Fachgebiet Simulation, Systemoptimierung und Robotik
Fachbereich Informatik
Technische Universität Darmstadt

Modulare Weltmodellierung und Kommunikation in heterogenen Robotersystemen

Modular worldmodeling and communication in heterogeneous robotic systems

Diplomarbeit von
Marcus Schobbe & Patrick Stamm

Aufgabensteller:	Prof. Dr. Oskar von Stryk
Betreuer:	Dipl.-Inform. Dirk Thomas
	Dipl.-Inform. Sebastian Petters
Abgabetermin:	31.08.2007

Ehrenwörtliche Erklärung

Hiermit versichern wir, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, August 2007

____
Marcus Schobbe

Darmstadt, August 2007

____
Patrick Stamm

Kurzzusammenfassung

Das Fachgebiet Simulation, Systemoptimierung und Robotik (SIM) der Technischen Universität Darmstadt beschäftigt sich mit der Forschung an mobilen autonomen Robotersystemen. Damit sich unterschiedliche autonome Robotersysteme in ihrer Umwelt zurechtfinden und miteinander kommunizieren können, wird eine Weltmodellierung mit heterogener Teamkommunikation benötigt.

Ziel dieser Arbeit ist es, basierend auf dem Framework *RoboFrame* eine möglichst modulare Weltmodellierung zu entwickeln, die eine solche Kommunikation ermöglicht.

Die Implementierung der Arbeit wird als Teil des *DD-RoboCup-Softwarepakets* im Frühjahr 2007 bei den *Internationalen RoboCup GermanOpen*, *Internationalen RoboCup JapanOpen* und im Sommer 2007 bei der *RoboCup Weltmeisterschaft* in der *Humanoid League* eingesetzt.

Abstract

The Simulation, Systems Optimization and Robotics Group (SIM) of the Department of Computer Science of the Technische Universität Darmstadt concerns itself with the research of mobile autonomous robotic systems. So that different autonomous robotic system can orient and communicate with each other, a worldmodel and heterogeneous communication is required.

The intention of this work is to develop a modular worldmodeling based on the framework *RoboFrame*, that allows such a communication.

The implementation of the work will be integrated in the *DD-RoboCup-Software* that will be used at the competitions in spring 2007 at the *Internationalen RoboCup GermanOpen*, the *International RoboCup JapanOpen* and in summer 2007 at the *RoboCup World Championchip* in the *Humanoid League*.

Inhaltsverzeichnis

Abbildungsverzeichnis **v**

Tabellenverzeichnis **vii**

Listingverzeichnis **ix**

1 Einleitung **1**
1.1 Definition: Weltmodell 1
1.2 Definition: Heterogene Robotersysteme 2
1.3 Ziel der Arbeit 2
1.4 Aufbau der Arbeit 3
1.5 Quellcode 4
1.6 Code Conventions 4
1.7 Autoren 4

2 Motivation **5**

3 Grundlagen **7**
3.1 Stand der Forschung 7
3.2 Eingesetztes Robotersteuerungs-Framework 9
3.3 Existierende Weltmodellierungen und Kommunikationsmodule 10
3.3.1 DD 2006 10
3.3.2 GT 2006 12
3.3.3 Zusammenfassung 13
3.3.3.1 Weltmodellierung 13
3.3.3.2 Kommunikation 14
3.4 Existierende Hardwarearchitekturen 14
3.4.1 HR18 14
3.4.2 Weitere Plattformen 16
3.4.2.1 Pioneer 2-DX 16
3.4.2.2 HR27 16

4 Anforderungen **19**

5 Konzept **21**
5.1 Weltmodellierung 22
5.2 Kommunikation 23
5.3 Verhalten 25

6 Realisierung **27**
6.1 Weltmodellierung . . . 27
6.1.1 RoboFrame-Modul Weltmodellierung . . . 27
6.1.2 Schlüsselverwaltung . . . 29
6.1.2.1 Factories . . . 31
6.1.3 Zentrale Datenverwaltung . . . 32
6.1.3.1 Perzepte . . . 33
6.1.3.2 Modelle . . . 34
6.1.3.3 Teammodelle . . . 36
6.1.4 Modellierungsverwaltung . . . 38
6.1.4.1 Modellierungen . . . 40
6.1.5 Modellierungsfunktionen . . . 44
6.1.6 GUI-ModelViewer . . . 45
6.2 Kommunikation . . . 47
6.2.1 Kommunikationsverwaltung . . . 48
6.2.2 Datenverwaltung . . . 50
6.2.3 Kommunikation der Weltmodellstruktur . . . 51
6.2.4 Kommunikation der Weltmodellattribute . . . 53
6.3 Verhalten . . . 55
6.3.1 Symbole . . . 56
6.3.2 BasicBehaviors . . . 57

7 Ergebnisse **59**
7.1 Weltmodellierung . . . 59
7.1.1 Daten . . . 60
7.1.2 Integration von Modellierungen . . . 61
7.1.3 Effizienz . . . 61
7.1.4 Stabilität . . . 61
7.2 Kommunikation . . . 63
7.2.1 Heterogenität . . . 63
7.2.1.1 Kommunikation heterogener Modelle . . . 63
7.2.1.2 Kommunikation heterogener Weltmodelle . . . 65
7.2.2 Effizienz . . . 67
7.2.3 Stabilität . . . 69
7.3 Verhalten . . . 69
7.4 RoboCup Wettbewerbe 2007 . . . 69

8 Zusammenfassung und Ausblick **71**
8.1 Zusammenfassung . . . 71
8.2 Mögliche Erweiterungen . . . 72
8.2.1 Anpassungen für das Spiel 3 gegen 3 . . . 72
8.2.2 Anpassung der Kommunikation auf allgemeine Datentypen . . . 72
8.2.3 Filterung der kommunizierten Modelle im ModelViewer . . . 73
8.2.4 Verwendung der Modelle in der Selbstlokalisierung . . . 73
8.2.5 Erweiterung der Funktionsbibliothek der Weltmodellierung . . . 74

8.2.6 Transcoder für kommunizierte Modelle 74

A Verwendung der Weltmodellierung 75

A.1 Definition neuer Welt- und Sensorschlüssel 75

A.2 Registrierung neuer Perzepte 76

A.3 Hinzufügen neuer Modellierungen 76

A.4 Verwendung von Weltmodellfunktionen 79

A.5 Empfang und Verwendung von Teammodellen 79

A.6 Versendung von Daten 80

A.7 Weitergabe der Daten an die Verhaltenssteuerung 81

B Abkürzungen 85

C Literaturverzeichnis 87

Abbildungsverzeichnis

3.1 Minerva 7
3.2 Stanley 8
3.3 Sony AIBO Spiel der Four-Legged League 9
3.4 Konzept von DD 2006 11
3.5 Konzept von GT 2006 12
3.6 HR18: Bruno 15
3.7 HR18: Jan 15
3.8 Pioneer 2-DX 16
3.9 Sony AIBO und HR27 17

5.1 Übersicht über das Konzept der Weltmodellierung und Kommunikation 21
5.2 Konzept der Weltmodellierung mit den einzelnen Komponenten 23
5.3 Konzept der Kommunikation mit den einzelnen Komponenten 24
5.4 Kommunizierte Weltmodelle mit dem daraus resultierenden globalen Weltmodell 25
5.5 Konzept der Verhaltenssteuerung mit den einzelnen Komponenten. . . . 26

6.1 Lebenszyklus von Modulen 28
6.2 Integration der Weltmodellierung als Modul in eine RoboFrame Anwendung 29
6.3 Schlüssel der Weltmodellierung 29
6.4 Schlüsselverwaltung der Weltmodellierung 30
6.5 Factories zur Erstellung konkreter Datenklassen 31
6.6 Zentrale Datenverwaltung in der Weltmodellierung 33
6.7 Percepte in der zentralen Datenverwaltung 34
6.8 Schnittstelle für Modelle 35
6.9 Eingehende Modelle in der zentralen Datenverwaltung 35
6.10 Ausgehende Modelle in der zentralen Datenverwaltung 36
6.11 Teammodelle der Weltmodellierung 37
6.12 Eingehende Teammodelle in der zentralen Datenverwaltung 37
6.13 Ausgehende Teammodelle in der zentralen Datenverwaltung 38
6.14 Verwaltung und Steuerung der Modellierungen 38
6.15 Lebenszyklus von Modellierungen 39
6.16 Schnittstelle für Modellierungen 40
6.17 Verhaltensmodellierung zur Steuerung der Spielerrolle 41
6.18 Odometriemodellierung 43
6.19 Darstellung des Odometriemodells in der GUI 44

6.20 Statische Modellierungsfunktionen für Modellierungen 44
6.21 Anzeige von Modellen und Teammodellen im ModelViewer 46
6.22 Anzeige eines Teammodells im ModelViewer 47
6.23 Auswahl der Anzeige der Modelle 47
6.24 Auswahl der Anzeige der Teammodelle 47
6.25 Schnittstelle der Kommunikation 49
6.26 Streamable zur Steuerung der Kommunikation über die GUI 50
6.27 CommunicationData 51
6.28 Streamable zur Kommunikation von Weltmodellstrukturen 52
6.29 Datenpaket zur Kommunikation des lokalen Weltmodells 52
6.30 Ablauf bei der Kommunikation der Weltmodellstrukturen dreier Roboter 53
6.31 Streamable zur Kommunikation von Weltmodellen 54
6.32 Datenpaket zur Kommunikation der Modelldaten des globalen Weltmodells 54
6.33 Verhaltenssteuerung mit XabslControl 55
6.34 BallSymbol als Verhaltenssymbol 56
6.35 MotionProvider zur Ansteuerung der Bewegungsbefehle 57
6.36 GoToBall als BasicBehavior 58

7.1 Visualisierung heterogener Modelle 64
7.2 Visualisierung heterogener Weltmodelle 65
7.3 Ausschnitt der lokalen Weltmodelldaten 66
7.4 Teammodelle von Roboter 2 66
7.5 Teammodelle von Roboter 3 67
7.6 Beispielhafte Darstellung einer Netzwerkkommunikation der aktuellen RoboCup Anwendung 68

Tabellenverzeichnis

3.1 Bewertung existierender Weltmodellierungen für Robotersysteme . . . 13
3.2 Bewertung existierender Kommunikationsmodule für Robotersysteme . 14

Listingverzeichnis

A.1 WorldModel07Keys.cpp Schlüsseldefinition 75
A.2 KeyManager07.cpp Perzeptregistrierung 76
A.3 WorldModelModule07.cpp Modellierungsregistrierung 77
A.4 BallModeling.cpp Modellierungserstellung 78
A.5 KeyManager07.cpp Modellregistrierung 78
A.6 ModelingOperations.cpp Modellierungsfunktionen 79
A.7 KeyManager07.cpp Teammodellregistrierung 80
A.8 BallModel.cpp Modellerstellung 80
A.9 BehaviorControl.cpp Verhaltensanbindung 82

1 Einleitung

1.1 Definition: Weltmodell

Als *Weltmodell* werden in der Robotik die roboterspezifischen Daten und die den Roboter umgebene Umweltsituation bezeichnet. Die eigenen Daten beinhalten die eigene Position in der Welt sowie verschiedene Zustände, die der Roboter annehmen kann, aber auch Charakteristika über die eigene Beschaffenheit wie Ausdehnung, Kinematik und Sensorik. Als Umwelt werden alle Objekte bezeichnet, die der Roboter mit seinen Sensoren wahrnehmen kann. Da man Eigenschaften eines Objektes oft von mehr als einem Sensor erhält, müssen diese Informationen bewertet werden und gehen gewichtet in die Darstellung dieses Objektes ein. Daher wird die Umweltrepräsentation des Weltmodells auch als Multisensorfusion bezeichnet [13].

Die interne Repräsentation der Umwelt kann metrisch oder topologisch sein. Ein metrisches Weltmodell bietet die Möglichkeit, zu allen Objekte eine Position in Weltkoordinaten anzugeben. Bei der topologischen Variante werden lediglich Beziehungen zwischen einzelnen Objekten dargestellt; Informationen über deren Distanz oder Richtung sind nicht vorhanden. Beim Roboter-Fußball wird das metrische Modell benutzt, da die bei der topologischen Darstellung fehlenden Eigenschaften von zentraler Bedeutung sind.

Des Weiteren lässt sich die Umwelt anhand verschiedener Charakteristika unterteilen. Die für die Robotik relevanten Kriterien sind die Bekanntheit und die Dynamik der Objekte. Unter Bekanntheit versteht man in diesem Zusammenhang, ob Informationen über Objekte vorhanden sind, die über die geometrische Beschaffenheit und Ausdehnung hinausgehen, wie die Position oder die Anzahl des jeweiligen Objekts. Von einer dynamischen Umwelt wird ausgegangen, sobald mindestens ein Objekt keine feste Position besitzt. Im Roboter-Fußball wird von einer dynamischen und unbekannten Umwelt ausgegangen, da der Ball sowie die Roboter sich auf dem Spielfeld bewegen und somit keine festgelegte Position besitzen.

Das Weltmodell wird im Folgenden als Produkt der *Weltmodellierung* verstanden, welche aus den gewichteten Darstellungen der von den Sensoren erkannten Objekten besteht. Die Weltmodellierung umfasst sämtliche *Modellierungen*, die einerseits die Sensordaten eines Objekttyps fusionieren und filtern und andererseits die eigene Position sowie den eigenen Zustand berechnen. Diese Ergebnisse werden jeweils als *Modell* bezeichnet und bilden zusammengenommen das Weltmodell des Roboters.

