

Computer Systems and Telematics — Distributed, Embedded Systems

Bachelor Thesis

Design und Implementierung einer Analysesoftware im Kontext eines Referenzsystems zur Indoorlokalisierung

Benjamin Aschenbrenner

Matr. 4292264

Betreuer: Prof. Dr-Ing. Jochen Schiller
Betreuender Assistent: Dipl.Inf. Heiko Will

Institut für Informatik, Freie Universität Berlin, Deutschland

3. Dezember 2011

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, sind als solche gekennzeichnet. Die Zeichnungen oder Abbildungen sind von mir selbst erstellt worden oder mit entsprechenden Quellennachweisen versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner Prüfungsbehörde eingereicht worden.

Berlin, den 3. Dezember 2011

(Benjamin Aschenbrenner)

Zusammenfassung

Zusammenfassung

Diese Arbeit steht im Kontext zum Aufbau eines Referenzsystems, welches in der Lage ist sich mobil indoor zu bewegen und dabei möglichst genau zu lokalisieren. Der Zweck dieses Referenzsystems ist es, Lokalisierungen als Referenz zur Verfügung zu stellen, um die Genauigkeit von Lokalisierungen mobiler Sensorknoten, die in einem *Wireless Sensor Networks* Wireless Sensor Network (WSN) organisiert sind, zu ermitteln. Das Referenzsystem ist durch einen Roboter realisiert, welcher autonom vorgegebene Wegpunkte in einer Karte abfährt. Bei einer solchen Fahrt zeichnet der Roboter sowie an ihm befestigte Sensorknoten einen Pfad durch regelmäßige Lokalisierung auf. In dieser Arbeit geht es um die Implementierung eines Analysewerkzeugs namens *Pathcompare*, welches ermöglicht, die Pfaddaten zusammenzuführen, für den Tester aufzuwerten und zu visualisieren. Neben mittleren Abstand (Median) zum gewählten Referenzpfad werden Parameter wie Pfadlänge, Anzahl der Pfadpunkte, Stichprobenvarianz und eine Liste der größten Abweichungen angezeigt. Alle Daten können als Comma Separated Values (CSV) exportiert werden. *Pathcompare* ist in das ROS integriert und so entwickelt dass es über Plug-ins erweitert und angepasst werden kann.

Abstract

This thesis is associated with the development of a reference system that has the ability to localize itself. The reason for the development of such a system is to provide localization data. This data is used as reference data for localization data of mobile sensor nodes organized in a *Wireless Sensor Network* WSN in order to evaluate the precision of their localization measurements. Such a reference system was built in the form of a mobile robot that is able to autonomously navigate to given waypoints. While moving, the robot and sensor nodes mounted on it, generate path data by continuously localizing. This work is about the creation of an analysing tool named *Pathcompare* that is used to merge the path data of different sources and visualize them for a tester. The software shows the median distance of a path to the given reference path, the overall pathlength, total number of points per path and also a list of the greatest distances to the reference path. All results can be exported as CSV. *Pathcompare* is integrated into the ROS and can be extended via a Plug-in mechanism.

Inhaltsverzeichnis

Abbildungsverzeichnis	ix
Tabellenverzeichnis	xi
Quellcodeverzeichnis	xiii
Glossar	xv
Akronyme	xvii
1 Einleitung	1
1.1 Motivation	1
1.2 Aufgabenstellung	2
1.2.1 Der Roboter	2
2 Pathcompare - Implementierung	5
2.1 Technischer Rahmen	5
2.1.1 Robot Operating System - ROS	5
2.1.2 Qt	8
2.2 Design der Software	9
2.2.1 Überblick Gesamtsystem	9
2.2.2 Plug-in Main Compare	10
2.2.3 Plug-in Konzept allgemein	10
2.3 Anwendung	10

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

2.1	ROS transformation message	6
-----	--------------------------------------	---

Glossar

Robot Operating System Ein Framework welches zahlreiche Pakete bzgl. Nachrichtenaustausch zwischen verteilten Programmen, Hardwareabstraktion und Robotik bietet. v, 3, 5–8, 10

