Marshalling*. Außerdem wird dieser `String` als Nachricht an alle Agenten gesendet. Als Performativ wird hier „Informieren", also `jade.lang.acl.ACLMessage.INFORM`, genutzt. Das *PlanningBehaviour* ist ein `CyclicBehaviour` mit zwei Status. Die Status sind wie in [1] vorgeschlagen, als `switch case` umgesetzt. Befindet sich dieses Verhalten im ersten Status, dann wird darauf gewartet, dass von allen Agenten, auf die gewartet wird, eine Nachricht eintrifft. Im zweiten Status findet dann der Berechnungsschritt, also die Wegplanung, wie in 2.3.5 beschrieben, statt.

MessageReceiveBehaviour Auch das *MessageReceiveBehaviour* ist ein `CyclicBehaviour`. Dieses kontrolliert regelmäßig, ob neue Nachrichten vorliegen und übernimmt dann das *Unmarshalling*.

MovementBehaviour Dieses Behaviour ist ein `TickerBehaviour`. In regelmäßigen, periodischen Abständen wird der aktuelle Bewegungsplan gelesen. Das *MovementBehaviour* steuert dann über den *Hardware Abstraction Layer* (HAL), die nächste durchzuführende Bewegung.

CoDyAgentDiscoveryBehaviour Das *CoDyAgentDiscoveryBehaviour* ist ein `OneShotBehaviour`. Es sucht jene Agenten, die bei dem `DFService` einen CoDy-Service registriert haben und speichert diese in eine dem Agenten verfügbare Liste.

Ein Agent kann sich nicht selber instantiieren. Diese Aufgabe wird von der Simulationsumgebung übernommen. Die Simulationsumgebung ist Teil eines *Controllers* des *JavaFX*-Projekts. Der *Controller* stellt den HAL, startet die Simulation und steuert die grafischen Bewegungen der Agenten. Um eine Simulation zu starten, wird im ersten Schritt eine *JSON*-Datei eingelesen, die die Karteninformationen, Parameter sowie die Start- und Zielpositionen der Agenten enthält. Daraufhin wird über die `jade.core.Runtime` die JADE-Umgebung gestartet. Über diese Umgebung werden die Agenten initialisiert und gestartet. Der angesprochene HAL ist als innere Klasse implementiert und erfüllt das `CoDyAgent`HAL-Interface. Die `Simulation`CoDyHAL ermöglicht dem `Controller`, die Bewegungen der CoDy-Agenten darzustellen. Außerdem werden die Bewegungen aufgezeichnet, so dass sie nach der eigentlichen Simulation wiedergegeben werden können. Nachdem die Agenten initialisiert und gestartet sind, wartet der Simulator, bis alle Agenten ihre jeweiligen Ziele erreichen und schließt dann die JADE-Umgebung.

4 Methodik der Evaluation

4.1 Validierung

In diesem Kapitel wird eine Strategie formuliert, mit der die Vereinigung des JADE-Framework und des CoDy-Algorithmus validiert werden kann.

In [3] ist eine Vielzahl an Experimenten definiert. Diese werden in einer geeigneten Simulationsumgebung wiederholt und mit den Ergebnissen aus [3] verglichen. Wenn diese Experimente wiederholt werden können, dann erfüllt auch das JADE-Framework seinen Zweck. Denn nur wenn das Scheduling und die Kommunikation der Agenten korrekt funktioniert, kann auch der CoDy-Algorithmus funktionieren. Dass die eigenen Messwerte oder Beobachtungen exakt mit denen aus [3] übereinstimmen, ist relativ unwahrscheinlich. Das liegt vor allem daran, dass nur eine relativ kleine Stichprobe von 30 Wiederholungen pro Experiment genommen wird. Aus den möglichen, hunderten von Millionen Kommunikationsreihenfolgen [3], die 30 gleichen oder ähnlichen zu erwischen, ist sehr unwahrscheinlich. Für die Messwerte bedeutet das, dass eine Annäherung der Werte ausreicht, um eine fundierte Aussage über die Qualität der Implementierung zu treffen. Von den Experimenten, die durchgeführt werden, sind aber für die Mehrheit keine Messwerte dokumentiert. Hier liegen dann allgemeine Beobachtungen vor. Für die Validierung der Implementation sollen diese genauso beobachtet werden. Falls es hier zu Abweichungen kommt, bedarf es einer tieferen Analyse beziehungsweise einer genauen Erklärung.

4.2 Simulationsumgebung

Um die Experimente ausführen zu können, wurde eine Multi-Agenten-Simulationsumgebung entwickelt. Es existieren zwar diverse Simulatoren für Multi-Agenten-Systeme, jedoch sind

Agenten hier meistens mit der OOP statt der AoSE umgesetzt. Im Beispiel von [14], sind Agenten Objekte, die über synchrone Methodenaufrufe „kommunizieren". Dieser Simulator ist zwar in der Lage, eine große Anzahl an Agenten zu simulieren, ist aber eine *Single-Thread-Applikation* und stemmt diese Aufgabe über das Skalieren der Hardware. Andere Simulatoren ermöglichen zwar den asynchronen Nachrichtenaustausch, geben aber bestimmte Nachrichtenformate und Verhaltensmuster für Agenten vor (zum Beispiel [15]). Da es in dieser Arbeit nicht nur darum geht, Experimente für einen verteilten Algorithmus zu wiederholen, sondern auch Software für ein Multi-Roboter-System zu entwickeln, sind diese Einschränkungen groß genug, um eine eigene Simulationsumgebung zu entwickeln.

Der Simulator wurde dabei rudimentär entwickelt. Der Simulator startet die JADE Umgebung und erzeugt alle Agenten. Außerdem stellt die Umgebung den Agenten die Karte und die Parameter für das jeweilige Experiment zur Verfügung und beobachtet, wann ein Experiment erfolgreich beendet wurde. Zusätzlich bietet der Simulator die Möglichkeit, die von den Agenten erzeugten Daten als Dateien zu speichern und visualisiert die Bewegungen der Agenten.

Da JADE jedoch über *Threads* skaliert [1], tut dies auch der Simulator. Für ein verteiltes System ist dieses Verhalten kein Problem, für einen einzelnen Rechner bedeutet dies aber, dass eine hohe Anzahl an Agenten nicht simuliert werden kann. Hier wäre ein potenter Rechner oder Server nötig oder die Simulationsumgebung müsste dahingehend erweitert werden, dass diese verteilt ausgeführt werden kann. Dies stellt auch die größte Schwachstelle des Simulators dar.

