Studienarbeit
Portierung der FUmanoid-*Software*

Daniel Seifert
Immatrikulationsnummer 149 122
dseifert@informatik.hu-berlin.de

17. August 2009

eingereicht am Institut für Informatik
der Freien Universität Berlin

Betreuer:
Tino Naphtali <tino.naphtali@fu-berlin.de>
Hamid Moballegh <moballegh@inf.fu-berlin.de>

Erklärung

Hiermit erkläre ich, daß ich diese Arbeit selbständig erstellt und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Berlin, den 17. August 2009

Daniel Seifert

Inhaltsverzeichnis

1 Einleitung **4**

1.1 Aufbau der Arbeit 4

2 Alte Plattform **4**

2.1 Hardware 4

2.1.1 CPU 5

2.1.2 Motoren 5

2.1.3 Kommunikation 5

2.1.4 Kamera 5

2.2 Software 6

2.2.1 Architektur 6

2.2.2 Bedienung und Kommunikation 8

2.2.3 Problemstellen in alter Software 8

3 Neue Plattform **8**

3.1 Hardware 8

3.2 OpenEmbedded 9

4 Portierung **9**

4.1 Anforderungen 9

4.2 CPU-bedingte Änderungen 10

4.3 Wechsel zu C++ 10

4.4 Aufbau der Software 11

4.5 Motoransteuerung 13

4.6 Kommunikation 16

4.7 Threads 18

4.8 Sonstiges 19

5 Installation und Kompilation **20**

5.1 Erstellung des Linux-Images 20

5.2 Bereitstellung einer Entwicklungsumgebung 20

5.3 Schnellstart und Watchdog 20

6 Zukünftige Änderungen **22**

6.1 Auslagerung Motorsteuerung 22

6.2 Schnellere Reaktion auf Ereignisse 23

7 Fazit **24**

1 Einleitung

In der zweiten Hälfte des 20. Jahrhunderts war Schach ein beliebtes Spiel um Verfahren in der Künstlichen Intelligenz zu erforschen und zu testen. Inzwischen hat sich das Augenmerk aber vom Schach abgewendet, nachdem 1997 der menschliche Weltmeister im Schach von einem Computer (Deep Blue) besiegt wurde. Hinzu kam das sich Schach als zu limitierend erwiesen hat, da alleinig die Spielstrategie und insbesondere auch die verfügbaren Rechenkapazitäten entscheidend waren. Als Alternative wurde vorgeschlagen, Roboter Fußball spielen zu lassen. Hier treffen sich mehrere Themengebiete aus dem Forschungsbereich Robotik und Künstliche Intelligenz, wie zum Beispiel das Laufen, das Erkennen der Umwelt, die Selbstlokalisierung und die Koordination mit weiteren Robotern. Dies alles passiert in einer sich ständig ändernden Umgebung.

Mit dem Ziel, bis zur Mitte des 21. Jahrhunderts ein Roboter-Team zu entwickeln, welches den amtierenden menschlichen Fußballweltmeister besiegen kann, wurde die RoboCup-Initiative gegründet. Das gesteckte Ziel liegt noch in weiter Ferne, aber durch Fortschritte in Technik und Forschung verbessern sich die Roboter jährlich. Bis zum Zusammentreffen mit menschlichen Teams spielen die Roboter nun untereinander und die Forschung gewinnt so Gelegenheit, verschiedene Lösungsansätze gegeneinander antreten zu lassen.

Auch an der Freien Universität Berlin wird im Rahmen des FUmanoid-Projektes an fußballspielenden Robotern der Kid-Size-Klasse (30-60cm Höhe) gearbeitet. 2007 war das erste Jahr des Projektes und die Roboter landeten beim RoboCup 2007 (Atlanta, USA) auf Anhieb auf dem dritten Platz. Im Folgejahr zeichnete sich aber ab, daß die bestehende Hardware ihr Potential ausgeschöpft hatte. Es wurde ein Wechsel beschlossen, in deren Ergebnis ein komplett neuer Roboter entwickelt wurde. Hierbei wurde im Rahmen eines Plattformwechsels von einem 8-Bit-Mikroprozessor zu einem 32-Bit-Linux-System eine Portierung der bisherigen Software notwendig.

Diese Portierung und begleitende Arbeiten wurden im Rahmen der vorliegenden Studienarbeit durchgeführt.

1.1 Aufbau der Arbeit

Nach dieser Einleitung werden im zweiten Kapitel die bis 2008 eingesetzten Roboter betrachtet, hierbei insbesondere auch die Softwarearchitektur. In Kapitel 3 wird dann kurz die neue Plattform, sowohl bezüglich Hardware als auch Betriebssystem, vorgestellt und in Kapitel 4 die Portierung und Motoransteuerung beschrieben. Kapitel 5 behandelt weitere unterstützende Arbeiten, die im Rahmen der Studienarbeit anfielen. Vorschläge für zukünftige Verbesserungen werden in Kapitel 6 dargestellt, bevor in Kapitel 7 das Fazit gezogen wird.

2 Alte Plattform

In diesem Kapitel werden kurz die bis 2008 eingesetzten Roboter in Bezug auf Hardware und Software vorgestellt. Der Schwerpunkt liegt hier auf der Beschreibung der Software-Architektur, da diese bei der Portierung beibehalten werden sollte.

2.1 Hardware

In den RoboCup-Wettkämpfen 2007/2008 basierten die FUmanoids im Kern noch auf dem von Robotis vertriebenen Bioloid-Kit[1], insbesondere der dort enthaltene CM-5[2] fand Anwendung. Hierbei handelt es sich um das Bioloid *C*ontrol *M*odule, welches den Prozessor (s.u.), Stromversorgung, Anschlüsse für die Motoren sowie Kontrolltaster umfaßt.

[1] http://www.robotis.com/zbxe/bioloid_en
[2] http://www.robotis.com/zbxe/controller_en

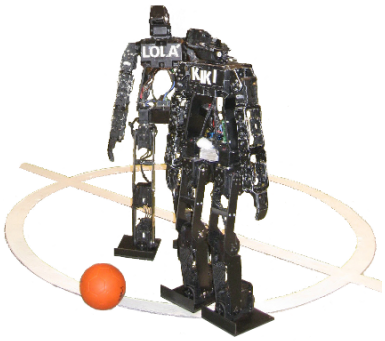

Abbildung 1: Kiki und Lola

2.1.1 CPU

Die im CM-5 eingesetzte CPU war ein ATMega128[3], eine 8-Bit-RISC-CPU von ATMEL mit 16-Bit-Befehls- und Adreßbus. Der Prozessor ist mit 16 MHz getaktet (Leistung hierbei 16 MIPS) und enthält 128 Kilobyte Flashspeicher, 4 Kilobyte SRAM sowie 4 Kilobyte EEPROM. Er verfügt desweiteren über zwei serielle Schnittstellen, womit einerseits die Motoren ansprechbar sind, und andererseits eine Verbindung mit einem PC hergestellt werden kann.

2.1.2 Motoren

Die im Roboter verbauten Aktuatoren waren Servomotoren vom Typ Dynamixel[4] AX-12 und RX-28 der Firma Robotis Inc., Korea. Sie verfügen jeweils über einen eingebauten Mikrokontroller, mit welchem über einen RS-485-Bus[5] kommuniziert werden kann. Der Mikrokontroller stellt neben Funktionen zum Setzen der Motorposition und anderer Parameter, wie z.B. Spielraumbeschränkungen, auch Informationen über die aktuelle Lage, Belastung und Geschwindigkeit zur Verfügung. Dies ermöglicht eine präzise Steuerung und Abfrage der insgesamt 18 Motoren.

2.1.3 Kommunikation

Der Roboter konnte über eine serielle Verbindung mit einem angeschlossenen PC kommunizieren. In einer Terminal-Sitzung konnten hier Befehle an den Roboter gegeben werden, um zum Beispiel Parameter zu setzen oder Aktionen zu initiieren.

Da während der Wettkämpfe eine Verkabelung der Roboter eher ungünstig ist, wurde auch ein WLAN-Modul[6] eingebaut, auf welches vom Roboter über eine serielle Schnittstelle zugegriffen werden konnte. Die WLAN-Verbindung wurde genutzt, um dem Roboter die Initialisierungsbefehle für ein Spiel zu übermitteln (u.a. die Information, welche Farbe das gegnerische Tor hat und welche Strategie gespielt werden soll) und um eine Teamkommunikation zu ermöglichen.

2.1.4 Kamera

Für die Objekterkennung wurde eine selbständige Einheit benutzt. Das sogenannte HaViMo[7] umfaßt hierbei eine integrierte Kamera mit einer Auflösung von 160x120 Pixeln mit 8 fps, sowie

[3] http://www.atmel.com/dyn/products/product_card.asp?part_id=2018
[4] http://www.robotis.com/zbxe/dynamixel_en
[5] http://de.wikipedia.org/wiki/EIA-485
[6] http://www.avisaro.com/tl/index.php/wlan-modul-10.html
[7] http://www.5am.de/robotics/index.php?lang=de

dem HV7131GP Chipset von Magnachip. Die Objekterkennung (Farbregionen) findet noch im Modul statt, so daß die Haupt-CPU des Roboters entlastet wird. Das HaViMo verfügt hierbei über einen RS485-Anschluß mit dem Bioloid-Kommunikationsprotokoll, womit es einfach mit an den Motorbus angeschlossen werden kann.

2.2 Software

Bei der Software handelte es sich um eine Eigenentwicklung, welche auf dem CM-5 lief. Als Entwicklungssprache wurde hierbei C gewählt.

