Freie Universität Berlin
Fachbereich Mathematik und Informatik
Lehrstuhl für Künstliche Intelligenz

Bachelorarbeit der Informatik

Portierung der FUmanoid-Software auf den NAO-Roboter

von *Michael Schmidt*

Gutachter:
Prof. Dr. Raúl Rojas

Betreuer:
Dipl.-Inf. Daniel Seifert

4. Dezember 2013

Eidesstattliche Erklärung

Ich versichere, die Bachelorarbeit selbstständig und lediglich unter Benutzung der angegebenen Quellen und Hilfsmittel verfasst zu haben. Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht im Rahmen eines anderen Prüfungsverfahrens eingereicht wurde.

Berlin, den 4. Dezember 2013 _____

Michael Schmidt

Zusammenfassung

An der Freien Universität Berlin werden im Rahmen des Projekts FUmanoids fußballspielende Roboter entwickelt. Um ihre Leistungsfähigkeit zu evaluieren, treten sie jedes Jahr im RoboCup gegen andere Mannschaften an. Dabei sind eine Vielzahl von Problemen zu lösen, für die es wünschenswert ist, sie auf verschiedenen Plattformen zu testen. Aus diesem Grund wurde entschieden, die vorhandene Software auf den NAO-Roboter zu portieren, welcher in der Standard Platform League des RoboCups verwendet wird. Dieses Vorhaben wird im Zuge dieser Arbeit realisiert. Dabei war es notwendig die Software der FUmanoids so anzupassen, dass sie eine generische Umgebung für verschiedene Roboterplattformen bildet. Insbesondere wurden passende Schnittstellen für Aktuatorik und Sensorik definiert und implementiert. Weiter wurden beispielhaft einige Anwendungsfälle für den NAO-Roboter implementiert.

Inhaltsverzeichnis

1 Einleitung

Bis in das letzte Jahrzehnt des 20. Jahrhunderts war das Schachspiel ein beliebtes Problem im Bereich der künstlichen Intelligenz. Spätestens nachdem 1997 der menschliche Weltmeister im Schach von einem Computer names *Deep Blue* besiegt wurde, begannen Forscher auf aller Welt sich ambitionierteren Zielen zu stellen. Alan Mackworth schlug dabei 1993 als erster vor, Roboter Fußball spielen zu lassen. [14] Dabei handelt es sich um eine weitaus anspruchsvollere Aufgabe als Schach, da viele verschiedene Themenschwerpunkte aus den Bereichen Robotik und Künstliche Intelligenz zusammenkommen.

Auch an der Freien Universität Berlin findet aus diesem Grund die Entwicklung Fußball spielender Roboter statt. Im Rahmen des Projekts *FUmanoids* konstruieren und programmieren Studenten und wissenschaftliche Mitarbeiter zusammen eigene Roboter, deren Leistungsfähigkeit anschließend in verschiedenen Wettbewerben evaluiert wird. Dabei ist es häufig hilfreich Lösungsstrategien für aufkommende Probleme auf verschiedenen Robotermodellen zu testen, um einen Ansatz in seiner Güte zu bewerten. Aus diesem Grund wurde entschieden die vorhandene Software auf den NAO-Roboter zu portieren, der seit 2006 von der französischen Firma Aldebaran Robotics hergestellt wird. Die Realisierung dieses Vorhabens im Rahmen dieser Arbeit hat jedoch nicht zum Ziel, die komplette Funktionalität der FUmanoid-Software auf den NAO-Roboter zu übertragen. Vielmehr geht es darum eine geeignete Infrastruktur zu schaffen, an die weitergehende Arbeit anknüpfen kann.

In der nun folgenden Arbeit werden zunächst einige Grundlagen dargelegt. Es folgt eine Übersicht über eine Reihe von Software-Frameworks, welche für sich beanspruchen, eine generische Umgebung für unterschiedlichste Roboterplattformen zu bilden. Anschließend werden der NAO-Roboter und die Roboter der FUmanoids in Bezug auf Hardware und Software vorgestellt. Zum Abschluss wird die erfolgte Umsetzung und die dazu notwendigen Schritte beschrieben. Außerdem wird skizziert, welche weiterführende Arbeiten darauf aufbauen können.

1.1 Der RoboCup

Im Jahr 1997 wurde der erste offizielle RoboCup veranstaltet, bei dem aus Robotern bestehende Mannschaften gegeneinander antreten. Als offizielles Ziel des RoboCups wurde proklamiert, im Jahr 2050 den amtierenden Fußballweltmeister der FIFA besiegen zu können. Bis heute finden deshalb jedes Jahr Weltmeisterschaften im RoboCup an wechselnden Orten statt, die sich stetig wachsendem Zuspruch erfreuen. So nahmen in diesem Jahr im niederländischen Eindhoven über 2500 Teilnehmer aus mehr als 40 Ländern teil. Mittlerweile finden im Rahmen des RoboCups aber auch Wettbewerbe statt, bei denen sich Roboter in Rettungseinsätzen oder bei Haushaltsaufgaben beweisen müssen. Neben der Weltmeisterschaft finden aber auch auf nationaler Ebene regelmäßig Wettkämpfe und andere Veranstaltungen statt, wie die jährlich stattfindenden German Open.

Die Wettbewerbe werden dabei in verschiedenen Ligen ausgetragen, um den unterschiedlichen Schwerpunkten der Forschung Rechnung zu tragen. Es existieren Ligen, in denen der Wettkampf in einer zwei- oder dreidimensionalen Welt simuliert wird. Durch die fehlende Hardware kann sich komplett auf das Spielgeschehen konzentriert werden. In der *Small Size League* und der *Middle Size League* werden dagegen reale Roboter eingesetzt, die sich jedoch auf Rändern fortbewegen.

Der NAO-Roboter, um den es in dieser Arbeit größtenteils geht, findet in der *Standard Platform League* Verwendung. Ihn zu nutzen ist sogar ausdrücklich im Regelwerk festgeschrieben, wobei auch Modifikationen am Roboter verboten sind. Die Spieler agieren vollständig autonom und dürfen nur mittels WLAN kommunizieren. [13].

Das Roboterfußball-Team der Freien Universität Berlin, die *FUmanoids*, spielen hingegen in der *Humanoid League*. Die Roboter müssen hier menschliche Proportionen besitzen und dürfen auch nur auf entsprechende Sensorik zurückgreifen. Kommunikation zwischen den Spielern mittels WLAN ist aber erlaubt. Die Liga ist außerdem in drei größenspezifische Spielklassen unterteilt. In der Kid-Size-Klasse ist eine Größe von 30 bis 60 cm zulässig, in der Teen-Size-Klasse 100 cm bis 120 cm und in der Adult-Size-Klasse 130 cm bis 160 cm, in Ausnahmefällen auch bis zu 180 cm. Die *FUmanoids* spielen in der Kid-Size-Klasse, in der je Team bis zu drei Spieler auf dem Feld stehen dürfen [12].

2 Grundlagen

2.1 Entwurfs- und Architekturmuster

Ein Entwurfsmuster definiert eine allgemeine Struktur mehrerer Klassen, die durch ihr Zusammenwirken eine bestimmte Problemstellung lösen. Im Gegensatz dazu beschreiben Architekturmuster die grundlegende Organisation und Interaktion zwischen den Komponenten einer Anwendung. [4] Da im Rahmen dieser Arbeit einige von ihnen diskutiert oder angewendet werden, sollen sie hier zunächst vorgestellt werden.

Das **Singleton Pattern** gehört zur Gruppe der Erzeugungsmuster und kommt zur Anwendung, wenn von einer Klasse nur eine Instanz, das sogenannte *Singleton*, existieren soll. Dazu wird die Sichtbarkeit des Konstruktors der Klasse nach außen hin eingeschränkt. Der Zugriff auf das *Singleton* ist nur durch eine statische Instanzoperation möglich. [4]

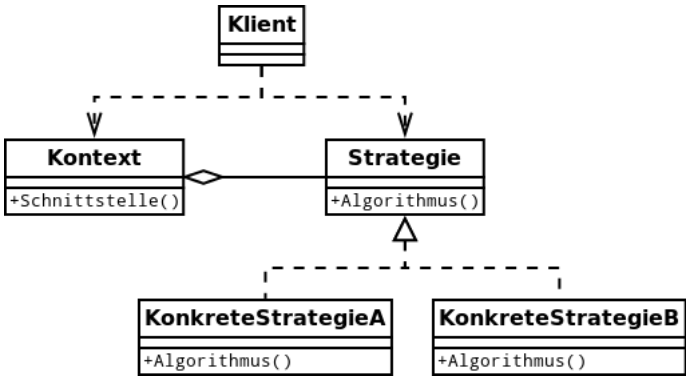

Abbildung 1: Strategie Pattern

Durch das **Strategie Pattern** wird eine Familie von austauschbaren Algorithmen definiert. Es gehört somit zur Kategorie der Verhaltensmuster. Die Algorithmen implementieren hierbei, wie in der Abbildung zu sehen, einen abstrakten Datentyp. Der *Kontext* besitzt wiederum eine Instanz dieses Datentyps und stellt seine Funktionalität nach außen hin für den Klienten zur Verfügung. Für diesen ist nun nicht von Bedeutung, welche konkrete Strategie sich hinter dem Kontext verbirgt. [4] Des weiteren können die verwendeten Algorithmen zur Laufzeit flexibel ausgetauscht werden.