4.3 Metriken

Um die Implementation des CoDy-Algorithmus bewerten zu können, ist es notwendig qualitative Merkmale zu definieren, gegen die getestet werden kann. Da in [3] schon Experimente durchgeführt wurden, werden diese Merkmale zum größten Teil übernommen. Jedes Experiment wird 30 Mal wiederholt. Da für die Wegstrecke und den Zeitbedarf nur Durchschnittswerte angegeben sind, kommt für die eigenen Messungen ein 95%-Konfidenzintervall mit t-Verteilung zum Einsatz, um die Stichprobe besser einordnen zu können.

- **Erfolg:** Ein Durchlauf gilt dann als erfolgreich, wenn alle Agenten auf ihrer Zielposition parken.
- **Prioritätsüberlauf:** Wenn es in einer Wiederholung zu mindestens einem Prioritätsüberlauf kommt, wird die Wiederholung entsprechend markiert.
- **Wegstrecke:** Die Anzahl der Schritte, die ein Agent braucht um seine Zielposition zu erreichen. Auf einer Position zu verharren, erhöht diesen Wert also nicht.
  - $s_{\mathbf{opt}}$**:** Die durchschnittliche Wegstrecke für die optimale Lösung.
  - $s_{\mathbf{CoDy}}$**:** Die durchschnittliche Wegstrecke für die Messung aus [3].
  - **[**$\bar{s}_{\mathbf{u}}$**,** $\bar{s}_{\mathbf{o}}$**]:** Die untere und obere Grenze für das Konfidenzintervall der durchschnittlichen Wegstrecke.
- **Zeitbedarf**: Gibt den Zeitpunkt wieder, zu dem der Agent final seine Zielposition erreicht hat.
  - $t_{\mathbf{opt}}$**:** Der durchschnittliche Zeitbedarf für die optimale Lösung.
  - $t_{\mathbf{CoDy}}$**:** Der durchschnittliche Zeitbedarf für die Messung aus [3].
  - **[**$\bar{t}_{\mathbf{u}}$**,** $\bar{t}_{\mathbf{o}}$**]:** Die untere und obere Grenze für das Konfidenzintervall des durchschnittlichen Zeitbedarfs.
- **Prioritätsverlauf:** Zeigt die Prioritäten der einzelnen Agenten im Zeitverlauf der Experimente.

5 Experimente

In diesem Kapitel wird eine Auswahl relevanter Experimente aus [3] beschrieben. Der Aufbau der Experimente wird durch Abbildungen und assistierend durch Text wiedergegeben. Auf den Abbildungen sind statische Hindernisse grau, freie Zellen weiß, die Startpositionen der Agenten blau und die Zielpositionen grün markiert. Die Experimente teilen sich in zwei Kapitel auf. Das erste Kapitel 5.1 beschreibt jene Experimente, für die keine Messwerte vorliegen und Kapitel 5.2 diejenigen, für die Messwerte dokumentiert sind.

Für die Experimente sind die Agenten wie in [3] konfiguriert. Die Parameter sind wie folgt:

Die Enfernungskarten der Agenten umfassen immer die gesamte Karte eines Experiments. Für jeden Bewegungsschritt führt ein Agent genau einen Berechnungsschritt durch. Ein Agent ist etwas kleiner als eine Zelle. Die Konfiguration für die Prioritätsangleichung ist wie folgt:
$BasePrio = 0$, $PrioNoBlock = 1$, $PrioBlock = 10$ und $PrioFullBlock = 19$. Ein Prioritätsüberlauf, im Falle einer ungelösten Verklemmung, tritt erst ab einem Wert von $PrioMax = 400$ auf. Wenn ein solcher Überlauf eintritt, setzt sich der Prioritätswert auf eine zufällige Zahl zwischen null und zehn zurück. Jedes Experiment wird 30 Mal wiederholt. Der Durchmesser (d) der Umgebungskarte variiert. Bei Experimenten mit Karten, die 30 Felder breit sind, ist $d = 21$. Für die Experimente mit Messwerten beträgt $d = 13$. Für die anderen Experimente gilt $d = 9$. Die zeitliche Berechnungstiefe ist abhängig von der Dimension der Umgebungskarte $t_{\text{max}} = 0.75 * d$.

Sofern nicht anders gekennzeichnet, ist [3] als Quelle für den Aufbau der Experimente, die Messwerte und die zu erwartenden Beobachtungen anzunehmen.

In den folgenden Unterkapiteln wird häufiger von Gruppen die Rede sein. Im Kontext des CoDy-Algorithmus sind damit keine logisch zusammenhängenden Agenten gemeint. Eine Gruppe bezeichnet hier lediglich Agenten, die in Nähe zueinander starten.

5.1 Allgemeine Experimente

Für die folgenden Experimente liegen keine Messwerte vor. Sie eignen sich aber, dank der Tatsache, dass bestimmte Verhaltensmuster erwartet werden, zum Überprüfen, ob der CoDy-Algorithmus richtig implementiert wurde. Bei den meisten Experimenten, geht der genaue Aufbau dieser nicht exakt hervor. Einige Positionen, Abstände und Maße wurden deshalb geschätzt. Da aber nur Verhaltensmuster beobachtet werden, stellt dies kein Hindernis dar.

Für jedes Experiment wird zuerst geklärt, was mit diesem gezeigt werden soll. Darauf folgend wird der Aufbau des Experiments beschrieben und die zu erwartenden Beobachtungen vorgestellt. Abschließend werden, für alle Experimente zusammenfassend, die eigenen Beobachtungen vorgestellt.

Für alle Experimente dieses Kapitels gilt die Erwartungshaltung, dass alle 30 Durchläufe erfolgreich absolviert werden.

5.1.1 Kurze Engstelle

Mit diesem Experiment wird gezeigt, dass Agenten Ausweichbewegungen durchführen und Wartephasen einlegen, um andere Agenten passieren zu lassen.

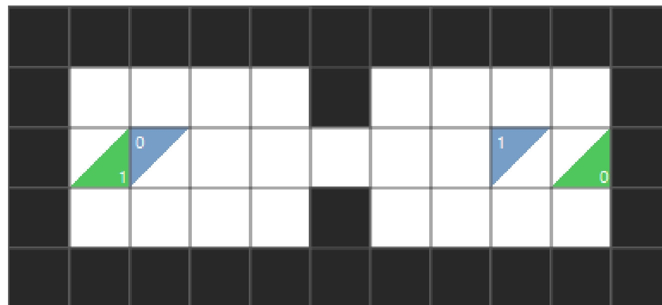

Abbildung 5.1: Aufbau für das Durchfahren zweier Agenten durch eine kurze Engstelle

Die Karte misst $9 * 3$ Felder. In der Mitte der Karte ist eine ein Feld große Engstelle. Auf beiden Seiten dieser Engstelle stehen sich Agenten gegenüber, die die Engstelle durchfahren müssen, um ihre Zielposition zu erreichen.

