Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Arthur Greb

Konzeptionierung und Programmierung der Kommunikationsstrukturen und -abläufe für eine dezentrale Produktionssteuerung bei der Verhandlung fahrerloser Transportfahrzeuge über Fahraufträge

Fakultät Technik und Informatik
Department Maschinenbau und Produktion

Faculty of Engineering and Computer Science
Department of Mechanical Engineering and Production Management

Arthur Greb

Konzeptionierung und Programmierung der Kommunikationsstrukturen und -abläufe für eine dezentrale Produktionssteuerung bei der Verhandlung fahrerloser Transportfahrzeuge über Fahraufträge

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Produktionstechnik und -management
am Department Maschinenbau und Produktion
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Erstprüfer: Prof. Dr. Henner Gärtner
Zweitprüfer: Dipl. -Ing. Frank Peters

Abgabedatum: 12.05.2021

Zusammenfassung

Arthur Greb

Thema der Bachelorthesis

Konzeptionierung und Programmierung der Kommunikationsstrukturen und -abläufe für eine dezentrale Produktionssteuerung bei der Verhandlung fahrerloser Transportfahrzeuge über Fahraufträge

Stichworte

Fahrerlose Transportsysteme, Programmieren, dezentrale Produktionssteuerung, Machine-to-Machine-Kommunikation

Kurzzusammenfassung

Als Ausgangspunkt der vorliegenden Arbeit ist ein zuvor an der HAW Hamburg entwickeltes Planspiel, welches eine dezentrale Produktion simuliert. Das Spiel ist in drei Maschinengruppen aufgeteilt und wird über selbststeuernde Kundenaufträge ausgelöst, welche sich am Auftragseingang befinden. Die Aufträge werden von fahrerlosen Transportfahrzeugen (FTFs) zwischen den Maschinengruppen durch die ganze Produktion transportiert. Dafür sendet jeder Kundenauftrag der Reihe nach über ein Netzwerk eine Transportanfrage an alle verfügbaren FTFs. Diese interagieren/ konkurrieren miteinander mit dem Ziel, den Kundenauftrag zu erhalten. Entscheidend hierfür ist die Verfügbarkeit der FTFs und deren Entfernung von der Abholstation. Die Kundenaufträge enthalten Daten, die Informationen beinhalten, welche die FTFs als eindeutigen Weg durch die Produktion erkennen. Als eine Kernproblematik konnte herausgearbeitet werden, welche Fahrzeuge wann miteinander kommunizieren bzw. wie die Kommunikationsdisziplin gestaltet werden muss, damit die gesendeten Daten nicht verlorengehen. Aus den beschriebenen Aufgaben ergibt sich die im Rahmen der vorliegenden Arbeit zu untersuchende Forschungsfrage:

Wie muss eine dezentrale Kommunikationsstruktur gestaltet werden, damit die Kundenaufträge so schnell wie möglich von einem verfügbaren FTF übernommen werden?

Arthur Greb

Title of the paper

Conception and programming of communication structures and processes for a decentralized production system, using the example of negotiation of driverless transport vehicles via the acceptance of driving orders.

Keywords

Automated guided vehicles (AGV), Programming, Decentralized Manufacturing control, Machine-to-Machine communication

Abstract

The fundamental part of the present paper deals with a game that simulates decentralized production, this game was previously developed at the University of Applied Sciences (HAW). The game is divided among three machine groups and is triggered by the input of customer orders. The orders are transported by AGVs between the machine groups across the entire manufacturing area. To achieve this task, each order sends a request to all available AGVs via a network. Thus, the AGVs communicate with each other to determine how this task shall be undertaken. The decisive factor here is the availability of the AGVs and the distance from the pick-up location. The customer order contains metadata which the AGVs recognize as a unique route through the production area. One of the main problems to be addressed is which vehicles interact with each other and when, and how the communication network should be designed so that the data being transmitted are not lost. The research question which will be deliberated in this work arises from the above tasks:

How should a decentralized communication structure be designed so that the customers orders can be completed as swiftly and efficiently as possible?

Inhaltsverzeichnis

Abbildungsverzeichnis II
Tabellenverzeichnis IV
Abkürzungsverzeichnis V
1 Einleitung 1
1.1 Einführung 1
1.2 Zielsetzung 2
1.3 Herangehensweise an die Arbeit 3
1.4 Abgrenzung 4
2 Wissenschaftliche und technische Grundlagen 5
2.1 Zentrale/ dezentrale Produktionssteuerung 5
2.2 Fahrerlose Transportsysteme und -fahrzeuge 6
2.3 Internet der Dinge und die industrielle Kommunikation 6
2.4 Methoden der Wegfindung 8
2.5 Vorgehensweise in der Softwareentwicklung 8
3 Anforderungsanalyse 10
3.1 Voraussetzungen für die Arbeit 10
3.2 Beschreibung des Planspiels 11
3.3 Anforderungserhebung 12
3.3.1 Gegebene Anforderungen aus dem Planspiel 13
3.3.2 Erweiterte Anforderungen für die Simulation 15
4 Konzept und Datenmodell der Verhandlungen 18
4.1 Ziel der Simulation 18
4.2 Kommunikation zwischen den Komponenten 19
4.2.1 Analyse der Verhandlungen im Planspiel 19
4.2.2 Welche Daten werden ausgetauscht? 23
4.2.3 Datenaustausch mit Hilfe von MQTT 25
4.3 Funktionen und deren Struktur 27
4.4 Darstellung der vollständigen Kommunikationsarchitektur 30
5 Implementierung der Simulation 37
5.1 Grafische Oberfläche und Tools zur Realisierung der Simulation 37
5.1.1 VMware Workstation Player 37
5.1.2 Texteditor Sublime Text 38
5.1.3 Terminalemulator Terminator 39
5.1.4 Zusammenschluss der Tools 40

5.2 Funktionen der Maschinen in der Simulation 41
5.3 Aufbau des Kundenauftrages 42
5.4 Verbindungsaufbau zu MQTT 44
5.5 Wegfindung mit dem A* Algorithmus 46
6 Prototypische Anwendung der Simulation 50
6.1 Vorbereitung zur Ausführung der Simulation 50
6.2 Beginn der Verhandlungen 52
6.3 Transportvorgang 54
6.4 Bearbeitung an den Maschinen 55
6.5 Entscheidung durch den Zufallsgenerator 56
7 Zusammenfassung und Ausblick 59
7.1 Zusammenfassung 59
7.2 Ausblick 63
Quellenangaben 64
Anhang A. Spielregeln Planspiel 67
1. Simulation einer dezentralen Produktionssteuerung 67
Anhang B. Installationsanleitung Software 74
1. Installation der Software VMware Workstation 16,0 Player 74
2. Installation von Ubuntu 79
3. Installation des Programmes Terminator 90
4. Installation von Sublime Text 91
5. Installation von Python3 93
6. Installation von Bibliotheken für die Simulation. 93
7 Vorbereitung für die Simulation 93
Anhang C. Quellcode Simulation 95
Selbstständigkeitserklärung 123

Abbildungsverzeichnis

Abbildung 1: Darstellung des Publizierens und der Subskribierung (Quelle: Raschbichler 2017) 7
Abbildung 2. Swimming-Pool-Modell (Gärtner 2018, S.84) 10
Abbildung 3: Planspiel (Quelle: eigene Darstellung) 11
Abbildung 4: Szenario aus dem Planspiel für die Analyse der Verhandlungen (Quelle: eigene Darstellung) 20

Abbildung 5: Szenario aus dem Planspiel für die Analyse der Verhandlungen (Quelle: eigene Darstellung) 20
Abbildung 6: PAP der Verhandlungen aus den Daten des Szenarios im Planspiel (Quelle: eigene Darstellung) 22
Abbildung 7: Programablaufplan der Funktion „FTF-init“ (Quelle: eigene Darstellung) 27
Abbildung 8: Programmablaufplan der Funktion „FTF_init_Antwort“ (Quelle: eigene Darstellung) 29
Abbildung 9: Vollständiges Konzept der Kommunikationsstruktur in der Simulation (Teil 1) (Quelle: eigene Darstellung) 33
Abbildung 10: Vollständiges Konzept der Kommunikationsstruktur in der Simulation (Teil 2) (Quelle: eigene Darstellung) 34
Abbildung 11: Vollständiges Konzept der Kommunikationsstruktur in der Simulation (Teil 3) (Quelle: eigene Darstellung) 35
Abbildung 12: Screenshot: VMware Workstation 16 Player mit der Linux Distribution Ubuntu (Quelle: aus dem Programm VMware Workstation 16 Player) 38
Abbildung 13: Screenshot: Sublime Text mit drei parallel geöffneten Programmen in einem Fenster (Quelle: eigene Darstellung) 39
Abbildung 14: Terminal im Terminator teilen (Both 2020, S. 412) 39
Abbildung 15: Mehrere Terminals im Terminator geöffnet (Both 2020, S. 414) 40
Abbildung 16: Mehrere Terminals im Terminator geöffnet (Both 2020, S. 414) 40
Abbildung 17: Screenshot: Ein Thread wird erstellt (Quelle: eigene Darstellung) 41
Abbildung 18: Screenshot: Thread wird gestartet (Quelle: eigene Darstellung) 42
Abbildung 19: Screenshot: Bearbeitungszeit kann für jede Maschine eingestellt werden (Quelle: eigene Darstellung) 42
Abbildung 20: Screenshot: Unterschiedliche Varianten eines Kundenauftrages (Quelle: eigene Darstellung) 43
Abbildung 21: Screenshot: Aktualisierung der „order“-Informationen (Quelle: eigene Darstellung) 43
Abbildung 22: Screenshot: Client-Deklaration (Quelle: eigene Darstellung) 45
Abbildung 23: Screenshot: Verbindungsaufbau zum Broker Mosquitto (Quelle: eigene Darstellung) 45
Abbildung 24: Screenshot: Kundenauftrag publiziert eine Nachricht über MQTT (Quelle: eigene Darstellung) 45
Abbildung 25: Screenshot: Methodenaufruf zum Subskribieren eines Topics (Quelle: eigene Darstellung) 46
Abbildung 26: Screenshot: Aufruf der Methode „message_callback_add“ (Quelle: eigene Darstellung) 46
Abbildung 27: Screenshot: Zufallsgenerator für die Startposition der FTFs (Quelle: eigene Darstellung) 47
Abbildung 28: Screenshot: Angepasste Produktionslinie für den A*-Algorithmus (Quelle: eigene Darstellung) 47
Abbildung 29: Screenshot: Maschinen und deren Anfahrposition (Quelle: eigene Darstellung) 47
Abbildung 30: Screenshot: Berechnung der Route durch den A*-Algorithmus (Quelle: eigene Darstellung) 48
Abbildung 31: Screenshot: Funktion „print_path“ (Quelle: eigene Darstellung) 48
Abbildung 32: Screenshot: Produktionslinie im Terminator inklusive Route (Quelle: eigene Darstellung) 49

Abbildung 33: Screenshot: Fortschrittsanzeige für die Darstellung des Fahrens eines FTFs (Quelle: eigene Darstellung) 49
Abbildung 34: Screenshot: Initialisierungsprozess der FTFs und Aktivierung der Maschinengruppen (Quelle: eigene Darstellung) 51
Abbildung 35: Screenshot: Das „Help?"-Menü und dessen Inhalt (Quelle: eigene Darstellung) 51
Abbildung 36: Screenshot: Kundenauftrag „a69" wurde erfolgreich an alle FTFs übermittelt (Quelle: eigene Darstellung) 52
Abbildung 37: Screenshot: Verhandlungen abgeschlossen. FTF C1 bekommt den Kundenauftrag „a69" (Quelle: eigene Darstellung) 53
Abbildung 38: Screenshot: FTF C1 transportiert den Kundenauftrag „a69" (Quelle: eigene Darstellung) 54
Abbildung 39: Screenshot: Maschine S1. Start und Ende der Bearbeitung des Kundenauftrages „a69" (Quelle: eigene Darstellung) 55
Abbildung 40: Screenshot: Aktualisierter Kundenauftrag nach der Bearbeitung in Maschine S1 (Quelle: eigene Darstellung) 56
Abbildung 41: Screenshot: Kundenauftrag „a69" wurde produziert (Quelle: eigene Darstellung) 56
Abbildung 42: Screenshot: Liste der produzierten Kundenaufträge (Quelle: eigene Darstellung) 56
Abbildung 43: Screenshot: Verhandlungen zwischen FTFs, um den Kundenauftrag mit der „order_ID" „ade" (Quelle: eigene Darstellung) 57
Abbildung 44: Screenshot: Zweite Verhandlungsrunde mit Hilfe des Zufallsgenerators um den Kundenauftrag mit der „order_ID" „ade" (Quelle: eigene Darstellung) 58
Abbildung 45: Anforderungsabgleich (Quelle: eigene Darstellung) 59

Tabellenverzeichnis

Tabelle 1: Zusammenfassung des theoretischen Datenaustausches ....

Tabelle 2: Interaktion aller Komponenten über die wichtigsten Topics ....

Tabelle 3: Implementierte Anforderungen und in welchem Kapitel die Anforderungen ....

Abkürzungsverzeichnis

AGV	Automated guided vehicles
FTF	Fahrerlose Transportfahrzeuge
FTS	Fahrerlose Transportsysteme
HAW	Hochschule für Angewandte Wissenschaften
IDE	Integrierte Entwicklungsumgebung
ID	Identifikator
IoT	Internet-of-Things
MQTT	Message Queue Telemetry Transport
M2M	Machine-to-Machine
PAP	Programmablaufplan
QoS	Quality-of-Service
UPC-UA	Open Platform Communications – Unified Architecture
UUID	Universally Unique Identifier
VM	virtuelle Maschine
ZE	Zeiteinheit

1 Einleitung

1.1 Einführung

Durch den stetig wachsenden Bedarf an Individualisierung und Flexibilisierung in der Produktion wird ein hohes Maß an Autonomie von den Automatisierungslösungen gefordert. Um den Materialfluss flexibler zu gestalten, werden zunehmend neben den gängigen Lösungen fahrerlose Transportfahrzeuge bzw. -systeme eingesetzt, wodurch längere Transportfahrten ermöglicht werden (vgl. Bubeck 2014, S. 221).

Laut Günther (2010, S.44) wird in der Intralogistik der Zukunft das Transportgut dezentral gesteuert. Durch die Dezentralisierung der Intralogistik soll sich die Erweiterbarkeit von Materialflusssystemen vereinfachen. Folglich soll die Planung, Realisierung und Inbetriebnahme von Materialflusssystemen einen finanziellen Vorteil bieten (vgl. Tenerowicz-Wirth 2012, S. 4). Des Weiteren wird „eine verbesserte Flexibilität und Wandelbarkeit sowie eine erhöhte Robustheit gegenüber Störungen und Komponentenausfällen“ in Aussicht gestellt (Tenerowicz-Wirth 2012, S. 25). Folglich müssen die Komponenten autonom auf eine Veränderung der Auftragslasten reagieren und passen sich der Situation an. Durch das Wegfallen einer zentralen Hierarchie werden dementsprechend keine Koordinationskomponenten benötigt (vgl. Günthner 2010, S. 44). Stattdessen wird an dieser Stelle auf die Machine-to-Machine (M2M) -Kommunikation gesetzt, in der es gilt, alle Komponenten zu vernetzen und mit sicheren Verbindungen zu verknüpfen (vgl. Knoll 2020, S. 573). Hierfür ist der Begriff Internet der Dinge (IoT) in der Industrie 4.0 ein zentraler Faktor, der eine dezentrale Selbstständigkeit der Interaktion zwischen den Förderkomponenten sicherstellt (vgl. Günthner 2010, S. 45).

1.2 Zielsetzung

Der Fokus in der vorliegenden Arbeit liegt auf der Konzeptionierung einer Kommunikationsstruktur, in der fahrerlose Transportfahrzeuge (FTF) untereinander Verhandlungen durchführen, mit dem Ziel, einen Fahrauftrag im innerbetrieblichen Transport für sich zu entscheiden. Hierbei steht der dezentrale Aspekt im Vordergrund. Demnach müssen die FTFs eigenständig eine Entscheidung treffen, welches der FTFs, das an den Verhandlungen teilnimmt, den Kundenauftrag berechtigterweise erhält.

Um das Ergebnis darstellen zu können, soll eine Simulation einer dezentralen Produktion entwickelt werden, die auf einem zuvor entwickelten Planspiel beruht (Anhang A). Für die Realisierung dieser Arbeit werden dafür Teilziele im folgenden Kapitel definiert.

Teilziel 1: Analyse des Planspiels und Erstellung einer Anforderungserhebung

Für die Realisierung einer Kommunikationsstruktur zwischen FTFs werden Anforderungen benötigt, aus denen Kommunikationsabläufe aufeinander abgestimmt und realisiert werden. Anhand eines Planspiels, das an der HAW zuvor entwickelt worden ist und eine dezentrale Schokoladenproduktion darstellt, werden durch eine Analyse die Grundanforderungen für eine Kommunikation erhoben. Um eine Kommunikationsstruktur in eine Simulation zu implementieren, werden zusätzlich erweiterte Anforderungen benötigt, die in den ersten Projektgesprächen mit den Teilnehmern/innen gemeinsam festgelegt wurden.

Teilziel 2: Konzipierung der Kommunikationsstruktur und Abläufe zur dezentralen Kommunikation zwischen den vorhandenen Komponenten

Im zweiten Schritt soll die Kommunikationsstruktur aus den herausgearbeiteten Anforderungen entwickelt werden. Hierfür werden Szenarien erstellt, die an das Planspiel angelehnt sind und die Abläufe durch Dokumentation erfasst. Aus den erfassten Abläufen soll ein Programmablaufplan erstellt werden, aus dem ersichtlich ist, in welcher Reihenfolge welcher Prozess erfolgt. Des Weiteren sollen die Abläufe mit einem Netzwerk verbunden werden, in dem die Komponenten untereinander interagieren können. Anschließend soll die komplette Kommunikationsstruktur in einem Diagramm abgebildet werden, aus dem zu entnehmen ist, welche Komponente wann und mit wem kommuniziert.

Teilziel 3: Validierung des erarbeiteten Konzeptes und Vorstellung der Simulation

Im letzten Teil soll eine in die Praxis umgesetzte Simulation vorgestellt werden. Um die Abläufe aus dem Planspiel in der Simulation darzustellen, sollen einzelne Szenarien anhand von

Ausschnitten aus der Simulation dargestellt werden, in denen ersichtlich ist, wie die Komponenten untereinander interagieren.

1.3 Herangehensweise an die Arbeit

Nachdem die wissenschaftlichen Grundlagen geklärt wurden, wird das Planspiel vorgestellt, das im Rahmen eines vorangegangenen Projektes entstanden ist und an der HAW entwickelt wurde. Das Planspiel, das eine dezentrale Schokoladenproduktion darstellt, ist grundlegend für die Erstellung einer digitalen Simulation, weil der Schwerpunkt der Arbeit die Kommunikation zwischen den FTFs ist, die im Planspiel untereinander Verhandlungen mit dem Ziel durchführen, einen Kundenauftrag für sich zu entscheiden. Hieraus resultieren Anforderungen, die für die Umsetzung der Simulation notwendig sind. Über das Planspiel hinaus werden zusätzliche Anforderungen benötigt, die im Rahmen einiger Projektgespräche mit den Teilnehmern entstanden sind.

Nachdem die Anforderungen im Planspiel geklärt wurden, wird das Planspiel auf seine Strukturen analysiert. Hierfür wird zu Beginn das Ziel der Simulation geklärt. Für die Analyse des Planspiels wird ein Szenario vorgestellt, das sich an das Planspiel anlehnt und eine Verhandlung zwischen den FTFs darstellt. Aus der Analyse werden Daten entnommen, die für die Erstellung eines Programmablaufplanes relevant sind. Der Programmablaufplan hilft bei der Strukturierung der Funktionen für die Simulation, woraus eine Kommunikationsstruktur abgeleitet werden kann. Schließlich können die Funktionen erstellt werden, die für die Realisierung der Simulation entscheidend sind. Hieraus resultiert ein Diagramm, das eine vollständige Struktur der Funktionen aller Komponenten inklusive der Kommunikationsstruktur darstellt.

Nach der generellen Übersicht der Struktur wird der technische Aspekt für die Realisierung der Simulation vorgestellt. Zu Beginn wird der Aufbau einer Plattform präsentiert, die elementar für die Umsetzung der Simulation ist. Anschließend wird auf die wichtigsten Funktionen der Komponenten ‚Kundenauftrag‘ und ‚Maschine‘ eingegangen. Abschließend wird die Wegfindung der Fahrzeuge erläutert.

Für die visuelle Darstellung der entwickelten Strukturen werden Szenarien vorgestellt, die den Schwerpunkt der Arbeit bilden. Des Weiteren werden die Eckpfeiler des Planspiels aus den Szenarien kenntlich gemacht, die den Transport und die Bearbeitung eines Kundenauftrages in der Simulation ermöglichen. Abschließend werden die Ergebnisse zusammengefasst und ein Ausblick für eine Weiterentwicklung des Projektes gegeben.

1.4 Abgrenzung

Das Konzept einer Verhandlung zwischen FTFs in der vorliegenden Arbeit dient als Prototyp für Erkenntnisse, die im späteren Verlauf des Projektes an realen FTFs getestet werden sollen. Wie im vorherigen Kapitel dargelegt, wird in erster Linie auf die Programmierung einer Simulation eingegangen, in der die Verhandlungen der FTFs um einen Kundenauftrag im Vordergrund stehen sollen. Hierfür wird ein Planspiel zur Verfügung gestellt, das in einer vorherigen Arbeit dieses Projektes entwickelt wurde. Nachdem das Planspiel analysiert wurde, hat sich herausgestellt, dass nicht alle Regeln des Planspiels und Funktionen der Komponenten, die im Planspiel enthalten sind, für die Umsetzung einer Interaktion zwischen den FTFs in der Simulation benötigt werden. Infolgedessen wird eine Interaktion zwischen den Komponenten ‚Maschine' und ‚Kundenauftrag' und zwischen den Maschinen nicht abgebildet. Des Weiteren werden die Puffer auf jeweils einen Puffer hinter jeder Maschinengruppe reduziert und an die dritte Maschinengruppe wird ein Kundenauftrag lediglich abgegeben und folglich als produziert markiert. Im Gegensatz zum Planspiel, in dem jede Aktion rundenbasiert verläuft, wird der Ablauf der Simulation in Echtzeit realisiert.

2 Wissenschaftliche und technische Grundlagen

Dieses Kapitel gibt einen Einblick in die wissenschaftlichen Grundlagen zu den Themen, die im Hauptteil dieser Arbeit zur Anwendung kommen. Als erstes folgt eine Gegenüberstellung der zentralen und der dezentralen Produktionssteuerung. Danach werden die Begrifflichkeiten fahrerlose Transportfahrzeuge und -systeme erläutert. Weiterhin wird der Begriff Internet der Dinge (IoT), der MQTT beinhaltet, erklärt, worauf im Folgenden die Methoden der Wegfindung vorgestellt werden. Anschließend wird die Vorgehensweise in der Softwareentwicklung beschrieben.

2.1 Zentrale/ dezentrale Produktionssteuerung

Die zentrale Produktionssteuerung ist darauf ausgerichtet, die Planung aller Fertigungsprozesse einer Produktion in einer zentralen Planungsinstanz zu vereinen. Hierdurch kann grundsätzlich eine optimale Planungseinheit erstellt werden (vgl. Loos 2020). Jedoch entsteht in der zentral gesteuerten Produktion ein Nachteil, wenn während des Planungsprozesses Änderungen vorgenommen werden. Demzufolge ist der bereits ausgearbeitete Produktionsplan veraltet (vgl. Weckmann 2020, S.6), was dazu führt, dass operative Ziele der Produktionssteuerung und die Detailplanung nicht eingehalten werden können (vgl. Loos 2020).

Im Gegensatz zur zentralen hat die dezentrale Produktionssteuerung keine zentrale Instanz, die Entscheidungen in der Produktion übernimmt. Stattdessen werden die Aufgaben einer Produktionssteuerung auf einzelne Komponenten verteilt, die autonom Entscheidungen treffen, indem sie untereinander interagieren (vgl. Tenerowicz-Wirth 2012, S.25). Für die Verarbeitung und Bewertung der Aufgaben benötigen die Komponenten einer dezentralen Steuerung die notwendigen Informationen. Für den Austausch der Informationen sind die Komponenten über ein Kommunikationssystem miteinander verbunden. Durch die direkte Interaktion untereinander werden die Reaktionszeit und die Entscheidungswege verkürzt, wodurch eine flexiblere Fertigung ermöglicht wird (vgl. Weckmann 2020, S.39), was jedoch zu einem erhöhten Datenaufwand innerhalb der internen Systemgrenzen führt. Durch das Wegfallen einer einzelnen Komponente fällt im Gegensatz zur zentral gesteuerten Produktion nur ein Teil der Produktion aus (vgl. Tenerowicz-Wirth 2012, S.25).

2.2 Fahrerlose Transportsysteme und -fahrzeuge

In der Intralogistik sind fahrerlose Transportsysteme (FTS) ein wichtiges Element. Durch die zunehmende Erfahrung in der Automatisierungstechnik und den technologischen Vorschriften wird vermehrt auf ein FTS in allen Produktionsbereichen gesetzt (vgl. Ullrich 2019, S.1). Ein FTS wird nach der VDI-Richtlinie 2510 wie folgt definiert:

> Fahrerlose Transportsysteme (FTS) sind innerbetriebliche, flurgebundene Fördersysteme mit automatisch gesteuerten Fahrzeugen, deren primäre Aufgabe der Materialtransport, nicht aber der Personentransport ist. Sie werden innerhalb und außerhalb von Gebäuden eingesetzt und bestehen im Wesentlichen aus den folgenden Komponenten:
> - einem oder mehreren fahrerlosen Transportfahrzeugen,
> - einer Leitsteuerung,
> - Einrichtungen zur Standortbestimmung und Lageerfassung,
> - Einrichtungen zur Datenübertragung,
> - Infrastruktur und peripheren Einrichtungen (VDI-Richtlinie 2510, S. 6).

Eine FTS Leitsteuerung hat die Aufgabe Transportaufträge zu verwalten. Des Weiteren ist es die Aufgabe einer FTS Leitsteuerung, Fahraufträge abzuwickeln und die Fahrzeugdisposition zu organisieren (vgl. Ullrich 2019, S.235). Im Gegensatz zu einem FTS, das mit einer Leitsteuerung vernetzt werden kann, sind FTFs flurgebundene Förderfahrzeuge, die autonom mit der Umgebung interagieren (vgl. Bubeck 2014, S.221). Ein fahrerloses Transportfahrzeug wird nach der VDI-Richtlinie 2510 wie folgt definiert:

> Fahrerlose Transportfahrzeuge (FTF) sind flurgebundene Fördermittel mit eigenem Fahrantrieb, die automatisch gesteuert und berührungslos geführt werden. Sie dienen dem Materialtransport, und zwar zum Ziehen und/oder Tragen von Fördergut mit aktiven oder passiven Lastaufnahmemitteln. In dieser Richtlinie werden Fahrzeuge mit Radantrieben betrachtet. Ausgeschlossen werden schienengeführte Fahrzeuge, Luftkissenfahrzeuge sowie Laufmaschinen (VDI-Richtlinie 2510, S. 7).

2.3 Internet der Dinge und die industrielle Kommunikation

Der Begriff Internet of Things (IoT) oder Internet der Dinge beschreibt die digitale Vernetzung von beliebigen Gegenständen des Alltags auf Basis von standardisierten Internettechnologien. Das Ziel ist es, dass vernetzte Gegenstände miteinander interagieren und folglich Daten austauschen können (vgl. Sinsel 2020, S.4).

Damit eine Vernetzung von herstellerübergreifenden Komponenten einbezogen werden kann, ist eine einheitliche Regelung der Schnittstellen sowie ein Ordnungsrahmen, der die Architektur beschreibt, notwendig. Für den Datenaustausch kann UPC-UA (Open Plattform Communica-

tions – Unified Architecture) eine Möglichkeit sein. Hierbei ist zu beachten, dass bei der Steuerung von Fahrzeugen nur wenige Daten übertragen und hochfrequent gesendet werden müssen. Dafür eignen sich Protokolle mit kleinen Nutzdaten wie MQTT, weil das Verhältnis von Verwaltungsdaten und Nutzdaten besser ist (vgl. Tödter 2020, S.173).

MQTT steht für Message Queue Telemetry Transport und ist ein Ereignis orientiertes Nachrichtenprotokoll zur asynchronen Kommunikation zwischen Geräten. Dafür wurde eine Publish/Subscribe-Architektur realisiert. Die Nachrichten werden über einen zentralen Broker übertragen, der die Nachrichten von den Publizisten empfängt und an die Subskribenten weiterleitet, die dann die Nachrichten auswerten (vgl. Trojan 2017, S.12).
Das Senden (publish) und Empfangen (subscribe) von Informationen funktioniert über Topics. Ein Topic ist ein Kommunikationskanal, welcher die Aufgabe hat, Nachrichten an den Broker zu senden. Der Broker prüft, welche Geräte diesen Kanal geöffnet haben und sendet die Nachricht an diese weiter. Die Datenunabhängigkeit von MQTT erlaubt es Strukturen wie JSON oder XML, Texte oder binäre Daten zu übertragen (vgl. Trojan 2017, S.13).

Auf Abbildung 1 ist eine vereinfachte Kommunikation zwischen einem Temperaturfühler, einem Broker und zwei Endgeräten zu sehen. Folglich ist zu erkennen, dass der Temperaturfühler

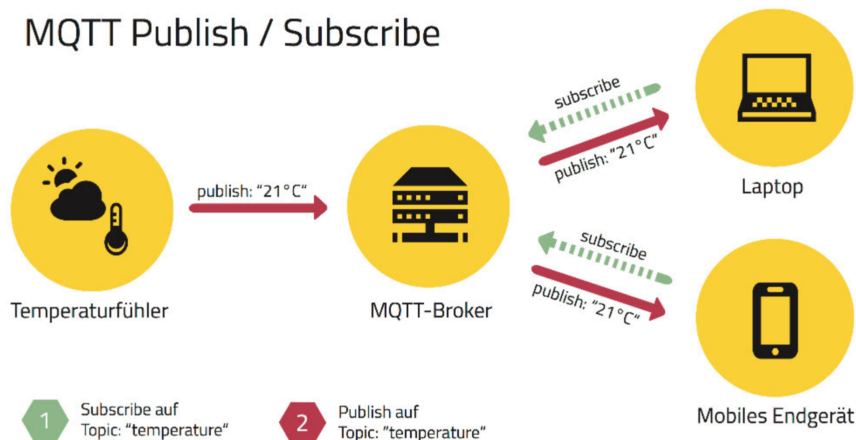


Abbildung 1: Darstellung des Publizierens und der Subskribierung (Quelle: Raschbichler 2017)

in diesem Beispiel die einzige Komponente ist, die eine Nachricht über das Topic „temperature“ über den Broker publizieren kann. Der Laptop und das mobile Endgerät hingegen haben das Topic „temperature“ subskribiert und sind demnach in der Lage die Nachrichten zu empfangen. Sollte die Kommunikation aufgrund eines technischen Problems in Form einer Netzunterbrechung, ein spontanes Abschalten eines Fahrzeuges oder einer leeren Batterie auftreten, so bietet MQTT eine „Last Will“- (letzter Wille oder Testament) Funktion an. Dadurch wird eine Nachricht auf dem Broker hinterlassen, die bei einem Ausfall der Komponente an die Subskribenten des Topics gesendet wird (vgl. Trojan 2017, S.38).

2.4 Methoden der Wegfindung

Damit eine Route von einem Ausgangspunkt zu einem Ziel berechnet werden kann, gibt es unterschiedliche Algorithmen für die Wegfindung. Alle Wegfindungsalgorithmen funktionieren mit einem Gittergraphen, in dem der kürzeste Weg zu ermitteln ist. Ein Gittergraph besteht aus Knoten und Kanten. Ein Knoten kann ein Anfangspunkt, ein Endpunkt oder ein Punkt zwischen beiden sein. Der Weg zwischen den Knoten wird Kante genannt. Die am häufigsten verwendeten Wegfindungsalgorithmen für die kürzeste Strecke sind der Dijkstra-Algorithmus und der A*-Algorithmus oder auch A-star genannt (vgl. Graphalgorithmen).

Der Dijkstra-Algorithmus kann den kürzesten Weg zwischen einem Anfangsknoten A und einem Zielknoten B berechnen, wenn die Kanten ein Gewicht (Kosten) haben und es mehrere Wege zum Zielknoten gibt. Die Gewichtung einer Kante kann unter anderem als ein Abstand in Metern definiert sein. Der Dijkstra-Algorithmus berechnet kreisförmig jeden möglichen Weg von Knotenpunk A nach Knotenpunkt B und ermittelt dabei die Kosten der Wege. Anschließend ist der Weg, der die geringsten Kosten hat, folglich der kürzeste (vgl. Velden 2014, Nr. 1).

Der A*-Algorithmus hingegen berechnet die Kosten gezielter, weil dieser Algorithmus die Möglichkeit hat, Kosten zu schätzen. Der Algorithmus kennt den Zielort des Zielknotens aber nicht den kürzesten Weg. Um den kürzesten Weg zum Zielknoten zu berechnen, untersucht der Algorithmus jeden Knoten, der wahrscheinlich am nächsten zum Zielknoten ist. Bei der Ausführung des A*-Algorithmus wird eine Warteschlange eingesetzt, in der an erster Stelle immer der Knoten besetzt wird, der die geringsten Kosten zum Zielknoten aufweist. Die geringen Kosten bestimmt der f-Wert. Der f-Wert setzt sich aus der Summe der Distanz vom aktuellen Knoten und dem Schätzwert vom aktuellen Knoten zum Zielknoten zusammen (vgl. Velden 2014, Nr. 2).

2.5 Vorgehensweise in der Softwareentwicklung

Laut Brandt-Pook und Kollmeier (2020, S.1) benötigt die Entwicklung einer neuen Software eine systematische Vorgehensweise.

Die Phase der Aufklärung

Häufig beginnt die Phase der der Aufklärung mit einer Ist-Analyse. Die Ist-Analyse dient der objektiven Ermittlung eines Ausgangszustandes in einem Unternehmen. Hierbei sollen Schwachstellen erkannt werden, die im Anschluss durch den Soll-Zustand beseitigt werden können (vgl. Brandt-Pook 2020, S.9).

Die Phase der Konzeption

In dieser Phase wird festgelegt, welche Anforderungen die Software beinhalten soll. Nachdem die Anforderungen erarbeitet wurden, kann entschieden werden, welche Funktionen bezüglich der Anforderungen in das System hinzuprogrammiert werden können (vgl. Brandt-Pook 2020, S.11 f.).

Die Phase Design

In der Designphase wird das Gerüst der Software entwickelt. Hierfür müssen die Baugruppen der Architektur erarbeitet und anhand der Anforderungen verknüpft werden (vgl. Brandt-Pook 2020, S.15).

Die Phase Realisierung

In dieser Phase werden die zuvor erarbeiteten Strukturen aus der Designphase in einen Algorithmus transformiert. Diese Phase wird auch als Implementierungsphase definiert (vgl. Brandt-Pook 2020, S.16).

Die Phase der Einführung

Bei der Einführung geht es um den praktischen Ansatz. Hierbei werden die in den Anforderungen festgelegten Elemente anhand von Szenarien in der Praxis dargestellt (vgl. Brandt-Pook 2020, S.16-18).

Die Phase Testen

Die Phase des Testens ist dafür da, um Fehler ausfindig zu machen und zu beseitigen. Im Vergleich zu den anderen Phasen, findet das Testen während der Konzeptions-, Design-, Realisierungs- und Einführungsphase statt (vgl. Brandt-Pook 2020, S.19 ff.).

Die in diesem Abschnitt beschriebenen Phasen dienen in dieser Arbeit lediglich als Orientierung und werden deshalb nicht vollständig übernommen. Die Phase der Aufklärung ist in der vorliegenden Arbeit nicht relevant, weil neu programmiert und deshalb keine Ist-Analyse benötigt wird.

3 Anforderungsanalyse

Zur Ermittlung der Anforderungen wird in diesem Kapitel zuerst ein Grundlagenmodell vorgestellt, aus dem ein Planspiel entworfen wurde. Das Planspiel dient für diese Arbeit als Grundlage und wird für die Anforderungserhebung vorgestellt und analysiert. Anschließend werden die Anforderungen ermittelt, die für eine Umsetzung des Planspiels in eine Simulation zusätzlich benötigt werden.

3.1 Voraussetzungen für die Arbeit

Als Grundlage der vorliegenden Arbeit gilt das Swimming-Pool-Beispiel von Prof. Gärtner aus der Vorlesung „Digitale Produktion", in dem ein Szenario vorgestellt wird, wie eine dezentrale Produktion aussehen kann. Das in Abbildung 2 enthaltene Modell einer dezentralen Produktionssteuerung dient als Vorlage des im Kapitel 3.2 beschriebenen Planspiels. Der Grundgedanke hierbei ist, dass die Aufträge den Takt im dargestellten Produktionsverlauf vorgeben und folglich unterschiedliche Funktionen auslösen. Dabei spielt die Unabhängigkeit der einzelnen Komponenten voneinander, in Bezug auf die Verhandlung von der Annahme eines Kundenauftrages, eine übergeordnete Rolle. Das heißt, dass die FTFs nicht von einer zentralen Instanz gesteuert werden, sondern, dass die FTFs nach einem Auftragseingang untereinander Verhandlungen mit dem Ziel durchführen, dass eines der FTFs den Kundenauftrag übernimmt.

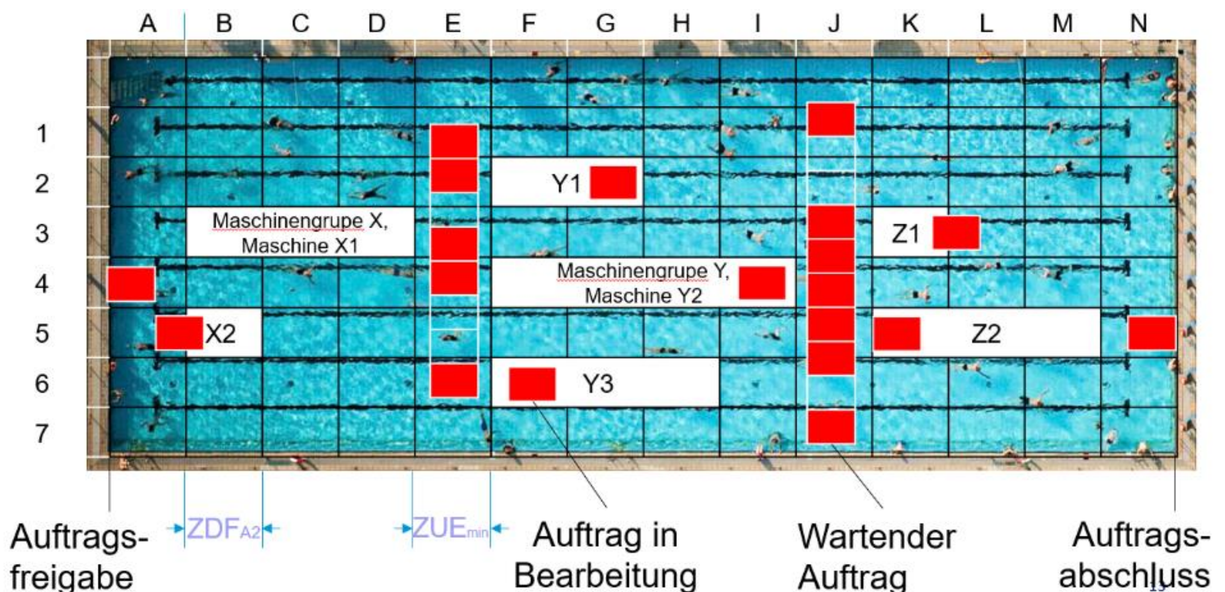


Abbildung 2. Swimming-Pool-Modell (Gärtner 2018, S.84)

Infolgedessen sind zwei Projekte entstanden. Das erste Projekt bezieht sich auf ein Fallbeispiel, in dem ein Konzept für eine dezentrale Produktion entwickelt wurde. Außerdem wurden ein FTF und ein Testfeld entwickelt, auf dem das FTF eigenständig einer vorgesehenen Spur folgt.

