

Dokumentation

Daniel Friedrich

Verbundprojekt 2018/2019

Daniel Friedrich

Verbundprojekt 2018/2019

Abschlusspräsentation eingereicht im Rahmen des Verbundprojektes

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

Betreuender Prüfer: Prof. Dr.-Ing. Ulfert Meiners
Zweitgutachter: Prof. Dr.-Ing. Jochen Maaß, Prof. Dr.-Ing. Michael Röther

Eingereicht am: 25. Mai 2019

Daniel Friedrich

Thema der Arbeit

Verbundprojekt 2018/2019

Stichworte

Smart Kamera, Objekterkennung, Farbdetektion, Gehäusekonstruktion, Pixy-Cam, Raspberry Pi

Kurzzusammenfassung

Im Rahmen dieser Arbeit werden zwei Kamerasysteme verglichen, welche farbige Objekte auf einem Laufband erkennen und deren Farbe auswerten. Über verschiedene Schnittstellen wird die Auswertung ausgegeben. Dazu werden Anwendungen entworfen, welche die Auswertung auf einem der Systeme ermöglichen. Die Arbeit beschreibt außerdem die konkrete Implementierung des Quellcodes der entworfenen Anwendungen für eines der Kamerasysteme. Ein weiterer zentraler Punkt der Arbeit ist der Entwurf einer geeigneten Halterung für die Kamerasysteme.

Daniel Friedrich

Title of Thesis

Verbundprojekt 2018/2019

Keywords

Smart Camera, Objecttracking, Colordetection, Construction, PixyCam, Raspberry Pi

Abstract

In this thesis, two camera systems are compared, which recognize colored objects on a conveyor belt and evaluate their color. The evaluation is output via various interfaces. For this purpose, applications are designed, which enable the evaluation on one of the systems. The thesis also describes the implementation of the source code of the designed applications for one of the camera systems. Another point of the work is the design of a suitable holder for the camera systems.

Inhaltsverzeichnis

1. Einleitung	1
1.1. Vorhaben	1
2. Grundlagen und Konzepte	3
2.1. Qt	3
2.1.1. Eventbasiertes System	3
2.1.2. State Machines in Qt	3
3. Aufgabenstellung	4
4. Schnittstellen	5
4.1. Kommunikation zu Gewerk 2	5
4.1.1. Sequenzdiagramm	5
4.1.2. Fehlertypen und Behebung	5
5. Implementierung	6
5.1. Programmstruktur	6
5.2. Databasehandler	6
5.3. UDP-Handler	6
5.4. Robotino	6
5.5. Auftrag, Prozess, Prozessschritt	6
5.5.1. Prozessschritt	7
5.5.2. Prozess	7
5.5.3. Auftrag	8
5.6. Fertigungsplanung	9
5.6.1. Initialisierung	9
5.6.2. Roboter Statusänderungen	12
5.6.3. Hard-Code Funktionen	14
5.6.4. Zustandsdiagramm	14
5.6.5. State Machine Implementierung	14
5.6.6. Shuffle-Algorithmus	18
5.7. Mainwindow	19
5.8. Main	19
5.9. Weitere Klassen	19
6. Visualisierung	21
6.1. Grundstruktur	21
6.2. Live-View	22
6.3. Auftragsübersicht	22
6.4. Tab-View	22
6.4.1. Hard-Code Bereich	22

6.5. Roboterstatus	22
6.6. Prozesseingabe	22
6.7. Auftragseingabe	22
6.8. Benutzerinteraktion	22
6.8.1. Tooltips	22
7. Simulation	29
7.0.1. Zustandsdiagramm simulierter Roboter	29
8. Datenbank	32
A. Inhalt der CD	35

1. Einleitung

Um an einem realen System Abläufe und Prozesse zu simulieren, wird den Studierenden eine Anlage zur Verfügung gestellt, welche hauptsächlich aus Teilen der Firma „Festo“ besteht (folgend: Festo-Transfersystem). An diesem Festo-Transfersystem können die Studierende Laufbänder, Ampelanlage, Lichtschranken, Weichen etc. mithilfe selbst geschriebener Programme ansteuern und auslesen.

Das Laufband kann aufgelegte Objekte transportieren. Dabei kann die aktuelle Position des Objekts durch diverse Lichtschranken erfasst werden. Am Anfang des Laufbands kann mithilfe eines Entfernungssensors die Höhe des Objekts erfasst werden. Mit den vorhandenen Komponenten kann das Festo-Transfersystem allerdings ausschließlich die Position und die Höhe der Objekte bestimmen und nicht deren Farbe, Form, Ausrichtung oder sonstige optische Merkmale.



Abbildung 1.1.: CMUcam5 Pixy

Eine Erweiterung um ein Kamerasystem kann diese Mängel beheben und sogar noch weitere Funktionen und Features hinzufügen.

1.1. Vorhaben

blabla

Ähnlich einer Smart Kamera [Sch09], bei der nicht nur das Kamerabild, sondern auch weitere schon verarbeitete Informationen ausgegeben werden sollen, wird

1. Einleitung

ein Prozessor benötigt, welcher in der Lage ist Bildverarbeitung durchzuführen. Es wurde als Möglichkeit für ein Prozessorsystem empfohlen, sich mit dem BeagleBone Black und Raspberry PI auseinanderzusetzen. Mithilfe einer Marktrecherche werden zunächst weitere geeignete Kamerasysteme gesucht und anschließend anhand der Eignung sortiert. Dabei wird auch eine geeignete Schnittstelle zu dem Kamerasystem ausgewählt, mit dem die Bilder und Informationen an den PC gesendet werden. Nachdem die beiden geeignetsten Kamerasysteme gewählt und bestellt sind, werden zugehörige Gehäuse und Befestigungen konstruiert und gefertigt. Nebenbei werden erste einfache Testprogramme geschrieben, um die Funktion von den gewählten Kamerasystemen zu gewährleisten. Der erste Prototyp wird dabei voraussichtlich mit Hilfe rapid-prototyping ein Erzeugnis aus dem 3D-Druck sein, um Kamera an der richtigen Position zu halten. Nachdem das Endprodukt montiert ist, folgt eine Auswertung der Kamerabilder. Folgend folgen optional das Erstellen einer Bibliotheksdatei und eine Automation der Kalibrierung und des Weißabgleichs. Der Zugriff auf die Kamera soll gewährleistet sein.



Abbildung 1.2.: PixyCam Gehäusefehler

2. Grundlagen und Konzepte

Grundlagen

2.1. Qt

2.1.1. Eventbasiertes System

2.1.2. State Machines in Qt

erwähnen, dass nur `entered()` benutzt wird

[Qt 19]

3. Aufgabenstellung

Um an

4. Schnittstellen

Schnittstellen

4.1. Kommunikation zu Gewerk 2

4.1.1. Sequenzdiagramm

Sequenzdiagramm Kommunikation

Telegramm Gewerk 2

4.1.2. Fehlertypen und Behebung

Fehlertyp 1

Fehlertyp 2

Fehlertyp 3

5. Implementierung

Implementierung

5.1. Programmstruktur

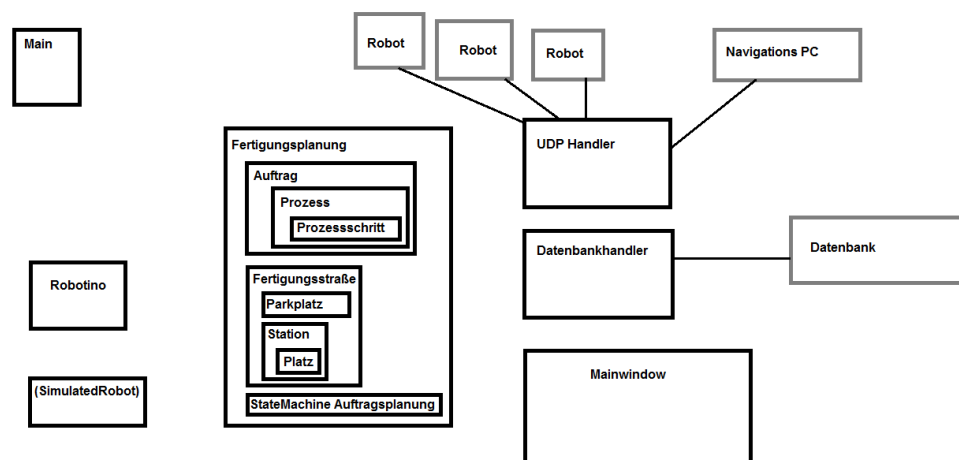


Abbildung 5.1.: Klassendiagramm

5.2. Databasehandler

5.3. UDP-Handler

5.4. Robotino

5.5. Auftrag, Prozess, Prozessschritt

Die im Programm auftretenden Aufträge sind in einer bestimmten Struktur organisiert. Diese variiert zu der in der Datenbank abgespeicherten Struktur. So wird über die Visualisierung ein Auftrag angelegt. Jeder so angelegte Auftrag beinhaltet ein oder mehrere Prozesse. In der Datenbank sind dies komplementär gesehen die Werkstücke. Jeder Prozess besteht aus mehreren Prozessschritten, die den genauen Ablauf eines Prozesses beschreiben.

5. Implementierung

Auftrag, Prozess und Prozessschritt sind in verschiedene Klassen geschrieben. Folgend werden diese Klassen näher beleuchtet.

5.5.1. Prozessschritt

Ein Prozessschritt besteht aus genau einer Stations-ID verknüpft mit einer festen Bearbeitungsdauer in Sekunden. Dies beschreibt für einen Teil eines Prozesses. In einer Fortschritts-Flag wird festgehalten, ob der Prozessschritt bereits bearbeitet wurde, oder noch bearbeitet werden muss.

Der Prozessfortschritt kann ein Signal emittieren, welches anzeigt, dass sich der Fortschritt geändert hat.