Erwartete Beobachtungen
Der Agent, der zuerst einen Weg plant, fährt direkt auf seine Zielposition. Der andere Agent fährt eine Ausweichposition in der unmittelbaren Nähe der Engstelle an. Nachdem der erste Agent die Engstelle passiert hat, wird sich der zweite Agent auf den direkten Weg zu seinem Ziel machen.

5.1.2 Umweg

Hier wird gezeigt wie ein Agent einen Anderen verdrängt, aber auch, dass Agenten Umwege in Kauf nehmen.

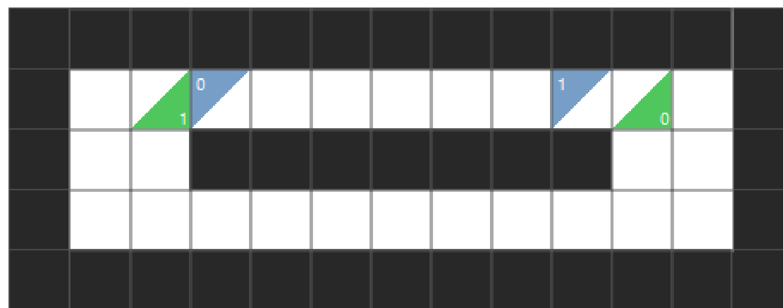

Abbildung 5.2: Aufbau für ein Szenario, in dem ein Agent einen Umweg in Kauf nimmt

Die Karte in Abbildung 5.2 misst $11 * 3$ Felder und wird mittig, horizontal durch einen Streifen aus sieben Feldern getrennt. Die Start- und Zielpositionen der Agenten sind so angeordnet, dass die nördliche Engstelle den kürzeren Weg darstellt und die südliche Engstelle den Umweg.

Erwartete Beobachtungen
Für kleine Umgebungskarten, also solche bei denen sich die Umgebungskarten der beiden Agenten erst nach dem Annähern überlappen, werden sich beide Agenten in der nördlichen Engstelle annähern. Wenn sich die Umgebungskarten dann überlappen, wird einer der Agenten den Zuschlag erhalten und den anderen Agenten verdrängen. Dieser wird dann über die südliche Engstelle, also über den Umweg, sein Ziel anfahren.

Für größere Umgebungskarten wird der Agent, der zuerst seinen Weg plant, direkt und der andere über den Umweg sein Ziel anfahren.

5.1.3 Tunnel

Dieses Experiment testet, wie anfällig die Agenten für Verklemmungen sind.

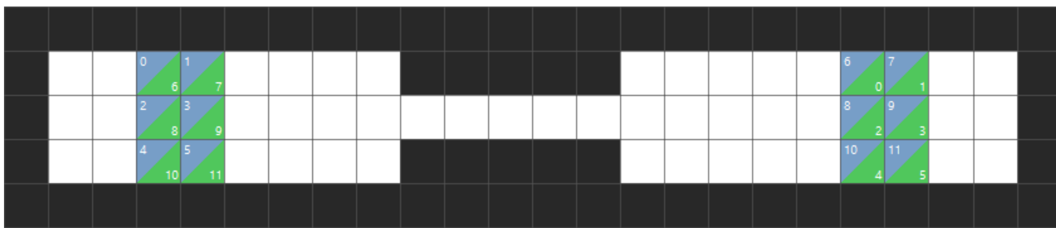

Abbildung 5.3: Ausgangssituation für das Durchqueren eines Tunnels von zwei Gruppen, bestehend aus jeweils sechs Agenten

In diesem Experiment durchqueren zwei Gruppen aus jeweils sechs Agenten eine Engstelle, die fünf Felder lang und ein Feld breit ist. Jeder Agent muss die Engstelle passieren um sein Ziel zu erreichen.

Erwartete Beobachtungen
Im ersten Schritt nähern sich beide Gruppen der Engstelle. Während eine Gruppe anfängt die Engstelle zu durchqueren, fahren die Agenten der anderen Gruppe Ausweichpositionen an und verlängern damit die Engstelle. Im Laufe des Experiments wechseln die Gruppen ihre Rollen häufiger.

5.1.4 Tunnel mit kleiner Ausweichbucht

Dieses Experiment dient zur Veranschaulichung der dynamischen Prioritäten. Diese Situation kann, dezentral, nämlich nicht durch feste Prioritäten gelöst werden.

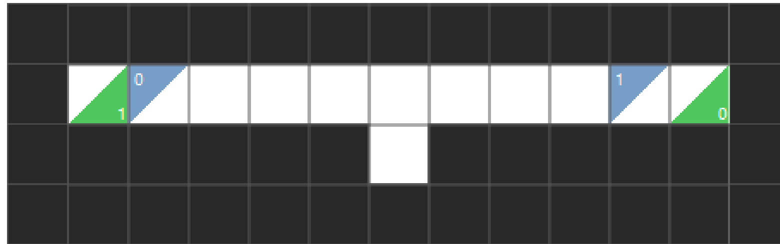

Abbildung 5.4: Aufbau für die Vorbeifahrt zweier Agenten in einem Tunnel mit einer kleinen Ausweichbucht

Zwei sich gegenüberstehende Agenten versuchen, in einer elf Felder langen und ein Feld schmalen Engstelle aneinander vorbei auf ihre Zielpositionen zu fahren. Der Tunnel hat in der Mitte ein zusätzliches freies Feld, das von den Agenten genutzt werden muss, um aneinander vorbei zu fahren.

Erwartete Beobachtungen
Zuerst werden die beiden Agenten aufeinander zu fahren. Dann wird einer der beiden Agenten zurückgedrängt werden. Wenn der zurückgedrängte Agent sich nicht weiter zurückdrängen lässt, wird dieser seine Priorität erhöhen und den zu erst drängenden Agenten in die Ausweichbucht drängen, was dann beiden Agenten ermöglicht aneinander vorbei zu ihren Zielpositionen zu fahren. Es kann auch passieren, dass ein Agent die Ausweichbucht direkt anfährt und es zu keiner Verdrängung kommt.

5.1.5 Durchfahren einer stehenden Menge

In diesem Experiment wird gezeigt, dass ein Agent, obwohl er sein Ziel schon erreicht hat, immer noch aktiv an der passiven Kooperation teilnimmt.