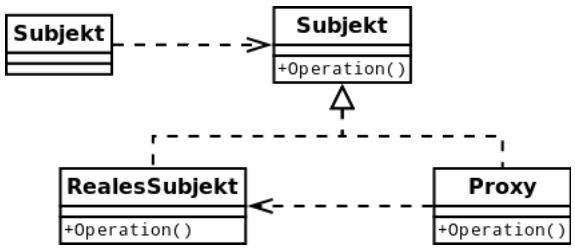

Abbildung 2: Proxy Pattern

Das **Proxy Pattern**, auch Stellvertreter oder *Surrogate* genannt, gehört zu der Kategorie der Strukturmuster. Es kann zur Anwendung kommen, wenn ein Klient auf ein Subjekt zugreifen möchte. Der *Proxy* implementiert dabei die gleiche Schnittstelle wie das reale Subjekt. Anschließend ruft er die Funktionalität des realen Subjekts auf und schiebt sich so zwischen den Klienten und das eigentliche Subjekt. Für den Klienten ist also nicht ersichtlich, ob er mit dem realen Subjekt oder einem *Proxy* kommuniziert. Diese Struktur ist besonders gut geeignet, wenn sich Klient und reales Subjekt in verschiedenen Adressräumen befinden. Dann agiert der *Proxy* als lokale Repräsentation. [4]

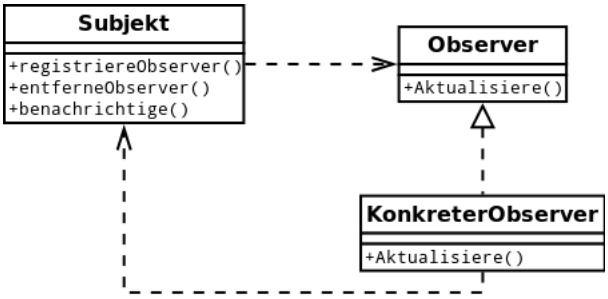

Abbildung 3: Observer Pattern

Das **Observer Pattern** gehört zur Kategorie der Verhaltensmuster. Alle *Observer* implementieren eine Schnittstelle, welches eine Funktion zur Benachrichtigung eben dieser definiert. Anschließend registrieren sich die *Observer* beim Subjekt. Ändert das Subjekt seinen Zustand, kann es die *Observer* mittels der Schnittstelle darüber informieren. Darauf können diese wiederum reagieren. [4] Es findet also eine Umkehrung des Kontrollflusses statt, da die *Observer* nicht aktiv handeln müssen, sondern benachrichtigt werden. Dieses als *Inversion of Control* bezeichnete Paradigma ist ein wesentliches Merkmal diverser Software-Frameworks.

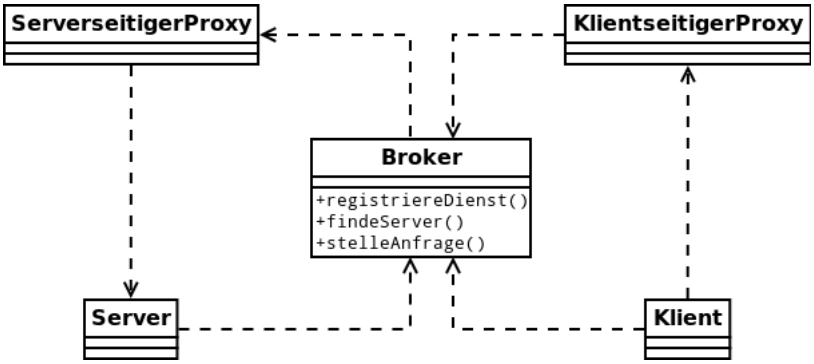

Abbildung 4: Broker Architektur

Bei einem **Broker** handelt es sich um eine Architektur, die sich besonders gut zur Strukturierung verteilter Systeme mit entkoppelten Komponenten anbietet. Es wird unterschieden zwischen Klienten, welche gewisse Dienst in Anspruch nehmen wollen, und Servern, welche diese Dienste anbieten. Dem *Broker* kommt dabei die Aufgabe zu, die Kommunikation zwischen ihnen zu abstrahieren. Dazu registrieren sich die Server mitsamt der von ihnen angebotenen Schnittstellen beim *Broker*. Klienten können anschließend erfragen, welche Dienste verfügbar sind. Will ein Klient auf einen Dienst zugreifen, macht der *Broker* den dazugehörigen Server ausfindig und leitet die Anfrage weiter. Anschließend übermittelt er das Ergebnis zurück zum Klienten. [2] In dieser Architektur können darüber hinaus bereits vorgestellte Entwurfsmuster zur Anwendung kommen. Befinden sich die verschiedenen Komponenten, einschließlich des *Brokers*, in unterschiedlichen Adressräumen, kann ein lokaler *Proxy* die entfernten Schnittstellen repräsentieren. Häufig ist auch eine eine Umkehr des Kontrollflusses mittels des *Observer Patterns* implementiert, wenn der Broker die Klienten über des Ergebnis einer Anfrage informiert.

Als **Blackboard** wird eine Architektur bezeichnet, welche sich zum Lösen komplexer Probleme anbietet. Das Blackboard stellt hierbei einen Datenstruktur dar, auf der von logisch getrennten Prozessen verschiedene Informationen abgelegt werden. Diese werden von den einzelnen Prozessen wiederum dafür genutzt, inkrementell Teillösungen des Gesamtproblems zu finden. Der Vorgang terminiert, wenn eine akzeptable Lösung gefunden worden ist oder dies auf Grund von Informationsmangel nicht möglich ist. Die Interaktion zwischen den Prozessen findet dabei ausschließlich über das Blackboard statt. Ein dezidierter Kontrollfluss ist dabei nicht vorgegeben. Jedoch liegt es nahe, dass die einzelnen Prozess über Änderungen auf dem Blackboard informiert werden, wobei das *Observer Pattern* dienlich ist. [16] Die Architektur eignet sich besonders für Probleme, für die es keine realisierbare deterministische Lösung gibt. [2]

2.2 Statische Bewegungen und Keyframing

In der Robotik wird zwischen dynamischen und statischen Bewegungen unterschieden. Menschen führen ihre Bewegungen dynamisch aus, das heißt sie variieren sie leicht je nach Gegebenheit. Zum Beispiel wird der Gleichgewichtssinn beim Gehen mit einbezogen, um ein Umkippen zu vermeiden. Dies ist für einen Roboter jedoch nicht einfach zu realisieren. Statische Bewegungen werden hingegen immer auf die gleiche Weise ausgeführt. Sie bilden also mathematisch die Zeit auf die Winkel der Gelenke des Roboters ab. Damit eignen sie sich gut, um ganz konkrete Probleme zu lösen. Leider können schon kleinere Modifikationen der Hardware oder ein veränderter Untergrund zu Instabilität bei der Bewegung führen. Da beim Abspielen eines statischen Bewegungsablaufs auch nicht auf veränderte Umwelteinflüsse reagiert wird, kann im schlimmsten Fall sogar eine Gefährdung für den Roboter oder andere bestehen. Deshalb werden für komplexere Probleme, wie zum Beispiel das Laufen, dynamische Bewegungen genutzt. [6]

Eine Technik namens **Keyframing** kommt zum Einsatz, um statische Bewegungen zu beschreiben. Dabei wird eine Menge von Gelenkwinkeln spezifiziert, die sogenannten *Keyframes*. Diese werden zu bestimmten Zeitpunkten durch die verschiedenen Gelenke den Roboter eingenommen. Um eine vollständige Trajektorie zu beschrieben, werden die dazwischen liegenden Werte interpoliert. Das dabei eingesetzte Interpolationsverfahren hat starken Einfluss auf die Bewegung. Eine lineare Interpolation würde so zum Beispiel in viele Fälle zur ruckartigen Änderungen im Bewegungsablauf führen. Deshalb sind Verfahren auf Grundlage von Bézierkurven oder B-Spline-Kurven verbreitet, die zu einer Glättung der Trajektorie führen. Es lässt sich resümieren, dass **Keyframing** die populärste Technik ist, um statische Bewegungen zu erzeugen. [7]

3 Verwandte Umsetzungen

Humanoide Roboter machen es notwendig auf eine Vielzahl heterogener Komponenten zuzugreifen. Wünschenswert ist es hierbei häufig, die konkrete Hardware soweit zu abstrahieren, dass die Software auf verschiedenen Systemen lauffähig ist. Dabei ist der Einsatz eines gut strukturierten Frameworks sinnvoll, das diese Aufgabe übernimmt. Wenn hier von einem Framework die Rede ist, dann ist damit eine Software gemeint, welche generische Funktionalitäten vorgibt, welche durch die Anwendung weiter spezifiziert wird. Im Gegensatz zu einer Bibliothek wird dabei neben der Bereitstellung von Basisfunktionalität auch der Kontrollfluss teilweise vorgegeben. Auf diese Weise wird die Implementierung der eigentlichen Anwendung erleichtert. [17]

Im Rahmen dieser Arbeit war es jedoch kein Ziel, grundsätzliche Änderungen an der Architektur der FUmanoid-Software vorzunehmen. Dennoch mussten Anpassungen in Bezug auf die Schnittstellen vorgenommen werden, damit Unterstützung für die verschiedenen Roboter gewährleistet ist. Um eine gewisse Vergleichbarkeit herzustellen, sollen hier nun andere Software-Plattformen vorgestellt werden, welche teilweise auch den Anspruch haben, ein umfassendes Framework für die heterogenen Systeme der Robotik zu bilden.

3.1 Player

Die Plattform *Player* wurde im Robotics Research Lab der University of Southern California (USC) entwickelt, um die dort eingesetzte Roboterplattform *Poineer* zu steuern. Ihr liegt die Idee zu Grunde die Hardware über einen zentralen Server, *Player Abstract Device Interface* genannt, zu abstrahieren. Der Server agiert dabei als eine virtuelle Maschine. Die Sensoren und Aktuatoren werden hierbei als Dateien modelliert, auf denen mit Datenströmen gearbeitet wird. Die verschiedenen Clients können auf die Komponenten nebenläufig zugreifen. Ähnliche Hardwarekomponenten werden in Geräteklassen zusammengefasst, für die jeweils eine einheitliche Schnittstelle definiert ist. Passende Gerätetreiber implementieren wiederum diese Schnittstellen, so dass große Abstraktion erreicht wird und die Plattform unverändert auch einer Vielzahl von realen und simulierten Geräten laufen kann. [24] Von Nachteil ist es jedoch, dass keine Vorgaben bezüglich höherer Abstraktionsschichten gemacht wird und die Implementierung komplett dem Anwender obliegt.