Das zweite Projekt, in dem ein Planspiel entwickelt wurde, bezieht sich überwiegend auf das Modell in Abbildung 2. Das Planspiel, das eine dezentrale Schokoladenproduktion darstellt, dient als Grundlage für die vorliegende Arbeit, weshalb das Planspiel im nachfolgenden Abschnitt detaillierter vorgestellt werden soll.

3.2 Beschreibung des Planspiels

Das Planspiel wurde als Brettspiel konzipiert und besteht aus einem Spielfeld, das aus einem Raster mit 9x13 Kästchen besteht. Jedes Kästchen steht hierbei für eine Zeiteinheit (ZE). Abbildung 3 bildet das vollständige Brettspiel ab. Die Produktionskette verläuft von links nach rechts bzw. vom Auftragseingang (Punkt 1) bis zum Lager, das sich hinter den drei Ausgängen V1, V2, V3 (Punkt 2) befindet. Dazwischen befinden sich zwei Maschinengruppen, die in S1, S2, S3 (Punkt 3) und E1, E2 (Punkt 4) aufgeteilt sind. Hinter jeder Maschinengruppe befinden sich Puffer, in denen die abgearbeiteten Aufträge zwischengelagert werden können. Zwischen dem Auftragseingang, den Maschinengruppen und dem Lager befinden sich insgesamt vier Fahrzeuge, deren Aufgabe darin besteht, Kundenaufträge zu transportieren (Punkt 5). Das Planspiel ist für 1 - 10 Spielern vorgesehen.

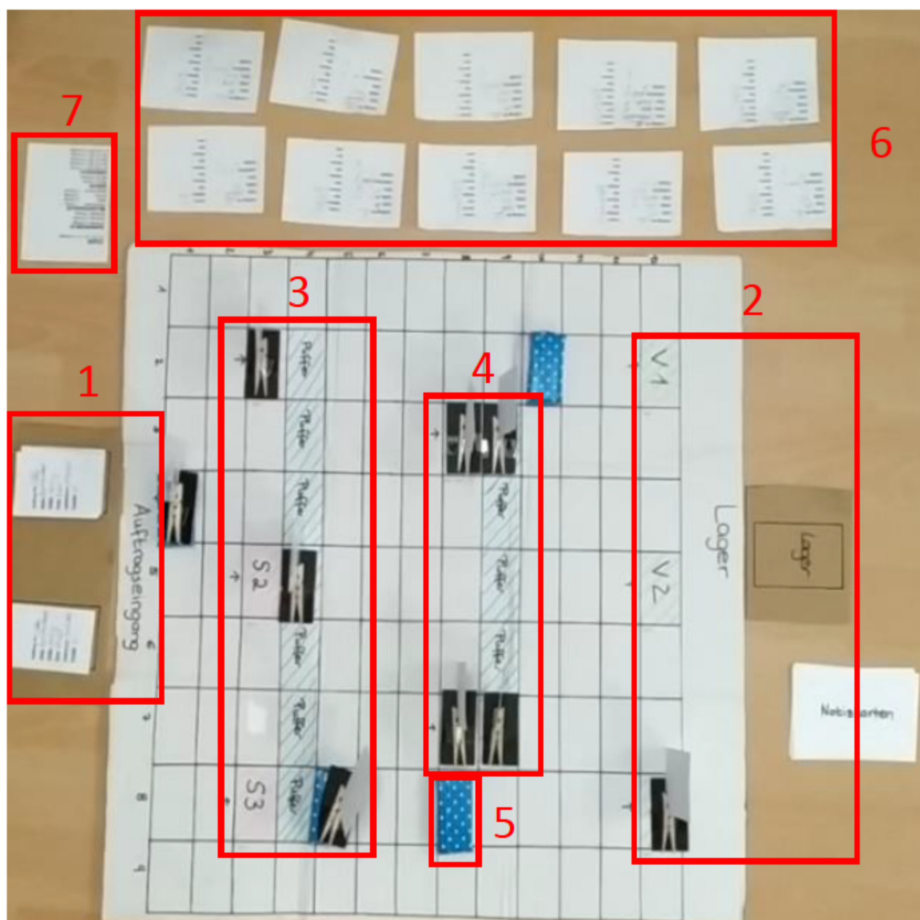


Abbildung 3: Planspiel (Quelle: eigene Darstellung)

Je nachdem, wie viele Spieler an der Durchführung beteiligt sind, wird jedem Spieler am Anfang eine gleichmäßige Anzahl von Kundenaufträgen zugewiesen. Auf den Auftragskarten stehen Informationen, die für eine vollständige Fertigstellung des Kundenauftrages benötigt werden. Für jeden Kundenauftrag wird eine Notizkarte (Punkt 6) geführt, in der die Informationen des Kundenauftrages von den Spielern auszufüllen sind. Die Kundenaufträge unterscheiden sich hierbei um die Faktoren Sorte, wie viel Zeit der Kundenauftrag für die Produktion maximal hat und dessen Priorität. Die Priorität ist in drei Gruppen (A, B, C) aufgeteilt, wobei A die höchste Priorität hat, also sehr wichtig und dementsprechend B und C übergeordnet ist. Um aus den ersten zehn Kundenaufträgen den wichtigsten zu ermitteln, werden Informationen aus den Infokarten entnommen. Die Infokarten (Punkt 7) enthalten Daten, die helfen, einen Wert zu ermitteln, der die Relevanz des Kundenauftrages bestimmt. Infolgedessen steht fest, welcher Kundenauftrag als erstes die Kommunikation mit den Transportfahrzeugen aufnehmen darf und dadurch die Simulation der Schokoladenproduktion auslöst. Die Spieler haben jetzt die Aufgabe, untereinander zu verhandeln, mit dem Ziel, dass die Kundenaufträge möglichst schnell von den Transportfahrzeugen durch die Produktion geleitet werden. Hierbei sind die Regeln des Planspiels streng einzuhalten. Das Muster der Verhandlungen wird in Kapitel 4.2.1 ausführlich analysiert. Das Ziel ist es, möglichst alle Aufträge durch die Produktion zu leiten. Dabei sollte kein Kundenauftrag den für ihn vorgesehenen Zeitrahmen überschreiten. Die genauen Spielregeln sind im Anhang zu finden (Anhang A).

3.3 Anforderungserhebung

Im ersten Teil dieses Abschnitts werden die einzelnen Komponenten, die sich in fahrerlose Transportfahrzeuge, selbststeuernder Kundenauftrag und Maschinen unterscheiden und deren Aufgaben im Planspiel aufgelistet. Im zweiten Teil werden die erweiterten Anforderungen aufgelistet, die für eine Realisierung der Simulation benötigt werden und die in mehreren Brainstorming-Sitzungen zusammengetragen wurden. Insgesamt sind 64 Anforderungen zusammengekommen, die einer dreistufigen Gewichtung in Bezug auf die Umsetzungsrelevanz für die Simulation unterzogen wurden.

- **Kann-Anforderungen:** Diese Kategorie ist von der Gewichtung am niedrigsten bewertet. Die Anforderungen beschreiben eine optionale Funktion an die Simulation, die eine untergeordnete Bedeutung hat. In diesem Sinne sind Kann-Anforderungen einfach gewichtet.
- **Soll-Anforderungen:** Anforderungen, die sich in der Soll-Kategorie befinden, beschreiben Merkmale, deren Vorhandensein von Vorteil, allerdings kein Zwang ist. Mit ihnen lässt sich der Basisumfang der Simulation um Funktionalitäten erweitern, die einen relevanten Mehrwert bieten, weswegen diese mit einer zweifachen Gewichtung ausgezeichnet sind.

- **Muss-Anforderungen:** Muss-Anforderungen umfassen Basisfunktionen, die für eine Umsetzung der Simulation wichtig und demnach zwingend zu erfüllen sind. Wegen ihrer hohen Relevanz werden diese dreifach gewichtet.

3.3.1 Gegebene Anforderungen aus dem Planspiel

Bevor die Anforderungen definiert werden können, muss geklärt werden, welche Aufgaben und Funktionen die einzelnen Komponenten im Planspiel haben. Daraus lässt sich anschließend ableiten, welche Funktionen die Komponenten für eine erfolgreiche Realisierung einer Simulation haben müssen und welche Erweiterungen die Komponenten zusätzlich dafür benötigen. Hierfür wird im Folgenden vorgestellt, welche Funktion und Aufgabe jede einzelne Komponente im Planspiel hat.

- **Fahrerlose Transportfahrzeuge:** Die FTFs haben die Aufgabe, die Kundenaufträge von Maschine zu Maschine durch die Produktion zu transportieren. Um einen Kundenauftrag zu erhalten, müssen die freien FTFs untereinander Verhandlungen durchführen, mit dem Ziel, den Kundenauftrag zu erlangen. Nachdem ein FTF als Gewinner aus der Verhandlung hervorgeht, muss der Kundenauftrag als erstes von dem Ort abgeholt werden, an dem er sich gerade befindet und anschließend an seinen Zielort transportiert werden. Während des Transports ist das FTF für eine weitere Verhandlung nicht ansprechbar. Nachdem das FTF den Kundenauftrag am Zielort abgeliefert hat, wird der Kundenauftrag der entsprechenden Maschine übergeben und steht für eine neue Verhandlung zur Verfügung. Demnach kann ein FTF nur einen Kundenauftrag pro Fahrt aufnehmen. Zudem bewegt sich ein FTF immer nur ein Raster pro ZE auf dem Spielfeld.
- **Kundenauftrag:** Jeder Kundenauftrag hat das Ziel, möglichst schnell durch die Produktion von Maschine zu Maschine bis zum Lager transportiert zu werden, um anschließend als ‚Fertig‘ markiert zu sein. Hierbei steht jeder Kundenauftrag, außer während des Transportes und während er in einer Maschine gefertigt wird, in Kommunikation mit allen verfügbaren FTFs und den Maschinengruppen. Jeder Kundenauftrag hat individuelle Eigenschaften, die ihn auszeichnen. Diese Eigenschaften helfen den FTFs, den Kundenauftrag durch die gesamte Produktion zu geleiten.
- **Maschinen:** Wie in Kapitel 3.2 erwähnt, gibt es insgesamt acht Maschinen, die in drei Maschinengruppen eingeteilt sind. Jede Maschinengruppe bearbeitet den für ihn vorgesehenen Fertigungsschritt. In der zweiten Maschinengruppe besteht die Möglichkeit flexibel auf einen weiteren Fertigungsschritt umzurüsten, wofür eine Rüstzeit beansprucht wird. Jede Maschine hat für die Fertigung eine festgelegte Bearbeitungszeit. Sobald eine Transportanfrage von einem Kundenauftrag an die Maschinengruppe gesendet wurde, verhandeln die Maschinen in der angefragten Maschinengruppe untereinander, welche Maschine den Kundenauftrag übernehmen kann. Wenn die Verhandlungen in der Maschinengruppe abgeschlossen sind und für eine der Maschinen erfolgreich war, ist die

Maschine exklusiv für den Kundenauftrag für die Dauer der Bearbeitung reserviert. Nachdem ein Kundenauftrag von einer Maschine bearbeitet wurde, wird er in einen der freien Puffer abgelegt, die sich hinter den Maschinen befinden.

Sobald bekannt ist, wie der Ablauf des Planspieles ist und welche Aufgaben die Komponenten haben, können daraus die Anforderungen abgeleitet werden. Anschließend wird jeder Anforderung eine Gewichtung zugeschrieben.

Anforderungen aus dem Planspiel:

Planspiel allgemein:

1. Das Planspiel ist rundenbasiert (Kann)
2. Es ist ein Spielfeld vorhanden (Produktionsstraße) (Muss)
3. Das Fahren der FTFs wird im Planspiel dargestellt (Muss)
4. Infokarten, Notizkarten zum Notieren von Eckdaten (Muss)
5. 3 Auftragseingänge (Muss)
6. 3 Auftragsausgänge (Muss)
7. 5 Fertigungsstationen (Muss)
8. Spieler führen Verhandlungen für die FTFs durch (FTFs in der Simulation) (Muss)
9. Die Spieler führen Verhandlungen für die Maschinen durch (Maschinen in der Simulation) (soll)
10. Pufferplätze hinter den Maschinen (Muss)

Fahrerlose Transportfahrzeuge:

11. FTF Status (verfügbar/nicht verfügbar) (Muss)
12. FTF hat Informationen des zu transportierenden Kundenauftrages (Muss)
13. Dauer der Fahrzeugbelegung (Soll)
14. 4 FTFs (Muss)
15. Reservierung eines Kundenauftrages durch ein FTF (Muss)
16. Transport eines Kundenauftrages durch ein FTF (Muss)
17. Individuelle ID für jedes FTF (Muss)
18. Aktuelle Position eines FTFs auf dem Raster kann abgelesen werden (Muss)

Selbststeuernder Kundenauftrag:

19. Jeder Kundenauftrag enthält Informationen über einen eindeutigen Weg durch die Produktion (Muss)
20. Der Kundenauftrag löst das Planspiel aus, die nachfolgenden Aufträge halten das Planspiel in Gang (Muss)
21. Sobald ein Kundenauftrag für die FTFs zur Verfügung steht, muss die aktuelle ZE notiert werden (Kann)

22. Jeder Kundenauftrag hat eine maximale Produktionsdauer bis zum Liefertermin in der ZE, die nicht überschritten werden darf (Kann)
23. Jeder Kundenauftrag hat eine eigene Priorität (Kann)
24. Der Kundenauftrag enthält Informationen über die aktuelle Position auf dem Raster (Kann)
25. Zeitstempel pro angesteuerter Maschine (Maschine, Puffer, FTF) (Kann)
26. Individuelle ID für jeden Kundenauftrag (Muss)
27. Welcher Kundenauftrag wurde von welchen FTF gefahren? (Soll)
28. Zufällige Aufträge werden am Auftragseingang generiert (Muss)

Maschinen:

29. Maschinen Status (verfügbar/nicht verfügbar) (Soll)
30. Dauer der Maschinenbelegung (Soll)
31. Reservierung einer Maschine durch einen Kundenauftrag (Soll)
32. Die Maschinen übernehmen zugestellte Aufträge (Muss)
33. Jede Maschine schiebt einen fertigen Kundenauftrag in einen freien Puffer (Muss)
34. Rüstzeit an den Maschinen in ZE (Soll)
35. Bearbeitungszeit an der Maschine ist frei einstellbar (Soll)

Nachdem das Planspiel analysiert wurde und die ersten 35 Anforderungen aus dem Planspiel notiert worden sind, kann eine erste Struktur der Kommunikation entwickelt werden, die als Basis für die Realisierung der ersten Funktionen in der Simulation dient. In Kapitel 4.2 wird die Herangehensweise an eine Kommunikationsstruktur erläutert, indem die Kommunikation anhand eines Szenarios aus dem Planspiel analysiert wird.

3.3.2 Erweiterte Anforderungen für die Simulation

Neben den Anforderungen aus dem Planspiel werden für die Simulation weitere Anforderungen benötigt, um diese zu realisieren. Hierfür muss sichergestellt werden, welche Funktionen die einzelnen Komponenten in der Simulation haben müssen, die im Planspiel von Spielern entweder manuell oder durch Kommunikation unter den Spielern durchgeführt werden. Für die Gestaltung einer Funktion muss demnach klar sein, was eine Funktion leisten soll. Im Rahmen einiger Brainstorming-Sitzungen, die im Laufe des Projektes stattfanden und an denen Prof. Gärtner und Herr Peters beteiligt waren, wurden Vorschläge gemacht, was jede Komponente in der Simulation leisten muss und wie die Komponenten es umsetzen sollen, damit eine Realisierung ermöglicht wird. Aus den Notizen lassen sich einzelne Schritte für die Simulation ableiten. Des Weiteren ist noch anzumerken, dass die Verhandlungen im Planspiel ausschließlich zwischen den Spielern stattfinden. In der Simulation übernehmen diese Aufgabe die FTFs,

wodurch den FTFs eine Schlüsselrolle in der Simulation zukommt und folglich für die Entwicklung einer Kommunikationsstruktur notwendig sind. Die zusätzlichen Anforderungen sind wie in Kapitel 3.3.1 in Komponenten unterteilt. Die Komponente FTF ist in diesem Abschnitt in Unterpunkte eingeteilt. Diese sind zum einen die Initialisierung, was den Anmeldevorgang eines FTFs darstellt, die Verhandlungen, in denen die FTFs untereinander kommunizieren, mit dem Ergebnis, dass eines der FTFs den Zuschlag für einen Kundenauftrag bekommt und schließlich der Transport eines Kundenauftrages.

Zusätzliche Anforderungen für die Realisierung der Simulation:

Fahrerlose Transportfahrzeuge:

- Initialisierung eines FTF
  36. Herstellung der Kommunikation mit einem Netzwerk (Muss)
  37. An- und Abmeldung eines FTFs ist jederzeit möglich. Wenn im Laufe der Simulation ein weiteres FTF benötigt wird, kann von außen eingegriffen und ein neues FTF in die Simulation integriert werden. Wenn zu viele oder nicht benötigte FTFs in der Simulation sind, kann das nicht benötigte FTF abgemeldet werden (Soll)
  38. Die FTFs teilen die eigene ID untereinander, wodurch jedes FTF weiß, welches und wie viele FTFs gerade aktiv sind (Muss)
  39. Manuelle ID-Vergabe für FTF möglich (Kann)

- Verhandlung
  40. Jedes FTF kann Transportanfragen von Kundenaufträgen empfangen (Muss)
  41. Jedes FTF muss eine neue Transportanfrage eines Kundenauftrages ablehnen, falls das FTF beschäftigt ist (Muss)
  42. Jedes FTF speichert Informationen des aktuell verhandelten Kundenauftrags (Muss)
  43. Jedes FTF teilt den eigenen Standort mit den anderen FTFs (Muss)
  44. Jedes FTF vergleicht den eigenen Weg zum Ziel mit dem Weg der anderen FTFs (Muss)
  45. Wenn zwei oder mehrere FTFs die gleiche Fahrstrecke zu einem Kundenauftrag haben, muss durch einen Mechanismus entschieden werden, welches FTF den Zuschlag für einen Kundenauftrag erhält, über den gerade verhandelt wird (Muss)
  46. Dem Kundenauftrag mitteilen, dass er von einem FTF übernommen wurde (Soll)
  47. FTFs sollen einen letzten Wunsch bei Kommunikationsabbruch haben (Last Will), falls es zu Komplikationen kommt (Akku leer) (Soll)

- Transport eines Kundenauftrages
  48. Den Maschinen mitteilen, dass ein Kundenauftrag zugestellt wurde (Muss)

49. Jedes FTF speichert jeden Kundenauftrag in einer Liste, in der zu erkennen ist, welcher Kundenauftrag an welcher Maschine war (Soll)
50. Beim Fahren soll jedes FTF dynamischen Hindernissen ausweichen können (Kann)
51. Jedes FTF hat ein Akkuladesystem (Kann)
52. Im Quellcode muss die Möglichkeit bestehen, die Fahrgeschwindigkeit eines FTF einzustellen (Soll)

Kundenauftrag:

53. Bei Auftragsablehnung muss der abgelehnte Kundenauftrag eine neue Transportanfrage an die FTFs senden (Muss)

Maschinen:

54. Jede Maschine enthält Informationen des Kundenauftrages, der gerade in einer Maschine produziert wird (Muss)
55. Die Maschine sendet dem Kundenauftrag eine Information, auf welchem Puffer der Kundenauftrag sich befindet (Muss)

Monitoring:

56. Ein Hauptmenü zur Übersicht von Eckdaten (Kann)
57. Der Anwender kann ablesen, welcher Kundenauftrag von welchem FTF übernommen wurde (FTF) (Soll)
58. Der Anwender kann ablesen, von welcher Position der Kundenauftrag abgeholt und wohin der Kundenauftrag transportiert werden soll (Muss)
59. Der Anwender kann ablesen, welche Strecke abgefahren werden soll (FTF) (Muss)
60. Der Anwender kann ablesen, dass ein Kundenauftrag zugestellt wurde (FTF) (Soll)
61. Der Anwender kann ablesen, dass ein Kundenauftrag in Bearbeitung ist (Maschine) (Muss)
62. Der Anwender kann ablesen, dass ein Kundenauftrag teilbearbeitet wurde (Maschine) (Muss)
63. Der Anwender kann ablesen, wenn ein Kundenauftrag fertiggestellt wurde (Maschine) (Muss)
64. Der Bearbeitungszustand an den Maschinen soll angezeigt werden (Maschine) (Kann)

Im weiteren Verlauf kann aus den aufgenommenen Anforderungen ein Konzept für die Umsetzung der Simulation erstellt werden. Hierbei bieten die Anforderungen eine Basis für die Gestaltung von Funktionen als auch den Nutzen, was die einzelnen Funktionen im späteren Quellcode leisten sollen.

4 Konzept und Datenmodell der Verhandlungen

In diesem Kapitel werden die Anforderungen aus dem vorherigen Kapitel zusammengetragen und das daraus resultierende Design der Simulation vorgestellt. Als erstes wird ein Ziel dahingehend festgelegt, was die Simulation leisten soll. Darauf folgt ein Szenario aus dem Planspiel, indem die Verhandlungen zwischen den FTFs analysiert werden. Aus dem Ergebnis wird eine Struktur entwickelt, die als Basis für den zu entwickelnden Algorithmus dient. Daraufhin wird ein kleines Szenario der Kommunikation zwischen den FTFs vorgestellt, indem das MQTT-Protokoll verwendet wird. Darüber hinaus werden die Funktionen für die Initialisierung anhand eines Programmablaufplans (PAP) vorgestellt. Abschließend soll die Kommunikationsarchitektur anhand einer Swimlane den Verhandlungsprozess bildlich darstellen.

4.1 Ziel der Simulation

In diesem Abschnitt wird das Ziel der Simulation beschrieben, an dem sich die Entwicklung einer Kommunikationsstruktur anlehnt. Nachdem die Anforderungen gewichtet wurden, ist bekannt, welche davon unabdingbar für die Realisierung der Simulation sind. Diese bilden das Grundgerüst, worauf eine Kommunikationsstruktur entwickelt werden kann. Zunächst müssen dennoch Ziele definiert werden, was die Simulation erreichen soll.

Allgemeines Ziel: Das grundlegende Ziel ist es, Kundenaufträge zu produzieren. Dafür werden, wie im Planspiel die drei Komponenten Kundenauftrag, FTF für den Transport und die Maschinen für die Produktion benötigt. Dabei ist der Kundenauftrag der Taktgeber der Simulation, solange er nicht vollständig produziert wurde. Die FTFs sollen nach Erhalt einer Transportanfrage eines Kundenauftrages selbstständig untereinander Verhandlungen durchführen, mit dem Ziel, dass eines der FTFs als Gewinner aus den Verhandlungen hervorgeht und den Zuschlag für die Abholung und den Transport des Kundenauftrages zu dessen Ziel erlangt. Nachdem das FTF an der Maschine angekommen ist, soll das FTF den Kundenauftrag der Maschine übergeben. Die Maschine muss den Kundenauftrag annehmen und mit der Teilproduktion beginnen. Nachdem die Maschine mit der Bearbeitung fertig ist, muss der Kundenauftrag von der Maschine in einen der freien Puffer gelagert werden, die sich hinter den Maschinen befinden. Sobald der Kundenauftrag in einem der Puffer abgestellt wurde, sendet er eine neue Transportanfrage an die FTFs, um ihn zu der nächsten Maschine zu transportieren. Infolgedessen verhandeln die FTFs erneut. Der Kundenauftrag muss die drei Maschinengruppen durchlaufen haben, um als produziert zu gelten. Wichtig ist, dass von Beginn an, also wenn der Kundenauftrag eingetroffen ist, alle Entscheidungen dezentral getroffen werden. Die Kommunikation zwischen allen Komponenten soll hierbei mit dem MQTT-Protokoll stattfinden.

Rolle des Anwenders (Monitoring): Mit Anwender ist eine Person gemeint, die auf einem Monitor die Ausführung der Simulation beobachtet. Insofern ist es nicht die Aufgabe des Anwenders, die Simulation in Bezug auf die Verhandlungen oder sonst irgendeine Tätigkeit einer Komponente zu manipulieren. Der Anwender soll lediglich Zugriff auf die Eckdaten der Simulation verfügen und diese jederzeit abrufen können. Jeder Schritt, von der Initialisierung der FTFs bis hin zum fertig produzierten Kundenauftrag, soll nachvollziehbar auf dem Bildschirm angezeigt werden. Hierfür sind die Anforderungen aus der Kategorie Monitoring aus dem Kapitel 3.3.2 gedacht.

Ziel der Komponenten: Das Ziel der einzelnen Komponenten, also der FTFs, des Kundenauftrags und der Maschinen orientiert sich an den erhobenen Anforderungen aus den Kapiteln 3.3.1 und 3.3.2.

4.2 Kommunikation zwischen den Komponenten

Für die Verhandlungen zwischen den Komponenten wird ein Konzept benötigt, in dem eine präzise Darstellung einer Kommunikationsstruktur enthalten ist, die anschließend in einen Quellcode transformiert werden kann. Dafür muss bekannt sein, welche Komponente wann mit wem kommuniziert. In diesem Abschnitt wird eingangs das Planspiel und dessen Verhandlungsmuster analysiert, wodurch ein Schema entwickelt werden kann, wie mit Hilfe von MQTT eine Kommunikation in der Simulation aussehen kann. Des Weiteren wird anhand der Initialisierung eine beispielhafte Verhandlung vorgestellt. Abschließend wird ein Konzept vorgestellt, in der die vollständige Kommunikationsstruktur der Simulation abgebildet ist.

4.2.1 Analyse der Verhandlungen im Planspiel

Für die Analyse der Verhandlungen ist es unabdingbar, sich mit dem Planspiel vertraut zu machen. Wie in Kapitel 3.2 bereits erwähnt, übernehmen die Spieler im Planspiel den Teil der Verhandlungen. Auf Abbildung 4 ist ein Szenario aus dem Planspiel abgebildet, in dem entschieden werden muss, welches FTF den Kundenauftrag A2, der sich im Puffer EP1 befindet, abholt und zu einem der Ausgänge fährt. Hierfür wird angenommen, dass der Spielverlauf schon fortgeschritten ist und sich mehrere Kundenaufträge auf dem Spielfeld bzw. in der Produktion befinden. Zudem sind einige FTFs nicht verfügbar, da entweder ein Kundenauftrag transportiert oder abgeholt wird. Diese FTFs sind in dem Szenario rot markiert. Die grün markierten FTFs sind frei und stehen für eine Verhandlung um Kundenauftrag A2 zur Verfügung. Die Spieler haben mehrere Aufgaben. Zum einen müssen die Spieler auf die eigenen Aufträge achten und notieren, wenn ein Kundenauftrag an einer Maschine ankommt und die Produktion beginnt und wann ein Kundenauftrag von einem FTF abgeholt wird. Zum anderen ist es die Aufgabe der Spieler zu entscheiden, welches Fahrzeug welchen Kundenauftrag als Nächstes abholt. Dies kann nur passieren, wenn alle Spieler sich untereinander absprechen und als Folge

sich ein Ergebnis herauskristallisiert, welches den Spielverlauf auf eine positive Art, also möglichst zu wenig bis keinen Verzug der gesamten Produktionsdauer, beeinflusst. Um ein Verhandlungsmuster in der Simulation zu realisieren, wird als erstes eine Liste erstellt, die einer ereignisgesteuerten Reihenfolge folgt, die zeigt, wie eine Verhandlung im Planspiel abläuft.

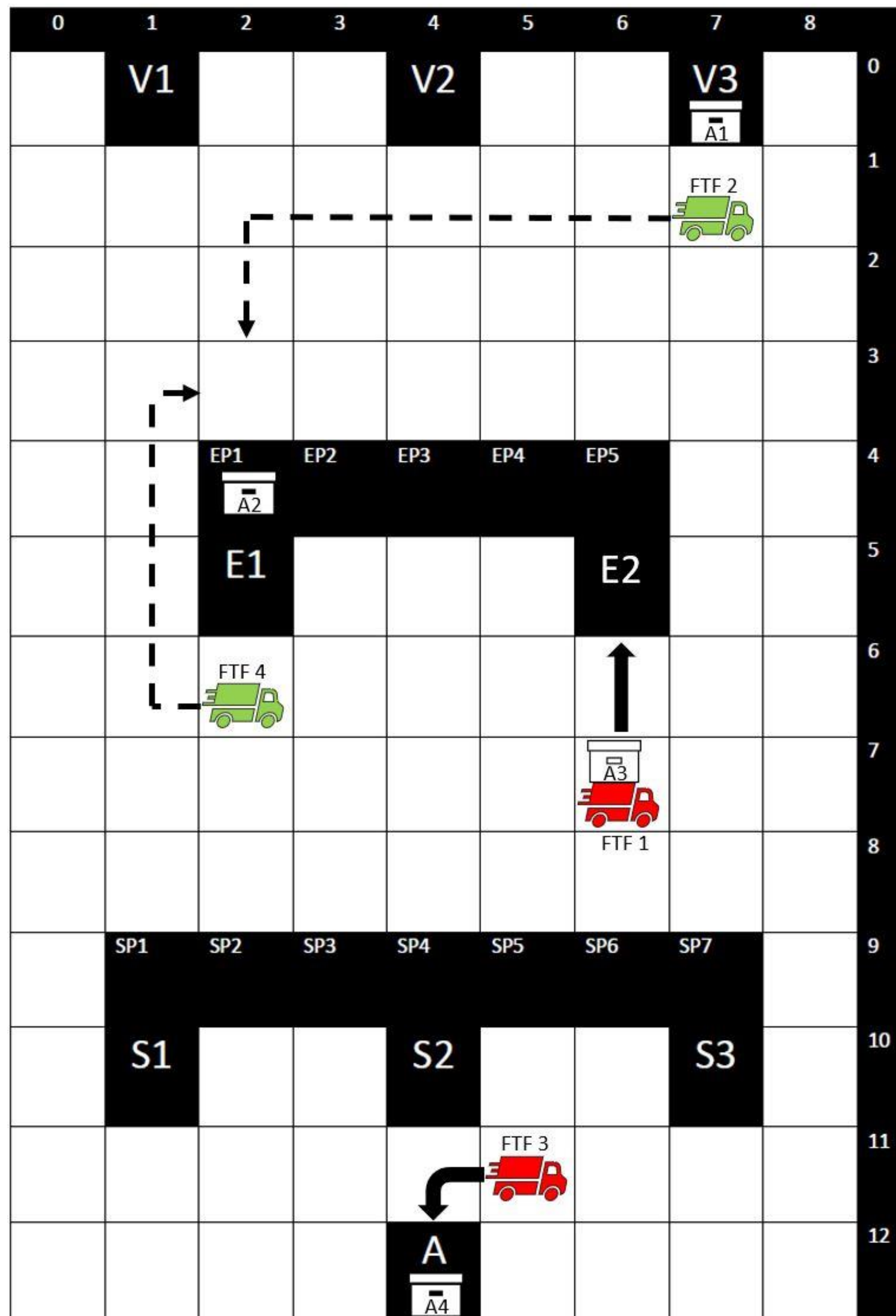


Abbildung 4: Szenario aus dem Planspiel für die Analyse der Verhandlungen (Quelle: eigene Darstellung)

Anzumerken ist, dass die Verhandlungen und dessen Abläufe in diesem Szenario wie im Planspiel in einer Spielrunde stattfinden.

Beginn der Verhandlung:

Kundenauftrag löst die Verhandlung aus:

1. Kundenauftrag A2 wurde teilproduziert und muss für den Weitertransport von einem verfügbaren FTF zu einem der Ausgänge transportiert werden
2. Kundenauftrag A2 befindet sich im Puffer EP1, also auf Position 2:4

Übersicht verschaffen über freie FTFs:

1. FTF 3 ist beschäftigt, weil es zum Kundenauftrag „A4“ fährt, um Ihn aus dem Auftragseingang „A“ abzuholen
2. FTF 1 ist beschäftigt, weil es Kundenauftrag A3 zur Maschine E2 transportiert
3. FTF 2 ist verfügbar
4. FTF 4 ist verfügbar

Vergleich der ZE der verfügbaren FTFs:

1. FTF 2 befindet sich auf dem Raster auf Position 7:1 und muss auf Position 2:3 fahren. Folglich hat FTF 2 einen Fahrweg von 7 ZE.
2. FTF 4 befinden sich auf dem Raster auf Position 2:6 und muss auf Position 2:3 fahren. Folglich hat FTF 4 einen Fahrweg von 5 ZE.
3. FTF 4 bekommt den Kundenauftrag, weil FTF 4 weniger ZE benötigt, um Kundenauftrag A2 abzuholen.
4. Kundenauftrag A2 ist für den Transport zu einem der Ausgänge reserviert.

Diese Liste dient als Basis für den Aufbau der für die Verhandlungen benötigten Funktionen in der Simulation. Zuvor müssen noch ein paar Fälle abgedeckt werden, die eine erfolgreiche Verhandlung ermöglichen. Dabei stellt sich die Frage:
Was muss passieren, wenn…

- … ein Kundenauftrag eingeht und alle Fahrzeuge belegt sind?
- … sich in der Verhandlung herausstellt, dass zwei oder mehrere FTFs eine gleich große ZE haben?

Diese Fälle kommen im Planspiel häufig vor. Für den ersten Fall muss der Kundenauftrag wissen, dass aktuell alle FTFs beschäftigt sind und benötigt eine Funktion, die diesen Fall erkennt und eine neue Transportanfrage nach einer von außen definierbaren Zeit sendet. Im Planspiel geschieht das jede Spielrunde, weil jeder Spieler für seine Kundenaufträge verantwortlich ist und darauf achten muss.

Für den zweiten Fall gibt es im Planspiel keine Regelung. Die Spieler entscheiden per Zufall, welches der FTFs den Zuschlag bekommt und welches nicht. Für die Simulation muss eine Funktion gestaltet werden, die in diesem Fall entscheidet, welches der FTFs den Zuschlag bekommt. Beide Fälle sind in den Anforderungen enthalten und von hoher Priorität.

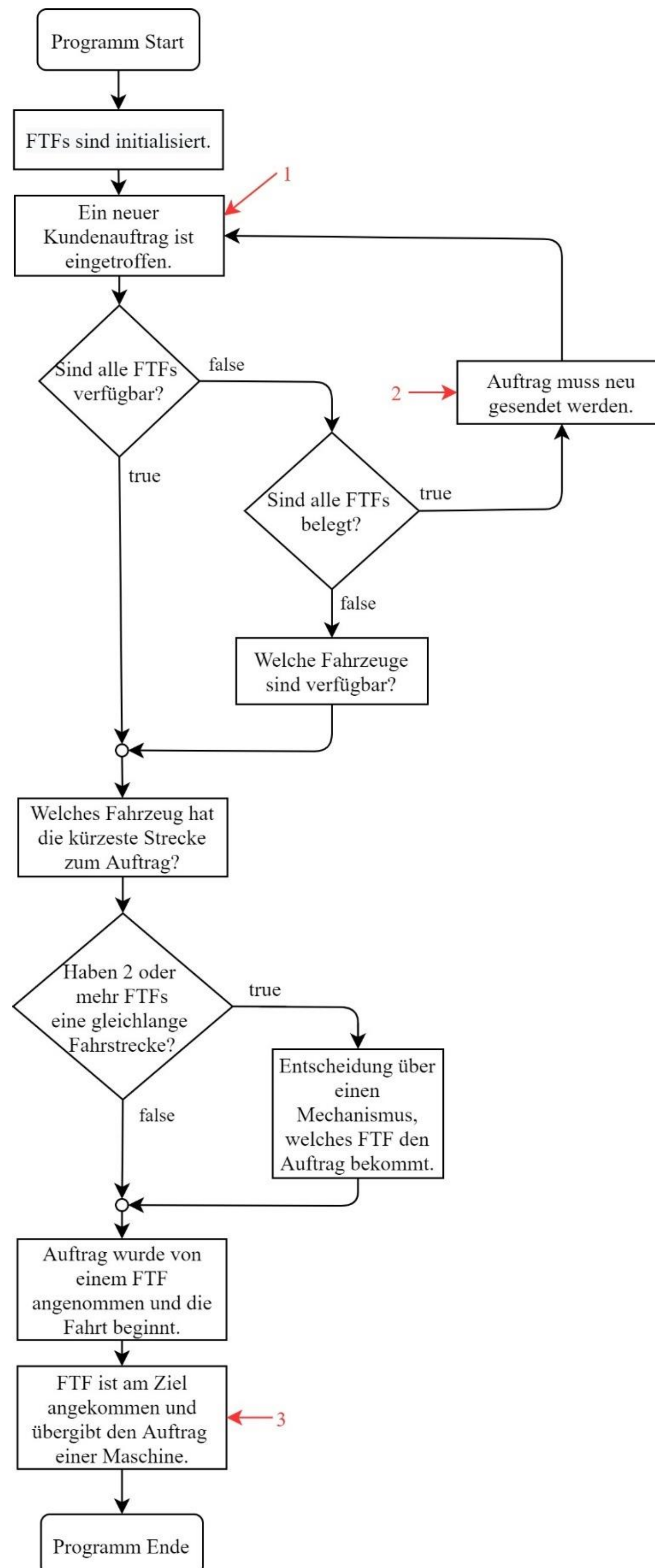


Abbildung 5: PAP der Verhandlungen aus den Daten des Szenarios im Planspiel (Quelle: eigene Darstellung)

Mit den gewonnenen Daten aus dem Szenario lässt sich eine Struktur der Verhandlungen in Form eines PAP entwickeln, der als Basis für den Aufbau der benötigten Kommunikationsstruktur und den damit verbundenen Funktionen dient. In Abbildung 5 ist eine erste Struktur für die Verhandlung dargestellt, um den Erhalt eines Kundenauftrages zu sehen. Anzumerken ist, dass die Kommunikation während der Verhandlung nur zwischen den aktiven und verfügbaren FTFs stattfindet. Das bedeutet, dass die Kommunikation mit den anderen Komponenten, also dem Kundenauftrag und den Maschinen, nur außerhalb der Verhandlungen vorkommt. Am Anfang (Punkt 1) findet eine Kommunikation mit der Komponente ‚Kundenauftrag' statt, in dem die Daten des Kundenauftrages an die FTFs übermittelt werden. Dabei wird als erstes geprüft, ob alle FTFs beschäftigt sind. Haben alle FTFs abgelehnt, wird eine Nachricht (Punkt 2) an den Kundenauftrag gesendet. Dieser stellt so lange eine neue Transportanfrage, bis der Kundenauftrag von einem freien FTF nicht abgelehnt wird. Ist das geglückt, beginnt die Verhandlung ausschließlich zwischen den FTFs. Nachdem ein Kundenauftrag angenommen und letztendlich am Ziel angekommen ist, beginnt eine Kommunikation mit der Komponente ‚Maschine', da das FTF den Kundenauftrag der vorgesehenen Maschinengruppe übergibt (Punkt 3). In den Verhandlungen müssen ebenfalls Daten zwischen den FTFs ausgetauscht werden, wodurch mehrere Funktionen benötigt werden, die die Daten auswerten, eventuell erweitern und versenden. Welche Daten zwischen den Komponenten ausgetauscht werden müssen, wird in folgendem Kapitel erläutert.