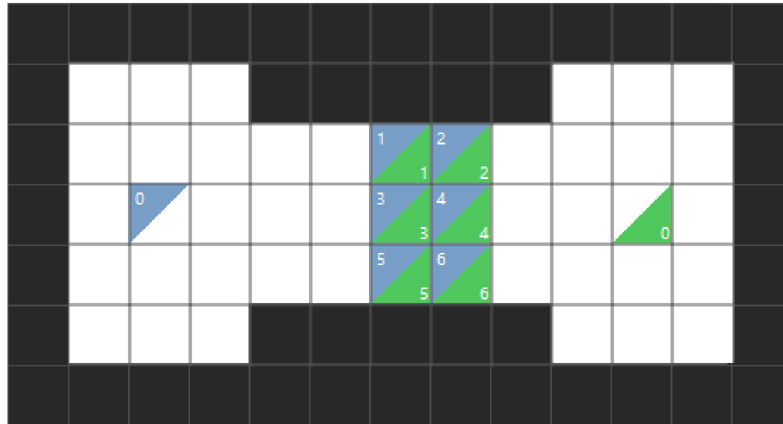

Abbildung 5.5: Aufbau für das Durchfahren eines Agenten durch ein Menge stehender Agenten

Die Abbildung 5.5 zeigt die Karte für dieses Experiment. Diese hat Dimensionen von $11 * 5$ Feldern. In der Mitte ist eine fünf Felder lange und drei Felder breite Engstelle. In dieser Engstelle stehen in zwei Reihen hintereinander sechs Agenten. Die Startpositionen dieser Agenten sind gleichzeitig ihre Zielpositionen. Ein weiterer Agent hat seine Startposition auf der linken Seite der Engstelle und seine Zielposition auf der Rechten. Er muss also durch die stehende Menge, um sein Ziel zu erreichen.

Erwartete Beobachtungen

„Agent 3“ und „Agent 4“ werden von „Agent 0“ verdrängt, fahren also von ihren Zielpositionen weg, um Platz für „Agent 0“ zu machen. Es ist auch möglich, dass dabei andere Agenten von ihren Zielpositionen verdrängt werden. Nachdem „Agent 0“ die Menge durchquert hat, fahren alle Agenten wieder zurück zu ihren Zielpositionen.

5.1.6 Kreuzung

Dieses Szenario dient der Beobachtung der Flexibilität der Agenten. Außerdem soll der Unterschied zu einem zentralen Ansatz deutlich gemacht werden und die Skalierbarkeit des Ansatzes gezeigt werden.

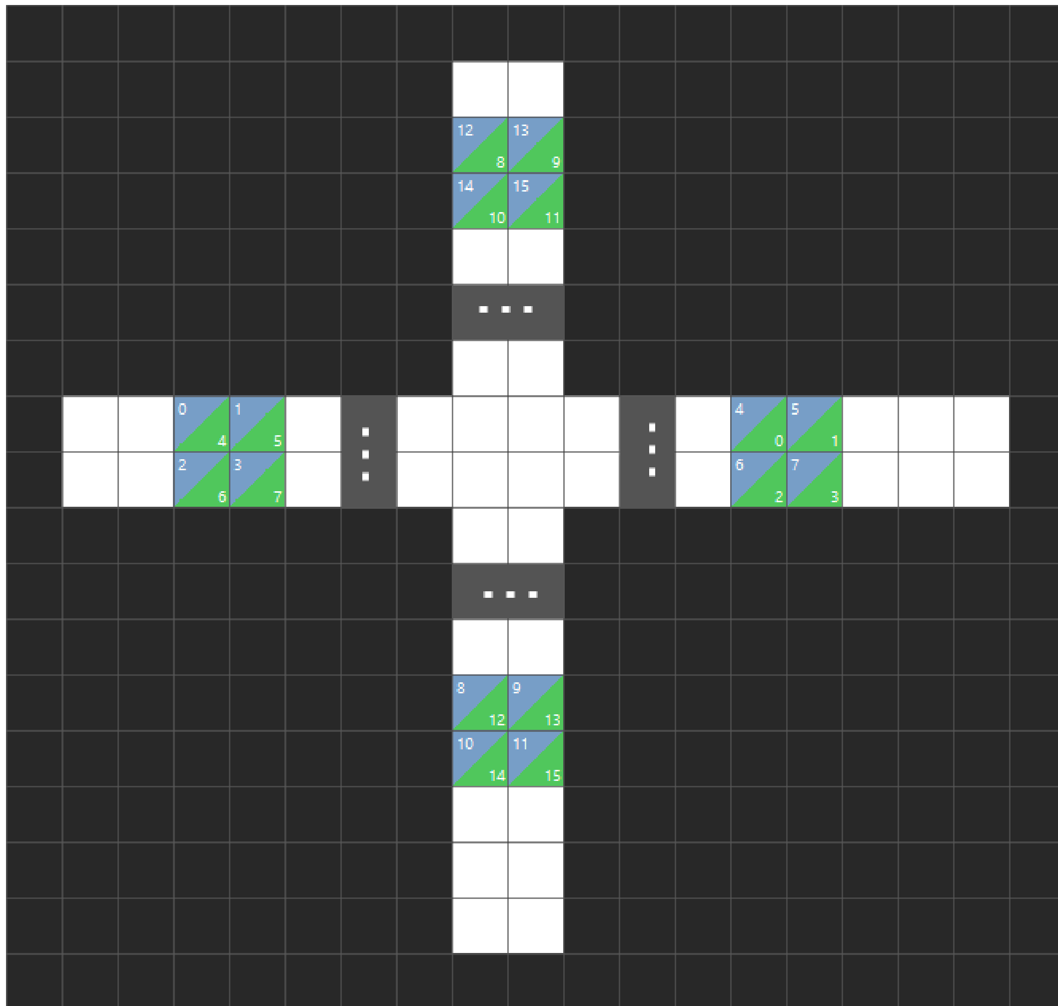

Abbildung 5.6: Aufbau für das Passieren einer Kreuzung von vier Gruppen, bestehend aus jeweils vier Agenten

In diesem Experiment versuchen vier Gruppen von jeweils vier Agenten eine Kreuzung zu durchqueren. Ziel jeder Gruppe ist es, die gegenüberliegende Seite in gleicher Formation zu erreichen. Der Kreuzungsbereich besteht aus acht freien Feldern, bietet also nicht genügend Platz für alle Agenten. Normale Verkehrsregeln, zum Beispiel das Rechtsfahrgebot, Vorfahrtsregeln oder das Bilden von Fahrspuren, sind keine Lösungen, die ein konfliktfreies

aneinander Vorbeifahren der Agenten ermöglichen. Die vier Fahrbahnen der Kreuzung sind jeweils zwei Felder breit und 14 Felder lang.