Akronyme

API Application Programming Interface. 5

CSV Comma Separated Values. v

IMU Inertial Measurement Unit. 2

MEMS Microelectromechanical systems. 2

MOC Meta Object Compiler. 8, 9

UIC User Interface Compiler. 9

WSN Wireless Sensor Network. v

XML-RPC XML - Remote Procedure Call. 6

KAPITEL 1

Einleitung

1.1 Motivation

Systeme zur Positionsbestimmung werden für zahlreiche Zwecke genutzt und deren Bedeutung wächst parallel zur Verbreitung immer neuer sogenannter *location based services* und deren wachsender Nutzung. Für Anwendungen im Freien haben sich Satelliten gestützte Systeme, welche hohe Genauigkeit bieten, etabliert. Als bekanntes Beispiel sei hier das *NAVSTAR-GPS* genannt, welches sich auch im zivil nutzen lässt. Allerdings ergeben sich viele Anwendungsumgebungen, in denen derartige Systeme gar nicht, bzw. nur ungenau funktionieren oder bewusst z.B. aus Kostengründen gemieden werden. Dies sind typischerweise Umgebungen in denen die Satellitensignale zu stark gedämpft werden oder vor allem durch Reflexionen bedingte Laufzeitverschiebungen, sich negativ auf die Genauigkeit auswirken, wie z.B.:

- innerhalb von Gebäuden (“indoor”)
- im Untergrund (Tunnel, Höhlen u.ä.)
- im Bereich dicht bebauter urbaner Gebiete (Mehrwegeausbreitung)

Um in solchen Umgebungen dennoch Lokalisierung zu ermöglichen wurden und werden viele theoretische Konzepte und konkrete Systeme entwickelt. Einen Überblick hierzu bietet [Quelle anbringen \(mobile entity localization and tracking in GPS less environments - Buch\)](#) [Quelle anbringen \(mobile entity localization and tracking in GPS less environments - Buch\)](#) Auch in der Arbeitsgruppe *Computer Systems & Telematics*, an der *FU-Berlin*, wurde dem Problem der indoor Lokalisierung mit der Entwicklung eines *Wireless Sensor Network (WSN)* basiertem Systems im Rahmen des Forschungsprojektes *FeuerWhere*, begegnet. Dieses Projekt entstand u.a. in Kooperation mit der Berliner Feuerwehr. [ist das wichtig zu wissen an dieser Stelle?](#) Ziel bei der Entwicklung war ein flexibles indoor Lokalisierungssystem zu schaffen, welches mit low-cost Komponenten bzw. ohne Spezialhardware konstruiert wurde. Im Kern ist das System in der Lage die Entfernung zwischen involvierten Sensorknoten zu bestimmen und dadurch Rückschlüsse auf deren Position zu ermöglichen. In dem WSN unterscheidet man zwei Arten von Knoten, mobile Knoten und Anker Knoten. Diese unterscheiden sich nur dadurch, dass die Position eines Anker Knotens bekannt ist. Bei

einer hinreichenden Zahl von Anker Knoten im WSN kann dann per Trilateration bzw. Multilateration die Position eines mobilen Knotens ermittelt werden. **add figure principle of trilateration ?** Die Entfernungsmessung zwischen zwei Knoten geschieht hierbei durch Laufzeitmessungen von per Funk gesendeten Round Trip Time (RTT) Paketen, wodurch eine teure sowie aufwendige Zeit-Synchronisierung zwischen den Knoten entfällt, da bei der Messung der RTT nur ein Knoten die Zeit berechnet. Diese Laufzeitmessungen sind jedoch durch in der Hardware auftretenden Jitter und in *non-line of sight (NLOS)* Umgebungen auftretende Mehrwegeausbreitung fehlerbehaftet. Die genaue Funktionsweise und Untersuchung der Auftretenden Fehler ist beschrieben in. **hier würde ich natürlich gerne Heiko's paper reffen. Frage: Ist das schon erlaubt?** Um diesen Fehler zu untersuchen, ist es sehr nützlich, ein möglichst genaues aber ebenso flexibles Testsystem **Referenzsystem?** zur Verfügung zu haben, welches mögliche Anpassungen, Konfigurationen und Einsatzszenarien des indoor Lokalisierungssystems, in Hinblick auf dessen Genauigkeit, evaluierbar macht. Der Implementierung und Analyse eines solchen Referenzsystems widmet sich diese Arbeit.

1.2 Aufgabenstellung

Bei der Anwendung kommen

Im folgenden wird zunächst zur Abgrenzung dieser Arbeit innerhalb des Referenzsystems, der grundlegende Aufbau des Referenzsystems beschrieben.

