Projektseminar Echtzeitsysteme

Ausarbeitung von Team TRAL
Proseminar eingereicht von
Tim Burkert, Robert Königstein, Lars Stein, Adrian Weber
am 31. März 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 Kulcsar

ES-B-0060

Erklärung zum Proseminar

Hiermit versichere ich, das vorliegende Proseminar selbstständig und ohne Hilfe Dritter angefertigt zu haben. Gedanken und Zitate, die ich aus fremden Quellen direkt oder indirekt übernommen habe, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen und wurde bisher nicht veröffentlicht.

Ich erkläre mich damit einverstanden, dass die Arbeit auch durch das Fachgebiet Echtzeitsysteme der Öffentlichkeit zugänglich gemacht werden kann.

Darmstadt, den 31. März 2016

_____

(Adrian Weber, Tim Burkert, Lars Stein, Robert Königstein)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Einleitung

1.1 Ne Ebene tiefer

1.1.1 Undsoweiter

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 16Bit Mikrocontroller der MB96300-Serie von *Fujitsu Microelectronics* (heute: *Cypress Semiconductor Corporation*). Die zweite Schicht der Hardware des Autos besteht aus einem vollwertigen Einplatinencomputer mit Quadcore-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 Hall-Sensoren an den Rädern, ein Liniensenor, eine USB-Webcam, sowie drei Ultraschallsensoren (rechts, linke und vorne) zur Verfügung. Zur Ansteuerung der Lenkung des Modellautos steht ein Servo zur Verfügung. Der Antrieb des Autos ist durch Gleichstrommotoren, die durch einen Fahrtregler fast stufenlos angesteuert werden können, 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.

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 gestellt, was zu ungenauen Sensorwerten führte. Die abbildbare Tiefe betrug etwa 20cm bis 3m bei 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. Dazu war es notwendig, die dafür zuständige Variable von 0x51 auf 0x52 zu inkrementieren. Dies hat die Genauigkeit, sowie den Messbereich auf ca. 5m erhöht.

3 Grundlegendes zu *ROS* und Lubuntu

Im Folgenden wird eine kurze Einleitung zu ROS 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 Hauptmotivationen, *ROS* einzusetzen sind Modularität, Portabilität und Wiederverwendbarkeit, sowie eine vereinfachte Softwareentwicklung (einfache Interfaces, Debugging, Monitoring, Testen).

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 viele-zu-viele-Nachrichtenaustausch ermöglichen, sind SServices"dazu da, einen direkten Nachrichtenaustausch zwischen zwei *Nodes* zu ermöglichen. Ein *Service* besteht dabei aus einem Nachrichtenpaar aus einer Anfrage und einer Antwort auf diese.

3.2 Betriebssystem

In dem vorgegebenen Software Framework war als Grundlage des Projektseminars neben *ROS* das Betriebssystem *Lubuntu* (auch in Echtzeitversion) vorgegeben. *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 die Anwendungen 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) 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.

4 Arbeitsumgebung

Hallo?

4.1 Noch grundlegender

Text

4.1.1 Grundlage?

Test

4.1.2 GRUNDLAGE!

Text

5 Projektkoordination

Hallo?

5.1 Noch grundlegender

Text

5.1.1 Grundlage?

Test

5.1.2 GRUNDLAGE!

Text

6 Aufgabenstellung

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

6.1 Pflichtimplementierung

Als Pflichtteil des Projektseminars wurde generell das Thema „Autonomes Fahren mittels Sensorik“, „Kamera Inbetriebnahme 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 des 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 das Auto seinen Abstand mittels Ultraschallsensorik absolut bestimmen kann

6.1.1 Vertiefungspakete

Als optionale Vertiefungspakete wurden „*ROS*-basierte Simulation“, „Fernsteuerung und Car-2-Car 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 auf einem anderen Kernel lief, wie 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 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 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 entwickelt.

7 Lösung Pflichtimplementierung

Hallo?

7.1 Noch grundlegender

Text

7.1.1 Grundlage?

Test

7.1.2 GRUNDLAGE!

Text

8 FSM

Wir betrachten nun etwas genauer wir unsere 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.

8.1 Klassenstruktur

Zur Modellierung einer FSM haben wir 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 einen schnelle Implementation von neuen abgeleiteten Klassen.

Erst die abgeleiteten Klassen implementieren eine genau Funktionalität, wie z.B. einer Wandfolgen oder eine Transition nach einem bestimmten Ereignis. Diese Klassen sind in dem dafür vorgesehen namespace `TRAL::STATES` und `TRAL::TRANSITIONS` zu finden.

8.1.1 FSM

Die Klasse `FSM` implementiert die komplette Kontrollflusssteuerung und kümmert sich ebenso um das Laden einer FSM die zuvor graphisch mit Umlet erstellt wurde. Ebenso hält diese Klasse immer eine aktuelle Referenz zum dem globalen `MachineState`, in dieser Klasse sind alle Sensorinformationen aufbereitet konsolidiert.

`FSM::tick`

Die wichtigste Funktion diese Klasse ist die `tick` Funktion, diese wird zyklisch von der Rosnode tral-fsm aufgerufen. Dabei wird der Kontrollfluss an den aktuell aktiven State weitergeben. Wenn nun der aktive State einen neuen Ausgabe gesetzt hat und die Kontrolle wieder abgibt werden nun alle an diesen State befindlichen Transition überprüft ob diese Ausgelöst haben, sollte dies der Fall sein wird eine `transit` vollzogen.