3.2 Yet Another Robot Platform

Yet Another Robot Platform (YARP) wurde am Italian Institute of Technology als offene Plattform für humanoide Roboter entworfen. Die Entwickler beschreiben YARP als Menge von Bibliotheken, Protokollen und Werkzeugen, um Softwaremodule und Hardwarekomponenten voneinander zu trennen. Um dies zu realisieren bedienen sie sich einer breiten Palette von Entwurfsmustern und Paradigmen der objektorientierten Programmierung. Komponenten werden in hierarchischen Gruppen zusammengefasst, welche über möglichst allgemeingültige Schnittstellen verfügen. Entwickler von Komponenten sind dazu aufgefordert zu den Schnittstellen passende Treibersoftware zu liefern. Weiter werden verschiedene abstrakte Dienst definiert, unter anderem ein Kommunikationsmodell, welches keine Vorgaben bezüglich der verwendeten Protokolle und Topologie macht. Der Kontrollfluss wird dabei mittels des *Observer Pattern* realisiert. [3]

3.3 OROCOS

Die *Open Robot Control Software* (OROCOS) wurde an der KU Löwen im Rahmen eines EU-Projekts entwickelt. Zentrales Merkmal ist die klare Trennung von Struktur und Funktionalität. Dazu wird als Grundlage ein generischer Roboter definiert. Die einzelnen Komponenten stellen anschließend, wie in einem Netzwerk, die Verbindung her. Die dabei verwendeten Schnittstellen werden mittels einer Komponentenbeschreibungssprache separat beschrieben. Die Definition umfasst unter anderem die angebotenen Dienste und des verwendeten Austauschformat. Zur Implementierung dieser Mechanismen stützt sich OROCOS auf etablierte Standards wie CORBA. Globale Daten werden über eine Art *Blackboard* gelesen und modifiziert. [17]

Dieser Ansatz macht die Plattform höchst flexibel und bietet auch die Möglichkeit verteilter Anwendungen. Andere Plattformen, wie zum Beispiel YARP, können mittels Erweiterungen integriert werden. Außerdem besteht die Möglichkeit aus der Beischreibung einer Komponente Quellcode zu generieren. Weiter existiert eine große Zahl an Hilfsbibliotheken für verschiedene Probleme der Kinematik und Dynamik.

3.4 Unterschiede und Gemeinsamkeiten

Alle untersuchten Plattformen weisen eine modulare Struktur auf. Die Entwickler von YARP schlussfolgern jedoch, dass Modularität ist nicht ausreichend sei, da gängige Architekturen in der Regel untereinander nicht kompatibel sind. [3] Deshalb wird in allen Beispielen versucht die spezialisierten Hardwarekomponenten mittels möglichst allgemeingültiger Schnittstellen weitestgehend zu abstrahieren. Die Unterschiede bestehen eher darin, wie weitgehend diese Abstraktion ist. An dieser Stelle hat OROCOS den flexibelsten Ansatz, der jedoch für kleinere Anwendungen auf Grund der Komplexität ungeeignet sein kann. Darüber hinaus bieten alle Plattformen verschiedene Dienste, Algorithmen und Datenstrukturen an. Dazu gehört eine zentrale Verwaltung globaler Daten und Kommunikationsschnittstellen für verteilte Prozesse.

4 Die FUmanoid Plattform

4.1 Hardware

Die Roboter der FUmanoids sind zu großen Teilen eine Eigenkonstruktion des Teams. Sie sind knapp 60 cm groß und wiegen etwa 4,5 kg. In allen Maßen orientieren sie sich an den Vorgaben in der Humanoid-League des RoboCups. [12]

Hauptplatine	Odroid-X2
CPU	1,7 GHz Quadcore ARM Cortex-A9
Speicher	2 GB RAM
Anschlüsse	Ethernet & 4x USB

Neben den in der Tabelle aufgeführten Eckdaten des Hauptcomputers besitzt der Roboter weitere Platinen. So existiert eine Platine zur Regulation der elektrischen Spannung, welche im Fehlerfall eine Notabschaltung durchführt. Weiter ist an den Hauptcomputer das sogenannte *Motorboard* angeschlossen. Es ist in der Lage mit den Aktuatoren der beiden Beine und des Oberkörpers parallel zu kommunizieren. Die Servomotoren der FUmanoids stammen von der koreanischen Firma Robotis Inc. Sie besitzen jeweils einen eingebauten Mikrocontroller, welcher Funktionen zum Setzen der Motorposition, Geschwindigkeit und anderer Parameter bereitstellt. Er liefert aber auch Informationen über die aktuelle Lage, Belastung und Geschwindigkeit zurück.

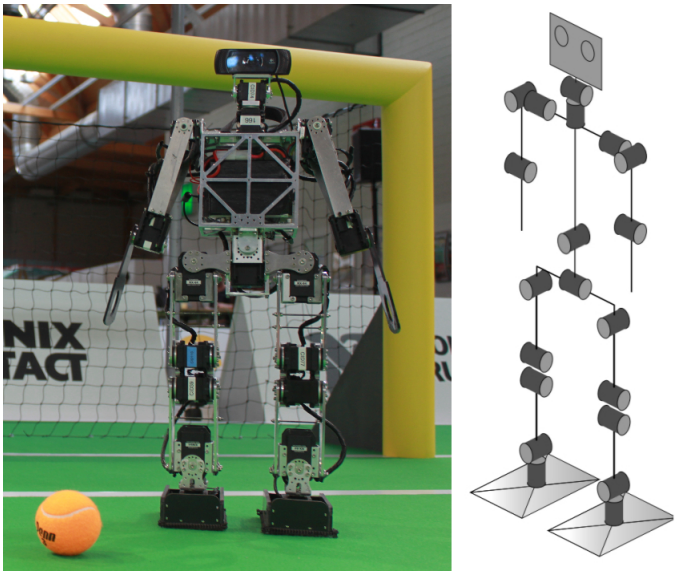

Abbildung 5: Roboter der FUmanoids und seine Kinematik

Während im Oberkörper des Roboters die schwächeren Motoren vom Typ RX-28 verbaut sind, befinden sich im Antriebsteil die stärkeren Motoren des Typs RX-64, welche sich vor allem durch ein circa doppelt so großes maximales Drehmoment auszeichnen. [9] Insgesamt besitzt der Roboter damit 20 Freiheitsgrade, fünf pro Bein, zwei am Oberkörper, drei pro Arm und zwei am Kopf.

Als optischer Sensor besitzt der FUmanoids-Roboter eine Kamera vom Typ *Logitech HD Pro Webcam C910* mit einer Auflösung von 640×480 Pixeln (VGA) und einer maximalen Frequenz von 30 Bildern pro Sekunde. [20] Der Öffnungswinkel von 70° erfordert die ständige Drehung des Kopfes, um einen größeren Ausschnitt der Umwelt wahrzunehmen.

Weiter ist eine inertiale Messeinheit (IMU) auf dem *Motorboard* angebracht. Sie verfügt über ein Gyroskop, Magnetometer und Accelerometer, welche jeweils die Drehung um die drei Achsen messen. Da die Werte teilweise stark verrauscht sind, kommt hier ein Kalman-Filters zum Einsatz. [5]

4.2 Software

4.2.1 Module-Framework

Die FUmanoid-Software basiert auf einer modularen Architektur, welche in der Programmiersprache C++ implementiert ist. Entwickelt wurde dieses Modul-Framework vom *Nao Team Humboldt* (NaoTH), welches in der Standard Platform League teilnimmt. Mit ihnen pflegen die FUmanoids eine Kooperation unter dem Namen *Berlin United*. Das gleichnamige Framework wird im nächsten Abschnitt beschrieben.

Den Kern der Struktur bildet ein *Blackboard*. Die Daten werden darauf in Form einer sogenannten *Representation* abgelegt. Module sind hingegen ausführbare Einheiten, welche Daten über das *Blackboard* austauschen. Sie werden vom sogenannten *Module Manager* verwaltet. Die einzelnen Module sollen untereinander keine Abhängigkeiten haben, so dass sie flexibel austauschbar sind. Sie besitzen entweder lesenden oder auch schreibenden Zugriff auf einzelne Repräsentationen. Die Funktionalität des Frameworks wird hinter C++ Makros versteckt, so dass die Übersichtlichkeit gewahrt bleibt und ein leichter Einstieg möglich ist [15].

4.2.2 Berlin United Framework

Das *Berlin United* Framework umfasst zahlreiche Klassen für verschiedene Dienste und das bereits beschriebene Module-Framework. Alle Bestandteile sind plattformu-

nabhängig gestaltet, so dass es auch bei anderen Projekten des Lehrstuhl zur Anwendung kommt. Realisiert ist unter anderem ein generisches Kommunikationsmodell, Werkzeuge zur Verwaltung von Konfigurationsparametern und für das Debugging, sowie Schnittstellen für das Erstellen von Logdateien. [23]

Um strukturierte Daten zu serialisieren und zu speichern wird die Software *Protocol Buffers* (Protobuf) der Firma Google verwendet. Mit dem *Protocol Buffer* lassen sich Nachrichten definieren, die aus einer Menge von typisierten Datenfeldern bestehen. Aus der Definition lässt sich anschließend automatisiert die dazugehörige Klassenstruktur mitsamt der nötigen Zugriffsmethoden generieren. [8]

4.2.3 Die FUmanoid-Software

Auf Grundlage dieser Struktur realisiert die FUmanoid-Software die konkrete Anwendung für den eigenen Roboter. Auch das *Nao Team Humboldt* setzt an dieser Stelle mit einer eigenen Lösung an, die auf den NAO-Roboter zugeschnitten ist. Interoperabilität ist deshalb trotz des gleichen Unterbaus nicht gegeben.