Erwartete Beobachtungen
Wegen der dynamischem Prioritäten ist eine Vielzahl an Lösungen zu beobachten. Der Berechnungsaufwand pro Agent steigt, trotz der gestiegenen Teilnehmerzahl, nicht. Ein möglicher zentraler Ansatz würde vermutlich zwei Gruppen blockieren und die übrigen gegenüberliegenden Gruppen durch die Kreuzung passieren lassen, um erst danach den blockierten Gruppen die Durchfahrt zu gewähren. Dies ist vergleichbar mit einer Ampel an einer Kreuzung im Straßenverkehr.

5.1.7 Beobachtungen

Alle zu erwartenden Beobachtungen konnten, so wie in [3] beschrieben, in den eigens ausgeführten Experimenten beobachtet werden.

5.2 Experimente mit Messwerten

Die folgenden Experimente wurden zum Überprüfen der Leistungsfähigkeit des Algorithmus entwickelt. Im Kern stehen sich immer zwei gleich große Gruppen von Agenten gegenüber, deren Ziel es ist die Startpositionen der Agenten der anderen Gruppen zu erreichen. Die Anzahl der Agenten variiert zwischen den Experimenten und der freie Raum wird tendenziell immer kleiner. Für jedes Experiment ist es das Ziel, dass in allen 30 Wiederholungen eine Lösung gefunden wird, also alle Agenten ihr Ziel erreichen.

Der Aufbau der Experimente und das zu erwartende Verhalten der Agenten werden vorgestellt. Im Kapitel 5.2.5 folgen dann die Messwerte aus [3] und die Eigenen.

5.2.1 Sechs gegen sechs: Lockere Vorbeifahrt

Das erste Experiment dieser Reihe bietet vergleichsweise viel Platz. Es ist vor allem für den späteren Vergleich interessant.

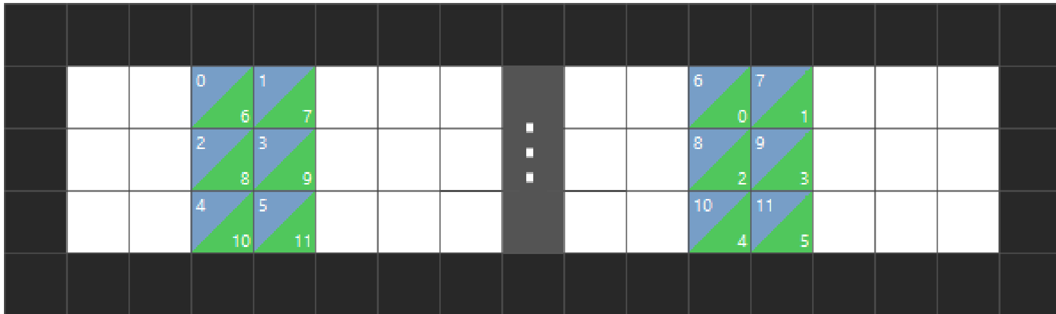

Abbildung 5.7: Aufbau für die lockere Vorbeifahrt zweier Gruppen, bestehend aus jeweils sechs Agenten

Die Abbildung 5.7 zeigt eine Karte die $3 * 30$ Felder misst. Es stehen sich zwei Gruppen, bestehend aus jeweils sechs Agenten, gegenüber. Zum rechten beziehungsweise linken Rand sind für die Gruppen noch ein paar freie Felder vorhanden. Damit soll es den Agenten möglich sein, Ausweichbewegungen nach hinten hin ausführen zu können.

5.2.2 Sechs gegen sechs: Enge Vorbeifahrt

In diesem Experiment ist der Platz für Bewegungen sehr beschränkt. Zwölf Felder sind von Agenten belegt und lediglich neun sind frei.

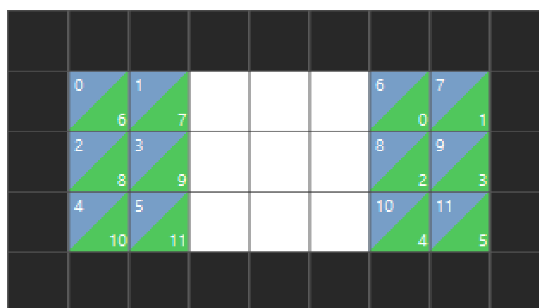

Abbildung 5.8: Aufbau für die enge Vorbeifahrt zweier Gruppen, bestehend aus jeweils sechs Agenten

Die Karte für dieses Experiment ist $7 * 3$ Felder groß. Es stehen sich zwei Gruppen aus jeweils sechs Agenten gegenüber. Wie in Abbildung 5.8 zu erkennen, befindet sich zwischen den beiden Gruppen ein Block aus $3 * 3$ freien Feldern.

5.2.3 Drei gegen drei: Enge Vorbeifahrt

Dieses Experiment dient als Vergleich zum Vorangegangenen. Die Auswirkung, die die Anzahl der Roboter hat, soll hiermit untersucht werden.

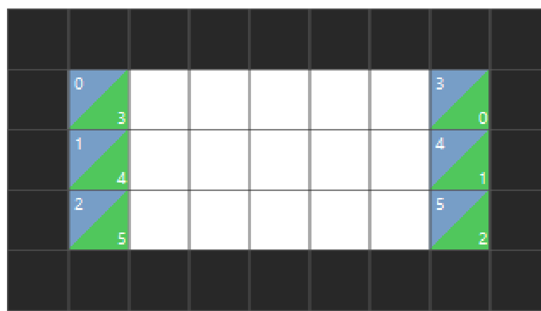

Abbildung 5.9: Aufbau für die enge Vorbeifahrt zweier Gruppen, bestehend aus jeweils drei Agenten

Die Karte für dieses Experiment ist $7 * 3$ Felder groß. Es stehen sich zwei Gruppen aus jeweils drei Agenten gegenüber. Wie in Abbildung 5.9 zu erkennen, befindet sich zwischen den beiden Gruppen ein Block aus $5 * 3$ freien Feldern.

5.2.4 Vier gegen vier: Enge Vorbeifahrt

Dieses Experiment schränkt den Platz der Agenten weiter ein. Die Zahl der belegten Felder ist hier größer als die Anzahl der freien Felder. Besonders für dieses Experiment ist, dass ein Graph vorliegt, der die Prioritätswerte der Agenten über die Zeit zeigt (siehe Abbildung 5.1 und 5.2).

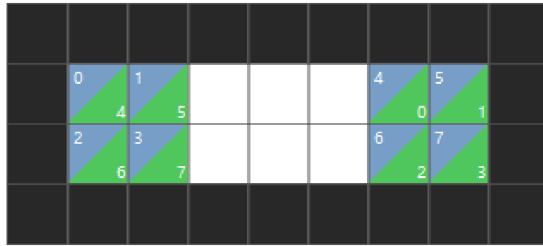

Abbildung 5.10: Aufbau für die enge Vorbeifahrt zweier Gruppen, bestehend aus jeweils vier Agenten

Die Karte für dieses Experiment misst $7 * 2$ Felder. Acht Agenten teilen sich in zwei gleich große Gruppen und stehen sich gegenüber.