2.2.1 Architektur

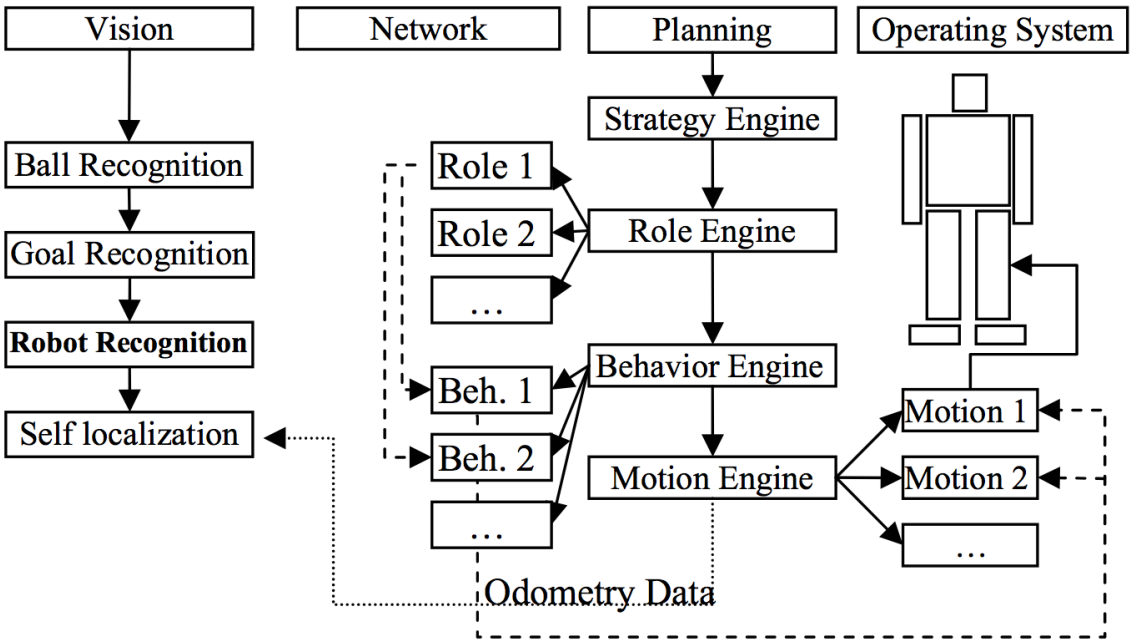

Abbildung 2: Software-Architektur (2008) [6]

Abbildung 2 zeigt die Gesamt-Architektur der FUmanoid-Software im Jahr 2008.

Das Betriebssystem ist hierbei nicht per se existent, da die Software direkt auf der CPU ohne Zwischenschicht läuft. Höhere Funktionalität mußte bei Bedarf selbst entwickelt werden, z.B. wurden Threads durch einen im FUmanoid-Programm enthaltenen Scheduler und durch Inanspruchnahme von Timer-Interrupts implementiert. Der Großteil der unter Vision aufgeführten Punkte wurde desweiteren bereits im Kamera-Modul (HaViMo) realisiert.

Primär von Interesse an der Architektur ist hier jedoch der mehrschichtige Aufbau. In jeder Schicht läuft ein sogenanntes Szenario, wobei es je Schicht mehrere mögliche Szenarien gibt, abhängig von der momentan benötigten Aktion. Ein Szenario ist hierbei Programmcode zur Lösung einer bestimmten Aufgabe, der kontinuierlich ausgeführt wird. Hierbei steigt bei den Schichten von unten nach oben die Abstraktion, während zugleich die Hardwarenähe sinkt.

Diese Architektur wird in [8] als Concurrent Scenario Based Planning (CSBP) bezeichnet. Sie ist in der Leistung äquivalent zu zustandsbasierten Architekturen[8], sie hat allerdings den Vorteil das der Programmcode wesentlich leichter zu erstellen und insbesondere auch zu debuggen ist.

Die Kommunikation zwischen den verschiedenen Schichten ist auf aneinandergrenzende Szenarien beschränkt und basiert auf dem *command/status model*[8]. Dies heißt, daß die Kommunikation von oben nach unten durch Austausch des in der tieferliegenden Schicht laufenden Szenarios erfolgt (*command*). Umgekehrt teilt eine tiefergelegene Schicht anhand von Statuswerten der höheren Schicht ihren Fortschritt mit.

In der FUmanoid-Software beliefen sich die Statuswerte im Regelfall auf Hinweise, ob das jeweilige Szenario erfolgreich beendet wurde (z.B. wenn das Ziel erreicht wurde) oder ob es fehlschlug.

Im Falle der FUmanoids laufen die Szenarien in vier Schichten. Diese sind, von *oben* nach *unten* gesehen:

- Strategie (Strategy)
- Rolle (Role)
- Verhalten (Behavior)
- Bewegung (Motion)

Bewegung

In der Bewegungs-Schicht werden die Bewegungen des Roboters realisiert. Primär ist hier die Laufbewegung implementiert, aber auch das Abspielen von vorab aufgenommenen Bewegungen, wie z.B. das Aufstehen, ist enthalten.

Verhalten

Die Verhaltens-Schicht beinhaltet die möglichen Verhaltensweisen des Roboters. Diese legen fest, was der Roboter gerade tut. Nennenswerte Beispiele sind hier das Suchen des Balles, das Laufen zum Ball oder zu einer anderen Position oder ebenso das Dribbeln und Schießen des Balles. Die mit dem jeweiligen Verhalten zusammenhängenden Bewegungen werden durch Setzen eines entsprechenden Szenarios in der Bewegungs-Schicht realisiert und im Falle des Laufens durch Übergabe von Parametern gesteuert.

Rolle

Die nächsthöhere Schicht ist für die Rolle des Roboters zuständig. Implementierte Rollen sind z.B. Stürmer, Verteidiger oder Torwart. Die Rolle definiert die Abfolge von Verhalten, größtenteils handelt es sich um einen Zustandsautomaten der abhängig vom Erfolg oder Mißerfolg eines Verhaltens das jeweils nächste Verhalten startet.

Strategie

Als oberste Schicht sind die Strategien implementiert. Diese sind für die Koordination der Rollen der Roboter des Teams zuständig. Sie agieren also ausschließlich auf Informationen der anderen Roboter bezüglich derer Rollen und dem aktuellen Status. Natürlich muß im Team jeder Roboter die gleiche Strategie benutzen, sie ändert sich also im Regelfall nur wenn durch das Bedienprogramm eine Änderung ausgelöst wird.

Falls keine Teamkommunikation möglich (oder - wie z.B. beim Testen - gewünscht) ist, besteht auch die Option keine Strategie zu setzen. In dem Fall ist die Rolle die höchste Schicht.

2.2.2 Bedienung und Kommunikation

Wie im Abschnitt zur Hardware ausgeführt, konnten die Roboter sowohl über eine serielle Schnittstelle als auch über WLAN kommunizieren.

Hierbei wurde die Teamkommunikation, also das Senden/Empfangen von Statuspaketen (unter anderem mit Informationen über die aktuell ausgeführten Szenarien in den verschiedenen Schichten), über WLAN abgewickelt, indem zweimal pro Sekunde ein UDP-Paket gesendet wurde.

Auch die Initialisierung des Roboters wurde über WLAN mit UDP-Paketen realisiert. Hierzu gab es ein PC-seitiges Programm (*Serverprogramm*), mit welchem sich für den Roboter die Spielparameter einstellen ließen. Diese waren u.a. die zu benutzende Strategie oder Rolle, die Farbe des gegnerischen Tores und ob das eigene Team Anstoß hatte. Zusätzlich zeigte das Serverprogramm die vom Roboter geschickten Log-Nachrichten an.

Alle weiteren Befehle wurden an der seriellen Schnittstelle entgegengenommen, zum Beispiel das Setzen der eindeutigen ID des Roboters, Anzeigen und Setzen von Motorwerten, Kontrolle der Kamera, usw.

2.2.3 Problemstellen in alter Software

Die alte Software wurde im Laufe von mehr als zwei Jahren entwickelt und insbesondere bei Teilnahmen am RoboCup oft unter großem Zeitdruck angepaßt. Hinzu kam, daß der ATMega mit seinen begrenzten Ressourcen, insbesondere im Bereich des Arbeitsspeichers, oftmals den Einsatz von „Tricks" erforderlich machte um die Software auch bei steigender Funktionalität noch laufen lassen zu können beziehungsweise zu erweitern. Desweiteren war die Entwicklungsplattform in einigen Punkten nicht optimal, so wurden zum Beispiel stets alle Dateien vor dem Kompilieren in eine einzelne große Quellcodedatei zusammengefaßt. Dieses führte dazu, daß dateibezogene Scopes[8] oftmals effektiv außer Kraft gesetzt waren und alle Funktionen global wurden - und schlußendlich auch so genutzt wurden.

Obige Punkte führten dazu, daß der Quellcode massiven Einsatz von globalen Variablen machte. Erschwerend kam hinzu, daß aufgrund des begrenzten Speichers einige Variablen doppelt belegt wurden. Zum Beispiel wurden für die Parameterübergabe an Verhaltens- und Bewegungsszenarien globale statische Integer-Arrays benutzt, bei denen Teile des Inhalts vom Kontext abhängig waren.

3 Neue Plattform

Ende 2008 begann die Planung und Arbeit für die neuen Roboter (Bild 9 auf Seite 24), von denen letztendlich vier Stück gebaut wurden. Im Gegensatz zu den vorigen Jahren wurde diesmal auch bei der Mechanik und Elektronik alles von Grund auf selber entwickelt.

In diesem Kapitel sollen von den vielen Änderungen nur die für diese Arbeit relevanten Neuheiten in Hardware und Betriebssystem-Software vorgestellt werden.

3.1 Hardware

Als Plattform für die neuen Roboter wurde der Gumstix[9] auserkoren. Hierbei handelt es sich um ein in Kaugummipackung-Größe gehaltenes Motherboard mit der Möglichkeit, Expansionskarten für zusätzliche Funktionalität zu nutzen.