1.2 Definition: Heterogene Robotersysteme

Robotersysteme, die sich in ihrem Aufbau, der Steuerung, Sensorik, Aktuatorik oder der Software unterscheiden, werden als heterogen bezeichnet. Homogene Robotersysteme gleichen sich in allen aufgeführten Eigenschaften [15].

Im Rahmen des "RoboCup" werden meist homogene oder schwach heterogene Roboter eingesetzt, da für gleichartige Systeme eine geringere Entwicklungs- und Testzeit benötigt wird, wodurch die Roboterteams in ihrem Gesamtsystem (Hard- und Software) stabiler sind.

Um mit gemischten Teams, wie beispielsweise einem kooperativen Team zwischen den Darmstadt Dribblers (DD) und den Darmstadt Dribbling Dackels (DDD), spielen zu können, ist eine heterogene Kommunikation nötig, damit sich die Mannschaftsspieler über die jeweilige Umweltsituation austauschen und – falls möglich – gemeinsam Spielzüge planen können.

Die Vorteile solcher heterogener Systeme liegen im Zusammenspiel unterschiedlicher Eigenschaften der einzelnen Roboter, da sich deren Sensoren und Aktuatoren gegenseitig ergänzen können und sich dadurch die Möglichkeit ergibt, komplexere Aufgaben zu lösen. In unserem alltäglichen Leben werden Roboter in Zukunft einen immer größeren Stellenwert einnehmen und zur Bewältigung verschiedenster Aufgaben eingesetzt. Da diese Roboter von verschiedenen Herstellern produziert und für verschiedene Aufgaben eingesetzt werden, ist es wichtig, dass diese heterogenen Systeme miteinander kommunizieren können und sich kooperativ verhalten, um ihren Einsatzbereich so flexibel und umfangreich wie möglich zu halten.

Ein weiteres Einsatzgebiet heterogener Robotersysteme liegt in der Kartografierung. Die einzelnen Roboter können einerseits verschiedene Teile ihrer Umgebung erfassen und diese Informationen an die anderen Robotersysteme kommunizieren, sodass jeder Roboter zu jeder Zeit eine komplette Karte des bisher erkundeten Gebietes besitzt. Andererseits besteht die Möglichkeit, ein Teilstück gemeinsam zu erkunden, verschiedenste Ebenen oder Eigenschaften der Umweltobjekte zu kommunizieren und auf jedem Roboter zu dem Weltmodell zu fusionieren.

1.3 Ziel der Arbeit

Ziel dieser Arbeit ist die Entwicklung einer objektorientierten Weltmodellierung sowie einer Kommunikationsschnittstelle zwischen heterogenen Robotersystemen. Dies beinhaltet die Entwicklung einer Softwarekomponente, im Folgenden *Weltmodellierung* genannt, welche das Hinzufügen beziehungsweise Entfernen von Modellierungen ermöglicht. In einem zweiten Schritt soll eine Kommunikationsschnittstelle entstehen, die einen effizienten Datenaustausch zwischen heterogenen Robotersystemen zulässt.

Um das Debuggen der eigenen und kommunizierten Weltmodelldaten zu ermögli-

chen, ist die bestehende GUI bezüglich des neuen Weltmodells anzupassen und zwecks der Anzeige der kommunizierten Daten zu erweitern.

Im Rahmen dieser Arbeit soll außerdem eine neue Instanz des RoboCup Codes – RoboCup07 – angelegt werden, die als Grundlage der Wettbewerbe im Jahr 2007 dienen wird. In dieser sind die Weltmodellierung und die Kommunikation zu integrieren und die bestehenden Elemente auf die neuen Komponenten anzupassen.

1.4 Aufbau der Arbeit

Diese Arbeit dokumentiert die Architektur- und Entwicklungsentscheidungen und dient als Handbuch zur Verwendung des Weltmodells mit der Kommunikation.

Zum besseren Überblick werden im Folgenden die Inhalte der weiteren Kapitel kurz dargestellt:

Kapitel 2 Beweggründe für die Erstellung einer Weltmodellierung und einer heterogenen Kommunikation.

Kapitel 3 Analyse der aktuellen Forschung und Entwicklung im Hinblick auf die in dieser Arbeit angesprochenen Themenbereiche sowie die einsetzbaren Roboterarchitekturen.

Kapitel 4 Anforderungen an die Softwarekomponenten dieser Arbeit.

Kapitel 5 Beschreibung des konzeptionellen Aufbaus der Weltmodellierung, Kommunikation und der Schnittstelle zum Verhalten.

Kapitel 6 Detaillierte Vorstellung der in dieser Arbeit realisierten Softwarekomponenten.

Kapitel 7 Beschreibung der mithilfe der entwickelten Anwendung gewonnenen Funktionalität.

Kapitel 8 Zusammenfassung der wichtigsten Ergebnisse dieser Arbeit mit einem Ausblick auf weiterführende Entwicklungen.

Anhang A Einbindung der neu entstandenen Softwarekomponenten in ein Robotersteuerungssystem.

Anhang B Liste der verwendeten Abkürzungen.

Anhang C Literaturverzeichnis.

Bei der Verwendung dieser Arbeit als Handbuch zur Integration der Weltmodellierung und Kommunikation in eine bestehende Robotersteuerungssoftware wird die Lektüre von Kapitel 5 und Anhang A empfohlen. Die API-Dokumentation ergänzt diese Arbeit um Detailinformationen zur Implementation und Schnittstellenverwendung.

1.5 Quellcode

Der im Rahmen dieser Arbeit entstandene Quellcode liegt auf einer CD bei und ist im aktuellen Repository der Darmstadt Dribblers zu finden. Auf der beiliegenden CD befindet sich außerdem die komplette Dokumentation des Quellcodes. Im aktuellen Repository ist diese mittels Doxygen[1] erstellbar.

1.6 Code Conventions

Bei der Entwicklung des Programmiercodes wurden die "Coding Standards für die Software-Entwicklung am Fachgebiet SIM" [6], zu finden im aktuellen Wiki[2] der Darmstadt Dribblers, verwendet. Mit dem Tool Uncrustify[3] wurde eine automatische Quellcodeformatierung gemäß der Coding Standards durchgeführt.

1.7 Autoren

Da diese Diplomarbeit zu zweit erstellt wurde, gilt es darzulegen, welchem Autor die einzelnen Teilabschnitte zuzuordnen sind.

Marcus Schobbe hat Abschnitt 3.3, Kapitel 4, Abschnitt 5.2, Abschnitt 5.3, Abschnitt 6.2, Abschnitt 6.3, Abschnitt 7.2, Abschnitt 7.3, Kapitel 8 und Anhang A, Patrick Stamm Kapitel 2, Abschnitt 3.1, Abschnitt 3.4, Abschnitt 5.1, Abschnitt 6.1 und Abschnitt 7.1 geschrieben. Die restlichen Teile wurden gemeinsam verfasst.

[1] Doxygen – Webseite: `http://www.doxygen.org/`
[2] SIM-Wiki – Webseite: `https://www.sim.informatik.tu-darmstadt.de/wiki/`
[3] Uncrustify – Webseite: `http://uncrustify.sourceforge.net`

2 Motivation

Die Entwicklung autonomer Robotersysteme lässt sich auf oberster Ebene in die beiden Bereiche Software und Hardware unterteilen, die optimal aufeinander abzustimmen sind. Um dies zu erreichen, sind in beiden Bereichen potentielle Problemstellen zu erkennen und bezüglich einer möglichen Optimierung zu analysieren.

Die derzeitigen Begrenzungen der Leistungsfähigkeit der Hardware basieren einerseits auf einer beschränkten Kapazität der Akkumulatoren (Akkus), andererseits auf der begrenzten Leistung der Motoren in den Gelenken. Dies hat zur Folge, dass die Robotersysteme möglichst energiesparsam und leicht gebaut werden müssen. Die daraus resultierende, eher geringe Rechenkapazität zwingt die Entwickler der Robotersteuerungssoftware zu einer effizienten Programmierung.

Die Leistungsfähigkeit Software wird durch nicht vorhandene Funktionalitäten und durch unzureichende Korrektheit eingeschränkt. Die Erfahrungen des GermanTeams (GT) beim RoboCup[1] in der Sony Four-Legged Robot League[2] zeigen, dass einem modularen Aufbau der Komponenten große Bedeutung zu kommt, da die dadurch gewonnene Flexibilität das Arbeiten in großen Teams sowie das Erweitern und Vergleichen einzelner Komponenten stark vereinfacht. Um einzelne Komponenten bezüglich ihrer Korrektheit analysieren zu können, ist es wichtig, geeignete Debugmöglichkeiten zur Verfügung zu stellen.

Neben der Optimierung von Soft- und Hardware zur Lösung von Aufgabenstellungen innerhalb eines Robotersystems muss auch der Aspekt der Roboterkooperation zur Bewältigung komplexer Problemstellungen beachtet werden. Damit koordinierte Zusammenarbeit möglich ist, müssen die Daten der einzelnen Roboter über eine einheitliche Kommunikationsschnittstelle ausgetauscht werden. Da sich die am Fachgebiet SIM eingesetzten Roboterarchitekturen bezüglich ihrer Hard- und Software unterscheiden, wird eine heterogene Kommunikationsschnittstelle benötigt.

Motivation für diese Arbeit ist die Entwicklung einer Weltmodellierung, die eine Interaktion zwischen den unterschiedlichen Modulen sowie eine heterogene Kommunikation ermöglicht. Die Kernkomponenten der Kommunikation sollen derart variabel konzipiert sein, dass keinerlei Anpassungen nach der Veränderung einer Softwarekomponente nötig sind. Bei der Entwicklung der Weltmodellierung ist ein modularer Aufbau der einzelnen Modellierungen von zentraler Bedeutung, sodass diese mit geringem Aufwand an- und abgekoppelt werden können. Um die unkomplizierte Erweiterbarkeit der Modellierungen zu gewährleisten, muss die Bereitstellung der benötigten Daten mög-

[1] RoboCup – Webseite: `http://www.robocup.de/`
[2] Sony Four-Legged Robot League – Webseite: `http://www.tzi.de/4legged/`

lichst intuitiv sein. Abschließend wird eine Schnittstelle für die Verhaltenssteuerung entwickelt, welche anhand der aktuellen Weltdaten situativ Entscheidungen trifft.

3 Grundlagen

In diesem Kapitel werden die Voraussetzungen für die Entwicklung einer Weltmodellierung und Kommunikation beschrieben. Dabei wird zunächst auf den Stand der Forschung bei Robotersystemen sowie speziell auf deren verschiedene Aufgabenbereiche und die für diese Arbeit resultierenden Problemstellungen eingegangen. Die folgenden Abschnitte behandeln die für die Software relevanten Grundlagen, wie das eingesetzte Robotersteuerungs-Framework, die Analyse bekannter Weltmodellierungen sowie eine Auswahl von Hardwarearchitekturen des Fachgebiets SIM, auf denen diese Arbeit eingesetzt werden kann.

3.1 Stand der Forschung

Roboter, die selbständig oder mit anderen Robotern komplexe Aufgaben lösen, benötigen eine Weltmodellierung, damit auf Basis dieser Daten eine geeignete Planung und Steuerung vorgenommen werden kann. Ein frühes Beispiel eines solchen autonomen Roboters, der auf seine Umwelt reagiert und diese mit in die Planung seiner Aufgabenlösung einbezieht, ist der Radroboter Minerva (siehe Abbildung 3.1), ein Projekt der Universität Bonn und der Carnegie Mellon Universität. Dieser 1998 entwickelte Roboter führt Menschen durch das Smithsonian's National Museum of American History, weicht dabei um ihn herumstehenden Gegenständen und Besuchern aus und reagiert auf das Interesse beziehungsweise Desinteresse der Besucher [23].

Abbildung 3.1: Minerva (Quelle [5])

Die Entwicklungen solcher, autark agierenden Systeme sind in der momentanen

Forschung am häufigsten im kommerziellen Bereich vertreten. Diese "Stand-alone"-Maschinen sind meist nur in der Lage, einen Aufgabenbereich abzudecken. Die Erweiterung dieses Aufgabenbereichs kann ausschließlich durch eine Funktionsanpassung des Roboters erreicht werden [10]. Ein aktuelles Beispiel für solche Robotersysteme sind die Teilnehmer der Grand Challenge der Defense Advanced Research Projects Agency[1], die 2004 erstmalig ausgetragen wurde. Schon im zweiten Jahr dieses Wettkampfes gelang es fünf autonomen Fahrzeugen, eine 212 km lange Strecke durch die Mojave-Wüste zu durchfahren. Gewonnen hatte der VW Touareg Stanley (siehe Abbildung 3.2) der Universität Standford. Durch die Erweiterung der teilnehmenden Fahrzeuge mit verschiedenen Sensoren, die die Umwelt erfassen, wurde ein autonomes Manövrieren durch ein unbekanntes Gelände ermöglicht.