5.5.2. Prozess

Ein Prozess, in der Datenbank Werkstück, beinhaltet eine Liste von Prozessschritten. Diese Liste bestimmt mit ihrer Reihenfolge und Länge die Bearbeitungsvorschrift für ein Werkstück. Jeder Prozess besitzt zudem einen Fortschritt, eine eindeutige ID und eine ReferenzID.

Mit der eindeutigen ID kann später ein Prozess innerhalb des Programms wiedergefunden werden. Nur die in Aufträgen befindlichen Prozesse erhalten eine aufsteigende laufende Nummer als eindeutige ID. Die in der Initialisierung erzeugten Prozesse oder die über die Visualisierung erzeugten Prozesse (folgend Referenzprozesse) erhalten keine spezielle eindeutige ID, sondern eine aufsteigende eindeutige ReferenzID. Dies führt dazu, nicht alle Informationen in den einzelnen Prozessen halten zu müssen, sondern auf den initialisierten Referenzprozess zugreifen zu können.

Die ReferenzID eines Prozesses enthält die eindeutige ID des zugehörigen Referenzprozesses.

Zur Prozesskontrolle gibt es zusätzlich noch verschiedene Flags die den Status des Prozesses widerspiegeln. Dazu gehört ein Blocked- und ein InProgress-Flag. Das InProgress-Flag wird gesetzt, sobald ein Werkstück an einer Station bearbeitet wird. Das Blocked-Flag kann über die Hard-Code Area (vgl. Kapitel 6.4.1) gesetzt werden und verhindert eine weitere Bearbeitung.

Jeder Prozess enthält einen Timer, der die verbrauchte Zeit an einem Arbeitsplatz kontrolliert. Der Timer wird gestartet, sobald ein Werkstück an einer Station bearbeitet wird, die nötige Zeit wird dem zugehörigen Prozessschritt entnommen. Bei Ablauf des Timers wird die Funktion TimerElapsed() aufgerufen, in der der zugehörige Prozessschrittfortschritt auf fertig gesetzt und das InProgress-Flag zurückgesetzt wird.

Über die beiden Signale ProzessFinished und FortschrittUpdated kann bekannt gemacht werden, dass sich ein Fortschritt eines Prozesses geändert hat oder die Arbeit an einer Station beendet wurde und das Werkstück bereit ist für Abholung.

5. Implementierung

Mit der im Prozess enthaltenen Funktion `UpdateFortschritt()` wird der Fortschritt eines Prozesses anhand der beinhalteten Prozessschritte aktualisiert. Dazu wird die Prozessschrittliste durchgegangen und für jeden Prozessschritt, der als fertig markiert wurde ein entsprechender Fortschritt zum Prozessfortschritt addiert. Die einzelnen Prozessschritte sind dabei so gewichtet, dass die Summe aus allen Prozessschritten, wenn alle abgeschlossen sind, 100 ergibt. Durch Rundungsfehler können in der aktuellen Umsetzung nicht mehr als 25 Prozessschritte je Prozess sauber dargestellt werden. Sollten mehr Prozessschritte erforderlich sein, so müsste die Fortschrittsberechnung angepasst werden.

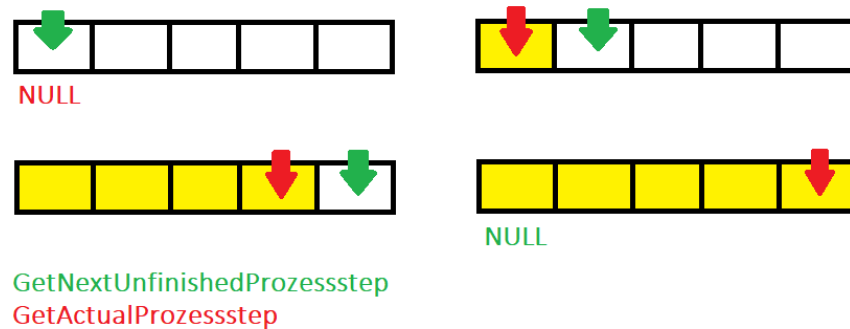


Abbildung 5.2.: Visualisierung des Rückgabewertes der Funktionen `getNextUnfinishedProzessstep()` und `getActualProzessstep()`

Um auf den richtigen Prozessschritt innerhalb der Liste zuzugreifen sind zwei unterschiedliche Funktionen implementiert. Beide Funktionen unterscheiden sich ausschließlich in ihrem Rückgabewert. In Abbildung 5.2 sind alle möglichen Fälle Prozessschrittliste leer oben links, ein oder mehrere Prozessschritte in der Liste oben rechts und unten links und Liste voll unten rechts dargestellt. Dabei ist zu erkennen, dass die Funktion `getNextUnfinishedProzessstep()` immer den nächsten Prozessschritt zurückliefert, der noch nicht bearbeitet wurde. Bei voller Liste wird **NULL** zurückgegeben. Anders hierzu die Funktion `getActualProzessstep()`, die **NULL** bei leerer Liste zurückgibt und sonst den letzten fertigen Prozessschritt.

Die Funktion `getActualProzessstep()` wird in der Fertigungsplanung zur Planung der Aufträge verwendet (siehe 5.6.5). `getNextUnfinishedProzessstep()` findet Anwendung sobald auf einem aktuellen Prozess eine Änderung auftritt oder umgesetzt werden soll. Wenn beispielsweise der Timer abgelaufen ist, wird in der Funktion `TimerElapsed()` über der zurückgegebene Prozessschritt als fertig markiert.

5.5.3. Auftrag

Die Klasse `Auftrag` enthält eine Liste von Prozessen und darin enthaltenen Prozessschritten und bündelt somit alle drei Klassen. Ähnlich dem Prozess gibt es einen Fortschritt, der sich anteilig aus den einzelnen Prozessfortschritten errechnet und eine eindeutige ID mit derer der Auftrag wiedergefunden und verwaltet werden kann.

5. Implementierung

Mit dem Signal `FortschrittUpdated` kann bekannt gemacht werden, dass sich der Fortschritt geändert hat und z.B. die Visualisierung aktualisiert werden. Das Signal wird gesendet, sobald die Funktion `UpdateFortschritt()` aufgerufen wurde. In dieser Funktion werden alle enthaltenen Prozesse auf ihren Fortschritt überprüft und im Auftrag gegebenenfalls angepasst. Die Fertigungsplanung verknüpft Auftrag, Prozess und Prozessschritt derartig, dass ein Fortschrittsupdate innerhalb eines Prozessschrittes eine Aktualisierung des Prozessfortschritts hervorruft. Dies wiederum lässt den entsprechenden Auftragsfortschritt aktualisieren (siehe Listing 5.1).

Ein bestimmter Prozess aus der Liste kann über die Funktion `GetProzessByID()` mit seiner eindeutige ID zurückgegeben werden.

Weiterhin werden zwei Funktionen bereitgestellt, die von der Fertigungsplanung genutzt werden um Prozesse oder Prozessschritte zu erhalten. Die Funktion `GetNextUnfinishedProzesssteps()` gibt, mit Hilfe der Funktion `GetNextUnfinishedProzessstep()` aus dem Prozess, eine Liste aller entsprechenden Prozessschritte zurück (vgl. Abschnitt 5.5.2). Über die Funktion `GetAllUnfinishedProzesses()` werden alle Prozesse zurückgegeben, die noch unfertige Prozessschritte enthalten, sofern diese nicht blockiert sind. Hiermit werden in der Fertigungsplanung die zu bearbeitenden Prozesse ausgewählt (siehe 5.6.5).

5.6. Fertigungsplanung

Die Klasse `Fertigungsplanung` enthält alle verfügbaren Roboter, die Datenbank-schnittstelle, die abzuarbeitenden Aufträge (vgl. Abschnitt 5.5), die Fertigungsstraße und eine State Machine zur Verteilung von Roboteranträgen. Außerdem wird auf Events reagiert, die von der Visualisierung oder anderen Quellen kommen. Zu den Events gehören Statusänderungen des Roboters, hinzufügen von Aufträgen oder Prozessen und Funktionen, die über die Hard-Code Area (vgl. Kapitel 6.4.1) angestoßen werden können. In der `Fertigungsplanung`-Klasse werden somit alle Funktionalitäten gebündelt, die nicht direkt mit der Visualisierung zusammenhängen.

5.6.1. Initialisierung

Um die `Fertigungsplanung` zu initialisieren wird zunächst in der Funktion `InitProzesses()` ein Grundzustand hergestellt, in der mindestens 5 Prozesse vorhanden sind. Dazu werden zunächst über einen Datenbankaufruf alle dort verfügbaren Prozesse in eine Liste geschrieben.

Sollte die Liste weniger als 5 Elemente beinhalten, also in der Datenbank zu Programmstart weniger als 5 Prozesse vorhanden sein, so werden 5 neue zuvor definierte Prozesse (Abschnitt 5.6.1) angelegt und die Liste so überschrieben.

Die erzeugten Prozesse werden abschließend in die Datenbank geschrieben. Außerdem erhalten alle Prozesse gemäß ihrer Schritte und Dauer in der Visualisierung einen entsprechenden Tooltip (vgl. Kapitel 6.8.1).

5. Implementierung

Die Prozessliste kann über die Funktion `AddProzess()` dynamisch während das Programm läuft erweitert werden. Dies geschieht bei absenden eines eingestellten Prozesses in der Visualisierung (6.6).

Aufträge hinzufügen

Durch Einstellen und Absenden eines Auftrags in der Visualisierung (6.7) wird die Funktion `AddAuftrag()` aufgerufen. In dieser Funktion wird zunächst ein leeres Auftragsobjekt erzeugt und diesem die nächst höhere Auftragsnummer aus der Datenbank zugeordnet.