5.2.5 Messergebnisse

Experiment	**Gelöst**	**Prioritäts-Überlauf**	$\mathbf{s_{opt}}$	$\mathbf{s_{CoDy}}$	$\mathbf{t_{opt}}$	$\mathbf{t_{CoDy}}$
6-vs-6-locker	30 von 30	0 von 30	19.5	20.97	22	24.33
6-vs-6-eng	26 von 30	1 von 30	5.45	11.5	10.35	21.89
3-vs-3-eng	30 von 30	3 von 30	5.5	6.35	6.33	7.68
4-vs-4-eng	27 von 30	4 von 30	5	10.85	10.5	24.96

Tabelle 5.1: Messwerte der Experimente aus [3]

Experiment	**Gelöst**	**Prioritäts-Überlauf**	$[\bar{s}_{\mathbf{u}}, \bar{s}_{\mathbf{o}}]$	$[\bar{t}_{\mathbf{u}}, \bar{t}_{\mathbf{o}}]$
6-vs-6-locker	30 von 30	0 von 30	[20.01, 22.33]	[23.92, 26.4]
6-vs-6-eng	18 von 30	18 von 18	[8.78, 9.81]	[19.52, 23.09]
3-vs-3-eng	27 von 30	0 von 27	[7.3, 7.58]	[9.53, 9.97]
4-vs-4-eng	20 von 30	20 von 20	[13.72, 25.2]	[23.04, 41.04]

Tabelle 5.2: Messwerte der selbst durchgeführten Experimente

Die eigenen Messwerte weichen teils stark von den vorgegeben Messwerten ab. Lediglich das Experiment „5.2.1 Sechs gegen sechs: Lockere Vorbeifahrt“ trifft die Erwartungen.

Für die eigenen Messungen ist anzumerken, dass für den Prioritätsüberlauf nicht immer „von 30“ angegeben ist. Das hat damit zu tun, dass die Simulationsumgebung nur Daten erzeugt, wenn ein Experiment erfolgreich war.

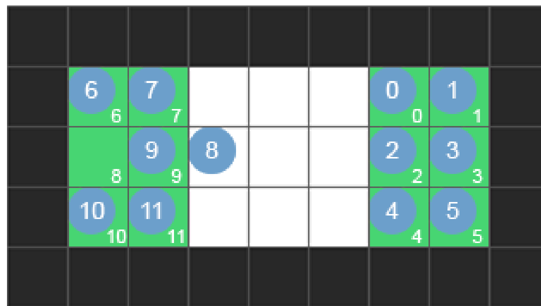

Abbildung 5.11: Beispielhafte Totalblockade für das Experiment „5.2.2 Sechs gegen sechs: Enge Vorbeifahrt„

Für das Experiment „5.2.2 Sechs gegen sechs: Enge Vorbeifahrt“ ist die zurückgelegte Strecke etwas besser als erwartet. Der Zeitbedarf trifft die Erwartung. Auffällig ist jedoch, dass nur 18 statt 26 Wiederholungen erfolgreich sind und es in allen, statt nur bei einer Wiederholung, zu Prioritätsüberläufen kam. Bei den zwölf ungelösten Wiederholungen ist eine Variation der in Abbildung 5.11 gezeigten Situation eingetreten.

Die Werte für „5.2.3 Drei gegen drei: Enge Vorbeifahrt“ weichen nur leicht von den erwarteten Werten ab und statt 30 werden nur 27 Wiederholungen gelöst. Dafür kommt es in den gelösten Wiederholungen aber nicht zu Prioritätsüberläufen. Die drei erwarteten Wiederholungen, die einen Prioritätsüberläufen haben, entstehen aus Situationen, in denen sich die Agenten in breiter Front aufeinander zu bewegen. In den eigenen Experimenten führte genau diese Situation zu den drei ungelösten Wiederholungen.

Die gemessenen Werte weichen für das Experiment „5.2.4 Vier gegen vier: Enge Vorbeifahrt“ am stärksten ab. Es werden nur in 20 statt 27 Wiederholungen Lösungen gefunden und in jeder Wiederholung kommt es zu Prioritätsüberläufen. Die Konfidenzintervalle für die durchschnittlich zurückgelegte Strecke und den durchschnittlichen Zeitbedarf sind im Vergleich besonders groß. Die erwartete Strecke ist nicht mal im Intervall enthalten.
Für dieses Experiment gibt es zusätzliche Daten. Abbildungen 5.13 und 5.12 zeigen den

Prioritätsverlauf über die Zeit. Die $x - Achsen$ bilden die Zeit ab und die $y - Achsen$ die Priorität. Es zeigt sich deutlich, dass die erwarteten Prioritätsverläufe verfehlt werden. Statt, dass die Priorität der Agenten über die Zeit stetig ansteigt und teilweise wieder zurückgeht, pendeln bei den eigenen Experimenten die Prioritätswerte zwischen minimalen und maximalen Werten hin und her.

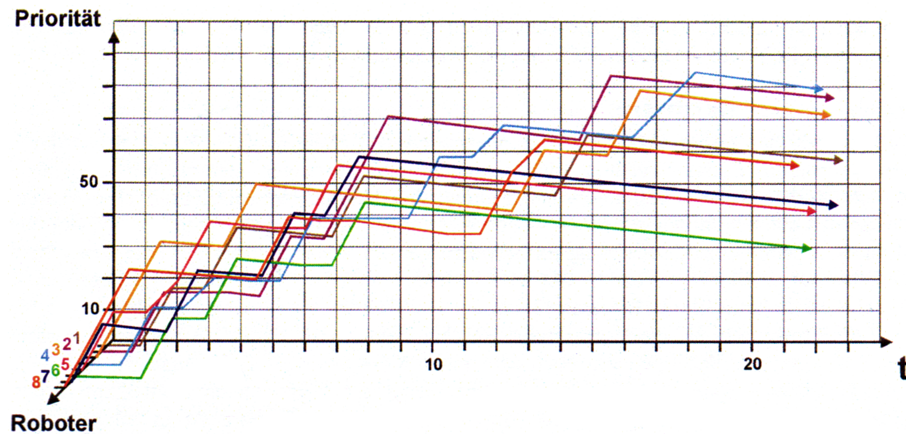

Abbildung 5.12: Prioritätsverlauf aller Agenten für das Experiment „5.2.4 Vier gegen vier: Enge Vorbeifahrt“ aus [3]