Abbildung 3.2: Stanley (Quelle [1])

Zukünftig werden Roboter entwickelt, die immer komplexere Aufgaben lösen können. Es wird jedoch kein System geben, welches jeder Aufgabe gewachsen ist. Aufgrund der Komplexität dieser Maschinen verursachen sie einen hohen Entwicklungs- und Kostenaufwand. Da für die Lösung einer Aufgabe mit kooperierenden Systemen weitaus weniger komplexe Roboter benötigt werden, können komplizierte Aufgaben mit Roboterteams ebenso effizient, jedoch kostengünstiger bewältigt werden. Solche sogenannten *Multi-Roboter-Systeme (MRS)* sind bezüglich ihres Aufgabenspektrums flexibler, da eine Erweiterung der kooperativ agierenden Systeme mit neuen Robotern die Lösungsmöglichkeiten vervielfältigt. Des Weiteren bieten solche Roboterteams eine erhöhte Ausfallsicherheit, da Sensorausfälle auf einem Robotersystem durch andere Roboter kompensiert werden können [11]. Aus diesem Grund ist es wichtig, dass Roboter untereinander interaktiv zusammenarbeiten können.

Multi-Roboter-Systeme haben den Vorteil, ihre kognitiven, motorischen und konstruktionsbedingten Fähigkeiten zur Lösung einer Aufgabe zu kombinieren und dadurch ein Höchstmaß an Effizienz zu erlangen. Im RoboCup gibt es viele Beispiele für Roboter, die in einem Team gemeinsam eine Aufgabe lösen, indem sie ihre kognitiven Fähigkeiten verbinden. In der Four-Legged League spielen vier homogene Sony AIBOs (siehe Abbildung 3.3) in einem Team, verständigen sich über die derzeitige Umwelt- und Robotersituation über WLan und sind dadurch in der Lage gemeinsam Spielzüge durchzuführen.

[1] Grand Challenge der Defense Advanced Research Projects Agency – Webseite: `http://www.darpa.mil/grandchallenge/`

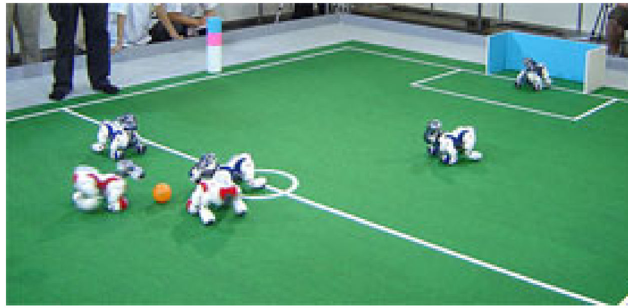

Abbildung 3.3: Sony AIBO Spiel der Four-Legged League (Quelle [21])

Die Entwicklung der Wirtschaft zeigt, dass von verschiedenen Firmen Systeme entwickelt werden, die für unterschiedliche Aufgaben unterschiedlich gut geeignet sind. Des Weiteren entwickelt die Firmen ihre Robotersysteme fortlaufend weiter, sodass in schnellen Zyklen in Soft- und Hardware veränderte Produkte auf den Markt kommen werden. Damit dieser Pool an heterogenen Systemen kooperativ komplexe Aufgaben lösen kann, kommt einer gut funktionierenden Kommunikation der Welt- und Roboterdaten eine immer größere Bedeutung zu, sodass die jeweilige Situation und die darauf aufbauenden Planungsschritte ausgetauscht und aufeinander abgestimmt werden können.

3.2 Eingesetztes Robotersteuerungs-Framework

Das dieser Arbeit zugrundeliegende Framework *RoboFrame* bietet die Möglichkeit, Steuerungsprogramme für mobile autonome Robotersysteme zu entwickeln, und entstand ebenfalls am Fachgebiet SIM im Rahmen einer Diplomarbeit [16]. Durch eine Plattformabstraktionsschicht konnte das Framework plattformunabhängig implementiert werden, sodass sich Steuerungsanwendungen unabhängig von dieser Problematik realisieren lassen und Anpassungen bei einem Plattformwechsel nur in den Klassen dieser vorgelagerten Schicht vorgenommen werden müssen.

Die Funktionalitäten dieser Steuerungsanwendungen, wie das Verhalten oder die Bewegungserzeugung, werden in Modulen gekapselt und in der Hauptsteuerungsklasse *Application* verschiedenen Prozessen hinzugefügt. Die Prozesse, wie der *Motion*- und der *Cognition-Prozess*, laufen in eigenen *Threads*, wodurch sich unterschiedliche Aufruffrequenzen und -prioritäten realisieren lassen.

Für den Datentransfer zwischen den einzelnen Modulen stellt das Framework die Klassen *In*- und *OutBuffer* zur Verfügung, die einerseits für den Datenempfang von anderen Modulen und andererseits für die Datenbereitstellung zuständig sind. Um auch mit anderen Robotern sowie der GUI Daten austauschen zu können, werden von RoboFrame die Stream-Klassen bereitgestellt, die eine Serialisierung von Objekten ermöglicht, um diese über das Netzwerk transportieren zu können. Damit solche Objekte mithilfe des Frameworks serialisiert und später deserialisiert werden können, müssen diese

von der Klasse *IStreamable* abgeleitet werden, um dann über einen *SocketConnector*, der die Schnittstelle zwischen dem Framework und dem Netzwerk darstellt, an andere Instanzen der Steuerungssoftware gesendet werden zu können.

Mithilfe der RoboGUI, die ebenfalls von RoboFrame bereitgestellt wird, lassen sich die versendeten Daten in entsprechenden, dafür entwickelten Dialogen darstellen, womit die Korrektheit einzelner Module überprüft werden kann.

3.3 Existierende Weltmodellierungen und Kommunikationsmodule

In den folgenden Abschnitten werden eingesetzte Weltmodellierungen und Kommunikationsmodule analysiert und mithilfe der in der Softwareentwicklung relevanten Qualitätskriterien bezüglich ihrer Anwendungsmöglichkeiten für die in dieser Arbeit betrachteten Einsatzgebiete beurteilt. Die in der Zusammenfassung aufgeführten Tabellen bieten einen Überblick über die gewonnenen Erkenntnisse.

3.3.1 DD 2006

Die Weltmodellierung der Darmstadt Dribblers (DD) (siehe Abbildung 3.4) besteht im Wesentlichen aus einer Klasse, dem *Worldmodelling*. In diesem Modul werden alle Darstellungen der über die Sensoren erkannten Objekte berechnet. Diese Darstellungen werden in der Datenklasse *World Model* gespeichert und den darunterliegenden Schichten zur Verfügung gestellt. Ausnahmen bilden die Bestimmung der eigenen Position, die Selbstlokalisierung sowie die Ballmodellierung, deren Berechnung in eigene Module ausgelagert ist.

Die Berechnungen der soeben genannten Komponenten basiert auf den Daten der verschiedenen Sensoren, den *Percepts*. Jedes Perzept speichert Informationen über ein bestimmtes Objekt, das von den Sensoren, beispielsweise einer Kamera, erkannt wird.

Neben der Erzeugung des World Models steuert Worldmodelling die Teamkommunikation, welche die eigenen Darstellungen mit den Darstellungen der Teammitglieder zusammenfasst. Dabei wird die Datenklasse World Model unter den Robotern ausgetauscht, wodurch nur homogene Systeme miteinander kommunizieren können.

Ein Vorteil dieses Konzepts ist die Informationsbündelung an einer zentralen Stelle, wodurch eine einfache Teamkommunikation möglich ist, da zur Übermittlung der kompletten Welt eines Roboters nur das Datenobjekt World Model ausgetauscht werden muss. Ein Nachteil dieser Kommunikation liegt in der fehlenden Flexibilität, da immer das komplette Datenpaket verschickt werden muss, auch wenn Roboter nur an Teilen der Welt interessiert sind, was ineffizient ist und ein unnötig hohes Datenaufkommen im WLan-Netz verursacht. Außerdem ist durch den Austausch von unterschiedlichen

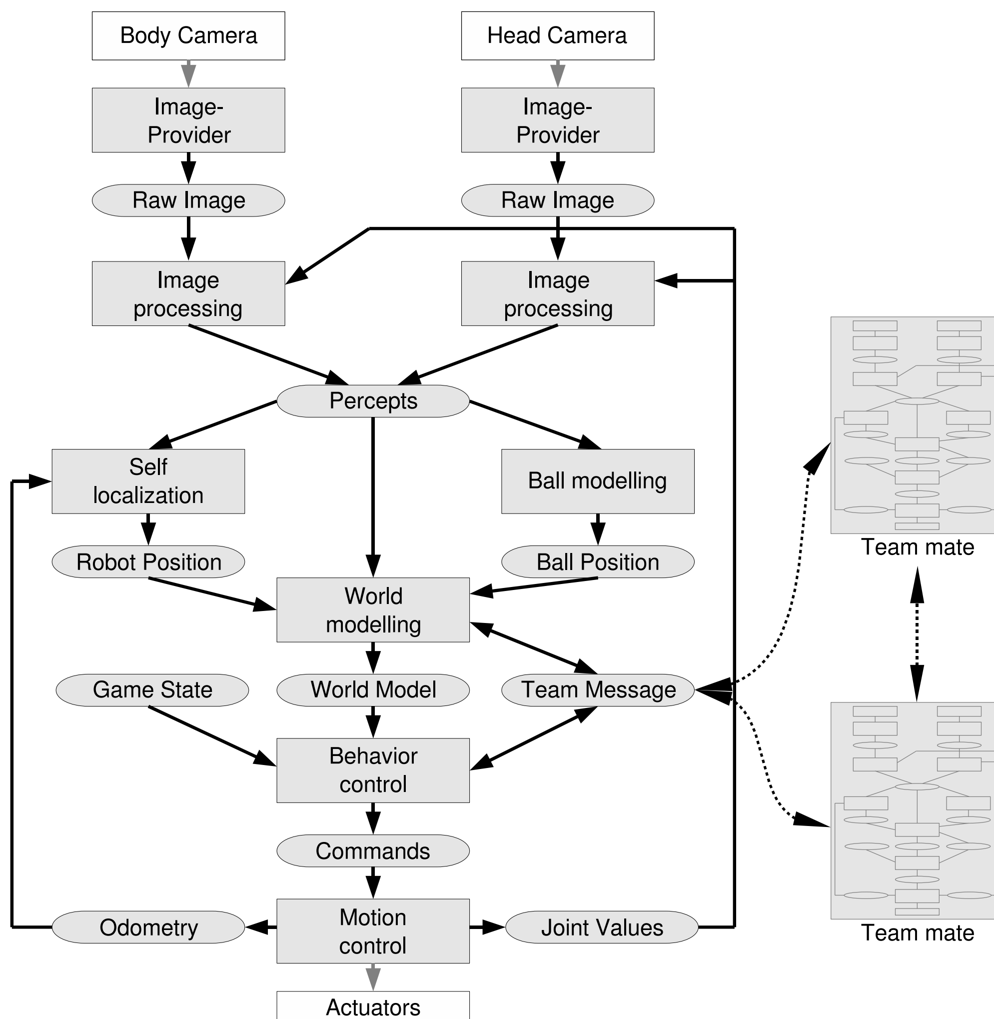

Abbildung 3.4: Konzept von DD 2006 (Quelle [7])

Versionen des Datenpakets World Model die Lauffähigkeit der Anwendung nicht weiter gewährleistet.

Wie beschrieben kapselt die Weltmodellierung den Großteil der Berechnung in einer zentralen Klasse, wodurch Erweiterungen dieser Komponente mit Filterungen neuer Objekt erschwert werden. Dies führt dazu, dass sie durch Weiterentwicklungen schlecht lesbar und durch die schwierige Wartbarkeit immer fehleranfälliger wird.

Durch die Entwicklung des “ModelViewer“-Dialogs ist eine gute Testbarkeit der gefilterten Objekte in Worldmodelling gewährleistet, sodass mögliche Fehler lokalisiert werden können. Mit der Anzeige dieser Objekte und der für ihre Berechnung benötigten Perzepte auf einem Spielfeld bietet der “ModelViewer“ eine intuitive Debugmöglichkeit. Das von anderen Robotern übermittelte World Model ist in diesem Dialog nicht darstellbar, sodass lediglich die lokalen Weltdaten testbar sind.

3.3.2 GT 2006

Die Weltmodellierung des GermanTeams (GT) (siehe Abbildung 3.5) besteht aus vielen einzelnen Modellierungen, im Folgenden Locator genannt, und ist modular aufgebaut. Als Eingabe erhalten diese Locator ein Perzept, in dem die Sensordaten über das zu betrachtende Objekte gespeichert sind und von jeweils einem Perzeptor erzeugt und bereitgestellt werden. Ein Locator modelliert das jeweilige Perzept und erzeugt seinerseits eine gefilterte Darstellung des Objekts, das im Verhaltensmodul weiterverwendet wird.