Für jeden Prozesseintrag aus der Visualisierung wird nun die angegebene Anzahl des jeweiligen Prozesses eine Prozesskopie erzeugt. Den so kopierten Prozessen werden alle zugehörigen Prozessschritte als Kopie angehängt, um eigenständige Aufträge zu erhalten. Zuletzt werden die Prozesse und Aufträge miteinander verknüpft (siehe Listing 5.1).

Listing 5.1: Auftragserzeugung und Verknüpfungen

```
Auftrag *A = new Auftrag();  
2 for (int prozesscounter = 0; prozesscounter < 5;  
   prozesscounter++) //5 Prozess UI Elemente  
4 {  
   for (int i = 0; i < Avalues[prozesscounter]; i++) //  
       Anzahl der eingegebenen Prozesszahl  
6   {  
       Prozess *p = new Prozess();  
       [...] // Prozess + Prozessschritte kopieren  
       A->Prozesse.append(p);  
10      QObject::connect(p, &Prozess::FortschrittUpdated ,  
        A, &Auftrag::UpdateFortschritt);  
       QObject::connect(p, &Prozess::ProzessFinished ,  
        this, &Fertigungsplanung::StationReady);  
12   }  
}
```

Abschließend wird der Auftrag der Auftragsliste angehängt, an die Datenbank gesendet und ein Auftragsitem in der Visualisierung erzeugt (Kapitel 6.3).

Standardprozesse

Folgend werden die in Abschnitt 5.6.1 erwähnten Standardprozesse näher beleuchtet.

Um ein möglichst ausgeglichenes Fertigungsbild zu schaffen wurden die fünf Standardprozesse so untereinander abgestimmt, dass diese unterschiedlichen Kriterien genügen.

5. Implementierung

Zunächst wurde darauf geachtet, dass die Prozesse unterschiedlich viele Prozessschritte beinhalten. Abbildung 5.3 kann entnommen werden, dass die Prozesse zwischen zwei und sechs einzelne Prozessschritte erfordern. In der Abbildung ist weiterhin dargestellt, welche Station als Start und Ziel je Prozessschritt angefahren wird.

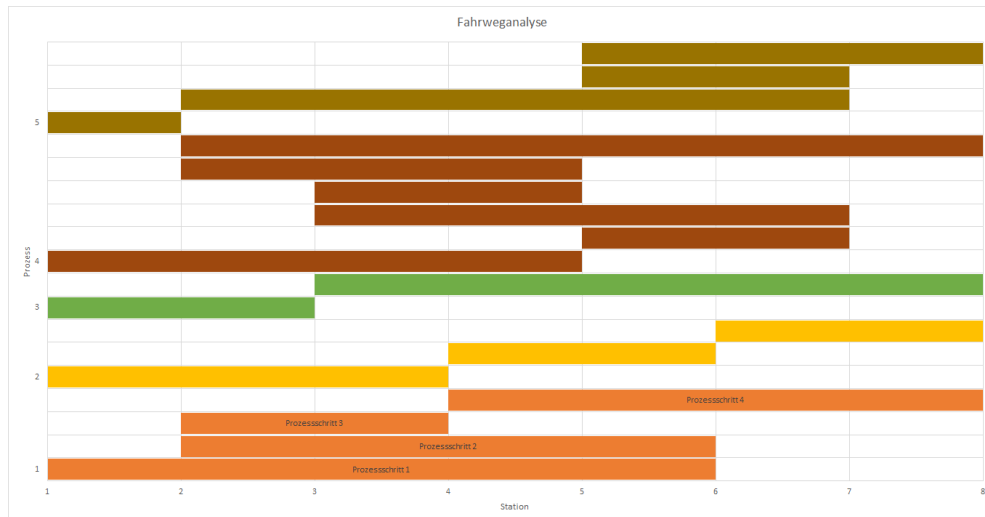


Abbildung 5.3.: Diagramm Fahrweganalyse

Zu erkennen ist, dass jeder Prozess bei Station 1 beginnt und in Station 8 endet. Die Einzelnen Prozesse (siehe farbliche Unterscheidung) sind dabei von unten nach oben zu lesen, beginnend mit Prozessschritt 1.

Aus dem Diagramm können weiterhin potentielle Kollisionsherde und „Staugefahren“ erkannt werden. Die Standardprozesse wurden so angepasst, dass alle Stationen möglichst gleichmäßig häufig durchfahren werden. Weiterhin kann aus dem Diagramm abgelesen werden, ob durch ungünstige Auftragsplanung ein Deadlock angefahren werden kann, also alle Aufträge irgendwann durch andere Aufträge blockiert sein können. Da keine Schleifen zwischen den Stationen entstanden sind, kann gewährleistet werden, dass alle Aufträge in jeder Belegung nach einiger Zeit abgearbeitet werden können, wenn die Werkstücke in Station 8 entnommen werden.

Das Diagramm in Abbildung 5.4 zeigt eine genauere Analyse der Stationen. Angenommen wird, dass jeder der Standardprozesse genau ein mal abgearbeitet wird. Auf der linken y-Achse, die zu den bunten Säulen gehört, ist die Belegungsdauer jeder Station in Sekunden angegeben. Die rechte, den hellgelben Säulen zugeordnete y-Achse bezeichnet die Anzahl der Anfahrten an eine Station.

Es ist zu erkennen, dass Station 1 und 8 genau fünf mal angefahren werden, also je Prozess ein mal. Die weiteren Stationen 2 bis 7 werden zwei- bis dreimal angefahren. So wird sichergestellt, dass die Stationen möglichst gleichmäßig belastet werden.

Die farblich unterschiedlichen Säulen zeigen die kürzeste Belegungsdauer der Arbeitsplätze an. Die Balken ergeben sich durch die Dauer eines einzelnen Prozessschrittes an einem Arbeitsplatz. Es Belegungsdauer für alle Prozesse zwischen 35

5. Implementierung

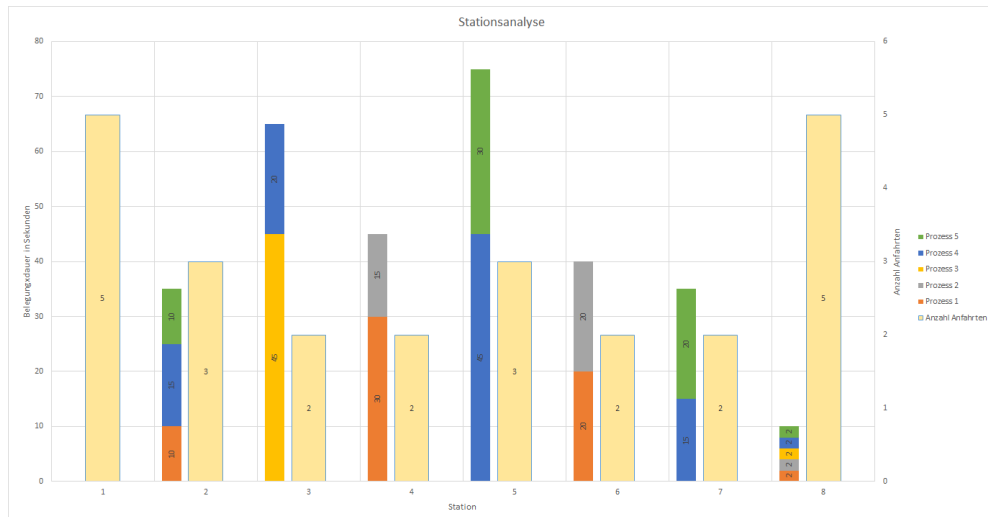


Abbildung 5.4.: Diagramm Stationsanalyse

und 75 Sekunden je Station liegt. Die 2 Sekunden Bearbeitungszeit je Prozess an Station 8 sind dabei vernachlässigt worden und gewährleisten nur ein sicheres Herausnehmen der Werkstücke durch den Menschen, ohne dass sich der Roboter noch im Entladezyklus befindet.

Als Besonderheit ist an Station 5 zu erkennen, dass es drei Anfahrten, aber nur zwei verschiedene Prozesse gibt. Dies liegt darin begründet, dass Prozess 4 Station 5 zwei mal anfährt mit verschiedenen langen Zeiten. Dadurch ist die Gesamtzeit in Station 5 auch am höchsten, gehört jedoch zu großen Anteilen zum selben Prozess. Station 2 hat die kürzesten Bearbeitungszeiten je Prozess, da diese Station von drei verschiedenen Prozessen angefahren wird. Gegenätzlich dazu ist die Bearbeitungszeit an Station 23 von Prozess 3 mit 45 Sekunden sehr hoch angesetzt, da dieser Prozess keine weitere Station anfährt.

Es wurde versucht, die durchschnittliche Belegungszeit der Stationen anzugleichen, um alle Arbeitsplätze möglichst homogen auszulasten. Zu erkennen ist, dass die minimale

Als Nebeneffekt ergibt sich, dass die Roboter möglichst ausgeglichen über die gesamte Fertigungsstraße fahren und die Gefahr von Staus oder Kollisionen dadurch verringert wird.

5.6.2. Roboter Statusänderungen

Wenn eine der in der Fertigungsplanung enthaltenen Roboterklassen ein Event feuert, welches eine Statusänderung beinhaltet (vgl. Kapitel 5.3), wird über Funktionen in der Fertigungsplanung auf diese reagiert. So wird der Greifer oder das Statusflag eines Roboters bei einer Änderung aktualisiert. Beim Empfang eines Erreignisses des Roboters wird je nach Error-Typ eine Meldung ausgegeben und individuell auf die Robotererror reagiert.

Weiterhin wird durch Ablauf des Timers eines Arbeitsplatzes dieser in der Fertigungsstraße aktualisiert und auf bereit geschaltet.

Roboterstatus geändert

Bei Änderung des Roboterstatus wird die Funktion `OnRobotStatusChanged()` aufgerufen. Dabei wird zunächst der aufrufende Roboter ausgewählt und zwischengespeichert. Sollte sich der Status des Roboters auf eins geändert haben, also der Roboter auf dem Weg zu etwas ist, so wird unterschieden ob sich der Roboter auf einem Parkplatz oder einem Arbeitsplatz befunden hat.