[8] Als Scope wird der Bereich in einem Programm bezeichnet, in dem eine Variable „sichtbar" ist.
[9] http://gumstix.com/store/catalog/images/verdex-pro6.jpg

Abbildung 3: Gumstix

Das benutzte Modell verdex pro XL6P[10] verfügt über einen 600 MHz XScale ARM-Prozessor vom Typ Marvell PXA270, 128 MB RAM und 32 MB Flashspeicher. Durch eine MicroSD-Karte läßt sich der Speicher erweitern. Es existieren desweiteren Schnittstellen für serielle Verbindungen.

An Erweiterungen wurde ein netpro-vx[11]-Modul eingesetzt, welches neben einer Ethernet-Schnittstelle auch ein WLAN-Modul einbindet.

Zusätzlich wurde eine im Rahmen einer anderen Diplomarbeit entwickelte Kameraplatine eingesetzt[12], welche zwischen Gumstix und netpro-vx-Modul gesteckt wurde. Die Kameraplatine verband hierbei zwei auf der Platine montierte Kameras mit dem Quick-Capture-Interface des PXA270.

Zuletzt kam auch noch eine Platine zum Einsatz, welche eine Verbindung des Gumstixs zu den Motoren schaffte.

3.2 OpenEmbedded

Abbildung 4: OpenEmbedded-Logo

Auf dem Gumstix wird eine OpenEmbedded[13]-basierte Linux-Distribution eingesetzt. OpenEmbedded (OE) ist ein System, mit dem sich mithilfe des bitbake-Werkzeuges und sogenannten Rezepten ein individuell konfiguriertes Linuxsystem erstellen läßt. Der Schwerpunkt liegt hierbei auf mobilen und eingebetteten Systemen.

4 Portierung

4.1 Anforderungen

Die im Rahmen der Studienarbeit zu lösende Aufgabe bestand - grob umrissen - darin, die existierende Software nach Linux zu portieren. Zum Zeitpunkt des Arbeitsbeginns war das Ziel, die existierenden Szenarien der alten Roboter mit minimalen Änderungen zu übernehmen. Hierbei

[10] http://gumstix.com/store/catalog/product_info.php?products_id=210
[11] http://gumstix.com/store/catalog/product_info.php?products_id=207
[12] Fischer, Bennet: *Hard- und Software-Entwicklung für ein Stereo-Vision-System in eingebetteter Linux-Umgebung* (Diplomarbeit, 2009), Technische Universität Berlin
[13] http://www.openembedded.org

bliebe das darin gesammelte, durch im Laufe von in Wettkämpfen durchgeführten Änderungen auch nicht immer komplett dokumentierte Wissen erhalten. Durch eine ähnliche Struktur wäre auch ein einfacherer Vergleich von Code in der neuen Version mit Code in der alten Version der Software möglich.

Insbesondere sollte somit auch die in Abschnitt 2.2.1 auf Seite 6 beschriebene Software-Architektur beibehalten werden.

Hinzu kam auch die Aufgabe, die portierte Software auf dem neuen Roboter laufen zu lassen und den anderen Teammitgliedern den Einstieg zu erleichtern. Diesbezügliche Maßnahmen werden in Kapitel 5 auf Seite 20 vorgestellt.

4.2 CPU-bedingte Änderungen

Die notwendigen Änderungen aufgrund des Wechsels von einer 8-Bit- auf eine 32-Bit-CPU hielten sich in Grenzen.

Ein potentielles Problem bei einem CPU-Wechsel ist stets die Endian-Kodierung, also wie Werte mit einer Datenlänge größer als ein Byte im Speicher abgelegt werden. Im vorliegenden Fall war dies jedoch nicht relevant, da der PXA270 für Little Endian konfiguriert ist und der ATMega128 ebenfalls eine Little-Endian-CPU ist.

Durch den Wechsel auf eine 32-Bit-Architektur war es an einigen Stellen jedoch notwendig die Datentypen anzupassen, da auf dem ATMega128 ein *int* 2 Byte groß war, es aber auf dem PXA270 die doppelte Länge hatte. Im Regelfall wurde dies dahingehend gelöst, die ehemaligen *int*-Werte auf *short int* umzustellen. Beziehungsweise wurden, um für zukünftige Plattformwechsel (z.B. von 32 auf 64 Bit) vorbereitet zu sein, die aus dem C99-Standard definierten Integertypen benutzt ([5, Abschnitt 7.18.1.1]). Das heißt, daß Integertypen durchgängig mit *(u)int*N_*t* bezeichnet wurden, wobei *N* die Datenbreite in Bits ist und das optionale *u* einen nicht-vorzeichenbehafteten-Datentyp kennzeichnet.

Da allerdings die gesamte Kommunikation mit der Server-Software neu entwickelt wurde (siehe hierzu Abschnitt 4.6), bestanden keine externe Abhängigkeiten bei den Datenformaten. Es mußte somit keine größere Rücksicht genommen werden, wenn Datentypen angepaßt wurden.

4.3 Wechsel zu C++

Wie in Abschnitt 2.2.3 angesprochen, war der alte Programmcode in einigen Punkten nicht mehr gut wartbar. Im Rahmen der Portierung wurde daher auch überlegt, wie ähnliche Entwicklungen in Zukunft erschwert werden können und ganz allgemein die Entwicklung der Software vereinfacht werden kann.

Ein naheliegender Gedanke war hierbei der Umstieg auf eine objektorientierte Programmiersprache. Da die Software bisher in C entwickelt wurde, fiel die Wahl auf C++. Hierdurch können die Vorteile einer objektorientierten Softwareentwicklung genutzt werden, also z.B.

- Abstraktion - Bündelung von Funktionalität/Werten innerhalb von Klassen, die klar definierte Aufgabenbereiche haben und auf denen agiert werden kann
- Datenkapselung - Implementationsdetails werden hinter einer „neutral" definierten Schnittstelle verborgen
- Persistenz - Variablenwerte eines Objektes bleiben gültig, solange das Objekt existiert
- Vererbung - Möglichkeit, Klassen abzuleiten und zu spezialisieren und damit u.a. auch bestehenden Code weiter zu nutzen

Insbesondere schon durch die Abstraktion der Funktionalität in klar definierte Klassen vereinfacht sich die Arbeit mit der Software ungemein. So läßt sich der Programmablauf wesentlich einfacher verinnerlichen und nachvollziehen. Durch die Datenkapselung ist desweiteren kein tieferes Wissen über die in einer Klasse genutzten Algorithmen oder internen Werte nötig. Insbesondere ist auch ein Zugriff auf diese klasseninternen Werte nicht auf Anhieb möglich.

4.4 Aufbau der Software

Im Zentrum des Programms steht die Klasse *Robot*. Diese symbolisiert den Roboter und enthält den Initialisierungscode sowie die Verwaltung der Lebenszeiten von Objekten, insbesondere der Szenarien in den verschiedenen Schichten.

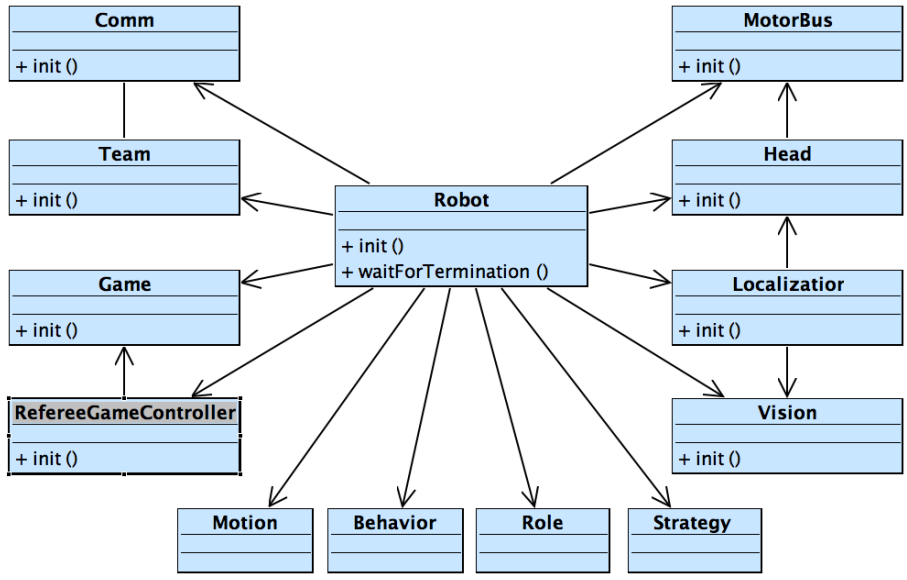

Abbildung 5: Die wichtigsten Klassen

Abbildung 5 zeigt die wichtigsten Klassen des FUmanoid-Programms:

Comm Die *Comm*-Klasse ist die Schnittstelle zur Außenwelt. Sie verwaltet die Kommunikation und verteilt eingehende Datenpakete an die entsprechenden Objekte beziehungsweise führt sie einige Befehle direkt aus.

Team Eine Instanz dieser Klasse repräsentiert das eigene Team. Informationen über die anderen Roboter werden gespeichert und können im Detail oder zusammengefaßt abgefragt werden. Desweiteren werden in regelmäßigen Abständen Statusnachrichten an das Team geschickt.

Game Der Spielstatus wird in dieser Klasse verwaltet. Den aktuellen Spielregeln folgend, existieren vier Spielzustände: *STOPPED* (Spiel ist pausiert, beendet oder noch nicht gestartet), *READY* (Roboter sollen sich vorbereiten, d.h. sich auf die gewünschten Startpositionen begeben), *SET* (Roboter darf sich nicht bewegen und wartet auf den Anstoßpfiff) und *PLAYING*. Ebenfalls in *Game* finden sich Informationen darüber, welches Team Anstoß hat. Desweiteren können einige vom Spielzustand abhängige Daten abgefragt werden, insbesondere ob der Ball im Spiel ist[14].

[14]Der Ball ist im Spiel, falls das eigene Team Anstoß hat, oder falls das gegnerische Team den Ball berührt hat oder bereits 10 Sekunden seit Anpfiff vergangen sind. Ist der Ball nicht im Spiel, darf der Roboter den inneren Kreis nicht betreten.