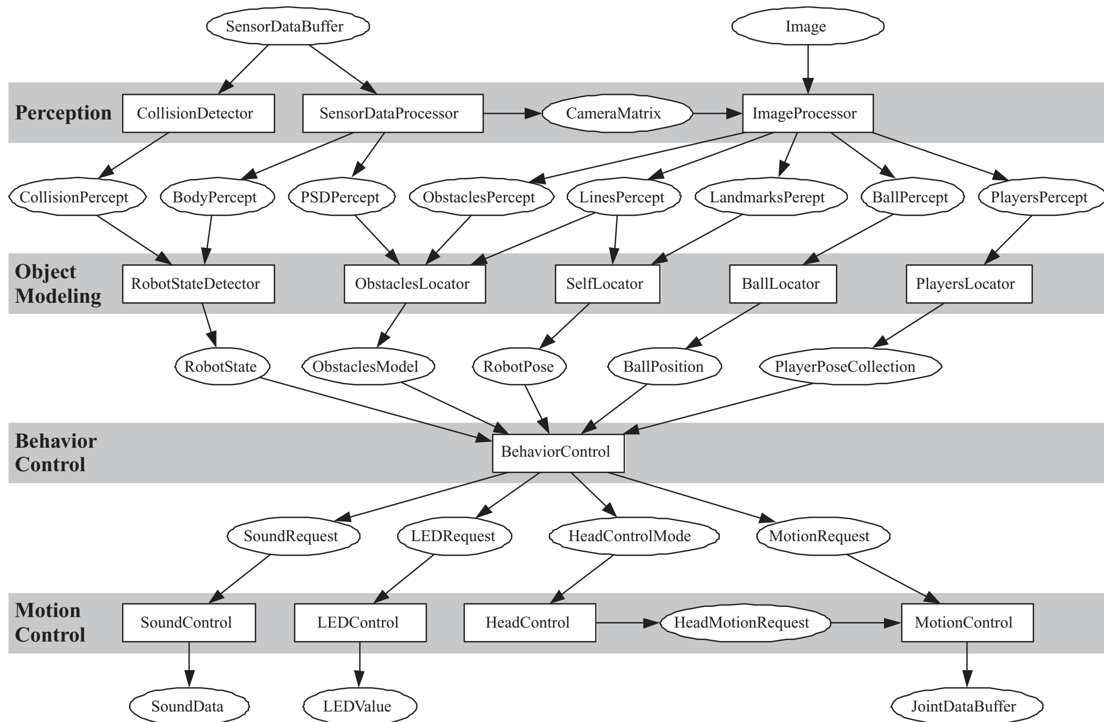

Abbildung 3.5: Konzept von GT 2006 (Quelle [18])

Dieser Zusammenhang lässt sich anhand des Ball-Objekts beispielhaft nachvollziehen. Wird bei der Bildverarbeitung vom *BallPerceptor* ein Ball erkannt, erzeugt dieser ein *BallPercept*, welches vom *BallLocator* für die Berechnung einer gefilterten Ballposition verwendet wird und in die Entscheidung des Verhaltens eingeht. Andere Modellierungen berechnen ihre Modelle aufgrund mehrerer Perzepte. Im Fall des Selflocators ist dies das *LinesPercept* sowie das *LandmarksPercept*.

Weitere Modellierungen, wie der *TeamBallLocator*, arbeiten nicht ausschließlich mit den eigenen Sensordaten. Diese Modellierungen setzen außer die eigenen Modelle auch die kommunizierten Modelle anderer Roboter ein und berechnen mithilfe dieser Daten ein Teammodell, das genutzt wird, sobald ein Objekt von der eigenen Sensorik nicht wahrgenommen wird.

Um die Ergebnisse der Modellierungen an andere Roboter kommunizieren zu können, werden diese in Form von *Teammessages* in *Messagequeues* abgelegt und über WLan verschickt. Die Empfänger können diese Messagequeues auslesen, die verschie-

denen Teammessages anhand ihrer ID zuordnen und weiterverarbeiten. Dies funktioniert nur dann zuverlässig, wenn der Inhalt einer Messagequeue für Sender und Empfänger identisch ist, wodurch nur homogene Robotersysteme miteinander kommunizieren können. GT verschickt ausschließlich Daten, die ein möglicher Empfänger für seine Berechnungen benötigt, wodurch eine effiziente Kommunikation erreicht wird.

Die Ergebnisse der Modellierungen können in der GUI auf einem Spielfeld visualisiert werden, wodurch die Weltmodellierung in ihren Einzelteilen sehr gut testbar ist. Die als Teammessages kommunizierten Modelle anderer Roboter werden nur in Form der gefilterten Teammodelle dargestellt.

3.3.3 Zusammenfassung

Die folgenden Tabellen fassen die Ergebnisse der analysierten Weltmodellierungen und deren Kommunikationsmodule anhand der Qualitätssicherungsmerkmale zusammen.

3.3.3.1 Weltmodellierung

	DD 2006	GT 2006
Modularität	-	+
Effizienz	-	+
Erweiterbarkeit	-	+
Korrektheit	-	+
Robustheit & Zuverlässigkeit	+	+
Testbarkeit	+	+
Plattformunabhängigkeit	+	-
Adaptierbarkeit	-	+
Erlernbarkeit	-	o
Wartbarkeit	-	+
Portabilität	-	-

Tabelle 3.1: Bewertung existierender Weltmodellierungen für Robotersysteme
(+ gut, o mittel, - schlecht)

3.3.3.2 Kommunikation

	DD 2006	GT 2006
Modularität	-	-
Effizienz	-	+
Erweiterbarkeit	-	-
Korrektheit	o	o
Robustheit & Zuverlässigkeit	-	-
Testbarkeit	-	-
Plattformunabhängigkeit	+	-
Adaptierbarkeit	+	+
Erlernbarkeit	+	+
Wartbarkeit	o	o
Portabilität	-	-

Tabelle 3.2: Bewertung existierender Kommunikationsmodule für Robotersysteme (+ gut, o mittel, - schlecht)

3.4 Existierende Hardwarearchitekturen

In diesem Abschnitt werden die Plattformen beschrieben, auf denen die in dieser Arbeit entstandenen Neuentwicklungen zum Einsatz kommen werden. Dabei wird das Hauptaugenmerk auf den HR18 gelegt, da aufgrund seines Einsatzes im RoboCup mit ihm die meisten Tests durchgeführt werden und die neuen Komponenten primär ihre Lauffähigkeit auf diesem System beweisen müssen.

3.4.1 HR18

Der von Hajime Sakamoto Institute Ltd.[2] entwickelte Roboter HR18 (siehe Abbildung 3.6) ist die Grundlage der Darmstadt Dribblers bei der Teilnahme an Wettbewerben in der Humanoid League [19] des RoboCups.

Mit einer Höhe von 55 cm und einem Gewicht von 3,3 kg spielen die Darmstadt Dribblers mit diesem Roboter in der Kid-Size Liga. Der Roboter besitzt 21 Motoren und somit 21 Freiheitsgrade, 6 je Bein, 3 je Arm, 1 in der Hüfte und 2 im Kopf, wodurch mit diesem System eine hohe Bewegungsfreiheit erlangt wird. Gyroskope und lineare Beschleunigungssensoren, die im Oberkörper angebracht sind, dienen der Regelung des Robotersystems.

Der Roboter ist mit 2 USB-Kameras mit gleicher Auflösung (640 x 480 Pixel) ausgestattet. Die am Kopf angebrachte Kamera hat einen Öffnungswinkel von 45° und wird

[2]Hajime Sakamoto Institute Ltd. – Webseite: `http://www.hajimerobot.co.jp`

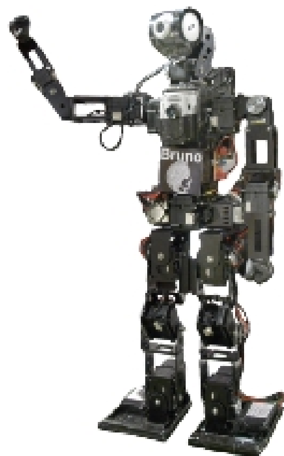

Abbildung 3.6: HR18: Bruno (Quelle [3])

Abbildung 3.7: HR18: Jan (Quelle [3])

über die Kopfmotoren bewegt. Sie wird dazu genutzt, bewegliche Objekte, wie den Spielball, zielgerichtet zu fixieren, wodurch auch weit entfernte Objekte erkannt werden können. Die zweite Kamera ist im Oberkörper montiert und mit einem Objektiv mit einem Öffnungswinkel von 95° ausgestattet, um große Objekte, wie Landmarken, Tore oder Hindernisse zu erkennen und sich mithilfe dieser Informationen auf dem Spielfeld lokalisieren zu können.

Die Ansteuerung des HR18 ist über zwei getrennte Systeme gelöst. Ein PC104 Board der Firma Digital-Logic AG[3] mit 256 MB DDR RAM und einem GEODELX800 500 MHz Prozessor dient der Softwaresteuerung [2]. Auf einer CompactFlash (CF)-Karte mit mindestens 32 MB ist das Echtzeitbetriebssystem Microsoft Windows CE 5 installiert. Die beiden Kameras sind über die USB-Schnittstelle eingebunden. Die Kommunikation ist wahlweise über den LAN-Kontroller des Boards oder über einen USB-WLan-Stick möglich. Die auf dem PC104-Board berechneten Bewegungssteuerungen werden über eine RS232-Schnittstelle an einen 32 bit Mikrocontroller mit 50 MHz geschickt, welcher die Gelenkwinkelstellungen für die Motoren berechnet und diese mit einer Ansteuerung von 50 Hz ausführt.

Das PC104 Board und der Mikrokontroller werden mit einem Lithium-Polymer-Akku mit 2 Zellen, 7.4 V versorgt, die Motoren mit 2 Akkus mit zusammen 4 Zellen, 14.8 V. Bei der Konstruktion des Roboters ist darauf geachtet worden, dass dieser einerseits robust, anderseits leicht gebaut ist, damit das Gesamtsystem hochdynamisch und flexibel bleibt. Aus diesem Grund besteht die Konstruktion des Roboters aus Aluminium bzw. kohlefaserverstärktem Kunststoff.

[3]Digital-Logic AG – Webseite: `http://www.digitallogic.com`

3.4.2 Weitere Plattformen

Aufgrund der Plattformunabhängigkeit von RoboFrame ist es möglich, die Robotersteuerungssoftware mit den Komponenten dieser Arbeit auf verschiedenen Robotersystemen einzusetzen. Dafür bieten sich unter anderem die am Fachgebiet SIM eingesetzten Roboter Pioneer 2-DX und der HR27 an.

3.4.2.1 Pioneer 2-DX

Abbildung 3.8: Pioneer 2-DX (Quelle [4])

Der von der Firma MobileRobots entwickelte, fahrende Roboter Pioneer 2-DX (siehe Abbildung 3.8) ist mit 2 Antriebsrädern vorne und einem Stützrad hinten ausgestattet. An Sensoren besitzt das verwendete Modell einen Ring aus 16 Sonarsensoren, fünf Tastsensoren sowie eine Kamera, die auf der oberen Platte des Roboters montiert ist. Die Kamera kann sich um 2 Achsen drehen und hat einen eingebauten Zoom. An der Vorderseite besitzt das System einen Greifer, mit welchem er bis zu 2 kg schwere Objekte bewegen kann. Ein eingebautes Gyroskop und ein linearer Beschleunigungssensor dienen als inertiale Lagesensoren [14].

Die Sensordatenaufnahme und Bewegungssteuerung ist über den Mikrocontroller SAB88C166 mit 1 MB internem RAM und 32 kb Flash-ROM realisiert, welcher seine Sensordaten über eine RS232-Schnittstelle an ein getrenntes System mit einem K6 III-400 MHz AMD Prozessor und 256 RAM sendet und von diesem die Bewegungssteuerungsbefehle erhält. Die Kommunikation kann wahlweise über Lan oder WLan erfolgen.

3.4.2.2 HR27

Der vom Fachgebiet SIM der TU-Darmstadt in Kooperation mit Hajime Sakamoto Institute Ltd. entwickelte vierbeinige Roboter HR27[4] wurde als offene, modulare Standard-

[4]HR27 – Webseite: `http://www.thenewrobot.com/`

plattform entworfen, um unter anderem den Sony AIBO[5] der RoboCup Four-Legged League in den kommenden Jahren als Standardroboter abzulösen (siehe Abbildung 3.9). Die zu diesem Zweck von der RoboCup Federation im November 2006 formulierten Voraussetzungen [20] dienten beim Design der Roboterarchitektur als Anforderungsprofil.

Abbildung 3.9: Sony AIBO und HR27 (Quelle Katrin Binner/TU Darmstadt)

Der vierbeinige Roboter besitzt drei Freiheitsgrade im Kopf sowie in jedem Bein, wodurch ähnlich schnelle Laufbewegungen wie bei einem AIBO ermöglicht werden. Mittels zweier in Kopf und Brust montierter Infrarot (IR)- Distanzsensoren können Objekte in 10-80 cm Entfernung detektiert werden. Durch ein Gyroskop und einen Beschleunigungssensor im Rumpf des Roboters ist es möglich, die Lage des Roboters zu erkennen, um gegebenenfalls Aufstehbewegungen durchführen zu können. Als wichtigster Sensor ist eine Philips SNC900 Kamera mit einer Auflösung von bis zu 640 x 480 Pixeln und 30 fps im Kopf montiert, die durch das Design der Kopfmotoren auch Objekte unterhalb des Oberkörpers erkennen kann. Über ein Display mit 16 Zeilen und vier Druckknöpfen ist es möglich, Einfluss auf die Steuerung des Roboters zu nehmen und sich Debugausgaben anzeigen zu lassen [8].

Die Berechnungen zur Steuerung der Motoren und Sensoren werden auf einem Mikrocontroller SH2/7125 MCU [17] der Firma Renesas[6] durchgeführt, welcher mit 8 k Flash-ROM und 4 k RAM ausgestattet ist und über einen seriellen Bus kommuniziert. Weitere Anschlüsse bietet diese MPU mittels acht Analog-Digital-Konverter (ADC) und 18 digitalen I/O Pins. Das bereits beim HR18 (siehe Abschnitt 3.4.1) eingesetzte PC 104 Board dient auch bei diesem Roboter als Plattform zur Steuerung und ist für den Einsatz der Betriebsysteme Linux und Windows CE ausgelegt.

