Projektseminar Echtzeitsysteme

Ausarbeitung von Team TRAL
Proseminar eingereicht von
Tim Burkert, Robert Königstein, Lars Stein, Adrian Weber
am 8. April 2016

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Echtzeitsysteme

Elektrotechnik und Informationstechnik (FB18)

Zweitmitglied Informatik (FB20)

Prof. Dr. rer. nat. A. Schürr
Merckstraße 25
64283 Darmstadt

www.es.tu-darmstadt.de

Gutachter: Prof. Dr. rer. nat. A. Schürr
Betreuer: Géza Kulcsár, MSc.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

Die Themengebiete Fahrassistenzsysteme und vor allem autonomes Fahren sind heute aktueller denn je. Zum einen, da vieles längst keine Zukunftsmusik mehr darstellt und schon in unserem alltäglichen Leben angekommen ist. Hierzu zählen Systeme zur Überwachung toter Winkel, vom elektrischen Spiegel bis hin zur Rundumsicht im Bird-View-System. Nicht nur Warnsysteme, die den Fahrer auf gefährliche Situationen hinweisen, sondern auch halbautomatische Piloten, die korrigierend eingreifen, sind Realität. Beispiele hierfür sind Lenk- und Parkassistenten, wie auch automatische Notbremssysteme, die nicht nur statische Hindernisse erkennen, sondern sogar die Bewegung kreuzender Autos vorausberechnen können. Hierbei scheint sogar oftmals nicht die Technik, sondern die Rechtslage das begrenzende Element darzustellen.
Zum anderen, da mit jeder weiteren Funktionalität die Gesamtkomplexität eines möglicherweise autonom fahrenden Fahrzeugs noch deutlicher vor Augen geführt wird. Es werden weitere Probleme erkannt, die beachtet und absolut zuverlässig gelöst werden müssen, da im Falle eines Ausfalls oder Fehlverhaltens des Systems Menschenleben auf dem Spiel stehen. Um eine Situation vollständig zu erkennen und korrekt auszuwerten, ist die Zusammenarbeit mehrerer unterschiedlicher Sensoren nötig. So werden Autos mit (Stereo-) Kameras, Ultraschall und sogar Radar ausgestattet, um auf unterschiedliche Distanz und Richtung Aussagen über die Umgebung machen zu können. Dabei müssen natürlich alle Berechnungen vor Ort und in Echtzeit durchgeführt werden.

1.2 Das Projektseminar Echtzeitsysteme

Bereits seit einigen Jahren bietet daher das Fachgebiet Echtzeitsysteme das gleichnamige Projektseminar an. Schwerpunkte waren dabei stets (halb-) autonomes Fahren und Car2X Kommunikation. Realisiert wurde dies mittels eines bzw. mehrerer Modellauto-Chassis mit Motoren und Sensoren, gesteuert von einem 16-Bit Mikrocontroller. Um den gesteigerten Anforderungen Rechnung zu tragen, wurden erstmalig in diesem Semester die Fahrzeuge mit einer weiteren Platine ausgestattet, die aufgrund des verbauten Prozessors komplexere und rechenaufwändigere Aufgaben ermöglichten. So zum Beispiel Videoverarbeitung zusammen mit der ebenfalls neu angebrachten Kamera. So wurde auch das Spektrum der Aufgabenstellung erweitert und erstreckt sich nun von geregeltem Fahren mit Abstandssensoren bis hin zum Anfahren eines Ziels, wobei das Ziel selbstständig mit der Kamera erfasst werden muss. Dabei war die genaue Realisierung der Anforderungen bewusst freigestellt. Auch weitere vertiefende Themen konnten von den Teams selbst ausgewählt werden. Dies ermöglichte ein sehr eigenständiges Arbeiten in der Gruppe, da hier in großem Umfang und für jedes Team individuell, sowohl die Interessen, als auch die Kompetenzen berücksichtigt werden konnten.

Ein übergeordneter Gedanke war dabei stets die Nachhaltigkeit, damit die Teams in kommenden Semestern von den Ergebnissen profitieren und im Idealfall darauf aufbauen können. Diese Nachhaltigkeit ist sehr wichtig und stellt auch ein Kernthema unserer Ergebnisse dar. Fernziel des Projektseminars ist das Erarbeiten einer Plattform, die eine Teilnahme am *Carolo Cup*[1] oder ähnlichen Veranstaltungen ermöglicht.

[1] https://wiki.ifr.ing.tu-bs.de/carolocup/

2 Zur Verfügung gestellte Hardware des Autos

Die Basis der zur Verfügung stehenden Hardware, also Chassis, Motoren, Servo, etc. sind handelsübliche Modellautos. Dabei besteht das Auto aus zwei Schichten. Die Basis bildet ein 16-Bit Mikrocontroller der MB96300-Serie von *Fujitsu Microelectronics* (heute: *Cypress Semiconductor Corporation*). Die zweite Schicht der Hardware des Autos besteht aus einem vollwertigen Einplatinencomputer mit Quad-Core-Prozessor, SSD-Speicher uvm. Der Mikrokontroller ist in der zu diesem Semester upgegradeten Hardware nur noch zur Vermittlung der Werte von Sensorik und Aktorik zwischen Hauptplatine und Peripherie zuständig. Somit hatten wir eine vollwertige API zur Hardware, sodass keine Software für Schnittstellen oder Treiber implementiert werden musste und (fast) keine Veränderungen am Mikrocontroller vorgenommen werden mussten. Als Sensoren stehen ein Hall-Sensor am Hinterrad, ein Liniensenor, eine USB-Webcam, sowie drei Ultraschallsensoren (rechts, links und vorne) zur Verfügung. Angesteuert wird die Lenkung des Modellautos über einen Servo-Motor. Der Antrieb des Autos ist durch einen Gleichstrommotor, der durch einen Fahrtregler fast stufenlos angesteuert werden kann, realisiert. Beides wird durch ein pulsweitenmoduliertes Signal vom Mikrokontroller gesteuert. Zudem ist ein WLAN-Modul integriert, sodass es möglich ist, auf das Auto per Ad-hoc-Verbindung zuzugreifen.

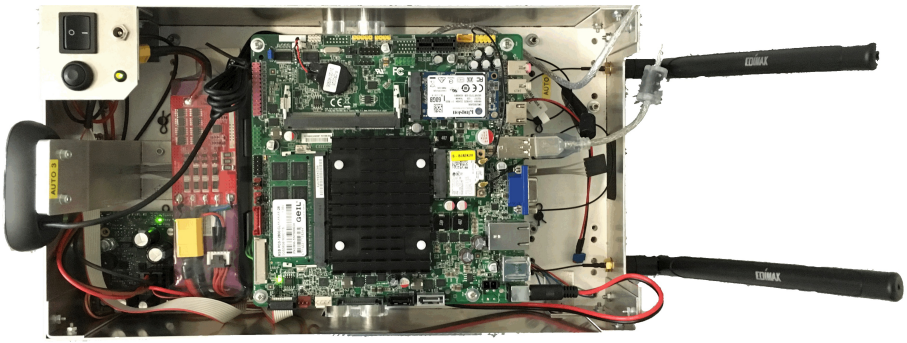

Abbildung 2.1: Das Auto

Die Software auf dem Mikrokontroller, die zur Kommunikation zwischen der Anwendungssoftware und der Sensorik bzw. Aktorik zuständig ist, ist bereits vorimplementiert. Durch die so zur Verfügung gestellte API war eine einfache Nutzung der Hardware möglich. Jedoch war dadurch auch unklar, inwieweit die Signale der Sensorik manipuliert wurden. Die Ultraschallsensoren waren zu Beginn auf cm-Auflösung eingestellt, was zu ungenauen Sensorwerten führte. Die abbildbare Tiefe betrug etwa 20 cm bis 3 m mit vielen Ausreißern und Diskrepanzen durch eine niedrige Auflösung. Um bessere Werte zu erhalten, haben wir auf dem Mikrokontroller von cm-Werte auf Mikrosekunden-Werte umgestellt. Dies hat die Genauigkeit, sowie den Messbereich erhöht.

3 Grundlegendes zu *ROS* und Lubuntu

Im Folgenden wird eine kurze Einleitung zu ROS[1] und dem Betriebssystem, das auf dem Auto installiert war, Lubuntu, gegeben. Diese Grundausstattung an Software war bei allen Gruppen identisch.

3.1 Was ist *ROS*?

Im Projektseminar wird die Middleware *ROS* eingesetzt. *ROS* steht für *Robot Operating System* und wird heute vor allem von der *Open Source Robotics Foundation* fortentwickelt. Die Software ist dabei kein klassisches Betriebssystem, wie der Name vermuten lässt, sondern eine Middleware, die auf einem der klassischen Betriebssysteme (aktuell werden Mac-OS und Linux stabil unterstützt) aufgespielt wird. *ROS* sorgt für eine starke Hardware-Abstraktion, sodass fast beliebige Hardware eingesetzt werden kann und diese auch fast beliebig austauschbar ist. *ROS* abstrahiert diese und stellt allgemeine Programmierschnittstellen bereit. Zudem ermöglicht *ROS* eine einfache, standardisierte Kommunikation zwischen den Hard- und Softwarekomponenten. Die drei wesentlichen Faktoren *ROS* einzusetzen sind Modularität, Portabilität und Wiederverwendbarkeit, sowie eine vereinfachte Softwareentwicklung (einfache Interfaces, Debugging, Monitoring, Testen).