1.2.1 Der Roboter

Auf der Ebene der Hardware steht der mobile Roboter. Dieser hat das Ziel sich genau zu lokalisieren um Referenzwerte für die montierten, mobilen Sensorknoten zu liefern. Da er sich während der Testfahrten indoor bewegt, kann er für seine Lokalisierung keine satelliten-gestützten Lokalisierungssysteme wie GPS verwenden, aus den in ?? genannten Gründen. Zwei weitere Ansätze, um die Bewegung des Roboters nachzuvollziehen und somit Positionsdaten zu gewinnen sind:

- Inertial Navigation
- Odometrie

Bei der inertial Navigation wird mithilfe von Beschleunigungs- und Gyromessungen auf die ausgeführte Bewegung geschlossen. Diese Messungen lassen sich durch Microelectromechanical systems (MEMS), die in einer Inertial Measurement Unit (IMU) zusammengefasst werden durchführen. MEMS werden in großen Stückzahlen produziert und sind kostengünstig. Typischerweise sind aber die Messungen, auch bei Stillstand, durch Jitter belastet. Dieser kann zwar durch geeignete Filter geglättet werden, lässt sich allerdings nicht ganz ausschließen. Auf längere Strecken entsteht durch die Aufsummierung der Fehler ein Drift, fort von der tatsächlichen Position.

Bei der Odometrie, werden die Antriebsdaten ausgewertet, um auf die Bewegung des Roboters zu schließen. Geht man davon aus, dass der Untergrund, auf dem der Roboter fährt, für dessen Räder geeignet ist und die Räder nicht wegen beispielsweise mangelnder Bodenhaftung stark durchdrehen. So kann sie auf kurzen Strecken sehr genaue Abschätzungen liefern.

Allerdings ist auch eine Odometriemessung stets mit einem Fehler behaftet. Dieser Fehler summiert sich über die Zeit auf und die geschätzte Position weicht immer weiter von der tatsächlichen ab. Man hat also auf längeren Strecken ebenfalls mit einem Drift zu rechnen.

Beide Methoden haben gemeinsam, dass sie unabhängig von Informationen aus der Umgebung des Roboters arbeiten. Somit können sie allerdings den beschriebenen Drift in der Lokalisierung niemals korrigieren, da sie nicht die Positions auf Plausibilität mit der Umgebung abgleichen. Für das Referenzsystem ist Drift aber nicht akzeptabel. Aus diesem Grund erfasst der Roboter Abstände zu Hindernissen seiner Umgebung mithilfe einer *Microsoft Kinect*. Die Kinect erstellt mithilfe eines, im infrarot Bereich gestrahltem, optischen Musters ein Tiefenbild. Die Reichweite liegt dabei bei maximal 10 Meter Entfernung bei einem Blickwinkel von ca 59°. Test haben gezeigt, dass die Genauigkeit der Tiefenmessung mit zunehmender Entfernung abnimmt. Im Nahbereich von zwei Metern aber überraschend präzise Abstandsauflösung im Zentimeterbereich ermöglicht. Außerdem verfügt der Roboter über eine Karte der Testumgebung. Diese Karte in Kombination mit der *Microsoft Kinect* ermöglicht während einer Testfahrt eine Lokalisierung durch folgende entscheidende Schritte durchzuführen:

1. Abschätzung der derzeitigen Pose durch Odometrie
2. Abgleich mit Karte und Korrektur der Pose

Zum Abtasten der Umgebung hätte alternativ auch ein Laserscanner gewählt werden können, welcher hohe Reichweite mit hoher Genauigkeit kombiniert. Dies wäre aber entgegen den Ziele des Referenzsystems, mit zu hohen Anschaffungskosten verbunden gewesen. Im Sinne günstiger Kosten wurde schlussendlich ein sogenannter *TurtleBot* gebaut. Dies ist ein von *WillowGarage* spezifizierter low-cost Roboter. Im Kern besteht dieser aus einem *Roomba* Staubsaugerroboter von *iRobot*, einer *Microsoft Kinect* und einem Tragegerüst. Das Tragegerüst dient als Abstellfläche für einen Laptop und bietet im Anwendungsfall des Referenzsystems Platz zum Montieren der Sensorknoten.