[5] Sony AIBO – Webseite: `http://support.sony-europe.com/aibo/`
[6] Renesas – Webseite: `http://eu.renesas.com`

4 Anforderungen

In diesem Kapitel werden die inhaltlichen und programmiertechnischen Anforderungen beschrieben, die auf den Grundlagen des Kapitels 3 aufbauen und die Ziele der Arbeit (Abschnitt 1.3) einbeziehen.

Die zu entwickelnde Weltmodellierung soll Schnittstellen bereitstellen, die einerseits einfache Zugriffsmöglichkeiten auf die vorhandenen Sensorinformationen bieten und andererseits das errechnete Weltmodell den übrigen Modulen zur Verfügung stellen. Über die Kommunikationsschnittstelle sollen Daten des Weltmodells an andere Robotersysteme versendet werden können. Diese Schnittstelle ist derart flexibel zu gestalten, dass damit auch zur Laufzeit berechnete Ausschnitte des Weltmodells verschickt werden können. Nach Fertigstellung dieser beiden Komponenten soll eine verteilte Kartografierung möglich sein, sodass zwei oder mehrere Roboter Objekte in ihrer Umgebung erkennen, in ihr Weltmodell eintragen und sich gegenseitig ihre "Weltkarte" zuschicken können, wodurch eine komplette Umgebungskarte errechnet werden kann.

Um das Debuggen der eigenen sowie der kommunizierten Weltmodelldaten zu ermöglichen, sind diese auf einem virtuellen Spielfeld in der GUI zu visualisieren. Die Anzeige dieser Daten muss für jedes Objekt selektiv möglich sein, damit auch bei umfangreichen Weltmodellen einzelne Komponenten getestet werden können.

Damit die von der Weltmodellierung berechneten Daten in die situative Entscheidungsfindung der Verhaltenssteuerung eingehen können, wird eine Schnittstelle benötigt, die so intuitiv zu gestalten ist, dass eine genaue Abbildung der Modelle auf die im Verhalten genutzten Werte möglich wird. Diese Werte gehen in die vom Verhalten benutzte XabslEngine [12] ein, die zustandsabhängige Bewegungssteuerungsbefehle berechnet.

Neben diesen inhaltlichen Anforderungen, die hauptsächlich das Konzept dieser Arbeit beeinflussen, sind die programmiertechnischen Anforderungen, auch Qualitätssicherungsmerkmale genannt, zu betrachten, welche bei der Realisierung der einzelnen Komponenten zu berücksichtigen sind. Im Folgenden werden die für diese Arbeit relevanten Qualitätskriterien sowie ihr Einfluss auf die einzelnen Module beschrieben.

Bei der Weltmodellierung, die sich aus einzelnen Modellierungen zusammensetzt, ist besonderer Wert auf die *Erweiterbarkeit* zu legen, da die verschiedenen Modellierungen meist von unterschiedlichen Programmierern entwickelt werden. Um dies zur erreichen, sollen die einzelnen Modellierungen als Module verstanden werden, die dadurch ähnlich einem Baukastensystem an die Weltmodellierung gekoppelt werden können. Durch diese *Modularität* wird gewährleistet, das unabhängig voneinander realisierte Kompo-

nenten einfach hinzugefügt und entfernt werden können.

Durch die hohe Fluktuationsrate in den studentischen Arbeitsgruppen ist ein gute *Erlernbarkeit* der Schnittstellen der Weltmodellierung anzustreben und soll damit möglichst leicht verständlich und intuitiv sein. Unterstützt wird dies durch eine klar strukturierte Klassenhierarchie und das konsequente Wiederverwenden existierender Methoden. Durch die eben genannten Aspekte und den gezielten Einsatz von Design Patterns wird zusätzlich eine gute *Wartbarkeit* erzielt.

Auch die *Effizienz* ist in der autonomen Robotik, in der meist nur begrenzte Speicherkapazität und Rechenzeit zur Verfügung steht, eine wichtige Anforderung. Speziell die Organisationsstrukturen der Weltmodellierung sind so effizient zu implementieren, dass die für die Verwaltung benötigte Rechenzeit möglichst geringe Auswirkungen auf die Laufzeit und die damit verbundene Taktrate hat. Neben der Laufzeit der einzelnen Komponenten ist die von der Kommunikation erzeugte Netzlast zu minimieren, indem darauf geachtet wird, dass nur von anderen Robotern angeforderte Daten versendet werden und die Größe dieser Daten so gering wie möglich gehalten wird.

Die *Korrektheit* ist eine fast selbstverständliche Anforderung. Um diese zu gewährleisten, sollen verschiedene Tests mittels der durch das Framework gegebenen Debugmöglichkeiten sowie durch die Entwicklung eigener Hilfsprogramme oder Dialoge durchgeführt werden. Diese Testmöglichkeiten sollen durch automatisch ablaufende Testfälle für die in dieser Arbeit entwickelten Funktionalitäten ergänzt werden, sodass eine gute *Testbarkeit* der Komponenten erreicht wird. Um diese auch während der Laufzeit zur Verfügung zu stellen, sollen Debugausgaben an kritischen Stellen hinzugefügt werden. Mittels einer GUI zur Visualisierung des Weltmodells ist das Debuggen der einzelnen, meist sehr komplexen Module zu ermöglichen. Durch die zentrale Rolle der Weltmodellierung im Datenfluss der Anwendung ist darauf zu achten, dass Fehler – falls möglich – abgefangen werden und die Funktionsfähigkeit der Applikation aufrecht erhalten wird, wodurch eine gute *Robustheit* und *Zuverlässigkeit* erreicht wird.

Weitere programmiertechnische Ziele wie *Plattformunabhängigkeit*, *Adaptierbarkeit* und *Portabilität* werden in dieser Arbeit nur geringfügig betrachtet. Da sie weitestgehend auf die Möglichkeiten von RoboFrame begrenzt sind, ist darauf zu achten, diese Ziele durch die Neuentwicklungen nicht weiter einzuschränken.

5 Konzept

In diesem Kapitel wird der konzeptionelle Aufbau und die Integration der Weltmodellierungen sowie der Kommunikation in eine auf dem RoboFrame aufbauende Anwendung beschrieben. Dieses Konzept orientiert sich größtenteils an den inhaltlichen Anforderungen von Kapitel 4, berücksichtigt aber auch soweit wie möglich die relevanten programmiertechnischen Anforderungen, die allerdings stärker in das Kapitel 6 (Realisierung) einfließen.

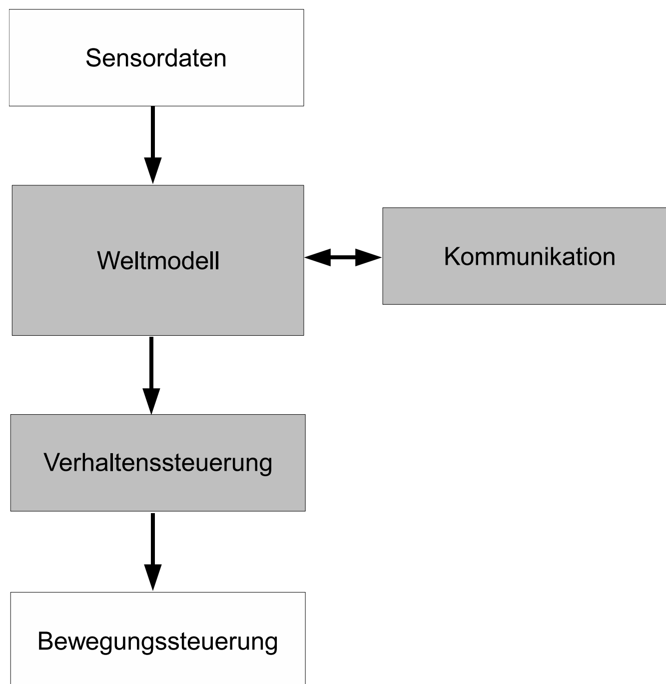

Abbildung 5.1: Übersicht über die Rolle der Weltmodellierung, Kommunikation und Verhaltenssteuerung in einer auf dem RoboFrame aufbauenden Robotersteuerungssoftware. Die grauen Rechtecke sind in dieser Arbeit behandelte Komponenten.

Das *Weltmodell* ist als Framework zu verstehen, welches durch Modellierungen erweitert werden kann (siehe Abbildung 5.1). Die *Kommunikation* ist ein Modul, welches von der Weltmodellierung initialisiert und gesteuert wird sowie den Austausch von Weltdaten zwischen heterogenen Robotern ermöglicht. Neben den kommunizierten Weltdaten stehen den Modellierungen die *Sensordaten* sowie die Berechnungen anderer Modellierungen zur Verfügung. Anhand dieser Informationen berechnet jede Modellierung ein Modell, welches mit den Modellen anderer Modellierungen in die *Verhal-*

tenssteuerung eingeht. Das Verhalten bestimmt anhand des aktuellen Weltmodells die nächste Aktion und generiert daraus die entsprechenden Bewegungssteuerungsbefehle.

Das Weltmodell soll den späteren Entwicklern die Möglichkeit geben, Modellierungen modular einzubinden, die zur Berechnung benötigten Daten einfach abzurufen und die aus den Berechnung resultierenden Modelldaten zu kommunizieren, ohne dabei nähere Details des Weltmodells oder der Kommunikation betrachten zu müssen. In den folgenden Abschnitten wird auf die Konzeption der dafür benötigten Komponenten näher eingegangen.

Die von RoboFrame vorgegebenen Funktionen werden soweit möglich verwendet und gegebenenfalls angepasst, sodass neue Fehlerquellen ausgeschlossen werden können und die Funktionalität des Frameworks erweitert wird.

5.1 Weltmodellierung

Die Weltmodellierung wird als modulares Verwaltungsmodul konzipiert, welches ein leichtes Austauschen und Erweitern von Modellierungen ermöglicht. Die Daten dieses Moduls werden von einer zentralen Datenkomponente verwaltet und über eine einheitliche Schnittstelle den Modellierungen sowie der Kommunikation zur Verfügung gestellt. Bei Verwendung der Schnittstelle werden Informationen über die von den Modellierungen verwendeten Eingabedaten gewonnen, die für Verwaltungsaufgaben genutzt werden.

Die Weltmodellierung als übergeordnete Verwaltungseinheit wird in Form eines Moduls in der RoboCup-Anwendung instantiiert (siehe Abbildung 5.2) und beinhaltet die *Modellierungsverwaltung*, die *zentrale Datenverwaltung* und die *Kommunikation* (siehe Abschnitt 5.2).

Die *Modellierungsverwaltung* initialisiert die einzelnen Modellierungen, sodass diese aufgrund ihrer Metainformationen gezielt gesteuert und beeinflusst werden können. Die *Modellierungen* berechnen anhand der Eingabedaten aus der zentralen Datenverwaltung jeweils ein Modell, welches über die Datenverwaltung anderen Modellierungen oder der Kommunikation zur Verfügung gestellt wird und in die Verhaltenssteuerung eingeht.

Das Konzept dieser Arbeit sieht eine Zentralisierung der Daten sowie deren Trennung von den Berechnungskomponenten vor, um diese Daten über eine einheitliche Schnittstelle anderen Modulen zur Verfügung stellen zu können. Die dafür vorgesehene *zentrale Datenverwaltung* hält alle für die Modellierungen relevanten Objekte, wie Sensordaten, Modelle anderer Modellierungen sowie kommunizierte Modelle anderer Roboter. Um den anfragenden Komponenten eine eigene Sicht auf die Objekte zu ermöglichen, werden diese über Buffer zur Verfügung gestellt. Da alle Berechnungskomponenten ihre Daten von dieser Verwaltung beziehen, ist es möglich auszuwerten, welche Daten die verschiedenen Komponenten bezogen haben, um diese Information für Steuerungszwecke zu verwenden.