Abbildung 3.1: Logo von *ROS*

3.1.1 Wie ist *ROS* aufgebaut?

ROS besteht hauptsächlich aus drei Komponenten, die beliebig kombiniert und vernetzt werden können: *Nodes*, *Topics* und *Services*. *Nodes* sind Softwareknoten, die dafür zuständig sind, bereitgestellte Daten aufzunehmen und zu prozessieren. In den *Nodes* ist die Intelligenz des Autos implementiert und dort werden durch die Sensorik gewonnene Daten in Befehle für die Aktorik des Autos umgesetzt. *Topics* sind asynchrone Kommunikationslösungen, mit denen die Knoten Daten und Nachrichten, sog. *Messages*, austauschen können. Dabei sind Produktion und Konsumption der Daten getrennt, indem *Messages* an einer Stelle *gepublished* werden, und anschließend dem gesamten System zum Abruf zur Verfügung stehen. Andere Knoten können diese Nachrichten nun Empfangen (*subscribe*). Es ist also völlig unerheblich, wo genau die Daten herkommen, was bedeutet, das beliebig viele *Nodes* die *Topics publishen* können und beliebig viele sie wiederum *subscriben* können. *Services* sind synchrone Kommunikationsmöglichkeiten. Während *Topics* einen n:m-Nachrichtenaustausch ermöglichen, sind Services dazu da, einen direkten 1:1-Nachrichtenaustausch zwischen zwei *Nodes* zu ermöglichen. Ein *Service* besteht dabei aus einem Nachrichtenpaar von Anfrage und Antwort auf diese.

[1] http://www.ros.org/

3.2 Betriebssystem

Zu dem vorgegebenen Software Framework des Projektseminars gehörte neben *ROS* ebenfalls das Betriebssystem *Lubuntu* (auch mit Echtzeitkernel). *Lubuntu* ist ein Derivat des Linux-Betriebssystems *Ubuntu*, das *LXDE* als Desktop-Umgebung verwendet. Linux ist allgemein bekannt, weshalb wir an dieser Stelle nicht mehr weiter darauf eingehen möchten. Zu bemerken ist jedoch, dass bei uns der Echtzeitkernel nicht zum Einsatz kam, da die Hardware des Autos durch unsere Implementierung nur zu ca. 30% ausgelastet war.

3.3 Welche Möglichkeiten von *ROS* haben wir genutzt?

Aufgrund unserer Codestruktur eines Zustandsautomaten, den wir in einem *ROS-Node* implementiert haben, kamen wir mit wenigen *ROS-Nodes* aus. Wie später beschrieben, haben wir lediglich drei *Nodes* für das generelle Management der Sensorik und Aktorik, der Kamera und für die FSM (Finite State Machine/Zustandsautomat) benötigt. Wie gerade beschrieben, nutzen wir für die Kommunikation zwischen den einzelnen Codebausteinen ausschließlich *Topics*. Zur Kamerakalibrierung haben wir eine Kombination der von *ROS* bereitgestellten Kalibrierungsmöglichkeiten und Methoden der *OpenCV*-Bibliothek genutzt. Später in der Ausarbeitung werden wir noch ausführlicher auf die Details der Implementierung und Umsetzung eingehen.

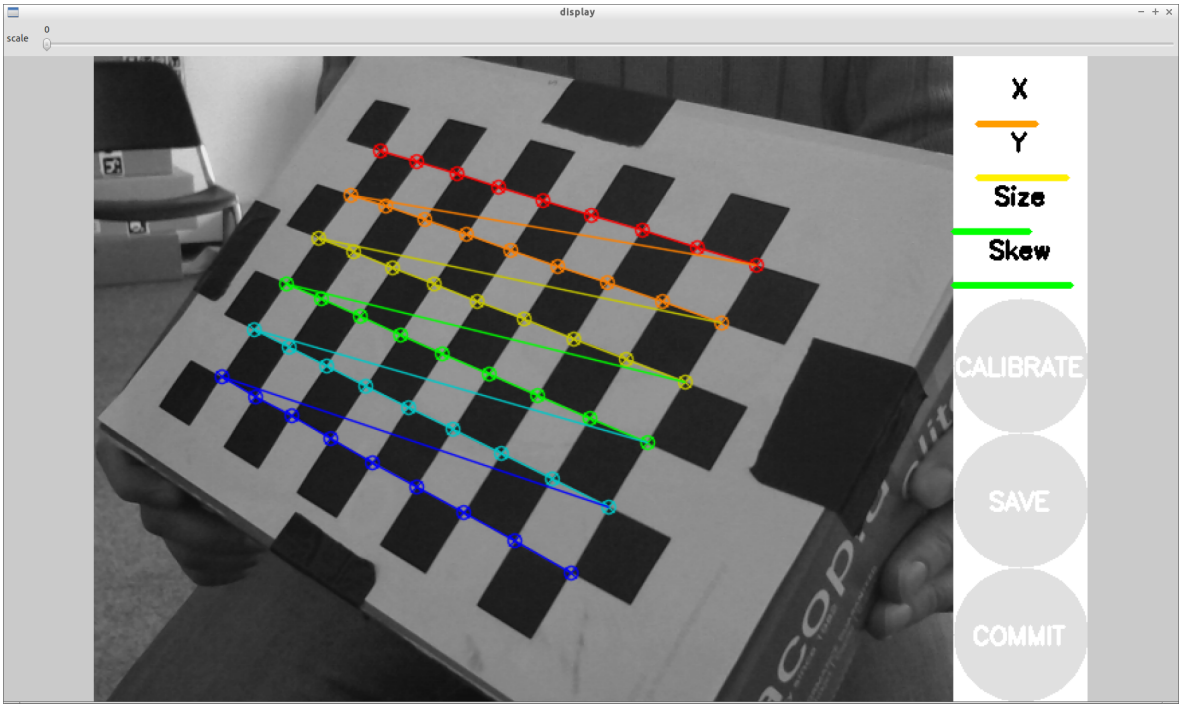

Abbildung 3.2: Kamerakalibrierung mit *ROS*

4 Arbeitsumgebung und Material

Als Arbeitsort stand uns ein studentischer Arbeitsraum des Fachgebiets Echtzeitsysteme zur Verfügung. Hier wurden die Fahrzeuge aufbewahrt und es waren Arbeitsplätze mit Bildschirmen vorhanden, um die Autos oder eigene Laptops anzuschließen.
Die Testfahrten fanden jedoch meist in den Fluren statt. Hier konnten die langen Wände zur Orientierung genutzt werden und es stand etwas mehr Platz zur Verfügung, als im Arbeitsraum. Auch waren von den Vorjahren bereits Querstreifen auf dem Boden angebracht, beispielsweise vor Kurven oder Kreuzungen. Das autonome Fahren gestaltete sich auf den Fluren interessanter, weil hier der recht hohe Wendekreis der Fahrzeuge nicht so ins Gewicht fiel. Außerdem stellten die Flure naturgemäß schon eine realistische Umgebung mit 90° Kurven, Kreuzungen und Hindernissen dar, die von jeder Gruppe individuell genutzt wurde. So war es praktisch keine Einschränkung, dass wir teilweise zur genaueren Positionsbestimmung eine Wand in messbarer Nähe voraussetzten. Geeignete Hindernisse und Tore, bestehend aus Pappkartons mit *ArUco*-Markern, ließen sich natürlich ebenfalls beliebig aufstellen und somit konnten wir immer unterschiedliche Szenarien kreieren. Diese *ArUco*-Marker sind optisch vergleichbar zu QR-Codes. Zudem besteht hier der Vorteil, dass bereits Bibliotheken existieren, die das Erkennen und Auswerten der Marker implementieren.

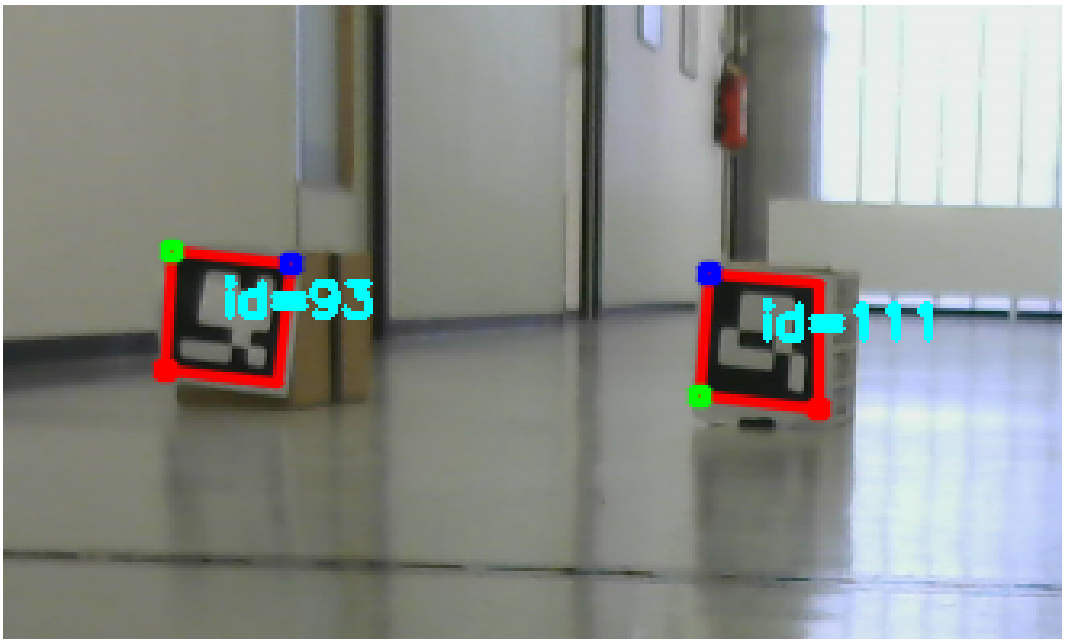

Abbildung 4.1: *ArUco*-Tor aus Sicht der Kamera

Dabei können nicht nur die *Ids* der einzelnen Marker ausgelesen werden, sondern auch deren Position im Raum. In der gezeigten Abbildung sehen wir zwei *ArUco*-Marker aus Sicht der Fahrzeugkamera.