RefereeGameController Um eine einheitliche Informationsübermittlung an die verschiedenen Roboter eines Spieles voranzubringen, wird seit einigen Jahren der Versuch unternommen, einen sogenannten *Game Controller*[15] einzusetzen. Dieser soll von einem Schiedsrichterassistenten bedient werden und die Software sendet alle 500 Millisekunden ein Informationspaket unter anderem über den Spielzustand (s.o.), den Spielstand, die Anstoßinformationen, die Teamfarben und Strafzeiten der einzelnen Spiele. Die Klasse *RefereeGameController* wertet nun diese Informationen aus und setzt die entsprechenden Informationen primär in der Klasse *Game*.

MotorBus Diese Klasse ist die Schnittstelle zu den Motoren und den weiteren Sensoren. Sie wird in Abschnitt 4.5 näher betrachtet.

Head Die Kontrolle über den Kopfmotor und damit die Ausrichtung der Kameras wird von dieser Klasse gehandhabt.

Localization Die Roboter sollen zunehmend selbständiger werden, dies schließt auch ein das sie sich eigenständig orten können. Die im Rahmen einer anderen Diplomarbeit entstandene Klasse *Localization* ist genau hierfür zuständig.

Vision Diese Klasse verarbeitet die Kamerabilder und stellt Informationen über gesehene Objekte bereit.

Abgesehen von den in der unteren Zeile von Abbildung 5 eingetragenen Klassen *Motion*, *Behavior*, *Role* und *Strategy*, die weiter unten näher betrachtet werden, handelt es sich jeweils um Klassen, die zum Programmstart einmalig instantiiert werden und dann während des gesamtes Programmablaufs zur Verfügung stehen. Einige hiervon laufen in einem eigenen Thread, genaugenommen sind dies *Comm*, *Team*, *Vision* und *Localization*.

Der Programmstart des FUmanoid-Programms sieht damit in etwa wie folgt aus:

```
int main(int argc, char* argv[]) {
   robot.init();
   robot.waitForTermination();
   return 0;
}
```

robot ist hierbei die globale Instanz der *Robot*-Klasse. In der *init()*-Funktion werden die verschiedenen anderen Objekte initialisiert. Mit *waitForTermination()* wird gewartet, bis sich das Programm beendet - dies ist allerdings primär bei Testläufen relevant, da in Spielen der Roboter meistens nicht korrekt heruntergefahren werden kann.

Bei Start des Programms werden noch keine Szenarien gestartet. Dies kann entweder durch von *FUremote* gesendete Befehle (siehe Abschnitt 4.6) erfolgen, oder optional durch manuelles Positionieren der Roboter-Arme in bestimmte Stellungen. Letzteres wurde eingebaut, damit bei Ausfall der WLAN-Funktionalität die Roboter trotzdem spielen können. Regulär werden die Szenarien aber über von der *Comm*-Klasse empfangene Datenpakete gestartet.

In einem Wettkampf bedeutet dies, daß ein Strategie-Szenario gestartet wird, welches dann dem Roboter ein Rollen-Szenario zuweist. Dieses startet wiederum ein Verhaltens-Szenario und damit wird dann auch ein Szenario in der Bewegungs-Schicht gesetzt.

Szenarien sind hierbei in Klassen implementiert, die sich - je nach Schicht - von einer abstrakten Basisklasse ableiten. Den Zusammenhang zwischen den Klassen und Schichten stellt Abbildung 6 dar.

[15] http://robocup.informatik.tu-darmstadt.de/humanoid/gc/

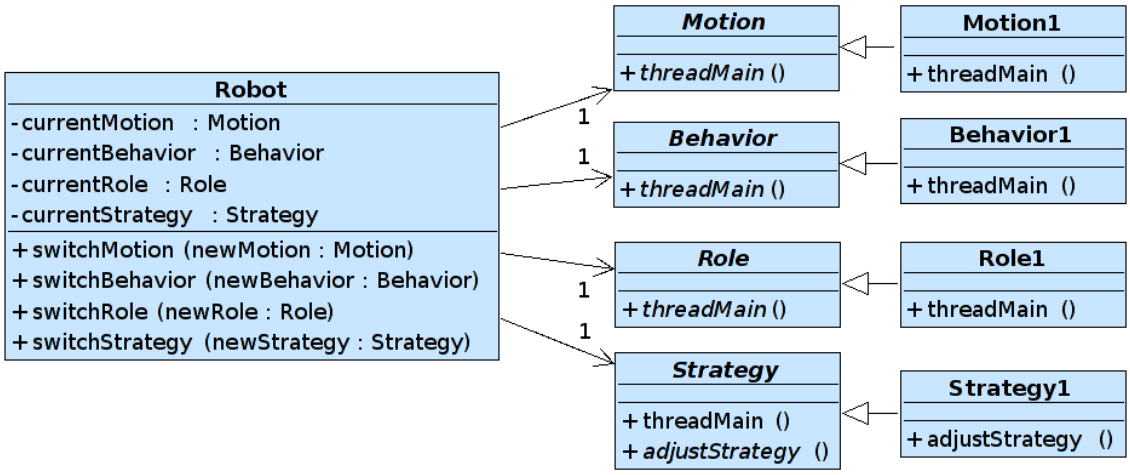

Abbildung 6: Klassendiagramm für die Schichten

Da eine nebenläufige Ausführung gewünscht ist, sind die abstrakten Klassen (*Motion*, *Behavior*, *Role* und *Strategy*) jeweils von einem Thread-Objekt (s.u.) abgeleitet.

Um nun zum Beispiel eine Strategie zu starten, wird *Robot::switchStrategy* aufgerufen. Dieses nimmt eine Instanz einer von *Strategy* abgeleiteten Klasse entgegen. Sollte bereits ein Strategie-Szenario laufen, so wird dieses zuerst beendet. Anschließend wird das neue Szenario gestartet. Dies erfolgt mit Hilfe der vom *Thread*-Objekt vererbten *run*-Methode, welche das Szenario als neuen Thread aktiviert. Die *switchRole*, *switchBehavior* und *switchMotion* Methoden der Klasse *Robot* funktionieren analog. Der für das Szenario auszuführende Code ist jeweils in der *threadMain*-Methode implementiert.

4.5 Motoransteuerung

In den Aufgabenbereich der Portierung fiel auch die Ansteuerung der Motoren.

Zu Beginn der Entwicklung wurde mit Hilfe eines USB2Dynamixels[16] mit den Robotern kommuniziert. Hierbei handelt es sich um ein USB-Gerät mit FTDI-Chip[17], welches den seriellen (USB-)Anschluß auf PC-Seite (oder Gumstix-Seite) mit dem RS485-sprechenden Motoren verbinden kann.

Die USB-Verbindung wurde genutzt, weil anfangs noch keine direkte Verbindung des Gumstix mit den Motoren verfügbar war. Desweiteren war das Ziel, zu Testzwecken auch ohne Einsatz des Gumstix die Motoren ansprechen zu können und das FUmanoid-Programm auf einem PC laufen zu lassen. Trotz einer eingestellten Baudrate zu den Motoren von 1 MBaud zeigte sich aber schnell, daß der USB-Durchsatz zu ungenügend ist um die gewünschte Framerate beim Laufen zu erhalten. Die erreichte Maximalgeschwindigkeit betrug 27 fps.

Rechnung:

$$\frac{\frac{\text{MaxRequests}}{s}}{N_{Motors}} = \frac{\frac{500}{s}}{18} = 27$$

> „Traffic on the USB is regulated using time. The unit of time is the frame. The length of each frame is governed by the bus clock, which runs at a rate of 1KHz, so

[16] http://www.robotis.com/zbxe/controller_en
[17] http://www.ftdichip.com

> there are 1,000 frames per second: one per millisecond. At the start of each frame a Start Of Frame (SOF) packet is sent over the bus, allowing isochronous devices to synchronise with the bus."[18]

Auf Seiten des Gumstix kommt noch hinzu, daß nur ein USB 1.1 Anschluß verfügbar ist, womit dann mit identischem Code Latenzzeiten von nur 8ms erreicht wurden. Dies entsprach einem Wert von 5 fps. Damit fiel die Option, die Motoren über USB anzusprechen (oder dies zumindestens als Ausweichmöglichkeit vorzuhalten), weg.

Als nächstes wurden die Motoren über eine serielle Schnittstelle mit dem Gumstix verbunden. Hierfür wurde eine vom Team entwickelte Platine mit einem MAX232-Chip eingesetzt.

Der auf dem Gumstix eingesetzte Marvell PXA270 Prozessor unterstützt Baudraten nach folgender Formel

$$Baudrate = \frac{14.7456 \text{ MHz}}{(16 \times Divisor)}$$

während der AVR-Mikrocontroller die durch

$$Baudrate = \frac{2000000 \text{ Hz}}{Divisor + 1}$$

bestimmten Geschwindigkeiten erreicht. Innerhalb eines kleinen Prozentbereiches (beim AVR-Mikrocontroller bis zu maximal 3%) können Abweichungen der Baudraten toleriert werden. Durch Vergleichen der möglichen Baudraten auf beiden Systemen kristallisiert sich eine maximale Geschwindigkeit von ca 184 Kilobaud heraus, und zwar 181818 Baud auf dem AVR-Mikrocontroller mit Divisor 10 und 184320 Baud auf dem PXA270 mit Divisor 5 mit einer Abweichung von 1%. Alle höheren Geschwindigkeiten haben eine Mindestabweichung von 4% (zum Beispiel 222222 Baud versus 230400 Baud) oder höher.

Die ursprüngliche Planung, daß weiterhin mit einer Geschwindigkeit von einem Megabaud mit den Motoren kommuniziert werden kann, mußte damit fallengelassen werden.