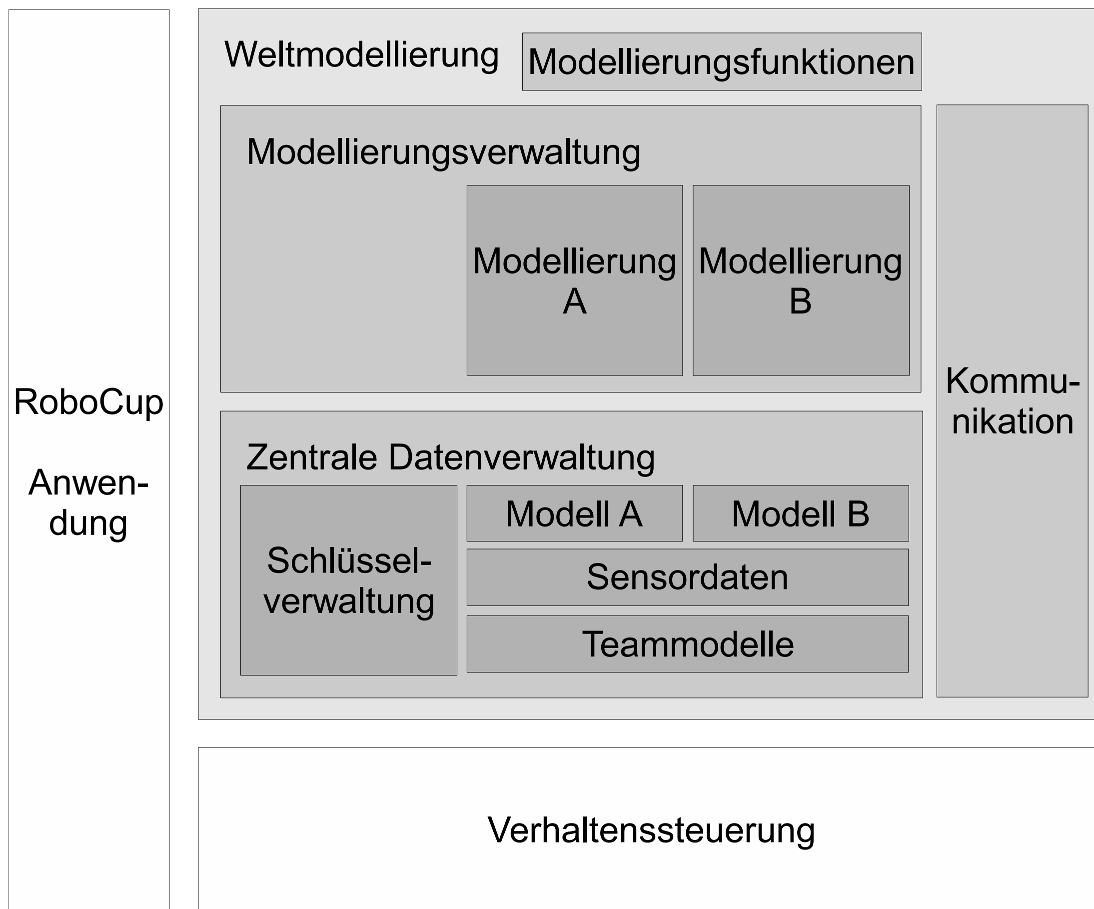

Abbildung 5.2: Konzept der Weltmodellierung mit den einzelnen Komponenten

Um Daten eines Weltobjektes, das über verschiedene Sensoren wahrgenommen und mittels unterschiedlicher Buffer verschickt werden kann, über einen eindeutigen Schlüssel den Modellierungen zur Verfügung stellen zu können, ist eine Abbildung der für die Buffer benötigten Schlüssel auf eigene Identifier nötig. Diese Zuordnung steht der zentralen Datenverwaltung in Form der *Schlüsselverwaltung* zur Verfügung, mit deren Hilfe die für die Buffer nötigen Initialisierungen und Registrierungen vorgenommen werden.

Mit den *Modellierungsfunktionen* als eine Standardbibliothek werden den einzelnen Modellierungen Funktionen zur Verfügung gestellt, deren Implementierung unabhängig von den in den Modellierungen verwendeten Objekten ist.

5.2 Kommunikation

Die Kommunikation wird als Teil der Weltmodellierung (siehe Abbildung 5.3) konzipiert, die für den Versand und Empfang der Weltdaten zuständig ist. Sie bildet somit die Schnittstelle zwischen externen und eigenen Weltmodellen und spaltet sich in eine Verwaltungs- (*Kommunikationsverwaltung*) und eine Datenklasse (*Kommunikation Da-*

tenverwaltung) auf.

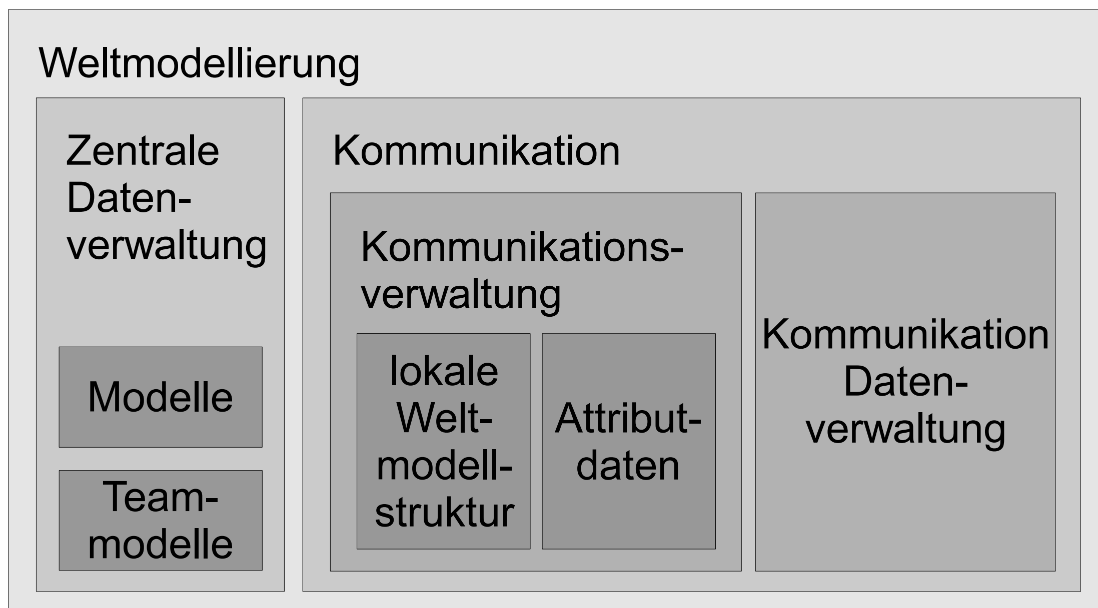

Abbildung 5.3: Konzept der Kommunikation mit den einzelnen Komponenten

Der Datenaustausch mit der eigenen Weltmodellierung erfolgt über die bereits beschriebene zentrale Datenverwaltung (siehe Abschnitt 5.1). Diese bietet der Kommunikation die Möglichkeit, die von Modellen für den Datentransfer vorgesehenen Attribute abzufragen. Die Attribute bestehen einerseits aus einem Identifier, der sie eindeutig innerhalb eines Modells kennzeichnet, und andererseits aus den Daten, die bei jeder Berechnung des Modells aktualisiert werden. Aus den Attributidentifiern wird zusammen mit den Identifiern der Modelle in der Datenverwaltungskomponente der Kommunikation die Struktur des lokalen Weltmodells berechnet. Dieses lokale Weltmodellstruktur wird über UDP mithilfe der Kommunikationsmechanismen von RoboFrame an alle Robotersysteme versendet.

Die empfangenen Weltmodellstrukturen werden zusammen mit der lokalen Struktur zu einem globalen Weltmodell verrechnet. Dieses Schnittmodell besteht aus allen Attributen, die von mindestens zwei Robotersystemen verstanden werden (siehe Abbildung 5.4) und bildet die Grundlage für den Versand der Attributdaten. Dadurch kann ein zyklischer Austausch der Modelle basierend auf den Rohdaten der Attribute realisiert werden, wodurch eine effiziente Kommunikation ermöglicht wird. Mithilfe der empfangenen Rohdaten sowie des globalen Weltmodells werden von der Kommunikationsverwaltung Teammodelle erzeugt, die über die zentrale Datenverwaltung anderen Modellierungen zur Verfügung gestellt werden.

Durch das Aufspalten des Weltmodells in einzelne Modelle und weitergehend in eine Ansammlung von Attributen wird die in den Anforderungen (siehe Kapitel 4) gewünschte Heterogenität erreicht. Aufgrund der modellunabhängigen Behandlung der Objekte in der zentralen Datenverwaltung muss keine Anpassung der Kommunikation

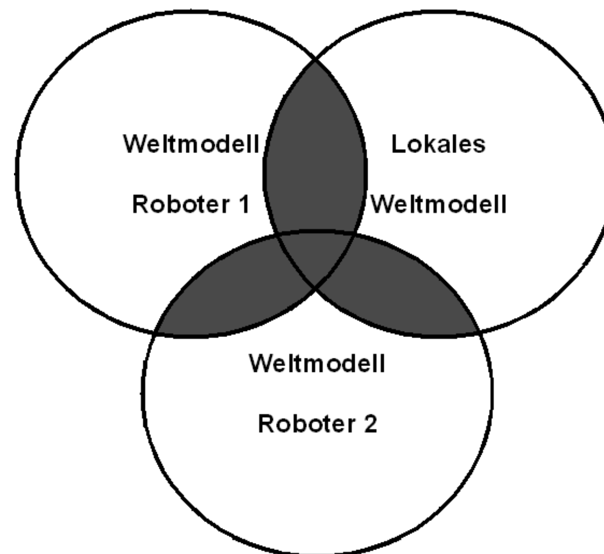

Abbildung 5.4: Kommunizierte Weltmodelle mit dem daraus resultierenden globalen Weltmodell. Die grauen Teilmengen bilden das berechnete globale Weltmodell.

bei einer Erweiterung oder Veränderung der Welt durch Modellierungen erfolgen, wodurch eine hohe Flexibilität erreicht wird.

5.3 Verhalten

Die Verhaltenssteuerung ist ein eigenes Modul, das von der RoboCup-Anwendung initialisiert wird (siehe Abbildung 5.5). Es verwendet die bestehende Bibliothek *XabslEngine*, die einen Zustandsautomaten beinhaltet und Schnittstellen zur Dateneingabe und zur Bewegungsausgabe bereitstellt, sowie das darauf aufbauende Framework *XabslControl* [22], das die Schnittstelle zwischen RoboFrame und der Zustandsmaschine bildet.

Die in der Programmiersprache XABSL[1] entwickelte Zustandsmaschine trifft anhand der aktuellen Eingabedaten situative Entscheidungen. Die in Form von *Symbolen* (Beispiel: Symbol A, B) bereitgestellten Eingabedaten werden von der XabslControl-Schnittstelle *SymbolProvider* verwaltet. Der von der Zustandsmaschine berechnete Endzustand generiert durch ein *BasicBehavior* (Beispiel: BB A, B) Bewegungssteuerungsbefehle. Diese BasicBehaviors werden von den *BasicBehaviorProvider* gekapselt, die zusammen mit den SymbolProvidern die Registrierungen in der XabslEngine übernehmen.

Des Weiteren sieht das neue Konzept vor, die aus der Weltmodellierung resultierenden Modelle in eigene Symbolklassen abzubilden, wodurch eine gute Übersichtlichkeit und Wartbarkeit erreicht wird und die Debugmöglichkeiten verbessert werden.

[1]XABSL – Website: `http://www2.informatik.hu-berlin.de/ki/XABSL/`

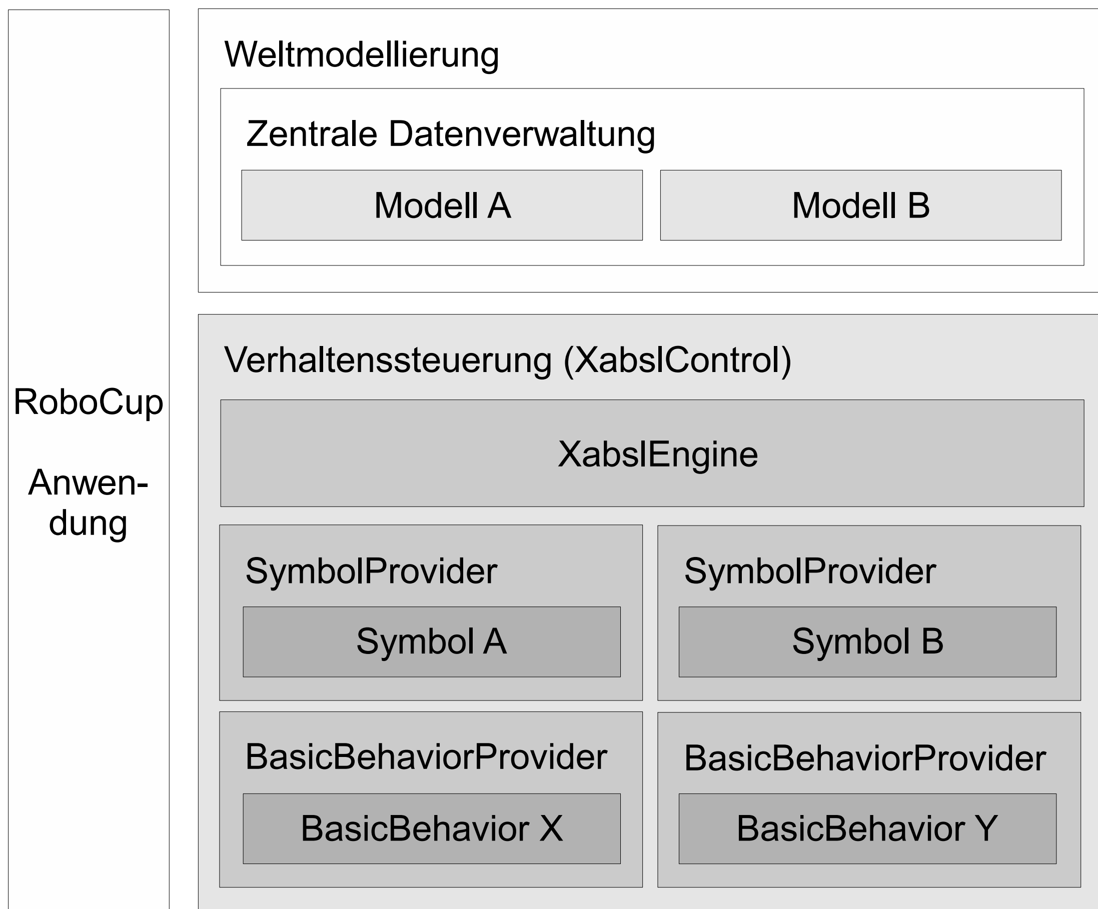

Abbildung 5.5: Konzept der Verhaltenssteuerung mit den einzelnen Komponenten.