5 Projektkoordination

Nicht nur die technischen Voraussetzungen, sondern auch die Organisation im Team stellte einen wichtigen Faktor für den Erfolg dar.

5.1 Verantwortungsbereiche

Zunächst analysierten wir genau sowohl die uns gegebenen technischen Möglichkeiten, als auch die Aufgabenstellung. Daraufhin verteilten wir Verantwortungsbereiche auf die einzelnen Gruppenmitglieder. Einige dieser Bereiche waren:

- Code - Verantwortlich für eine lesbare Struktur und Kommentare im Code
- Zeitmanagement - Verantwortlich für Zeitplanung und Einhaltung der Fristen
- Dokumentation - Verantwortlich für die schriftliche Dokumentation der Team-Absprachen
- Vorträge, LateX und einige mehr

Es handelte sich dabei ganz bewusst um Verantwortungsbereiche und nicht um Aufgabenteilung. Idee hierbei war, dass die einzelnen Gebiete nicht ausschließlich von dem jeweils Verantwortlichen beachtet oder bearbeitet wurden, sondern die jeweils Zuständigen darauf achteten, dass alle den entsprechenden Rahmen einhielten. So konnte sich jeder auf seine Verantwortlichkeiten konzentrieren, ohne fürchten zu müssen andere Aspekte zu vergessen. Auf diese Weise war eine gegenseitige Kontrolle und Erinnerung an die Einhaltung der Aufgaben gegeben. Jedes Teammitglied konnte sich stets darauf verlassen, dass eine wichtige Information notiert und an alle Fristen und Termine, wie die zweiwöchentlichen Treffen mit dem Betreuer, erinnert wurde.

5.2 Zeitplan und Meilensteine

Um einen Zeitplan erstellen zu können, notierten wir zunächst alle extern vom Veranstalter vorgegebenen Ziele mit der entsprechenden Deadline als Meilensteine in unserem Zeitplan. Dazu zählen unter anderem die Endergebnisse und auch die beiden Vorträge. Nachdem wir uns mit der Technik vertraut gemacht hatten, konnten wir auch Zwischenziele mit der benötigten Zeit und der daraus folgenden Deadline abschätzen. Daraus ergab sich ein Zeitplan, der die gesamte Bearbeitungszeit über Gültigkeit besaß.

5.3 Dokumentation: *Trello*

Wie bereits erwähnt, legten wir von Anfang an Wert darauf, Absprachen, Entscheidungen, aber auch offene Fragen immer schriftlich festzuhalten. Dies sollte natürlich möglichst übersichtlich dargestellt, aber auch stets für alle zugänglich sein. Wir entschieden uns für **Trello**[1]. Dies ist ein Online-Organisationsboard, zugänglich über eine Website, womit alle Informationen immer aktuell an einem Ort vorlagen. In Trello können einzelne Listen aus Karten erstellt werden um die Themen zu gliedern. Während Listen die Oberthemen repräsentieren, stellen die Karten die Unterpunkte zu den einzelnen Themenbereichen dar.

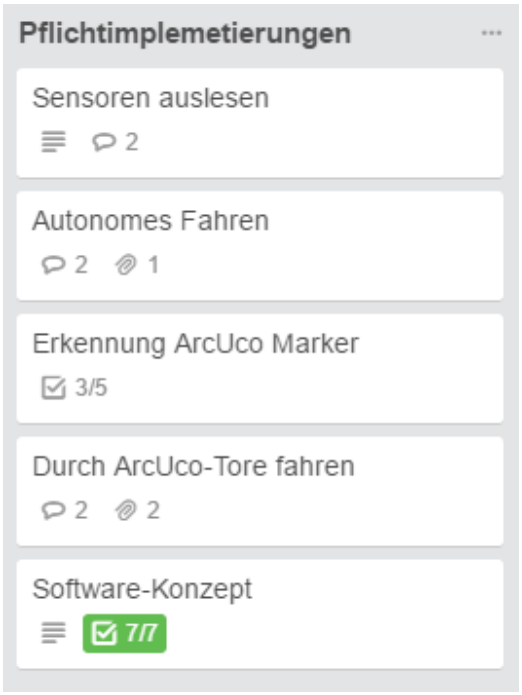

Abbildung 5.1: Trello-Liste mit Karten

Dabei werden in der Liste jeweils die Titel der Karten zusammen mit den wichtigsten Informationen angezeigt. So weisen Sprechblasen auf Kommentare und Heftklammern auf Dateianhänge hin. Auch die Anzahl der erledigten Unterpunkte wird angezeigt und bei vollständiger Bearbeitung grün markiert.

[1] https://trello.com/

Wählt man nun eine Karte aus, kann man alle weiteren Informationen einsehen. Zuordnungen zu Teilnehmern lassen sich hier realisieren, um, zum Beispiel, Aufgabenteilung oder Verantwortlichkeiten aufzuzeigen. Unterpunkte kann man abhaken, was sich an einem Fortschrittsbalken ablesen lässt. Kommentare mit angehängten Dateien lassen sich ebenso realisieren, wie Verweise auf Karten aus anderen Listen, um Zusammenhänge darzustellen und schnellen Zugriff zu ermöglichen. Am Ende findet sich unter *„Aktivität“* ein Log mit allen Ereignissen dieser Karte zusammen mit dem entsprechenden Zeitstempel.

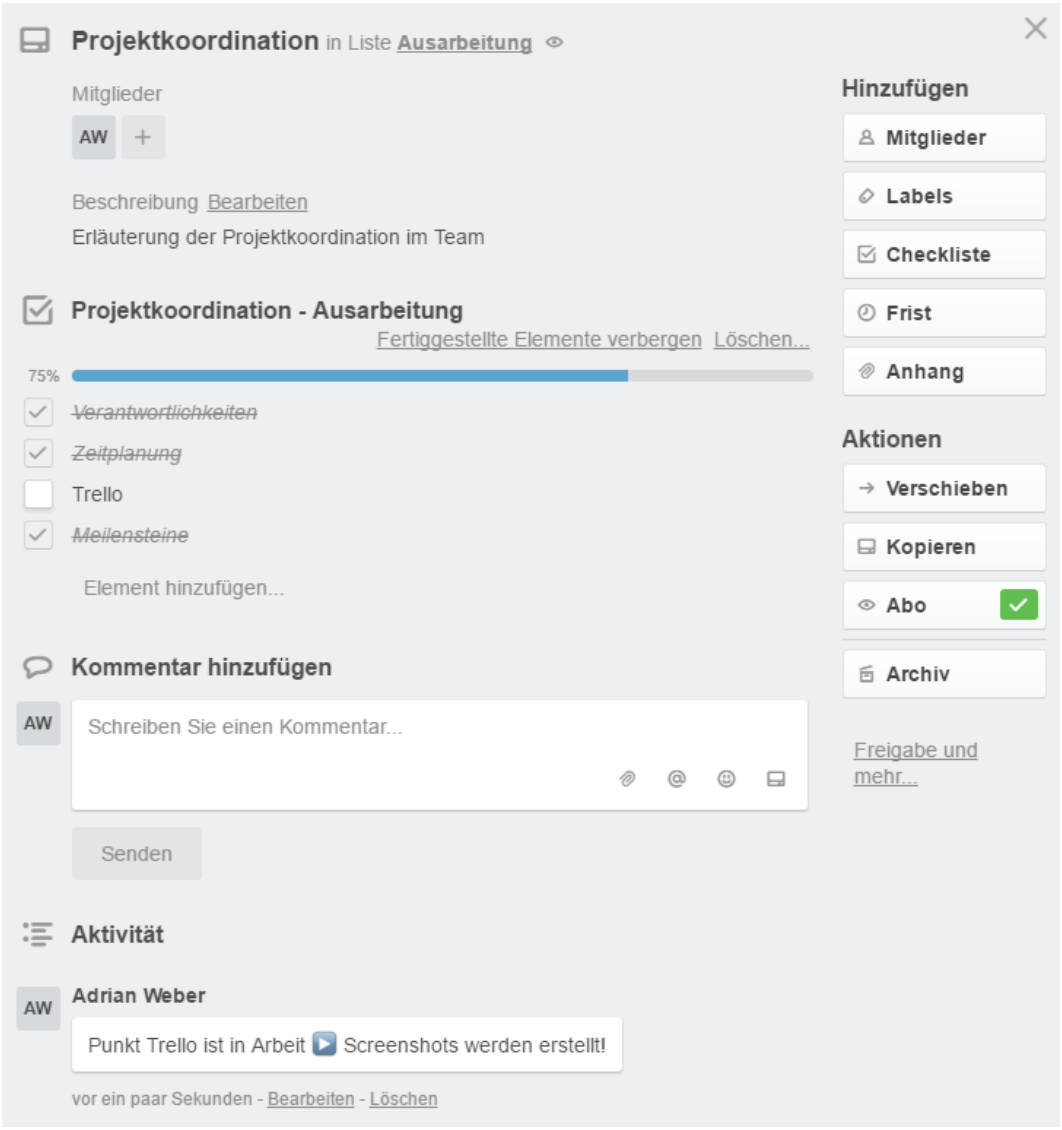

Abbildung 5.2: *Trello*-Karte mit Fortschrittsbalken und weiteren Informationen

6 Aufgabenstellung

Im Folgenden soll die genaue Aufgabenstellung, sowie unsere daraus abgeleiteten Themen für das Projektseminar erläutert werden.