Werden ausschließlich die Kommunikationszeiten betrachtet und Bearbeitungs- und Latenzzeiten ignoriert, so können bei 181818 Baud in jeder Sekunde 1136 einzelne Motorwerte gelesen, bzw. 2272 einzelne Motorwerte gesetzt werden. Das Setzen aller 21 Motoren kann pro Sekunde also ca. 257 mal, ein Lesen aller 21 Motoren ca. 54 Mal erfolgen. In der Praxis sind diese Idealwerte natürlich nicht erreichbar, da durch die Latenz zwischen Senden des letzten Anfragebits und Empfang des ersten Antwortbits und durch die Verarbeitung der Daten sowohl im Kernel- als auch im Userspace die Maximalgeschwindigkeit sinkt.

Implementation

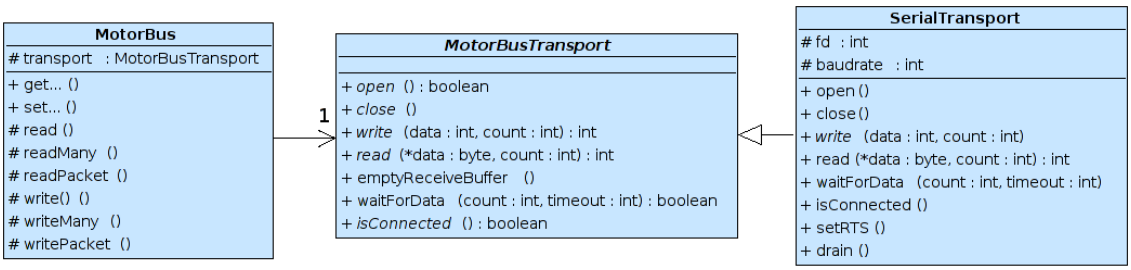

Abbildung 7: Klassen der Motorkommunikation

[18] http://www.tech-pro.net/intro_usb.html

Um die Implementationsdetails der Motorkommunikation zu verstecken, wurde die Klasse *MotorBus* erstellt.

MotorBus enthält hierbei Zugriffsfunktionen und Zuweisungsfunktionen für alle verfügbaren Motorwerte. Intern werden die Funktionsaufrufe innerhalb der *MotorBus*-Klasse in die benötigten Datenpakete für die Dynamixel-Motoren umgewandelt und synchron über die benutzte Instanz einer von SerialTransport abgeleiteten Klasse an die Motoren geschickt beziehungsweise empfangen.

In der vorliegenden Implementation gibt es insgesamt drei Transportklassen. *MotorBusTransportSerial* ist hierbei die auf dem Gumstix eingesetzte, welche über den seriellen Port via dem RS232-zu-RS485-Wandler die Motoren anspricht. Für den Einsatz auf einem PC existieren zwei zusätzliche Klassen, die über USB (USB2Dynamixel) mit den Motoren kommunizieren, und zwar einmal mit der libftdi und einmal mit einem proprietären Treiber von FTDI.

Wie im vorigen Abschnitt angesprochen, ist durch die begrenzte Baudrate eine Kommunikation im Vergleich zum Vorjahr nur mit einer wesentlich geringeren Geschwindigkeit möglich. Um diesen Geschwindigkeitsverlust auszugleichen, wurden verschiedene Optimierungen vorgenommen.

In der *MotorBus*-Klasse war die Hauptoptimierung das Hinzufügen der Option, beim Setzen von Motorwerten verschiedener Motoren dies gebündelt durchzuführen. Das wurde realisiert, indem Unterstützung für den im Dynamixel-Protokoll vorhandenen SYNC_WRITE-Befehl hinzugefügt wurde. Mit diesem können mit einem Datenpaket an die gewünschten Motoren Werte der gleichen Länge an die jeweils gleiche Adresse geschrieben werden. Hierbei wird der Overhead gespart, der bei individuellen WRITE-Befehlen angefallen wäre. So benötigt das Setzen von 10 Motorzielpositionen anstelle von 90 nur noch 37 Bytes.

Kernel-Anpassungen

Um die Kommunikation auch auf der seriellen Ebene zu beschleunigen, wurde *low latency* im Kernel aktiviert, das heißt das der User-Space bereits beim Eintreffen eines einzelnen Bytes benachrichtigt wird. Dies ist nötig, da anderenfalls erst bei vollem Puffer oder bei Erreichen eines Timeouts die empfangenen Daten weitergegeben werden. Da die Pakete von den Motoren jedoch relativ klein sind und zwar weit unterhalb der Puffer-Größe, würde dabei jedesmal der Timeout zum Tragen kommen. Der Nachteil von *low latency* ist allerdings eine erhöhte CPU-Last.

Weitere Änderungen waren notwendig, um den MAX232 beim Senden umzuschalten, da auf einem RS485-Bus stets nur gesendet oder empfangen werden kann. Die Umschaltung wurde im vorliegenden Fall mit Hilfe der RTS-Leitung realisiert. Da diese Umschaltung in der FUmanoid-Software im Userspace stets mit einer gewissen Verzögerung erfolgt, müßte für die Motoren eine höhere Latenzzeit bei den Antworten eingestellt werden, damit keine Daten verlorengehen. Dies würde eine weitere Verschlechterung der Kommunikationsgeschwindigkeit mit sich führen. Aus diesem Grund wurde die Umschaltung in den Kernel-Space verlegt, indem der serielle Treiber angepaßt wurde.

Hierzu wurden in *drivers/serial/pxa.c* die folgenden Änderungen vorgenommen:

- Bei aktiviertem *low latency*-Modus wird im FIFO Control Register[19] auch der *Interrupt Trigger Level* dahingehend gesetzt, daß bereits beim Eintreffen eines einzelnen Bytes ein *Received Data Available*-Interrupt ausgelöst wird. Dies führt, ebenso wie die Einstellung des *low latency*-Modus im Kernel, zu einer erhöhten CPU-Belastung, da mehr Interrupts generiert und behandelt werden müssen. Allerdings würde anderenfalls der Interrupt erst nach 32 empfangenen Bytes ausgelöst oder nachdem für die Dauer von 4 Bytes[20] keine weiteren Daten empfangen wurden. Wie ausgeführt sind aber die Antwortpakete der Motoren oftmals sehr klein (8 Byte) und wir würden jedesmal den Timeout bemühen müssen.

[19] [2, Abschnitt 10.5.6]
[20] Dies entspricht ca. 220 Mikrosekunden.

- Das FIFO Control Register (FCR) wurde desweiteren so gesetzt, daß erst dann ein Transmitter-Interrupt gesendet wird, sobald der (Sende-)FIFO der seriellen Schnittstelle leer ist. Standardmäßig würde der Interrupt gesendet werden, wenn der FIFO halbleer ist.
- Beim Senden von Daten wird zuallererst die RTS-Leitung freigegeben. Dies signalisiert dem MAX232, daß Daten gesendet werden.
- Desweiteren wird die RTS-Leitung gesetzt, sobald keine weiteren zu sendenden Daten mehr vorliegen und ein Transmitter-Interrupt empfangen wurde. Da im FCR eingestellt wurde, den Transmitter-Interrupt erst dann zu senden wenn der Sende-FIFO leer ist, haben wir damit den frühstmöglichen Zeitpunkt bestimmt ab dem das Senden wirklich beendet ist.

Insgesamt können die im Kernel vorgenommenen Änderungen dahingehend zusammengefaßt werden, daß auf Kosten der CPU-Auslastung die Latenzzeiten bei der seriellen Kommunikation mit den Motoren gesenkt wurden. Eine alternative Lösung wird in Abschnitt 6.1 vorgeschlagen.

4.6 Kommunikation

Im Gegensatz zum alten Roboter sollte bei dem neuen Modell die gesamte Kommunikation über das Netzwerk erfolgen, also keine serielle Terminalverbindung mehr angeboten werden. Hierzu wurde in der Software ein eigener Thread geschaffen (Klasse *Comm*), der auf dem designierten Port auf eingehende Datenpakete wartet.

Auf PC-Seite wurde parallel hierzu im Rahmen einer Bachelorarbeit das bisherige in Delphi geschriebene *ServerProgramm* durch das Java-basierte Programm *FUremote* ersetzt. Dieses basiert auf Eclipse RCP[21] und bietet mithilfe von Plugins verschiedene Funktionalitäten, die vorher in getrennten Programmen implementiert waren (z.B. das *ServerProgramm* aber auch der *Motion-Editor* zum Aufnehmen und Abspielen von statischen Bewegungen).

Da während Wettkämpfen eine Kommunikation nur per UDP erfolgen darf, wurde der Einfachheit halber die gesamte Netzwerkkommunikation via UDP realisiert. Dies hat den Vorteil, daß sowohl auf Roboter- als auch auf PC-Seite jeweils nur eine Netzwerkverbindung verwaltet werden muß. Die Nachteile von UDP sind natürlich, daß die Reihenfolge der empfangenen nicht der Reihenfolge der gesendeten Datenpakete entsprechen muß, desweiteren beinhaltet UDP keine Mechanismen um einen Verlust von Datenpaketen zu verhindern. Außerhalb von Spielen erfolgt die Kommunikation in der Regel jedoch relativ lokal und abgeschirmt über eine direkte Kabelverbindung oder einfach-indirekt via WLAN-Router, so daß verlorene Datenpakete und eine falsche Reihenfolge so gut wie nie auftreten und keine größeren Auswirkungen haben.

Beim Entwurf des Kommunikationsprotokolls wurde wiederum viel Wert auf eine einfache Struktur mit minimalem Overhead gelegt. Jedes Paket beginnt mit den folgenden Werten:

```
uint16_t operationID
uint8_t  flags
```

Hierbei ist *operationID* die Kennung des Befehls. *flags* sind optionale Flags, von denen aktuell nur eines namens *FLAG_IS_ANSWER (1)* existiert, welches signalisiert das es sich um ein Antwortpaket auf eine vorhergehende Anfrage handelt. Optional folgen - abhängig vom gesendeten Befehl - weitere Bytes mit dem Payload. Dadurch, daß Antwortpakete die gleiche *operationID* mit gesetztem *FLAG_IS_ANSWER* enthalten, ist übrigens auch die Problematik einer Änderung in der Reihenfolge der empfangenen Pakete gelöst. Durch die operationID, und u.U. unter Zuhilfename der im Payload enthaltenen Informationen, ist jedes Antwortpaket nämlich der jeweiligen Anfrage eindeutig zuzuordnen.