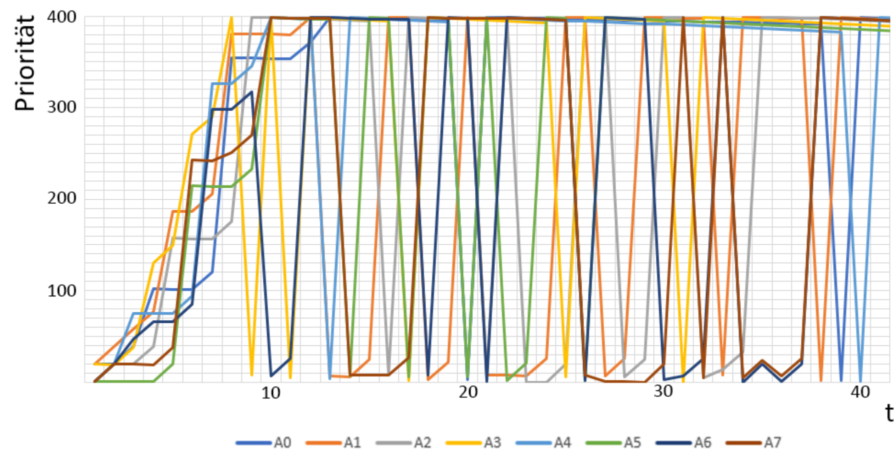

Abbildung 5.13: Prioritätsverlauf aller Agenten für das Experiment „5.2.4 Vier gegen vier: Enge Vorbeifahrt“ der selbst durchgeführten Experimente

6 Diskussion

Für die Experimente ohne Messwerte wurden bestimmte Verhaltensmuster der CoDy-Agenten beobachtet. In den eigenen Experimenten konnte gezeigt werden, dass Agenten warten, Umwege in Kauf nehmen, sich verdrängen lassen und auch ihre Zielposition verlassen. Es wurden also die Mechanismen der passiven Kooperation des CoDy-Algorithmus beobachtet. Die Beobachtungen entsprechen dabei den Erwarteten.

Interessanter sind jedoch jene Experimente und Werte die von den Erwartungen abweichen.

Im Experiment „5.2.3 Drei gegen drei: Enge Vorbeifahrt“ kommt es in drei , die nach einem Prioritätsüberlauf gelöst werden, in der eigenen Implementierung zu einer Verklemmung die sich nicht auflöst. Bei den Experimenten „5.2.2 Sechs gegen sechs: Enge Vorbeifahrt“ und „5.2.4 Vier gegen vier: Enge Vorbeifahrt“ sind deutlich weniger Wiederholungen erfolgreich. Bei allen gelösten Wiederholungen kommt es zu Prioritätsüberläufen und die Konfidenzintervalle weichen, teilweise stark, von den Erwartungen ab.

Da die Experimente ohne Messwerte und das Experiment „5.2.1 Sechs gegen sechs: Lockere Vorbeifahrt“ alle Erwartungen erfüllen, lässt sich daraus schließen, dass ein Mechanismus, der vor allem in engen Situationen greift, von [3] abweicht.
Abbildung 5.13 ist ein starkes Indiz dafür, dass die Prioritätsanpassungen in solchen Situationen unterschiedlich funktionieren. Abbildung 5.13 zeigt, dass die Prioritäten auf das Maximum schnellen. Das passiert nur, wenn ein Notweg erzwungen wird. Es ergeben sich also zwei Erklärungen. Entweder wird zu häufig ein Notweg erzwungen oder die Priorität für den Notweg ist abweichend konfiguriert. In Abbildung 5.12 ist kein Agent zu erkennen dessen Priorität auf das Maximum schnellt; es ist aber auch nicht bekannt, ob es in der Wiederholung, die die Abbildung zeigt, zu einer Situation kam, in der das Erzwingen eines Notweges notwendig war. Eine eindeutige Aussage über den Notweg lässt sich daher nicht treffen.

Die Abbildungen 5.12 und 5.13 gewähren jedoch noch weitere Einblicke. Wenn man die ersten fünf Sekunden betrachtet, fällt auf, dass in [3] Prioritäten bis maximal 50 erreicht werden. In den eigenen Messungen liegen die Prioritätswerte nach fünf Sekunden bereits bei 200 und mehr. Die Prioritäten entwickeln sich im Vergleich also schneller. Daraus folgt, dass es häufiger zu Konflikten kommt, für deren Lösung die Prioritäten steigen. Wenn ein Agent seinen Weg zum lokalen Ziel berechnet, werden freie Zellen, denen von Agenten mit niedrigerer Priorität belegten, bevorzugt. Das lokale Ziel wird jedoch zufällig gewählt. Das Konfliktpotenzial und der damit verknüpfte rapide Anstieg der Prioritäten, lässt sich durch eine klügere Wahl des lokalen Ziels verringern. Es lässt sich nur vermuten, ob eine solche Optimierung in [3] stattgefunden hat.

Zu den Experimenten lässt sich abschließend sagen, dass sich die eigene Implementierung zum großen Teil mit der aus [3] deckt. In den „Extremsituationen“ weicht das Verhalten der eigenen Implementierung jedoch ab. Als mögliche Quelle dieser Abweichung wurden die Wahl des lokalen Ziels, die Häufigkeit des Erzwingen eines Notweges und der Prioritätssprung beim Erzeugen eines Solchen identifiziert. Die Abweichung ergeben sich aus der fehlenden Referenzimplementation und den Interpretationsspielräumen in [3]. Es bedarf einer tieferen Analyse um die Quellen der Abweichungen eindeutig zu identifizieren.

Folgend wird über die entstandene Anwendung mit den, zu Beginn dieser Arbeit definierten, Anforderungen bewertet.

Skalierbarkeit JADE ist für verteilte Systeme entwickelt und bietet viele Möglichkeiten, um Systeme zu skalieren. Die Berechnung eines Weges pro Agent skaliert, da die Komplexität mit der Größe der Umgebungskarte und nicht mit der Anzahl der Agenten steigt. Jedoch steigt die Anzahl der Nachrichten, die ein Agent verschickt, linear mit der Anzahl der Agenten. Für eine überschaubare Anzahl an Agenten, stellt dies kein Problem dar. Mit steigender Anzahl an Agenten wird der Nachrichtenverkehr aber immer ineffizienter, weil ein Agent alle Nachrichten verwirft, deren Umgebungskarte des Senders nicht mit der Eigenen überlappt. Dies kann aber sehr leicht auf Seite des Senders optimiert werden, indem diese Nachrichten gar nicht erst versendet werden. Dies könnte die AP bei einer sehr großen Anzahl an Agenten deutlich entlasten. Die Skalierbarkeit der Anwendung kann optimiert werden. Im Rahmen dieser Arbeit ist sie trotzdem ausreichend.