6.1 Pflichtimplementierung

Als Pflichtteil des Projektseminars wurde generell das Thema „Autonomes Fahren mittels Sensorik“, „Kamerainbetriebnahme und Erkennung von *ArUco*-Markern“, sowie als Anwendung von Letzterem, das „Durchfahren von *ArUco*-Toren“ vorgegeben.
Aus dem Standardprogramm abgeleitet haben wir aufgrund der örtlichen Rahmenbedingungen festgelegt, dass sich unser Auto autonom in der vorgegebenen Umgebung bewegen können soll. Wir haben folgende Annahmen zur Aufgabenstellung getroffen:

- Das Auto hat mindestens eine Wand rechts oder links von sich, zu der es seinen Abstand mittels eines Ultraschallsensors absolut bestimmen kann
- Es wird lediglich mit der vorgegebenen Hardware eine Realisierung des autonomen Fahrens durchgeführt, sodass unser Code auch von nachfolgenden Gruppen einfach genutzt werden kann
- Das autonome Fahren wird durch einen Zustandsautomaten realisiert
- *ArUco*-Tore bestehen aus zwei Markern, die in beliebigem Abstand (jedoch begrenzt durch den Kamerablickwinkel) voneinander nebeneinander im Flur stehen
- Der Winkel von der Verbindungsgerade zwischen den Markern und den Normalen auf den Markern ist nahe 90°
- Es ist kein starkes Gegenlicht vorhanden, da dies den Kontrast der Kamera und damit die Erkennung der *ArUco*-Marker, stark beeinträchtigt
- Beim Durchfahren von *ArUco*-Toren befindet sich entweder rechts oder links des Autos eine Wand, zu der es seinen Abstand mittels Ultraschallsensorik absolut bestimmen kann

6.2 Vertiefungspakete

Als optionale Vertiefungspakete wurden „*ROS*-basierte Simulation“, „Fernsteuerung und Car2Car Kommunikation“, sowie „Inertialsensorik und erweiterte Regelung“ vorgeschlagen.
Von den vorgeschlagenen Möglichkeiten haben wir die Fernsteuerung, sowie die erweiterte Regelung aufgegriffen. Bezüglich der Fernsteuerung haben wir im Rahmen der Hobit-Berufsbildungsmesse eine Steuerung mittels XBOX-Controller eingebunden. Dieses Projekt wurde jedoch nicht weitergehend verfolgt, da es einen anderen Kernel benötigte, als unsere Hauptimplementierung. Als erweiterte Regelung haben wir uns als Ziel gesetzt, eine Regelung nach einer linearen Funktion zu implementieren, mittels welcher auch die Durchfahrt der *ArUco*-Tore realisiert werden soll. Dazu haben wir die

Annahme getroffen, dass sich das Auto rechts oder links entlang einer Wand bewegt, zu der der Abstand linear zu- oder abnehmen soll.
Als Schwerpunkt haben wir uns jedoch als Ziel gesetzt, ein System zur modularen Programmierung des Autos zu entwickeln, das es ermöglicht, beliebige Funktionsbausteine zu programmieren und einfach zur Funktionalität des Autos hinzuzufügen. Dies schafft eine größtmögliche Flexibilität und Erweiterbarkeit der Software des Autos. Zur Implementierung des Systems haben wir uns für einen Zustandsautomaten entschieden, der, um Logik und Implementierung zu trennen, mit einer *JSON*-Datei (genaueres zu *JSON*, s. Abschnitt 8) parametrisiert werden soll. Aufbauend auf oben beschriebene Funktionalität und um das Verhalten des Autos noch einfacher festzulegen, wurde im Laufe des Projektseminars das Ziel der graphischen Programmierung mittels eines *UML*-Tool (genaueres zur graphischen Programmierung, s. Abschnitt 9) entwickelt.

7 Lösung Pflichtimplementierung

Wie bereits im Abschnitt „Aufgabenstellung“ beschrieben, war es Gegenstand der Pflichtimplementierung, dass das Auto autonom mittels Sensorik fahren kann, die Kamera *ArUco*-Marker erkennt, sowie das Auto durch ein Tor aus solchen hindurchfahren kann. Bereits in obigem Abschnitt wurden die Rahmenbedingungen, welche wir für unsere Lösung angenommen haben definiert. Im Folgenden wird nun erläutert, wie wir, auf Grundlage des Zustandsautomaten, dessen genaue Implementierung erst später erläutert wird, die drei Hauptaufgaben realisiert haben.

7.1 Autonomes Fahren

Unser Konzept zum autonomen Fahren baut sehr stark auf den Möglichkeiten, die uns eine FSM bietet, auf. Das Konzept beruht grundlegend darauf, dass das Auto eine FSM lädt, die entweder per JSON-File oder per graphischer Oberfläche konfiguriert wird. Die FSM ist ein Graph aus verschiedenen Zuständen, die jeweils eine Funktion des Autos, wie z.B. geradeaus an einer Wand entlang fahren, repräsentieren. Diese verschiedenen Funktionen oder Zustände des Autos sind über Transitionen verbunden, d.h. Ereignisse, die zu einem Zustandswechsel führen. Das kann z.B. eine abrupte Abstandsänderung der führenden Wand sein.
Mittels der Kombination aus Zuständen, in denen sich das Auto befinden kann und Ereignissen, die zu einem Zustandswechsel führen, kann ein autonomes Verhalten, je nach Umgebung des Autos realisiert werden. Alternativ kann ein Weg in einem Bekannten Umfeld einprogrammiert werden, indem eine Folge von Zuständen konkateniert wird. Der Basiszustand für das autonome Fahren ist ein einfacher Wandfolger-Zustand, der das geregelte Geradeausfahren entlang einer Wand realisiert.

Ein Regelkreis sorgt dafür, dass ein System in einen stabilen Zustand überführt wird. Dazu wird eine Eingangsgröße, in diesem Fall der Abstand zur Wand, in einen Regelkreis geführt. Ausgangsgröße ist dann der Lenkeinschlag, welcher den Abstand des Autos zur Wand beeinflusst. Um nun den Wandabstand stabil zu halten, wird die Differenz zwischen der Regelgröße (Lenkeinschlag) und Führungsgröße (Wandabstand) bestimmt (Regelabweichung). Da die einzelnen Regelkreisglieder ein Zeitverhalten haben, muss der Regler den Wert der Abweichung verstärken, sowie das Zeitverhalten unterdrücken. Anfangs haben wir versucht, das Regelverhalten durch einen PID-Regler abzubilden, jedoch führten Tests und Hinweise der Gruppe *Nullpointer* des vergangenen Semesters zu einer Vernachlässigung des *I*-Anteils. Generell sorgt der *P*-Anteil für eine proportionale Verstärkung der Regelabweichung. Der *D*-Anteil sorgt für ein differenzierendes Verhalten, ist also abhängig von der Änderungsgeschwindigkeit der Regelabweichung. Der vernachlässigte *I*-Anteil sorgt für eine zeitliche Integration der Regelabweichung. Dies führt bei dem Auto zu einer sich mit höherem *I*-Anteil einstellenden Trägheit, die das Auto unflexibler macht.
Der Basiscode, mit dem der Regler implementiert wird, ist in der Klasse `BasicFollowWall` zu sehen:

```
double y = _p*e + _esum*_i*PID_S + _d*(e - _eold)*PID_INVS;
```

Abbildung 7.1: Reglerimplementierung, `BasicFollowWall.cpp`, Z.53

Hierbei stellen `PID_S` und `PID_INVS` Konstanten dar (0.0125 und 80). `_p` ist der Proportionalanteil, `_i` der Integralanteil und `_d` der Differentialanteil. e ist die Eingangsgröße, `_esum` die aufaddierten Wandabstände der vergangenen Durchläufe und `_eold` der Wert des letzten Durchlaufes. y ist der daraus berechnete Lenkeinschlag.

Mittels des oben vorgestellten *PD*-Reglers wird im Zustand `BasicFollowWall` das Geradeausfahren implementiert. Erste Tests haben gezeigt, das entgegen der Erwartungen auch Kurven mit dem gleichen Zustand umfahren werden können. Daraus ergab sich dann die Einsicht, dass kein neuer Kurvenzustand eingeführt werden musste. Es lassen sich also alle grundlegenden Bewegungen durch eine Konkatenation verschieden parametrisierter Wall-Follow Zustände erreichen. Zudem hat sich herausgestellt, dass Kurven umso besser funktionieren, wenn unmittelbar vorher die Geschwindigkeit gedrosselt wird und im Anschluss ein größerer Wandabstand gewählt wird, da sonst die Gefahr besteht, dass die Ultraschallsensoren einen zu steilen Winkel zur Wand haben. Nun ist es also möglich mit einem einzigen Zustand `BasicFollowWall` die grundlegenden Bewegungen (Geradeausfahren, Kurvenfahren) abzudecken. Als Transitionen zwischen den Zuständen haben wir vier Optionen implementiert: `Always, Distance, Finished, SensorLevel`.