Von einem Parkplatz (auch Ladestation) kommend wird dieser in der Datenbank und der Visualisierung wieder freigegeben. Wenn der Roboter zuletzt an einer Station war, so wird diese jetzt für andere Roboter freigegeben und der Roboter wird in der Datenbank aus dem Arbeitsplatz gelöscht.

Greiferstatus geändert

Die Prozessaktualisierungen basieren auf der Änderung des Greifers des Roboters. Es wird vorausgesetzt, dass sich der Greifer zu keinem Zeitpunkt willkürlich öffnet oder schließt, sondern nur bei Aufnahme oder Abgabe eines Werkstücks an einer Station.

Das Schließen des Greifers eines Roboters bedeutet somit immer, dass ein Werkstück aufgenommen wurde. Somit wird in der Datenbank der Arbeitsplatz, an dem sich der Roboter befindet freigegeben und kann für andere Aufträge genutzt werden. Der Arbeitsplatz wird in der Visualisierung und der Datenbank freigegeben. Zuletzt wird der Tooltip des Arbeitsplatzes (vgl. Kapitel 6.8.1) auf einen leeren String gesetzt.

Beim Öffnen des Greifers wird ein Werkstück in einen Arbeitsplatz einer Station gelegt. Daraufhin wird die Zuordnung des Werkstücks zu dem entsprechenden Roboter gelöscht. Mit der Werkstücks-ID kann nun aus der Auftragsliste der zugehörige Prozess identifiziert werden. Der Timer des so gefundenen Prozesses wird jetzt gestartet, somit läuft die Bearbeitung des Werkstücks an dem Arbeitsplatz. In der Visualisierung und Datenbank wird der Arbeitsplatz von reserviert auf belegt aktualisiert.

Weiterhin wird das Werkstück, dass dem Roboter zu diesem Zeitpunkt zugeordnet ist an der Station an der sich der Roboter zuletzt aufgehalten hat gelöscht und der zugehörige Arbeitsplatz wird freigegeben.

Roboter Error geändert

Bei Auftreten eines Errors im Roboter wird je nach Typ eine Error-Meldung ins Log geschrieben.

Die Error-Meldungen des Roboters wurden bewusst nicht automatisiert behandelt, da dem Benutzer die Kontrolle über das Gesamtsystem behalten sollte. Somit muss je nach Fehlertyp eine unterschiedlicher Lösungsweg verfolgt werden.

Die auftretenden Fehler und der Lösungsweg ist in Kapitel 4.1.2 zu finden.

5.6.3. Hard-Code Funktionen

Die mit der Hard-Code Area erzeugten Befehle aus der Visualisierung (vgl. Kapitel 6.4.1) werden in verschiedenen Funktionen verarbeitet. So wird beispielsweise der Parkplatz oder eine Station in Datenbank und Visualisierung freigegeben, ein Roboter defekt geschaltet oder ein Arbeitsplatz als defekt markiert oder repariert.

5.6.4. Zustandsdiagramm

Das Zustandsdiagramm in Abbildung 5.5 beschreibt die Logik der Auftragsvergabe an die Roboter. Das Diagramm wurde anschließend als State Machine (vgl. 2.1.2) implementiert.

In der Abbildung sind die Zustände mit Namen dargestellt. Alle Zustände haben eine komplementäre Funktion, die nach einem Zustandswechsel als Entry-Action aufgerufen wird. Eine genaue Funktionsbeschreibung der Zustände ist in Kapitel 5.6.5 zu finden. Um zwischen den Zuständen zu wechseln werden die mit Pfeilen eingezeichneten Transitionen verwendet. Die Pfeilbeschriftung beinhaltet dabei das Signal, welches für den Zustandswechsel genutzt wird. Um das Diagramm übersichtlich zu gestalten, wurden die Timeout-Transitionen nicht vollständig eingezeichnet, sondern enden in einem kreisförmigen Xa-Zustand. Dieser führt direkt zu dem XM-Zustand oben links und leitet in den Zustand Timeout weiter.

5.6.5. State Machine Implementierung

Im Quellcode werden zunächst alle Zustände erzeugt und der State Machine hinzugefügt. Anschließend werden alle Transitionen zwischen den Zuständen erzeugt und der State Machine hinzugefügt. Hierbei werden definierte Signale genutzt, die zuvor in der Header Datei deklariert wurden. In der Main-Funktion werden abschließend die Zustände mit den entsprechenden zugehörigen Funktionen verknüpft (vgl. 5.8).

Um die State Machine zu starten wird, nachdem der Initialzustand bekannt gemacht wurde, die Start-Funktion aufgerufen. Eine verkürzte Darstellung der Aufrufe ist in Listing 5.2 abgebildet.

Listing 5.2: Start und Initialisierung der State Machine

```
2   QStateMachine StateMachine;  
   QState *StateCheckRobots = new QState;  
  
4   StateMachine.addState(StateCheckRobots);  
   StateCheckRobots->addTransition(this, SIGNAL(nextState  
       ()), StateCheckAkku);  
  
6   StateMachine.setInitialState(StateCheckRobots);  
8   StateMachine.start();
```

10

```
QObject::connect(StateCheckRobots, &QState::entered,
                 this, &Fertigungsplanung::CheckRobots);
```

Zustand CheckRobots

Im Initialzustand der State Machine, CheckRobots, werden alle vier Roboter nacheinander auf Bereitschaft überprüft. Die Reihenfolge der zu überprüfenden Roboter wird dabei durch einen hierfür entwickelten Shuffle-Algorithmus (5.6.6) bestimmt.

Ein Roboter gilt als bereit, wenn er als Status 0 zurückgibt, nicht als defekt markiert ist und als lebend gilt. Sobald ein geprüfter Roboter alle drei Kriterien erfüllt wird dieser ausgewählt und in den weiteren Zuständen, gefolgt von CheckAkku, genutzt. Die anderen Roboter werden bis zum nächsten Aufruf des Zustands CheckRobots vernachlässigt.

Sollte keiner der vier Roboter alle drei Kriterien erfüllen wird in den Zustand CheckParkplatzRobots gewechselt.

Zustand CheckParkplatzRobots

Wie im Zustand CheckRobots werden alle vier Roboter anhand von Kriterien auf Bereitschaft überprüft. Dazu wird ebenfalls die Reihenfolge zufällig bestimmt.

Anstatt den Roboterstatus wie beim Zustand CheckRobots auf 0 zu prüfen, wird ein Status zwischen 201 und 204 erwartet. Diese vier Stati repräsentieren die vier Parkplätze auf denen sich ein Roboter befinden kann.

Durch die Aufteilung der Zustände CheckRobots und CheckParkplatzRobots erfolgt eine Priorisierung der Roboter, die sich bereits in den Stationen befinden und einen Auftrag fertig abgearbeitet haben. Nur wenn kein Roboter in den Stationen verfügbar ist wird auf die auf den Parkplatz befindlichen Roboter zurückgegriffen. Dadurch werden Situationen vermindert, in denen sich die Roboter bei der Stationsarbeit gegenseitig behindern.

Sollte keiner der auf den Parkplatz befindlichen Roboter alle weiteren Kriterien erfüllen wird in den Zustand Wait gewechselt.

Zustand Wait

Da in Qt aufgrund der eventbasierten Programmierung (vgl. 2.1.1) keine Endlosschleifen vorgesehen sind wurde im Zustand „Wait“ mittels Timer die Zustandsmaschine künstlich verlangsamt. Ein Entfernen des Zustands führt zum Programmabsturz, da in kürzester Zeit die interne Eventliste voll geschrieben wird. Auf externe Events wie Mauseingaben oder Ereignisse vom Betriebssystem könnte somit nicht mehr reagiert werden.

5. Implementierung

Durch den Zustand Wait können andere Programmevents und Ereignisse des Betriebssystems abgearbeitet werden, während 2000 ms auf das timeout()-Signal des Timers gewartet wird.

Nach Ablauf des Timers wird in den Initialzustand CheckRobots gewechselt.

Zustand CheckAkku

Der Zustand CheckAkku gewährleistet einen Tiefentladungsschutz der Roboter. Sobald ein Roboter in den ersten beiden Zuständen ausgewählt wurde wird überprüft, ob sein Akkustand größer gleich 25 Prozent ist. Es wird davon ausgegangen, dass kein Auftrag plus eine Fahrt zum Laden mehr als 25 Prozent Akku verbraucht. Der Wert wurde empirisch ermittelt.

Sollte der Akkustand unter 25 Prozent liegen, wird der Roboter über Zustand SendLoadingTask zu einer freien Ladestation geschickt. Wenn keine Ladestation frei ist wird in den Zustand Wait gewechselt. Bei einem Akkustand von über 25 Prozent wird der Zustand CheckAuftrag aufgerufen.

Zustände SendLoadingTask, SendParkingTask, SendPositionTask

Um bei Kommunikation mit den Robotern in den Zuständen SendLoadingTask, SendPositionTask und SendParkingTask sicherzustellen, dass die Nachricht empfangen wurde, wird diese mehrfach gesendet, bis die Übertragung erfolgreich war (siehe Kapitel 4.1.1). Es wird nach jedem Sendevorgang ein kurzer Wartezustand aufgerufen, der, gleich dem Zustand Wait, einen Programmabsturz verhindert. Mit dem individuellen Wartezustand kann ebenfalls das bei aktuell 700 ms angesetzte Sendeintervall eingestellt werden.

Aus allen den drei Send.....Task Zuständen und ihren zugehörigen Wartezuständen kann über ein Ablauf eines Timers in den Zustand Timeout gewechselt werden. Der Timer wird beim erstmaligen Eintritt in einen der Zustände Send.....Task gestartet und läuft nach 25 Sekunden ab. Dieser Timer gewährt eine Absturzsicherheit des Programms bei Roboterfall oder anderweitigem Fehlschlagen der Kommunikation.