`FSM::transit`

Beim Statewechsel wird zuerst dem aktuell noch aktiven State signalisiert das der nun verlassen wird, dabei kann der State zum Beispiel genutzte Ressourcen wieder freigeben. Darauffolgend wird dem neuen State signalisiert das dieser nun betreten wird und nötige Ressourcen belegen kann.

8.1.2 State

Die Klasse `TRAL::State` ist eine abstrakte Basisklasse von dieser werden alle States abgeleitet, für unsere implementation wurden zum Beispiel folgende States abgeleitet:

- ApproachPoint
- ArucoGateCenter
- BasisFollowWall

  - FollowWall
  - FollowWallRamp

- Idle
- Motor
- Stop

Jeder instantiierbarer State muss alle virtuellen Funktion der State Klasse implementieren. Dadurch wird gewährleistet das die FSM Klasse mit jede beliegen State Implementation arbeiten kann.

Die statische Funktion `createFromJson` erlaubt ein State Instanz aus einem json Objekt erzeugen, dazu mehr unter 8.2. Als virtuelles Interface sind die Funktion `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 neuen Ausgabevektor.

8.1.3 Transition

Text

8.2 Deserialisieren

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 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

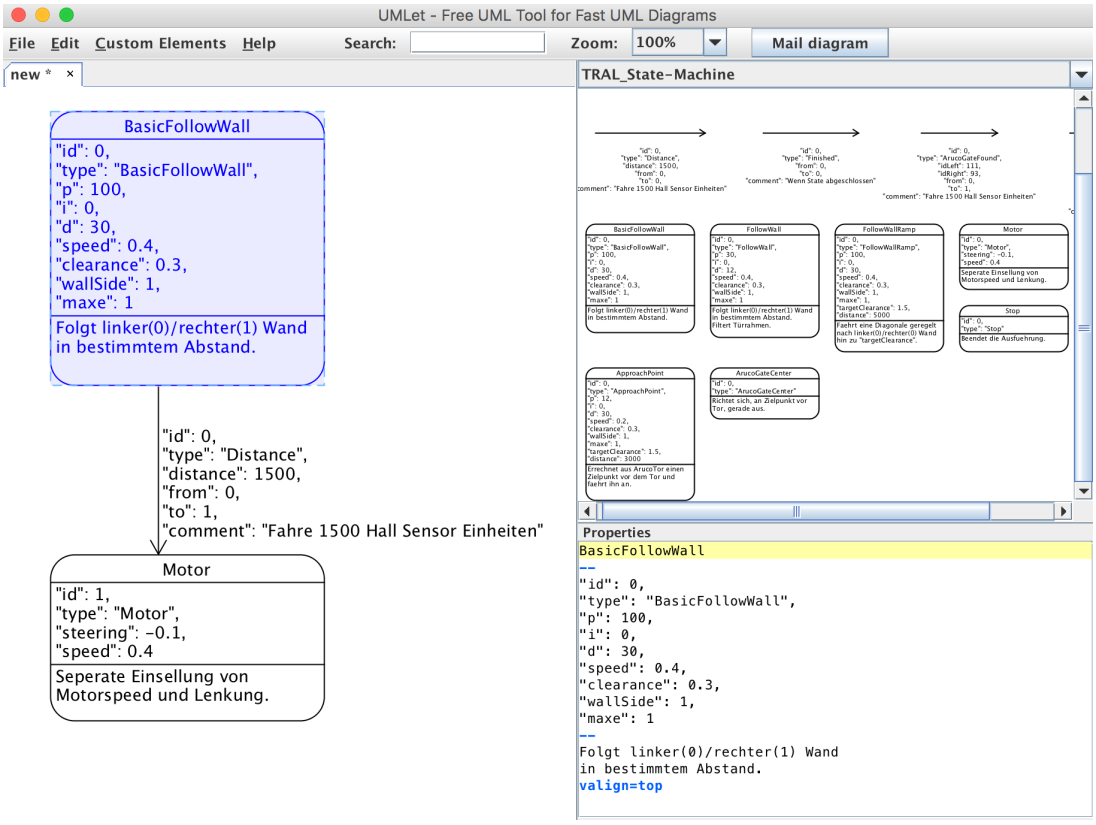

Abbildung 10.1: UMLet Oberfläche

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 in den Arbeitsbereich gezogen und entsprechend den eigenen Wünschen miteinander verbunden werden. Die Vorlagen-Datei „TRAL_State-Machine.uxf“ muss hierfür im innerhalb des Programmordners unter „*../palettes*“ hinterlegt sein. Sie kann natürlich ebenfalls von dort geöffnet und gegebenenfalls erweitert werden. Der wichtigste Bereich ist das

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

Fenster 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 und lässt sich in unserer Dokumentation am Ende dieses Dokumentes nachlesen. 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 „xyz“ 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. (skript?!)

10.3 Einlesen einer JSON-Datei

Unsere Klasse „**FSM**“ besitzt ein 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"][ëlement"]"*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 ist.

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 für die Zustands- und Transitionsklasse Methoden implementiert, die daraus konkrete Objekt für unsere FSM erstellen können (**Transition.cpp** / **State.cpp**). Hier kann, anhand der Information die unter dem Schlüssel „*type*“ im JSON-Objekt gespeichert ist, entschieden werden welche Art von Zustand/Transitions 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 Doku

Hallo?

11.1 Noch grundlegender

Text

11.1.1 Grundlage?

Test

11.1.2 GRUNDLAGE!

Text

12 Fazit

Hallo?

12.1 Noch grundlegender

Text

12.1.1 Grundlage?

Test

12.1.2 GRUNDLAGE!

Text

A Erster Anhang