4.2.2 Welche Daten werden ausgetauscht?

Im Folgenden wird eine Auflistung von Informationen bereitgestellt, die für die Verhandlung zwischen allen Komponenten relevante Daten beinhaltet. Als Basis für die Erstellung der Datenstruktur dient die Abbildung 5 aus dem vorherigen Kapitel 4.2.1.

- **Initialisierung:** Zu Beginn sowie im Verlauf der Simulation müssen sich die FTFs eigenständig initialisieren, insofern diese in die Produktion benötigt werden. Das bedeutet für die FTFs, dass die FTFs voneinander wissen müssen, um eine erfolgreiche Verhandlung durchführen zu können. Hierfür ist es ausreichend, wenn jedes FTF die ID (Car_ID) (Identifikator) des jeweils aktiven FTF in einer Liste speichert, um im Anschluss mit der ersten Verhandlung um den ersten Kundenauftrag zu beginnen. Die ID sollte aus mindestens drei Zeichen bestehen, damit die Individualität jeder ID sichergestellt ist.
- **Kundenauftrag:** Jede Verhandlung beginnt mit dem Eintreffen eines Kundenauftrages (order) bei allen aktiven FTFs. Dieser enthält Informationen, die den FTFs helfen den Kundenauftrag durch die Produktion zu geleiten. Jeder Kundenauftrag besteht aus einer Struktur, in dem jeweils vier Eigenschaften enthalten sind. Diese Eigenschaften sind wie folgt beschrieben:

1. Eine ID, die aus mindestens drei Zeichen besteht, um so die Einmaligkeit der ID sicherzustellen (Order_ID)
2. Der Auftragseingang A, von dem der Kundenauftrag zu Beginn abgeholt wird (A)
3. Das erste Ziel in der ersten Maschinengruppe (S1, S2 oder S3)
4. Die Maschine, die in der zweiten Maschinengruppe angefahren werden soll (E1 oder E2)

Ein Kundenauftrag ist demnach wie folgt zusammengestellt:

order = {ID, A, (S1, S2 oder S3), (E1 oder E2)}

Daraus ergeben sich sechs mögliche Varianten, die ein Kundenauftrag haben kann. Diese sind:

1. (ID, A, S1, E1)
2. (ID, A, S1, E2)
3. (ID, A, S2, E1)
4. (ID, A, S2, E2)
5. (ID, A, S3, E1)
6. (ID, A, S3, E2)

- **Beginn der Verhandlungen:**
    1. Kundenauftrag wurde von allen FTFs abgelehnt. Folgende Information wird an den Kundenauftrag gesendet (Car_ID, order).
    2. Sind ein oder mehrere FTFs verfügbar, muss der Weg, also die ZE von jedem FTF zum Kundenauftrag, mit allen FTFs, die an der Verhandlung teilnehmen, verglichen werden. Hierfür wird die aktuelle Position des Kundenauftrages und der FTFs benötigt, um daraus eine Strecke zu ermitteln. Jedes FTF sendet seine eigene ID (Car_ID) und die Fahrstrecke (path) an alle FTFs, die an der Verhandlung teilnehmen. Zusammengefasst werden folgende Daten benötigt: (order, Car_ID, path).
    3. Hat sich herausgestellt, dass zwei oder mehr FTFs eine gleich lange Fahrstrecke haben, dann muss entschieden werden, welches der FTFs den Zuschlag bekommt. Hierfür kann eine Random-Funktion benutzt werden, die einen Zufallsgenerator darstellt. Infolgedessen würfeln alle FTFs, die in diesem Teil der Verhandlung gelandet sind und vergleichen die Ergebnisse. Folgende Daten werden für die Ermittlung des Gewinners der Verhandlung benötigt: das Ergebnis der Würfelfunktion jedes Fahrzeugs (dice) und für den Abgleich die Anzahl der FTFs (cars_count).

4. Nachdem das FTF den Kundenauftrag zu einer der Maschinen gefahren hat, benötigt die Maschine die Informationen des Kundenauftrages, um den Kundenauftrag zu aktualisieren, wenn der Kundenauftrag in einen Puffer gelagert wird (order, Car_ID).

Zur besseren Übersicht werden die eben ermittelten Daten in eine kleine Tabelle zusammengefügt, aus der zu entnehmen ist, welche Daten von welcher Komponente gesendet wurden und an welche Komponente diese gerichtet ist.

Daten	Herkunft	Ziel
Car_ID	FTF	FTF
order	Auftrag	FTF
Car_ID, order	FTF	Auftrag
order, car_ID, path	FTF	FTF
dice, cars_count	FTF	FTF
order	FTF	Maschine

Tabelle 1: Zusammenfassung des theoretischen Datenaustausches

Nachdem das Verhandlungsmuster aus dem Planspiel analysiert wurde und daraus ein erster Prototyp der Kommunikationsstruktur entwickelt worden ist, wird im nächsten Kapitel erklärt, wie die Daten während der Kommunikation zwischen den FTFs ausgetauscht werden.

4.2.3 Datenaustausch mit Hilfe von MQTT

Im Vergleich zum Planspiel, welches rundenbasiert abläuft und jeder Schritt bzw. jede Aktion in Ruhe ausgeführt werden kann, werden die Verhandlungen in der Simulation in Echtzeit abgewickelt. Um dies zu ermöglichen, wird ein reaktionsschnelles Netzwerk benötigt. Wie in Kapitel 2.3 erwähnt, eignet sich MQTT dank seiner hochfrequenten Kommunikation für die Realisierung der Verhandlungen besonders gut. Hervorzuheben ist, dass ein MQTT-Broker keine Steuerungsfunktionen in Bezug auf den Verlauf eines Programmes hat. Der Broker empfängt Daten und leitet diese weiter. Hierdurch ist er für eine dezentrale Steuerung geeignet. Nachdem in Kapitel 4.2.1 die Struktur der Verhandlungen im Planspiel analysiert wurde, werden die Ergebnisse erweitert und in einem weiteren Szenario anschaulich vorgestellt. In Abbildung 6 wird ein Beispiel für eine Kommunikation über MQTT präsentiert, in dem ein neues Transportfahrzeug FTF 4 in die Simulation initialisiert wird. Folglich ist anzunehmen, dass FTF 1, FTF 2

und FTF 3 einander bereits kennen. Des Weiteren sind zwei Topics (Topic 1, Topic 2) abgebildet, worüber die FTFs Informationen untereinander austauschen. Voraussetzung für die Kommunikation miteinander ist, dass alle FTFs die beiden Topics subskribiert haben und in die Topics publizieren können.

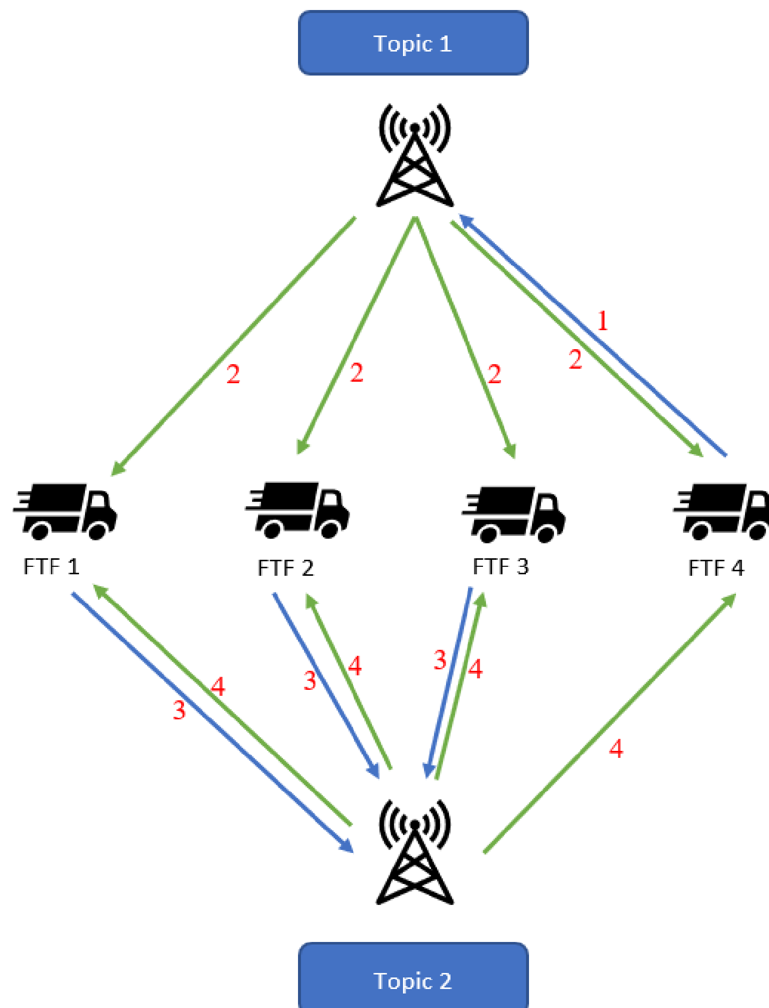


Abbildung 6: Beispiel einer Kommunikation mit Hilfe von MQTT
(Quelle: eigene Darstellung)

Anzumerken ist, dass jedes Topic nur für seine vorgesehene Funktion zur Verfügung steht, wodurch ein geregelter Datenaustausch gewährleistet ist.

Beginn der Initialisierung:

---

1. FTF 4 publiziert seine eigene ID (FTF_4) an das Topic 1 (Schritt 1)
2. Der MQTT-Broker publiziert die ID von FTF_4 weiter an alle FTFs (Schritt 2)

3. Die FTFs vergleichen die ID von FTF 4 mit einer Fahrzeugliste, in der alle anderen IDs der FTFs gespeichert sind. Ist die ID nicht dabei, wird diese in die Liste aufgenommen
4. FTF 1, FTF 2 und FTF 3 publizieren anschließend die eigene ID (FTF_1, FTF_2 und FTF_3) an das Topic2 (Schritt 3)
5. Der MQTT-Broker publiziert diese weiter an alle anderen FTFs (Schritt 4)
6. Die FTFs vergleichen die IDs mit einer Fahrzeugliste, in der alle anderen FTFs gespeichert sind. Ist eine der IDs nicht dabei, wird diese in die eigene Fahrzeugliste aufgenommen. Die Initialisierung ist abgeschlossen

---

An diesem Beispiel ist zu erkennen, wie die Kommunikation der Initialisierung aufgebaut ist. Dabei macht es keinen Unterschied, ob Daten während der Verhandlung, also nur zwischen den FTFs, oder Daten außerhalb der Verhandlungen, also zwischen den unterschiedlichen Komponenten, ausgetauscht werden. In jeder Komponente werden immer nur die Funktionen ausgelöst, in die publiziert wurde. Um eine Endlosschleife zu vermeiden, wurde bewusst auf zwei Topics und zwei Funktionen gesetzt.

4.3 Funktionen und deren Struktur

Durch das Analysieren der einzelnen Schritte in den vorangegangenen Kapiteln kann mit der Planung der Funktionen begonnen werden. Im Folgenden wird das Konzept der Initialisierung anhand eines PAPs vorgestellt.

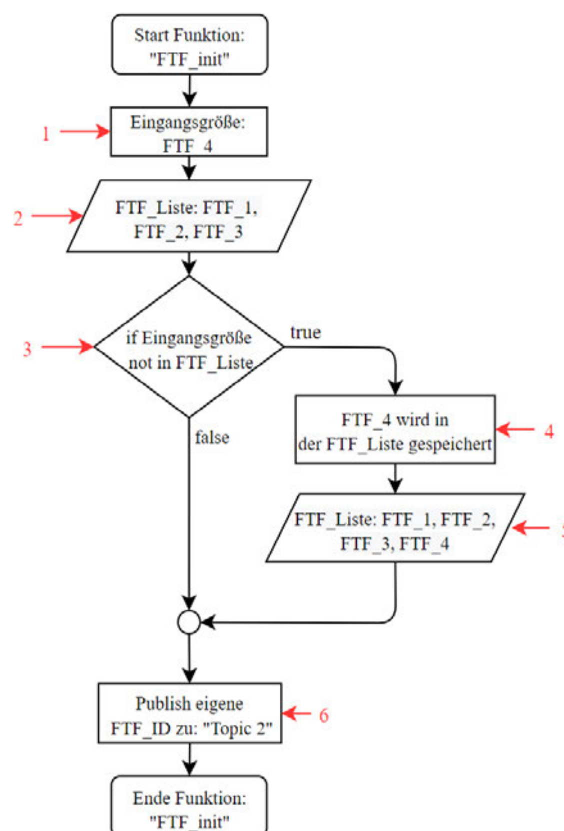


Abbildung 7: Programablaufplan der Funktion „FTF-init“ (Quelle: eigene Darstellung)

Als Vorlage wird die Auflistung der einzelnen Schritte der Initialisierung aus Kapitel 4.2.3 verwendet. Damit die Daten gespeichert werden, die von einem neuen FTF publiziert und anschließend vom Broker weitergeleitet werden, bedarf es in diesem Fall zwei Funktionen für die Initialisierung. Hierfür ist in Abbildung 7 die erste Funktion als PAP zu sehen, die von der Nachricht aus Topic 1 ausgelöst wird. Hierbei handelt es sich um einen Pseudo-Code, der am umgesetzten Modell der Initialisierung angelehnt ist. Wie in Abbildung 6 bereits dargestellt, erfolgt die Datenübertragung über zwei Topics.

Wie bereits erwähnt, wird die Funktion „FTF_init“ durch eine Nachricht mit dem Inhalt FTF_4 ausgelöst und beginnt den Programmcode durchzulaufen. Anzumerken ist, dass jedes Fahrzeug die gleichen Funktionen hat. Folglich spielt es keine Rolle, welchem FTF, also FTF_1, FTF_2 oder FTF_3 die hier abgebildete Funktion zuzuordnen ist, weil alle den gleichen Inhalt in der Fahrzeugliste haben.

Funktion „FTF_init“:

Schritt:

1. Im ersten Schritt ist die Eingangsgröße (FTF_4), also die Nachricht, die angekommen ist, enthalten.
2. Hier wird eine Ausgabe gemacht. In diesen Fall ist das die aktuelle Fahrzeugliste (FTF_Liste). Diese beinhaltet drei FTFs, FTF_1, FTF_2 und FTF_3.
3. In diesem Schritt wird durch eine „if“ Anweisung abgefragt, ob sich FTF_4 nicht in der Fahrzeugliste (FTF_Liste) befindet. In diesem Fall ist das wahr (true).
4. Hier wird die ID FTF_4 in die Fahrzeugliste zu den anderen IDs aufgenommen und gespeichert.
5. Es wird eine Ausgabe der Fahrzeugliste (FTF_Liste) gemacht, in der alle vier IDs der vier FTF vorhanden sind.
6. Im letzten Schritt wird dann die eigene ID an das Topic 2 publiziert.

Nachdem die Funktion „FTF_init“ durchgelaufen ist und FTF_4 gespeichert wurde, wird die zweite Funktion „FTF_init_Antwort“ durch das Publizieren ausgelöst. In Abbildung 8 ist die zweite Funktion des Initialisierungsprozesses dargestellt. Mit dieser Funktion endet der Initialisierungsprozess, weil aus der Funktion nichts publiziert wird.

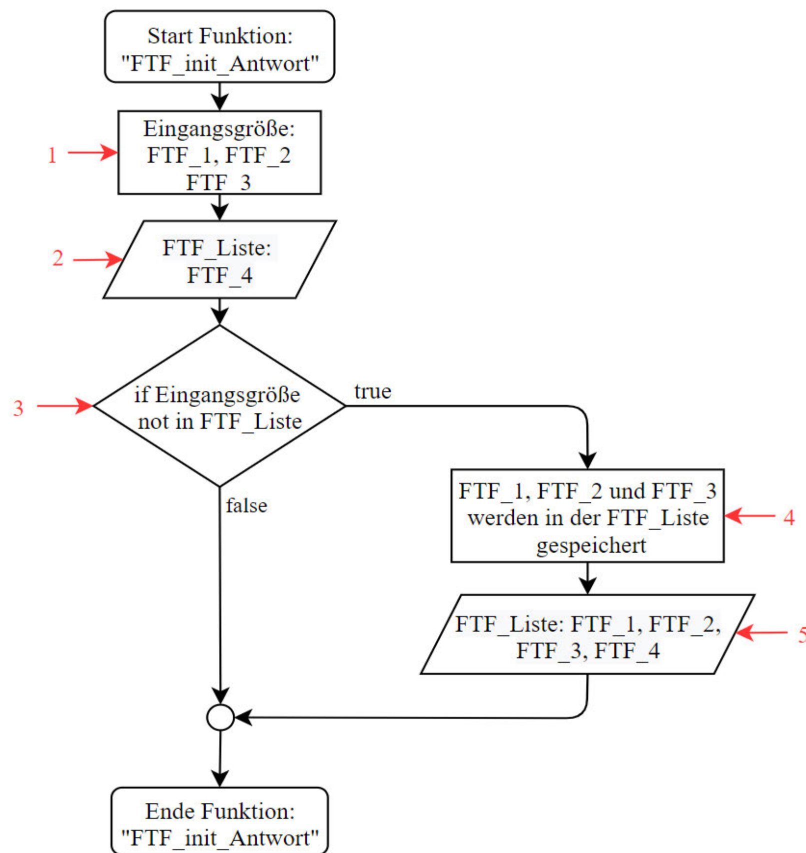


Abbildung 8: Programmablaufplan der Funktion „FTF_init_Antwort“
(Quelle: eigene Darstellung)

Funktion „FTF_init_Antwort“:

Schritt:

1. Im ersten Schritt der Funktion ist die Eingangsgröße (FTF_1, FTF_2, FTF_3) der anderen FTFs zu sehen.
2. Hier wird eine Ausgabe gemacht. In diesen Fall ist das die aktuelle Fahrzeugliste (FTF_Liste) mit dem Inhalt der vorhandenen FTFs (FTF_1, FTF_2 und FTF_3).
3. In diesem Schritt wird durch eine „if“ Anweisung abgefragt, ob sich FTF_1, FTF_2 und FTF_3 nicht in der Fahrzeugliste (FTF_Liste) befinden. In diesem Fall ist das wahr (true).
4. Hier werden die IDs der FTFs neu hinzugekommenen FTFs (FTF_1, FTF_2 und FTF_3) in die Fahrzeugliste zu den andere IDs aufgenommen und gespeichert.
5. Es wird eine Ausgabe der Fahrzeugliste (FTF_Liste) gemacht, in der alle vier IDs der vier FTF vorhanden sind.

Sobald ein neues FTF vom Anwender in der Simulation aktiviert wird, beginnt der Initialisierungsprozess automatisch. Vorher wird eine ID über einen ID-Generator generiert. Wie zu erkennen ist, werden zwei Funktionen benötigt, um den Prozess der Initialisierung durchführen

zu können. Dies garantiert einen überschaubaren Datenfluss. Zusätzlich wird eine Endlosschleife vermieden, die entsteht, wenn die Funktion „FTF_init_Antwort“ nicht existiert. In diesem Fall würde die Funktion „FTF_init“ durch das Publizieren einer Nachricht im sechsten Schritt immer wieder in sich selbst publizieren. Eine Initialisierung kann auch dann durchgeführt werden, wenn ein oder mehrere FTFs beschäftigt sind.

Aus den vorhergegangenen Abschnitten ist zu entnehmen, wie die Herangehensweise zur Gestaltung einer Funktion in dieser Arbeit umgesetzt wird.

4.4 Darstellung der vollständigen Kommunikationsarchitektur

Aufbauend auf den Erkenntnissen der vorhergegangenen Abschnitte kann mit der Entwicklung des Quellcodes der zur Realisierung der Simulation benötigten Funktionen begonnen werden. In diesem Kapitel wird das Konzept der Verhandlungen zwischen den Komponenten, das umgesetzt wurde, in einem Diagramm dargestellt.

Auf Abbildung 9, Abbildung 10 und Abbildung 11 ist eine Swimlane abgebildet, die den kompletten Ablauf der Verhandlungen und die am relevantesten verbundenen Funktionen der Simulation von der Initialisierung eines FTF bis zum Transport eines Kundenauftrages in einen der Ausgänge abbildet. Jeder Kundenauftrag muss diesen Zyklus durchlaufen, um am Ende als fertig produziert zu gelten. In der Kopfzeile sind die einzelnen Komponenten FTF (mqtt-car), Kundenauftrag (mqtt-order) und Maschinen (mqtt-station) zu erkennen. Zudem ist in der Kopfzeile ein Broker zu sehen, der zur besseren Übersicht in zwei geteilt wird (mqtt-Broker). Jede Komponente hat eine eigene Spalte, in der die Funktionen abgebildet sind, deren Inhalt nur für die jeweilige Komponente bestimmt ist. Die Spalte der Broker enthält alle für die Verhandlung um einen Kundenauftrag relevanten Topics, die im Anschluss zur besseren Übersicht in Tabelle 2 zusammengefasst sind. In den Topics sind zudem Informationen zu sehen, die gesendet werden. In der Legende sind Pfeile zu erkennen, die den kompletten Datenfluss darstellen. Die blauen Pfeile stellen das Publizieren einer Komponente dar. Die grünen Pfeile sagen hingegen aus, dass der Broker aus einem Topic an eine Komponente eine Nachricht publiziert. Die schwarzen Pfeile dienen der Darstellung einer Funktion, die innerhalb einer Funktion aufgerufen wird. Jede Funktion ist nummeriert und folgt einer Reihenfolge, die strikt eingehalten wird. Im Folgenden ist eine Liste der Ereignisse aufgeführt, die zusammenfassend jeden Schritt erklärt. Anzumerken ist, dass die Liste bis Schritt 5 erläutert wird, weil nachfolgend derselbe Rhythmus stattfindet.

Beginn der Simulation:

---

1. Beginn der Initialisierung der FTFs. Die ID eines FTFs wird generiert und an die anderen FTFs gesendet.
   1.1. Die ID des neu initialisierten FTFs wird gespeichert und die ID der vorhandenen FTFs wird gesendet.
   1.2. Die ID der eingegangenen FTFs wird gespeichert. Der Initialisierungsprozess ist abgeschlossen.
2. Der Kundenauftrag sendet seine Informationen an die verfügbaren FTFs über das Topic „new-order“.
   2.1. Die verfügbaren FTFs erhalten einen neuen Kundenauftrag und die Funktion „on_new_order“ wird ausgelöst. Der Kundenauftrag wird hierbei gespeichert. Die strichpunktlinienförmige Umrandung hat die Bedeutung, dass die „Is not busy?“ Anweisung und der Schritt 2.3 in der Funktion „on_new_order“ ausgeführt werden. Die „Is not busy?“ Anweisung fragt, ob ein FTF nicht beschäftigt ist. Trifft das zu, wird die Funktion „share_my_path“ aufgerufen. Ist die Anweisung falsch und das FTF ist beschäftigt, wird an die Funktion „on_order_refused“ über das Topic „order-refused“ eine Nachricht gesendet.
   2.2. Wenn während einer Verhandlung eines der FTFs beschäftigt ist, wird eine Nachricht an diese Funktion gesendet. Diese Funktion sammelt Informationen über jeden Kundenauftrag und jedes FTF, das einen Kundenauftrag abgelehnt hat. Haben alle FTFs einen Kundenauftrag abgelehnt, so sendet die Funktion eine Nachricht an die Funktion „on_order_lost“ über das Topic „order-lost“. Das bedeutet, dass ein Kundenauftrag verlorengegangen ist und dass der Kundenauftrag eine erneute Transportanfrage an die FTFs senden muss.
      2.2.1. Nachdem alle FTFs einen Kundenauftrag abgelehnt haben und die Nachricht angekommen ist, vergleicht jeder Kundenauftrag die Informationen mit den eigenen Informationen, mit dem Ziel, herauszufinden, ob es sich um den eigenen Kundenauftrag handelt. Wurde der Kundenauftrag identifiziert, sendet der abgelehnte Kundenauftrag eine neue Transportanfrage an die FTFs.
   2.3. Wenn die Bedingung „true“ ist, wird mit der Funktion „share_my_path“ über das Topic „car-order-path“ der Weg, der zuvor berechnet wurde, an alle verfügbaren FTFs gesendet.
   2.4. In dieser Funktion werden alle notwendigen Informationen über jedes FTF zusammengetragen, die an der aktuellen Verhandlung teilnehmen. Anschließend werden die Rasterpunkte aller FTFs miteinander verglichen, um herauszufinden, welches FTF die kürzeste Fahrstrecke zum Kundenauftrag hat. Sobald bekannt ist welches der FTFs die kürzeste Fahrstrecke zum Kundenauftrag hat, bekommt dieses den Zuschlag und sendet

über das Topic „car-order-taken“ eine Nachricht an die Funktion „on_car_order_taken“. Haben zwei oder mehr als zwei FTFs die gleiche Anzahl an Rasterpunkten zum Kundenauftrag, so muss durch einen Zufallsgenerator ein Gewinner der Verhandlungen ermittelt werden. Infolgedessen wird zuerst gewürfelt und das Ergebnis anschließend über das Topic „diceroll“ an die Funktion “on_diceroll“ gesendet.

2.5. In dieser Funktion werden alle Würfelergebnisse miteinander verglichen und der Gewinner bekommt den Zuschlag für den Kundenauftrag. Infolgedessen sendet das FTF, das das Würfeln gewonnen hat, eine Nachricht über das Topic „car-order-taken“ an die Funktion „on_car_order_taken“.

2.6. Hat sich ein Gewinner aus den Verhandlungen herausgestellt, wird in der Funktion „on_car_order_taken“ überprüft, ob der Kundenauftrag aus dem Auftragseingang abgeholt werden muss oder ob der Kundenauftrag sich in einem Puffer befindet. In diesem Fall befindet sich der Kundenauftrag im Auftragseingang und die Funktion „drive_order“ wird aufgerufen.

3. Ab diesem Schritt beginnt das FTF die Fahrt.

3.1. Zuerst fährt das FTF zum Auftragseingang und holt den Kundenauftrag ab.

3.2. Anschließend transportiert das FTF den Kundenauftrag zu einer der Maschinen in der ersten Maschinengruppe. Sobald es angekommen ist, sendet das FTF der Maschine über das Topic „order-delivered“ die notwendigen Informationen und hat den Kundenauftrag der Maschine übergeben.

4. Nachdem die Maschine den Kundenauftrag übernommen hat, wird überprüft, welchen Produktionsstatus der Kundenauftrag hat. Da sich der Kundenauftrag in der ersten Maschinengruppe befindet, beginnt die Produktion und die Maschine ruft die Funktion „working“ auf.

4.1. In der Funktion „working“ wird die Bearbeitung des Kundenauftrages simuliert. Sobald der Kundenauftrag bearbeitet wurde, wird die Funktion „move_to_buffer“ aufgerufen.

4.2. Die Funktion „move_to_buffer“ entscheidet, auf welchem Pufferplatz der Kundenauftrag abgestellt wird. Anschließend wird über das Topic „order-buffer-ready“ eine Nachricht an den Kundenauftrag gesendet.

5. Die Funktion „on_order_buffer_ready“ wird erst dann angesprochen, wenn der Kundenauftrag in einer der beiden Maschinengruppen bearbeitet wird. Indem der Kundenauftrag eine Nachricht über das Topic „new-order-buffer“ an die FTFs sendet, beginnt der Prozess der Verhandlung von Anfang an.

Bei der Entwicklung des Algorithmus ist zu beachten, dass ein Test der kompletten Simulation nach jeder Änderung im Quellcode einer Funktion stattfinden muss.

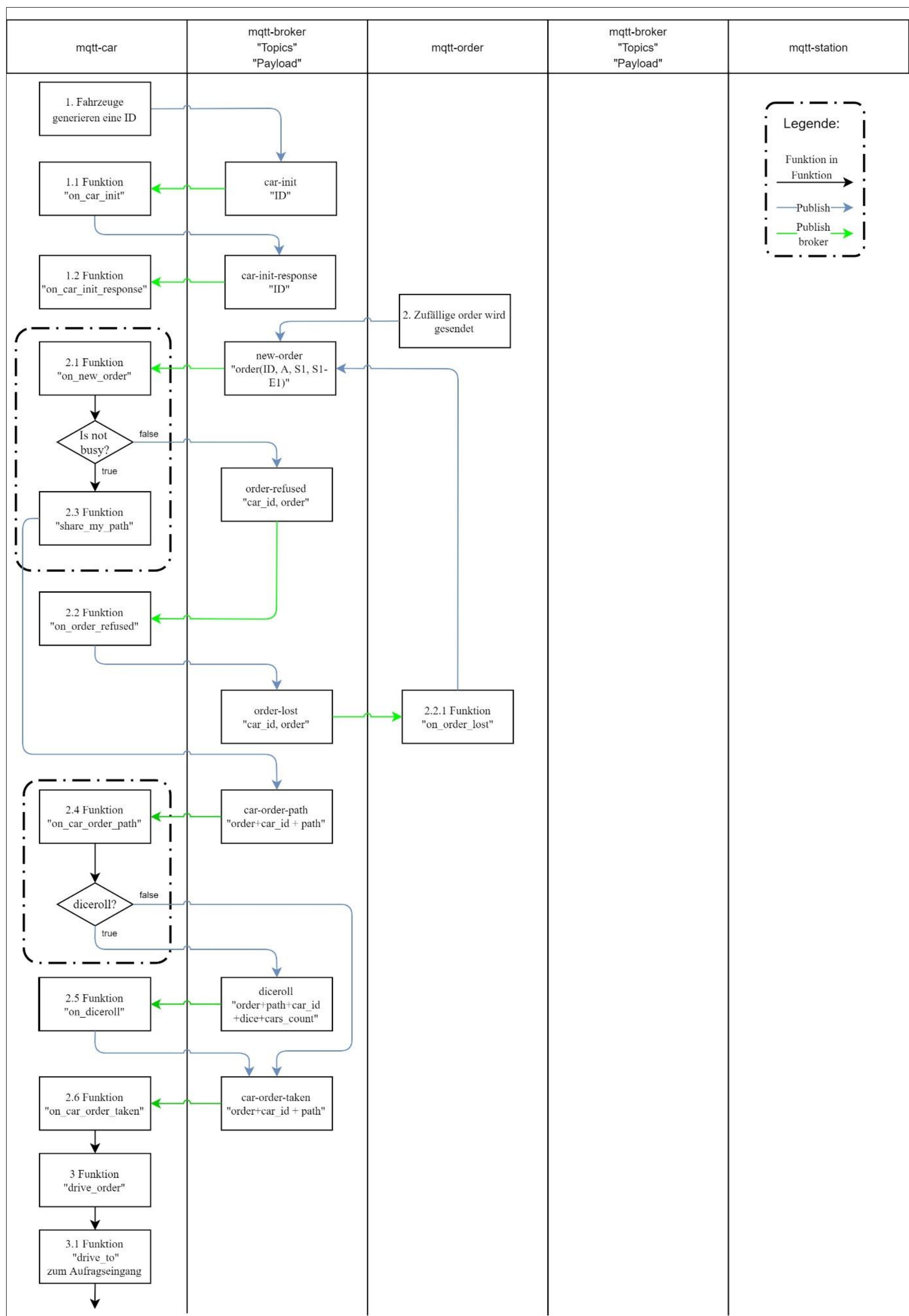


Abbildung 9: Vollständiges Konzept der Kommunikationsstruktur in der Simulation (Teil 1) (Quelle: eigene Darstellung)

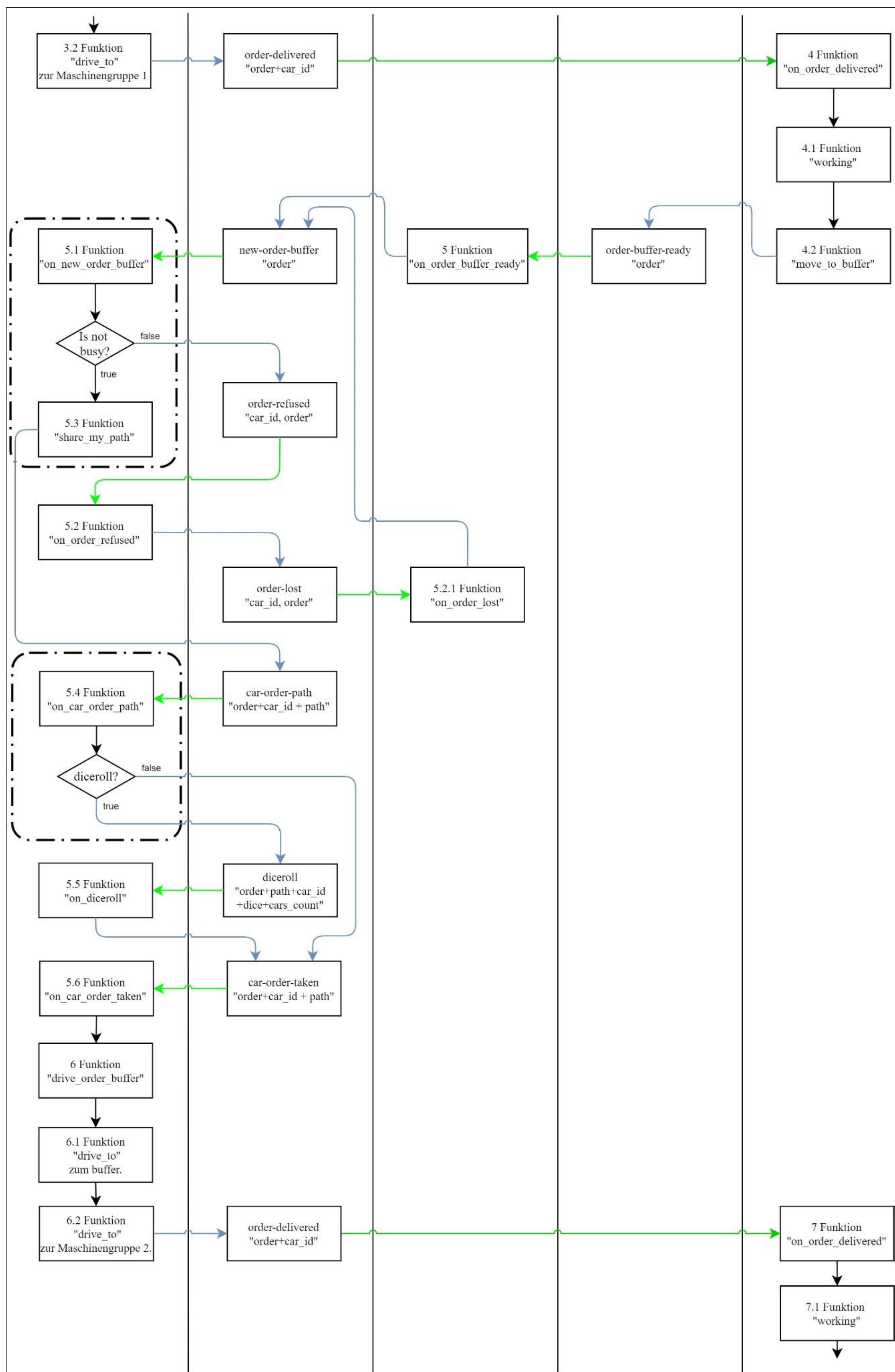


Abbildung 10: Vollständiges Konzept der Kommunikationsstruktur in der Simulation (Teil 2) (Quelle: eigene Darstellung)

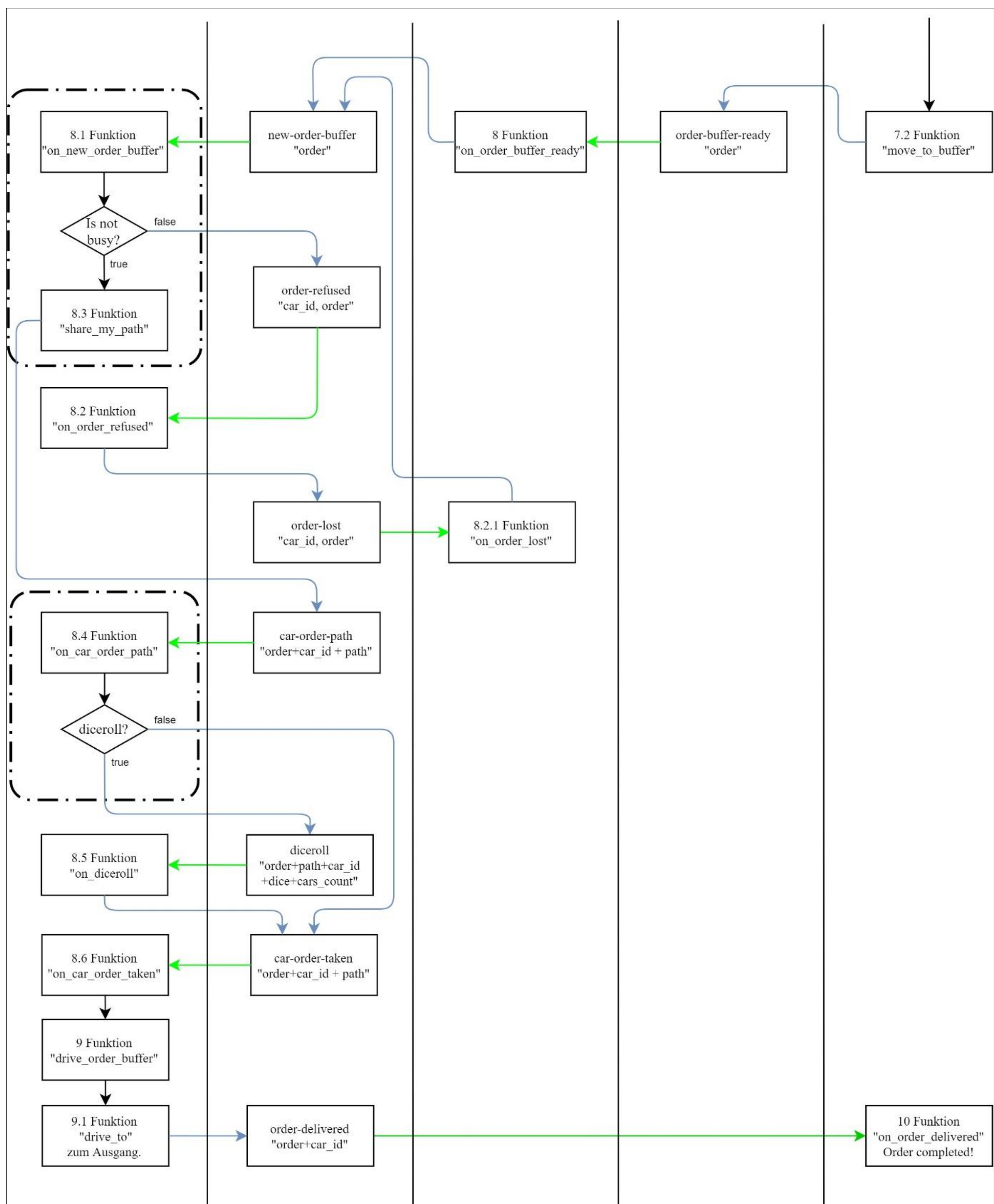


Abbildung 11: Vollständiges Konzept der Kommunikationsstruktur in der Simulation (Teil 3) (Quelle: eigene Darstellung)

Wie in der Swimlane abzulesen ist, sind die Bezeichnungen der Topics und der Funktionen ähnlich gestaltet, wodurch eine erleichterte Interpretation der Kommunikationsstruktur erzielt werden soll. Dieses Schema zieht sich durch den kompletten Interaktionsverlauf in der Simulation.