- `Always`: Diese Transition feuert sofort.
- `Distance`: Diese Transition feuert nach der übergebenen Distanz. Die aktuelle Distanz wird immer über die Hall-Sensoren aus dem aktuellen Maschinen-Zustand (`MachineState`) ausgelesen.
- `Finished`: Um zu feuern, muss die Methode `isFinished()` des vorangegangenen Zustands `true` zurückgeben. Diese Transition ist also nur möglich, wenn der vorige Zustand das Interface `IFinishable` implementiert.
- `SensorLevel`: `SensorLevel` ist die meistgenutzte Transition. Sie feuert immer dann, wenn ein vorgegebener Sensorwert über oder unterschritten wurde. Diese Transition kann Werte aller drei Ultraschallsensoren als Referenz nehmen und sowohl bei Über- oder Unterschreiten eines Grenzwertes aktiv werden.

Neben dem `BasicFollowWall`-State sind noch einige weitere States implementiert, die jeweils spezielle Aufgaben erfüllen und im Folgenden knapp vorgestellt werden:

- `ApproachPoint`: Es kann ein Punkt übergeben werden, der dann geregelt angefahren wird. Dieser Zustand wurde von uns dafür erstellt, einen Punkt einen Meter vor einem *ArUco*-Tor anzufahren.
- `ArucoGateCenter`: Dieser Zustand richtet das Auto aus, wenn es vor einem *ArUco*-Tor steht, um das Tor anschließend gerade zu durchfahren. Dies ist nötig, da

das Auto nach Transition aus dem `ApproachPoint`-Zustand nicht rechtwinklig zur Verbindungsgeraden der beiden Marker-Mittelpunkte ausgerichtet ist.

- `FollowWall`: `FollowWall` ist eine Modifikation des `BasicFollowWall`-Zustandes. Ziel war, dass die Regelung Türrahmen ignoriert, sodass sich das Auto anschließend nicht stark aufschaukelt. Dies wurden erreicht, indem Abweichungen der Ultraschallsensorwerte, die größer als 2cm sind, ignoriert werden. Dies bringt den Vorteil mit sich, dass das Auto sehr stabil gerade aus fährt. Nachteilig ist jedoch, dass es nicht mehr oder nur noch sehr schwach auf größere Änderungen der Umgebung reagiert. Zur Erkennung von Kurven ist dies jedoch kein Problem, da diese ja in der folgenden Transition getriggert werden.
- `FollowWallRamp`: Dieser Zustand ermöglicht eine Abstandsregelung zu einer das Auto umgebenden Wand nach einer linearen Funktion. Er ermöglicht somit z.B. einen Spurwechsel oder das Umfahren von Hindernissen.
- `FSMState`: Dieser Zustand ermöglicht das Einbinden von Zustandsautomaten als eigenen Zustand in einer FSM. Dies schafft eine sehr starke Abstraktion, sodass eine Zustandsfolge, z.B. zur Kurvenfahrt, abstrakt eingebunden werden kann und die Größe des Gesamtautomaten überschaubar bleibt.
- `Idle`: Dieser Zustand ändert nichts am zustand des Autos.
- `Motor`: Hier fährt das Auto einfach mit festgelegter Geschwindigkeit eine festgelegte Distanz geradeaus.
- `Stop`: Dieser Zustand stoppt das Auto.

Mittels der vorgestellten Konstrukte kann nun eine beliebige Abfolge von Aktivitäten des Autos abgebildet werden. Es ist möglich, ihm eine Strecke oder alternativ ein Verhalten einzuprogammieren. Ein autonomes Verhalten wird z.B. über mehrere parallele Zweige im Automaten realisiert. Als Beispiel fährt das Auto geradeaus und falls ein Hindernis auftritt, feuert eine Transition, die mehrere Zustände zu dessen Umfahren triggert. Werden Marker erkannt, könnte eine andere Transition feuern und je nach ID ein Verhalten des Autos auslösen.
Durch die vorgestellte Modularität ist das autonome Verhalten des Autos um fast beliebige Szenarien erweiterbar.

7.2 Erkennung *ArUco*-Marker

Die Erkennung der *ArUco*-Marker ist durch einen ROS-Node implementiert. Dort haben wir zur Kamerakalibrierung eine Kombination der ROS-internen Kalibrierung und der Kalibrierung der **OpenCV**[1] Open Source Software Bibliothek verwendet. Zur Kalibrierung wird ein Schachbrettmuster verwendet. Dabei übergibt man der OpenCV-Routine die Anzahl der Kästchen und deren Größe. Nachdem die Eckpunkte erkannt sind, werden verschiedene Orientierungen im Raum dargestellt, um eine robuste Kalibrierung zu

[1] http://opencv.org/

erreichen. Anhand der Zuordnungen von Raum zu Bildkoordinaten kann nun die Kalibrierungsmatrix berechnet werden. *OpenCV* erkennt nun die *ArUco*-Marker und kann deren Position errechnen. Die Daten dazu werden anschließend in einer *ROS*-Topic gepublisht. Unserer Anwendungssoftware stehen so die genauen Bildkoordinaten der Marker und ihre Ausrichtung im Raum zur Verfügung.

7.3 Durchfahren von Toren

Nachdem nun dargestellt wurde, wie *ArUco*-Marker erkannt werden, bedarf es nur noch eines weiteren Schrittes, um das erkannte *ArUco*-Tor zu durchfahren. Man muss anhand der erhaltenen Koordinaten den weiteren Streckenverlauf so umplanen, dass das Tor durchfahren wird.
Dazu gibt es verschiedene mögliche Ansätze. Zuerst war geplant, eine S-Kurve zu berechnen, deren Startpunkt die aktuelle Position und deren Endpunkt ein Punkt unmittelbar vor dem *ArUco*-Tor ist. Der Vorteil dieser Implementierung ist, dass das Auto vor dem Tor schon senkrecht zur Verbindungsstrecke der beiden Markermittelpunkte ausgerichtet ist, sodass es dann nur noch geradeaus fahren muss. Dazu war es nötig, sowohl Kenntnis über die Strecke, welche durch eine Hallsensoreinheit beschrieben wird, sowie über die Lenkwinkel, passend zu den eingestellten Lenkeinschlägen, zu erhalten. Dazu haben wir mehrere Messstrecken durchgeführt. Leider lieferte der Hall Sensor teils unzuverlässige Werte. Für die Bestimmung der Lenkwinkel haben wir eine ganze Messreihe aufgenommen. Daraus konnten nun zwei Viertelkreisbögen errechnet werden, die umgekehrt zusammengesetzt eine S-Kurve ergeben. Dabei haben wir festgestellt, dass das Auto bei Lenkeinschlag 0 nicht exakt geradeaus fährt, weshalb ein Offset abgezogen werden musste. Zudem erzielten wir nicht reproduzierbare Ergebnisse. Manchmal fuhr das Auto perfekt durch das erkannte Tor, manchmal lag es weit daneben. Dieser Umstand ist nach unseren Auswertungen dem schlechten Gewichts-Leistungsverhältnis geschuldet. Die Servomotoren schaffen es nicht, gerade aus dem Stand, in den gewünschten Lenkeinschlag zu lenken. Dies führt dazu, dass das Ergebnis von zwei Eingangsgrößen, dem aktuellen Lenkeinschlag und der Geschwindigkeit, abhängt, sodass es uns nicht möglich war diesen Ansatz weiter zu verfolgen.

Daher haben wir uns entschlossen, wie oben bereits erwähnt, eine lineare Regelung zu implementieren. Dies bedingt jedoch (im Gegenteil zur Lösung mittels S-Kurve) eine Wand auf einer Seite des Fahrzeuges, zu der es sich relativ bewegen kann.

Der Regler um eine lineare Funktion erhält als Eingangsgrößen den Startabstand, den Endabstand, sowie die Strecke in Hallsensoreinheiten. Anhand dieser Werte wird, wie auch im normalen `BasicFollowWall`-State, eine Regelung durchgeführt. Unterschied ist diesmal jedoch, dass sich der Abstand zur Wand, der sonst ja konstant ist, linear ändert. Das heißt, dass die Regelung mit zunehmender zurückgelegter Strecke um einen weiter von der Wand entfernten Abstand regelt.

Diese Funktion wurde im `FollowWallRamp`-State implementiert. In diesem State gibt es eine Funktion, `setTarget(Vec3f target)`, die den Zielpunkt der Regelung festlegt.

Dazu wird, je nach Wandseite, die x-Koordinate den übergebenen Parametern angepasst. Dabei wird die x-Koordinate des Ziels um den aktuellen Wandabstand erhöht bzw. erniedrigt, sodass sie den wirklichen Wandabstand enthält. Des Weiteren wird der y-Abstand des Zielpunktes für die messbaren Halldistanzwerte normalisiert.

```
void FollowWallRamp::setTarget(Vec3f target)
{
  _finished = false;
  if(_wallSide == WallSide::Right)
    _targetClearance = -(target[0])+_startClearance;
  else
    _targetClearance = target[0]+_startClearance;
  _distance = 1500/1.60f*target[2];
}
```

Abbildung 7.2: Anpassen des Wandabstandes, `FollowWallRamp.cpp`, Z.19

In der `tick()`-Methode sind nur folgende fünf Befehle relevant:

```
unsigned int curDistance = mstate.getHallDistance()-_startDistance;

if(curDistance >= _distance)
  _finished = true;

double m = (_targetClearance-_startClearance)/(_distance);
_clearance = m*curDistance + _startClearance;
...
BasicFollowWall::tick();
```

Abbildung 7.3: Reglungsparemeter aktualisieren, `FollowWallRamp.cpp`, Z.58

Hier wird die aktuell zurückgelegte Distanz aktualisiert und mit der Zieldistanz verglichen. Sollte diese erreicht sein, wird die nächste Transition gefeuert, indem der Wert `_finished`, der aufgrund der Implementierung des `IFinishable`-Interfaces vorhanden sein muss, gesetzt wird. Ansonsten wird die aktuelle Steigung berechnet, sowie mit ihr der aktuelle Wandabstand `_clearance`. Mit diesem wird am Ende der `tick()`-Methode des `FollowWallRamp`-Zustands die `tick()`-Methode des `BasicFollowWall`-Zustands aufgerufen. Dies sorgt, wie im Standardfall auch, für eine Regelung um den festgelegten Wandabstand.
Wie man sehen kann, ist für die Regelung nach einer linearen Funktion nur eine geringe Änderung des vorhandenen Codes nötig gewesen.