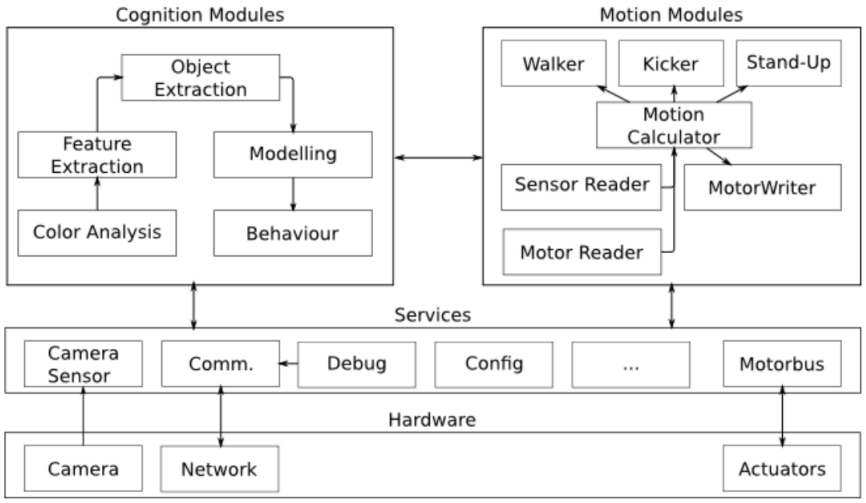

Abbildung 6: Architektur der FUmanoid-Software [20]

Im Detail gliedert sich die FUmanoid-Software auf oberer Ebene in zwei Blöcke, in denen wiederum durch verschiedene Module in einer vordefinierten Reihenfolge Teilprobleme gelöst werden. Der in der Abbildung mit *Cognition* bezeichnete Strang ist dafür verantwortlich, dass der Roboter sein Umwelt wahrnimmt. Er wird mit jedem neuen Bild der Kamera durchlaufen, was einer ungefähren Frequenz von 30 Hz entspricht. [20]

Das Verhalten ist an der obersten Stelle dieses Strangs definiert. Es wird mit Hilfe der domänenspezifischen *Extensible Agent Behavior Specification Language* (XABSL) modelliert. In ihr werden eine Menge von Optionen in einem gerichteten azyklischen Graphen angeordnet. Die einzelnen Optionen sind wiederum als endlicher Automat definiert, in deren Zuständen *Entscheidungen* und *Aktionen* stattfinden. Entscheidungen werden auf Grundlage von *Symbolen* gefällt, die durch den Roboter bereit gestellt werden und Informationen über die Umwelt enthalten. [10]

Die Bewegung wird durch den zweiten oberen Block des Modells kontrolliert und gesteuert, die sogenannte *Motion*. Hier existieren Module, welche unter anderem einen stabilen Gang und einen Torschuss realisieren. Weiter ist das Abspielen statischer Bewegungsabläufe implementiert. Der Strang der *Motion* wird mit einer statischen Frequenz von 100 Hz ausgeführt. [20]

Die Kommunikation zwischen den beiden Blöcken wird über Ereignisse realisiert. Dieses Muster wurde bereits mit dem *Observer Pattern* vorgestellt. Konkrete Bewegungen werden in der Regel durch die *Aktionen* im Verhalten angefordert. Der sogenannte *MotionExecutur* sorgt anschließend nach dem Muster eines endlichen Automaten für die Ausführung der verschiedenen Bewegungen. Damit folgt die Software im Wesentlichen dem Paradigma *Sense-Plan-Act*. [23]

In der unteren Ebene des Modells werden verschiedene Dienste angeboten, welche von den Modulen der oberen Schichten genutzt werden können. Insbesondere wird hier der Zugriff auf die Hardware, also die Kamera, das Netzwerk und die Aktuatorik, realisiert. Die Module sind so konzipiert, dass sie verschiedene Hardware unterstützen können. Deshalb sind sie zum Teil bereits in das Framework *Berlin United* integriert. Jedoch trifft dies nicht auf alle Dienste zu, wie zum Beispiel die Steuerung der Aktuatorik und die Kommunikation mit der Sensorik. Zur Zeit besteht eine vollständige Unterstützung für die Hardware der verschiedenen Generationen der FUmanoids, sowie die Simulatoren *Sim** und *SimSpark*. [20] Die Unterstützung des NAO-Roboters ist Gegenstand dieser Arbeit.

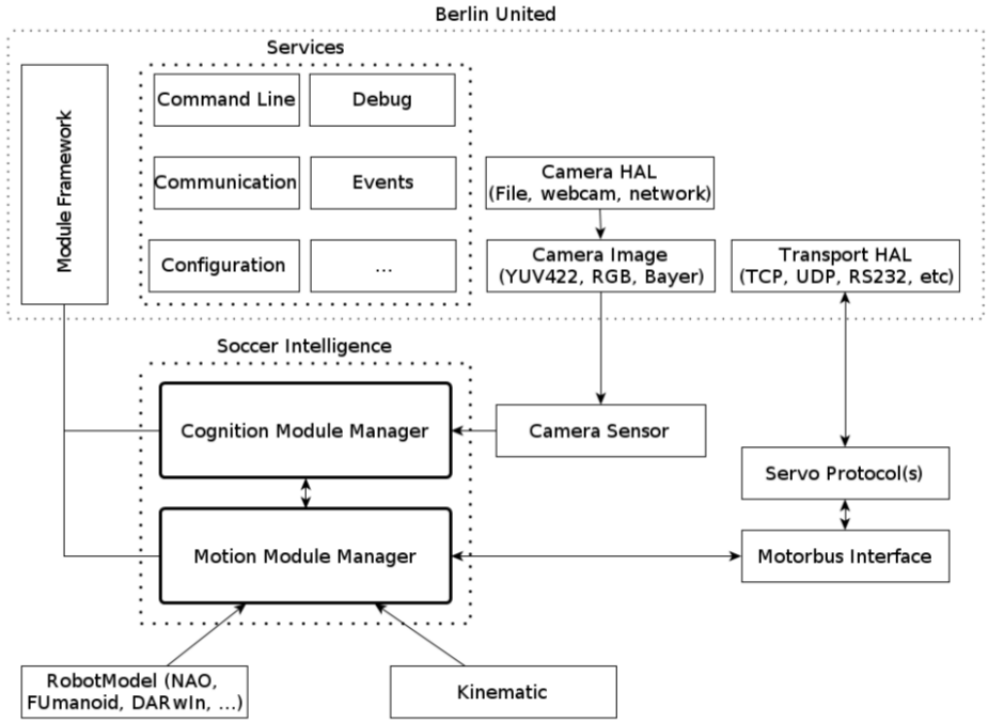

Abbildung 7: Aufteilung der Komponenten in FUmanoid [23]

Auf dem Roboter wird als Betriebssystem ein angepasstes Linux der *Linaro* Distribution verwendet. Weiter existieren einige Hilfsprogramme und Skripte, wobei insbesondere die Software *FUremote* zu nennen ist. In ihr sind verschiedene Werkzeuge für das Debugging, die Konfiguration und Simulation zusammengefasst. Weiter lassen sich im integrierten *Motion Editor* statische Bewegungsabläufe erzeugen, die hinterher auf dem Roboter ausgeführt werden können (siehe Abschnitt 4.4). *FUremote* ist in der Lage sich mit dem Roboter über ein Netzwerk zu verbinden und kann dann Daten empfangen, visualisieren und versenden. Die Nachrichten werden dabei mittels Googles *Protobuf* definiert. Die Software basiert auf der *Java Eclipse Rich Client Platform* und ist somit auf allen gängigen Betriebssystemen ausführbar. Durch eigene Plugins lässt sie sich flexibel erweitern [7].

5 Der NAO-Roboter

5.1 Hardware

Der NAO-Roboter wird seit dem Jahr 2006 von der französischen Firma Aldebaran Robotics hergestellt. Er ist etwa 58 cm groß und wiegt 5,8 kg. Seit seiner Vorstellung wurde er kontinuierlich verbessert und erweitert, weshalb hier die aktuelle Version mit der Typbezeichnung H25 vorgestellt wird.

Im Kopf es Roboters befindet sich die Hauptplatine, auf der ein *Intel Atom Z530* Prozessor mit einer Taktung von 1.6 GHz verbaut ist. Weiter verfügt das System über 1 GB RAM. [18] Mittels USB ist eine separate Platine angeschlossen, welche im Bereich des Torsos befestigt ist. Diese kommuniziert über zwei serielle Bus-Leitungen mit den Servomotoren im Oberkörper und in den Beinen, wobei einzelnen Gruppen von Motoren zum Teil wiederum über eigene Mikrocontroller angesteuert werden. Für jeden Motor ist in ihnen unter anderem ein Regelkreis realisiert, welcher Abweichungen der Soll-Position kontinuierlich entgegenwirkt. Hierbei kommt ein *PID-Regler* zum Einsatz, der sich aus einem proportionalen, einem integrativen und einem differentialen Anteil zusammensetzt. Der Ist-Wert wird mit ein bis zwei Potentiometern je Servomotor erhoben. [19] [11]

Insgesamt besitzt der NAO-Roboter somit fünf Freiheitsgrade pro Arm und Bein, sowie zwei Freiheitsgrade für den Kopf und jede Hand. Zusätzliche Stabilität erhält er im Gegensatz zu den Robotern der FUmanoids durch die ihn umfassende Verschalung.

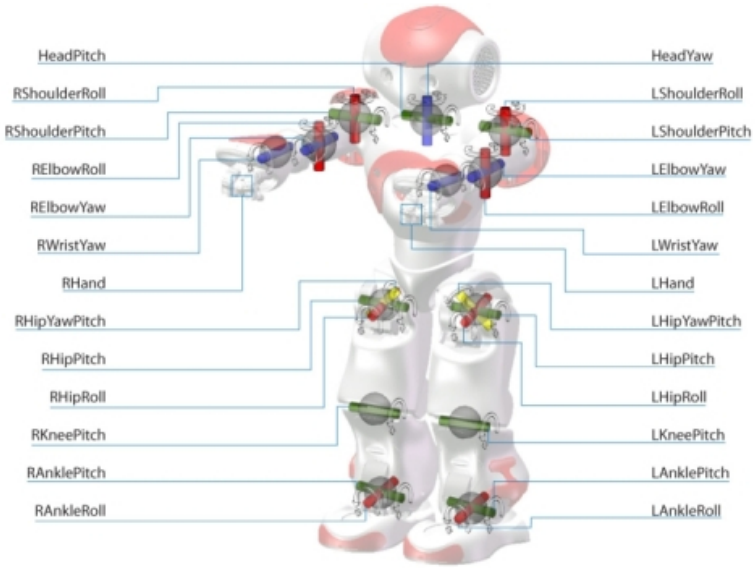

Abbildung 8: Motorik des NAO-Roboters [19]

Im Kopf des NAO-Roboters befinden sich zwei Kameras vom Typ *MT9M114*, welche mit einer maximalen Frequenz von 30 Bildern pro Sekunde und einer maximalen Auflösung von 1,2 Megapixeln arbeiten. Weiter verfügt er über ein Gyroskop, welches Drehungen um die Roll- und Nick-Achse misst, sowie ein dreiachsiges Accelerometer. In den Fußsohlen sind jeweils vier Wägezellen eingelassen, welche die Gewichtskraft bei Berührung des Bodens messen. Außerdem sind an der Vorderseite des Oberkörpers jeweils zwei Sonar-Emitter und -Empfänger verbaut, welche bis zu einer Entfernung von 2,55 Metern Objekte im Raum detektieren können. [18]

5.2 Software

Aldebaran Robotics stellt für den NAO-Roboter ein umfangreiches Paket verschiedener Software zur Verfügung. So ist für den Betrieb des Roboters ein Betriebssystem namens *OpenNAO* verfügbar. Hierbei handelt es sich auch um ein speziell angepasstes Linux, basierend auf der *Gentoo* Distribution, welche alle notwendigen Treiber enthält. [18] Vorinstalliert ist ein Debugger und weitere nützliche Dienstprogramme. Außerdem enthält es die proprietäre NAOqi-Software, welche den Roboter steuert und Zugriff auf die Hardware ermöglicht.