In Tabelle 2 befindet sich eine Zusammenfassung der Swimlane, in der zu erkennen ist, welche Komponente mit wem über welchen Topic kommuniziert. Anzumerken ist, dass der Schwerpunkt dieser Arbeit auf den Verhandlungen zwischen den FTFs beruht, was in dieser Tabelle zu erkennen ist.

Wer kommuniziert mit wem?							
	Topic	mqtt-car		mqtt-order		mqtt-station	
		Publish	Subscribe	Publish	Subscribe	Publish	Subscribe
1	car-init	✓	✓	-	-	-	-
2	car-init-response	✓	✓	-	-	-	-
3	new-order	-	✓	✓	-	-	-
4	order-refused	✓	✓	-	-	-	-
5	order-lost	✓	-	-	✓	-	-
6	car-order-path	✓	✓	-	-	-	-
7	diceroll	✓	✓	-	-	-	-
8	car-order-taken	✓	✓	-	-	-	-
9	order-delivered	✓	-	-	-	-	✓
10	order-buffer-ready	-	-	-	✓	✓	-

Tabelle 2: Interaktion aller Komponenten über die wichtigsten Topics

5 Implementierung der Simulation

Das Ziel der Implementierung ist es, die notwendigen Werkzeuge vorzustellen, die eine Realisierung der Simulation ermöglichen. Danach folgen bedeutende Funktionen in der Maschine und des Kundenauftrages, die erläutert werden. Anschließend wird die Wegfindung der FTFs und die Verbindung zu einem MQTT-Broker erklärt. Zuvor werden noch die Anforderungen an die grafische Oberfläche und die Tools zur Realisierung der Simulation vorgestellt.

5.1 Grafische Oberfläche und Tools zur Realisierung der Simulation

Zur Realisierung der Verhandlungen zwischen den FTFs wurde zu Beginn des Projekts mit mehreren Arduinos experimentiert. Hierbei hat sich herausgestellt, dass die Fehlersuche beim Entwickeln des Quellcodes zeitaufwändig ist, weil das Hochladen des Quellcodes von der integrierten Entwicklungsumgebung (IDE) auf einen Arduino nach jeder Veränderung im Quellcode neu durchgeführt werden muss. Deshalb wurde eine Plattform gesucht, die es ermöglicht, gleich mehrere Schritte zu überspringen und den Aufwand neben dem Entwickeln des Quellcodes zu reduzieren. Das heißt, dass alle FTFs, Maschinen und die Kundenaufträge möglichst auf einem Bildschirm abgebildet sein sollen. Die Plattform sollte möglichst auf einem vorhandenen Windows-Betriebssystem lauffähig sein, damit es zum einen ressourcensparend ist, also keine zweite Partition auf der Festplatte erstellt werden muss, zum anderen soll es leicht sein, die Plattform zu vervielfachen. Das bedeutet, dass die Plattform mit wenig Aufwand durch Benutzung eines Massenspeichers auf ein anderes Windows-Betriebssystem transferiert wird und sofort einsatzbereit ist. In den folgenden Abschnitten werden Programme vorgestellt, die zur Realisierung der Anforderungen benötigt werden.

5.1.1 VMware Workstation Player

Um die Anforderungen aus Kapitel 5.1 zu realisieren, wird als erstes eine „virtuelle Maschine“ (VM) benötigt. Eine VM ist ein abstraktes Rechnersystem, das auf einem realen Rechnersystemen simuliert wird (vgl. Kenngott 2001, S. 11).

Mit der Software VMware Workstation 16 Player lassen sich virtuelle Maschinen erstellen, die anschließend konfiguriert und auf dem Desktop ausgeführt werden können. Hierfür bietet die Software eine intuitive Benutzeroberfläche zum Erstellen von VMs oder zum Ausführen vorkonfigurierter VMs an, die mit VMware Workstation erstellt wurden (Getting Started Guide 2009, S. 7). Abbildung 12 zeigt einen Screenshot der VMware Workstation 16 Player.

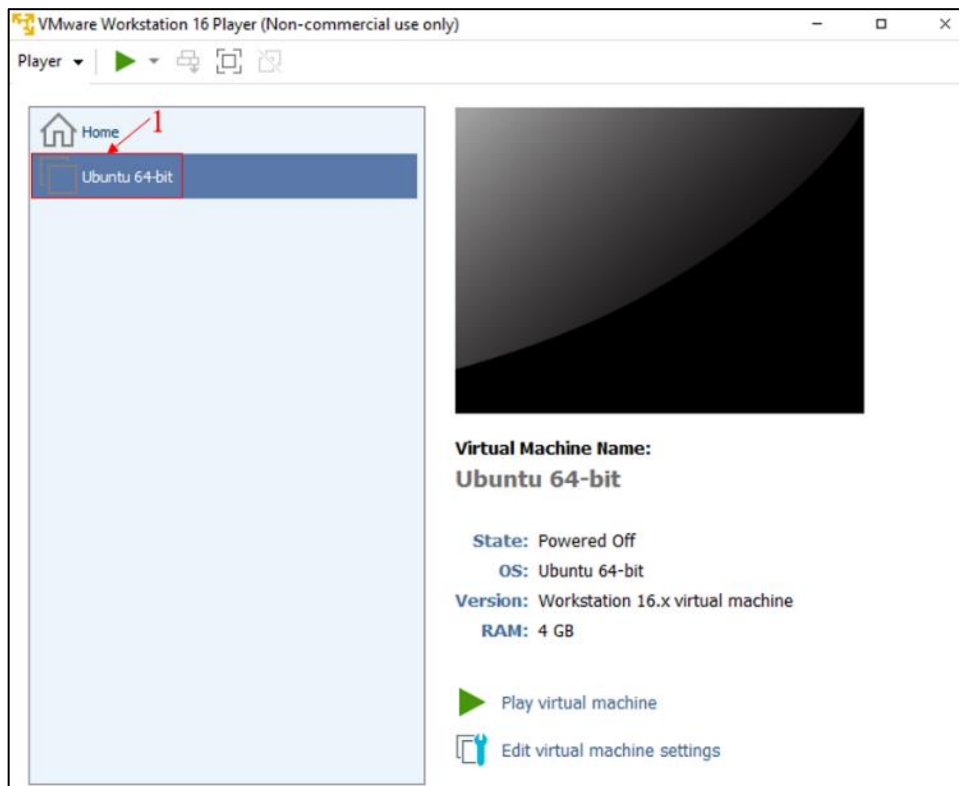


Abbildung 12: Screenshot: VMware Workstation 16 Player mit der Linux Distribution Ubuntu (Quelle: aus dem Programm VMware Workstation 16 Player)

Nachdem VMware installiert wurde, wird im weiteren Verlauf ein Betriebssystem benötigt. Wie in der Abbildung 12 Punkt 1 zu erkennen ist, ist das Betriebssystem Ubuntu installiert. Ubuntu ist eine Linux-Distribution, die im wissenschaftlichen Bereich weit verbreitet ist (vgl. Erben 2017, S. 21).

5.1.2 Texteditor Sublime Text

Das Programmieren eines Codes erfordert Zeit und Sorgfalt. Folglich ist es von Vorteil einen Texteditor zu verwenden, der das Lesen und das Bearbeiten eines Codes erleichtert, indem er die Struktur hervorhebt oder eine automatische Einrückung beherrscht. Ebenso sind sowohl Tastaturkürzel für mehrmals vorkommende Vorgänge als auch ein Marker für die Zeilenlänge von Vorteil (vgl. Matthes 2017, S. 563).

Sublime Text ist ein Texteditor, der es erlaubt, alle geschriebenen Programme direkt im Editor anstatt in einer Konsole (Terminal) auszuführen. Zudem wird eine Syntaxkennzeichnung angeboten, wodurch die einzelnen Codebestandteile wie z.B. „Funktionen“ oder „if“ Anweisungen farbig voneinander abgesetzt werden (vgl. Matthes 2017, S. 10). Wenn Benutzereingaben in Sublime Text entgegengenommen werden sollen, ist es nicht möglich, diese im Editor auszuführen. Hierfür muss ein Terminal verwendet werden (vgl. Matthes 2020, S. 130).

In Abbildung 13 ist ein Screenshot aus Sublime Text abgebildet, in dem drei Fenster mit jeweils einem Quellcode zu erkennen sind. Der Vorteil von drei parallel geöffneten Fenstern ermöglicht eine gute Übersicht über die Quellcodes, die miteinander interagieren.

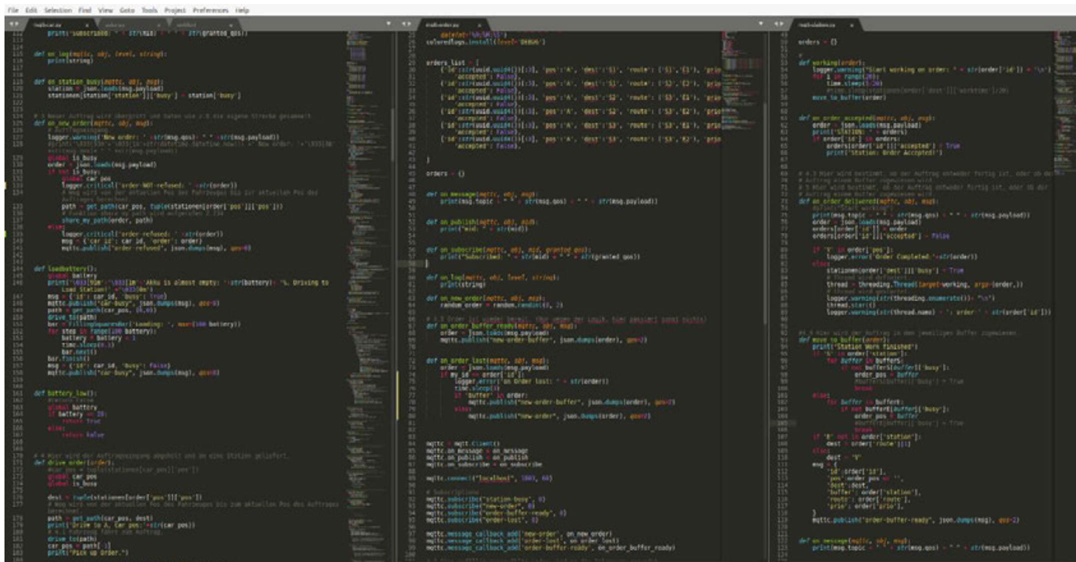


Abbildung 13: Screenshot: Sublime Text mit drei parallel geöffneten Programmen in einem Fenster (Quelle: eigene Darstellung)

Um den Quellcode zu testen, wird zusätzlich ein Terminalemulator verwendet, in dem alle Aktivitäten der Komponenten zu sehen sind.

5.1.3 Terminalemulator Terminator

Jede Linux-Distribution hat von Anfang an einen eigenen Terminalemulator, der sich von den Funktionen kaum unterscheidet.

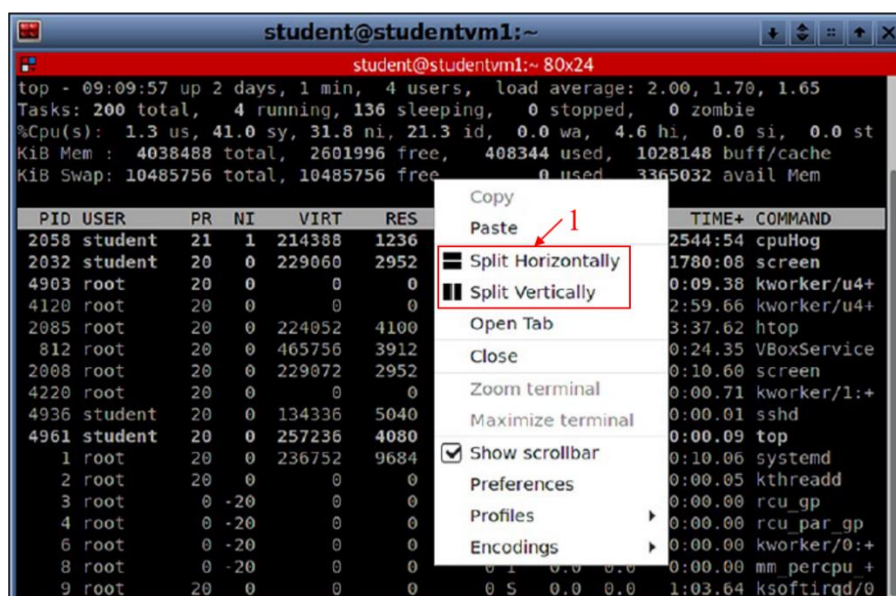


Abbildung 14: Terminal im Terminator teilen (Both 2020, S. 412)

Der Terminalemulator Terminator hingegen hat neben den Standardfunktionen eines Terminals einen entscheidenden Vorteil gegenüber einem Standard-Terminal. Durch einen Rechtsklick in den Terminal lassen sich beliebig viele Fenster öffnen, die jeweils einen eigenständigen Terminal darstellen (vgl. Both 2020, S. 412). In Abbildung 14 ist der Terminator abgebildet und das Menü, das durch Rechtsklick geöffnet wurde. Im Menü besteht die Auswahl, den Terminal entweder horizontal oder vertikal zu teilen. In Abbildung 15 ist zu sehen, dass sechs Terminals im Terminator geöffnet sind.

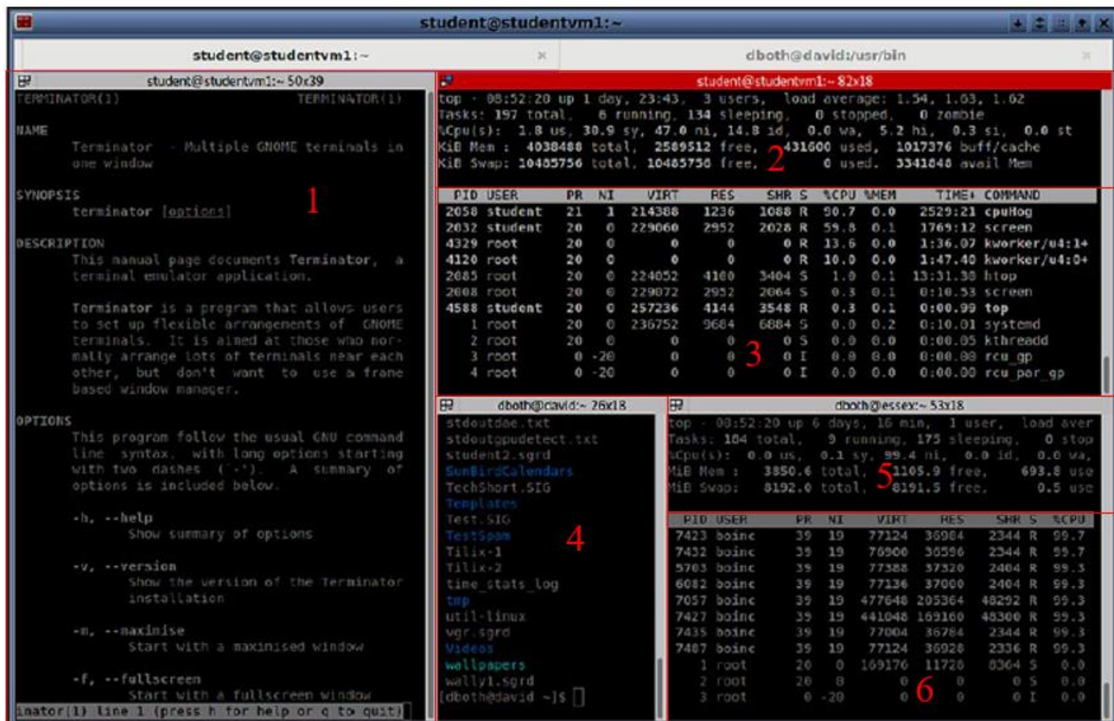


Abbildung 15: Mehrere Terminals im Terminator geöffnet (Both 2020, S. 414)

Jeder einzelne Terminal im Terminator kann wie ein vollständiger Terminal unabhängig von den anderen operieren. Der Terminator wird in dieser Arbeit genutzt, weil alle Komponenten auf einen Blick zu sehen sind. Hierdurch kann die vollständige Simulation an nur einem Bildschirm beobachtet werden. In Kapitel 6, in dem der praktische Teil der Arbeit vorgestellt wird, wird der Terminator und das Zusammenwirken aller Komponenten dargestellt.

5.1.4 Zusammenschluss der Tools

Im Folgenden wird beschrieben, wie die Software zusammenhängt und welche Software wann gestartet wird. In Abbildung 16 ist ein Verteilungsdiagramm zu sehen, das den Zusammenhang der Tools darstellt. Im Diagramm ist zu erkennen, dass die vollständige Software auf einem Windows-Rechner operiert. Das bedeutet, dass VMware auf einer Windows-Partition installiert

werden muss. Anschließend kann ein virtuelles Betriebssystem, in dem Fall Ubuntu, installiert werden. In Ubuntu werden der Terminator und Sublime Text installiert und ausgeführt.

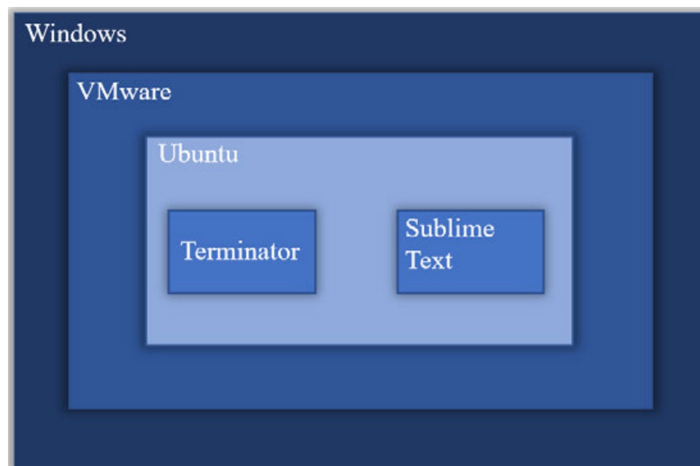


Abbildung 16: Verteilungsdiagramm für die Darstellung der verwendeten Software (Quelle: eigene Darstellung)

Für die Software ist im Anhang die vollständige Installationsanleitung, in der auch die Bedienanleitung integriert ist, zu finden (Anhang B). Anzumerken ist, dass die Software, die für die Simulation verwendet wird, eine Open Source und demnach frei verfügbar ist.

5.2 Funktionen der Maschinen in der Simulation

Die bedeutenden Funktionen der Maschinengruppen und deren Maschinen in der Simulation sollen im Folgenden beschrieben werden. Zu Beginn wird die Verwendung von Threading erklärt. Anschließen wird die Einstellmöglichkeit der Bearbeitungszeit der Maschinen vorgestellt. Die Screenshots in diesem Abschnitt wurden aus der Datei „matt-station.py“ entnommen.

Im Vergleich zum Planspiel haben die Maschinen in der Simulation nicht die Funktion untereinander zu verhandeln und sind nicht voneinander unabhängig. Dadurch wird die Durchführung der Simulation erschwert, weil die Software der Maschinen immer nur einen Kundenauftrag für alle Maschinen abarbeiten kann und die Produktion infolgedessen verlangsamt wird. Um diesen Effekt zu umgehen, wird auf Threading zurückgegriffen. Das Threading ermöglicht dem Quellcode, alle Kundenauftrage parallel zu bearbeiten. Abbildung 17 zeigt eine Zeile aus dem Quellcode, in der das Threading ausgeführt wird.

```
thread = threading.Thread(target=working, args=(order,))
```

Abbildung 17: Screenshot: Ein Thread wird erstellt (Quelle: eigene Darstellung)

Das Modul „threading“ ruft die Klasse „Thread“ auf und wird in der Variable „thread“ gespeichert. Im Thread wird die Funktion „working“ mit dem Argument „order“ aufgerufen. Die Funktion „working“ ist für die Bearbeitungszeit der Maschinen zuständig und ruft nach der Bearbeitung die die Funktion „move_to_buffer“ auf, in der entschieden wird, auf welchen Puffer der Kundenauftrag platziert wird. Mit dem Methodenaufruf in Abbildung 18 wird der Thread ausgeführt. In Abbildung 19 ist ein Screenshot zu sehen, in dem die „worktime“ zu erkennen ist.

```
thread.start()
```

Abbildung 18: Screenshot: Thread wird gestartet (Quelle: eigene Darstellung)

```
stationen = {
    'S1':{'pos':[11,1], 'worktime': 6},
    'S2':{'pos':[11,4], 'worktime': 4},
    'S3':{'pos':[11,7], 'worktime': 8},
    'E1':{'pos':[6,2], 'worktime': 5},
    'E2':{'pos':[6,6], 'worktime': 8},
    'A':{'pos':[12,4], 'worktime': 1}
}
```

Abbildung 19: Screenshot: Bearbeitungszeit kann für jede Maschine eingestellt werden (Quelle: eigene Darstellung)

Mit der Manipulation der „worktime“ im Quellcode lässt sich die Bearbeitungszeit an jeder Maschine in Sekunden einstellen.

Abschließend ist zu erwähnen, dass die Puffer nicht dieselbe Funktion/ Aufgabe verrichten wie im Planspiel. Die Puffer sind in der Simulation enthalten, allerdings wurde in dieser Arbeit kein Wert auf die Funktionalität gesetzt. Insofern wird jeder Kundenauftrag immer auf den ersten Puffer, also Puffer „S1“ und Puffer „E2“ einer Maschinengruppe gelagert.

5.3 Aufbau des Kundenauftrages

In diesem Kapitel wird der Aufbau des Kundenauftrages erläutert. Hierfür wird zuerst die Struktur eines Kundenauftrages vorgestellt. Anschließend werden die Daten eines Kundenauftrages anhand eines Beispiels aktualisiert.

Der Kundenauftrag hat die primäre Aufgabe die Simulation auszulösen. Hierfür benötigt jeder Kundenauftrag Daten, anhand derer die FTFs intern Entscheidungen treffen. In Kapitel 4.2.2 wurde bereits die Datenstruktur des Kundenauftrages vorgestellt, die in der Simulation übernommen wurde. Abbildung 20 zeigt alle verfügbaren Varianten eines möglichen Kundenauftrages, die einer Variable „orders_list“ übergeben werden.

```
orders_list = [
    {'id':str(uuid.uuid4())[:3], 'pos':'A', 'dest':'S1', 'route': ('S1','E1')},
    {'id':str(uuid.uuid4())[:3], 'pos':'A', 'dest':'S1', 'route': ('S1','E2')},
    {'id':str(uuid.uuid4())[:3], 'pos':'A', 'dest':'S2', 'route': ('S2','E1')},
    {'id':str(uuid.uuid4())[:3], 'pos':'A', 'dest':'S2', 'route': ('S2','E2')},
    {'id':str(uuid.uuid4())[:3], 'pos':'A', 'dest':'S3', 'route': ('S3','E1')},
    {'id':str(uuid.uuid4())[:3], 'pos':'A', 'dest':'S3', 'route': ('S3','E2')},
]
```

1 2 3

Abbildung 20: Screenshot: Unterschiedliche Varianten eines Kundenauftrages (Quelle: eigene Darstellung)

In der Liste befinden sich sechs Varianten eines Kundenauftrages, die einzigartig sind und die jeweils in einer Dictionary gespeichert wurden. Jede Dictionary besteht aus vier Schlüssel-Wert-Paaren und einem Tupel. In Punkt 1 ist das erste Schlüssel-Wert-Paar abgebildet. Der Schlüssel ist hierbei die „id“. Der Wert wird durch eine „UUID4“ erzeugt und auf drei Zeichen beschränkt. Universally Unique Identifier (UUID) ist ein Standard für Identifikationsnummern und besteht in der Regel aus einer hexadezimalen Zahl, die 16 Byte groß ist. Die vier in der Abkürzung „UUID4“ steht für die Version des Universally Unique Identifier (vgl. Augsten 2019). Infolgedessen hat jeder Kundenauftrag eine einmalige ID. In Punkt 2 sind zwei Schlüssel-Wert-Paare abgebildet, die aktuelle Position „pos“ und das Ziel, „dest“, wohin der Kundenauftrag transportiert wird. In Punkt 3 ist wiederum ein Schlüssel-Wert-Paar zu sehen. Der Schlüssel hat in diesem Fall einen zweistelligen Wert, der in einem Tupel gespeichert ist.

Einer der Kundenaufträge wird mit einem Zufallsgenerator ausgewählt und anschließend über den MQTT-Broker an die Komponente FTFs publiziert. Dieses Muster eines Kundenauftrages dient als Referenz und wird bei jeder Ankunft an eine Maschine oder in einen Puffer aktualisiert und die Daten gespeichert. In Abbildung 21 ist ein Beispiel einer Aktualisierung zu erkennen, die vor der ersten Maschinengruppe stattfindet.

```
msg = {
    'id': order['id'],
    'pos': order['dest'],
    'dest': order['route'][1],
    'station': order['dest'],
    'route': order['route'],
    'car_id': car_id,
}
```

Abbildung 21: Screenshot: Aktualisierung der „order“-Informationen (Quelle: eigene Darstellung)

Hierfür wird eine neue Dictionary erzeugt und in die Variable „msg“ gespeichert. In der folgenden Auflistung ist die Aktualisierung eines Kundenauftrages und dessen Inhalt zu sehen.

Zur Darstellung der Aktualisierung wird der erste Kundenauftrag aus Abbildung 21 verwendet. Der Kundenauftrag bekommt eine frei gewählte ID „HAW“ für dieses Beispiel.

Beginn der Aktualisierung des Kundenauftrages „HAW“:

---

1. Die „id“ wird vom Kundenauftrag übernommen.
2. Die „pos“ beschreibt die aktuelle Position und wird mit der „dest“, also dem aktuellen Ziel des Kundenauftrages, besetzt.
3. Dem Schlüssel „dest“ wird das erste Element aus dem Schlüssel „route“ in der „order“ zugewiesen.
4. Der Schlüssel „station“ ist neu in der Liste und diesem wird das aktuelle Ziel des Kundenauftrages „order['dest'] zugewiesen. Mit diesem Schlüssel-Wert-Paar orientiert sich der Quellcode der Maschine und weiß, an welcher Maschine der Kundenauftrag bearbeitet werden soll.
5. Dem Schlüssel „route“ wird der gleiche Wert zugewiesen, der in dem Kundenauftrag ist.
6. Zum Schluss wird das Schlüssel-Wert-Paar „'car_id': car_id“ erstellt das Auskunft darüber gibt, welches FTF den Kundenauftrag zu der Maschine transportiert hat.

Nachdem eine neue Dictionary aus den Daten des Kundenauftrages erstellt wurde, sieht der Inhalt der Dictionary wie folgt aus:

msg = {'id': HAW, 'pos': 'S1', 'dest': 'E1', 'station': 'S1', 'route':('S1': 'E1'), 'car_id': car_id}

Die Dictionary „msg“ wird anschließend an die Maschine gesendet, in der die Dictionary als „order“ deklariert wird. Dieser Prozess der Aktualisierung erfolgt bei jedem Kundenauftrag in der Simulation.

5.4 Verbindungsaufbau zu MQTT

Dieses Kapitel befasst sich mit der Herstellung der Verbindung zum MQTT-Broker, der in der Simulation verwendet wird. Zunächst wird der MQTT-Broker vorgestellt, anschließend wird anhand von Ausschnitten aus dem Quellcode die Verbindungsoption erläutert.

Um MQTT nutzen zu können, wird ein Broker benötigt. Hierfür gibt es viele unterschiedliche Lösungen wie HiveMQ oder MQTT.fx, die eine grafische Oberfläche anbieten, in der jede Nachricht am Bildschirm mitverfolgt werden kann. Allerdings wird diese Eigenschaft für eine

Interaktion nicht benötigt, weshalb die Entscheidung auf Mosquitto gefallen ist. Der Vorteil eines Mosquitto-Brokers ist, dass der Broker lokal auf einem Rechner ohne Anschluss an das Internet funktioniert und im Hintergrund arbeitet. Nach der Installation beginnt Mosquitto mit seiner Arbeit. Damit die Komponenten miteinander interagieren können, muss vorher eine Verbindung zu Mosquitto hergestellt werden. Für die Darstellung des Verbindungsaufbaus werden Screenshots aus der Datei „mqtt-order.py“ verwendet. Der erste Schritt für den Verbindungsaufbau ist in Abbildung 22 zu sehen (vgl. Eclipse Paho™ MQTT Python Client 2020). Hierfür wird ein Objekt der Klasse „Client“ erstellt und zur Vereinfachung in der Variable „mqttc“ gespeichert und in den nachfolgenden Schritten verwendet.

```
mqttc = mqtt.Client()
```

Abbildung 22: Screenshot: Client-Deklaration (Quelle: eigene Darstellung)

In Abbildung 23 ist der Teil des Quellcodes zu sehen, der für die Verbindung zum Broker zuständig ist. In diesem Fall wird der localhost angesteuert, der Port 1833 verwendet und alle 60 Sekunden wird ein Signal an den Broker gesendet, um die Verbindung aufrecht zu erhalten (keep alive) (vgl. Eclipse Paho™ MQTT Python Client 2020). Unter localhost ist der eigene Rechner zu verstehen, wodurch ohne Kenntnisse der eigenen IP-Adresse eine Verbindung zu einer Anwendung auf dem Rechner hergestellt werden kann (vgl. Plenk 2017. S. 56). Der Port 1833 hat die Bedeutung, dass die Daten, die versendet werden, unverschlüsselt sind.

```
mqttc.connect("localhost", 1883, 60)
```

Abbildung 23: Screenshot: Verbindungsaufbau zum Broker Mosquitto (Quelle: eigene Darstellung)

Zu erwähnen ist, dass die Verwendung des Port 1833 in Kombination mit localhost nicht benötigt wird, weil Port 1833 für Netzwerke gedacht ist und demnach für den Aufbau einer Verbindung zum Broker nicht relevant ist. Nachdem eine Verbindung aufgebaut ist, hat der Kundenauftrag die Möglichkeit Topics zu subskribieren und in Topics zu publizieren. Um eine Nachricht in ein Topic zu publizieren, muss man die Instanz „mqttc“ und die Methode „publish“ aufrufen, die sich in der Klasse „Clients“ befindet. Abbildung 24 zeigt den Methodenaufruf „publish“, dessen Inhalt das Topic, in diesen Fall „new-order“, in das publiziert wird, die

```
mqttc.publish("new-order", json.dumps(orders_list[el]), qos=2)
```

Abbildung 24: Screenshot: Kundenauftrag publiziert eine Nachricht über MQTT (Quelle: eigene Darstellung)

Information eines Kundenauftrages „json.dumps(orders_list[el])“, die in eine Json Datei umgewandelt wird und den Quality-of-Service (QoS) „qos=2“, welche eine sichere Ankunft der Information sicherstellt (vgl. Eclipse Paho™ MQTT Python Client 2020). Damit die FTFs die Nachricht empfangen können, müssen diese das Topic „new-order“ subskribiert haben. In Abbildung 25 ist der Methodenaufruf zu erkennen, wodurch das Subskribieren ermöglicht wird (vgl. Eclipse Paho™ MQTT Python Client 2020).

```
mqttc.subscribe("new-order", 0)
```

Abbildung 25: Screenshot: Methodenaufruf zum Subskribieren eines Topics (Quelle: eigene Darstellung)

Neben dem Topic „new-order“ muss in den Methodenaufruf eine QoS eingegeben werden. Auf die Vergabe einer sinnvollen QoS im Quellcode und für den Austausch der Daten wird in dieser Arbeit kein Wert gelegt. Anschließend wird der Methodenaufruf „message_callback_add“ aufgerufen (Abbildung 26) (vgl. Eclipse Paho™ MQTT Python Client 2020).

```
mqttc.message_callback_add('new-order', on_new_order)
```

Abbildung 26: Screenshot: Aufruf der Methode „message_callback_add“ (Quelle: eigene Darstellung)

Diese Methode ermöglicht bei Ankunft einer Nachricht in das Topic, das in die Methode eingetragen wurde, eine Funktion aufzurufen. In diesem Fall wird die Funktion „on_new_order“ ausgelöst und die FTFs beginnen mit der Verhandlung.

5.5 Wegfindung mit dem A* Algorithmus

Im Folgenden soll die Wegfindung der FTFs in der Simulation erläutert werden. Für die Visualisierung der Wegfindung werden Screenshots aus dem Quellcode der Datei „mqtt-car.py“ und dem Terminator dargestellt.

Für die Berechnung der Route wird der A*-Algorithmus verwendet. Hierfür wird ein bereits programmierter Quellcode verwendet und an die Bedingungen der Produktionslinie aus dem Planspiel angepasst (vgl. Swift 2017). Für die Berechnung der Route werden zwei Informationen benötigt: zum einen die Startposition und zum anderen die Zielposition. Die Startposition wird zu Beginn der Simulation für jedes FTFs mit einem Zufallsgenerator ermittelt. Hierfür kann im Quellcode ein Rahmen eingestellt werden, in dem die FTFs erscheinen sollen. Abbildung 27 zeigt einen Ausschnitt aus dem Quellcode, der für die Generierung der Startposition zuständig ist. Das Ergebnis wird in der Variable „car_pos“ gespeichert.

```
# Hier kann entschieden werden, wo die Fahrzeuge zufällig auf dem Spielfeld generieren.
# car_pos=(x(Zeile),y(Spalte))
car_pos = (random.randint(11, 12), random.randint(0, 8))
```

Abbildung 27: Screenshot: Zufallsgenerator für die Startposition der FTFs (Quelle: eigene Darstellung)

In Abbildung 28 ist die angepasste Produktionslinie zu sehen, die vertikal verläuft. Hierbei symbolisieren die Ziffer eins entweder eine Maschine oder ein Puffer. In Punkt 1 ist zu erkennen, in welchem Bereich die FTFs nach der Aktivierung erscheinen sollen. Der Definitionsbereich für die Startposition ergibt sich aus dem Zufallsgenerator in Abbildung 27. Punkt 2 zeigt die erste Maschinengruppe, Punkt 3 die zweite Maschinengruppe und Punkt 4 die dritte Maschinengruppe. Zudem sind die Puffer hinter den ersten beiden Maschinengruppen zu erkennen.

```
#          0  1  2  3  4  5  6  7  8
maze = [[0, 1, 0, 0, 1, 0, 0, 1, 0,], #0
        [0, 0, 0, 0, 0, 0, 0, 0, 0,], #1
        [0, 0, 0, 0, 0, 0, 0, 0, 0,], #2
        [0, 0, 0, 0, 0, 0, 0, 0, 0,], #3
        [0, 0, 1, 1, 1, 1, 1, 0, 0,], #4
        [0, 0, 1, 0, 0, 0, 1, 0, 0,], #5
        [0, 0, 0, 0, 0, 0, 0, 0, 0,], #6
        [0, 0, 0, 0, 0, 0, 0, 0, 0,], #7
        [0, 0, 0, 0, 0, 0, 0, 0, 0,], #8
        [0, 1, 1, 1, 1, 1, 1, 1, 0,], #9
        [0, 1, 0, 0, 1, 0, 0, 1, 0,], #10
        [0, 0, 0, 0, 0, 0, 0, 0, 0,], #11
        [0, 0, 0, 0, 0, 0, 0, 0, 0,]] #12
```

Abbildung 28: Screenshot: Angepasste Produktionslinie für den A*-Algorithmus (Quelle: eigene Darstellung)

Die Zielposition wird aus den Informationen des eingetroffenen Kundenauftrages entnommen, die in Kapitel 5.3 erläutert wurde und in ein Raster, welches im Quellcode definiert ist, auf dem Spielfeld umgewandelt. In Punkt 1 der Abbildung 29 sind die Raster der Maschinen und in Punkt 2 der Abbildung die Raster der Puffer „S1-S7“ und „E1-E5“ zu erkennen.

```
1
'S1':{'pos':[11,1],
'S2':{'pos':[11,4],
'S3':{'pos':[11,7],
'E1':{'pos':[6,2],
'E2':{'pos':[6,6],
'A':{'pos':[12,4],
'V1':{'pos':[1,1],
'V2':{'pos':[1,4],
'V3':{'pos':[1,7],
```

```
2
'S1':{'pos':[8,1],
'S2':{'pos':[8,2],
'S3':{'pos':[8,3],
'S4':{'pos':[8,4],
'S5':{'pos':[8,5],
'S6':{'pos':[8,6],
'S7':{'pos':[8,7],
'E1':{'pos':[3,2],
'E2':{'pos':[3,3],
'E3':{'pos':[3,4],
'E4':{'pos':[3,5],
'E5':{'pos':[3,6],
```

Abbildung 29: Screenshot: Maschinen und deren Anfahrposition (Quelle: eigene Darstellung)

Ein FTF muss demnach diese Raster anfahren, damit ein Kundenauftrag einer Maschine übergeben oder von einem Puffer abgeholt werden kann. Infolgedessen berechnet der A*-Algorithmus die kürzeste Route für den Transport, in der die Maschinengruppen umfahren werden. Hierfür wird die Klasse „astar“ in der Funktion „get-path“ aufgerufen und die Route anschließend an die Variable „path“ übergeben (Abbildung 30).

```
def get_path(start, end):
    path = astar.astar(maze, start, end)
    print_path(path)
    return path
```

Abbildung 30: Screenshot: Berechnung der Route durch den A*-Algorithmus (Quelle: eigene Darstellung)

Anschließend wird die Funktion „print-path“ aufgerufen, in der die Produktionslinie und die Route eines FTFs gezeichnet wird, die im Anschluss im Terminator zu sehen ist (Abbildung 32). Hierfür wird zu Beginn die Referenzproduktionslinie kopiert. Grund hierfür ist, dass die Referenzproduktionslinie nicht überschrieben wird und nur die Route auf der Kopie der Produktionslinie zu sehen ist, die für den aktuellen Transport benötigt wird (Abbildung 31, Punkt 1). Nachdem ein FTF die Route gefahren ist, wird die Kopie verworfen. In Punkt 2 werden die Koordinaten der berechneten Route in die Kopie der Produktionslinie eingetragen. Hierfür wird die Ziffer 0 durch die Ziffer 2 ersetzt.

```
def print_path(path):
1   temp_maze = copy.deepcopy(maze)

2   for step in path:
        temp_maze[step[0]][step[1]] = 2

3   for row in temp_maze:
        line = []

        for col in row:
            if col == 1:

                line.append("\u2588")
            elif col == 0:

                line.append(" ")
            elif col == 2:

                line.append(".")
        print("".join(line))
```

Abbildung 31: Screenshot: Funktion „print_path“ (Quelle: eigene Darstellung)

In Punkt 3 beginnt die Funktion mit der Zeichnung der Produktionslinie inklusive der berechneten Route. Hierfür wird eine neue Liste „line = []“ erstellt, in der jedem Raster aus der Pro-

duktionslinie ein Symbol zugeordnet wird. Infolgedessen wird eine „1“ durch die Unicodenummer „\u2588“ ersetzt, was einen vollen Block repräsentiert. Eine „0“ wird durch ein freies Feld ersetzt und eine „2“ durch einen Punkt, der die Route eines FTFs darstellt. Im Anschluss wird das erzeugte Bild in einem Terminal eines FTFs wiedergegeben. In Abbildung 32 Punkt 1 ist ein Screenshot aus dem Terminator abgebildet, in dem die Produktionslinie, die durch die Funktion „print_path“ erstellt wurde, inklusive der berechneten Route zu sehen ist. Punkt 2 zeigt die aktuelle Position und das Ziel an. In diesem Fall fährt das FTF von Maschine E1 zum Puffer E1.