Eine Übersicht der wichtigsten Befehle:

[21] *R*ich *C*lient *P*latform

Befehl	Payload	Richtung	Beschreibung
LOG	2+N	Robot→PC	Log-Nachricht (N Zeichen). Die Log-Nachricht enthält Informationen über das entsprechende Subsystem (z.B. Vision, Motoren, Verhalten, usw.) sowie den Schweregrad (Debug, Info, Warnung, Fehler), so daß FUremote bei Bedarf eine Filterung durchführen kann.
STATUS	64	Robot→PC	Status-Paket des Roboters mit Informationen über aktuell ausgeführte Szenarien, etc. (s.u.)
READYSET	13	PC→Robot	Setzt den Roboter wahlweise auf Spielzustand READY (Positionieren) oder SET. Hierbei werden auch alle für das Spiel nötigen Informationen mitgeliefert, u.a. die zu spielende Strategie (oder Rolle), die Farbe des gegnerischen Tores und des eigenen Teams.
STARTGAME	0	PC→Robot	Startet das Spiel.
STOPGAME	0	PC→Robot	Unterbricht das Spiel.
GET_STRATEGIES	N	PC↔Robot	Nach Anfrage übermittelt der Roboter eine Liste der verfügbaren Strategie-Szenarien.
GET_ROLES	N	PC↔Robot	Nach Anfrage übermittelt der Roboter eine Liste der verfügbaren Rollen-Szenarien.
GETMOTORSTATUS	11	PC↔Robot	Der Roboter sendet Informationen über den angefragten Motor (Temperatur, Volt, Torque, Position, Geschwindigkeit, usw.)
SETMOTOR	4	PC→Robot	Aktiviert (oder deaktiviert) einen Motor und setzt die Zielposition.
IMAGECAPTURE	4 / N	PC↔Robot	Nach Aufforderung von FUremote, z.B. alle X Sekunden ein Bild zu schicken, sendet der Roboter ein (verkleinertes) JPEG-Bild der aktuellen Kameraansicht (mit eingezeichneten Objekten).

Durch Einsatz der Befehle zum Abfragen der verfügbaren Strategien und Rollen ist es für *FUremote* möglich, unabhängig vom FUmanoid-Programm zu bleiben. Im Gegensatz zum bisherigen *ServerProgramm* würde also eine neue Strategie oder Rolle keine Anpassung auf PC-Seite erforderlich machen.

Für die Zukunft ist geplant, die Kommunikation während der Spiele auf das Protokoll des GameControllers umzustellen, da einerseits die Möglichkeit besteht das über kurz oder lang die Nutzung des GameControllers während eines Spieles vorgeschrieben wird und desweiteren durch das kontinuierliche Senden der Daten einem möglichen Verlust von UDP-Paketen entgegengewirkt wird.

Teamkommunikation

Die Kommunikation der Roboter untereinander erfolgt durch kontinuierliches Aussenden eines

Status-Paketes aller 500 Millisekunden. Dieses ist wie im vorigen Abschnitt beschrieben aufgebaut und beinhaltet unter anderem die folgenden Werte:

- ID des Roboters
- momentan laufende Szenarien in den verschiedenen Schichten
- Position des Roboters ((x,y)-Koordinate und Orientierung) aus der Selbstlokalisierung
- Position des Balles (absolut)
- geschätzter Batterieladestand
- Spielzustand, d.h. gegnerische Torfarbe, Teamfarbe, Anstoßinformationen und in welchem Stadium (READY, SET, PLAYING oder STOPPED) sich das Spiel befindet

Diese Daten werden unter anderem wie folgt eingesetzt:

- Informationen über die momentan laufenden Szenarien werden von der Strategieschicht dazu genutzt, die Rollen der Roboter zu verteilen. Anhand des gerade aktiven Verhalten-Szenarios kann ein Roboter in der Stürmer-Rolle zum Beispiel sehen, ob ein anderer Roboter aktuell im Ball-Besitz ist. In diesem Fall würde die Strategie ihm die Stürmer-Rolle entziehen, damit er dem ballführenden Roboter nicht in die Quere kommt.
- Da es während eines laufenden Spieles nicht erlaubt ist, von Seiten der Bediener an die Roboter Befehle/Daten zu senden, würde ein neu gestarteter Robot anfänglich keine Aktionen ausführen. Bei Spielen übernehmen die Roboter daher automatisch die Strategie und weitere Informationen der sich im Team befindlichen Roboter.
- Sollte ein Roboter den Ball nicht sehen, so kann er mithilfe der von seinen Teammitgliedern gesendeten Informationen zur Position des Balles eine Richtung festlegen, in welcher sich der Ball befindet. Dies setzt natürlich voraus, daß die Selbstlokalisierung entsprechend akkurat ist.
- Schlußendlich werden die gesendeten Daten dazu genutzt, in *FUremote* die Position der Roboter auf dem Spielfeld graphisch darzustellen, sowie ihre aktuellen Szenarien und den geschätzten Akkustand anzuzeigen.

4.7 Threads

Threads sind jetzt mit Hilfe der POSIX Threads implementiert und zwar in der Klasse *Thread*. Die Thread-Funktionalität ist damit komplett gekapselt.

Thread selber ist ein abstrakter Datentyp. Um einen Thread zu erstellen, muß der auszuführende Code in einer von Thread abgeleiteten Klasse in der Klassenmethode threadMain implementiert werden.

Beispiel:

```
#include "thread.h"
class BeispielThread : public Thread
{
protected:
   void threadMain() {
      // hier der im Thread auszuführende Code
   }
}
```

Die Kapselung der Thread-Funktionalität in einer Basisklasse hat den Vorteil, daß die eigentliche Implementierung komplett verborgen bleibt und im Bedarfsfall ersetzt werden kann. Desweiteren ist eine Parameterübergabe an Threads relativ einfach möglich, indem zum Beispiel dem Constructor die Parameter übergeben werden, welche dann in *threadMain* aufgegriffen werden können.

Zum Verwalten des Threads stehen folgende Klassenmethoden zur Verfügung:

- *run()* - Legt einen neuen Thread an und startet in diesem die *threadMain*()-Methode.
- *isRunning()* - Gibt *true* zurück, falls der Thread läuft (und laufen soll, s.u.).
- *wait()* - Wartet, optional mit Timeout, darauf, daß sich der Thread beendet.
- *cancel()* - Setzt ein Flag, welches dem Thread mitteilt das er sich beenden soll. Genaugenommen bedeutet dies, daß der in *threadMain()* enthaltene Code regelmäßig via *isRunning()* überprüfen muß, ob er sich beenden muß. Im Falle der Szenarien ist dies relativ trivial umsetzbar, da diese sowieso in einer großen *while()*-Schleife laufen.
- *setNiceness()* und *setRealTimePriority()* - setzt den *Nice*-Level des Threads oder schaltet in den Realtime-Scheduler mit entsprechender Priorität.

Bezüglich der Szenarien wird die gesamte Thread-Funktionalität umfassend durch die *switch...*-Methoden der *Robot*-Klasse behandelt.

Zusätzlich zu der *Thread*-Klasse existieren weitere auf der *pthread*-Bibliothek aufbauende Klassen:

- *Event* - Symbolisiert ein Ereignis, welches ausgelöst oder auf welches gewartet werden kann.
- *Mutex* - Diese Klasse repräsentiert eine Ressource, die nur von jeweils einer Partei genutzt werden kann.
- *CriticalSection* - Dies ist im Grunde ein Mutex und demzufolge von der *Mutex*-Klasse abgeleitet, welcher eingesetzt wird um bei Bedarf Code-Bereiche vor einer parallelen Ausführung zu schützen.

Die Klassen *Thread* und *Mutex* (sowie davon abgeleitete Klassen) enthalten jeweils Code, um Deadlocks zu erkennen und zu melden. Da während eines Spieles ein Deadlock in der Regel zur Folge hat, daß ein Roboter das Spiel einstellt, greift hier ein automatischer Timeout der dazu führt, daß bei erfolglosem Warten auf ein *Mutex::lock()* oder *Thread::wait()* nach zwanzig Sekunden der Roboter neu gestartet wird.

4.8 Sonstiges

Zusätzlich zu den allgemeinen Portierungsaufgaben wurde auch der gesamte Quellcode einheitlich formatiert und lange Dateien wurden nach Möglichkeit in mehrere aufgesplittet. So waren im alten Programm sämtliche Verhalten in einer Datei implementiert, während nun jede Verhaltens-Klasse in einer eigenen Datei abgelegt ist. Desweiteren wurden flächendeckend Kommentare eingeführt, hierbei wurde von *Doxygen*[22] Gebrauch gemacht.

Die im alten Programm vorhandenen globalen Variablen konnten eliminiert werden. Dies gelang durch Übergabe von Parametern an den Constructor einer zu instantiierenden Klasse, sowie durch Bereitstellung von öffentlich aufrufbaren Member-Funktionen der entsprechenden Szenarien. Hierbei können dann auch Rückgabewerte von Szenarien direkt dem entsprechenden Objekt entnommen werden und müssen nicht mehr global abgelegt werden.

Um desweiteren bei Änderung von Parametern eine Neukompilierung der Software zu vermeiden, wurde außerdem eine Konfigurationsdatei eingeführt. Diese beinhaltet z.B. die ID des Roboters, sowie Offsets für Kamera-Werte. Auch die Datei mit den Motoren-Offsets wird in ihr angegeben.

[22] www.doxygen.org

5 Installation und Kompilation

Zusätzlich zu der eigentlichen Portierung der Software gab es weitere Schwerpunkte im praktischen Teil der Studienarbeit. Hierbei handelte es sich zum einen um die Erstellung und die Pflege des Linux-Systems, zum anderen um die Unterstützung der anderen Teammitglieder durch die Bereitstellung einer Entwicklungsumgebung.