5.2.1 NAOqi

Die NAOqi-Software realisiert einige nützliche Dienste, wie zum Beispiel das Protokollieren aller wichtigen Ereignisse in Logdateien. Hauptsächlich agiert sie jedoch auf dem Roboter in Form eines *Brokers*. Wenn sie gestartet wird liest sie mehrere Konfigurationsdateien ein, aus denen ersichtlich wird, welche Bibliotheken geladen werden sollen. Jede Bibliothek besteht wiederum aus mindestens einem Modul, welches Funktionalität über den Broker anbieten kann. Zusätzlich wird zwischen Modulen unterschieden, welche lokal auf dem Roboter ausgeführt werden und solchen, die mit dem *Broker* über ein Netzwerk kommunizieren. Die Schnittstellen werden in diesem Fall mittels des vorgestellten *Proxy-Pattern* realisiert. Jedoch garantiert nur eine lokale Ausführung den Zugriff in Echtzeit. [19]

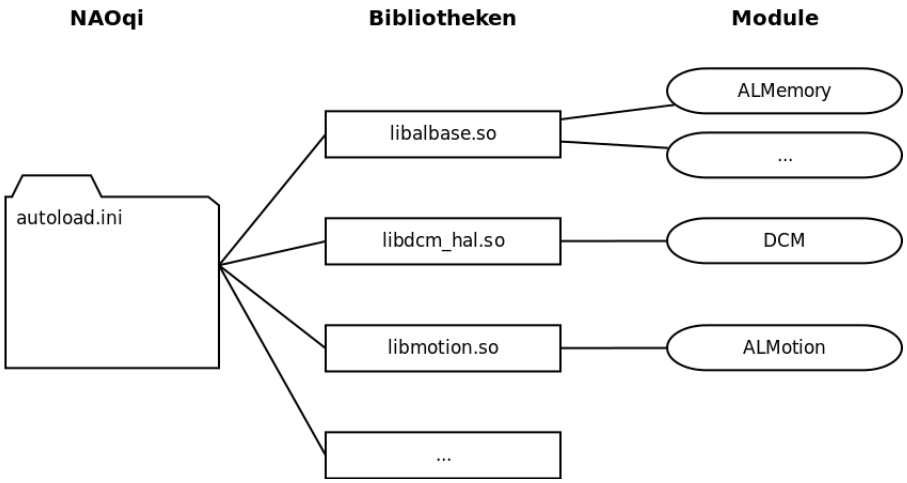

Abbildung 9: Modulstruktur der NAOqi-Software

In der Abbildung ist exemplarisch die Struktur der NAOqi-Software, zusammen mit einigen wichtigen Modulen, dargestellt. Eine weitere Schlüsselrolle spielt der sogenannte *Device Communication Manager* (DCM). Er erlaubt den schreibenden Zugriff auf die Parameter der Hardware. Dies umfasst unter anderem auch die Steuerung der Servomotoren. Das Auslesen der Sensorik wird hingegen mittles des Moduls *ALMemory* realisiert. *ALMemory* dient auch darüber hinaus als zentrales *Blackboard*, das in Form eines assoziativen Containers für verschiedene Datentypen realisiert ist und nebenläufigen Zugriff erlaubt. Hier werden alle Werte abgelegt, welche von den Sensoren empfangen werden.

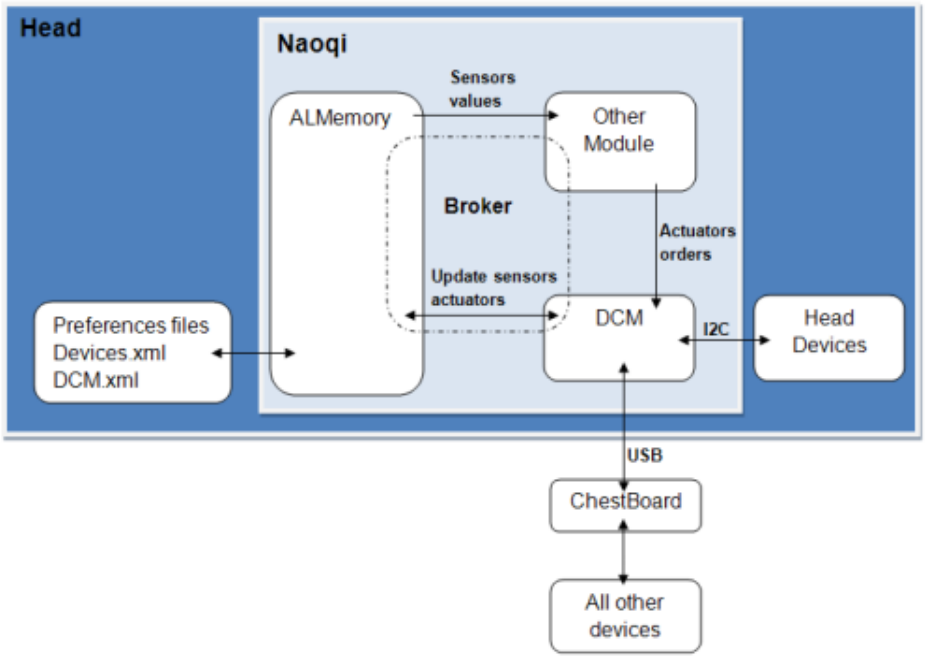

Abbildung 10: Zugriff auf die Hardware des NAO-Roboters [19]

Darüber hinaus existieren weitere Bibliotheken, die zahlreiche Funktionen bereitstellen. Zu nennen ist hier das Modul *ALMotion*. Es umfasst verschiedene Algorithmen, welche bei der Implementierung der Kinematik des Roboters behilflich sind. Dazu gehören unter anderem verschiedene Funktionen zur Realisierung eines stabilen Gangs. Weiter ist eine *inverse Kinematik* implementiert, deren Ausführung in einer Frequenz von 50 Hz erfolgt. [19] Jedoch sind all diese Algorithmen spezifisch auf den NAO-Roboter zugeschnitten.

Um die Programmierung des Roboters zu erleichtern, hat Aldebaran Robotics das sogenannte *NAOqi-Framework* veröffentlicht. Es enthält die nötigen Bibliotheken, um Module oder Remote-Anwendungen für die NAOqi-Software zu schreiben. Es unterstützt die Programmiersprachen Python, Java, C#, VisualBasic, Matlab und Urbiscript. Die volle Funktionalität ist aber bisher nur in der Programmiersprache C++ nutzbar.

5.2.2 Weitere hilfreiche Anwendungen

Um das Erstellen von eigenen Programmen und Bibliotheken für den NAO-Roboter zu vereinfachen, stellt Aldebaran Robotics eine eigene Software unter dem Namen *qi-Build* zur Verfügung. Diese basiert auf dem etablierten Programmierwerkzeug *CMake* und dient zur automatisierten Verwaltung der Abhängigkeiten beim Bauen der Software.

Weiter wird eine Software namens *Choregraphe* angeboten, welche für alle gängigen Betriebssysteme verfügbar ist. Mit ihr können unter anderem statische Bewegungsabläufe und fertige Verhaltensroutinen für den Roboter zusammengestellt und anschließend abgespielt werden. *Choregraphe* kommuniziert dabei direkt mit der NAOqi-Software.

6 Umsetzung der Portierung

Im diesem Abschnitt werden die Entwicklungsschritte beschrieben, die bei der Portierung der Software durchgeführt worden sind. Das Testen erfolgte auf einem NAO-Roboter vom Typ *H25 Version 4*.

6.1 Cross-Development

Software-Entwicklung für eingebettete Systeme erfolgt in der Regel in Form eines sogenannten *Cross-Developments*. Dabei befindet sich die Entwicklungsumgebung mit dem Compiler auf einem normalen PC und nur das ausführbare Kompilat wird auf das Zielsystem geladen. [1]

Diese Form der Software-Entwicklung kommt auch bei den FUmanoids zum Einsatz, weshalb die Verwendung des von Aldebaran Robotics angebotenen Tools *qiBuild* ausgeschlagen wurde. Um eine lauffähige Toolchain für den NAO-Roboter zu schaffen, wurde zuerst ein sogenannter Cross-Compiler erstellt, der ein ausführbares Programm für die x86-Architektur generiert. Hierbei fiel die Wahl auf die GNU Compiler Collection in der Version 4.7, die auch den aktuellen Sprachstandard C++11 unterstützt, der bei den FUmanoids zum Einsatz kommt. Die eigentliche Erstellung der Cross-Toolchain wurde mit Hilfe der Software Crosstool-NG erledigt. Im Anschluss erfolgte die Anpassung der Konfigurationsskripte in der Programmiersprache LUA, mit denen die Software *Premake* das automatische Bauen der FUmanoid-Software bewerkstelligt. Mit Hilfe der Software *Scratchbox*, die eine virtuelle Umgebung für das Zielsystem bereitstellt, wurden alle benötigten Bibliotheken übersetzt und auf den Roboter übertragen.

Die FUmanoid-Software war auf dem NAO-Roboter nun lauffähig. Das Bauen der Software erfolgt außerdem auf dem selben, komfortablen Weg, wie dies für die Roboter der FUmanoids der Fall ist.

6.2 Integration der Kamera

Um die Bilder der Kamera des Roboters lesen zu können, nutzt die FUmanoid-Software die Schnittstelle *Video4Linux*. Diese ist auch auf dem NAO-Roboter verfügbar, dessen Kamera auch das passende YUV422-Bildformat liefert. [18] In der ursprünglichen Konfiguration führte aber das verwendete *Interlacing* zu schwer nachvollziehbaren Abstürzen auf Ebene des Kernels. Die Nutzung eines *progressiven* Formats schaffte hierbei Abhilfe. Insgesamt waren an dieser Stelle somit nur wenige Anpassungen nötig.