6 Realisierung

Die Implementierung dieser Arbeit ist in C++ und vollständig objektorientiert realisiert. Sie orientiert sich am im vorherigen Kapitel 5 beschriebenen Konzept und erfüllt die programmiertechnischen Anforderungen aus Kapitel 4, sodass die Qualität der Arbeit als zentraler Bestandteil einer Robotersteuerungssoftware den hohen Ansprüchen gerecht wird. Im Ergebnisteil, dem Kapitel 7, wird auf die von der Software erfüllten Qualitätsmerkmale eingegangen.

In den folgenden Abschnitten wird die Realisierung der einzelnen Komponenten der Weltmodellierung, der heterogenen Kommunikation und der Verhaltensanbindung anhand von einzelnen UML-Diagrammen erläutert. Aufgrund der Komplexität der Klassen wird auf die Darstellung der Methodenattribute verzichtet und nur die für den Benutzer relevanten Methoden beschrieben.

Die im Rahmen dieser Arbeit entstandenen Klassen wurden in das neu für den RoboCup 2007 angelegt Projekt eingebunden. Die gesamte API ist direkt im Quellcode vollständig dokumentiert und kann mit Doxygen generiert werden.

6.1 Weltmodellierung

Die Weltmodellierung wird als eigenständiges Projekt in die Robotersteuerungssoftware integriert und beinhaltet die Kommunikation sowie bereits implementierte Modellierungen mit dem jeweiligen Modell. Dieses Projekt ist unabhängig von den Perzeptoren und dem Verhalten und lässt sich aus diesem Grund in eine beliebige RoboFrame Anwendung einbinden und mit Modellierungen erweitern.

Im Folgenden werden alle im Konzept erwähnten Komponenten der Weltmodellierung beschrieben, Beziehungen untereinander dargestellt und detailliert auf die Implementierung eingegangen.

6.1.1 RoboFrame-Modul Weltmodellierung

Das WorldModelModule ist die Hauptverwaltungsklasse der Weltmodellierung, in welcher die Unterkomponenten *WorldData* (siehe Abschnitt 6.1.3), *Communication* (siehe Abschnitt 6.2) und der *ModelingHandler* (siehe Abschnitt 6.1.4) erstellt und verwaltet werden. Sie stellt die Schnittstelle zur Hauptanwendung dar und implementiert alle

dafür notwendigen Methoden und steuert die benötigten Methodenaufrufe der untergeordneten Komponenten. Der von RoboFrame vorgegebene Lebenszyklus eines Moduls (siehe Abbildung 6.1) wird hier implementiert und zur Steuerung der Weltmodellierungskomponenten genutzt.

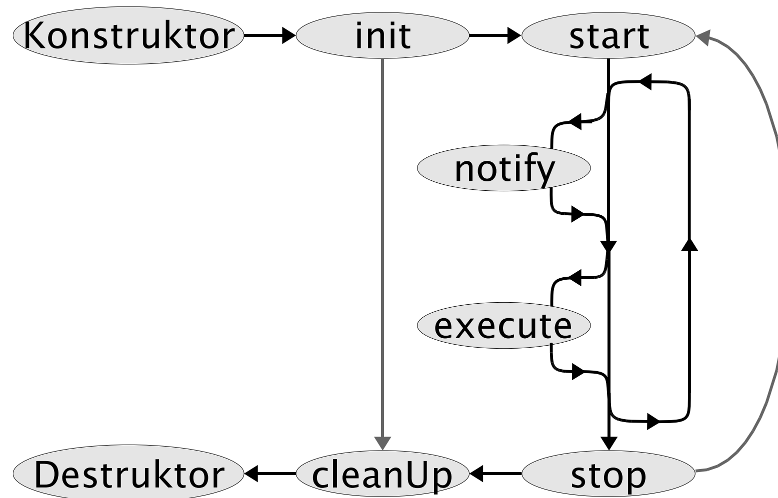

Abbildung 6.1: Lebenszyklus von Modulen (Quelle [16])

Damit die Weltmodellierung von einer RoboFrame Anwendung (*RoboCup07App*) aufgerufen werden kann, muss sie in Form eines *Moduls* in einen Prozess eingebunden werden (siehe Abbildung 6.2). Um die für dieses Modul spezifischen Methoden nicht für jede Anwendung erneut implementieren zu müssen, ist die von *Module* erbende Klasse *WorldModelModule* als eine spezifische, anwendungsabhängige Instanz (*WorldModelModule07*) einzubinden. Durch diese Hierarchie wird eine einfache Integration des Weltmodells in eine RoboFrame Anwendung mit einer beliebigen Kombination an Modellierungen ermöglicht, indem diese im Konstruktor von WorldModelModule07 an den ModelingHandler übergeben werden, der für die Verwaltung und Steuerung der Modellierungen zuständig ist (siehe Abschnitt 6.1.4).

Damit die zu Debugzwecken verwendete RoboGUI mit den Modellierungen Informationen austauschen kann, sind die dafür nötigen RoboFrame-Schlüssel im Konstruktor der WorldModelModule07 Klasse an die jeweilige Modellierung zu übergeben. Die von WorldData bereitgestellten Daten, wie Perzepte, Modelle und Teammodelle, werden über eine eigene Schlüsselverwaltung auf die RoboFrame Schlüssel abgebildet und können über diese eingebunden und angefragt werden (siehe Abschnitt 6.1.2). Daher müssen sie nicht dem Konstruktor von WorldModelModule07 übergeben werden.

Wie im Konzept der Kommunikation (siehe Abschnitt 5.2) beschrieben, werden für den Datentransfer die Kommunikationsmechanismen von RoboFrame genutzt. Die dafür benötigten Konnektoren können über die Methode *getConnectors* von WorldModelModule, die ihrerseits die Konnektoren von der Kommunikation abruft, auf der RoboCup07App registriert werden (siehe Abschnitt 6.2). Die Klasse WorldModelModule07 ist damit die einzige Schnittstelle zur RoboCup-spezifischen Anwendung RoboCup07App.

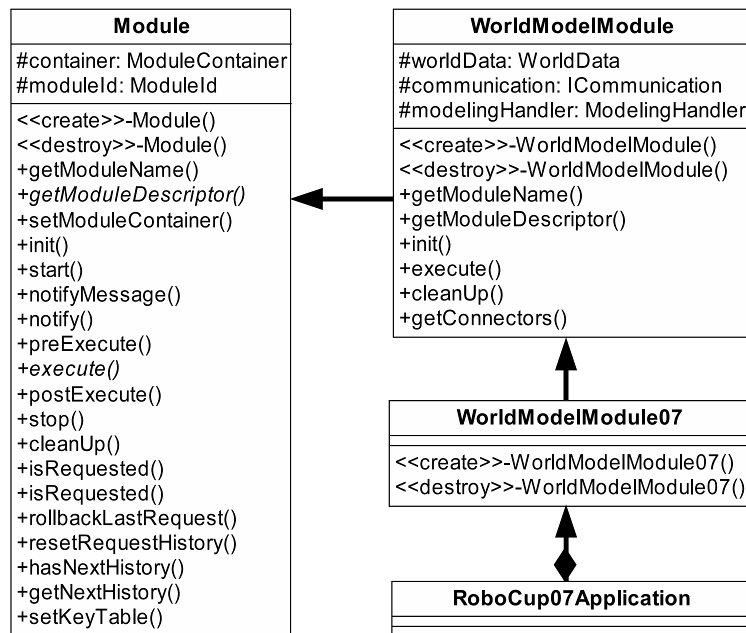

Abbildung 6.2: Integration der Weltmodellierung als Modul in eine RoboFrame Anwendung

6.1.2 Schlüsselverwaltung

Um die in der Weltmodellierung enthaltenen Daten verwalten und den Modellierungen sowie der Kommunikation zur Verfügung stellen zu können, wird für jedes in der Welt vorhandene Objekt, was über mehrere Sensoren wahrgenommen werden kann, ein eindeutiger Schlüssel benötigt. Dieser wird von einer Schlüsselverwaltung auf die Schlüssel des RoboFrames abgebildet, um über die internen Buffer an andere Module übertragen werden zu können.

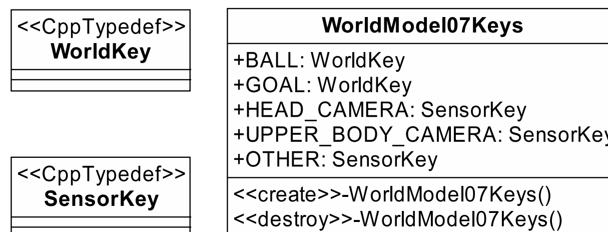

Abbildung 6.3: Schlüssel der Weltmodellierung

Die Schlüssel der Weltmodellierung können in *WorldKeys*, welche die Art der Daten identifiziert (zum Beispiel *BALL*), und in *SensorKeys* (zum Beispiel *HEAD_CAMERA*), welcher den Sensor zu einem Datensatz beschreibt (siehe Abbildung 6.3), unterschieden werden. Dies ermöglicht einem Entwickler, Daten eines bestimmten Typs anzufordern und innerhalb der Datenklasse getrennt nach dem jeweiligen Sensor auswerten und

fusionieren zu können. Da sich die in der Welt vorhandenen Objekte je nach Robotersteuerungsanwendung unterscheiden, sind diese Objekte in einer für die Anwendung spezifischen Definitionsklasse (den *WorldModel07Keys*) anzulegen.

Diese Schlüssel werden über den *KeyManager* in der Weltmodellierung registriert, der für diesen Zweck benötigte Registrierungsmechanismen sowie Methoden zum Abruf dieser Daten bereitstellt (siehe Abbildung 6.4). Die ebenfalls notwendige anwendungsspezifische Klasse wird in Form des *KeyManager07* der Robotersteuerungssoftware hinzugefügt.

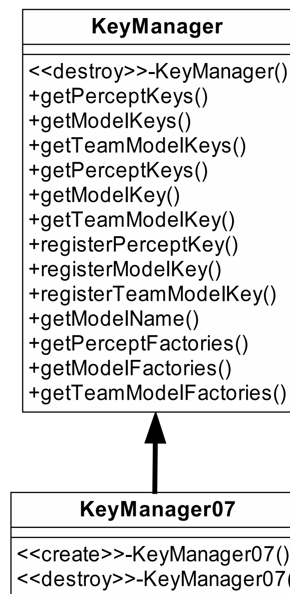

Abbildung 6.4: Schlüsselverwaltung der Weltmodellierung

Für die zu registrierenden Daten stehen verschiedene Template-Methoden zur Verfügung, die im *Konstruktor* vom KeyManager07 aufgerufen werden müssen. Da für jede Datenart andere Zusatzinformationen benötigt werden, wird bei der Registrierung zwischen Perzepten, Modellen und Teammodellen unterschieden, bei denen jeweils der Weltmodellschlüssel, der RoboFrame-Schlüssel und der Datentyp benötigt werden. Um die Perzepte anhand des Sensors unterscheiden zu können, wird bei der Registrierung mittels *registerPerceptKey* neben diesen Standardinformationen der Sensorschlüssel übergeben. Dies ermöglicht einer Modellierung, Datenobjekte eines bestimmten Typs mit dem Weltmodellschlüssel abzufragen und anhand des Sensorschlüssels zu unterscheiden. Dadurch wird sicher gestellt, dass eine Modellierung alle Datenobjekte eines angeforderten Typs bekommt. Um die Modelle mittels der Kommunikation mit anderen Robotern austauschen zu können, ist bei der Registrierung in *registerModelKey* ein Identifier in Form einer Zeichenkette nötig, der einen Datentyp eindeutig und intuitiv markiert und über *getModelName* abrufbar ist. Die Teammodelle benötigen keine zusätzlichen Informationen und werden lediglich mit den Standardinformationen über *registerTeamModelKey* gespeichert.

In den beschriebenen Registrierungsmethoden wird zusätzlich zur Verknüpfung der verschiedenen Schlüssel mithilfe des Datentyps eine Factory (siehe Abschnitt 6.1.2.1) erstellt, die über die Methoden *getPercept-, getModel-* und *getTeamModelFactories* von der zentralen Datenverwaltung abgerufen werden. Die gespeicherten Schlüssel können entweder vollständig oder einzeln anhand des Weltmodellschlüssels über verschiedene Get-Methoden (beispielsweise *getPerceptKeys*) getrennt nach Perzepten, Modellen oder Teammodellen abgefragt werden.

Vor der Schlüsselregistrierung sind die durch die Schlüssel repräsentierten Daten der Weltmodellierung nicht bekannt und können weder von den Verwaltungseinheiten noch von den Modellierungen und der Kommunikation verwendet werden.

6.1.2.1 Factories

Damit die in der Schlüsselverwaltung registrierten Daten mit ihrem jeweils unterschiedlichen Typ über eine Schnittstelle der zentralen Datenverwaltung angefordert werden können, sind sogenannte *Factories* nötig, die zu jedem registrierten Datentyp in der Schlüsselverwaltung angelegt werden. Jede Factory ist eine Templateklasse, die für einen Datenbereich zuständig ist, die *ModelFactory* für Modelle, die *PerceptFactory* für Percepte und die *TeamModelFactory* für Teammodelle (siehe Abbildung 6.5).