Die Aspekte der Software, zum Betrieb des Roboters, wurde innerhalb einer anderen Bachelorarbeit, ebenfalls im Rahmen der Entwicklung des Referenzsystems, ausführlich erarbeitet.

Zusammenfassend kann man in dieser Hinsicht feststellen, dass zum autonomen Fahren Programme des ROS genutzt werden sowie eine Software implementiert wurde, die den Roboter vorgegebene Wegpunkte abfahren lässt und dabei gesammelte Lokalisierungsdaten innerhalb von ROS bereitstellt. Genauer wird im Teil ?? auf die Funktionsweise von ROS eingegangen.

KAPITEL 2

Pathcompare - Implementierung

2.1 Technischer Rahmen

2.1.1 Robot Operating System - ROS

Innerhalb des Referenzsystems wird ROS im Zusammenhang mit der Steuerung des Roboters genutzt und um die, während einer Testfahrt gewonnenen Pfaddaten, zur Verfügung zu stellen. Im Folgenden wird genauer auf die Fähigkeiten und Ziele von ROS eingegangen.

Obwohl der Name zunächst anderes vermuten lässt ist ROS kein Betriebssystem im klassischen Sinne. Es ist ein Framework, welches auf ein Betriebssystem angewiesen ist um ausgeführt werden zu können. Es bietet aber Funktionalitäten, die abstrahiert betrachtet Betriebssystemfunktionen ähneln. Charakteristisch ist hierbei ROS Fähigkeit lokal- oder nicht-lokalausgeführte Programme zu Verbinden und eine strukturierte Kommunikation zwischen diesen zu ermöglichen. Die wesentlichen Elementareigenschaften der Grundphilosophie sind:

- multi-tool Ansatz
- peer-to-peer Kommunikation
- keine feste Bindung an Programmiersprache
- frei und Open-Source

Multi-tool Ansatz bedeutet, dass ROS die Fähigkeiten verschiedener Programme und Libraries zur Verfügung stellt. Diese sind jedoch nicht fest in den Kern von ROS eingebaut sondern modular integriert. Als analoges Beispiel in Hinblick auf Betriebssysteme kann man ROS also als einen Mikrokern verstehen. Dies bietet den Vorteil, dass ROS selbst vergleichsweise klein ist und nur wirklich gebrauchte tools geladen werden müssen. Die peer-to-peer Kommunikation bezieht sich auf die Kommunikation zwischen diesen, in ROS integrierten, Modulen. Diese wird durch den ROS Kern gesteuert. Der Kern von ROS ist nativ in C++ implementiert, es existieren jedoch bereits Portierungen in andere Sprachen wie Python, Octave und Lisp um die ROS-Application Programming Interface (API) einer größeren Zahl von Entwicklern und Projekten die Nutzung zu ermöglichen. Weitere Portierungen sollen sich in der Implementierung befinden.

ROS ist darüberhinaus frei Verfügbar und Open-Source. Man kann beliebige Programme, als Module zur Erweiterung und Nutzung von ROS hinzufügen, wie es auch im Rahmen des Referenzsystems geschehen ist. Bei allgemeinem Nutzern und gegebener Pflege, besteht die Möglichkeit, dass diese offiziell zu ROS hinzugefügt werden.

Um die konkreten Abläufe und Komponenten innerhalb von ROS veranschaulichen zu können und damit auch den Bezug zu Pathcompare herstellen zu können, ist es zunächst erforderlich die Begrifflichkeiten innerhalb von ROS zu klären. Im Folgendem werden diese aufgezeigt.

Im Zentrum von ROS steht der sogenannte *master*. Dieser wird als einzelne Instanz gestartet und wartet dann darauf, dass sich tools, die im Kontext von ROS gestartet werden, bei ihm anmelden. Ein gestartetes tool wird innerhalb von ROS als *node* bezeichnet. Ist der *master* nicht gestartet können auch keine *nodes* gestartet werden. Die *nodes* sind also alle zunächst auf Kommunikation mit dem *master* angewiesen. Diese Kommunikation kann lokal oder nichtlokal ausgeführt werden, d.h. der *master* kann sich auch auf einem anderen Rechner wie der *node* befinden solange eine http Verbindung zwischen beiden hergestellt werden kann. Denn das Anmelden des *nodes* beim master erfolgt über einen XML - Remote Procedure Call (XML-RPC), getragen vom http. Für den tool Entwickler auf Anwendungsebene ist diese Kommunikation zur Anmeldung allerdings völlig unsichtbar und er muss sich nicht darum kümmern. Das Ausführen von *nodes* auf unterschiedlichen Rechnern ist natürlich ebenso möglich und wie zuvor bereits erwähnt eines der Kernfunktionen von ROS. Dies lässt sich vorteilhaft ausnutzen durch zum Beispiel:

- Verteilung oder Auslagerung rechenintensiver *nodes* auf potente Hardware
- Zusammenführung von an unterschiedlichen Stellen gewonnener Daten.