In allen drei Send.....Task Zuständen wird ein erfolgreicher Sendevorgang damit erkannt, dass der Roboterstatus nicht mehr 0 ist, und der Roboter sich nicht mehr auf einem Parkplatz befindet. Solange die beiden Bedingungen nicht erfüllt sind wird weiter an den Roboter gesendet.

Wenn im Zustand SendLoadingTask die Nachricht erfolgreich versendet wurde, wird der Timer für das Timeout gestoppt, und die benötigte Ladestation in der Visualisierung und in der Datenbank reserviert. Abschließend wird in den Initialzustand gewechselt.

Der Zustand SendParkingTask verhält sich Analog dem SendLoadingTask, nur wird bei erfolgreicher Nachrichtenversendung der benötigte Parkplatz reserviert.

5. Implementierung

Bei dem Zustand `SendPositionTask`, in dem ein Auftrag aus der Planung an den Roboter verschickt wurde, werden nach erfolgreichem Nachrichtenversand mehrere Aktionen durchgeführt. Zunächst wird der Timeout-Timer gestoppt. Sollte der Auftrag den ersten Prozessschritt, also das Abholen eines Werkstücks im Lager an Station 1, beinhalten wird der Schreibkopf des RFID neu beschrieben. In der Datenbank wird außerdem dem Start- und Ziellarbeitsplatz der genutzte Roboter zugeordnet. Am Startarbeitsplatz wird das vorhandene Werkstück in der Datenbank entfernt und dem Ziellarbeitsplatz zugeordnet. Die beiden Stationen werden außerdem reserviert um zu verhindern, dass ein anderer Roboter an die zu bearbeitenden Stationen geschickt wird. Somit ist eine erste Kollisionsvermeidung implementiert. Zuletzt wird in der Datenbank das genutzte Werkstück dem Roboter zugewiesen. Auch in der Visualisierung werden Start- und Ziellarbeitsplatz reserviert. Schlussendlich wird in der Visualisierung der Tooltip (vgl. Kapitel 6.8.1) des Start- und Ziellarbeitsplatz aktualisiert.

Zustand `CheckAuftrag`

Der Zustand `CheckAuftrag` enthält die Vergabe der Aufträge an die Roboter. Dazu wird zunächst in der Auftragsliste jeder Auftrag auf unfertige Prozesse untersucht und diese folgend an eine neu erzeugte Prozessliste angehängt (vgl. Abschnitt 5.5). Jeder Prozess der so erzeugten Prozessliste wird anschließend auf die Möglichkeit der Bearbeitung geprüft. Sofern eine der folgenden Prüfungen fehlschlägt wird der nächste Prozess geprüft.

Dazu wird zunächst überprüft ob die Auftragsplanung pausiert ist oder sich der Prozess schon in Bearbeitung befindet.

Es wird geprüft ob die Start und Zielstation nicht durch einen Roboter reserviert ist.

Sollte der nächste Prozessschritt der bearbeitet werden muss an der Lagerstation (Station 1) sein, so wird gewährleistet, dass sich ein Werkstück im Lager befindet und dieses ausgewählt.

Es wird geprüft ob eine der beiden Ziellarbeitsplätze der Zielstation frei ist, also weder reserviert noch defekt und anschließend ausgewählt.

Wenn alle Vorbedingungen für einen der Prozesse eintreffen wird der Zustand direkt verlassen und der Auftrag wird an den ausgewählten Roboter im Zustand `SendPositionTask` gesendet.

Sollten alle Prozesse zu keinem Sendevorgang geführt haben, das heißt es ist entweder kein Auftrag vorhanden oder das senden wurde blockiert, so wird der aktuell ausgewählte Roboter zum Parken geschickt (Zustand `SendParkingTask`), sofern er sich noch nicht auf einem Parkplatz befindet. Sonst wird in den Initialzustand gewechselt.

5.6.6. Shuffle-Algorithmus

Der Shuffle-Algorithmus dient dazu, die Roboter in zufälliger Reihenfolge abzu-
arbeiten, um eventuelle Dead-Locks zu vermeiden. Ein solcher Dead-Lock könnte
zum Beispiel eintreten, wenn Roboter 1 und 2 an der Ladestation sind, Roboter 3
zum laden geschickt werden müsste, jedoch warten muss. Roboter 4 würde dabei,
selbst wenn er bereit wäre einen Auftrag abzuarbeiten nie überprüft werden.

Zurückgegeben wird eine Liste in der die Zahlen 1 bis 4 in zufälliger Reihenfolge
vorliegen. Der Algorithmus ist optimiert gegenüber der Anzahl an Systemaufrufen
für eine Zufallszahl. Andernfalls wäre es einfacher solange eine Zufallszahl
zurückgeben zu lassen, wie die Liste diese noch nicht enthält.

Um die Roboterreihenfolge festzulegen werden vier Schritte durchgeführt. Im ers-
ten Schritt wird eine leere Liste erzeugt und eine zufällige Zahl zwischen 1 und 4
an die erste Stelle geschrieben (Listing 5.3 Zeile 1-3).

Listing 5.3: Shuffle-Algorithmus

```

2      QList<int> liste ;
      int i = QRandomGenerator::global()->bounded(1, 5);
      liste.append(i);
4      i = QRandomGenerator::global()->bounded(1, 4);
      (liste.contains(i)) ? liste.append(i+1) : liste.append
        (i);
6      i = QRandomGenerator::global()->bounded(1, 3);
      if (liste.contains(i))
8      {
          if (liste.contains(i+1))
10         {
             liste.append(i+2);
12         }
          else
14         {
             liste.append(i+1);
16         }
      }
18     else
      {
20         liste.append(i);
      }
22
24     if (!liste.contains(1)) {liste.append(1);}
      else if (!liste.contains(2)) {liste.append(2);}
      else if (!liste.contains(3)) {liste.append(3);}
26     else if (!liste.contains(4)) {liste.append(4);}
      return liste ;

```

Im zweiten Schritt wird eine Zufallszahl zwischen 1 und 3, wenn sie noch nicht
enthalten ist, in die Liste geschrieben oder um eins inkrementiert und in die Liste

5. Implementierung

geschrieben (Listing 5.3 Zeile 4f).

Um die nächste Zahl hinzuzufügen wird im dritten Schritt (Listing 5.3 Zeile 6-21) eine Zufallszahl zwischen 1 und 2 erzeugt und in die Liste geschrieben, sollte sie nicht vorhanden sein. Wenn sie schon existiert wird sie um eins inkrementiert und erneut überprüft ob die inkrementierte Zahl in der Liste ist. Aufgrund der Maximalgröße von 4 kann der Algorithmus so alle Fälle abdecken.

Zuletzt wird die noch fehlende Zahl der Liste ergänzt (Listing 5.3 Zeile 22ff) und die Liste zurückgegeben.

5.7. Mainwindow

5.8. Main

StateMachine Connections beschreiben / erwähnen - genau eine funktion zu einem State