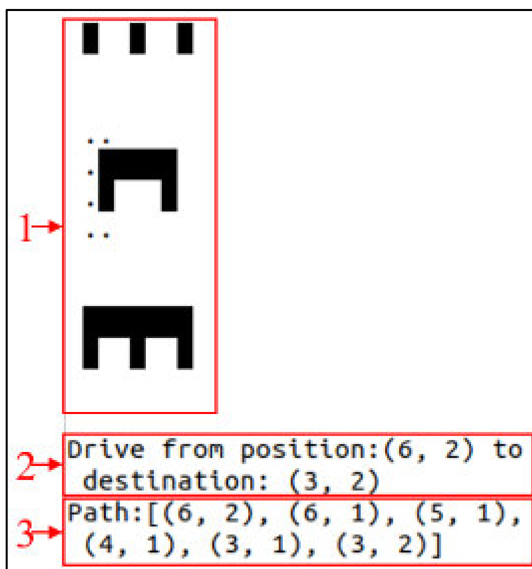


Abbildung 32: Screenshot: Produktionslinie im Terminator inklusive Route (Quelle: eigene Darstellung)

Abschließend ist in Punkt 3 die berechnete Route zu sehen, die in der Produktionslinie in die jeweiligen Raster eingetragen wurde und welche das FTF abfahren wird. Die Fahrt eines FTFs wird anhand einer Fortschrittsanzeige dargestellt (Abbildung 33).

Driving ████████████████████████████████████ 100%

Abbildung 33: Screenshot: Fortschrittsanzeige für die Darstellung des Fahrens eines FTFs (Quelle: eigene Darstellung)

Wenn das FTF losfährt, beginnt die Fortschrittsanzeige bei 0% und endet bei 100%, wenn das FTF am Ziel angekommen ist. Der komplette Prozess der Berechnung der Route bis zum Darstellen der Produktionslinie inklusive der Route erfolgt für jeden neuen Fahrauftrag von neuem.

6 Prototypische Anwendung der Simulation

Nach Abschluss der Implementierung wird in diesem Kapitel die praktische Komponente der vorliegenden Arbeit thematisiert. Für die Visualisierung der Simulation werden zunächst Vorbereitungen getroffen, um die Simulation zu starten. Anschließend wird die Verhandlung anhand ausgewählter Screenshots demonstriert, in denen grundlegende Elemente aus Kapitel 3 und Kapitel 4 zu erkennen sind. Im Hinblick auf eine vollständige Darstellung der Abläufe in der Simulation werden hierfür zwei Szenarien vorgestellt. Im ersten Szenario wird ein vollständiger Verhandlungsprozess eines Kundenauftrages dargestellt. Im zweiten Szenario wird der Quellcode so manipuliert, dass eine Entscheidungsfindung durch den Zufallsgenerator in den Verhandlungen der FTFs benötigt wird.

6.1 Vorbereitung zur Ausführung der Simulation

Damit die Simulation gestartet werden kann, müssen ein paar Einstellungen vorgenommen werden. Zu diesem Zweck muss der Anwender auf der Ubuntu-Oberfläche zum einen den Terminator starten, zum anderen muss sichergestellt werden, dass alle Dateien im selben Ordner zu finden sind, die zum Starten der Simulation benötigt werden. Folgende Dateien werden für die Simulation benötigt:

1. mqtt-car.py
2. mqtt-order.py
3. mqtt-station.py
4. astar.py

Nachdem der Terminator geöffnet wurde, kann nach Belieben ausgewählt werden, wie viele FTFs benötigt werden. Wie im Planspiel vorgegeben, werden vier FTFs ausgewählt und initialisiert. Zu diesem Zweck werden vier Terminals im Terminator benötigt, da jede Komponente in einem Terminal simuliert wird. Der Anwender hat die Auswahl, ob eine ID für jedes FTF automatisch generiert wird, oder die ID manuell erstellt wird. Zur besseren Übersicht wird die Verteilung der IDs manuell vorgenommen und die FTFs bekommen die Bezeichnung A1, B1, C1, D1. Des Weiteren wird ein Terminal für die Maschinengruppen benötigt, für die immer ein Terminal im Terminator reserviert ist. Ein Kundenauftrag, für den ebenfalls ein Terminal erstellt werden muss, rundet die Voraussetzungen für den Beginn der Simulation ab.

Abbildung 34 zeigt den Terminator, indem die FTFs auf der rechten Seite mit der ausgewählten Bezeichnung zu sehen sind (Punkt 1). Die beiden Kundenaufträge sind auf der linken Seite mit der Bezeichnung „order“ zu erkennen (Punkt 2). Unten links ist der Terminal mit den Maschinengruppen zu erkennen (Punk 3). Bevor die Simulation durch einen der Kundenaufträge ausgelöst wird, müssen alle FTFs initialisiert sein und die Maschinengruppen aktiviert werden.

Anschließend kann einer der Kundenaufträge in der Simulation aktiviert werden. In der Installationsanleitung werden die Schritte für die Herangehensweise der Vorbereitung im Einzelnen genauer erklärt. Die Anordnung der Terminals ist zufällig gewählt.

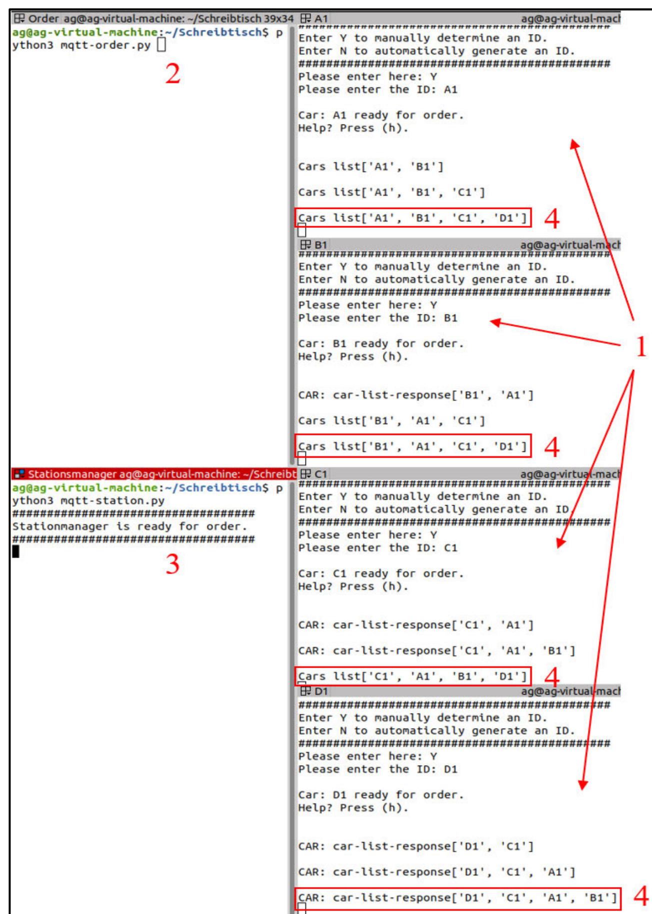


Abbildung 34: Screenshot: Initialisierungsprozess der FTFs und Aktivierung der Maschinengruppen (Quelle: eigene Darstellung)

Die FTFs sind nach alphabetischer Reihenfolge in der Simulation aktiviert und initialisiert. Auf Abbildung 34 in Punkt 4 ist das Ergebnis der erfolgreichen Initialisierung zu erkennen. Des Weiteren wird in jedem Terminal der FTFs eine „Help?"-Abfrage gestellt. Diese öffnet ein Menü, in dem alle benötigten Eckdaten angezeigt werden können. Abbildung 35 zeigt das Menü und deren Inhalt. Das „Help?"-Menü kann zu jeder Zeit während der Simulation in jedem Terminal eines FTFs durch das Betätigen der „h" Taste aufgerufen werden. Anzumerken ist, dass nur die Eckdaten eines FTFs angezeigt werden, welches ausgewählt wurde. Die Eckdaten werden angezeigt, sobald einer der Buchstaben im Terminal eingegeben wird. In der folgenden Liste werden die Eckdaten aus dem „Help?"-Menü erklärt.

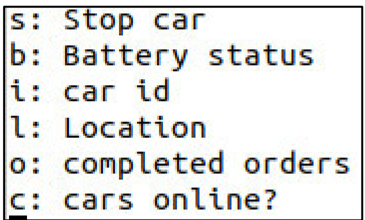


Abbildung 35: Screenshot: Das „Help?"-Menü und dessen Inhalt (Quelle: eigene Darstellung)

1. Stop car: Dieser Punkt im Menü erfüllt die Anforderung 37.
2. Battery status: Zeigt den Batterieladezustand des ausgewählten FTFs an (Anforderung 51).
3. Car id: Zeigt die ID des ausgewählten FTFs an.
4. Location: Zeigt an, auf welcher Position sich das ausgewählte FTF befindet.
5. Completed orders: Zeigt an, welche Kundenaufträge wohin von dem ausgewählten FTF gefahren wurden.
6. Cars online?: Zeigt an, welche FTFs online sind.

Die Vorbereitung ist abgeschlossen und der Kundenauftrag kann in der Simulation aktiviert werden.

6.2 Beginn der Verhandlungen

Nachdem alle notwendigen Schritte eingeleitet wurden, damit alle gewünschten Komponenten im Terminator sichtbar sind, kann die Simulation gestartet werden.

Abbildung 36: Screenshot: Kundenauftrag „a69“ wurde erfolgreich an alle FTFs übermittelt (Quelle: eigene Darstellung)

Dies wird durch die Aktivierung des Kundenauftrages ermöglicht. Abbildung 36 zeigt den aktivierten Kundenauftrag mit der „order_ID“ „a69“ im eigenen Terminal (Punkt 1). Im Terminal aller FTFs ist zu erkennen, dass der Kundenauftrag bei allen FTFs angekommen ist, wodurch die Verhandlung beginnt (Punkt 2).

Abbildung 37 zeigt das Ergebnis der Verhandlungen. In jedem Terminal eines FTFs ist die Produktionslinie zu sehen, die 1:1 aus dem Raster des Planspiels nachgebaut wurde.

Abbildung 37: Screenshot: Verhandlungen abgeschlossen. FTF C1 bekommt den Kundenauftrag „a69“ (Quelle: eigene Darstellung)

Durch die Punkte auf dem Raster wird der Fahrweg angegeben, den ein FTF zum Auftragseingang fahren muss. Jedes FTF bekommt eine zufällig generierte Startposition, die während der Initialisierung automatisch festgelegt wird. Diese lautet in diesem Fall A1(12,1), B1(11,3), C1(11,3) und D1(12,7). Der genaue Fahrweg von jedem FTF wird in einer Zeile unter der Produktionslinie angegeben (Punkt 1).

Der A*-Algorithmus errechnet als erstes den Fahrweg und bildet diesen in Form von Punkten in der Produktionslinie in jedem Terminal eines FTFs ab. In Punkt 2 ist die „Car path list“ zu sehen, in der abzulesen ist, um welchen Kundenauftrag verhandelt wird und wie viel Raster jedes FTF zum Auftragseingang benötigt, um den Kundenauftrag abzuholen. Aus dem Vergleich ergibt sich, dass FTF C1 zwei Raster zum Auftragseingang benötigt und am schnellsten beim Kundenauftrag ist. Infolgedessen kann das FTF C1 die Verhandlung für sich entscheiden und kann den Kundenauftrag am Auftragseingang abholen und anschließend zu der Maschine S1 transportieren. In Punkt 3 ist zu sehen, dass das FTF C1 den Kundenauftrag erhalten hat. Dadurch wird das FTF C1 als beschäftigt markiert und die Information an die anderen FTFs übermittelt. Diese speichern den Zustand des FTF C1 in der eigenen „New car busy list“ (Punkt 4).

6.3 Transportvorgang

Nachdem FTF C1 die Verhandlung und den Kundenauftrag für sich entscheiden konnte, beginnt der Transport des Kundenauftrages „a69“. Hierfür wird das Terminal von FTF C1 betrachtet, indem der Transport simuliert wird.

```
C1    ag@ag-virtual-machine: ~/Schreibtisch 61x37
Drive to A, car_pos:(12, 3), destination: (12, 4)
Path:[(12, 3), (12, 4)]
Driving |████████████████████████████████| 100%
Pick up Order.
Drive to: S1
Path:[(12, 4), (12, 3), (12, 2), (12, 1), (11, 1)]
Driving |████████████████████████████████| 100%
          09:34:44 ag-virtual-machine __main__[67833] WARNIN
G Car: C1 delivered order: a69
          09:34:44 ag-virtual-machine __main__[67833] DEBUG
New car busy list: []
```

Abbildung 38: Screenshot: FTF C1 transportiert den Kundenauftrag „a69“ (Quelle: eigene Darstellung)

Die Terminals der übrigen FTFs sind in diesem Teil der Simulation nicht betrachtet. Der Transport des Kundenauftrages „a69“ von FTF C1 ist in Abbildung 38 zu sehen. In Punkt 1 ist in der ersten Zeile das Ziel des FTFs zu sehen: der Auftragseingang A, die eigene Position

„car_pos“ und auf welche Position das FTF fahren muss, um an den Kundenauftrag abzuholen „destination“. Zusätzlich ist in der zweiten Zeile der komplette Fahrweg „path“ abgebildet. Der Fortschrittsbalken in Punkt 2 simuliert die Fahrt des FTFs. 100% bedeutet demnach, dass das FTF am Auftragseingang angekommen ist. Punkt 3 ist symbolisch und gibt an, dass der Kundenauftrag abgeholt wurde. In Punkt 4 ist in der ersten Zeile das Ziel des FTF zu erkennen, welches auch im Kundenauftrag gespeichert ist. In der zweiten Zeile wird der Fahrweg „path“ abgebildet. Nachdem das FTF an der Maschine S1 angekommen ist, was in Punkt 5 zu erkennen ist, wird in Punkt 6 der Kundenauftrag „a69“ an die Maschine übergeben. Im Anschluß sendet FTF C1 eine Nachricht an alle FTFs mit dem Inhalt, dass der Status „beschäftigt“ aus der Liste „New car busy list“ aufgehoben werden kann.

6.4 Bearbeitung an den Maschinen

In Abbildung 39 ist die Übergabe des Kundenauftrages „a69“ an die Maschine zu sehen. Infolgedessen beginnt die Maschine mit der Bearbeitung (Punkt 1). Die Bearbeitungszeit wurde willkürlich auf sechs Sekunden eingestellt. Nach der Bearbeitung wird der Kundenauftrag auf Pufferplatz S1 zwischengelagert. Anschließend wird eine Nachricht an den Kundenauftrag mit einer aktualisierten Beschreibung seines aktuellen Status gesendet.

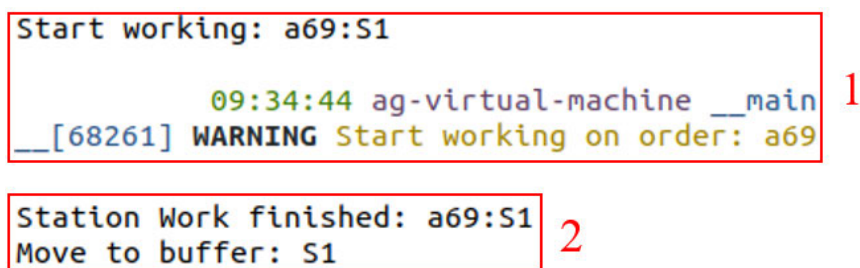


Abbildung 39: Screenshot: Maschine S1. Start und Ende der Bearbeitung des Kundenauftrages „a69“ (Quelle: eigene Darstellung)

In Abbildung 40 ist der aktualisierte Kundenauftrag zu erkennen. Auf Position „pos“ und dem Zielort „dest“ ist jetzt ein aktualisierter Eintrag zu erkennen. Hierdurch kann der Kundenauftrag eine neue Transportanfrage an die FTFs durchführen und die Verhandlung zwischen den FTFs beginnt wieder von vorne.

```
09:34:49 ag-virtual-machine __main
__[68541] ERROR on_order_buffer_ready: {'id':
 'a69', 'pos': 'S1', 'dest': 'E2', 'buffer':
'S1', 'route': ['S1', 'E2']}
```

Abbildung 40: Screenshot: Aktualisierter Kundenauftrag nach der Bearbeitung in Maschine S1 (Quelle: eigene Darstellung)

Hiernach wiederholt sich der Vorgang so lange, bis der Kundenauftrag in der dritten Maschinengruppe abgegeben wurde und als produziert gilt. Ein produzierter Kundenauftrag wird im Terminal der Maschinengruppen als „completed order“ markiert. In Abbildung 41 ist der Kundenauftrag „a69“ zu sehen, der als produziert gekennzeichnet ist.

```
09:35:15
ERROR Order Completed:{'id': 'a69',
 'pos': 'V',          'station': 'V', 'r
oute': ['S1', 'E2'], 'car_id': 'C1'}
```

Abbildung 41: Screenshot: Kundenauftrag „a69“ wurde produziert (Quelle: eigene Darstellung)

Als Zusatzinformation ist zu sehen, dass das FTF C1 den Kundenauftrag in die dritte Maschinengruppe transportiert hat. Jetzt macht der Aufruf der Option „completed orders“ im „Help?“-Menü Sinn, weil dort ein neuer Eintrag einzusehen ist.

```
s: Stop car
b: Battery status
i: car id
l: Location
o: completed orders
c: cars online?

completed orders: {'S1': ['a69'], 'E2': ['a69'], 'V':
['a69']}
```

Abbildung 42: Screenshot: Liste der produzierten Kundenaufträge (Quelle: eigene Darstellung)

In Abbildung 42 ist die ausgewählte Option „completed orders“ zu sehen. Hier kann über die gesamte Dauer der Simulation die Information abgerufen werden, an welcher Maschine welcher Kundenauftrag bearbeitet wurde.

6.5 Entscheidung durch den Zufallsgenerator

In diesem Szenario soll der Zufallsgenerator vorgestellt werden. Hierfür ist die Startposition für alle FTFs im Quellcode manipuliert worden, um die Entscheidung über den Zufallsgenerator zu erzwingen. Die FTFs haben in diesem Szenario die gleichen Bezeichnungen wie in Kapitel

6.1. Dieser Abschnitt dient ausschließlich der Darstellung des Zufallsgenerators in der Simulation. Das Szenario beginnt mit dem Eintreffen eines neuen Kundenauftrages mit der „order_ID“ „ade“ (Abbildung 43, Punkt 1), wodurch die Verhandlung unter den FTFs erfolgt (Punkt 2). In der ersten Zeile von Punkt 2 ist zu erkennen, dass jedes FTF die gleiche Fahrstrecke hat. Zusätzlich ist in der zweiten Zeile bei jedem FTF eine Rasteranzahl von fünf eingetragen. Infolgedessen hat jedes FTF den gleichen Fahrweg zum Auftragseingang, um den Kundenauftrag „ade“ abzuholen. Daraus resultiert, dass eine Entscheidung über den Zufallsgenerator getroffen werden muss. Punkt 3 zeigt das Ergebnis der Entscheidung, die zu Gunsten von FTF C1 gefallen ist, da FTF C1 den höchsten Wert hat.

Abbildung 43: Screenshot: Verhandlungen zwischen FTFs, um den Kundenauftrag mit der „order_ID“ „ade“ (Quelle: eigene Darstellung)

Nachdem der Kundenauftrag von der Maschine S3 bearbeitet und in den Puffer S1 gespeichert wurde, wird eine neue Transportanfrage an die FTFs gesendet (Punkt 1). In Abbildung 44 ist

das Ergebnis der Verhandlungen zu sehen. In der ersten Zeile von Punkt 2 ist der Fahrweg zu Puffer S1 zu sehen. In der zweiten Zeile ist zu erkennen, dass FTF C1 zwölf Raster zurücklegen muss und alle anderen jeweils sechs Raster. Infolgedessen scheidet FTF C1 aus der Verhandlung aus. Weil die übrigen FTFs einen Fahrweg von sechs Rastern haben, muss erneut ein Gewinner durch den Zufallsgenerator ermittelt werden. In Punkt 3 ist das Ergebnis der Entscheidungsfindung zu sehen. Das FTF D1 hat das höchste Ergebnis und darf den Kundenauftrag in die zweite Maschinengruppe zur Maschine E1 transportieren.

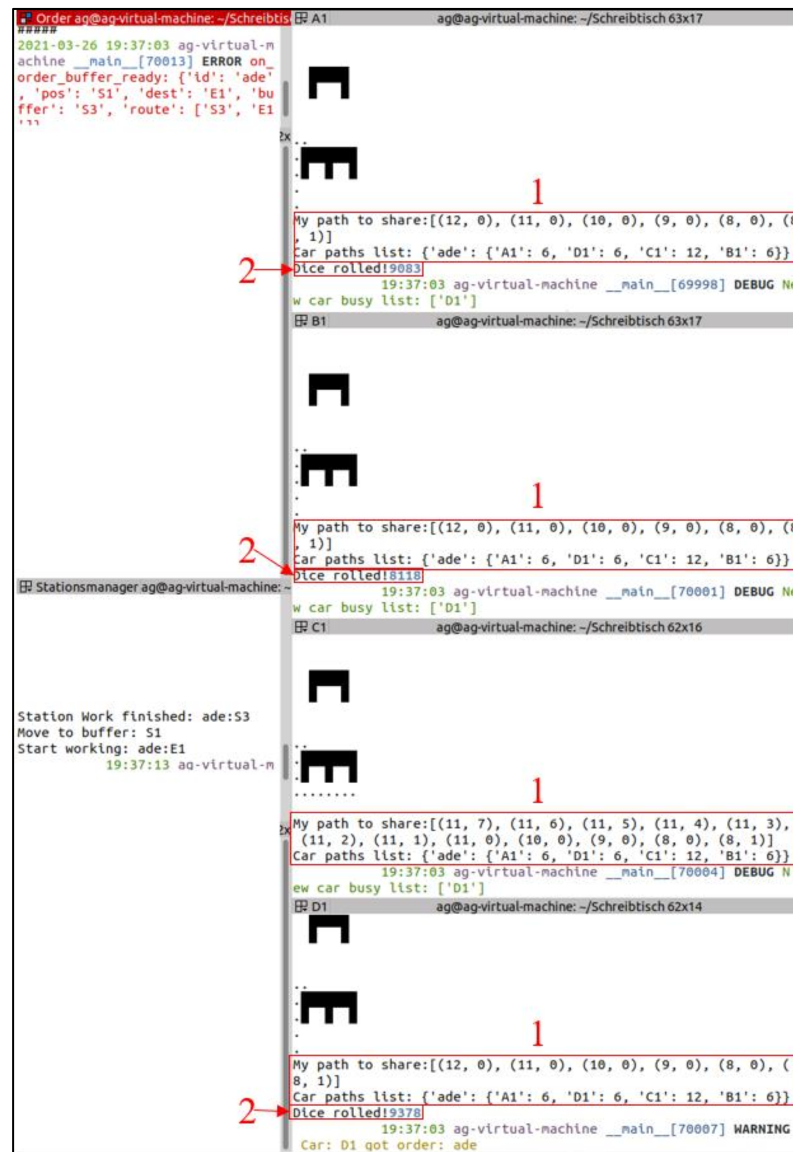


Abbildung 44: Screenshot: Zweite Verhandlungsrunde mit Hilfe des Zufallsgenerators um den Kundenauftrag mit der „order_ID" „ade" (Quelle: eigene Darstellung)

Anschließend transportiert das FTF D1 den Kundenauftrag „ade" in die dritte Maschinengruppe und der Kundenauftrag wird letztendlich als „produziert" gekennzeichnet.

7 Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, die Kommunikationsstrukturen und -abläufe aus einem Planspiel, das eine dezentrale Produktionssteuerung simuliert, zu analysieren, in einen Algorithmus zu transformieren und in eine Simulation zu integrieren. Um die Ergebnisse zu präsentieren, wird im ersten Abschnitt dieses Kapitels ein Anforderungsabgleich vorgestellt. Im zweiten Teil erfolgt ein Ausblick auf mögliche Erweiterungen der Simulation.

7.1 Zusammenfassung

Entsprechend der Teilziele aus Kapitel 1.2 wurde eine Simulation realisiert, die sich an das vorgestellte Planspiel aus Kapitel 3.2 anlehnt. Hierbei sendet ein Kundenauftrag eine Transportanfrage an die FTFs, die FTFs bearbeiten die Transportanfrage dezentral und transportieren den Kundenauftrag durch die Produktion. Hierfür wurden von den 64 Anforderungen, die insgesamt zusammengestellt wurden, 40 mit der Gewichtung „Muss“ bewertet, weil diese ausschlaggebend für die Realisierung der Simulation sind. Der Grund hierfür ist, dass der Schwerpunkt auf ein funktionierendes Programm gesetzt wurde. Somit wurden nur die Anforderungen implementiert, die zum einen eine erfolgreiche Verhandlung ermöglichen und zum anderen eine Simulation, die dem definierten Ziel der Simulation aus Kapitel 4.1 entspricht.

Bei näherer Betrachtung der in der Simulation implementierten Funktionen ist in Abbildung 45 zu erkennen, dass von den insgesamt 64 erhobenen Anforderungen 51, also 79% der gesamten Anforderungen realisiert wurden. Hierbei ist festzuhalten, dass alle Anforderungen mit der höchsten Gewichtung „Muss“, die insgesamt 38 (59%) der Anforderungen ausmachen, in der Simulation vertreten sind. Von den Anforderungen mit der Gewichtung „Soll“, von denen es insgesamt 15 gibt, wurden 46%, also 7 (11%) der Anforderungen realisiert. Von der Gewichtung „Kann“, von denen es insgesamt 11 Anforderungen gibt, wurden 54%, also 6 (9%) der Anforderungen realisiert.

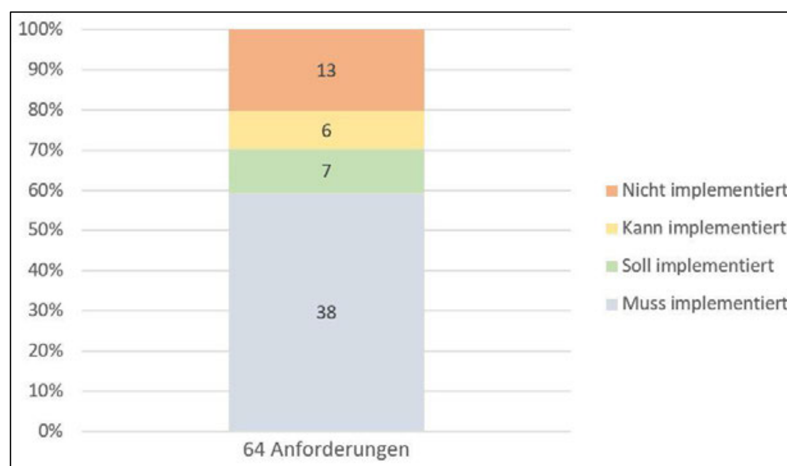


Abbildung 45: Anforderungsabgleich (Quelle: eigene Darstellung)

Nr.	Anforderungen	Kapitel	Ge-wicht	Impl.
1	Das Planspiel ist rundenbasiert	x	Kann	Nein
2	Es ist ein Spielfeld vorhanden (Produktionsstraße)	5.5	Muss	Ja
3	Das Fahren der FTFs wird im Planspiel dargestellt	5.5	Muss	Ja
4	Infokarten, Notizkarten zum Notieren von Eckdaten	6.1	Muss	Ja
5	3 Auftragseingänge	5.5	Muss	Ja
6	3 Auftragsausgänge	5.5	Muss	Ja
7	5 Fertigungsstationen	5.5	Muss	Ja
8	Spieler führen Verhandlungen für die FTFs durch (FTFs in der Simulation)	4.4	Muss	Ja
9	Die Spieler führen Verhandlungen für die Maschinen durch (Maschinen in der Simulation)	x	Soll	Nein
10	Pufferplätze hinter den Maschinen	5.5	Muss	Ja
11	FTF Status (verfügbar/nicht verfügbar)	6.2	Muss	Ja
12	FTF hat Informationen des zu transportierenden Kundenauftrages	6.2	Muss	Ja
13	Dauer der Fahrzeugbelegung	x	Soll	Nein
14	4 FTFs	6.1	Muss	Ja
15	Reservierung eines Kundenauftrages durch ein FTF	6.2	Muss	Ja
16	Transport eines Kundenauftrages durch ein FTF	6.3	Muss	Ja
17	Individuelle ID für jedes FTF	6.1	Muss	Ja
18	Aktuelle Position eines FTFs auf dem Raster kann abgelesen werden	6.3	Muss	Ja
19	Jeder Kundenauftrag enthält Informationen über einen eindeutigen Weg durch die Produktion	5.3	Muss	Ja
20	Der Kundenauftrag löst das Planspiel aus, die nachfolgenden Aufträge halten das Planspiel im Gang	6.2	Muss	Ja
21	Sobald ein Kundenauftrag für die FTFs zur Verfügung steht, muss die aktuelle ZE notiert werden	x	Kann	Nein
22	Jeder Kundenauftrag hat eine maximale Produktionsdauer bis zum Liefertermin in ZE, die nicht überschritten werden darf	x	Kann	Nein
23	Jeder Kundenauftrag hat eine eigene Priorität	x	Kann	Nein
24	Der Kundenauftrag enthält Informationen über die aktuelle Position auf dem Raster	5.3	Kann	Ja
25	Zeitstempel pro angesteuerte Maschine (Maschine, Puffer, FTF)	x	Kann	Nein
26	Individuelle ID für jeden Kundenauftrag	5.3	Muss	
27	Welcher Kundenauftrag wurde von welchen FTF gefahren?	6.4	Soll	Ja
28	Zufällige Aufträge werden am Auftragseingang generiert	5.3	Muss	Ja
29	Maschinen-Status (verfügbar/nicht verfügbar)	x	Soll	Nein
30	Dauer der Maschinenbelegung	x	Soll	Nein

31	Reservierung einer Maschine durch einen Kundenauftrag	x	Soll	Nein
32	Die Maschinen übernehmen zugestellte Aufträge	6.4	Muss	Ja
33	Jede Maschine schiebt einen fertigen Kundenauftrag in einen freien Puffer	6.4	Muss	Ja
34	Rüstzeit an den Maschinen in ZE	x	Soll	Nein
35	Bearbeitungszeit an der Maschine ist frei einstellbar	5.2	Soll	Ja
36	Herstellung der Kommunikation mit einem Netzwerk	5.4	Muss	Ja
37	An- und Abmeldung eines FTFs ist jederzeit möglich. Wenn im Laufe der Simulation ein weiteres FTF benötigt wird, kann von außen eingegriffen und ein neues FTF in die Simulation integriert werden. Wenn zu viele oder nicht benötigte FTFs in der Simulation sind, kann das nicht benötigte FTF abgemeldet werden.	6.1	Soll	Ja
38	Die FTFs teilen die eigene ID untereinander aus, wodurch jedes FTF weiß, welches und wie viele FTFs gerade aktiv sind	4.2.3, 4.3, 6.1	Muss	Ja
39	Manuelle ID Vergabe für FTF möglich	6.1	Kann	Ja
40	Jedes FTF kann Transportanfragen von Kundenaufträge empfangen	6.2	Muss	Ja
41	Jedes FTF muss eine neue Transportanfrage eines Kundenauftrages ablehnen, falls das FTF beschäftigt ist	4.4	Muss	Ja
42	Jedes FTF speichert Informationen des aktuell verhandelnden Kundenauftrag	4.4,6.2	Muss	Ja
43	Jedes FTF teilt den eigenen Standort mit den anderen FTFs	4.4	Muss	Ja
44	Jedes FTF vergleicht den eigenen Weg zum Ziel mit dem Weg der anderen FTFs	4.4, 6.2	Muss	Ja
45	Wenn zwei oder mehrere FTFs die gleiche Fahrstrecke zu einem Kundenauftrag haben, muss durch einen Mechanismus entschieden werden, welches FTF den Zuschlag für einen Kundenauftrag, über den gerade verhandelt wird.	4.4, 6.2	Muss	Ja
46	Dem Kundenauftrag mitteilen, dass er von einem FTF übernommen wurde	x	Soll	Nein
47	FTFs sollen einen letzten Wunsch bei Kommunikationsabbruch haben (Last Will), falls es zu Komplikationen kommt (Akku leer)	Anhang. C	Soll	Ja
48	Den Maschinen mitteilen, dass ein Kundenauftrag zugestellt wurde	6.4	Muss	Ja
49	Jedes FTF speichert jeden Kundenauftrag in einer Liste, in der zu erkennen ist, welcher Kundenauftrag an welcher Maschine war	6.4	Soll	Ja
50	Beim Fahren soll jedes FTF dynamischen Hindernissen ausweichen können	x	Kann	Nein

51	Jedes FTF hat ein Akkuladesystem	Anhang. C	Kann	Ja
52	Im Quellcode muss die Möglichkeit bestehen, die Fahrgeschwindigkeit eines FTF einzustellen	Anhang. C	Soll	Ja
53	Bei Auftragsablehnung muss der abgelehnte Kundenauftrag eine neue Transportanfrage an die FTFs senden	4.4	Muss	Ja
54	Jede Maschine enthält Informationen des Kundenauftrages, der gerade in einer Maschine produziert wird	5.3, 6.4	Muss	Ja
55	Die Maschine sendet dem Kundenauftrag eine Information, auf welchen Puffer der Kundenauftrag sich befindet	6.4	Muss	Ja
56	Ein Hauptmenü zur Übersicht von Eckdaten	6.1	Kann	Ja
57	Der Anwender kann ablesen, welcher Kundenauftrag von welchem FTF übernommen wurde	6.1	Soll	ja
58	Der Anwender kann ablesen, von welcher Position der Kundenauftrag abgeholt und wohin der Kundenauftrag transportiert werden soll	6.2	Muss	Ja
59	Der Anwender kann ablesen, welche Strecke abgefahren werden soll (FTF)	6.2	Muss	Ja
60	Der Anwender kann ablesen, dass ein Kundenauftrag zugestellt wurde (FTF)	6.1	Soll	Ja
61	Der Anwender kann ablesen, dass ein Kundenauftrag in Bearbeitung ist (Maschine)	6.4	Muss	Ja
62	Der Anwender kann ablesen, dass ein Kundenauftrag teilbearbeitet wurde (Maschine)	6.4	Muss	Ja
63	Der Anwender kann ablesen, dass ein Kundenauftrag fertiggestellt wurde (Maschine)	6.4	Muss	Ja
64	Der Bearbeitungszustand an den Maschinen soll angezeigt werden (Maschine)	6.4	Kann	Ja

Tabelle 3: Implementierte Anforderungen und in welchem Kapitel die Anforderungen behandelt werden

In Tabelle 3 sind alle Anforderungen zusammengetragen, die in dieser Arbeit erhoben wurden. In der Spalte „Kapitel“ kann abgelesen werden, in welchem Kapitel die Anforderung behandelt wird. In der Spalte „Gewicht“ ist die Gewichtung der Anforderungen eingetragen. Die Spalte „Impl.“ zeigt an, welche Anforderungen in der Simulation implementiert wurden und welche nicht. Des Weiteren wurden alle Anforderungen mit der Gewichtung „Muss“ hellblau markiert, weil sie unverzichtbar für die Realisierung der Simulation sind.

Aufgrund der Anforderungen mit der Gewichtung „Muss“ und der Analyse der Kommunikation und Abläufe des Planspiels ist eine Kommunikationsstruktur entstanden, die in Kapitel 4.4 in einer Swimlane festgehalten wurde. In der Swimlane kann jeder Schritt der Verhandlungen und der Abläufe der einzelnen Komponenten nachvollziehbar verfolgt werden. Zum Beweis wurde

in Kapitel 6 im praktischen Teil der Arbeit demonstriert, wie anhand von Szenarien die Verhandlungen stattfinden. Hierdurch die vorliegende Forschungsfrage beantwortet:

> Wie muss eine dezentrale Kommunikationsstruktur gestaltet werden, damit die Kundenaufträge so schnell wie möglich von einem verfügbaren FTF übernommen werden?

Bei der Entwicklung des Quellcodes für die Software wurde bewusst auf eine einfach nachvollziehbare Verbalisierung im Quellcode geachtet. Hierdurch soll die Erweiterungsmöglichkeit der Simulation und die Einarbeitung in die Software für dritte erleichtert werden.

Um die Verhandlungen visuell darzustellen, wurden im sechsten Kapitel zwei Szenarien vorgestellt. Für die Szenarien wurden Abbildungen dahingehend bearbeitet, dass eine übersichtliche Darstellung der Abläufe erkennbar ist. Dadurch wurde ein fließender Vorgang der Interaktion zwischen den Komponenten dargestellt.

Bei der Erfüllung der gesetzten Teilziele der vorliegenden Arbeit sind Aspekte aufgefallen, die die Umsetzung der Arbeit erschwerten. Bei der Analyse des Planspieles stellte sich heraus, dass die Durchführung des Planspieles durch eine einzelne Person erschwert wurde. In fortgeschrittenen Phasen des Planspieles vermehrten sich Daten und damit verbundene Tätigkeiten, die ein einzelner Spieler ausführen muss, wobei die Übersicht erschwert wurde. Infolgedessen wird die Einhaltung der Spielregeln beeinträchtigt, weil die Fehlerhäufigkeit bei allen Abläufen im Planspiel zunimmt. Um das Problem zu minimieren, wird empfohlen, das Planspiel mit mehreren Personen durchzuführen. Des Weiteren werden bei einem Spieler keine Verhandlungen vorgenommen, was grundlegend sowohl für das Planspiel als auch für die Erkenntnis einer dezentralen Kommunikation ist.