Allerdings ist die FUmanoid-Software in der aktuellen Version nur dafür ausgelegt mit dem Bild einer Kamera zu arbeiten, während der NAO-Roboter über zwei übereinander liegende Kameras im Kopf verfügt. Der Abstand zwischen beiden Kameras beträgt circa 4 cm, wobei die untere Kamera um etwa 40° geneigt ist. Der Öffnungswinkel der Kameras beträgt allerdings nur $34,8°$, so dass Binokularsehen nicht möglich ist. Trotzdem würde die Nutzung beider Kameras die nötige Drehung um den Nick-Winkel reduzieren, wenn nahe und ferne Objekte abwechselnd fokussiert werden sollen. Viele Teams der SPL nutzen dennoch nur eine Kamera, da die Bilder nicht simultan geliefert werden können. [21] Es wurde sich deshalb dazu entschieden erst einmal nur das Bild der weiter nach unten gerichteten Kamera zu verwenden, da diese auch den wichtigen Bereich vor den Füßen des Roboters abdecken kann.

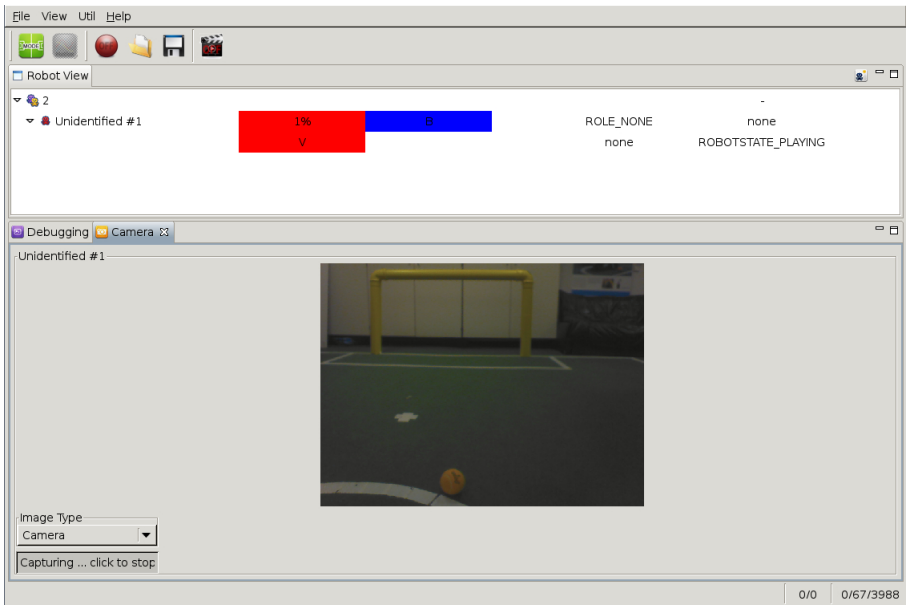

Abbildung 11: Kamerabild des NAO-Roboters in FUremote

6.3 Aktuatorik und Sensorik

Die FUmanoid-Software bündelte bisher alle Zugriffe auf die Aktuatoren und Sensoren, mit Ausnahme der Kamera, zentral in der Schnittstelle *MotorBus*. Dies resultierte aus dem Umstand, dass in den früheren Versionen der FUmanoids die komplette Peripherie über eine Datenleitung mit der Hauptplatine verbunden war (nachzulesen in [22]). Weiter waren spezialisierte Implementierungen im *MotorBus* durchaus vorgesehen. Im Rahmen der Portierung stellte sich aber die Fokussierung auf die Dynamixel-Motoren als problematisch heraus.

Wie bereits beschrieben, unterscheiden sich die Anforderungen für die Steuerung der beiden Roboter erheblich. Auf den Robotern der FUmanoids erfolgt die Kommunikation mit den Servomotoren auf direktem Weg. Die Positionswerte besitzen eine Auflösung von 300° und werden in einem 10 Bit breiten Wert kodiert. Dieser Wert wird an das *Motorboard* gesendet, beziehungsweise von diesem empfangen. Auf dem NAO-Roboter werden hingegen die einzelnen Werte über die Schnittstellen der proprietären NAOqi-Software gesendet und empfangen. Die Positionen der einzelnen Gelenke werden dabei in Radiant angegeben. Da eine zukünftige Portierung auch für andere Roboter, wie zum Beispiel den *DARwIn-OP* der Firma Robotis, angedacht ist, können an dieser Stelle aber auch beliebige andere Werte und Wertebereiche auftreten.

Aus diesem Grund bestand ein zentraler Teil der Portierungsarbeit in der Konzeption und Implementierung neuer Schnittstellen zur Steuerung der Aktuatorik und dem Zugriff auf die Sensorik für unterschiedliche Robotermodelle. Die folgenden Abschnitte beschreiben erst die allgemeinen Überlegungen zu Anforderungen und Design, anschließend wird die konkrete Implementierung für die beiden Roboter vorgestellt.

6.3.1 Anforderungen und Design

Aus den Vorbedingungen ergaben sich folgende allgemeine Designentscheidungen:

- Die Schnittstellen für den Zugriff auf die Aktuatork und Sensorik werden getrennt, um konkrete Implementierung flexibel austauschen zu können.
- Es werden möglichst generische Datentypen genutzt. Die Angabe einer Position erfolgt in Grad, während die Geschwindigkeit durch Grad pro Sekunde beschrieben wird. Der Vorteil dieser Einheiten liegt vor allem in der guten Lesbarkeit durch Menschen, was einen Vorteil beim Aufspüren von Fehlern mit sich bringt.

- Der Zugriff auf die Schnittstellen erfolgt seriell, um die Komplexität zu senken. Ein nachträgliches Hinzufügen von Nebenläufigkeit soll aber ohne großen Aufwand erfolgen können.
- Möglichst viel Quellcode, der sich bereits im produktiven Einsatz befunden hat, soll wieder verwertet werden. Dies betrifft insbesondere die Schichten, die direkt mit der Hardware oder den Simulatoren reden.
- Jeder Motor ist durch einen Wert des Datentyps *uint8_t* identifizierbar. Abzüglich der zwei Konstanten für einen nicht existierenden Motor und alle Motoren, lassen sich auf diese Weise 253 Motoren adressieren. Die Zahl dürfte auch für die ferne Zukunft mehr als ausreichend sein sollte.
- Aus Gründen der Kompatibilität mit den vorhandenen Modulen wird der Kontrollfluss nicht umgedreht. Die Werte der Sensoren werden also weiterhin aktiv erfragt. Eine Umkehrung, wie sie mit dem *Observer Pattern* vorgestellt wurde, wäre eher von Vorteil für die Integration des NAO-Roboters gewesen, da die Werte aller Sensoren in einem festen Takt durch die NAOqi-Software in den Speicher geschrieben werden.

Der resultierende Entwurf ist in der folgenden Abbildung dargestellt. Die Klassen *Actuators* und *IMUSensor* stehen dabei als Schnittstelle für andere Module zur Verfügung. Sie besitzen als Member-Variablen die abstrakten Datentypen *ActuatorInterface* und *IMUInterface*. Für sie existieren je Robotermodell konkrete Implementierungen, welche die eigentliche Funktionalität realisieren. Die dazu notwendigen Kommunikationsprotokolle sind auf der untersten Ebene des Entwurf wiederum in verschiedenen Klassen definiert.

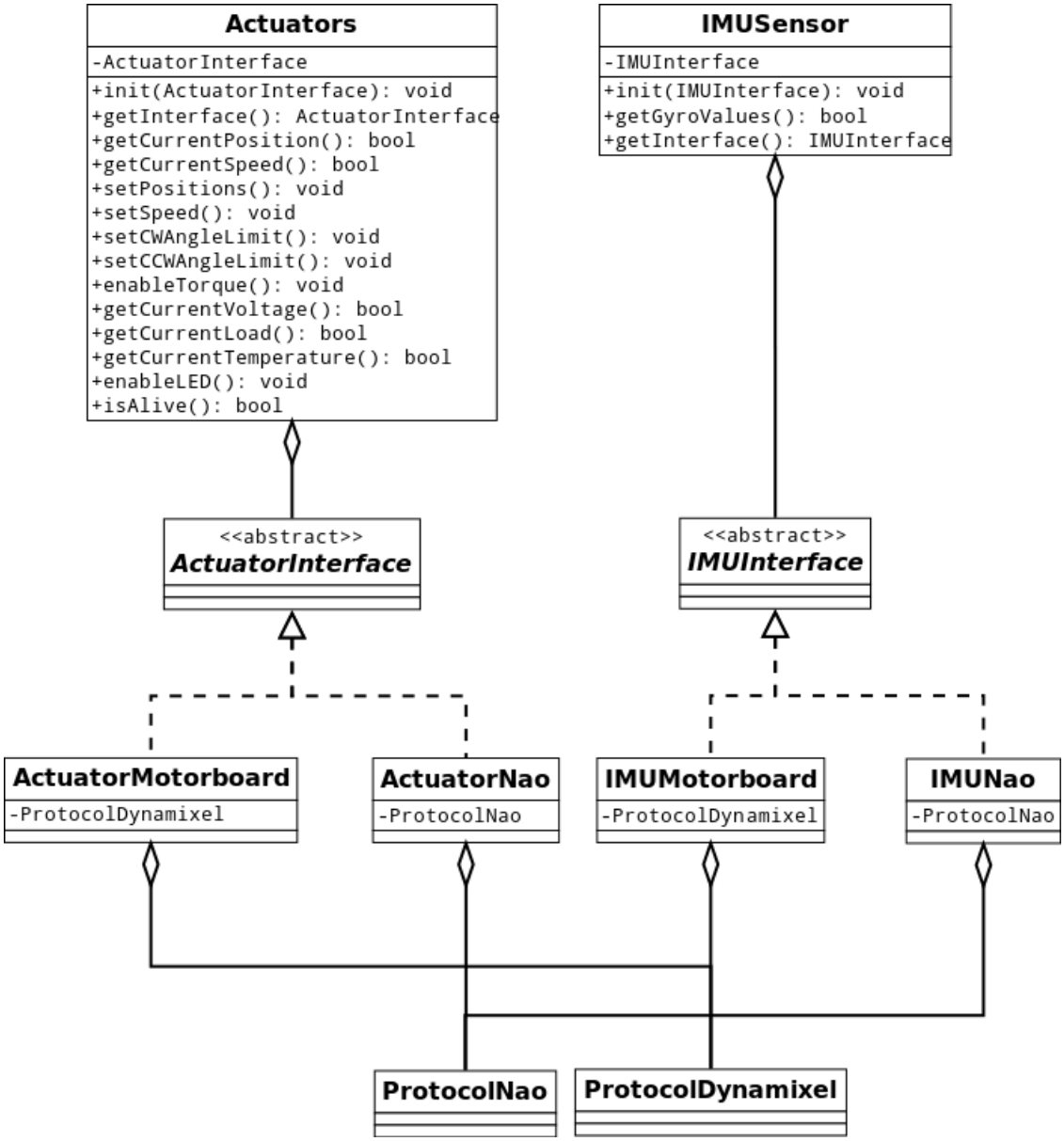

Abbildung 12: Entwurf der Schnittstelle für Aktuatorik und Sensorik

Das beschriebene Design nutzt als Entwurfsmuster das *Strategy-Pattern*. Als weiteres Entwurfsmuster kommt das *Singleton-Pattern* zum Einsatz, um sicher zu stellen, dass von jeder Schnittstelle nur genau ein Objekt existieren kann.