Diese neue Funktion des Autos wurde nun von uns verwendet, um ein durch die Kamera erkanntes *ArUco*-Tor, dessen Koordinaten im Raum übergeben wurden, auf einen Punkt

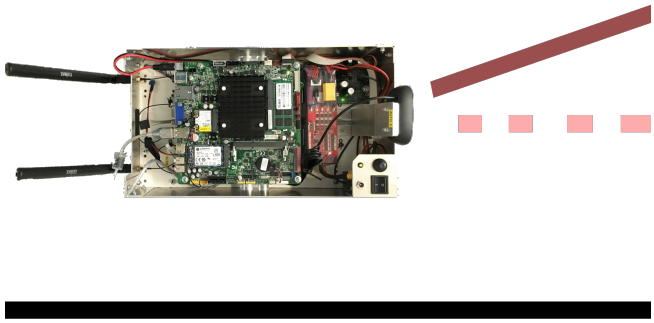

Abbildung 7.4: Schematische Darstellung des Fahrens einer Rampe

einen Meter vor dem Tor anzusteuern. Weiter oben wurde bereits der Umstand erwähnt, dass es mit einer S-Kurve möglich ist, eine gerade Ausrichtung des Autos zu erreichen. Dies ist mit der Rampenfunktion nicht möglich. Vielmehr ist die Ausrichtung fast nie gleich, da durch die Regelung unterschiedliche Lenkwinkel angenommen werden. In der Regel ist das Auto jedoch in Fahrtrichtung um einen geringen Winkel zwischen 0° und 45° zur Parallelen zur Wand durch den Tormittelpunkt versetzt. Daher muss die gefeuerte Transition nun noch in zwei weitere Zustände, einen zum Ausrichten des Autos (ArucoGateCenter) und einen zum geradeaus fahren (Motor), überführen. Ersterer macht nichts anderes, als zu prüfen, ob der Tormittelpunkt auf der linken oder rechten Seite des Autos ist und den Lenkeinschlag in entsprechender Richtung voll auszulenken und das Auto so weit zu bewegen, bis es ausgerichtet ist. Der Motor-State liest aus seiner *JSON*-Konfiguration den übergebenen Lenkeinschlag und die Geschwindigkeit und fährt mit dieser Konfiguration so lange, bis die folgende Transition feuert.

Wie man sehen kann, benötigen wir lediglich drei Zustände, um die Tordurchfahrt zu realisieren. Dank unserer FSM-Struktur ist es uns so einfach möglich, z.B. indem das Erkennen und Durchfahren eines *ArUco*-Tores parallel zu einem anderen Pfad geschaltet wird, flexibel auf auftauchende Tore zu reagieren und diese zu durchfahren.

8 Finite-state-machine

Wir betrachten nun etwas genauer, wie wir unser Konzept einer FSM zur Kontrollflusssteurung in *C++11* umgesetzt haben. Dazu gehört neben der Wahl der Klassenstruktur auch die Einbindung nützlicher Features von *C++11*, um die Speicherverwaltung zu optimieren.
Zur Modellierung einer FSM haben wir uns ein Konzept aus einer Kontroller Klasse `FSM` und zwei abstrakten Basisklassen `State` und `Transition` überlegt. Dabei stellen die abstrakten Basisklassen nur eine standardisiertes Interface bereit und erlauben somit eine schnelle Implementation von neuen, abgeleiteten Klassen.
Erst die abgeleiteten Klassen implementieren eine genaue Funktionalität, wie z.B. einer Wand zu folgen oder eine Transition nach einem bestimmten Ereignis auszulösen. Diese Klassen sind in den dafür vorgesehenen namespaces `TRAL::STATES` und `TRAL::TRANSITIONS` zu finden.

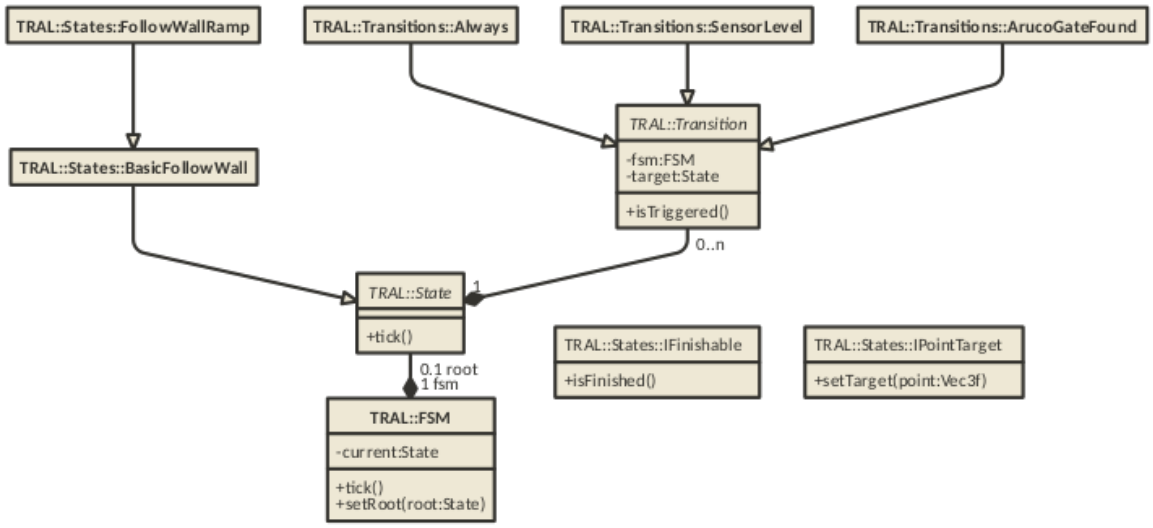

Abbildung 8.1: Vereinfachtes Klassendiagram der FSM

8.1 FSM

Die Klasse `FSM` implementiert die komplette Kontrollflusssteuerung und kümmert sich auch um das Laden einer FSM, die zuvor graphisch mit UMLet (siehe Abschnitt 10, graphische Programmierung) erstellt wurde. Ebenso hält diese Klasse immer eine aktuelle Referenz zu dem globalen `MachineState`. In dieser Klasse sind alle Sensorinformationen aufbereitet konsolidiert.

`FSM::tick`
Die wichtigste Funktion dieser Klasse ist die `tick` Funktion. Diese wird zyklisch vom ROS-Node tral-fsm aufgerufen. Dabei wird der Kontrollfluss an den aktuell aktiven State weitergeben. Wenn nun der aktive State eine neue Ausgabe gesetzt hat und die Kontrol-

le wieder abgibt, werden nun alle an diesem State befindlichen Transitionen daraufhin überprüft, ob sie ausgelöst haben. Sollte dies der Fall sein, wird eine `transit` vollzogen.

`FSM::transit`
Beim Statewechsel wird zuerst dem aktuell noch aktiven State signalisiert, dass er nun verlassen wird. Dabei kann der State zum Beispiel genutzte Ressourcen wieder freigeben. Darauf folgend wird dem neuen State signalisiert, dass dieser nun betreten wird und nötige Ressourcen belegen kann.

8.1.1 State

Die Klasse `TRAL::State` ist eine abstrakte Basisklasse, von der alle weiteren States abgeleitet werden. Für unsere Implementierung wurden zum Beispiel folgende States abgeleitet:

- ApproachPoint
- ArucoGateCenter
- BasisFollowWall
  - FollowWallRamp
- FollowWall
- Idle
- Motor
- Stop

Jeder instantiierbarer State muss alle virtuellen Funktionen der State-Klasse implementieren. Dadurch wird gewährleistet, dass die FSM-Klasse mit jeder beliegen State-Implementierung arbeiten kann.
Die statische Funktion `createFromJson` erlaubt, ein State-Instanz aus einem JSON-Objekt zu erzeugen. Als virtuelles Interface sind die Funktionen `tick`, `onEnter`, `onExit` und weitere Debug-Funktionen vorgesehen.
`onEnter` und `onExit` signalisieren das zuvor beschriebene Betreten und Verlassen eines States bei der Ausführung. Die Funktion `tick` wird für den aktiven State zyklisch ausgeführt und berechnet einen neue Ausgabe. Dabei kann über den globalen `MachineState` auf die aktuellen Sensorwerte zugegriffen und die Aktoren angesteuert werden.

8.2 Transition

Ähnlich der State-Klasse fungiert die `TRAL::Transition` als abstrakte Basisklasse für alle Transitionen. Ebenso existiert auch eine statische Funktion `createFromJson`, um gespeicherte Instanzen aus einem JSON-Objekt zu laden.