7.2 Ausblick

Im Rahmen einer künftigen Arbeit kann die Simulation vervollständigt werden. Hierfür kann eine Kommunikation zwischen den Maschinen auf Basis der in dieser Arbeit entwickelten Kommunikationsstruktur entstehen. Dabei ist festzuhalten, dass die entwickelte Kommunikationsstruktur als Referenz für weitere Projekte dienen kann. Für die Umsetzung der Projekte kann die zusammengestellte Plattform aus Kapitel 5.1 verwendet werden, die für erste Testbedingungen eingesetzt werden kann. Der Vorteil bei der Verwendung der Plattform ist, dass für einen Test eines Algorithmus keine zusätzliche Hardware, wie ein Arduino oder ein Raspberry Pi benötigt wird.

Weiterhin kann der Quellcode an einige Raspberry Pis angepasst werden, mit denen zunächst eine Kommunikation untereinander hergestellt wird. Der Vorteil besteht darin, dass die Simulation auf einer Linux-Distribution ausgeführt wird und ein Raspberry Pi Raspbian als Standard-Betriebssystem hat, das ebenfalls eine Linux-Distribution ist. Jeder Raspberry Pi kann anschließend als Basis für ein FTF verwendet werden.

Quellenangaben

Augsten, Stephan (2019): *Was ist eine UUID?,* [online: https://www.dev-insider.de/was-ist-eine-uuid-a-788491/, letzter Zugriff am 11.09.2020].

Brandt-Pook, Hans; Kollmeier, Rainer (2020): *Softwareentwicklung Kompakt und verständlich. Wie Softwaresysteme entstehen,* 3. Auflage, Wiesbaden: Springer Fachmedien.

Bubeck, Alexander; Gruhler, Matthias; Reiser, Ulrich; Weißhard, Florian (2014): Vom fahrerlosen Transportsystem zur intelligenten mobilen Automatisierungsplattform, in: Bauernhansl, Thomas; ten Hompel, Michael; Vogel-Heuser, Birgit (Hrsg), *Industrie 4.0 in der Produktion, Automatisierung und Logistik. Anwendung Technologien Migration,* Wiesbaden: Springer Vieweg, S. 221-233.

Both, David (2020): *Using and Administering Linux: Volume 1, Zero to SysAdmin: Getting Started,* NC: Springer Science+Business Media New York.

Eclipse Paho™ MQTT Python Client (2020)*,* [online: https://github.com/eclipse/paho.mqtt.python#more-information, letzter Zugriff am 25.09.2020].

Erben, Thomas (2017): *Einführung in Unix/Linux für Naturwissenschaftler.Effizientes wissenschaftliches Arbeiten mit der Unix-Kommandozeile,* Berlin: Springer-Verlag.

Getting Started Guide. VMware Player 3.0 (2009), [online: https://www.vmware.com/pdf/vmware_player310.pdf, letzter Zugriff am 01.09.2020].

Graphalgorithmen, [online: https://www-m9.ma.tum.de/Allgemeines/GraphAlgorithmen, letzter Zugriff am 10.04.2021].

Gärtner, Henner (2018): *Dezentrale Produktionssteuerung*, [online: https://emil.haw-hamburg.de/pluginfile.php/1759403/mod_resource/content/2/Produktionssteuerung-Digital_Auszug_SwimmingPool.pdf, letzter Zugriff am 11.11.2020].

Günthner , Wilibald A.; Chisu, Razvan; Kuzmany, Florian (2010): Die Vision vom Internet der Dinge, in: Günthner, Wilibald; ten Hompel, Michael (Hrsg.), *Internet der Dinge in der Intralogistik* , Berlin Heidelberg: Springer-Verlag, S. 43-46.

Kenngott,Christian (2001): *Virtuelle Maschinen mit erweiterbarem Befehlssatz*, [online: https://www.pervasive.jku.at/Research/Publications/_Documents/DissertationKenngott.pdf, letzter Zugriff am 16.04.2021].

Knoll, Thomas; Lautz, Alexander; Deuß, Nicolas (2020): Machine-To-Machine Communication. From Data to Intelligence, in: Michael ten Hompel, Thomas Bauernhansl, Birgit Vogel-Heuser (Hrsg.), *Handbuch Industrie 4.0*, Band 3: Logistik, 3. Auflage, Berlin: Springer Vieweg, S. 573-582.

Loos, Peter (2020): Fertigungssteuerung, in: *Enzyklopädie der Wirtschaftsinformatik Online-Lexikon,* [online: https://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/informationssysteme/Sektorspezifische-Anwendungssysteme/Produktionsplanungs--und--steuerungssystem/Fertigungssteuerung, letzter Zugriff am 24.04.2021].

Matthes, Eric (2017): *Python Crashkurs. Eine praktische, projektorientierte Programmiereinführung,* Heidelberg: dpunkt.verlag GmbH.

Michael ten Hompel, Thomas Bauernhansl, Birgit Vogel-Heuser (Hrsg.), *Handbuch Industrie 4.0*, Band 3: Logistik, 3. Auflage, Berlin: Springer Vieweg.

Plenk, Valentin (2017): *Angewandte Netzwerktechnik kompakt. Dateiformate, Übertragungsprotokolle und ihre Nutzung in Java-Applikationen,* Wiesbaden: Springer Vieweg.

Raschbichler, Florian (2017): *MQTT – Leitfaden zum Protokoll für das Internet der Dinge,* [online: https://www.informatik-aktuell.de/betrieb/netzwerke/mqtt-leitfaden-zum-protokoll-fuer-das-internet-der-dinge.html, letzter Zugriff am 10.01.2021].

Sinsel, Alexander (2020): *Das Internet der Dinge in der Produktion. Smart Manufacturing für Anwender und Lösungsanbieter,* Berlin: Springer-Verlag GmbH.

Swift, Nicholas (2017): *Easy A* (star) Pathfinding*, [online: https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2, letzter Zugriff am 10.10.2020].

Tenerowicz-Wirth, Peter (2013): *Kommunikationskonzept für selbssteuernde Fahrzeugkollektive in der Intralogistik,* in: Günthner, Willibald A. (Hrsg.), München: Technische Universität München.

Trojan, Walter (2017): *Das MQTT-Praxisbuch*, Aachen: Elektor-Verlag GmbH.

Tödter, Joachim; Abel, Bengt; König, Ralf; Schüthe, Dennis (2020): Flurförderzeuge für ein interaktives Zusammenspiel von Mensch und Maschine, in: Michael ten Hompel, Thomas

Bauernhansl, Birgit Vogel-Heuser (Hrsg.), *Handbuch Industrie 4.0*, Band 3: Logistik, 3. Auflage, Berlin: Springer Vieweg, S. 171-186.

Ullrich, Günter; Albrecht Thomas (2019): *Fahrerlose Transportsysteme. Eine Fibel – mit Praxisanwendungen – zur Technik – für die Planung*, 3.Auflage, Wiesbaden: Springer Vieweg.

VDI-Richtlinien 2510. Fahrerlose Transportsysteme (FTS) (2005), [online: https://www.vdi.de/richtlinien/details/vdi-2510-fahrerlose-transportsysteme-fts, letzter Zugriff am 14.04.21].

Velden, Lisa (2014): *Der A*-Algorithmus*, [online: https://www-m9.ma.tum.de/graph-algorithms/spp-a-star/index_de.html, letzter Zugriff am 10.04.2021], Nr.2.

Velden, Lisa (2014): *Der Dijkstra-Algorithmus*, [online: https://www-m9.ma.tum.de/graph-algorithms/spp-dijkstra/index_de.html, letzter Zugriff am 10.04.2021], Nr.1.

Weckmann Sebastian (2020): *Dezentrale, automatisierte und energieflexible Steuerung der Produktion*, Stuttgart: Frauenhofer Verlag.

Anhang A. Spielregeln Planspiel

1. Simulation einer dezentralen Produktionssteuerung

Beispiel: Schokoladenproduktion

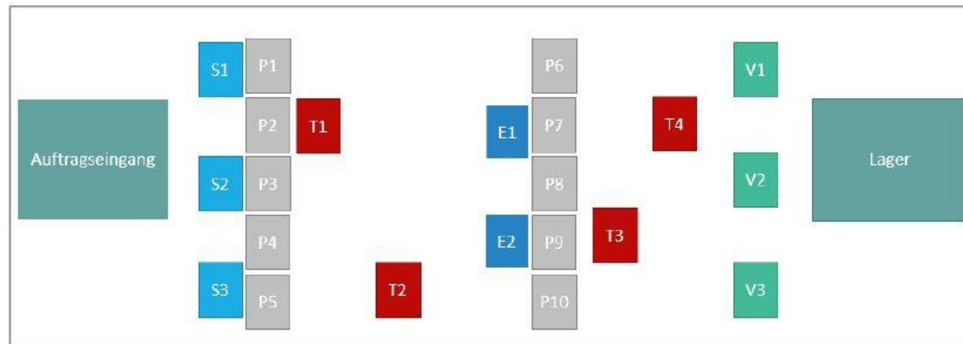
Am Beispiel einer Schokoladenproduktion soll mit unterschiedlichen Konfigurationen eine dezentralen Produktionssteuerung simuliert werden. Dabei sollen die Vor- und Nachteile der jeweiligen Konfiguration aufgezeigt werden.

Am Anfang der Produktionskette befindet sich der Auftragseingang. Die Aufträge stehen dort jeweils in einer Kiste mit den benötigten Informationen und Zutaten zur Abholung bereit. Die Aufträge werden von einem Transportfahrzeug abgeholt und zu einer freien Maschine der ersten Maschinengruppe (S1, S2, S3) transportiert. Über ein Netzwerk schickt der Auftrag zunächst eine Anfrage an die vier Transportfahrzeuge. Diese diskutieren nun untereinander wer die Anfrage bearbeitet. Entscheidend für die Auftragsvergabe ist in diesem Fall die Verfügbarkeit des Fahrzeugs und die Entfernung zum Auftrag. Als nächster Schritt/ Zum selben Zeitpunkt wird vom Transportfahrzeug bzw. Auftrag angefragt, welche der Maschinen S1, S2 und S3 zum Zeitpunkt der Ankunft des Auftrages als nächstes verfügbar ist.

Nach der Bearbeitung auf der ersten Maschinengruppe gibt es zwei Möglichkeiten für den Auftrag. Entweder holt ein verfügbares Transportfahrzeug den Auftrag direkt danach ab oder der Auftrag wird im Puffer (P1-P5) der ersten Maschinengruppe zwischengelagert.

Im nächsten Schritt muss der Auftrag, für den Fall der dieser Extras enthält, zur zweiten Maschinengruppe transportiert werden. Dies erfolgt ebenfalls auf Anfrage und anschließender „Verhandlung“. Enthält der Auftrag „Nuss+Frucht“ wird mit beiden Maschinen verhandelt und die Maschine angesteuert, die zuerst frei wird. Es befinden sich hinter der 2. Maschinengruppe ebenfalls 5 Puffer (P6-10).

Vom Puffer bzw. der zweiten Maschinengruppe muss der Auftrag nun zur 3. Maschinengruppe (V1, V2, V3) gefahren werden. Wie an den anderen Stationen zuvor geschieht die über eine Anfrage, die über Verhandlungen geregelt wird. Nach erfolgter Bearbeitung wird der Auftrag direkt ins Lager gefördert.



Eckdaten zu den einzelnen Komponenten:

Auftrag:
- Schokoladensorte (Zartbitter/ Vollmilch/ Weiß)
- Größe (100 g/ 300 g)
- Extras (kein/ Nuss/ Frucht/ „Nuss+Frucht")
- Zeitpunkt des Auftragseingangs (ZE x)
- Liefertermin (x ZE bis zur Auslieferung)
- Priorität (A-, B-, C-Kunde)
- Position auf dem Raster (x/x)
- Zeitstempel pro angesteuerte Station (Maschine, Puffer, Transportfahrzeug)

Maschinen:
- Status (Verfügbar/ Nicht verfügbar)
  Wenn die Maschine belegt ist:
- Informationen des zu produzierenden Auftrages
- Dauer der Maschinenbelegung (möglich Rüstzeit addieren)

Netzwerk:
- Datenübermittlung (Kommunikationstool für die Komponenten)
- Datensammlung pro Zeiteinheit

Transportfahrzeuge:
- Status (Verfügbar/ Nicht verfügbar)
  Wenn das Fahrzeug belegt ist:
- Informationen des zu transportierenden Auftrages
- Aktuelle Pos. Des Fahrzeugs
- Ziel (Pos.) des Fahrzeugs
- Dauer der Fahrzeugbelegung

Belegzeiten pro Auftrag:

S1	5 ZE	E1	4 ZE	V1	4 ZE	T1	1 Raster pro ZE
S2	7 ZE	E2	6 ZE	V2	4 ZE	T2	1 Raster pro ZE
S3	7 ZE			V3	8 ZE	T3	1 Raster pro ZE
						T4	1 Raster pro ZE

Parameter:

- Wenn alle Pufferplätze belegt sind, muss min. ein Platz frei geräumt werden bevor die erste Maschinengruppe weiter produzieren kann
- Transportzeit der Fahrzeuge 1 Raster pro ZE
- Einführen einer Rüstzeit (z.B. beim Wechsel von 100 g auf 300 g oder einer andere Schokoladensorte) pro Umrüstung 2 ZE
- In der zweiten Maschinengruppe „Extras“ kann E1 nur Nüsse und E2 nur Früchte hinzufügen
- Den Aufträgen sind je nach Kunde verschiedene Prioritäten zugeordnet (A-, Bund C-Kunde)

Regeln:

- Ein Auftrag mit dem Extra „Nuss+Frucht“ erhält an der zweiten Station in der zweiten Maschinengruppe den Vorrang vor allen anderen Aufträgen
- Ein Zufallsgenerator entscheidet, wenn zwei Aufträge die gleichen Voraussetzungen mitbringen
- Die Puffer werden nach FIFO/KOZ Regelung abgearbeitet
- Es befinden sich immer 10 Aufträge im Auftragseingang, die über Verhandlungen untereinander entscheiden wer zuerst bearbeitet wird. Geregelt wird das über die Kombination von verschiedenen vorgegebenen Merkmalen
- Es werden kontinuierlich Aufträge durch einen Zufallsgenerator in den Auftragseingang nachgeschoben
  - Kundenpriorität, minimale DLZ, Liefertermin, Schokoladengröße (in Bezug auf den Umsatz)
- Bei den Transportfahrzeugen wird die Auftragsannahme über die Entfernung bzw. voraussichtliche Ankunftszeit entschieden; jedes Fahrzeug wird angefragt und gibt eine voraussichtliche Ankunftszeit an
- Für das Be- und Entladen wird eine „Nullzeit“ berechnet

Spielanleitung: Dezentrale Schokoladenproduktion

Vorbereitung:

Anzahl der Mitspieler: 1-10 Spieler

Spielmaterial: 1 Spielbrett, 8 Stationen (braune Schachteln), 4 Transportfahrzeuge (blau-weiße Schachteln), 12 Puffer (Klammern), 30 Auftragskarten, 30 Notizkarten, 10 Infokarten, 1 Würfel, 2 Karten Ablageplätze (Lager, Auftragseingang)

Aufstellung: Stellen sie die Maschinen und Puffer auf die markierten Felder und verteilen sie die Transportfahrzeuge zufällig auf dem Spielfeld. Die Karten Ablageplätze müssen neben das Spielbrett gelegt werden. Anschließend werden die Auftragskarten gemischt und 10 Karten auf das Feld „Auftragseingang“ gelegt. Die restlichen Auftragskarten werden auf das Feld „Zufallsgenerator“ gelegt. Jeder Mitspieler benötigt mehrere Notizkarten und einen Stift und ggf. einen Blatt Papier für weitere Notizen.

Allgemeine Informationen: Ein Auftrag (Schokolade) unterscheidet sich durch die Schokoladensorte (Zartbitter/ Vollmilch/ Weiß), die Größe (100 g/ 300 g) und die Extras (kein/ Nuss/ Frucht/ „Nuss+Frucht“). An der ersten Maschinengruppe wird die Schokolade hergestellt, an der zweiten Maschinengruppe ggf. die Extras hinzugefügt. Hier ist die Besonderheit, dass „E1“ nur Nüsse und „E2“ nur Früchte hinzufügen kann. Die dritte Maschinengruppe ist für die Verpackung der Schokolade zuständig. Ebenfalls im Auftrag als Information enthalten sind die Kundenpriorität und den Liefertermin (mit dem Auftragseingang startet die Zeit für den Liefertermin; der Auftrag muss in x ZE erledigt sein). Diese dienen später zur Ermittlung der Priorität eines Auftrages.

Festlegung der Regeln für die Spielrunden: In der ersten Spielrunde werden im Puffer alle Aufträge nach dem Prinzip „First in, First out“ (FIFO) abgearbeitet. Für die zweite Spielrunde sollen die Spieler sich gemeinsam eine Regeländerung überlegen und diese schriftlich festhalten, um spätere Diskussionen zu vermeiden.
Beispiel: Änderung der Parameter zur Ermittlung der Priorität.

Spielverlauf

Aufgabe der Mitspieler: Wichtig für das gesamte Spiel ist die Einhaltung der dezentralen Produktionssteuerung des Systems. Allgemein zeichnen sich diese Systeme dadurch aus, dass Produktionsteilbereichen bzw. Produktionsstellen Entscheidungsbefugnisse zur eigenständigen Erfüllung bestimmter Funktionen übertragen werden.

Die Aufgabe jedes Spielers ist es für einen Auftrag bzw. mehrere Aufträge in einer Spielrunde die Verhandlungen in der dezentralen Schokoladenproduktion zu übernehmen. Für jeden Auftrag muss ab dem Auftragseingang eine Notizkarte für die spätere Auswertung der Spielrunden geführt werden. Der Spieler, der den Auftrag übernommen hat, muss die Notizkarte mit den Infos des Auftrages auszufüllen und ist nun für die Spielrunde für diesen Auftrag verantwortlich. Jeder Auftrag wird nach Erreichen einer neuen Station (Transportfahrzeug, Maschine etc.) mit einem „Zeitstempel“ versehen. Hierfür sind die Zeiteinheiten (ZE) auf den Notizkarten hinter der jeweiligen Station zu notieren.

Ein Spieler hat zusätzlich die Aufgabe die Zeiteinheiten zu zählen und in Form einer Strichliste für alle Mitspieler sichtbar zu notieren.

Ziel des Spiels: Auf spielerische Art soll die Vorgehensweise einer dezentralen Produktionssteuerung kennen gelernt werden und durch Regeländerungen und verschiedene Parametereinstellungen die Vor- und Nachteile aufgezeigt werden.
Ein Auftrag muss möglichst unter Einhaltung der Bedingungen produziert werden. Durch Transportfahrzeuge wird dieser an eine Maschine der jeweiligen Maschinengruppe transportiert und dort produziert. Nach Erreichen einer der Maschinen der dritten Maschinengruppe landet der Auftrag nach erfolgter Bearbeitung direkt im Lager. Wann welcher Auftrag bearbeitet wird und welche Maschinen bzw. welches Transportfahrzeug diese Aufgabe übernimmt, wird über Verhandlungen entschieden.

Beginn: Als Erstes werden die 10 Aufträge im Auftragseingang gleichmäßig auf die Mitspieler verteilt. Jeder Mitspieler ermittelt für „seine" Aufträge mit Hilfe der Infokarte die Punktzahl für die Priorität. Die Punktzahlen werden miteinander verglichen und der Auftrag mit der höchsten Punktzahl wird als erstes bearbeitet. Sollten mehrere Aufträge die gleiche Punktzahl haben, entscheidet der Zufall (Erläuterung siehe unter Besonderheiten).

Für die Bearbeitung an einer Station der ersten Maschinengruppe (S1, S2, S3) muss der Auftrag zunächst von einem der Transportfahrzeuge am Auftragseingang (Felder 4/1, 5/1, 6/1) abgeholt werden. Alle Mitspieler überlegen nun gemeinsam, welches der Fahrzeuge am schnellsten den Auftrag erreichen kann und welche der Maschinen (S1, S2, S3) den Auftrag bearbeiten kann. Da zu Beginn noch alle Maschinen frei sind, wird S1 aufgrund der kürzeren Bearbeitungszeit zuerst angesteuert.

Im nächsten Schritt wird mit den übrig gebliebenen Aufträgen wieder neu verhandeln. Sind die 4 Transportfahrzeuge belegt, startet die Zeitmessung. Die Fahrzeuge setzen sich nun pro Zeiteinheit auf dem Spielfeld in Bewegung. Da sich immer 10 Aufträge im Auftragseingang befinden müssen, legen die Spieler die übrig gebliebenen Aufträge (Verlierer der Verhandlung) nach einer Verhandlungsrunde wieder zurück in den Auftragseingang. Es muss anschließend ein neuer Auftrag aus dem Zufallsgerator genommen werden und in den Auftragseingang gelegt werden.

<u>Tipp</u>: Um den Überblick zu behalten, wann welche Maschine bzw. welches Transportfahrzeug wieder verfügbar ist, sollten die Spieler sich Notizen machen.
Beispiel: Der Spieler notiert sich, dass Transportfahrzeug x momentan auf dem Weg ist, um Auftrag y abzuholen und somit nicht verfügbar ist. Außerdem sollten die Aufträge (Auftragskarten) immer auf dem Spielfeld in der jeweiligen Station (Maschine, Transportfahrzeug oder Puffer) befinden.

Verlauf: Die Aufträge sind immer wieder mit den Maschinen und Transportfahrzeugen in Verhandlungen, wer den Vorzug erhält. Alle Spieler müssen einen Überblick haben, wann welche Maschinen und Transportfahrzeuge wieder zur Verfügung stehen. Bei den Maschinen S1, S2, S3 ist ggf. eine Rüstzeit zu berücksichtigen (siehe unter Besonderheiten).

Diese Verhandlungen werden bis zum Ende einer Spielrunde immer wiederholt. Im gesamten Spielverlauf muss die Zeit (ZE) beachtet und an den jeweiligen Stationen für die Aufträge notiert werden.

Ist ein Auftrag an einer Maschine bearbeitet worden, wird dieser auf einem der Plätze im Puffer zwischengelagert. In der ersten Spielrunde verhandelt aus dem Puffer einer Maschinengruppe nur der Auftrag, der als erstes im Puffer angekommen ist (Regel im Puffer: FIFO) mit den anderen Aufträgen (z.B. aus dem Auftragseingang).

Nach der Bearbeitung an der letzten Maschinengruppe (V1, V2, V3) wird der Auftrag direkt ins Lager geschoben und bleibt dort bis zum Ende einer Spielrunde liegen.

Besonderheiten: Sollten zwei Aufträge die gleiche Punktzahl bei den Verhandlungen erreichen, wird zufällig mit Hilfe eines Würfels ermittelt, welcher der beiden Aufträge den Vorzug erhält. Die höhere Augenzahl gewinnt, bei gleicher Augenzahl wird nochmal gewürfelt usw.

Bei der Berechnung der Priorität muss berücksichtigt werden, wie viel Zeit der Auftrag noch bis zum Liefertermin hat (Liefertermin + ZE des Auftragseingangs - Aktuelle ZE). Beispiel: Liefertermin: 90 ZE; Auftragseingang: nach 50 ZE; Aktuelle ZE: 100 ZE ▪ 90 ZE + 50 ZE – 100 ZE = noch 40 ZE bis zum Liefertermin.

Puffer und Maschinen müssen umfahren werden. Die Maschinen sind nur von einer Seite (diese ist mit einem Pfeil gekennzeichnet) für die Transportfahrzeuge zu erreichen. Das Be- und Entladen der Transportfahrzeugen wird mit 0 ZE berechnet.

<u>Belegzeiten in Zeiteinheiten (ZE) pro Auftrag:</u>

S1	5 ZE	E1	4 ZE	V1	4 ZE	T1	1 Raster pro ZE
S2	7 ZE	E2	6 ZE	V2	4 ZE	T2	1 Raster pro ZE
S3	7 ZE			V3	8 ZE	T3	1 Raster pro ZE
						T4	1 Raster pro ZE

Für den Fall, dass eine Maschine (S1, S2, S3) umgerüstet werden muss, wird jeweils 1 ZE für den Wechsel der Schokoladensorte und 1 ZE bei Änderung der Schokoladengröße zur Belegzeit der Maschine addiert.

Enthält ein Auftrag den Zusatz „Nuss+Frucht“, erhält dieser an der zweiten Station den Vorzug vor allen anderen Aufträgen.

Es kommt zum Ausfall einer Maschinengruppe, sollten alle Pufferplätze belegt sein. In diesem Fall muss mindestens ein Platz im Puffer frei geräumt werden, um weiter produzieren zu können.

Ende einer Spielrunde bzw. Ergebnis

Eine Spielrunde wird so lange gespielt bis 20 Aufträge das Lager erreicht haben. Dann wird eine weitere Spielrunde mit einer Regeländerung gestartet. Für einen Vergleich der verschiedenen Regeln bzw. Parametereinstellungen müssen mindestens zwei Spielrunden gespielt werden.

Beispiel: In der ersten Spielrunde wurden im Puffer alle Aufträge nach FIFO abgearbeitet. Als Regeländerung wird der Puffer in der zweiten Spielrunde nach dem Prinzip der kürzesten Operationszeit abgearbeitet.

Zur Veranschaulichung bzw. für spätere Diskussionen der verschiedenen Regeln können die Informationen, die während jeder Spielrunde auf den Notizkarten notiert wurden, genutzt werden, um z.B. Grafiken o.ä. zu erstellen. Fragen, wie z.B. „Wie wirken sich die Prioritäten auf das System aus?“ oder „Wie wirkt sich ein Regelwechsel von FIFO auf KOZ aus?“ könnten erörtert werden.

Anhang B. Installationsanleitung Software

1. Installation der Software VMware Workstation 16,0 Player

1. Als erstes muss die Software von der offiziellen Seite heruntergeladen werden. Hierfür muss man den folgenden Link öffnen: https://www.vmware.com/de/products/workstation-player/workstation-player-evaluation.html und den Butten „Jetzt herunterladen“ drücken (Bild 1).

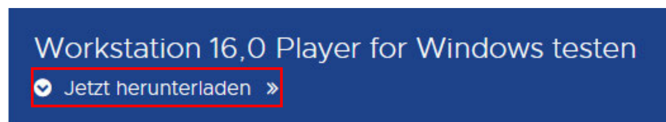


Bild 1: VMware Workstation runterladen

2. Nachdem VMware heruntergeladen wurde, in den Download-Ordner gehen und die Datei ausführen. Folglich erscheint das Installationsmenü von VMware (Bild 2). Hier auf „Next“ klicken.

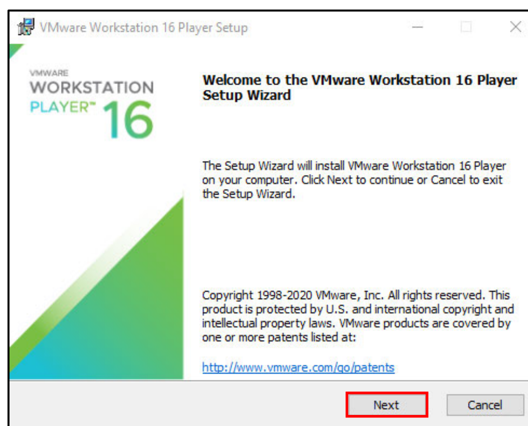


Bild 2: VMware Setup

3. Auf diesem Fenster ein Häkchen bei „I accept the terms in the License Agreement“ setzen und anschließend auf „Next“ klicken (Bild 3).

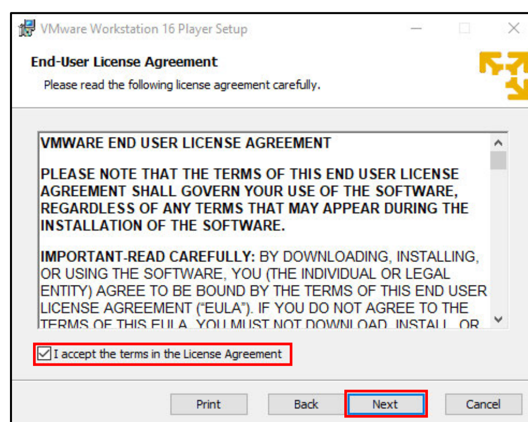


Bild 3: Lizenzvereinbarung

4. Hier kann der Ort ausgewählt werden, wo auf der Festplatte VMware gespeichert werden soll. Anschließend auf „Next“ klicken (Bild 4).

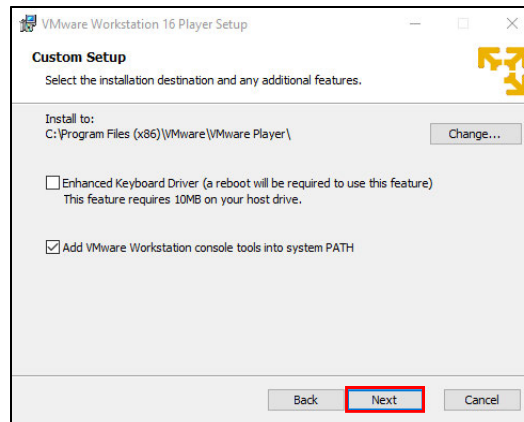


Bild 4: Speicherort aussuchen

5. Jetzt wird empfohlen, alles so zu lassen und auf „Next“ zu klicken (Bild 5).

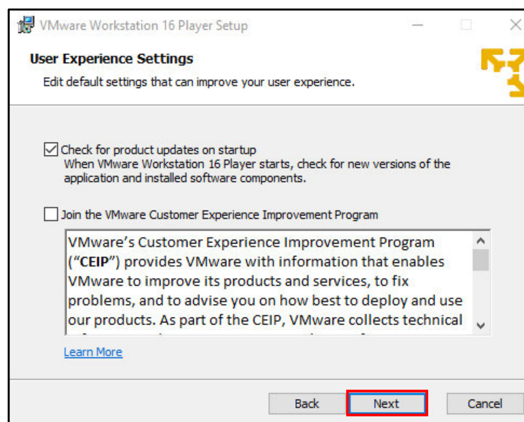


Bild 5: Freiwillige Teilnahme am Erfahrungsbericht

6. In diesem Schritt kann ausgewählt werden, ob das VMware-Symbol auf dem Desktop und im Startmenü nach der Installation eingeblendet werden soll (Bild 6).

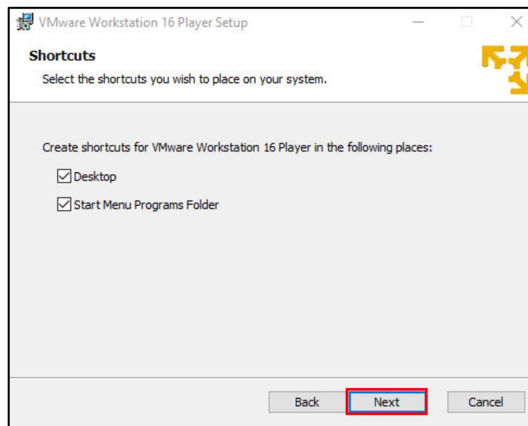


Bild 6: VMware Icon Auswahl

7. Im jetzt angezeigten Fenster auf „Install“ (Bild 7) klicken und die Installation von VMware beginnt (Bild 8). Anschließend auf „Finish“ klicken und die Installation ist abgeschlossen (Bild 9).

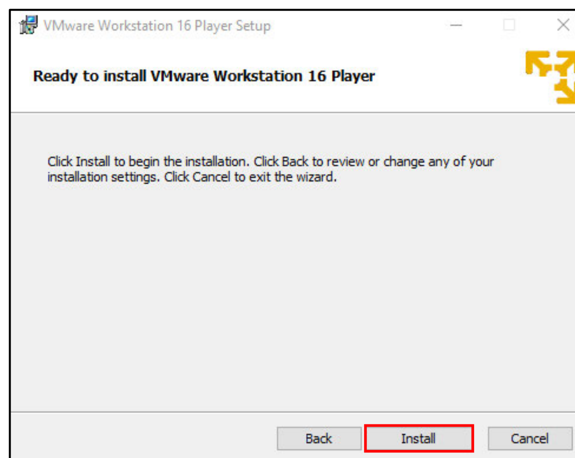


Bild 7: Bereit für die Installation

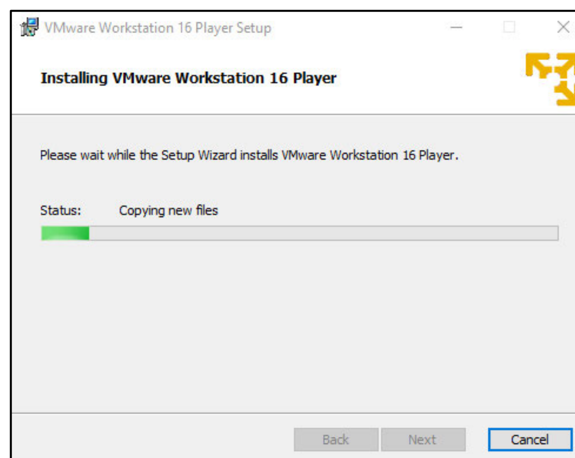


Bild 8: Die Installation von VMware beginnt

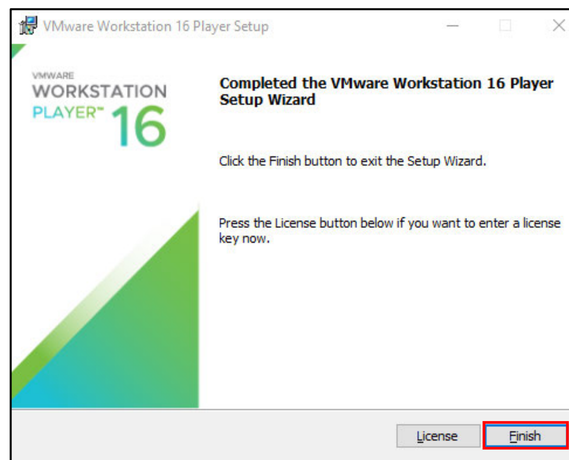


Bild 9: Installation abgeschlossen

8. Nachdem die Installation abgeschossen ist, muss die VMware-Software aufgerufen werden. Hierfür im Startmenü oder im Desktop auf das VMware-Symbol klicken (Bild 10).

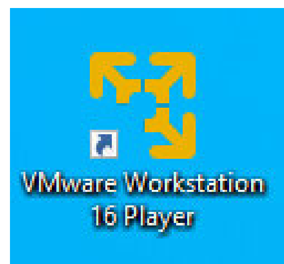


Bild 10: VMware Symbol

9. Im ersten Schritt die erste Option für die kostenlose Nutzung von VMware auswählen und auf „Continue“ klicken (Bild 11). Anschließend auf „Finish“ klicken (Bild 12).

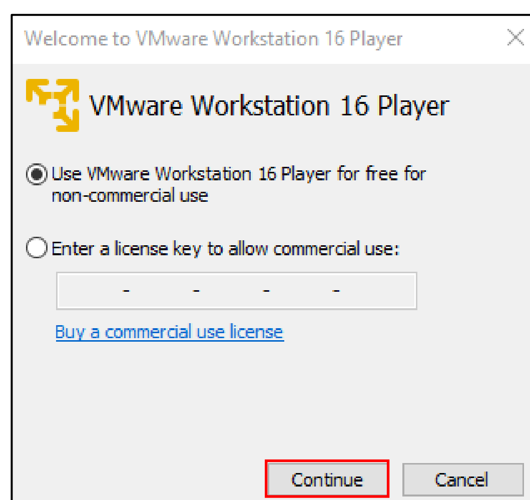


Bild 11: Kostenlose Nutzung von VMware

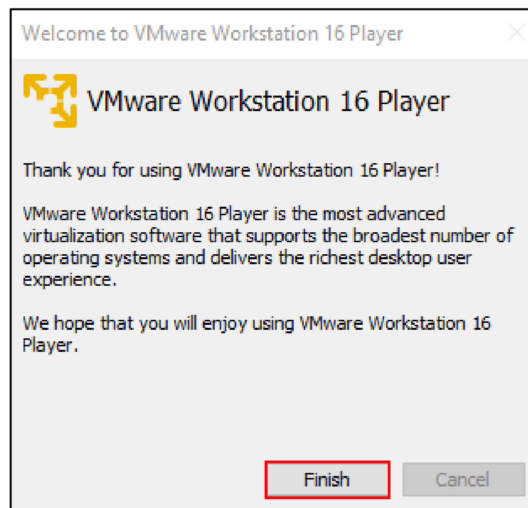


Bild 12: Kostenlose Benutzung akzeptiert

10. Im letzten Schritt wird die VMware Pro-Version angeboten, hier einfach auf „Skip this Version“ (Bild 13) klicken. VMware ist jetzt vollständig installiert und kann verwendet werden (Bild 14).

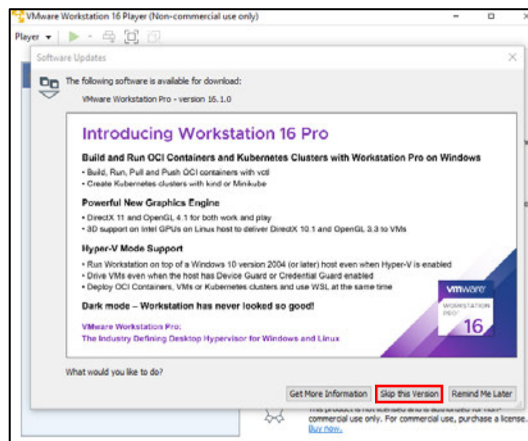


Bild 13: VMware Pro Version ablehnen

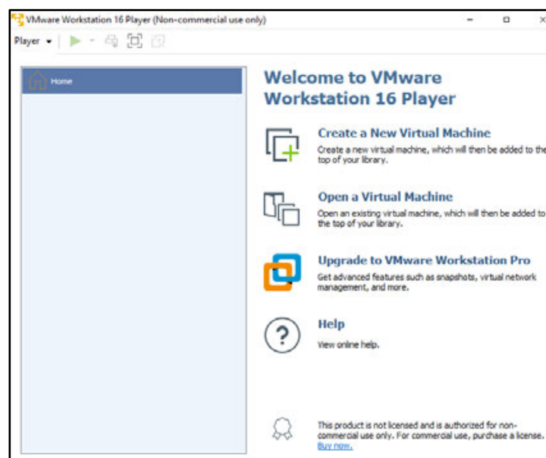


Bild 14: VMware ist bereit für den nächsten Schritt

2. Installation von Ubuntu

1. Als erstes muss die Software von der offiziellen Seite heruntergeladen werden. Hierfür den folgenden Link öffnen: https://ubuntu.com/download/desktop und die neueste Version herunterladen. Aktuell ist das die Version 20.04.1. Hierfür auf Download Klicken (Bild 15).