5.9. Weitere Klassen

5. Implementierung

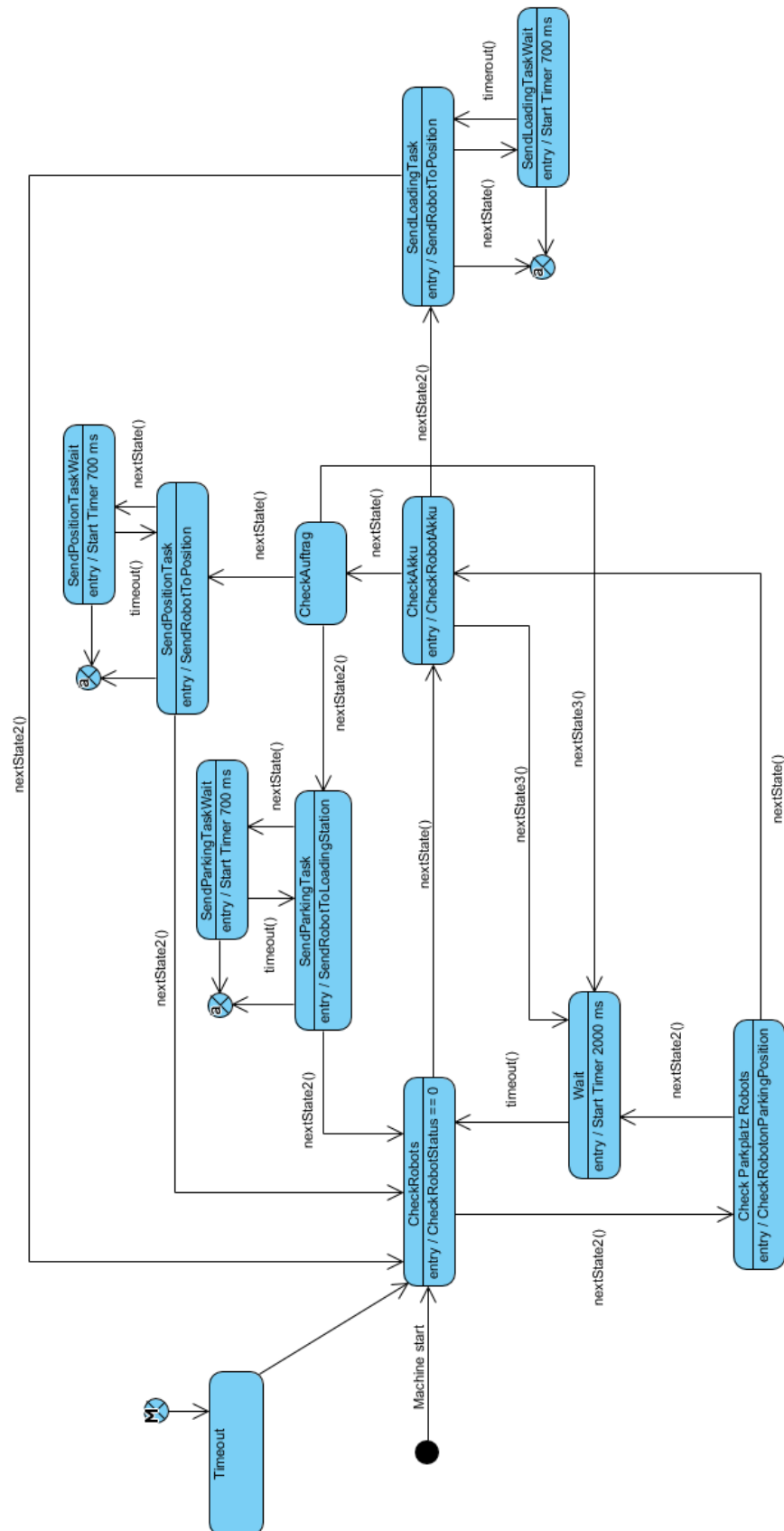


Abbildung 5.5.: Zustandsdiagramm der Auftragsplanung

6. Visualisierung

6.1. Grundstruktur

Coorporate Design

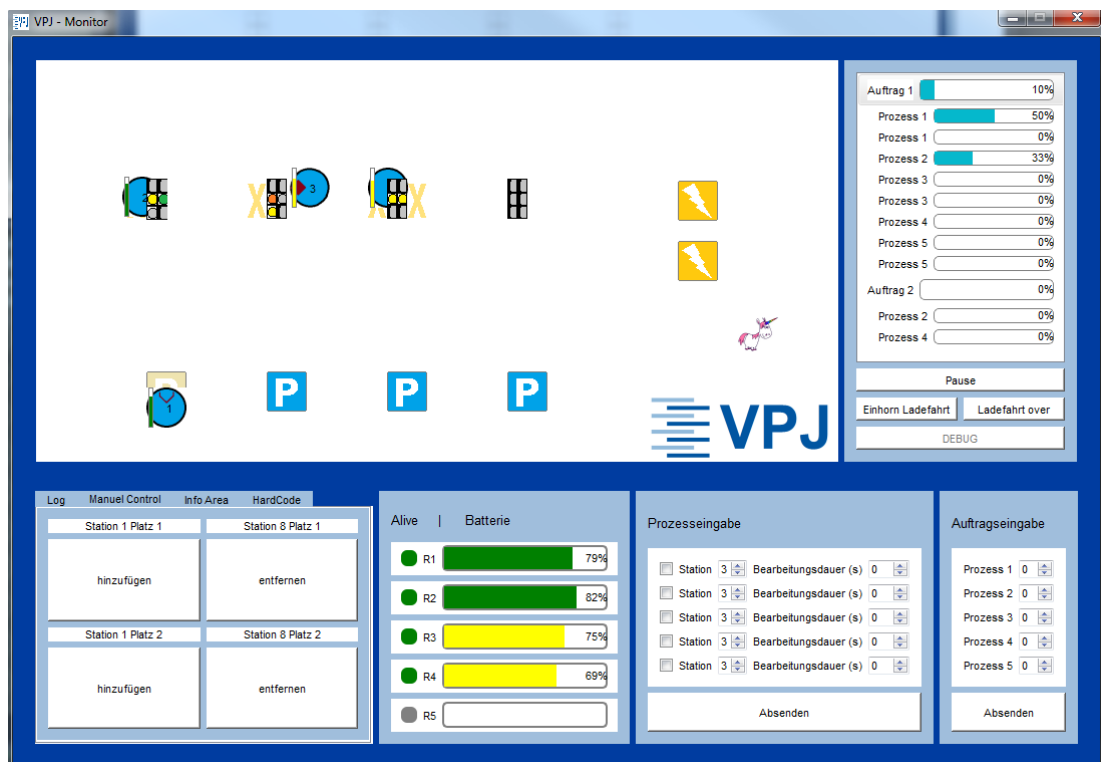


Abbildung 6.1.: Übersicht Visualisierung

6. Visualisierung

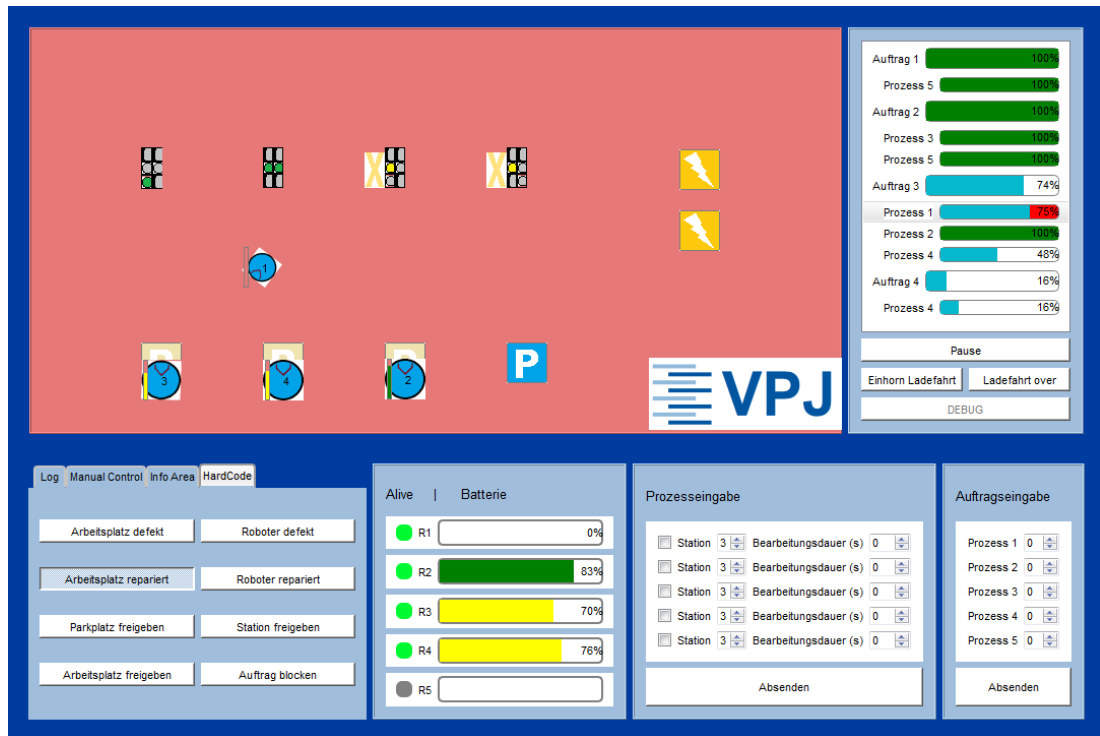


Abbildung 6.2.: Übersicht Visualisierung im Hard-Code Modus

6.2. Live-View

6.3. Auftragsübersicht

6.4. Tab-View

6.4.1. Hard-Code Bereich

6.5. Roboterstatus

6.6. Prozesseingabe

6.7. Auftragseingabe

6.8. Benutzerinteraktion

6.8.1. Tooltips

Tooltips mit Bildern her und erläutern

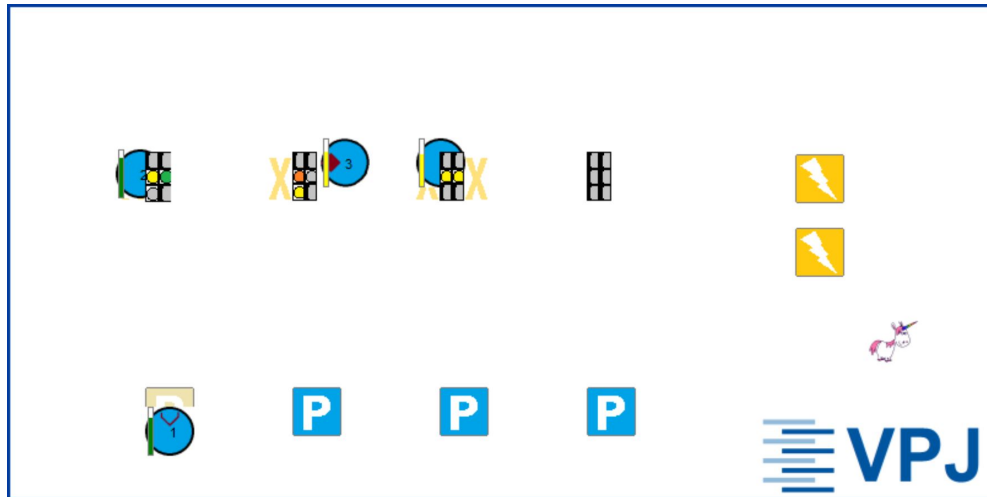


Abbildung 6.3.: Visualisierung Live-View



Abbildung 6.4.: Visualisierung Parkplatz

In Kapitel 1 Ähnlich einer Smart Kamera, bei der nicht nur das Kamerabild, sondern auch weitere schon verarbeitete Informationen ausgegeben werden sollen, wird ein Prozessor benötigt, welcher in der Lage ist Bildverarbeitung durchzuführen. Es wurde als Möglichkeit für ein Prozessorsystem empfohlen, sich mit dem BeagleBone Black und Raspberry PI auseinanderzusetzen. Mithilfe einer Marktrecherche werden zunächst weitere geeignete Kamerasysteme gesucht und anschließend anhand der Eignung sortiert. Dabei wird auch eine geeignete Schnittstelle zu dem Kamerasystem ausgewählt, mit dem die Bilder und Informationen an den PC gesendet werden. Nachdem die beiden geeignetsten Kamerasysteme gewählt und bestellt sind, werden zugehörige Gehäuse und Befestigungen konstruiert und gefertigt. Nebenbei werden erste einfache Testprogramme geschrieben, um die Funktion von den gewählten Kamerasystemen zu gewährleisten. Der erste Prototyp wird dabei voraussichtlich mit Hilfe rapid-prototyping ein Erzeugnis aus dem 3D-Druck sein, um Kamera an der richtigen Position zu halten. Nachdem



Abbildung 6.5.: Visualisierung Ladestation

6. Visualisierung

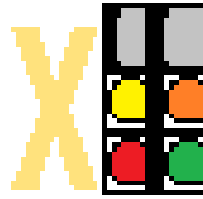


Abbildung 6.6.: Station mit verschiedenen Arbeitsplatzstati



Abbildung 6.7.: Roboter in verschiedenen Stati (vlnr: Greifer offen, Greifer geschlossen, Defekt)

das Endprodukt montiert ist, folgt eine Auswertung der Kamerabilder. Folgend folgen optional das Erstellen einer Bibliotheksdatei und eine Automation der Kalibrierung und des Weißabgleichs. Der Zugriff auf die Kamera soll gewährleistet sein.