Die beiden Schnittstellen stellen möglichst allgemeingültige Methoden zum Lesen und Setzen von verschiedenen Parametern zur Verfügung. Um die Übersichtlichkeit zu erhalten, sind in der Abbildung nur die wichtigsten Funktionen ohne ihren genauen Signaturen aufgeführt. Es ist jedoch anzumerken, dass in der Klasse *Actuators* ein Großteil der Funktionalität in doppelter Ausführung existiert. Zum einen ist es möglich Werte für einzelne Aktuatoren zu Setzen und Lesen, aber es besteht auch die Möglichkeit dies für Gruppen von Motoren zu tun. Dies wird durch die Klas-

se *ActuatorValues* realisiert, welche als assoziativer Container dient. Sie speichert Werte mitsamt der dazugehörigen ID des Aktuators und einem Zeitstempel.

Die Parameter der Methoden sind wenn möglich als konkrete Einheiten definiert, also zum Beispiel *Degree* für Grad und *RPM* für Umdrehungen pro Minute. Die Definition dieser Einheiten erfolgt mit Hilfe der Bibliothek *Boost*. Die Einheiten werden während des Kompilierens in elementare Datentypen überführt, so dass bei Ausführung des Programms kein Mehraufwand besteht.

Trotzdem ist es an einigen Stellen notwendig geworden die Kapselung zu brechen und direkt auf die konkrete Implementierung zuzugreifen, da Funktionalität existiert, die sich nicht auf alle Roboter verallgemeinern lässt. Dies trifft zum Beispiel auf einige Funktionen zu, welche das *Motorboard* der FUmanoids initialisieren.

6.3.2 Umsetzung für die Roboter der FUmanoids

Bei der Umsetzung für die Roboter der FUmanpids konnte eine große Menge an bereits genutztem Code wiederverwendet werden. Dies trifft insbesondere auf die Implementierung des Dynamixel-Protokolls und die dabei verwendeten Klassen zur Ansteuerung der seriellen RS-232 Schnittstelle zu. Aufgabe war es somit nur noch die angesprochenen Einheiten wenn nötig zu konvertieren und an die darunter liegenden Schichten weiter zureichen. Im Normalfall umfasst der steuerbare Bereich 300° mit einer Auflösung von 1024 Schritten.

Dabei waren jedoch einige Spezialfälle zu beachten. Zum Beispiel existiert für die Servomotoren ein sogenannter *Wheel-Mode*. In diesem Modus können die Motoren ohne Einschränkung rotieren, wodurch sich einige Parameter des Protokolls in ihrer Bedeutung ändern. [9] Aus diesem Grund wird unter anderem für jeden Motor gespeichert, in welchem Modus er sich befindet.

Anschließend erfolgte die Anpassung aller Module der FUmanoid-Software, welche auf die alte Schnittstelle *MotorBus* zugriffen. Da dies an vielen Stellen in der FUmanoid-Software erfolgte, war dieser Arbeitsschritt sehr umfangreich. Zum Großteil wurden auch die dazugehörigen Berechnungen auf die neuen Einheiten umgestellt, sofern dies nicht schon der Fall war.

Keine Anpassung erfolgte bisher für externe Hilfsprogramme, wie den in die FUremote-Software integrierten *MotionEditor*, da dies sehr umfangreiche Änderungen nötig gemacht hätte. Stattdessen werden die Motorpositionen einfach beim Einlesen und Winkel konvertiert.

6.3.3 Umsetzung für den NAO-Roboter

Auf dem NAO-Roboter ist der Zugriff auf Sensorik und Aktuatorik nur auf indirektem Weg möglich. Deshalb wurde als erster Schritt eine Bibliothek mit einem dazugehörigen Modul implementiert, welche von der NAOqi-Software geladen werden kann. Das Modul implementiert spezielle Callback-Funktionen, welche vom *Device Communication Manager* aufgerufen werden. Dieses Vorgehen wurde eingangs mit dem *Observer Pattern* vorgestellt. Der Aufruf geschieht immer bevor Parameter an die Hardware gesendet werden oder nachdem neue Sensorwerte verfügbar sind. Auf diese Weise kann das Modul in Echtzeit mit der Hardware des Roboters kommunizieren, wobei die gemessene Frequenz bei diesem Vorgang circa 100 Hertz beträgt.

Das Modul tauscht dann die Daten mit der FUmanoid-Software über einen geteilten Speicherbereich aus. Die Synchronisierung der Prozesse erfolgt mittels einer Semaphore. Daten werden dabei nur ausgetauscht, wenn neue Werte verfügbar sind. Auf Seiten der FUmanoid-Software erfolgt beim Zugriff auf die Daten anschließend nur noch die Konvertierung von Einheiten, also zum Beispiel Radiant in Grad und umgekehrt. Weiter ist es möglich für jeden Aktuator einen *Offset* in der Konfiguration abzulegen, welcher beim Senden der Positionen addiert und beim Empfangen subtrahiert wird.

Das beschriebene Vorgehen hat den Vorteil, dass die FUmanoid-Software sauber von der eigentlichen Steuerung der Hardware getrennt ist. Kommt es zum Beispiel zu einem Fehler in der FUmanoid-Software, schlimmstenfalls einem Absturz, werden entsprechende Gegenmaßnahmen getroffen um einen schweren Sturz des Roboters zu verhindern.

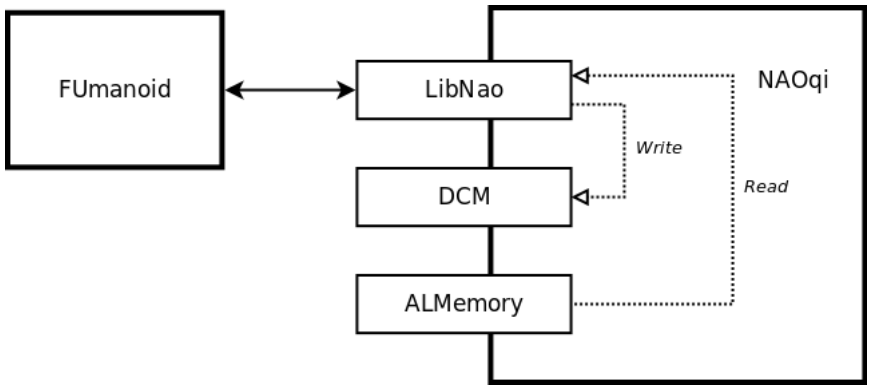

Abbildung 13: Kommunikation der FUmanoid-Software mit NAOqi

Über den *Device Communication Manager* ist es nicht möglich die Geschwindigkeit der einzelnen Servomotoren zu regulieren. Als Alternative besteht die Möglichkeit, die Zeitpunkte anzugeben, an denen bestimmte Positionen eingenommen werden sollen. Die Positionswerte innerhalb dieses Zeitintervalls werden anschließend von einem Interpolationsalgorithmus berechnet. Auf diese Weise kann eine ganze Abfolge von Positionen definiert werden, die dann abgearbeitet wird. Dieses Konzept hat den Vorteil, dass der *Device Communication Manager* mehrere zukünftige Positionen kennen kann und somit eine unterbrechungsfreie Ausführung gewährleistet. Die Bewegungsabfolge wird durch die Interpolation geglättet, was auch die Belastung der Motoren reduziert. [19]

Von Nachteil wäre jedoch, dass die FUmanoid-Software nicht mehr die volle Kontrolle über die genaue Bewegungsabfolge ausüben würde. Deshalb wurde sich gegen dieses Vorgehen entschieden. Jede Position, welche an den *Device Communication Manager* gesendet wird, soll sofort eingenommen werden. Stattdessen kann die Regulation einzelner Bewegungen durch das aufgebrachte Drehmoment erfolgen. Diese Vorgehensweise ist auch bei anderen Teams der *Standard Platform League* gängig. [21]

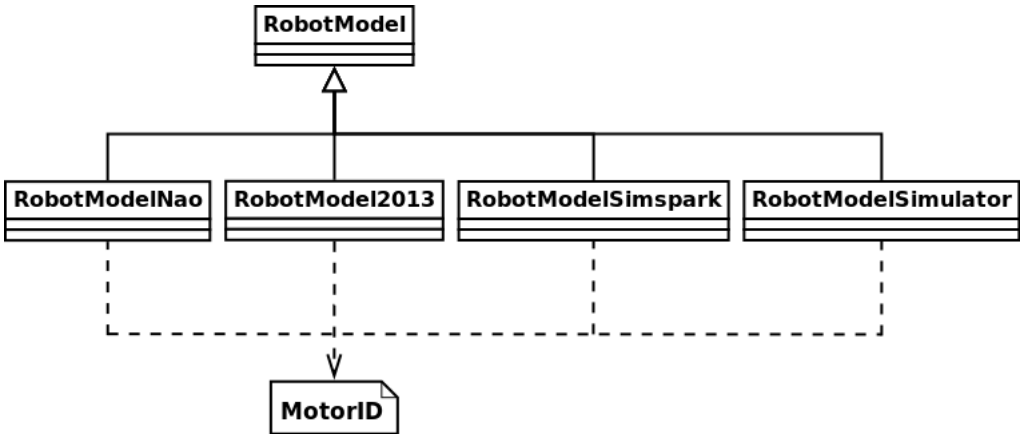

Abbildung 14: RobotModel für die verschiedenen Plattformen

Anschließend war es für den NAO-Roboter notwendig ein eigenes *RobotModel* zu implementieren. Hierbei handelt es sich um eine Klassenstruktur, in welcher der Roboter in Bezug auf die Aktuatorik definiert und initialisiert wird. Zur Integration der neuen Schnittstellen waren auch Änderungen in den bestehenden Implementierungen notwendig. Jedoch operieren diese, im Gegensatz zum NAO-Roboter, alle auf der selben Menge von Aktuatoren. Da sich die Roboter aber in Bezug auf die Motorik kaum unterscheiden, wurde die bestehende Aufzählung der Motoren einfach ergänzt und nicht komplett neu definiert. Mittels C++ Makros wird während

des Kompilierens festgelegt, welche Motoren verfügbar sind. Für Motoren, welche in einen spezifischen Roboter nicht existieren, liefern die Schnittstellen keine Funktionalität oder reagieren mit einer negativen Antwort.