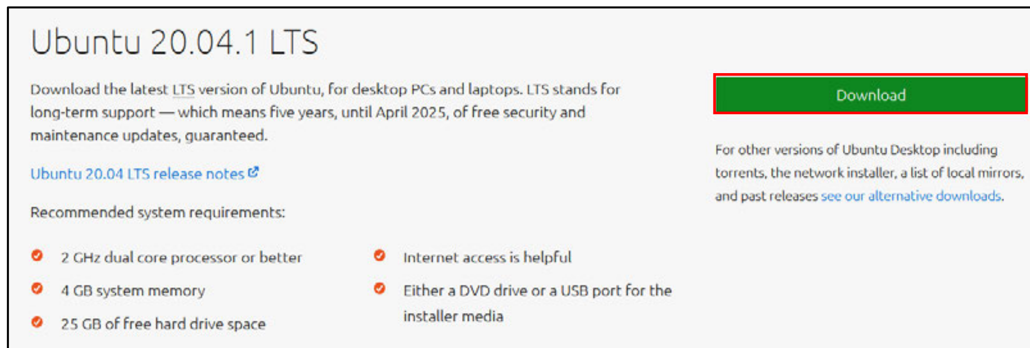


Bild 15: Ubuntu Download

2. Nachdem der Download abgeschlossen ist, muss die VMware Software geöffnet werden. Anschließend auf „Create a New Virtual Machine“ klicken (Bild 16).

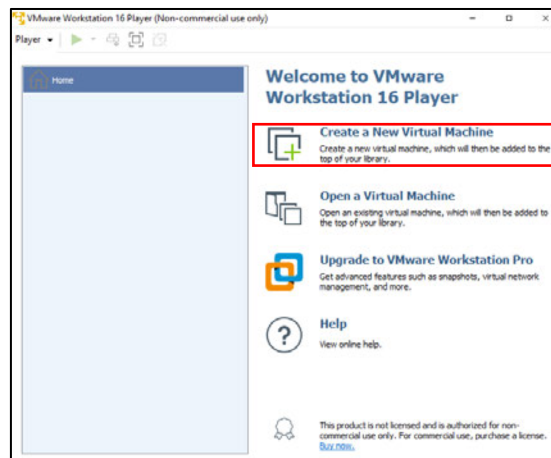


Bild 16: VMware Software aufrufen

3. „I will install the operating system later“ auswählen und anschließend auf „Next“ klicken (Bild 17).

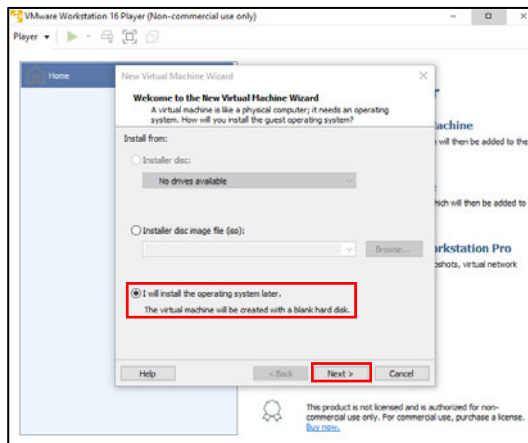


Bild 17: Neue virtuelle Maschine erstellen

4. In diesem Schritt muss als erstes Linux ausgewählt werden und anschließend die Ubuntu 64-Bit-Version. Anschließen auf „Next“ klicken (Bild 18).

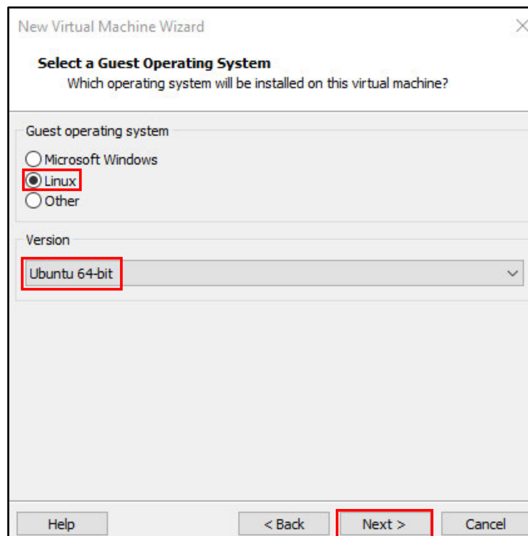


Bild 18: Betriebssystem auswählen

5. Hier kann ein Name für die virtuelle Maschine vergeben werden. Außerdem kann der Ort ausgewählt werden, an dem die virtuelle Maschine gespeichert werden soll. Anschließend auf „Next“ klicken (Bild 19).

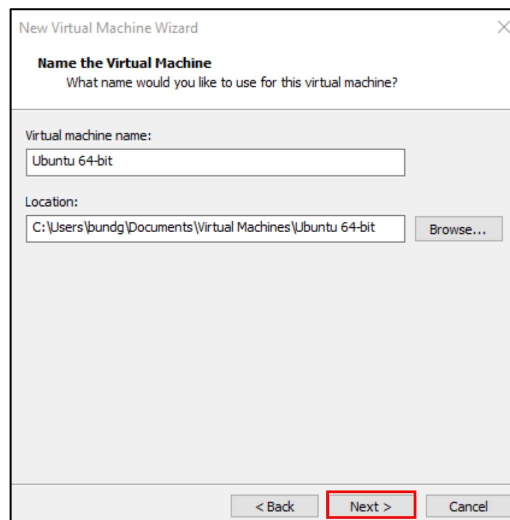


Bild 19: Name und Speichertort der Virtual Machine

6. In diesem Schritt kann die Festplattengröße bestimmt werden, die der virtuellen Maschine zur Verfügung gestellt werden soll. Anschließend auf „Next“ klicken (Bild 20).

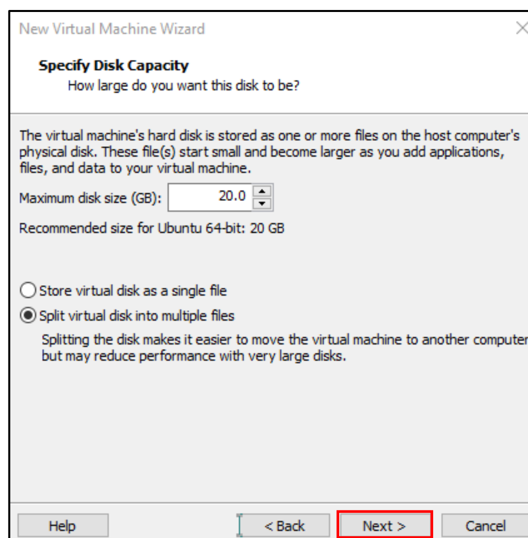


Bild 20: Auswahl des Festplattenspeichers für die Virtual Machine

7. VMware bietet in diesem Schritt die Möglichkeit der Virtual Machine, Hardware-Ressourcen zu Verfügung zu stellen. Dafür auf die Schaltfläche „Customize Hardware“ klicken (Bild 21).

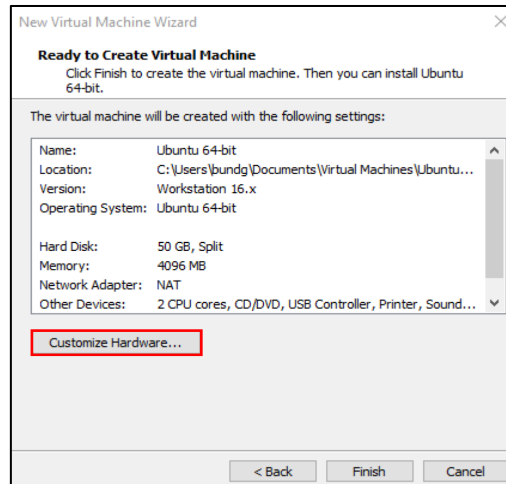


Bild 21: Benutzerdefinierte Hardware

8. In der ersten Option kann eingestellt werden, wie viel Arbeitsspeicher maximal verwendet werden darf (Bild 22).

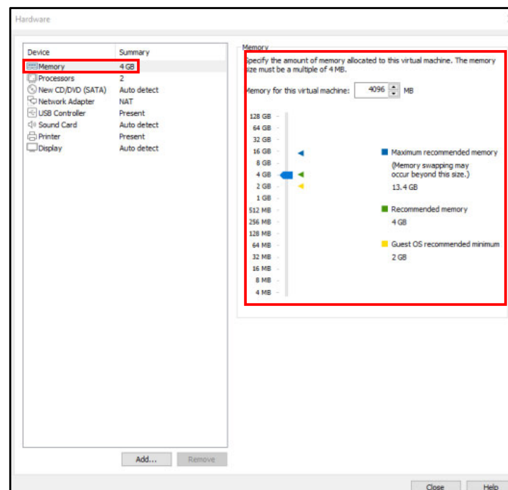


Bild 22: Auswahl der Größe des Arbeitsspeichers

9. In diesem Schritt kann eingestellt werden, wie viele Kerne die Virtual Machine verwenden darf (Bild 23).

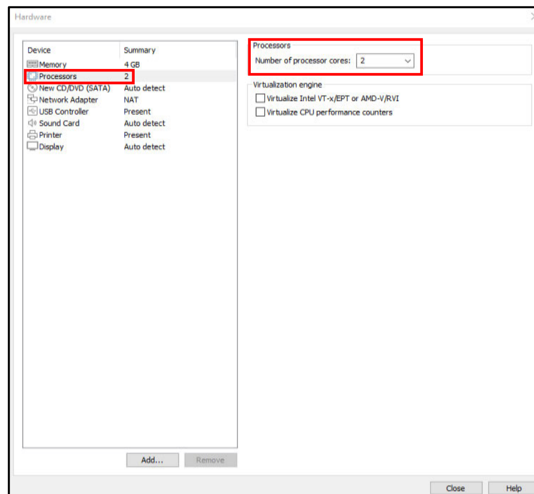


Bild 23: Auswahl der Anzahl der Prozessor Kerne

10. In dieser Option wird die Ubuntu Datei, die im ersten Schritt heruntergeladen wurde, eingebunden. Dafür auf die Option „Use ISO image file“ auswählen und unter Browse, die Ubuntu-20.04.01 Datei suchen, die auf der Festplatte hinterlegt wurde. Anschließend auf „Close“ (Bild 24) und auf „Finish“ klicken (Bild 25). Die Virtual Machine wurde jetzt erstellt und ist im VMware-Menü sichtbar (Bild 26).

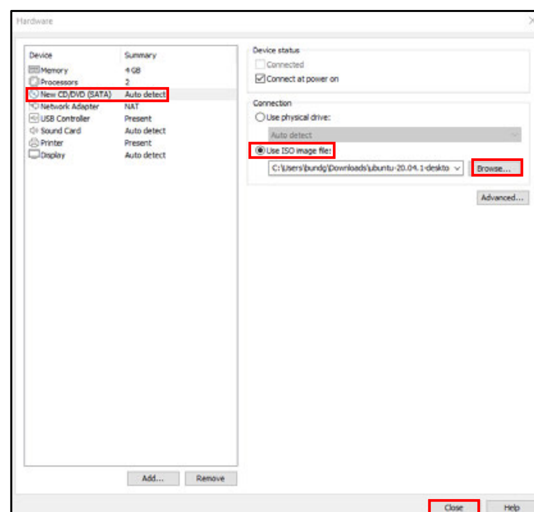


Bild 24: Ubuntu Dateien einbinden

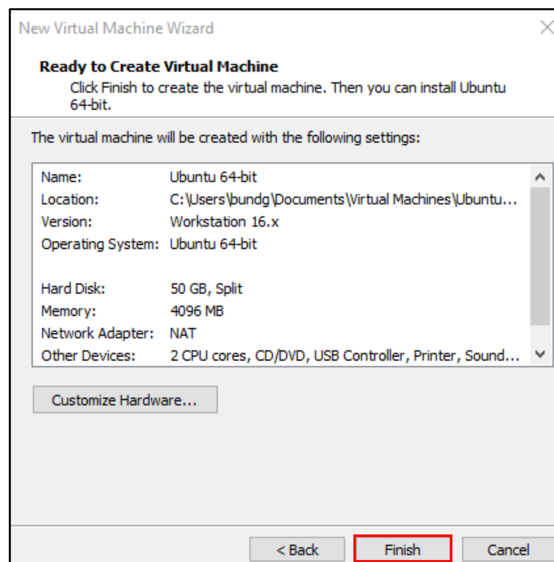


Bild 25: Virtual Machine erstellen

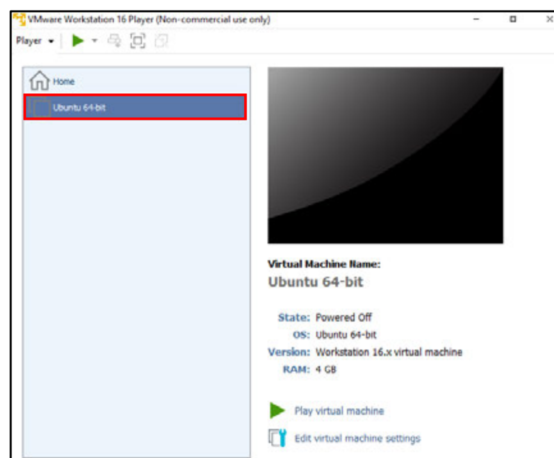


Bild 26: Virtual Machine kann jetzt ausgewählt werden

11. Um die eben erstellte Virtual Machine zu starten, gibt es zwei Optionen. Entweder direkt mit der linken Maustaste auf das Symbol „Ubuntu 64-bit“, oder auf „Play virtual machine“ klicken. Zudem gibt es die Möglichkeit, jederzeit die Schritte 8 und 9 zu wiederholen. Hierfür auf „Edit virtual machine settings“ klicken (Bild 27).

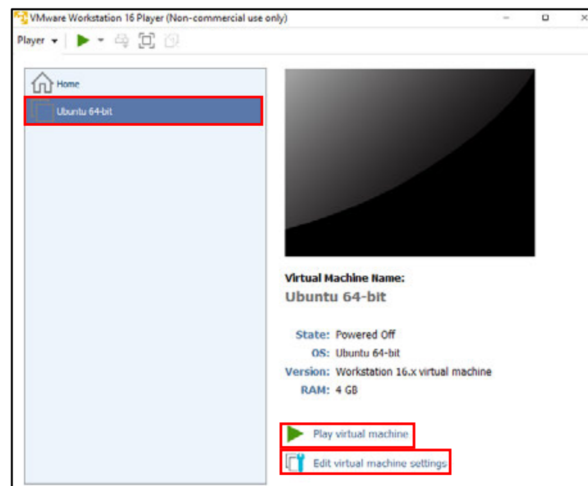


Bild 27: Virtual Machine starten

12. Nachdem die Virtual Machine gestartet wurde, kann es vorkommen, dass eine Fehlermeldung erscheint (Bild 28). Um diesen Fehler zu beheben, muss das SVM Mode im Bios auf enable gesetzt werden (Bild 29). Anschließend auf „OK" klicken, den Rechner neu starten und in die Bios-Option mit der Taste „entf" auswählen. Im Bios SVM Mode auf enable stellen und speichern. Nachdem der Computer neu gestartet wurde, wieder die VMware-Software öffnen. Anschließend Schritt 11 nochmal ausführen.

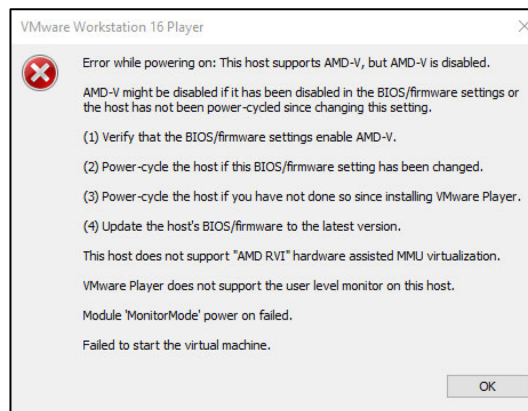


Bild 28: Fehlermeldung AMD-V

Bild 29: SVM Mode im Bios

13. Jetzt beginnt die Installation von Ubuntu. Folglich erscheint zusätzlich ein Fenster, in welchem angeboten wird, ein Software-Update zu machen. Hier „Download and Install“ klicken (Bild 30).

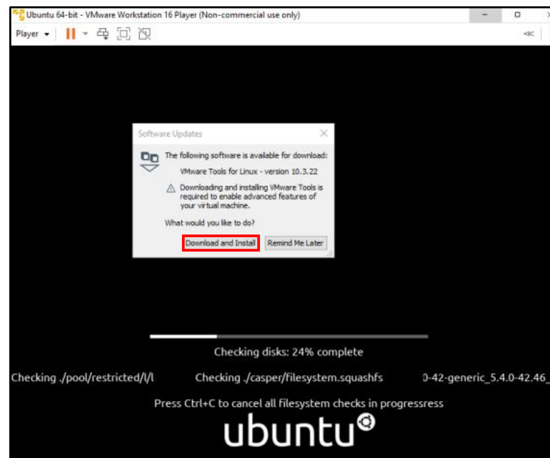


Bild 30: Beginn der Installation von Ubuntu

14. Die Sprache auswählen und auf „Install Ubuntu“ klicken (Bild 31).

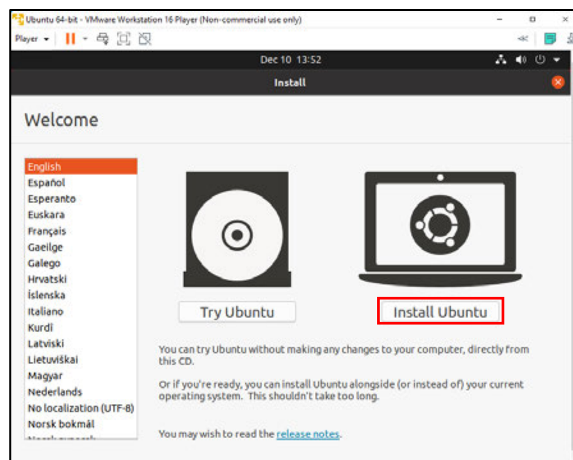


Bild 31: Sprache auswählen

15. Das Tastaturlayout festlegen und auf „Continue“ klicken (Bild 32).

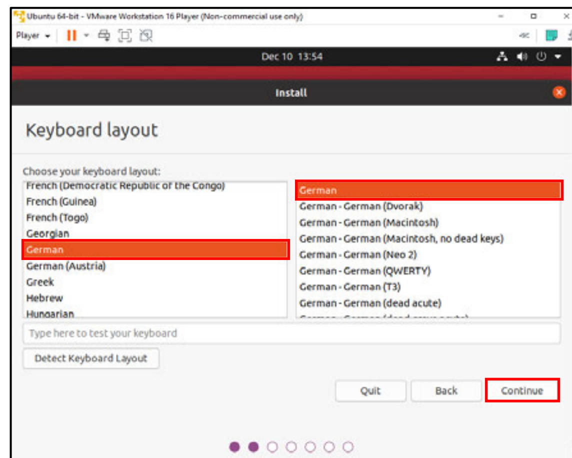


Bild 32: Tastaturlayout

16. In diesen Schritt ein Häkchen bei „1“ setzten und auf „Continue“ klicken (Bild 33).

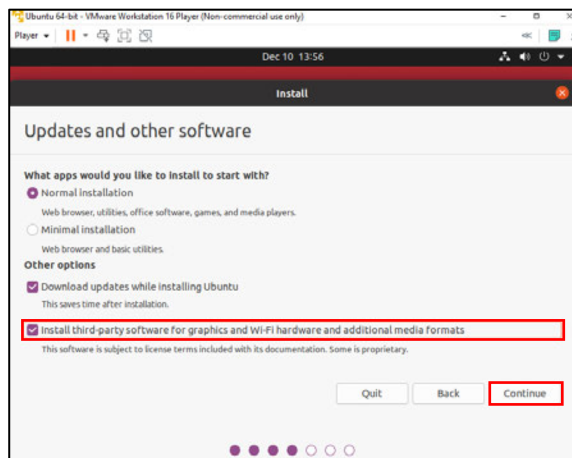


Bild 33: Updates an other software

17. Hier alles so lassen, wie es ist und auf „Install Now“ klicken (Bild 34). Anschließend auf „Continue“ klicken (Bild 35).

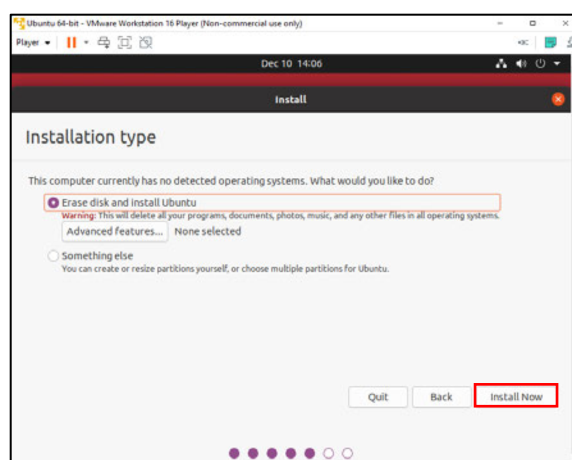


Bild 34: Installation type

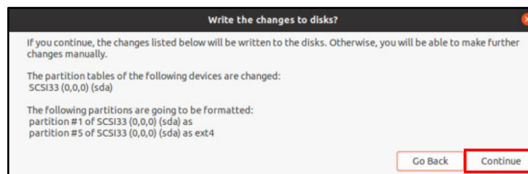


Bild 35: Write the changes to disk

18. Zeitzone auswählen und auf „Continue“ klicken (Bild 36).

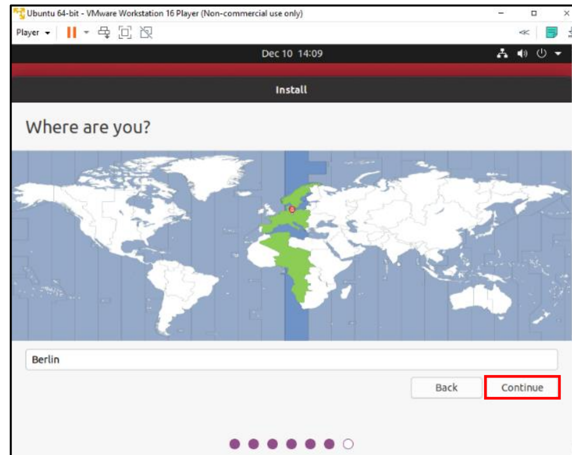


Bild 36: Zeitzone auswählen

19. In diesem Schritt kann ein Name/Username und ein Passwort für den Zugang festgelegt werden. Anschließend auf „Continue“ klicken (Bild 37).

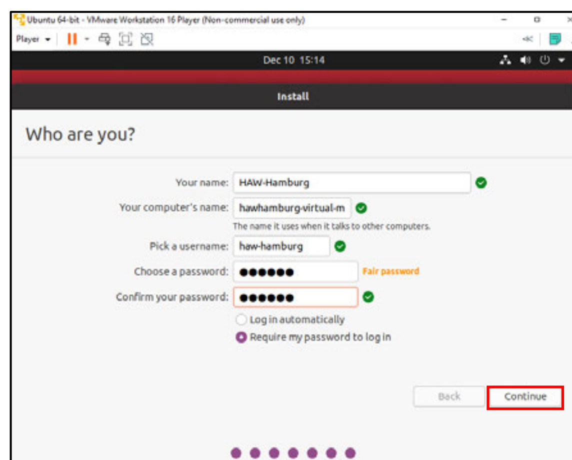


Bild 37: Name und Passwort festlegen

20. Ubuntu wird jetzt installiert (Bild 38).

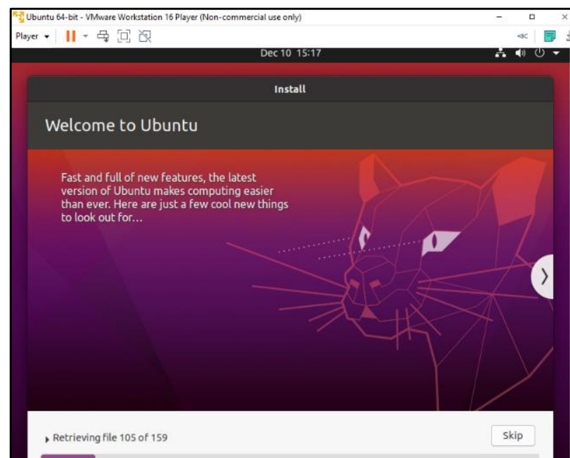


Bild 38: Ubuntu wird installiert

21. Nachdem die Installation abgeschlossen ist, wird geraten, die Virtual Machine neu zu starten. In diesem Fall auf „Restart Now“ klicken (Bild 39).

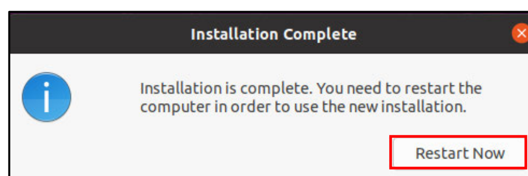


Bild 39: Installation abgeschlossen

22. Nachdem der Neustart durchgeführt wurde, erscheint der Sperrbildschirm. Indem man auf das Profil (HAW-Hamburg) klickt und im Anschluss das festgelegte Passwort eingibt, gelangt man auf den Desktop von Linux. Linux kann ab jetzt genutzt werden (Bild 40).

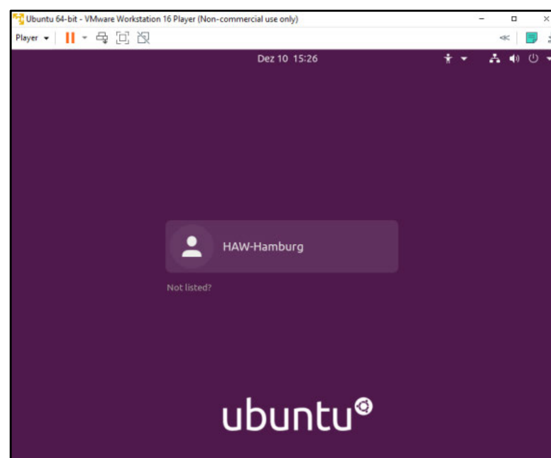


Bild 40: Sperrbildschirm Linux

3. Installation des Programmes Terminator

1. Um Terminator oder auch andere Programme zu installieren, gibt es zwei Möglichkeiten. Zum einen über den Softwaremanager oder über das Terminal. In dieser Anleitung wird nur die Möglichkeit über den Softwaremanager beschrieben. Hierfür als erstes die Windows-Taste drücken und in der Suchleiste „Ubuntu Software“ eingeben. Anschließend auf das Symbol klicken (Bild 41).

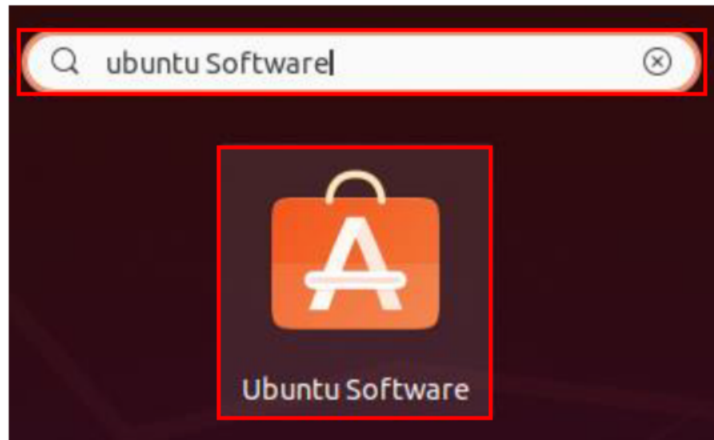


Bild 41: Softwaremanager

2. Oben in der Suchleiste „Terminator“ eingeben und „Terminator“ anklicken (Bild 42).

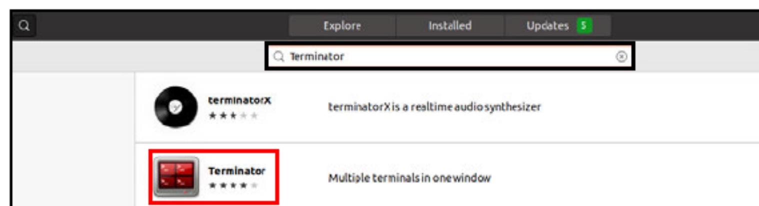


Bild 42: Suche nach Terminator

3. In diesem Schritt auf „Install“ klicken (Bild 43).

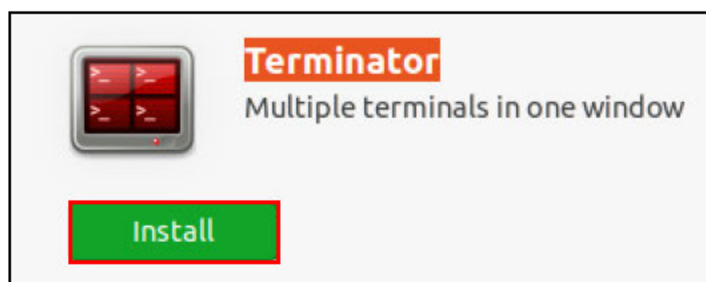


Bild 43: Terminator installieren

4. Die zweite Möglichkeit ist es folgenden befehl in einem Terminal einzugeben. Hierfür muss zuerst ein Terminal geöffnet. Mit einem Rechtsklick auf die Maus auf eine freie Fläche auf dem Desktop klicken und im Menü auf „Im Terminal öffnen“ drücken. Anschließend folgenden Befehl eingeben: „sudo apt-get install terminator“ und mit Taste „Enter“ bestätigen.

5. Nachdem Terminator installiert ist, die Windowstaste drücken und in die Suchfunktion Terminator eingeben und auf das Symbol klicken (Bild 44).

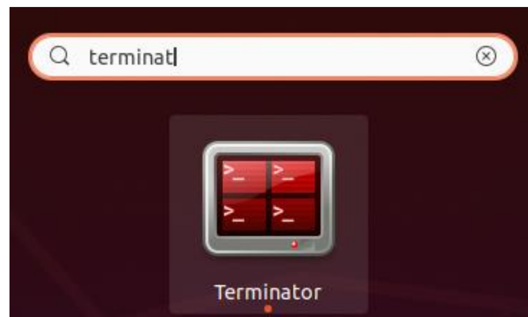


Bild 44: Terminator starten

6. Auf Abbildung 44 ist das Terminal des Programmes Terminator zu sehen. Indem man im Terminal mit der Maus auf die rechte Taste drückt, erscheint ein Auswahlmenü, in dem das Terminal entweder vertikal oder horizontal in zwei Terminale teilen kann. In Abbildung 45 wurde Terminator vertikal geteilt.

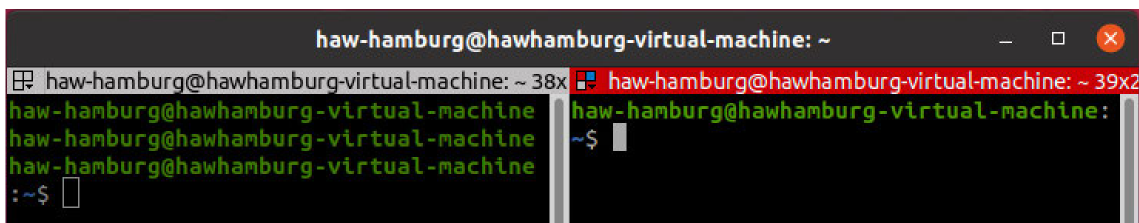


Bild 45: Terminator wurde vertikal in zwei Terminale geteilt

4. Installation von Sublime Text

1. Die Installation von Sublime Text verläuft genauso wie die von Terminator (Schritt 1, 2 und 3). Sublime Text In den Softwaremanager eingegeben und das Programm ausgewählen (Bild 46). Anschließend auf „Install“ klicken. Anschließend in der Ubuntu Suchfunktion „Sublime Text“ eingeben und das Programm ausführen. Sublime Text kann jetzt verwendet werden (Bild 47).

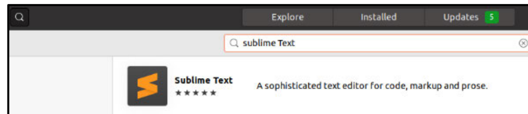


Bild 46: Suche nach Sublime Text

Bild 47: Sublime Text installieren

2. Die zweite Möglichkeit ist es folgenden befehl in einem Terminal einzugeben. Hierfür muss zuerst ein Terminal geöffnet. Mit Rechtsklick auf die Maus auf eine freie Fläche auf dem Desktop klicken und im Menü auf „Im Terminal öffnen“ drücken. Anschließend folgende Befehle eingeben und mit „Enter“ bestätigen:

2.1. wget -qO - https://download.sublimetext.com/sublimehq-pub.gpg | sudo apt-key add –
2.2. echo "deb https://download.sublimetext.com/ apt/stable/" | sudo tee /etc/apt/sources.list.d/sublime-text.list
2.3. sudo apt-get update
2.4. sudo apt-get install sublime-text

3. Nachdem Sublime Text installiert ist, die Windowstaste drücken und in die Suchfunktion Sublime Text eingeben und auf das Symbol klicken (Bild 48).

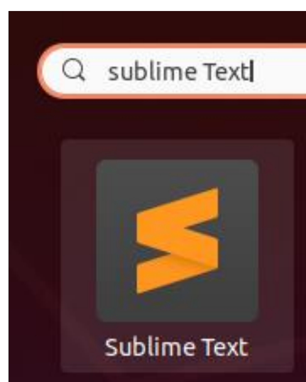


Bild 48: Sublime Text symbol

5. Installation von Python3

Python3 ist normalerweise schon in Ubuntu installiert. Um sicher zu gehen, den folgenden Befehl in ein Terminal eingeben:

python3 -version

Durch die Eingabe des Befehls wird die aktuelle Version von python3 angezeigt. Sollte python3 nicht installiert sein, kann durch folgenden Befehl python3 installiert werden:

Sudo apt install python3

Anschließend das Password des Admin eingeben und danach mit „y“ bestätigen.

6. Installation von Bibliotheken für die Simulation.

Als erstes wird Paho-MQTT installiert. Hiermit ist es möglich MQTT zu benutzen. Hierfür folgenden Befehl eingeben:

pip3 install paho-mqtt

Anschließend wird Mosquitto installiert. apt-get update && apt-get upgrade

sudo apt-get install mosquitto

Als nächsten folgt die Installation von einer Bibliothek, die es ermöglicht Ausgaben in Farbe, das Datum und die Uhrzeit auszugeben. Für die Installation folgenden Befehl eingeben:

pip3 install coloredlogs

Die nächste Bibliothek ermöglicht die Verwendung von Ladebalken. Hierfür den folgenden Befehl eingeben:

pip3 install progress

Hiermit ist die Installation abgeschlossen.

7 Vorbereitung für die Simulation

Für die Ausführung der Simulation sollten alle Dateien in sich in einem Ordner befinden. Folgende Dateien werden für den Start der Simulation benötigt.

mqtt-car.py
mqtt-order.py
mqtt-station.py
astar.py

Als erstes wird der Terminator aufgerufen und die benötigte Anzahl an Terminals geöffnet. Anschließend muss in jedem Terminal die Umgebung eingegeben werden, in der sich alle Dateien befinden. In diesem Beispiel befinden sich alle Dateien auf dem Schreibtisch weshalb der der folgende Befehl „cd Schreibtisch“ eingegeben wird und mit der „Enter“ Taste bestätigt (Bild 49, Punkt 1,2,3).

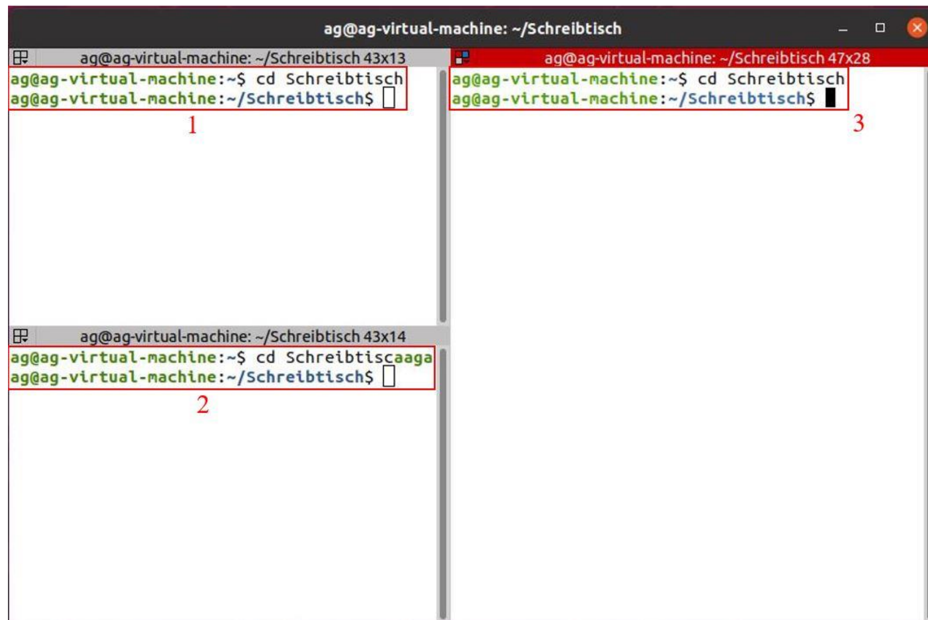


Bild 49: Verzeichnisauswahl Schreibtisch

Nachdem sich alle Terminale im Verzeichnis befinden, wo die Dateien gespeichert sind, können die Dateien als nächstes aktiviert werden. In Abbildung 50 ist zu sehen, wie der Dateiaufruf ausgeführt werden muss. Im ersten Schritt wird die Datei „mqtt-station.py“ aufgerufen, wodurch die Maschinen aktiviert werden (Punkt 1). Im zweiten Schritt wird die Datei „mqtt-car.py“ aufgerufen, wodurch ein FTF aktiviert wird (Punkt 2). Im letzten Schritt wird die Datei „mqtt-order.py“ aufgerufen, die einen Kundenauftrag darstellt und wodurch die Simulation ausgelöst wird (Punkt 3).

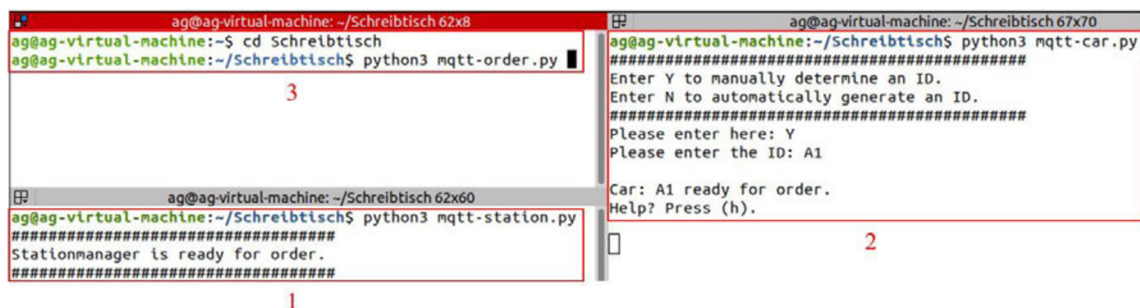


Bild 50: Aufrufen der Komponenten FTF, Maschine und Kundenauftrag

Hiermit ist die Vorbereitung abgeschlossen.