6. Visualisierung

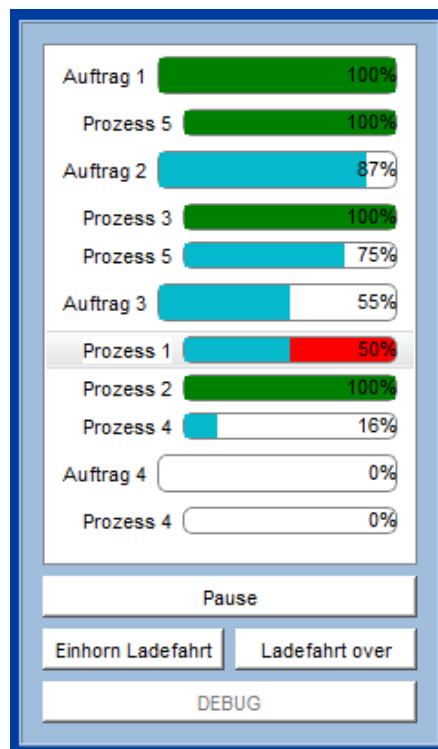


Abbildung 6.8.: Auftragsfortschritt

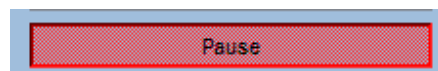


Abbildung 6.9.: Pause Button gedrückt

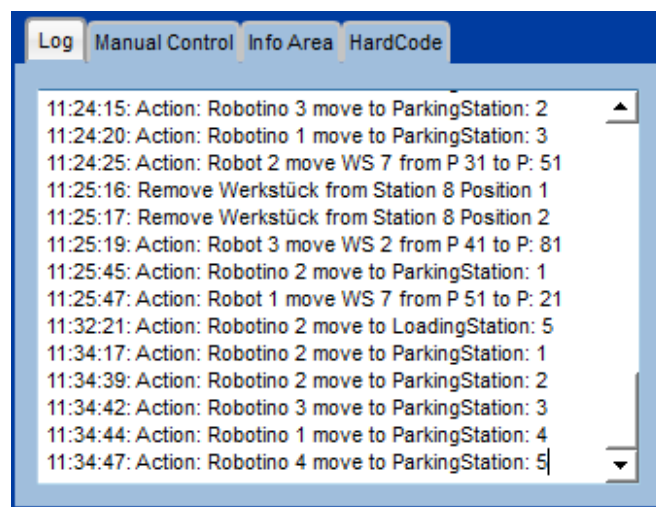


Abbildung 6.10.: Tab: Log View

6. Visualisierung

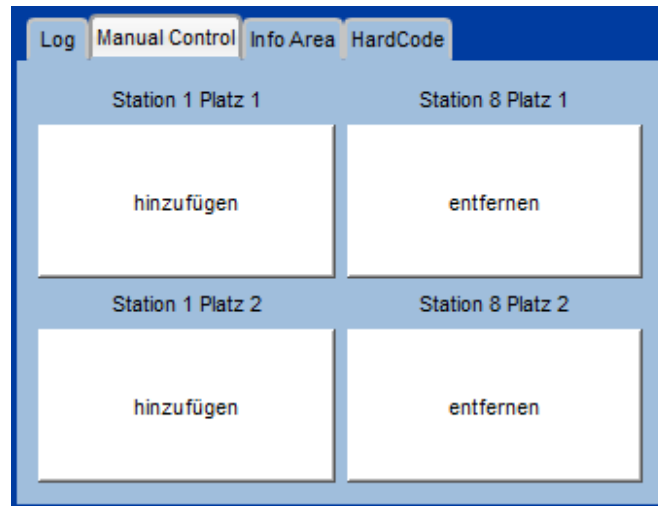


Abbildung 6.11.: Tab: Manual Control

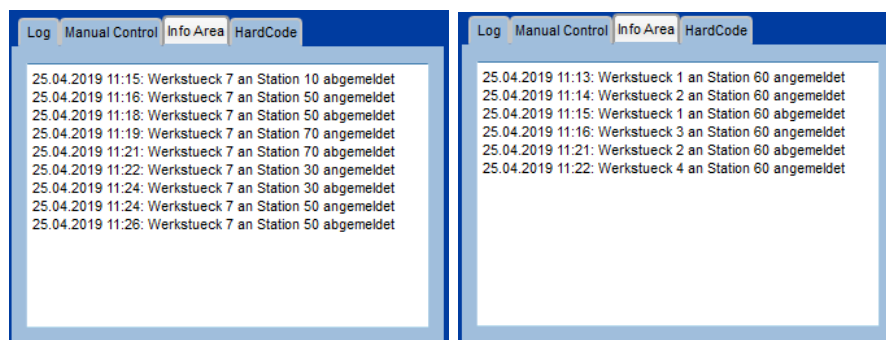


Abbildung 6.12.: Tab: Info Area

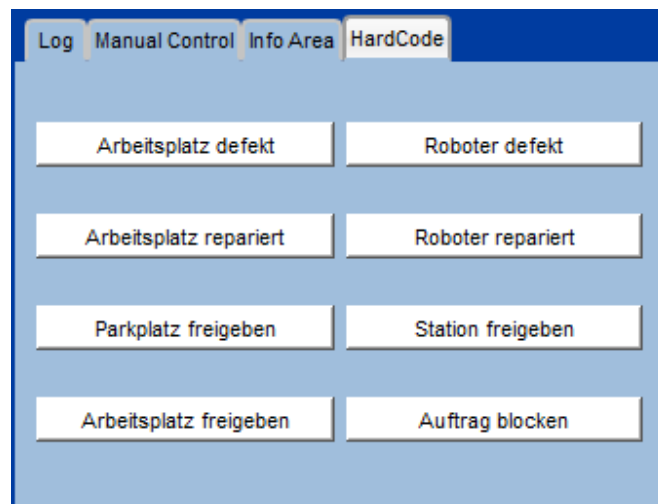


Abbildung 6.13.: Tab: Hard-Code

6. Visualisierung

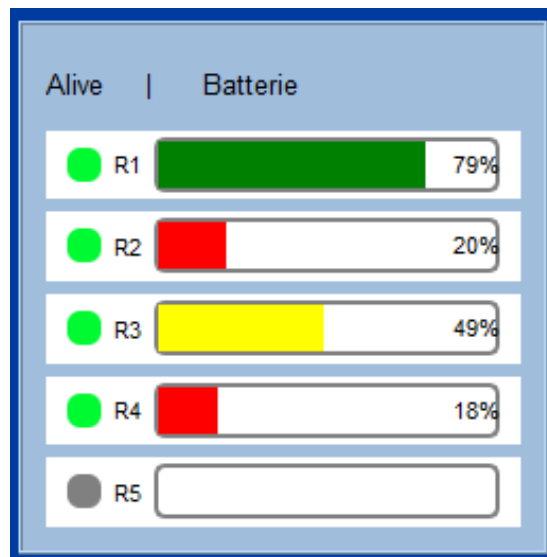


Abbildung 6.14.: Batterie und Statusanzeige

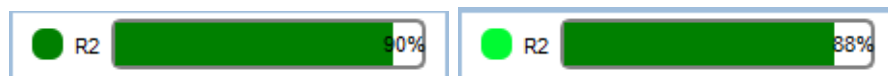


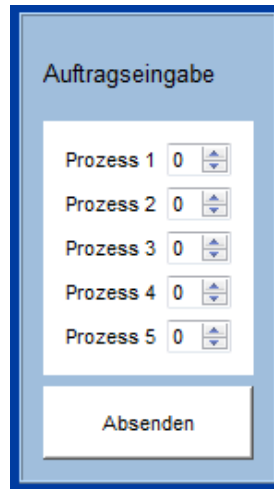
Abbildung 6.15.: Roboterstatus-LED blinkend

Abbildung 6.16. zeigt eine Benutzeroberfläche mit dem Titel 'Prozesseingabe'. Darunter sind fünf Zeilen für die Eingabe von Stationen und Bearbeitungszeiten. Jede Zeile enthält ein Kontrollkästchen, eine Stationennummer, einen Pfeil für die Stationennummer und eine Bearbeitungszeit in Sekunden. Ein 'Absenden' Button befindet sich am unteren Rand.

Station	Bearbeitungsdauer (s)
<input checked="" type="checkbox"/> Station 3	20
<input checked="" type="checkbox"/> Station 4	0
<input type="checkbox"/> Station 3	10
<input type="checkbox"/> Station 5	5
<input type="checkbox"/> Station 6	0

Absenden

Abbildung 6.16.: Visualisierung - Prozesseingabe



Auftragseingabe

Prozess 1	0
Prozess 2	0
Prozess 3	0
Prozess 4	0
Prozess 5	0

Absenden

Abbildung 6.17.: Visualisierung - Auftragsvergabe



Abbildung 6.18.: PixyCam Gehäusefehler

7. Simulation

Um verschiedene Programmabläufe und Funktionen zu testen war es wichtig, auch ohne realen Roboter die standardmäßige Programmfunktionalität darstellen zu können. Die Klasse `SimulatedRobot` erfüllt genau diese Anforderung.