Jede Transition gehört zu einem eindeutigen Besitzer *owner* und zu einem eindeutigen Ziel *target*. Sollte der *owner*-State aktuell aktiv sein, wird jede Transition mit der Funktion `isTriggerd` auf Auslösung durch die FSM Instanz nach der Ausführung des States überprüft.
Für unsere Implementierung haben wir folge Transitionen abgeleitet:

- Always
- ArucoGateFound
- Distance
- Finished
- SensorLevel

8.3 Interfaces

Um die Übergabe von Informationen zwischen, zum Beispiel, einem State und einer Transtion zu gewährleisten, haben wir zwei Interfaces für States verwendet. States, die das Interface `IFinishable` einbinden, können entscheiden, wann sie Abgeschlossen sind. Zum Beispiel implementiert der State `ApproachPoint` dieses Interface und signalisiert die Ankunft an dem definierten Punkt.
Ebenso können States, die das Interface `ApproachPoint` implementieren, einen Zielpunkt gesetzt bekommen. Dieses Interface wird zum Beispiel auch vom State `ApproachPoint` implementiert und erlaubt so ein Verlegen des definierten Punktes bei der Ausführung.
So muss der *owner* State der `Finished` Transition das dazugehörige Interface implementieren. Sollte dies nicht der Fall sein, kommt es zu einem Laufzeitfehler während der Ausführung.

9 JSON

Da bei unserer Lösung die Logik zur Steuerung des Autos, also der Zustandsautomat, von der konkreten Implementierung getrennt wird, benötigten wir eine Möglichkeit, diesen Automaten zu erstellen, zu speichern und einzulesen. Unsere Wahl fiel auf die „**JavaScript Object Notation**“ (*JSON*), einem einfachen und kompakten Datenformat, welches alle benötigten Funktionalitäten mitbringt und für das es bereits einige Parser in verschiedenen Programmiersprachen gibt. Hauptsächlich war unsere Wahl allerdings durch die einfache Lesbarkeit des Codes begründet, da wir zu Beginn des Projektes die Automaten per Hand eintippten. Eine *JSON*-Datei besteht im Wesentlichen aus Objekten, welche in geschweiften Klammern gekapselt sind. Diese Objekte bestehen aus beliebig vielen Eigenschaften, die einen eindeutigen Schlüssel und einen zugeordneten Wert haben. Arrays von Objekten sind in eckigen Klammern eingeschlossen. Eine vereinfachte Ansicht eines Automaten könnte also so aussehen:

```
{"root": 0,
 "states":
    [{"id": 0,
     "type": "WandFolgen",
     "p": 12,
     "i": 0,
     "d": 30},
    {"id": 1,
     "type": "FollowWall",
     "p": 12,
     "i": 0,
     "d": 30}],
 "transitions": [ ... ]
}
```

Abbildung 9.1: Aufbau einer einfachen *JSON*-Datei

Um diese *JSON*-Datei zu deserialisieren, also konkrete Objekte unserer Zustände und Transitionen zu erstellen, benötigten wir einen Parser, der dazu in der Lage ist. Wir verwenden hierzu die *JSON*-Library „**JSON for modern C++**“ [1] von Niels Lohmann. Die komplette Implementierung dieser Library befindet sich in einer einzigen Datei, der `json.hpp`, die in unserer `FSM.cpp` inkludiert wird. Für unsere Zwecke verwenden wir den Iterator der Library, um mittels einer for-Schleife über alle Zustände und Transitionen iterieren zu können und die *parse*-Funktion, um aus gegebenen Strings ein *JSON*-Objekt erstellen zu können. Der genaue Vorgang des Einlesens wird im nächsten Abschnitt erläutert.

[1] https://github.com/nlohmann/json

10 Graphische Programmierung

Trotz der einfachen Lesbarkeit der *JSON*-Datei, verliert man bei immer größer werdenden Automaten leicht den Überblick und der Vorteil der Trennung von Logik und Programmierung geht verloren. Um dieses Problem zu beheben setzen wir das Programm ***UMLet*** [1] ein. Es ist ein freies Open-Source UML-Tool, welches mehrere Diagramm-Arten unterstützt, unter anderem State-Charts. Gespeichert werden diese Diagramme im xml-Format, welches wir mit dem Programm „**xml2json**“ [2] in eine *JSON*-Datei, für die weitere Verwendung, umwandeln.

10.1 UMLet - Das Programm

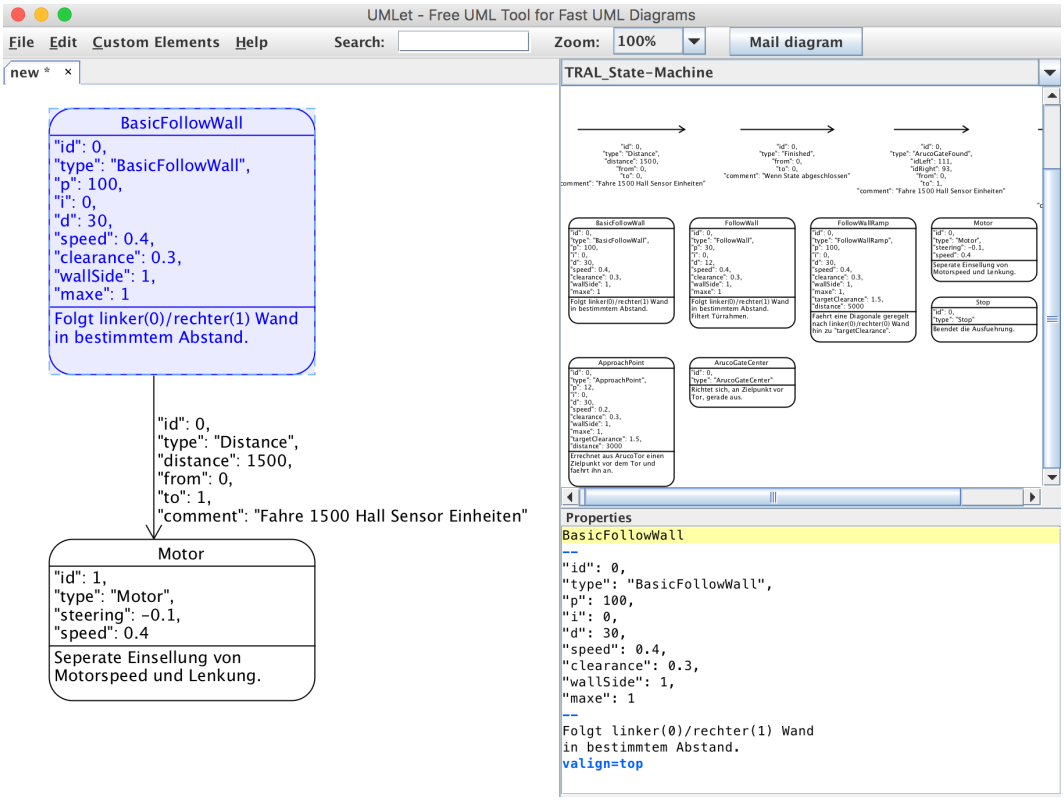

Abbildung 10.1: *UMLet* Oberfläche

Das Programm *UMLet* besteht aus drei Teilbereichen. Im linken Teil, dem Arbeitsbereich, wird der eigentliche Automat erstellt. Hierfür können schon vordefinierte Zustände und Transitionen aus der Vorlage *TRAL_State-Machine*, oben rechts im Fenster, in den Arbeitsbereich gezogen und entsprechend den eigenen Wünschen miteinander verbunden werden. Die Vorlagen-Datei *TRAL_State-Machine.uxf* muss hierfür innerhalb des Programmordners unter *../palettes* hinterlegt sein. Sie kann natürlich ebenfalls von dort geöffnet und gegebenenfalls erweitert werden. Der wichtigste Bereich ist das Fens-

1 http://www.umlet.com
2 https://github.com/Cheedoong/xml2json

ter für die Eigenschaften der Objekte unten rechts in der Ecke. Bei Zuständen werden zwischen den beiden horizontalen Trennlinien, bei Transitionen direkt unter dem stilisierten Pfeil *lt*=->, alle Parameter angegeben, die zur späteren Erstellungen der Objekte benötigt werden. Diese Liste muss stets vollständig sein. Sie ist durch unsere konkrete Implementierung im Quellcode vorgegeben. Wichtig hierbei ist, den Zuständen und Transitionen eindeutige *ids* zu vergeben, da diese grundlegend für den logischen Fluss sind.

10.2 uxf-Datei in *JSON*-Format umwandeln

Die uxf-Dateien, welche von *UMLet* erstellt werden, sind intern im xml-Format gespeichert. Sofern man das Tool *xml2json* bereits heruntergeladen und kompiliert hat, lassen sich diese Dateien mittels des einfachen Konsolen-Befehls

```
./xml2json automat.uxf >> automat.json
```

in eine *JSON*-Datei umwandeln. Da man mittels des Aufrufs

```
rosrun tral_fsm tral_fsm_node -g automat.json
```

unseren FSM-Node starten kann und ihm diese *JSON*-Datei übergeben muss, lässt sich dieser ganze Abschnitt leicht mit einem Shell-Skript zum Umwandeln und Starten der FSM realisieren.