So muss beispielsweise ein mobiler Roboter Bilderkennungs Aufgaben nicht selbst ausführen, sondern kann diese an einen *node* weiterleiten der auf einem Rechencluster ausgeführt wird. Die zweite genannte Möglichkeit ist auch besonders im Bezug auf diese Arbeit wichtig, da Pfaddaten des Roboters und der zu testenden Sensorknoten für *Pathcompare* verfügbar gemacht werden müssen. Die Kommunikation zwischen Nodes erfolgt über sogenannte *messages*, diese enthalten die serialisierte Form der zu übertragenden Daten. ROS bietet in seinen Kernpaketen bereits zahlreiche Definitionen für unterschiedliche *message* Typen, aber es ist auch möglich eigene zu generieren und dies wird von zahlreichen Paketen getan um Daten maßgeschneidert übertragen zu können. Einmal definierte *message* Typen können wiederum rekursiv in anderen *message* Typen verwendet werden. Ein Beispiel für eine message ist in Quellcode 2.1 dargestellt; eine Transformation die in ROS *geometry_msgs* definiert ist. Sie besteht wie erkennbar aus den zwei Typen *Vector3* und *Quaternion*. Letzterer beschreibt die Rotation und der erste die Translation. Zusammengefügt ergibt dies eine Transformationsnachricht.

```

1 geometry_msgs/Vector3 translation
2   float64 x
3   float64 y
4   float64 z
5 geometry_msgs/Quaternion rotation
6   float64 x
7   float64 y
```



```

8   float64 z
9   float64 w

```

 Quellcode 2.1: ROS transformation message

Soll ein *node* seine *messages* anderen *nodes* senden können, so muss dies zunächst durch festlegen einer sogenannten *topic* beim *master* angemeldet werden. Dann wird diese *topic* für andere *nodes* im ROS über den *master* sichtbar. Eine *topic* besteht im wesentlichen aus zwei Teilen, einer *topic-id* und einem *message-type*. Wobei der *message-type* angibt welcher Typ von *messages* über diese *topic* verschickt wird. Die *topic-id* dient zur eindeutigen Identifikation innerhalb des ROS. *Nodes* welche *messages* einer *topic* empfangen sollen, müssen diese *topic* dann beim *master* abonnieren. In programmatischer Hinsicht wird beim Empfang neuer Nachrichten innerhalb des *nodes* eine festgelegte callback Methode aufgerufen um die *messages* zu bearbeiten. Treffen dabei Nachrichten mit einer höheren Frequenz ein, als abgearbeitet werden können, so kommt es irgendwann zu Verlusten wenn die einstellbare Größe der *message queue* überschritten wird.

Zwei weitere wichtige Begriffe in ROS betreffen die Organisierung der Dateien die zu den einzelnen tools gehören. Dies sind:

- package
- stack

Ein *package* beinhaltet den Code, Libraries sowie die ausführbare Datei eines tools bzw. *nodes*. In ROS sind für packages bestimmte Ordnerstrukturen und Dateien festgelegt sodass mithilfe der von ROS mitgebrachten tools *packages* leicht gebaut, gesucht und gestartet werden können. Beispielsweise basiert das ROS build System auf *cmake* und so ist eine vorkonfigurierte *CMakeLists.txt* in jedem *package* grundsätzlich enthalten. Eine Zusammenfassung mehrerer *packages* wird als *stack* bezeichnet.

Pathcompare ist also im Bezug auf ROS, während der Ausführung, ein einzelner *node* welcher *topics* abonniert um *messages* zu empfangen. Es wird in einem späteren Teil darauf eingegangen, welche *messages* das genau sind. Alle zum Kompilieren und Ausführen nötige Dateien sind dabei in zwei *packages* namens *pathcompare* und *pathcompareplugins* aufgeteilt.