6.4 Statische Bewegungen

Da nun die Ebene der grundlegenden Dienste portiert war, wurde sich dazu entschieden statische Bewegungen für den NAO-Roboter zu implementieren. Durch sie können Basisfunktionalitäten realisiert werden, wie zum Beispiel das Aufstehen nach einem Sturz. Dazu musste evaluiert werden, ob die vorhandenen Komponenten zum Abspielen statischer Bewegungsabläufe grundsätzlich für den NAO-Roboter in Frage kommen.

Das entsprechende Modul in der in der FUmanoid-Software schien dafür zunächst geeignet, da es nur die adjazenten *Keyframes* interpoliert und an die Motoren weiterleitet. Die nötigen Werte werden dazu aus Nachrichten im Protobuf-Format gelesen. Dort sind die Positionen der Motoren nach wie vor als 10 Bit breiter Wert im Dynamixel-Format gespeichert. Die Nachrichten zu konvertieren stellte aber eine mögliche Option dar. Im Anschluss wurde, der in FUremote integrierte, *Motion Editor* betrachtet. Um ihn anzupassen bedarf es zunächst dem Erstellen eines sogenannten *Robot Description File* (beschrieben in [6] und [7]), was auch für den NAO-Roboter realisierbar ist. Als weitaus problematischer stellte sich jedoch auch hier die Fokussierung auf die Dynamixel-Motoren heraus, deren Wertebereich als Grundlage für einen Großteil der Funktionalität dient. Obwohl eine Portierung an dieser Stelle durchaus wünschenswert ist, wurde sich gegen eine Realisierung dieses Vorhabens entschieden, da der nötige Aufwand einem sichtbaren Fortschritt bezüglich der Spielfähigkeit des NAO-Roboters nicht dienlich ist.

Stattdessen wurden vergleichbare Implementierungen durch andere Mannschaften der Standard Platform League betrachtet. Die Wahl fiel dabei auf die naheliegende Möglichkeit, die entsprechende Implementierung des *NAO Team Humboldt* zu nutzen. Dabei werden fest kodierte Bewegungsabläufe aus speziellen Datei geladen, wozu ein geeigneter Parser verwendet wird. Anschließend erfolgt ein Abspielen der definierten Werte mit nach dem Keyframing-Verfahren. Ausschlaggebend war darüber hinaus, dass für das Erzeugen und Bearbeiten der Bewegungsabläufe ein in Java implementierter Editor verfügbar ist.

7 Fazit und Ausblick

Im Rahmen dieser Arbeit wurde die Kernfunktionalität der FUmanoid-Software auf den NAO-Roboter portiert und getestet. Weiter wurden die Infrastruktur der Software so angepasst, das andere Roboterplattformen leichter zu integrieren sind. Dazu wurden generische Schnittstellen geschaffen und für beide Roboter implementiert. Alle vorhandenen Module wurden an diese Schnittstellen angepasst.

Da die Portierung nahezu alle Teilbereiche der FUmanoid-Software tangiert, konnte allein auf Grund der schieren Masse an Funktionalität nur ein kleiner Teil der Module auf höherer Ebene evaluiert und angepasst werden. In der näheren Zukunft ist aber auch die Umsetzung weiterer Aspekte der Portierung geplant. Neben den statischen Bewegungen, sind vor allem die dynamischen Bewegungen von Interesse, da sich nur so elementare Fähigkeiten für das Fußballspiel realisieren lassen. Zu nennen ist hier ein stabiler Gang. Die dafür notwendigen kinematischen Grundlagen sind Gegenstand laufender Arbeit.

Der zweite entscheidende Strang der FUmanoid-Software, die *Cognition*, stand bisher nicht im Fokus. Dennoch ist auch sie von entschiedener Bedeutung, um dem Roboter die Wahrnehmung der Umwelt zu ermöglichen. Hier muss eine hinreichend gute Anpassung und Kalibrierung der vorhandenen Vision erfolgen. Darauf aufbauend ist zu klären, ob die unterschiedlichen Öffnungswinkel der Kameras mit der existierenden Blickfeldkontrolle vereinbar sind.

8 Literatur

[1] Uwe Brinkschulte and Theo Ungerer. *Mikrocontroller und Microprozessoren, 3. Auflage*. Springer Media, 2010.

[2] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-orientierte Software-Architektur*. Addison-Wesley, 2000.

[3] Paul Fitzpatrick, Giorgio Metta, and Lorenzo Natale. Towards long-lived robot genes. Italian Institute of Technology, 2007. Online verfügbar: `http://wiki.icub.org/yarp/media/humantech07towards.pdf`.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1997.

[5] Simon Gene Gottlieb. Erweiterter Kalman-Filter zur Orientierungsabschätzung von humanoiden Robotern. Bachelor Arbeit, Freie Universität Berlin, 2013. Online verfügbar: `http://www.inf.fu-berlin.de/inst/ag-ki/rojas_home/documents/Betreute_Arbeiten/Bachelor-Gottlieb.pdf`.

[6] Steffen Heinrich. Development of a Multi-Level Sensor Emulator for Humanoid Robots. Diplomarbeit, Freie Universität Berlin, 2012. Online verfügbar: `https://maserati.mi.fu-berlin.de/fumanoids/wp-content/papercite-data/pdf/heinrich2012.pdf`.

[7] Simon Philipp Hohberg. Interactive Key Frame Motion Editor for Humanoid Robots. Bachelor Arbeit, Freie Universität Berlin, 2012. Online verfügbar: `https://maserati.mi.fu-berlin.de/fumanoids/wp-content/papercite-data/pdf/hohberg2012.pdf`.

[8] Google Incorporation. Protocol Buffers. Technical report, 2012. Nur online verfügbar: `https://developers.google.com/protocol-buffers/`.

[9] Robotis Incorporation. RX Series Manual. Technical report, 2010. Nur online verfügbar: `http://support.robotis.com/en/product/dynamixel/dxl_rx_main.htm`.

[10] Rico Jonschkowski. Verhaltenssteuerung für autonome humanoide Fußballroboter mit XABSL. Bachelor Arbeit, Freie Universität Berlin, 2010. Online verfügbar: `https://maserati.mi.fu-berlin.de/fumanoids/wp-content/papercite-data/pdf/jonschkowski2010.pdf`.

[11] Johannes Kulick. Ein stabiler Gang für humanoide Fußball spielende Roboter. Master's thesis, Freie Universität Berlin, 2011. Online ver-

fügbar: https://maserati.mi.fu-berlin.de/fumanoids/wp-content/papercite-data/pdf/kulick2011.pdf.

[12] RoboCup Humanoid League. Rules, 2013. Online verfügbar: http://www.tzi.de/humanoid/pub/Website/Downloads/HumanoidLeagueRules2013-05-28.pdf.

[13] RoboCup Standard Platform League League. Rule Book, 2013. Online verfügbar: http://www.tzi.de/spl/pub/Website/Downloads/Rules2013.pdf.

[14] A. K. Mackworth. On seeing robots. Vancouver, BC, Canada, 1993.

[15] Heinrich Mellmann, Yuan Xu, Thomas Krause, and Florian Holzhauer. NaoTH Software Architecture for an Autonomous Agent. *Proceedings of the International Workshop on Standards and Common Platforms for Robotics (SCPR 2010)*, Darmstadt, November 2010.

[16] H. Penny Nii. The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures. *AI Magazine*, 7(2), 1986.

[17] Kristian Regenstein. *Modulare, verteilte Hardware-Software-Architektur für humanoide Roboter*. PhD thesis, Karlsruher Institut für Technologie, 2010.

[18] Aldebaran Robotics. NAO H25 Datasheet. Technical report, 2011.

[19] Aldebaran Robotics. NAO software 1.14.5. Technical report, 2013. Nur online verfügbar: https://community.aldebaran-robotics.com/doc/1-14/.

[20] Raul Rojas, Daniel Seifert, et al. Berlin United - FUmanoids Team Description Paper 2013. Technical report, Freie Universität Berlin, 2013.

[21] T. Röfer et al. B-Human Team Report 2011. Technical report, Universität Bremen, 2011.

[22] Daniel Seifert. Portierung der FUmanoid-Software, Studienarbeit. Studienarbeit, Freie Universität Berlin, 2009. Online verfügbar: https://maserati.mi.fu-berlin.de/fumanoids/wp-content/papercite-data/pdf/seifert2009.pdf.

[23] Daniel Seifert and Raúl Rojas. FUmanoids Code Release 2012. Freie Universität Berlin, Dezember 2012. Nur online verfügbar: http://www.fumanoids.de/publications/coderelease.

[24] Richard T. Vaughan, Brian P. Gerkey, and Andrew Howard. On device abstractions for portable, reusable robot code. *Proceedings of the IEEE / International Conference on Intelligent Robots and Systems*, 2003.

Alle Online-Quellen wurden das letzte Mal am 4. Dezember 2013 abgerufen.

9 Abbildungsverzeichnis