Terminiertheit Diese Anforderung ist nicht erfüllt. Es kann passieren, dass die Agenten in eine Verklemmung geraten, die nicht gelöst wird. Bei der aktuellen Implementierung läuft die Anwendung in diesen Situationen unendlich lang weiter. Es müsste eine Überprüfung solcher Situationen implementiert werden, um die Anwendung ohne befriedigende Lösung zu stoppen, damit dieser Anforderung gerecht wird.

AoSE Diese Anforderung wurde eingehalten. Ein Agent repräsentiert einen *Smart Chair*. Es ist jedoch anzumerken, dass die Kommunikation zwischen den Agenten nicht sonderlich komplex ist. Dies ist aber eine wichtige Eigenschaft die ein Multi-Agenten-System definiert. Die Vielfalt und Komplexität der Nachrichten steigt jedoch, wenn weitere Funktionen, wie zum Beispiel nicht kommunizierende dynamische Hindernisse, hinzugefügt werden. Momentan besteht auch noch keine Möglichkeit, die tatsächlichen *Smart Chairs* mit dieser Anwendung zu steuern. Dies war aber keine formulierte Anforderung.

Kollisionsvermeidung Die Agenten sind in der Lage statischen und dynamischen Hindernissen auszuweichen. In der Ausführung der Anwendung kommt es nicht zu Kollisionen. Die Agenten sind jedoch blind. Die statischen Hindernisse müssen zur Initialisierung bekannt sein und die dynamischen Hindernisse beschränken sich auf die anderen Agenten. Diese Anforderung ist also nur zur Hälfte erfüllt. Hindernisse werden umfahren, aber nicht erkannt.

7 Fazit

Ziel dieser Arbeit war es, eine Anwendung für einen bestimmten Anwendungsfall zu erarbeiten. Die Anforderungen an die Anwendung konnten zum Großteil erfüllt werden. Vor allem die wichtigen Anforderungen wurden erfüllt. So skaliert die Anwendung größtenteils dadurch, dass die Anwendung als Multi-Agenten-System umgesetzt wurde. In den Experimenten wurde gezeigt, dass der CoDy-Algorithmus nicht fehlerfrei umgesetzt werden konnte. Als Produkt ist trotzdem eine skalierende Multi-Agenten-Anwendung entstanden, die mit minimalen Anpassungen, für den Einsatz mit den *Smart Chairs* bereit ist.

7.1 Ausblick

Die Anwendung kann iterativ für die *Smart Chairs* erweitert werden. Mit einem entsprechendem HAL, der präzise Bewegungen erlaubt, kann die Anwendung fast unverändert aufgespielt werden. Wenn sich die *Smart Chairs* präzise steuern lassen und man, außer den *Smart Chairs*, keine anderen dynamischen Hindernisse erlaubt, bestehen mit der Simulation vergleichbare Verhältnisse. Der CoDy-Algorithmus bietet noch einige Mechanismen, die in dieser Arbeit kaum angesprochen wurden. So könnte ein Agent, im Falle einer Totalblockade, zentral die Wege aller beteiligten Agenten planen und als Vorschläge an die Agenten verteilen [3]. Hierbei treten die Agenten in potenziell komplexe Verhandlungen. Das is etwas für das sich die AoSE, vor allem durch die ACL, auszeichnet. Andere Mechanismen sind zum Beispiel das Erforschen von unbekannten Karten, der Umgang mit nicht kommunizierenden dynamischen Hindernissen und weiteren Mechanismen, die die Robustheit steigern, wie beispielsweise der Ausfall der Sensorik, des Antriebs oder der Kommunikation [3]. Bevor man die Anwendung aber erweitert und auf das eigentliche Roboter-System überträgt, sollte die aktuelle Prioritätsanpassung verbessert werden.

Auch die Erforschung der AoSE im Kontext von IoT könnte vertieft werden.

Literatur

[1] F. Bellifemine, G. Caire und D. Greenwood, *Developing multi-agent systems with JADE*. Wiley, 2008, ISBN: 978-0-470-05747-6.

[2] L. Padgham und M. Winikoff, *Developing Intelligent Agent Systems - A Practical Guide*. New York: John Wiley & Sons, 2005, ISBN: 978-0-470-86121-9.

[3] R. Regele, *Kooperative Multi-Roboter-Wegplanung durch heuristische Prioritätsanpassung*. Berlin: Logos Verlag Berlin GmbH, 2008, ISBN: 978-3-832-52028-1.

[4] M. Wooldridge, *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2002, ISBN: 047149691X.

[5] L. Padgham und J. Thangarajah, "Agent oriented software engineering: Why and how", *VNU Journal of Science, Natural Sciences and Technology*, Jg. 27, 2011.

[6] H. Mubarak, "Developing flexible software using agent-oriented software engineering", *IEEE Software*, Jg. 25, Nr. 5, S. 12–15, Sep. 2008, ISSN: 0740-7459. DOI: 10.1109/MS.2008.135.

[7] J. Odell, "Objects and agents compared.", *Journal of Object Technology*, Jg. 1, S. 41–53, Mai 2002.

[8] M. Wooldridge und N. Jennings, "Intelligent agents: Theory and practice", *The Knowledge Engineering Review*, Jg. 10(2), S. 115–152, 1995.

[9] S. Sorrell, *Iot the internet of transformation 2018*, White Paper, Juniper Research Ltd, Mai 2018.

[10] *Publicly available agent platform implementations*, http://www.fipa.org/resources/livesystems.html, zuletzt besucht am 25.03.2019.

[11] *Financing the future of 5g*, https://www.greensill.com/whitepapers/financing-the-future-of-5g/, zuletzt besucht am 25.03.2019, Feb. 2019.

[12] *Jack - autonomous software*, http://www.agent-software.com/products/jack/, zuletzt besucht am 15.04.2019.

[13] *Erfahren sie mehr über die java-technologie*, https://www.java.com/de/about/, zuletzt besucht am 25.03.2019.

[14] J. Weyl, *Multi-agent modelling with mars - a handbook*, https://mars-group.org/modeling-handbook/, version 1.5.1, 2019.

[15] *Mass - multi agent system simulator*, http://mas.cs.umass.edu/research_old/mass/, zuletzt besucht am 23.03.2019.

[16] T. Reenskaug, "Models - views - controllers", https://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf, zuletzt besucht am 26.03.2019, Dez. 1979.

[17] H. Buhr u. a., *Smart chair - sensors*, https://github.com/Transport-Protocol/SmartChair/wiki/Sensors, zuletzt besucht am 08.06.2018, 2016.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 16. April 2019 Dennis Schröder