- Warum brauchen wir ROS?
- Wer entwickelt ROS? (Stanford University & WillowGarage & community)
- Was sind die Grundeigenschaften
- multitool Ansatz
- strukturierte Kommunikation zwischen den Tools
- free and open source
- multilingual
- Begrifflichkeiten des ROS (Stack, Package, Node, Master, Topic, Message)
- Topologie (figure: Roboter, Master, Pathcompare, Sensorknoten)
- plattformen: Ubuntu
- standard Buildsysteem *cmake*

2.1.2 Qt

Pathcompare ist darauf ausgelegt alle Informationen für den Nutzer in einer Benutzeroberfläche (fortan als GUI bezeichnet) zu visualisieren. Da die Anbindung an ROS über C++ erfolgt, lag nahe auch die GUI in C++ umzusetzen. Dazu wurde das Qt Framework gewählt. Qt ist in C++ implementiert, wobei allerdings auch Anbindungen für zahlreiche andere Sprachen wie z.B. Java, C#, Ruby oder Python existieren. Die Entwicklung von Qt begann 1991 und ist zum Zeitpunkt des Schreibens in der Version 4.7.4 verfügbar. Das Framework besteht dabei mittlerweile nicht mehr nur aus reinen GUI Bibliotheken, sondern stellt auch z.B. Netzwerk-, SQL- und andere Anwendungs-Bibliotheken zur Verfügung.

Für die Entwicklung von *Pathcompare* waren neben den GUI Bibliotheken, besonders folgende Konzepte und Funktionen beim Entwickeln vorteilhaft:

- Signal-Slot Konzept
- Plattformunabhängigkeit
- Containerklassen mit praktischen Hilfsmethoden
- graphischer GUI Designer

Auf einige dieser Punkte und deren Bezug zu Pathcompare wird nun kurz eingegangen.

Das *Signal-Slot* Konzept dient dazu Objekten, bestimmte Veränderungen an Objekten mitzuteilen, welche über diese Änderung informiert werden sollen. Es realisiert das Entwurfsmuster des *Observer patterns*. *Signal-Slot* erspart es dem Programmierer einen Verweis auf das beobachtende Objekt, durch einen registrierenden Methodenaufruf beim aktualisierenden Objekt, zu hinterlegen. Stattdessen emittiert, im Falle einer mitzuteilenden Aktualisierung, das aktualisierende Objekt eine Methode, welche durch einen Qt Makro als *signal* ausgezeichnet ist. Bei Beobachter Objekten, die auf dieses Signal reagieren sollen, wird dann eine als *slot* deklarierte Methode selbständig aufgerufen, sofern die Objekte, durch einen Aufruf einer von Qt bereitgestellten, statischen Methode, verbunden wurden. Dieses Konzept vereinfacht die Entwicklung wegen seiner Flexibilität deutlich, allerdings ist der durch Qt erzeugte Code, welcher bei Auflösung der Makros entsteht, aufblühend und dadurch theoretisch langsamer bei der Ausführung als eine klassische callback Implementierung. In der Praxis ist diese Verlangsamung aber unmerklich und wiegt nicht die Vorteile für den Entwickler auf. Deshalb wurde das Konzept auch bei der Entwicklung von *Pathcompare* eingesetzt.

Im Zusammenhang mit dem Signal-Slot Konzepts wurde bereits erwähnt, dass Qt C++ mit verschiedenen Makros erweitert. Diese Makros werden dabei nicht immer direkt in gültigen C++ Code übersetzt, sondern dienen als Annotationen. Dies hat Folgen für den Build Vorgang, denn die mit Annotationen versehenen Klassen müssen dann zuerst, mit dem von Qt bereitgestelltem Meta Object Compiler (MOC) übersetzt werden. Standardmäßig wird in Qt das Build Programm qmake verwendet, welches die nötigen Aufrufe des MOC automatisch veranlasst. In Hinblick auf *Pathcompare* war aber eine Integration in das von ROS genutzte Build verfahren cmake notwendig. Hierbei muss beachtet werden, dass cmake die Dateien welche einen MOC Aufruf notwendig machen, in der derzeitigen Version, nicht selbständig erkennt. Diese müssen manuell in der *CmakeLists.txt* deklariert werden. Die sonstige Einbindung Qt spezifischer Libraries und deren Verlinkung mit der Applikation ist in cmake leicht möglich. Somit war die Verwenung von Qt in *pathcompare* kein Hindernis für die Verwendung von cmake. Die Nutzung von cmake wird darüberhinaus, für große Projekte,

von der Qt Dokumentation selbst empfohlen.