5.1 Erstellung des Linux-Images

Um die neue Software überhaupt auf den Robotern laufen zu lassen, mußte zunächst das Betriebssystem erstellt werden. Wie eingangs aufgeführt, wurde hierbei auf die bereits vorinstallierte OpenEmbedded-Distribution gesetzt. Das vorinstallierte und einzig von Gumstix, Inc. erhältliche Image basierte allerdings noch auf Kernel 2.6.21. Da für die Kamera aber das Quick-Capture-Interface des PXA270 genutzt werden sollte, mußte ein Update auf Kernel 2.6.28 durchgeführt werden, da erst hier die nötige Hardwareunterstützung implementiert war. Hierzu mußten die für den Gumstix nötigen Patches aktualisiert und neue Patches eingepflegt werden.

Desweiteren wurden spezielle, in der Standard-Distribution nicht enthaltene, Software-Pakete benötigt (z.B. OpenCV). Diese mußten in das Image eingefügt werden.

Im Laufe des Projektes wurden weitere Anpassungen vorgenommen, um z.B. Unterstützung für den Hardware-Watchdog (s.u.) und den Autostart der Software einzubauen.

5.2 Bereitstellung einer Entwicklungsumgebung

Damit die anderen Team-Mitglieder nicht jeweils eigenständig das gesamte OpenEmbedded-System kompilieren müssen (eine zeit- und speicherplatzaufwendige Aktion), wurde eine Binärversion des Cross-Compilers sowie ein SDK (Bibliotheken und Header-Dateien) nebst Installationsanleitung zusammen- und zur Verfügung gestellt. Desweiteren wurde für Nutzer ohne Linux-Betriebssystem eine virtuelle Maschine mit Linux-Betriebssystem erstellt, in welcher die *Eclipse*-Entwicklungsumgebung mit Compiler und SDK vorinstalliert wurde.

Zusätzlich wurde ein Installationsskript geschrieben, welches den Nutzer in die Lage versetzt, Aktionen einfach auf einem oder mehreren Robotern auszuführen, insbesondere Installation und Start/Stop der Software sowie Aktualisierung von Konfigurationsdateien.

5.3 Schnellstart und Watchdog

Zu Beginn der Entwicklung kam es häufiger vor, daß bei Stürzen des Roboters der Gumstix neu startete. Auch anderweitige Hardwareprobleme erzwangen gelegentliche Neustarts. So startete z.B. die Kamera gelegentlich nicht korrekt. Desweieren treten bei Wettkämpfen bis heute ungelöste Konflikte bei der Netzwerkkommunikation auf, die bei starkem Netzwerkverkehr und größeren Abstand ($>$ 1m) zum WLAN-Access Point zu einem Abschalten des WLAN-Modules führen. Zusätzlich gab es anfangs auch den gelegentlichen Segmentation Fault oder einfach nur ein ungeplantes Verhalten eines Roboters.

Während eines Spieles auftretende Abstürze oder ähnliche Situationen, in denen der Gumstix zwar noch läuft aber die FUmanoid-Software sich beendet hat, würden nach einer gewissen Verzögerung ein menschliches Eingreifen erforderlich machen. Dazu müßte der Roboter aus dem Spiel genommen werden und dürfte erst nach einer dreißigsekündigen Auszeit wieder am Spiel teilnehmen. Um diese Auszeiten zu minimieren, wurde auf den Watchdog des PXA270 zurückgegriffen. Bei einem Watchdog handelt es sich um Software oder Hardware, die bei Ausbleiben eines bestimmten Ereignisses einen automatischen Neustart des Systems veranlaßt. Im vorliegenden Fall

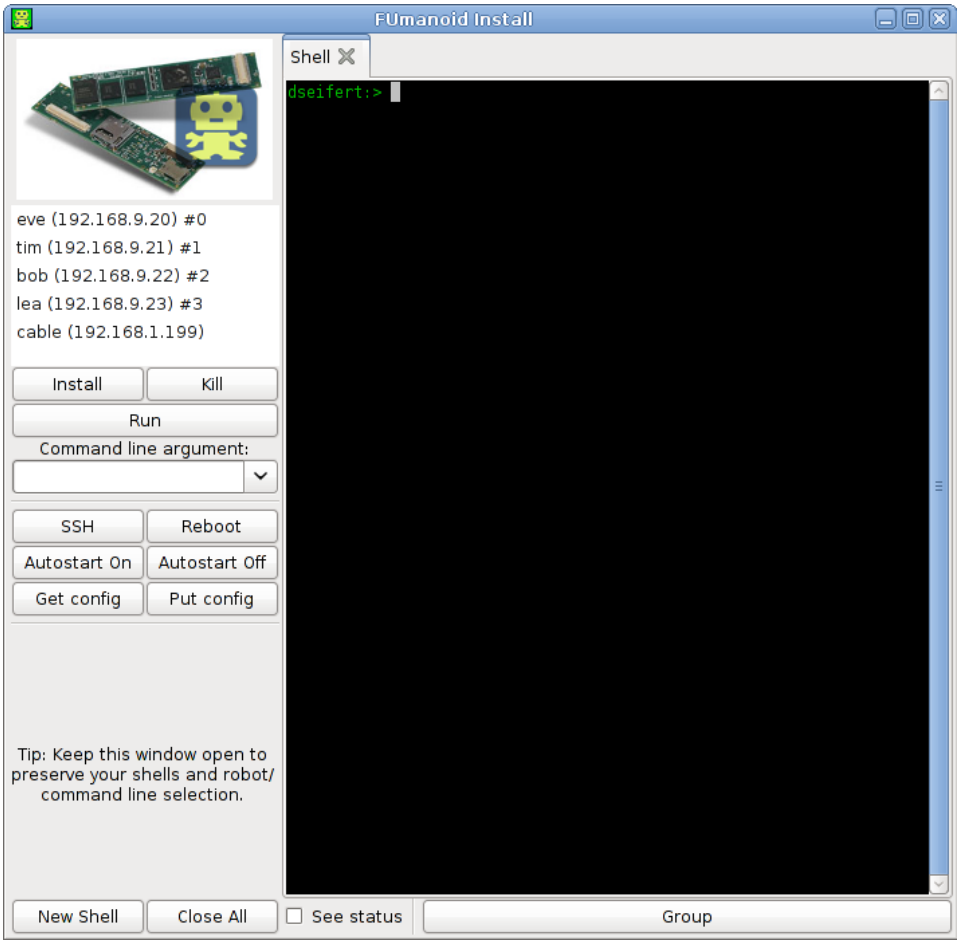

Abbildung 8: Installationsprogramm

war der Watchdog bereits im Prozessor integriert und durch den Linux-Kernel im Kernelmodul *sa1100_wdt* unterstützt. Beim Starten des FUmanoid-Programs wird der Watchdog aktiviert und ab diesem Zeitpunkt muß das FUmanoid-Programm spätestens nach drei Sekunden einen beliebigen Wert in das Device */dev/watchdog* schreiben. Verstreichen drei Sekunden, ohne daß dies passiert, führt der Prozessor automatisch einen Reset durch.

Ein Boot-Vergang des Gumstix dauerte allerdings zu Beginn der Entwicklung ca. 30 Sekunden (ohne Start des FUmanoid-Programmes). Dies war für Wettkämpfe denkbar schlecht. Zum einen sollte bei einem Entfernen des Roboters dieser nach 30 Sekunden wieder eingesetzt werden, desweiteren schreiben die Spielregeln vor, daß ein Roboter innerhalb von 20 Sekunden nach einem Sturz wieder stehen muß. Sollte sich bei dem Sturz der Gumstix neu gestartet haben, so wäre dies nicht schaffbar und es bestünde die Gefahr, daß der Schiedsrichter den betreffenden Roboter für 30 Sekunden aus dem Spiel entfernt.

Aus diesem Grund wurde auch der Bootprozess optimiert. Letztendlich gelang es, daß das FUmanoid-Programm nach einem Neustart nach etwa 13 Sekunden lief. Hierbei ist der eigentliche Bootvorgang bereits nach unter 10 Sekunden beendet. Die hierfür durchgeführten Aktionen waren:

- Entfernen der Wartezeit der Bootloader-Eingabeaufforderung durch ein neues U-Boot[23]-Skript (2s)

[23] *Das U-Boot* ist ein Boot-Loader für eingebettete Systeme - http://www.denx.de/wiki/U-Boot

- statisches Setzen des Kernel-lpj-Wertes (0,2s)
- Setzen der Kernelparameter *rootdelay=0 rootwait*, damit der Bootvorgang sofort fortgesetzt wird, wenn die SD-Karte verfügbar wird und nicht erst nach einer statischen Wartezeit (0,9s)
- Netzwerkverbindungen beim Booten parallel zu anderen Bootskripten aufbauen lassen
- unbenötigte Bootskripte gelöscht und zeitintensive Operationen beim Booten entfernt, bzw. nur einmal nach Neuinstallation ausführen lassen (z.B. *ldconfig*, *depmod* oder Konfiguration des sowieso unbenutzten Package-Managers)
- mit wenigen Ausnahmen alle Kernelmodule fest in den Kernel integriert, um das Nachladen von SD-Karte zu minimieren
- Minimierung der Größe des Linux-Kernels durch Entfernen von unbenutzten Treibern oder Funktionalität, um die für das Laden des Kernels von der (langsamen) SD-Karte benötigte Zeit zu minimieren
- Parallelisierung einiger Initialisierungen im FUmanoid-Programm

Während eine Reduktion der Bootzeit von dreißig auf zehn Sekunden schon eine beachtliche Geschwindigkeitssteigerung bedeutet, ist dies noch nicht vollkommen zufriedenstellend. Idealerweise sollten weitere Optimierungen durchgeführt werden, um diese Zeit noch weiter zu senken. Zum Beispiel wäre durch ein Wechsel von der glibc auf die uClibc[24] eine weitere Zeitersparnis von bis zu anderthalb Sekunden möglich[25].