Anhang C. Quellcode Simulation

```
import paho.mqtt.client as mqtt

import astar as astar
import NonBlockingConsole as console
import uuid
import json
import time
import random
import copy
import datetime
import coloredlogs, logging
from progress.bar import FillingSquaresBar, FillingCirclesBar
from pprint import pformat

# Farbwiedergabe im Terminal.
class color:
   PURPLE = '\033[95m'
   CYAN = '\033[96m'
   DARKCYAN = '\033[36m'
   BLUE = '\033[94m'
   GREEN = '\033[92m'
   YELLOW = '\033[93m'
   RED = '\033[91m'
   BOLD = '\033[1m'
   UNDERLINE = '\033[4m'
   END = '\033[0m'

# Farbwiedergabe und Uhrzeit im Terminal.
logger = logging.getLogger(__name__)
logging.basicConfig(
        format='%(asctime)s %(levelname)-8s %(message)s',
        datefmt='%H:%M:%S')
```

```
coloredlogs.install(level='DEBUG')

#logger.debug("this is a debugging message")          #green
#logger.info("this is an informational message")     #white
#logger.warning("this is a warning message")          #yellow
#logger.error("this is an error message")             #red
#logger.critical("this is a critical message")       #red-thick

# Produktionslinie für den A* Algorithmus.
#       0  1  2  3  4  5  6  7  8
maze = [[0, 1, 0, 0, 1, 0, 0, 1, 0,], #0
        [0, 0, 0, 0, 0, 0, 0, 0, 0,], #1
        [0, 0, 0, 0, 0, 0, 0, 0, 0,], #2
        [0, 0, 0, 0, 0, 0, 0, 0, 0,], #3
        [0, 0, 1, 1, 1, 1, 1, 0, 0,], #4
        [0, 0, 1, 0, 0, 0, 1, 0, 0,], #5
        [0, 0, 0, 0, 0, 0, 0, 0, 0,], #6
        [0, 0, 0, 0, 0, 0, 0, 0, 0,], #7
        [0, 0, 0, 0, 0, 0, 0, 0, 0,], #8
        [0, 1, 1, 1, 1, 1, 1, 1, 0,], #9
        [0, 1, 0, 0, 1, 0, 0, 1, 0,], #10
        [0, 0, 0, 0, 0, 0, 0, 0, 0,], #11
        [0, 0, 0, 0, 0, 0, 0, 0, 0,]] #12

# Alle Maschinengruppen + Auftragseingang.
stationen = {
            'S1':{'pos':[11,1], 'busy': False,},
            'S2':{'pos':[11,4], 'busy': False,},
            'S3':{'pos':[11,7], 'busy': False,},
            'E1':{'pos':[6,2], 'busy': False,},
            'E2':{'pos':[6,6], 'busy': False,},
            'A':{'pos':[12,4], 'busy': False,}
}
```

```
# Puffer der Maschinengruppe S.
bufferS = {
        'SB1':{'pos':[8,1], 'busy': False,},
        'SB2':{'pos':[8,2], 'busy': False,},
        'SB3':{'pos':[8,3], 'busy': False,},
        'SB4':{'pos':[8,4], 'busy': False,},
        'SB5':{'pos':[8,5], 'busy': False,},
        'SB6':{'pos':[8,6], 'busy': False,},
        'SB7':{'pos':[8,7], 'busy': False,},
}

# Puffer der Maschinengruppe E.
bufferE = {
        'EB1':{'pos':[3,2], 'busy': False,},
        'EB2':{'pos':[3,3], 'busy': False,},
        'EB3':{'pos':[3,4], 'busy': False,},
        'EB4':{'pos':[3,5], 'busy': False,},
        'EB5':{'pos':[3,6], 'busy': False,},
}

# Ausgänge.
outputs = {
        'V1':{'pos':[1,1], 'busy': False, 'worktime': 0.1},
        'V2':{'pos':[1,4], 'busy': False, 'worktime': 0.1},
        'V3':{'pos':[1,7], 'busy': False, 'worktime': 0.1},
}

action_time = 1                 #Zeiteinheit für die Fahrzeuggeschwindigkeit.
completed_orders = {}
delivered_orders = []

car_order = {}
```

```
car_dice = {}
my_order = ''
refused_orders = {}

is_busy = False
car_pos = (0,0)

# Batteriekapazität.
battery = 100

# 1.1 Hier wird ein neues Transportfahrzeug gespeichert und die eigene ID gesendet.
def on_car_init(mqttc, obj, msg):
        car = json.loads(msg.payload)
        if car['id'] not in cars_list:
                # Mit der Methode "append" wird eine neue "car_id" in die Liste cars_list hin-
zugefügt.
                cars_list.append(car['id'])
                print('\nCars list' + str(cars_list))
        msg = {'id':car_id,}
        mqttc.publish("car-init-response", json.dumps(msg), qos=0)

# 1.2 Vorhandene Fahrzeugliste wird verglichen und aktualisiert.
def on_car_init_response(mqttc, obj, msg):
        car = json.loads(msg.payload)
        if car['id'] not in cars_list:
                # Mit der Methode "append" wird eine neue car_id in die Liste cars_list hinzu-
gefügt.
                cars_list.append(car['id'])
                print('\nCAR: car-list-response' + str(cars_list))

# 2.1 Neuer Kundenauftrag wird überprüft und die Daten wie z.B. die eigene Strecke gesam-
melt.
def on_new_order(mqttc, obj, msg):
        order = json.loads(msg.payload)
```

```
        logger.warning('New order: ' +str(order))
        global is_busy
        if not is_busy:
                global car_pos
                # Weg wird von der aktuellen Position des Transportfahrzeuges bis zur aktuel-
len Position des Kundenauftrages berechnet.
                path = get_path(car_pos, tuple(stationen[order['pos']]['pos']))
                #  Die Funktion share my path wird aufgerufen.
                # 2.2 Die eigene Strecke wird mit den anderen Transportfahrzeugen geteilt.
                share_my_path(order, path)
        else:
                logger.critical('order refused: ' +str(order))
                msg = {'car_id': car_id, 'order': order}
                mqttc.publish("order-refused", json.dumps(msg), qos=0)
                print('I am Busy.')


# Hier wird der A* Algorithmus ausgeführt und die Strecke kalkuliert.
# Start-, Endkoordinaten.
def get_path(start, end):
        # Berechnung der kürzesten Strecke in Betrachtung der vorhandenen Hindernisse.
        path = astar.astar(maze, start, end)
        # Die Funktion "print_path" zeichnet die Produktionslinie, sowie die Strecke für ein
FTF, die berechnet wurde.
        print_path(path)
        return path


def print_path(path):# A*Funktion: Hier wird die Produktionslinie im Terminal gezeichnet.
        # Es wird eine Kopie der Referenz Produktionslinie erzeugt,
        # damit das Original nicht überschrieben wird.
        temp_maze = copy.deepcopy(maze)
        # In diesem Schritt wird die Strecke in die Produktionslinie hinzugefügt.
        # Hierfür werden die berechneten Raster von "0" auf "2" geändert.
        for step in path:
                temp_maze[step[0]][step[1]] = 2
```

```
    # Hier wird die Produktionslinie für im Terminal gezeichnet.
    for row in temp_maze:
        line = []
        # Anstatt der "1" wird "u2588" gesetzt.
        for col in row:
          if col == 1:
                # Ein "Full Block" wird hinzugefügt.
                line.append("\u2588")
          elif col == 0:
                # Zeichen bleibt leer.
                line.append(" ")
          elif col == 2:
                # Ein Punkt wird hinzugefügt.
                line.append(".")
        print("".join(line))

# 2.2 Hier werden Informationen über jeden Kundenauftrag gesammelt, der abgelehnt wurde.
# 5.2 Hier werden Informationen über jeden Kundenauftrag gesammelt, der abgelehnt wurde.
# 8.2 Hier werden Informationen über jeden Kundenauftrag gesammelt, der abgelehnt wurde.

def on_order_refused(mqttc, obj, msg):
    data = json.loads(msg.payload)
    logger.error('refused_orders input: ' + str(data))
    order_id = data['order']['id']
    order_pos = data['order']['pos']
    global cars_list
    global refused_orders

    # Hier wird der Kundenauftrag in einer Liste von abgelehnten Aufträgen gespeichert.
    if order_pos not in refused_orders:

        refused_orders[order_pos] = { }
        refused_orders[order_pos][order_id] = { }
```

```
                refused_orders[order_pos][order_id]['order'] = data['order']
                refused_orders[order_pos][order_id]['cars'] = []
                refused_orders[order_pos][order_id]['cars'].append(data['car_id'])
                # Wenn alle FTFs abgelehnt haben, dann gilt der Kundenauftrag als verloren.
                if len(cars_list) == len(refused_orders[order_pos][order_id]['cars']):
                        mqttc.publish("order-lost", json.dumps(data['order']), qos=2)


        else:
                # Wenn der Kundenauftrag zwei Mal abgelehnt wurde und sich an der selben
Position befindet, wird hier die ID des Kundenauftrages überprüft.


                # Order Pos already saved
                if order_id not in refused_orders[order_pos]:
                                        # New Order ID
                        refused_orders[order_pos][order_id] = data['order']
                        refused_orders[order_pos][order_id]['cars'] = []
                        refused_orders[order_pos][order_id]['cars'].append(data['car_id'])
                        # Wenn alle FTFs abgelehnt haben, dann gilt der Kundenauftrag als
verloren.
                        if len(cars_list) == len(refused_orders[order_pos][order_id]['cars']):
                                mqttc.publish("order-lost", json.dumps(data['order']), qos=2)
                        # order lost


                else:
                                                        # Order ID already saved
                        if data['car_id'] not in refused_orders[order_pos][order_id]['cars']:
        # New Car ID
                                refused_orders[order_pos][order_id]['cars'].ap-
pend(data['car_id'])


                                # Wenn alle FTFs abgelehnt haben, dann gilt der Kundenauftrag
als verloren.
                                if len(cars_list) == len(refused_orders[order_pos][or-
der_id]['cars']):
                                        mqttc.publish("order-lost", json.dumps(data['order']),
qos=2)          # order lost
```

```
                    else:
                                                                  # Car ID already saved
                              # Wenn alle FTFs abgelehnt haben, dann gilt der Kundenauftrag
als verloren.
                              if len(cars_list) == len(refused_orders[order_pos][or-
der_id]['cars']):
                                        mqttc.publish("order-lost", json.dumps(data['order']),
qos=2)          # order lost


# (2.3) Die eigene Strecke wird mit den anderen Transportfahrzeugen geteilt.
# (5.3) Die eigene Strecke wird mit den anderen Transportfahrzeugen geteilt.
# (8.3) Die eigene Strecke wird mit den anderen Transportfahrzeugen geteilt.
def share_my_path(order, path):
          #Die Länge von Variable Path wird im Dictionary order im Key 'path' gespeichert
          order['path'] = len(path)
          order['car_id'] = car_id
          print('My path to share:'+str(path))
          mqttc.publish("car-order-path", json.dumps(order), qos=0)


# 2.4 Verhandlung zwischen den Transportfahrzeugen.
# 5.4 Verhandlung zwischen den Transportfahrzeugen.
# 8.4 Verhandlung zwischen den Transportfahrzeugen.
def on_car_order_path(mqttc, obj, msg):
          global is_busy
          global car_id
          # An "order" wird die order_id, order_pos, order_dest, order_route, car_id und die
Länge des Weges übergeben.
          order = json.loads(msg.payload)
          # Wenn der Kundenauftrag nicht fertiggestellt ist, geht der Kundenauftrag in die
Schleife.


          if not is_completed(order):
                    if not is_busy:
                              # Hier wird die "order" gespeichert.
```

```
                        if order['id'] not in car_order:
                            # Dictionary car_order mit dem Key "order" und dem Value "id"
wird erstellt.
                            car_order[order['id']] = { }
                        # Hier wird geprüft, ob sich alle Transportfahrzeuge gemeldet haben.
                        if not (len(cars_list) - len(cars_busy)) == len(car_order[order['id']]):
                            # Hier werden die Strecke und die Transportfahrzeuge gespei-
chert.
                            car_order[order['id']][order['car_id']] = order['path']
                            # Hier wird die Fahrzeugliste verglichen, ob sich alle Transport-
fahrzeuge gemeldet haben.
                            if (len(cars_list) - len(cars_busy)) == len(car_order[order['id']]):
                                print('Car paths list: '+str(car_order))
                                # Hier wird die Liste mit den Strecken der Transportfahr-
zeuge ermittelt.
                                minval = min(car_order[order['id']].values())
                                # Hier wird verglichen, wer die kürzeste Strecke hat.
                                shortest_cars_ids = [k for k, v in car_order[or-
der['id']].items() if v==minval]
                                car_order[order['id']].clear()
                                # Wenn ein oder mehrere Transportfahrzeuge in der Liste
sind, wird entschieden, ob gewürfelt werden muss, oder ob der Kundenauftrag direkt verge-
ben wird.
                                if car_id in shortest_cars_ids:
                                    order['car_id'] = car_id
                                    # Dictionary car_dice wird erstellt (leer).
                                    car_dice = { }
                                    # Wenn mehr als ein Transportfahrzeug die kür-
zeste Strecke hat, wird gewürfelt.
                                    if len(shortest_cars_ids) > 1:
                                        order['dice'] = random.randint(0, 10000)
                                        order['cars_count'] = len(shor-
test_cars_ids)
                                        print('Dice rol-
led!'+'\033[94m'+'\033[1m'+str(order['dice'])+'\033[0m')
                                        mqttc.publish("diceroll", json.dumps(or-
der), qos=0)
```

```
                                    # Ab hier geht es in 3.3.
                                else:
                                    # Ansonsten wird der Kundenauftrag an-
genommen.
                                    is_busy = True
                                    msg = {'id': car_id, 'busy': True}
                                    mqttc.publish("car-busy",
json.dumps(msg), qos=0)
                                    logger.warning("Car: "+car_id+" got or-
der: " + order['id'])
                                    mqttc.publish("car-order-taken",
json.dumps(order), qos=0)
        else:
            mqttc.publish("order-lost", json.dumps(order), qos=2)        # or-
der lost
            logger.critical('Order lost after ACCEPTED: ' + pformat(order))


# 2.5 Hier werden die gewürfelten Werte verglichen.
# 5.5 Hier werden die gewürfelten Werte verglichen.
# 8.5 Hier werden die gewürfelten Werte verglichen.
def on_diceroll(mqttc, obj, msg):
    order = json.loads(msg.payload)
    if order['id'] not in car_dice:
        car_dice[order['id']] = {}
    # Hier wird geprüft, ob sich alle Transportfahrzeuge gemeldet haben.
    if not order['cars_count'] == len(car_dice[order['id']]):
        car_dice[order['id']][order['car_id']] = order['dice']
        # Hier wird die Fahrzeugliste verglichen, ob sich alle Transportfahrzeuge ge-
meldet haben.
        if order['cars_count'] == len(car_dice[order['id']]):
            # Hier wird die Liste mit den Strecken der Transportfahrzeuge ermittelt.
            minval = max(car_dice[order['id']].values())
            smallest_car_dice_ids = [k for k, v in car_dice[order['id']].items() if
v==minval]
            car_dice[order['id']].clear()
            if car_id in smallest_car_dice_ids:
```

```
                order['car_id'] = car_id
                is_busy = True
                msg = {'id': car_id, 'busy': True}
                mqttc.publish("car-busy", json.dumps(msg), qos=2)
                logger.warning("Car: "+car_id+" got order: " + order['id'])
                mqttc.publish("car-order-taken", json.dumps(order), qos=0)


# 2.6 Hier wird entschieden, ob ein Transportfahrzeug zum Auftragseingang
# oder zu einem Buffer fahren soll.
# 5.6 Hier kann das Transportfahrzeug nur noch die Funktion drive_order_buffer() aufrufen.
# 8.6 Hier kann das Transportfahrzeug nur noch die Funktion drive_order_buffer() aufrufen.
def on_car_order_taken(mqttc, obj, msg):
    order = json.loads(msg.payload)
    # Hier wird ein Kundenauftrag in die Liste "completed_orders" aufgenommen und als
beschäftigt markiert.
    add_to_completed(order)
    if order['car_id'] == car_id:
        if 'buffer' in order:
            # Wenn von 4.4 kommt.
            drive_order_buffer(order)
        else:
            # Wenn von 3.2 oder 3.3 kommt.
            drive_order(order)


# 3 Hier wird der Kundenauftrag abgeholt und an eine Station geliefert.
def drive_order(order):
    global car_pos
    global is_busy


    dest = tuple(stationen[order['pos']]['pos'])
    # Der Weg wird von der aktuellen Position des Transportfahrzeuges bis zur aktuellen
Position des Kundenauftrages berechnet.
    path = get_path(car_pos, dest)
    print('Drive to: A, car pos:'+str(car_pos) + ", destination: " + str(dest))
```

```
        # 3.1 Das Transportfahrzeug fährt zum Auftragseingang.
        drive_to(path)
        car_pos = path[-1]
        print("Pick up Order.")
        destination = tuple(stationen[order['dest']]['pos'])
        # Hier wird der Weg vom Kundenauftrag zum Ziel berechnet.
        path = get_path(tuple(stationen[order['pos']]['pos']), destination)
        # 3.2 Hier wird der Weg vom Auftragseingang zu der ersten Maschinengruppe abge-
fahren.
        print("Drive to: "+ str(order['dest']))
        drive_to(path)
        car_pos = path[-1]


        # Informationen eines Kundenauftrages werden aktualisiert.
        # Es wird ein neues Dictionary erstellt, welches die Daten der "order" enthält und zu-
sätzlich einen Schlüssel "Station" erhält.
        msg = {
                'id': order['id'],
                'pos': order['dest'],
                'dest': order['route'][1],
                'station': order['dest'],    # Kommt neu hinzu.
                'route': order['route'],
                'car_id': car_id,
        }
        logger.warning("Car: "+car_id+" delivered order: " + order['id'])


        if 'V' not in order['dest']:
                add_to_completed(order)
        else:
                delivered_orders.append(order['id'])


        mqttc.publish("order-delivered", json.dumps(msg), qos=0)
        mqttc.publish("check-battery", json.dumps({'id': car_id}), qos=0)
```

```
# In dieser Funktion wird der Weg abgefahren.
def drive_to(path):
        global car_pos
        global battery
        print('Path:' + str(path))
        # Hier wird die Kapazität des Akkus überprüft.
        if battery <= 20:
                print('\033[91m'+'\033[1m'+'Akku is almost empty: '+str(battery)+ '%'
+'\033[0m')
        bar = FillingSquaresBar('Driving', max=len(path))


        for cell in path:
                battery = battery-1
                time.sleep(action_time) # /len(path)
                car_pos = cell
                bar.next()
        bar.finish()


def loadbattery():
        global battery
        print('\033[91m'+'\033[1m'+'Akku is almost empty: '+str(battery)+ '%. Driving to Load
Station!' +'\033[0m')
        msg = {'id': car_id, 'busy': True}
        mqttc.publish("car-busy", json.dumps(msg), qos=0)
        # Auf der Position (y=6:x=0) befindet sich die Batterieladestation.
        path = get_path(car_pos, (6,0))
        drive_to(path)
        # Hier wird der Lademodus der Batterie simuliert.
        bar = FillingSquaresBar('Loading: ', max=(100-battery))
        for step in range(100-battery):
                battery = battery + 1
                time.sleep(0.1)
                bar.next()
        bar.finish()
```

```
        # Hier wird das FTF wieder als "verfügbar" gekennzeichnet.
        msg = {'id': car_id, 'busy': False}
        mqttc.publish("car-busy", json.dumps(msg), qos=0)


def battery_low():
        #return False
        global battery
        if battery <= 20:
                return True
        else:
                return False


def add_to_completed(order):
        if order['dest'] not in completed_orders:
                completed_orders[order['dest']] = []
        if order['id'] not in completed_orders[order['dest']]:
                completed_orders[order['dest']].append(order['id'])


# In dieser Funktion wird überprüft, ob der Akku für die Fahrt noch ausreicht.
def on_check_battery(mqttc, obj, msg):
        data = json.loads(msg.payload)
        global car_id
        global is_busy
        if data['id'] == car_id:
                if battery_low():
                        # Hier wird die Funktion für das Laden der Batterie aufgerufen.
                        loadbattery()
                is_busy = False
                msg = {'id': car_id, 'busy': False}
                mqttc.publish("car-busy", json.dumps(msg), qos=0)


# 5.1 Neuer Kundenauftrag (aus einem Buffer) wird überprüft, die Strecke berechnet und die
eigene Strecke mit anderen  Transportfahrzeugen geteilt.
```

```
# 8.1 Neuer Kundenauftrag (aus einem Buffer) wird überprüft, die Strecke berechnet und die eigene Strecke mit anderen  Transportfahrzeugen geteilt.


def on_new_order_buffer(mqttc, obj, msg):
    order = json.loads(msg.payload)
    logger.warning('New order from Buffer: ' +str(order))
    global is_busy
    if not is_completed(order):
        if not is_busy:
            global car_pos


            if 'S' in order['buffer']:
                print(bufferS[order['pos']]['pos'])
                dest = tuple(bufferS[order['pos']]['pos'])
            else:
                dest = tuple(bufferE[order['pos']]['pos'])
            # Der Weg wird von der aktuellen Position des Transportfahrzeuges bis zur aktuellen Position des Kundenauftrages berechnet.
            path = get_path(car_pos, dest)
            # 5.2 Die eigene Strecke wird mit den anderen Transportfahrzeugen geteilt.
            # 8.2 Die eigene Strecke wird mit den anderen Transportfahrzeugen geteilt.
            share_my_path(order, path)
        else:
            msg = {'car_id': car_id, 'order': order}
            mqttc.publish("order-refused", json.dumps(msg), qos=0)


# (6) Hier wird der Kundenauftrag von einem Buffer abgeholt und zur nächsten Station gefahren.
# (9) Hier wird der Kundenauftrag von einem Buffer abgeholt und zur nächsten Station gefahren.
def drive_order_buffer(order):
    global car_pos
    global is_busy
```

```
        if 'S' in order['buffer']:
                dest = tuple(bufferS[order['pos']]['pos'])
        else:
                dest = tuple(bufferE[order['pos']]['pos'])

        path = get_path(car_pos, dest)
        print('Drive from position:'+str(car_pos) + ' to destination: '+str(dest))
        # 6.1 Der Weg wird zum Buffer abgefahren.
        # 9.1 Der Weg wird zum Buffer abgefahren.
        drive_to(path)
        print("Pick up Order.")

        if 'V' in order['dest']:
                path = None
                for count, output in enumerate(outputs):
                        if not outputs[output]['busy']:
                                if count == 1:
                                        path = get_path(car_pos, tuple(outputs[output]['pos']))

                                if count > 1:
                                        if len(get_path(car_pos, tuple(outputs[output]['pos']))) <
len(path):
                                                path = get_path(car_pos, tuple(outputs[out-
put]['pos']))
                delivered_orders.append(order['id'])
        else:
                destination = tuple(stationen[order['dest']]['pos'])
                if 'S' in order['buffer']:
                        dest = tuple(bufferS[order['pos']]['pos'])
                else:
                        dest = tuple(bufferE[order['pos']]['pos'])
                print(tuple(order['pos']), destination)
                path = get_path(dest, destination) # Hier wird der Weg vom Kundenauftrag
zum Ziel berechnet.
```

```
        print('Drive from Buffer car_pos:'+str(car_pos) + ' to destination: '+ str(order['dest']))
        # 6.2 Der Weg wird zur Station oder einem Ausgang gefahren.
        print("Drive to: "+ str(order['dest']))
        drive_to(path)
        car_pos = path[-1]
        msg = {
                'id': order['id'],
                'pos': order['dest'],
                'dest': order['route'][1],
                'station': order['dest'],
                'route': order['route'],
                'car_id': car_id,
        }

        logger.warning("completed_orders: "+str(completed_orders))
        if 'V' not in order['dest']:
                add_to_completed(order)
        else:
                delivered_orders.append(order['id'])

        logger.warning("Car: "+car_id+" delivered order: " + order['id'])
        mqttc.publish("order-delivered", json.dumps(msg), qos=0)
        mqttc.publish("check-battery", json.dumps({'id': car_id}), qos=0)

def is_completed(order):
        if order['dest'] in completed_orders:
                if (order['id'] in completed_orders[order['dest']]) or (order['id'] in delivered_or-
ders):
                        return True
                else:
                        return False
        else:
```

```
                return False


# Hier wird an die anderen Transportfahrzeuge eine Nachricht gesendet,
# dass ein Transportfahrzeug aus dem Simulation genommen wurde.
def stop_car():
        msg = {'id': car_id,}
        mqttc.publish("car-stop", json.dumps(msg), qos=0)


def get_battery():
        global battery
        bar = FillingSquaresBar('Battery', max=100)
        for i in range(battery):
            bar.next()
        bar.finish()


# Eine Nachricht kommt aus der Funktion stop_car(), wodurch die anderen Transportfahr-
zeuge wissen,
# dass die "Id" des Transportfahrzeuges aus der "car_list" rausgenommen werden kann.
def on_car_stop(mqttc, obj, msg):
        msg = json.loads(msg.payload)
        if msg['id'] in cars_list:
                cars_list.remove(msg['id'])
                print('Car removed from list: ' + msg['id'])
                print('New car list: ' + str(cars_list))


# Sobald ein Transportfahrzeug beschäftigt ist, kommt hier eine Nachricht rein und das
# Transportfahrzeug wird von den anderen Transportfahrzeugen als beschäftigt erkannt.
def on_car_busy(mqttc, obj, msg):
        msg = json.loads(msg.payload)
        if msg['id'] in cars_list:
                if msg['busy']:
                        cars_busy.append(msg['id'])
                else:
```

```
                if msg['id'] in cars_busy:
                        cars_busy.remove(msg['id'])
        logger.debug('New car busy list: ' + str(cars_busy))
```

Die Funktion für die Last-Will Funktion. Ein Transportfahrzeug wird aus der Simulation genommen, falls es sich nicht mehr meldet.

```
def on_car_disconnect(mqttc, obj, msg):
        msg = json.loads(msg.payload)
        if msg['id'] in cars_list:
                cars_list.remove(msg['id'])
                print('Car ' + msg['id'] +' removed from car list.')
                print('New car list: ' + str(cars_list))
                print('Last position: ' + str(car_pos))


# Connect to MQTT
mqttc = mqtt.Client(str(uuid.uuid4()))
print("##############################################")
print("Enter Y to manually determine an ID.")
print("Enter N to automatically generate an ID.")
print("##############################################")
eingabe = input("Please enter here: ")


# 1 Das Transportfahrzeug generieren eine ID.
# ID Generator.
if 'Y' in eingabe:
        car_id = input("Please enter the ID: ")
else:
        car_id = str(uuid.uuid4())[:3]
```

Hier kann entschieden werden, wo die Transportfahrzeuge zufällig auf der Produktionslinie generieren. car_pos=(x(Zeile),y(Spalte)).

```
car_pos = (random.randint(11, 12), random.randint(0, 8))


cars_list = [car_id]
```

```
cars_busy = []

# Last-will Testament.
msg = {'id':car_id, 'pos':car_pos}
mqttc.will_set('car-disconnect', json.dumps(msg), qos=0)

# Hier wird die Verbindung zu MQTT aufgebaut.
mqttc.connect("localhost", 1883, 60)

# Subskriptionen.
mqttc.subscribe("new-order", 0)
mqttc.subscribe("diceroll", 0)
mqttc.subscribe("new-order-buffer", 0)
mqttc.subscribe("car-init", 0)
mqttc.subscribe("car-init-response", 0)
mqttc.subscribe("car-order-path", 0)
mqttc.subscribe("car-order-taken", 0)
mqttc.subscribe("car-busy", 0)
mqttc.subscribe("car-stop", 0)
mqttc.subscribe("check-battery", 0)
mqttc.subscribe("car-disconnect", 0)
mqttc.subscribe("order-refused", 0)

# Wenn eine Nachricht an ein Topic ankommt, wird die jeweilige Funktion ausgeführt.
mqttc.message_callback_add('diceroll', on_diceroll)
mqttc.message_callback_add('new-order', on_new_order)
mqttc.message_callback_add('new-order-buffer', on_new_order_buffer)
mqttc.message_callback_add('car-init', on_car_init)
mqttc.message_callback_add('car-init-response', on_car_init_response)
mqttc.message_callback_add('car-order-path', on_car_order_path)
mqttc.message_callback_add('car-order-taken', on_car_order_taken)
mqttc.message_callback_add('car-stop', on_car_stop)
mqttc.message_callback_add('car-busy', on_car_busy)
```

```
mqttc.message_callback_add('check-battery', on_check_battery)
mqttc.message_callback_add('car-disconnect', on_car_disconnect)
mqttc.message_callback_add('order-refused', on_order_refused)


# Aufbau der ersten Nachricht an das Topic car-init.
msg = {'id':car_id, 'car_pos': car_pos}
# Hier wird "msg" in das Topic car-init gesendet.
mqttc.publish("car-init", json.dumps(msg), qos=0)


mqttc.loop_start()
# Hier wartet die Konsole auf den Aufruf des Menüs. Das Menü wird mit der Taste "h" in ei-
nem Terminal aufgerufen.
print('\nCar: ' + str(car_id) + ' ready for order.')
print("Help? Press (h).\n")
# Hier wird auf eine Eingabe gewartet, ohne den Programmfluss zu unterbrechen.
with console.NonBlockingConsole() as nbc:
    while 1:
        if nbc.get_data() == 'h':
            print("\ns: Stop car\nb: Battery status\ni: car id\nl: Location")
            print("o: completed orders\nc: cars online?")
        if nbc.get_data() == 's':
            stop_car()
            mqttc.loop_stop()
        if nbc.get_data() == 'b':
            get_battery()
        if nbc.get_data() == 'l':
            print('\nLocation: '+str(car_pos))
        if nbc.get_data() == 'i':
            print('\ncar_id: '+str(car_id))
        if nbc.get_data() == 'o':
            print('\ncompleted orders: '+str(completed_orders))
        if nbc.get_data() == 'c':
            print('\n'+'-'+str(len(cars_list))+'-'+' cars online. Car IDs: '+str(cars_list)+'\n')
```

```
        if nbc.get_data() == '\x1b':  # x1b is ESC
            break
```

```
import paho.mqtt.client as mqtt
import uuid
import json
import time
import random
import coloredlogs, logging

# Farbwiedergabe im Terminal.
class color:
   PURPLE = '\033[95m'
   CYAN = '\033[96m'
   DARKCYAN = '\033[36m'
   BLUE = '\033[94m'
   GREEN = '\033[92m'
   YELLOW = '\033[93m'
   RED = '\033[91m'
   BOLD = '\033[1m'
   UNDERLINE = '\033[4m'
   END = '\033[0m'
# Farbwiedergabe und Uhrzeit im Terminal.
logger = logging.getLogger(__name__)
logging.basicConfig(
    format='%(asctime)s %(levelname)-8s %(message)s',
    datefmt='%H:%M:%S')
coloredlogs.install(level='DEBUG')

# Alle möglichen Variationen eines Kundenauftrages.
orders_list = [
            {'id':str(uuid.uuid4())[:3], 'pos':'A', 'dest':'S1', 'route': ('S1','E1')},
            {'id':str(uuid.uuid4())[:3], 'pos':'A', 'dest':'S1', 'route': ('S1','E2')},
```

```
        {'id':str(uuid.uuid4())[:3], 'pos':'A', 'dest':'S2', 'route': ('S2','E1')},
    {'id':str(uuid.uuid4())[:3], 'pos':'A', 'dest':'S2', 'route': ('S2','E2')},
    {'id':str(uuid.uuid4())[:3], 'pos':'A', 'dest':'S3', 'route': ('S3','E1')},
    {'id':str(uuid.uuid4())[:3], 'pos':'A', 'dest':'S3', 'route': ('S3','E2')},

]

orders = {}

# 5 Kundenauftrag ist wieder bereit (Nur wegen der Logik, hier passiert sonst nichts).
# 8 Kundenauftrag ist wieder bereit (Nur wegen der Logik, hier passiert sonst nichts).
def on_order_buffer_ready(mqttc, obj, msg):
    order = json.loads(msg.payload)
    logger.error('on_order_buffer_ready: ' + str(order))
    mqttc.publish("new-order-buffer", json.dumps(order), qos=2)

def on_order_lost(mqttc, obj, msg):
    order = json.loads(msg.payload)
    if my_id == order['id']:
        logger.error('on Order lost: ' + str(order))
        # Zeiteinheit, nachdem ein Kundenauftrag wieder gesendet wird.
        time.sleep(2)
        if 'buffer' in order:
            mqttc.publish("new-order-buffer", json.dumps(order), qos=2)
        else:
            mqttc.publish("new-order", json.dumps(order), qos=2)

# Erzeugung der Instanz "mqttc".
mqttc = mqtt.Client()

# Methodenaufruf der Methode "connect" durch die Instanz "mqttc".
mqttc.connect("localhost", 1883, 60)
```

```
# Subskriptionen.
mqttc.subscribe("order-buffer-ready", 0)
mqttc.subscribe("order-lost", 0)

# Nachricht kommt im Topic an, wodurch die jeweilige Funktion ausgelöst wird.
mqttc.message_callback_add('order-lost', on_order_lost)
mqttc.message_callback_add('order-buffer-ready', on_order_buffer_ready)

# 2 Ein zufällig ausgewählter Kundenauftrag wird an die Transportfahrzeuge gesendet.
el = random.randint(0, 5)

my_id = orders_list[el]['id']
print("#########################################################################"
)
print('Order '+str(my_id)+' send request.')
print('Order Information: ' + str(orders_list[el]))
print("#########################################################################"
)
mqttc.publish("new-order", json.dumps(orders_list[el]), qos=2)
mqttc.loop_forever()
```

```
import paho.mqtt.client as mqtt
import uuid
import json
import time
from progress.bar import ChargingBar
import threading
import coloredlogs, logging

logger = logging.getLogger(__name__)
logging.basicConfig(
        format='%(asctime)s %(levelname)-8s %(message)s',
        datefmt='%H:%M:%S')
coloredlogs.install(level='DEBUG')
```

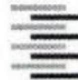
```
is_busy = False

# Worktime funktioniert in dieser Version nicht mit threads.
station = {
    'S1':{'pos':[11,1], 'busy': False, 'worktime': 3},
    'S2':{'pos':[11,4], 'busy': False, 'worktime': 7},
    'S3':{'pos':[11,7], 'busy': False, 'worktime': 5},
    'E1':{'pos':[6,2], 'busy': False, 'worktime': 5},
    'E2':{'pos':[6,6], 'busy': False, 'worktime': 8},
    'A':{'pos':[12,4], 'busy': False, 'worktime': 1}
}

bufferS = {
    'SB1':{'pos':[8,1], 'busy': False,},
    'SB2':{'pos':[8,2], 'busy': False,},
    'SB3':{'pos':[8,3], 'busy': False,},
    'SB4':{'pos':[9,4], 'busy': False,},
    'SB5':{'pos':[8,5], 'busy': False,},
    'SB6':{'pos':[8,6], 'busy': False,},
    'SB7':{'pos':[8,7], 'busy': False,},
}

bufferE = {
    'EB1':{'pos':[3,2], 'busy': False,},
    'EB2':{'pos':[3,3], 'busy': False,},
    'EB3':{'pos':[3,4], 'busy': False,},
    'EB4':{'pos':[3,5], 'busy': False,},
    'EB5':{'pos':[3,6], 'busy': False,},
}

outputs = {
    'V1':{'pos':[1,1], 'busy': False, 'worktime': 1},
```

```
        'V2':{'pos':[1,4], 'busy': False, 'worktime': 1},
        'V3':{'pos':[1,7], 'busy': False, 'worktime': 1},
}


# 4 Hier wird bestimmt, ob der Kundenauftrag entweder fertig ist, oder ob der
# Kundenauftrag einem Buffer zugewiesen wird.
# 7 Hier wird bestimmt, ob der Kundenauftrag entweder fertig ist, oder ob der
# Kundenauftrag einem Buffer zugewiesen wird.
# 10 Der Kundenauftrag wurde an einen der Ausgänge zugestellt. Der Zyklus ist somit been-
det.
def on_order_delivered(mqttc, obj, msg):
        order = json.loads(msg.payload)
        print("Start working: " + str(order['id']) + ":" + order['station'])


        if 'V' in order['pos']:
                logger.error('Order Completed:'+str(order))
        else:
                station[order['dest']]['busy'] = True
                # Thread wird definiert. Working wird in einem sepparaten Thread aufgerufen.
                thread = threading.Thread(target=working, args=(order,))
                # Thread wird gestartet.
                logger.warning(str(threading.enumerate())+ "\n")
                thread.start()
                logger.warning(str(thread.name) + ': order-' + str(order['id']))


# 4.1 In dieser Funktion wird das Produzieren simuliert.
# 4.1 In dieser Funktion wird das Produzieren simuliert.
def working(order):
        logger.warning("Start working on order: " + str(order['id']) + '\n')
        time.sleep(station[order['station']]['worktime'])
        move_to_buffer(order)


# 4.4 Hier wird der Kundenauftrag in den jeweiligen Buffer zugewiesen.
```

```
# 4.4 Hier wird der Kundenauftrag in den jeweiligen Buffer zugewiesen.
def move_to_buffer(order):
    print("Station Work finished: " + str(order['id']) + ":" +order['station'])
    if 'S' in order['station']:
        for buffer in bufferS:
            if not bufferS[buffer]['busy']:
                order_pos = buffer
                print('Move to buffer: '+str(order_pos))
                break
    else:
        for buffer in bufferE:
            if not bufferE[buffer]['busy']:
                order_pos = buffer
                print('Move to buffer: '+str(order_pos))
                break
    if 'E' not in order['station']:
        dest = order['route'][1]
    else:
        dest = 'V'
    msg = {
        'id':order['id'],
        'pos':order_pos or '',
        'dest':dest,
        'buffer': order['station'],
        'route': order['route'],


    }
    mqttc.publish("order-buffer-ready", json.dumps(msg), qos=2)


mqttc = mqtt.Client()


mqttc.connect("localhost", 1883, 60)
```

```
# Subskriptionen.
mqttc.subscribe("order-delivered", 0)
print("###################################")
print("Stationmanager is ready for order.")
print("###################################")

# Nachricht kommt im Topic an, wodurch die jeweilige Funktion ausgelöst wird.
mqttc.message_callback_add('order-delivered', on_order_delivered)
run = True
while run:
    mqttc.loop_start()
```

Selbstständigkeitserklärung



Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Dieses Blatt, mit der folgenden Erklärung, ist nach Fertigstellung der Abschlussarbeit durch den Studierenden auszufüllen und jeweils mit Originalunterschrift als letztes Blatt in das Prüfungsexemplar der Abschlussarbeit einzubinden.
Eine unrichtig abgegebene Erklärung kann -auch nachträglich- zur Ungültigkeit des Studienabschlusses führen.

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: Greb

Vorname: Arthur

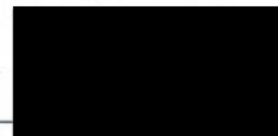
dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Konzeptionierung und Programmierung der Kommunikationsstrukturen und -abläufe für eine dezentrale Produktions-steuerung bei der Verhandlung fahrerloser Transportfahrzeuge über Fahraufträge

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

- die folgende Aussage ist bei Gruppenarbeiten auszufüllen und entfällt bei Einzelarbeiten -

Die Kennzeichnung der von mir erstellten und verantworteten Teile der -bitte auswählen- ist erfolgt durch:

Hamburg	11.05.2021	
Ort	Datum	Unterschrift im Original