10.3 Einlesen einer JSON-Datei

Unsere Klasse `FSM` besitzt eine Methode namens `loadFile`, welche den Automaten im *JSON*-Format einlesen kann. Die Implementierung muss natürlich an den Aufbau der Datei durch das graphische Tool angepasst sein. Die resultierende *JSON*-Datei besteht aus zwei geschachtelten Objekten mit verschieden Angaben zum Programm und zur Programmversion. Im inneren Objekt ist eine Eigenschaft mit dem Schlüssel `element` angelegt. Diese Eigenschaft enthält ein Array aller Zustände und Transitionen. Mittels der Anweisung `j["diagram"]["element"]` greift man darauf zu und kann mittels des Iterators `auto jelem` darüber iterieren.

```
for(auto jelem: j["diagram"]["element"])
```

Abbildung 10.2: `FSM.cpp`, Zeile 32

Jedes Element besteht aus vier Eigenschaften: `id`, `coordinates`, `panel_attributes` und `additional_attributes`. Relevant ist die `id`, um zwischen Zuständen und Transition zu unterscheiden und die *panel_attributes*. Hier ist ein String mit der Parameterliste gespeichert.
Es muss insgesamt zwei Mal über alle Elemente iteriert werden. Beim ersten Durchgang werden alle Zustände erstellt, beim Zweiten alle Transitionen. Dies ist deshalb notwendig, weil alle Transitionen einen Verweis auf ihren Vor- und Nachfolgezustand enthalten.

Um diesen Verweis erstellen zu können, muss das entsprechende Zustandsobjekt bereits existieren.
Der String der Parameterliste enthält noch Steuerzeichen und es wird, je nachdem, ob Zustände oder Transitionen erstellt werden, die Funktion `manipulateString` entsprechend aufgerufen, um diese zu entfernen. Der verbleibende String ist jetzt ebenfalls nach dem Schema eines *JSON*-Objektes aufgebaut und enthält Eigenschaften mit Schlüsselwort und zugeordnetem Wert. Daraus kann nun ein *JSON*-Objekt erstellt werden:

```
json jstate = json::parse(jstr);
```

Abbildung 10.3: *JSON*-Objekt erstellen, `FSM.cpp`, Z.37

Für ein solches *JSON*-Objekt hatten wir bereits in den Zustands- und Transitionsklassen Methoden implementiert, die daraus konkrete Objekte für unsere FSM erstellen können (`Transition.cpp` / `State.cpp`). Hier kann, an Hand der Information die unter dem Schlüssel `type` im *JSON*-Objekt gespeichert ist, entschieden werden, welche Art von Zustand/Transition vorliegt und der entsprechende Konstruktor aufgerufen werden. Im Falle eines Zustandes ist das Prozedere des Einlesens beendet und das Zustandsobjekt wird in einem Array am Index seiner `id` gespeichert.

```
states[(int)jstate["id"]] = state;
```

Abbildung 10.4: Array mit Zuständen, `FSM.cpp`, Z.39

Bevor Transitionen in ihrem Array gespeichert werden können, müssen zuvor noch Vorgänger und Nachfolgezustand gesetzt werden. Ebenso wird dem Vorgängerzustand die von ihm abgehende Transition zugeordnet.

```
trans->setOwner(states[from]);
trans->setTarget(states[to]);
states[from]->addTransition(trans);
```

Abbildung 10.5: Behandlung von Transitionen, `FSM.cpp`, Z.53-56

11 Fazit

11.1 Das Fahrzeug

Die durch das Projektseminar Echtzeitsysteme gewonnenen Erfahrungen sind sehr vielschichtig. Wie schon in der Einleitung zu vermuten war, galt es, technisch gesehen, Hürden aus den unterschiedlichsten Bereichen zu überwinden. Von Verständnis komplexer Software wie *ROS* und Einsicht in das eingesetzte Linux Echtzeitbetriebssystem zu erlangen, bis zur Änderung der Software auf dem *Fujitsu*-Board. Auf der anderen Seite vom Interpretieren und Verwenden der Sensorwerte bis hin zum Ansteuern der Motoren und ermitteln der Regler-Werte.

Im Laufe der Bearbeitungszeit stellten wir schnell fest, dass vor allem in der Hardware die Schwierigkeiten lagen. Dabei war weniger die Prozessorleistung oder Rechengeschwindigkeit das Problem, diese reichte ohne Probleme auch für das Verarbeiten des Videostroms, als vielmehr die Sensorik, Aktorik und der Aufbau des Fahrzeugs. Selbst mit eigens entwickelter Filterung waren die Ultraschall Werte nicht immer zuverlässig, vor allem, wenn der Winkel zum reflektierenden Objekt ungünstig war, im Besonderen an Ecken und Kanten. Da diese in den Fluren des Instituts jedoch oft ähnlich aufgebaut waren, ließ sich auch hier mit Software ganz gut Abhilfe schaffen.

Deutlich problematischer erwies sich der Hall-Sensor. Da er ungünstig am Fahrzeug positioniert ist, reicht es schon beispielsweise, beim Einschalten den Aufbau versehentlich leicht gegenüber dem Fahrgestell zu verschieben, sodass der Abstand zum im Rad montierten Magneten minimal vergrößert wurde und eine Messung somit unmöglich. Auch traten nicht immer erklärbare Sprünge in den vom Hall-Sensor gelieferten Werten auf, was verständlicherweise ein präzises Fahren sehr erschwerte.

Als recht umständlich erwies sich auch die Lenkung. Nicht der große Wendekreis und das langsame Einlenken, unter anderem verursacht durch den schweren Aufbau, stellten hier die größte Schwierigkeit dar. Sondern vor allem die Nichtlinearität, die zudem in jede Richtung unterschiedlich war. Einen Pfad zu berechnen und das Fahrzeug entsprechend zu steuern stellte sich, wie in Kapitel 7.3 bereits beschrieben, als stellenweise sehr unpräzise bzw. unmöglich heraus. Wir haben daher wieder auf eine Regelung, unterstützt von einer geraden Wand zurückgegriffen, was für unsere Zwecke sehr gut funktionierte, das allgemeine Problem mit der Lenkung jedoch nur umgeht. Die Kamera funktionierte nach korrektem Kalibrieren und Ansprechen mit *ROS* recht zuverlässig. Hier haben wir lediglich den Aufbau etwas modifiziert und den Neigungswinkel der Kamera leicht aufgerichtet, sodass wir vor dem Fahrzeug eine größere Fläche einsehen konnten.

11.2 Projektkoordination und Aufgabenstellung

Die Projektkoordination war ebenfalls ein wichtiger Aspekt, den wir bedenken mussten. Unser Vorgehen basierte hier auf Erfahrungswerten vorhergehender Gruppenarbeiten.

In diesem Projektseminar ergab sich jedoch die Möglichkeit, diese auch über ein ganzes Semester hinweg anzuwenden und zu optimieren, wodurch alle Beteiligten ihren Erfahrungsschatz erweitern konnten. Das Gleiche gilt natürlich auch für das Erstellen und Vortragen von Präsentationen. Dazu hatten wir in dieser Veranstaltung gleich zweimal die Möglichkeit, was natürlich auch sinnvoll ist, da auch die Aufgabenstellung in zwei Blöcke geteilt werden kann. Dabei wurde unser Vorgehen, wie bereits erwähnt, in den zweiwöchentlichen Treffen unterstützt, in denen die wichtigsten Fortschritte und Ziele besprochen wurden. Zu Beginn waren dabei natürlich vornehmlich die Grundlagen und Pflichtimplementierungen oder der Zwischenvortrag Thema. Gegen Ende ging es mehr um die eingeschlagene Richtung und das Thema der vom Team gewählten Vertiefung. Diese musste zwar abgesprochen werden, war jedoch ansonsten komplett freigestellt. Hier war positiv, dass dies nicht nur als Zusatzaufgabe gedacht war. Im Gegenteil stellte die Pflichtimplementierungen eher nur die Basis dar und ermöglichte so eine Einarbeitung in die Thematik. So stand für das selbst gewählte Thema ausreichend Zeit zur Verfügung, um auch neue oder alternative Ideen zu verfolgen.
Generell ist hier anzumerken, dass die fast komplett freie Aufgabenstellung es ermöglichte, dass jedes Team seine eigenen Interessen im Kontext autonomes Fahren verwirklichen konnte. Dies hat auch den Vorteil, am Ende des Semesters nicht fünf gleiche Projekte zu sehen, sondern vielfältige Lösungen ganz unterschiedlicher Probleme.

11.3 Unsere Lösung und Ausblick

Wie bereits in der Einleitung angekündigt und später wieder aufgegriffen, stellt die Nachhaltigkeit und deren praktische Umsetzung das Kernthema unserer Lösung dar. Dies wurde uns schon von Beginn an mit auf den Weg gegeben. Die Wichtigkeit dieses Themas kristallisierte sich jedoch immer mehr heraus. Vor allem bei den bereits angesprochenen Problemen mit der Sensorik und Aktorik darf nicht in jedem Semester von neuem bei null begonnen werden müssen. Dies kostet viel Zeit und bringt einen deutlich geringeren Lerneffekt, als umfangreiche Projekte umzusetzen. Hier ist es sehr wichtig einzelne Aspekte, wie zum Beispiel die Steuerung oder die Parameter für die Regelung, korrekt und zuverlässig zur Verfügung zu stellen. Unsere Lösung erlaubt dann, das Verknüpfen solcher einzelnen Einheiten zu einer abstrahierten, sehr modularen Software. Nur so kann auf Dauer eine Entwicklung stattfinden, bei der am Ende auch ein sowohl stabiles als auch nachvollziehbares Ergebnis steht.

Gerade im Hinblick auf den *Carolo Cup*, der als Fernziel des Projektseminars genannt wurde, erscheint es uns sehr wichtig, eine Plattform zu entwickeln, auf der die einzelnen Teams aufbauen und diese um unterschiedliche Funktionalitäten erweitern können. Dazu bietet unser Konzept der FSM eine sehr einfache, gute und effiziente Lösung, da hier unabhängig verschiedene Funktionalität von verschiedenen Teams entwickelt und einfach verteilt werden kann.