6 Zukünftige Änderungen

Neben den im Rahmen dieser Studienarbeit durchgeführten Portierungs- und Verbesserungsarbeiten sind natürlich noch weitergehende Maßnahmen zur Verbesserung der FUmanoid-Software möglich, welche die Zeitvorgaben sowie den Rahmen der Studienarbeit überschritten hätten. Einige Ideen sollen in diesem Kapitel kurz angeschnitten werden.

6.1 Auslagerung Motorsteuerung

Wie in Abschnitt 4.5 auf Seite 13 ausgeführt, kann die Kommunikation mit den Motoren nur mit einer geringen Geschwindigkeit ausgeführt werden. Desweiteren ist aufgrund der Geschwindigkeitsoptimierungen und des Setzens der RTS-Leitung die Kommunikation mit dem RS485-Bus ein vergleichsweise ressourcenintensiver Prozess. Um die CPU-Last zu reduzieren und die Motorkommunikationsgeschwindigkeit zu erhöhen wäre es sinnvoll, die Motorsteuerung auszulagern.

Eine Option ist, den vorher im CM-5 eingesetzten ATMega128-Prozessor als Mikrocontroller für die Motoren zu verwenden. Der ATMega128 verfügt über einen RS485-Port und kann mit einer Geschwindigkeit von einem Mbaud mit den Motoren reden. Der hierzu nötige Code kann größtenteils aus der Software der alten Roboter übernommen werden. Desweiteren verfügt der ATMega128 auch über eine RS232-kompatible Schnittstelle, die bei Nutzung einer Oszillatorfrequenz von 7.3728 MHz mit 921,6 Kbaud betrieben werden könnte[26].

Selbst wenn der ATMega128 also nur als Proxy eingesetzt wird, d.h. die Datenpakete nur zwischen Gumstix und Motoren vermittelt, so wären trotz erhöhter Latenz durch das doppelte Senden

[24] Die uClibc ist eine insbesondere für eingebettete Systeme konzipierte, schlanke C-Bibliothek.

[25] Aufgrund von dadurch notwendig würdenden Änderungen am Programmcode und erneuten Tests wurde auf diese Umsetzung zum gegebenen Zeitpunkt verzichtet.

[26] [1, S. 195]

und Empfangen sowie durch die zusätzliche Verarbeitung im ATMega128 schätzungsweise eine Verdoppelung der Geschwindigkeit möglich.

Würde das Kommunikationsprotokoll zwischen Gumstix und ATMega128 geändert, um z.B. Lese- und Schreibbefehle für mehrere Motoren effizienter zu kodieren, so sind weitere Geschwindigkeitssteigerungen möglich. In der aktuellen Implementation muß z.B. jeder Motor seine aktuelle Position in einem 8 Byte großen Paket an den Gumstix schicken. Werden z.B. alle 22 Motoren mit dem ATMega128 abgefragt, könnte dieser die 22 Werte á 2 Byte anstelle in 176 Bytes in durchaus weniger als 50 Byte senden, indem die Werte vorab zusammengefaßt werden.

Zusätzlich wäre es interessant zu prüfen, ob ein auf diese Weise eingesetzter Mikrocontroller nicht pauschal alle Motoren abfragen kann ohne dabei die effektive Geschwindigkeit auf Gumstix-Seite zu beeinflussen. Sollte dies möglich sein, so könnte die auf dem Mikrocontroller laufende Software dahingehend erweitert werden, einerseits die Motoren auf Überhitzung und Überlastung zu kontrollieren, und andererseits Ausnahmezustände autonom zu erkennen und an den Gumstix weiterzuleiten. Letzteres könnte zum Beispiel bedeuten das ein Sturz des Roboters schneller erkannt werden kann.

6.2 Schnellere Reaktion auf Ereignisse

In der 2008-er und auch aktuellen Fassung der Software erhalten die Szenarien keine sofortigen Benachrichtigungen, wenn Ereignisse auftreten. Sämtliche Sensoren/Aktuatoren werden nur auf Anfrage der Software überprüft, so wird z.B. ein Sturz des Roboters erst dann erkannt, wenn der laufende Programmcode eine explizite Prüfung vornimmt. Aktuell passiert dies primär in den Verhalten, falls der Laufalgorithmus einen Sturz erkennt und den entsprechenden Statuscode setzt. Falls aber zum Beispiel in neuen Verhalten diese Abfrage vergessen wird oder anderweitig nicht oder nur selten zum Einsatz kommt, kann es schon mal passieren, daß sich der Roboter für eine gewisse Zeit selber aus dem Gefecht zieht.

Bei der Vision zeigt sich ein ähnliches Bild. Auch hier werden Ereignisse von den Szenarien explizit abgefragt. Die Verhalten rufen Funktionen des Vision-Objektes auf um Positionen von Ball, Tor, u.ä. zu erhalten. Es gibt einige Wartefunktionen, z.B. „warte maximal X Bildframes auf den Ball", aber während dieser ist natürlich das laufende Verhalten stillgelegt.

Eine mögliche Erweiterung der Architektur wäre also der Einsatz von unabhängigen Sensoren und einem stärker ereignisgesteuertem Programmablauf. Die Idee hierbei ist, daß erstens entweder durch externe Mikrocontroller (siehe vorigen Abschnitt zur Motoransteuerung) oder durch unabhängige Threads die permanent Sensoren und Aktuatoren überwachen, stets die aktuellen Sensorwerte im Speicher abrufbereit sind. Zweitens werden anhand der nun asynchron vorliegenden Sensorwerte regelmäßige Überprüfungen parallel zu den laufenden Verhalten möglich. Sollten hier bestimmte Bedingungen erfüllt sein, so wird ein Ereignis generiert, welches im Code angemessene Aktionen auslöst.

Als sehr einfaches Beispiel diene wieder der Sturz des Roboters. Lägen die Werte der Aktuatoren permanent vor, könnte ein Sturz unabhängig vom laufenden Verhalten erkannt werden. Dabei wird ein Ereignis gesendet, welches dann zum Beispiel dazu führt, daß das laufende Verhalten beendet und das Aufsteh-Verhalten gestartet wird.

Die Vorteile eines solchen Ansatzes liegen auf der Hand und sind unter anderem die Ersparnis von doppelten Programmcode um selbständig auf Ereignisse wie z.B. einen Sturz zu testen, und desweiteren eine signifikant schnellere Reaktion auf sich ändernde Situationen. Der Nachteil besteht potentiell darin, daß sich der Programmcode und insbesondere auch das Debuggen verkompliziert.

7 Fazit

Der im Rahmen dieser Studienarbeit portierte und erstellte Programmcode wurde seit Ende Februar 2009 für die Entwicklung der neuen FUmanoid-Software benutzt. Diese wurde im Juli erfolgreich im RoboCup 2009 in Graz eingesetzt, wo das Team den zweiten Platz und damit den Vizeweltmeistertitel in der Humanoid League (Kid-Size) errang.

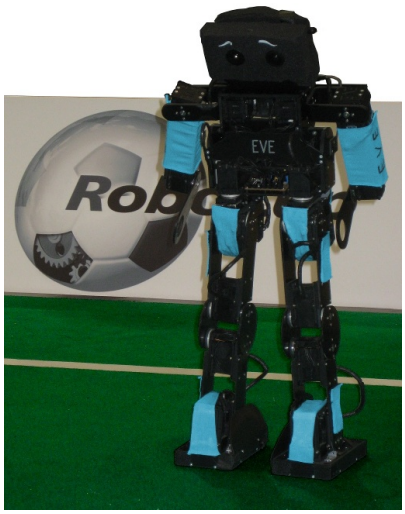

Abbildung 9: Roboter Eve beim RoboCup

Während die Portierung damit als abgeschlossen gelten kann, ist natürlich die Software und die Hardware noch nicht am Ende ihrer Möglichkeiten angelangt. Durch die im Abschnitt 6.1 vorgestellte Idee, die Motorsteuerung auf einen externen Mikrocontroller auszulagern, können wertvolle Ressourcen auf dem Gumstix gespart werden und gleichzeitig die Motoren schneller angesprochen werden, sowie schneller und besser auf Stürze reagiert werden. Durch weiteren Ausbau der Nutzung objektorientierter Prinzipien und Konstrukte kann auch die Arbeit am und mit dem Programmcode weiter vereinfacht werden.

Literatur

[1] Atmel Corporation: *Atmel AVR® ATmega 128*. Juli 2008 - http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf

[2] Intel: *Intel® PXA27x Processor Family - Developer's Manual* (Oktober 2004) - http://pubs.gumstix.org/documents/PXA%20Documentation/PXA270/-PXA270%20Processor%20Developer%27s%20Manual%20%5b280000-002%5d.pdf

[3] Brinkschulte, Uwe; Ungerer, Theo: *Mikrocontroller und Mikroprozessoren*. Springer, Berlin, 2. Auflage.

[4] Robotis: *Dynamixel AX-12 User's Manual*. 14.06.2006 - http://www.robotis.com/zbxe/dynamixel_en

[5] *ISO/IEC 9899:TC3* (hier als Comittee Draft, WG14/N1256) http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf

[6] Moballegh; Guilbourd; Hohl; Oertel; Rojas: *FUmanoid Team Description 2007/2008* http://www.fumanoids.de/wp-content/uploads/2009/07/fumanoids-tdp-2008.pdf

[7] Moballegh; Hohl, et al: *FUmanoid Team Description 2009* http://www.fumanoids.de/wp-content/uploads/2009/07/TDP2009Fumanoid.pdf

[8] Moballegh, Hamid Reza: Concurrent Scenario-Based Planning, a New Approach to Implement Highly Cooperated Multi-Agent Systems. (unveröffentlicht) https://dev.spline.de/trac/fumanoid/export/271/trunk/resources/Concurrent%20-Scenario.pdf