Mittels einer `simulated`-Flag kann in der `Main`-Funktion ein Simulationsbetrieb gestartet werden. In diesem ist kein realer Roboter notwendig, und trotzdem findet eine normale Auftragsplanung- und Abarbeitung statt. Die `simulated`-Flag wird in der Initialisierung dem UDP-Handler übergeben, der die Kommunikation mit den Robotern übernimmt. Somit wird in der Simulation kein Socket erzeugt und verbunden (siehe 5.3), sondern die Klasse `SimulatedRobot` genutzt.

`SimulatedRobot` enthält eine State-Machine mit 13 Zuständen, die den Roboter ausreichend nachbilden. Weiterhin werden Funktionen für das simulierte Senden des Roboters an den UDPHandler und Timer für die Zustandswechsel bereitgestellt.

Wie im echten Roboter wird der aktuelle Auftrag und Auftragstyp zwischengespeichert.

7.0.1. Zustandsdiagramm simulierter Roboter

Das Zustandsdiagramm in Abbildung 7.1 beschreibt die Zustände und Transitionen eines simulierten Roboters. Das Diagramm wurde anschließend als State Machine (vgl. 2.1.2) implementiert.

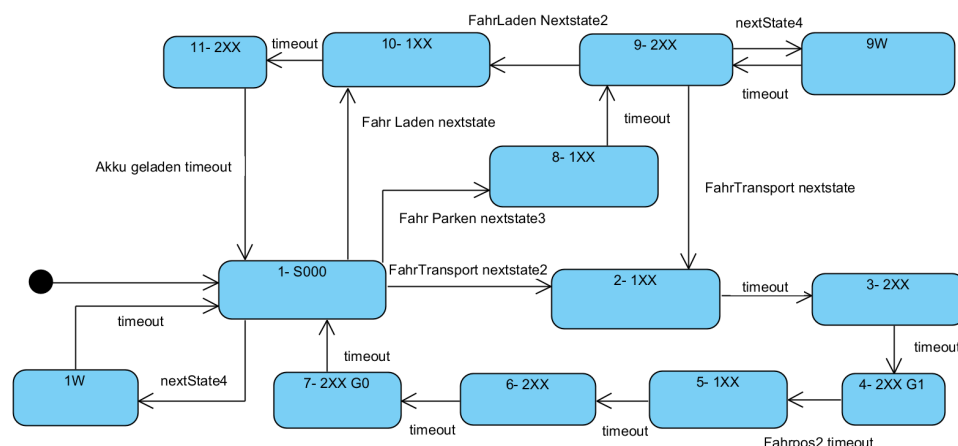


Abbildung 7.1.: Ablaufdiagramm des simulierten Roboters

7. Simulation

Da in der Fertigungsplanung für die Vergabe der Aufträge die tatsächliche Position des Roboters nicht berücksichtigt wird, wird im simulierten Roboter ausschließlich die für die Planung benötigte Status- und Greiferinformation nachgepflegt.

Die Änderung der Statusinformation ist im Diagramm in den Zuständen als Zahl mit zwei folgenden Dont-Care (XX) gekennzeichnet, da die Position keine weitere Relevanz hat. Eine Änderung des Greifers ist mit G0 für Greifer öffnet sich oder G1 für Greifer schließt sich gekennzeichnet.

Zur Initialisierung wurden ähnlich der State Machine aus Kapitel 5.6.5 zunächst alle Zustände und Transitionen hinzugefügt, verknüpft und die State Machine gestartet.

Der Initialzustand 1 kann über Empfang eines Auftrags zum Laden zu Zustand 10, zum Parken zu Zustand 8 oder zum Transport zu Zustand 2 verlassen werden. Außerdem wird das aktive Abfragen eines Auftrags an den Roboter alle 3 Sekunden über den Zustand 1W getriggert. Über den Wartezustand wird ein bearbeiteter noch anliegender Auftrag zurückgesetzt.

Der Ablauf innerhalb der State-Machine zwischen zwei Aufträgen erfolgt über fest definierter Timer, die Zustandswechsel bewirken und ein Senden des Roboterstatus an den UDP-Handler hervorrufen.

Da innerhalb von QT keine Events verloren gehen kann auf ein periodisches Senden, wie der echte Roboter es tut verzichtet werden.

Simulierte Ladefahrt

Wenn in Zustand 1 am simulierten Roboter ein Ladefahrtauftrag anliegt, also ein Auftrag mit der ID 3, so wird direkt in Zustand 10 gewechselt. Es wird an den UDP-Handler ein Status 100 zurückgegeben (auf dem Weg zur Ladestation) und ein Timer von 3 Sekunden gestartet. Nach Ablauf des Timers wird in Zustand 11 der Status 2XX zurückgegeben, was bedeutet, dass die Ladestation erreicht wurde. Nach Ablauf weiterer 5 Sekunden gilt das Laden als beendet und der Roboter geht in Zustand 1 über und sendet Status 000.

Simulierter Parkvorgang

Bei einem Parkauftrag in Zustand 1, also ein Auftrag mit der ID 2, wird in Zustand 8 gewechselt. An den UDP-Handler wird, da der simulierte Roboter jetzt auf dem Weg zum Parken ist, als Status 100 zurückgegeben. Nach einer Fahrzeit von 3 Sekunden hat der Roboter sein Ziel erreicht und gibt in Zustand 9 als Status 200 an den UDP-Handler.

Zustand 9 kann entweder über einen Auftrag mit der ID 3 in Zustand 10 verlassen werden, und eine Ladefahrt wird simuliert, oder mit einem Auftrag und der ID 1 einen Transportauftrag in Zustand 2 verlassen werden.

7. Simulation

Über den Wartezustand 9W kann auf weitere Aufträge reagiert werden. Dieser wird alle 3 Sekunden aufgerufen.

Simulierter Transport

Sobald ein simulierter Roboter in Zustand 2 kommt wird an den UDP-Handler der Status 100 übermittelt, da der Roboter sich auf dem Weg zur ersten Auftragsposition befindet. Nach 3 Sekunden wird in Zustand 3 der Status 2XX gesendet, da der Roboter an der ersten Position angekommen ist. Eine weitere Sekunde später wird das Werkstück gegriffen, was durch Senden des Greiferstatus 1 in Zustand 4 übermittelt wird. In Zustand 5 wird über Ablauf eines zweisekündigen Timers der Status 100 gesendet, da sich der Roboter auf dem Weg zur zweiten Station des Auftrags befindet.

In Zustand 6, der nach 3 Sekunden bearbeitet wird, wird zunächst der Status auf 2XX gesetzt, da der Roboter angekommen ist. Nach einer Sekunde wird über Zustand 7 der Greifer geöffnet und der Greiferstatus auf 0 zurückgesetzt. Zwei Sekunden später wird wieder in Zustand 1 auf einen neuen Auftrag gewartet.

8. Datenbank

Abbildungsverzeichnis

1.1. CMUcam5 Pixy	
Quelle: http://charmedlabs.com/default/products/	1
1.2. PixyCam Gehäusefehler	2
5.1. Klassendiagramm	6
5.2. Visualisierung des Rückgabewertes der Funktionen getNextUnfinishedProzessstep() und GetActualProzessstep()	8
5.3. Diagramm Fahrweganalyse	11
5.4. Diagramm Stationsanalyse	12
5.5. Zustandsdiagramm der Auftragsplanung	20
6.1. Übersicht Visualisierung	21
6.2. Übersicht Visualisierung im Hard-Code Modus	22
6.3. Visualisierung Live-View	23
6.4. Visualisierung Parkplatz	23
6.5. Visualisierung Ladestation	23
6.6. Station mit verschiedenen Arbeitsplatzstati	24
6.7. Roboter in verschiedenen Stati (vlnr: Greifer offen, Greifer geschlossen, Defekt)	24
6.8. Auftragsfortschritt	25
6.9. Pause Button gedrückt	25
6.10. Tab: Log View	25
6.11. Tab: Manual Control	26
6.12. Tab: Info Area	26
6.13. Tab: Hard-Code	26
6.14. Batterie und Statusanzeige	27
6.15. Roboterstatus-LED blinkend	27
6.16. Visualisierung - Prozesseingabe	27
6.17. Visualisierung - Auftragsvergabe	28
6.18. PixyCam Gehäusefehler	28
7.1. Ablaufdiagramm des simulierten Roboters	29

Literaturverzeichnis

- [Qt 19] The Qt Company Ltd. *The State Machine Framework*, 2019. Stand am 09.05.2019:
<https://doc.qt.io/qt-5/statemachine-api.html>.
- [Sch09] Schober. *Festo Dokumentation Handbuch MPS Transfersystem*. Festo Didactic, 2009.

A. Inhalt der CD

- Dieses Dokument als PDF „VPJ.pdf“
- Alle konstruierten Bauteile sowohl als STL-, als auch als CATIA-Part-Datei im Ordner „Konstruktion“
- Den Quellcode aller Anwendungen im Ordner „Code“
 1. config.ini
 2. config_bandposition.py
 3. config_farbton.py
 4. config_referenzposition.py
 5. config_playground.py - als Template und zum Testen von Einstellungen
- Alle Abbildungen der Arbeit im Ordner „Abbildungen“
- Ein Video „LED_Video.mp4“ zur Veranschaulichung der LEDs des Raspberry Pi
- Ein Video „Raspberry Pi Test.mp4“ zur Veranschaulichung der Anwendung des Raspberry Pi auf dem Festo-Transfersystem
- Ein Video „Pixy Test auf Anlage.mp4“ zur Veranschaulichung der Anwendung der PixyCam auf dem Festo-Transfersystem
- Marktrecherche ImageProcessingPlatform.pdf
„How to choose the best embedded processing platform for on-board UAV image processing ?“ von Dries Hulens, Jon Verbeke und Toon Goedem