Neben dem MOC existiert noch ein sogenannter User Interface Compiler (UIC). Dieser tritt im Zusammenhang mit dem *QtDesigner* in Erscheinung. Der *QtDesigner* ermöglicht es graphisch Qt GUIs zu erstellen. Dabei wird eine XML Datei vom Designer erstellt, welche den Aufbau der GUI abbildet. Der UIC ist dann dafür verantwortlich diese Datei in C++ Klassen zu übersetzen. Auch hier muss cmake veranlasst werden zunächst für einen glsUIC entsprechender Dateien zu sorgen.

- Warum braucht Pathcompare Qt?
- GUI framework, native in C++ (auch Implementierungen für Java, Ruby,..)
- bietet auch zahlreiche nicht GUI spezifische Funktionalität
- cross platform
- native UI rendering
- UI designing with Qt Designer
- bereichert C++ durch embedded Macros (z.B. signal&slot) -> moc compiler
- standard Buildsysteem native qmake
- Unterstützung für cmake gegeben, für große Projekte sogar empfohlen

2.2 Design der Software

Der Hauptfokus beim Design der GUI lag darauf, einfache Benutzung und übersichtliche Testdatendarstellung, für den Nutzer zu gewährleisten. Außerdem sollte die Software leicht erweiterbar sein.

2.2.1 Überblick Gesamtsystem

Im Folgendem wird die Software in einer Gesamtansicht dargestellt und anschließend auf die Funktionsweise ihrer Einzelkomponenten genauer eingegangen.

Wie in vorherigen Teilen bereits erwähnt ist *Pathcompare* durch Plug-ins erweiterbar. Dadurch kann man zunächst das Gesamtsystem logisch in zwei Teile untergliedern, nämlich in

- Rahmen
- Plug-ins

Diese Aufteilung spiegelt sich am offensichtlichsten auch in der GUI wider. Hierbei bildet das UI des Rahmens die Grundlage für die Visualisierung von Plug-ins. Dabei wird den Plug-ins, nachdem sie geladen wurden, ein Platz innerhalb des Rahmens zugewiesen. Innerhalb dieses zugewiesenen Platzes können sie beliebige eigene UI Elemente laden und haben die volle Kontrolle über diese. Konkret gehören dabei folgende UI Elemente zum Rahmen:

- Hauptfenster
- Topic Übersicht
- Plug-in Tab-Fenster

Das Hauptfenster dient dabei als Grundlage Topic Übersicht und das Plug-in Tab-Fenster. Topic Übersicht und das Plug-in Tab-Fenster sind vertikal getrennt. Die Topic-Ansicht ist dabei ganz links angeordnet, siehe . Rechts neben der Topic Übersicht schließt unmittelbar das Plug-in Tab-Fenster an. Wird ein Plug-in geladen, wird in diesem Tab-Fenster ein neuer Tab angelegt. Die Fläche dieses Tabs wird diesem Plug-in dann zur Verfügung gestellt um seine UI darauf aufzubauen. Auf Details bezüglich des Ladens der Plug-ins und deren Funktionsweise wird in einem folgendem Abschnitt eingegangen.

Die Topic Übersicht zeigt die momentan im ROS verfügbaren Topics an. Um die Übersicht auf die Topics zu strukturieren sie derart gestaltet, dass Topics desselben *message* Typs Gruppieren werden. Diese Gruppen werden dann kompakt im Topic Übersicht Fenster angezeigt und können dort durch den Nutzer aufgeklappt werden um alle *topics* eines *message* Typs anzuzeigen.

- Rahmen mit Topic TreeView
- Anbindung an ROS durch ROSManager
- TabFlächen für einzelne PlugIns
- PluginLoader lädt Plugins

2.2.2 Plug-in Main Compare

- Funktionen erklären
- Referenz Pfad Selektion
- Export (+Format der csv Datei)
- Tabellenansicht erklären

Pfadvergleichsverfahren

2.2.3 Plug-in Konzept allgemein

- allgemein Qt Plugins
- Vorstellen der Interfaces zum Schreiben eigener Plug-ins für Pathcompare
- Beispiel Camera Plugin

2.3 Anwendung