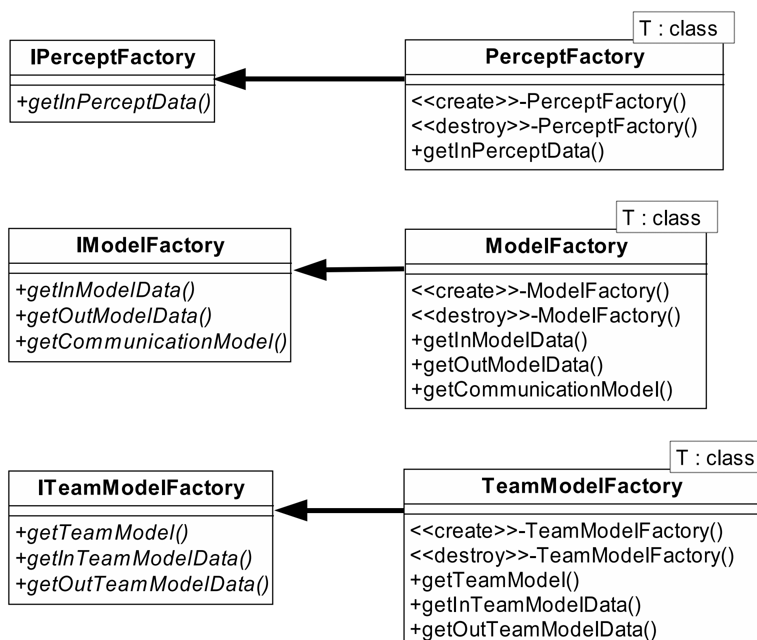

Abbildung 6.5: Factories zur Erstellung konkreter Datenklassen

Die PerceptFactory stellt über die Methode *getInPerceptData* eine neue Instanz eines InPerceptData-Objekts bereit, der einen Buffer mit dem jeweiligen Perzept enthält. Die Funktionalität wird in Abschnitt 6.1.3.1 näher erläutert.

Die ModelFactory bietet über die Methoden *getInModelData* beziehungsweise *getOutModelData* die Möglichkeit, eine neue Instanz eines In- beziehungsweise OutModelData-Objekts zu beziehen. Sie kapseln einen In- beziehungsweise OutBuffer des jeweiligen Modells (siehe Abschnitt 6.1.3.2). Um die von diesen Modellen für die Kommunikation vorgesehenen Attribute abfragen zu können, steht die Methode *getCommunicationModel* zur Verfügung.

Die TeamModelFactory hat analog zur ModelFactory Get-Methoden, die neue Instanzen auf In- und OutTeamModelData-Objekte zurückgeben. Sie kapseln In- und OutBuffer für Teammodelle, auf die in Abschnitt 6.1.3.3 näher eingegangen wird. Über die Methode *getTeamModel* kann bei der Factory ein Teammodell abgerufen werden, in das die Kommunikation die empfangenen Daten speichert.

Diese Factories werden von der zentralen Datenverwaltung benötigt, um den anfragenden Komponenten in der Initialisierungsphase der Weltmodellierung die gekapselten Datenobjekte zur Verfügung stellen zu können.

6.1.3 Zentrale Datenverwaltung

Die in der Weltmodellierung vorhandenen Daten (Perzepte, Modelle und Teammodelle) werden nach dem Konzept dieser Arbeit in einer Klasse, *WorldData* (siehe Abbildung 6.6), zentral verwaltet. Dies hat den Vorteil, dass alle Module über eine einheitliche Schnittstelle Weltdaten bei dieser einen Klasse anfordern können. Dabei werden die Daten nicht gespeichert, sondern die bereits vorhandenen RoboFrame-Strukturen in gekapselter Form weitergeben. Den anfragenden Komponenten wird ein Großteil des Verwaltungsaufwands abgenommen. Dieser Verwaltungsaufwand beinhaltet erstens die Registrierung der Buffer über die Methode *getModuleDescriptor* auf RoboFrame, die von dieser Klasse bei der Datenanfrage automatisch vorgenommen wird, und zweitens die durch die Erstellung der Daten-Objekte im Destruktor vorzunehmenden Aufräumarbeiten. Der Kommunikation wird zusätzlich ermöglicht, die zum Versenden und Empfangen vorgesehenen Modelle typenunabhängig und flexibel zu behandeln (siehe Abschnitt 6.2).

Für die Datenanfrage bietet WorldData die Möglichkeit, Daten-Objekte über die Methoden *getInPercept-, -Model-* und *-TeamModelData* sowie *getOutModel-* und *-TeamModelData* anzufragen, die als Referenz zurückgegeben werden. Die Instanzen dieser Datenklassen werden in WorldData verwaltet, um die erwähnten Aufräumarbeiten durchführen zu können. Um eine solche Datenverwaltung realisieren zu können, wurden die bestehenden In- und OutBuffer des RoboFrames um *ExtendedGenericIn-* und *OutBuffer* erweitert, welche im Gegensatz zum In- und OutBuffer eine Initialisierung der Größe des Buffers im Konstruktor ermöglichen. Diese Erweiterung war erforderlich, da die Größe der von den Modellierungen benötigten Buffer erst beim Abrufen aus WorldData bekannt ist und somit zur Kompilezeit der Anwendung noch nicht feststeht.

Infolge der Datenanfrage ist der Datenverwaltung bekannt, welche Informationen von

WorldData

-keyManager: KeyManager

<<create>>-WorldData()
<<destroy>>-WorldData()
+init()
+getTeamModel()
+getInPerceptData()
+getInModelData()
+getOutModelData()
+getInTeamModelData()
+getOutTeamModelData()
+getInputPercepts()
+getInputModels()
+getInputTeamModels()
+getRegisteredModels()
+getCommunicationModel()
+getModuleDescriptor()

Abbildung 6.6: Zentrale Datenverwaltung in der Weltmodellierung

den Modellierungen zur Berechnung des eigenen Modells genutzt werden. Die Datenverwaltung kann diese Metainformationen für Steuerungszwecke zur Verfügung stellen. Die für diesen Zweck entwickelten Methoden *getInputPercepts-, Models-* und *TeamModels* bieten die Möglichkeit, anhand eines Weltmodell-Schlüssel abzufragen, welche Perzepte, Modelle und Teammodelle bei der Berechnung eines Modells benötigt werden.

Die bei der Registrierung der Modellierungen festgelegten, eindeutigen Schlüssel können mit der Methode *getRegisteredModels* abgefragt werden. Diese Funktionalität wird von der Kommunikation genutzt, um die Struktur des lokalen Weltmodells aufzubauen (siehe Abschnitt 5.2). Die für die Struktur ebenfalls benötigten Attributinformationen eines Modells sind über die Methode *getCommunicationModel* abrufbar.

In den folgenden Abschnitten werden die in der Datenverwaltung abzufragenden Objekte erläutert und wird auf ihre Implementierung eingegangen.

6.1.3.1 Perzepte

Die Perzepte werden in der *InPerceptData*-Templateklasse gekapselt und beinhalten die einzelnen Buffer des RoboFrames mit den Daten der Sensoren des jeweils zugehörigen Weltmodellschlüssels (siehe Abbildung 6.7). Die Initialisierung sowie die Registrierung dieser Buffer werden wie beschrieben durch WorldData automatisch vorgenommen.

Diese Datenklasse verwaltet die Perzepte eines Weltmodellschlüssels zu verschiedenen Sensorschlüsseln, die über die polymorphe Methode *getBuffer* abgerufen werden kann. Diese Methode bietet die Möglichkeit, sowohl den ersten gespeicherten Buffer als auch den mit einem Sensorschlüssel spezifizierten zurückzugeben, sodass bei InPerceptData-Objekten mit nur einem Buffer dieser auch ohne Angabe des Sensorschlüssels abgefragt werden kann.

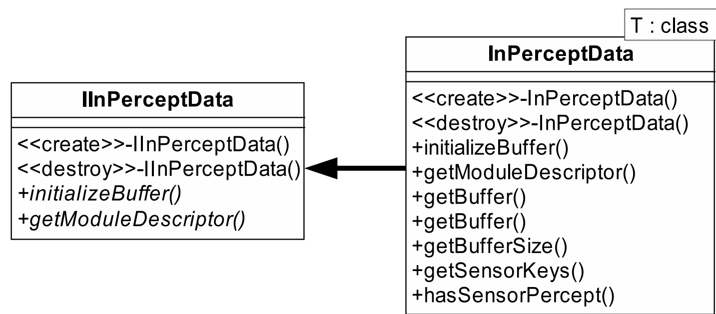

Abbildung 6.7: Percepte in der zentralen Datenverwaltung

Die Anzahl der enthaltenen Buffer und somit die Anzahl der unterschiedlichen Sensorperzepte eines Perzepttyps kann über die Methode *getBufferSize* angefragt werden, enthaltene Sensorschlüssel stehen über *getSensorKeys* zur Verfügung. Um sicher zu stellen, dass ein von dem Entwickler gewünschtes Perzept zu einem Sensorschlüssel existiert, wurde die Methode *hasSensorPercept* implementiert. Aufgrund dieser Methoden werden den Modellierungen alle verfügbaren Sensorinformationen zu dem jeweiligen Datenobjekt bereitgestellt, sodass diese bei Erweiterungen des Datentyps durch neue Sensorinformationen unmittelbar verwendbar sind.

Wird ein Perzept über einen nicht vorhandenen Sensorschlüssel abgefragt, ist die Funktionsfähigkeit der Anwendung nicht weiter gewährleistet. In diesem Fall wird eine Fehlermeldung mit dem entsprechenden Sensorschlüssel ausgegeben.

6.1.3.2 Modelle

Die von den Modellierungen berechneten Modelle erben von der Basisklasse *Model* (siehe Abbildung 6.8) und implementieren die für den Datenaustausch notwendige Schnittstelle *IStreamable*. Diese Elternklasse ermöglicht der Kommunikation sowie der GUI, alle Modelle über diesen allgemeinen Datentyp behandeln zu können.

Damit die für die Kommunikation benötigten Informationen zur Bildung des lokalen Weltmodells aus den Modellen gelesen werden können, steht die statische Methode *getAttributeIdentifier* bereit, welche die Identifier der zu kommunizierenden Attribute zurückgibt. In dieser Methode werden der übergebenen Attributliste mit *setAttribute-Identifier* die Identifier hinzugefügt. Damit die Kommunikation die entsprechenden Attribute aus dem allgemeinen Modell auslesen und in dieses setzen kann, werden über die Methoden *registerAttributePtrs* und *registerAttribute* Zeigerregistrierungen auf die Variablen angestoßen. Die Registrierung wird mittels *registerValue* in gleicher Reihenfolge wie bei den Identifiern vorgenommen. Die von anderen Robotern empfangenen Daten werden dem Modell über *setAttribute* hinzugefügt (siehe Abschnitt 6.2.1).

Um die Modelle in der GUI visualisieren zu können, müssen die Methoden *getDrawingInRobot* und *getDrawingInWorld* überschrieben werden. Um der GUI die Mög-

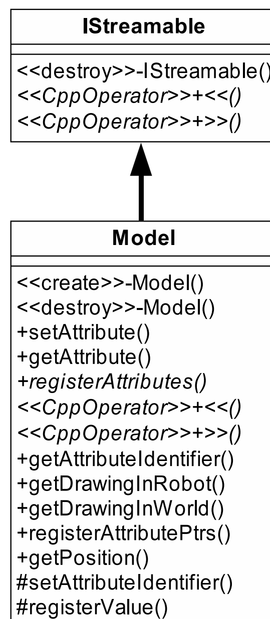

Abbildung 6.8: Schnittstelle für Modelle

lichkeit zu geben, zu dem Objekt Zusatzinformationen darstellen zu können, kann die Methode *getPosition* in den konkreten Modellen implementiert werden. Sie gibt die aktuelle Position des Objektes in Roboterkoordinaten zurück.

Analog zu den Perzepten können auch die Modelle über ihren Weltmodellschlüssel bei WorldData angefordert werden. Dabei wird unterschieden, ob diese als Eingabe benötigt oder von einer Modellierung bereitgestellt werden.

Die zur Berechnung notwendigen Daten können über die Templateklasse *InModelData* (siehe Abbildung 6.9) eingebunden werden. Diese kapselt einen RoboFrame-InBuffer, der das Modell bereitstellt. Dieser InBuffer kann mittels *getBuffer* abgerufen werden, sodass die vollständige Funktionalität des Buffers zur Verfügung steht. Damit das Schnittstellenobjekt *Model* aus dieser Templateklasse geladen werden kann, steht die Methode *getModel* zur Verfügung, die dieses Objekt in Form der Basisklasse zurückgibt.

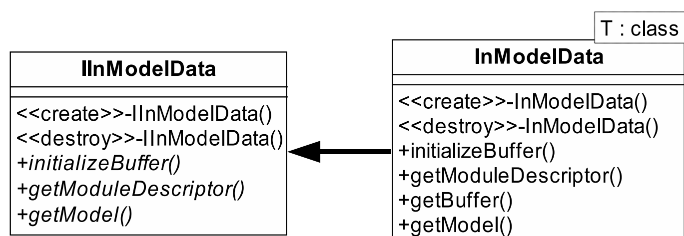

Abbildung 6.9: Eingehende Modelle in der zentralen Datenverwaltung

Wenn dagegen ein berechnetes Modell anderen Komponenten zur Verfügung gestellt werden soll, wird die Templateklasse *OutModelData* (siehe Abbildung 6.10) benötigt. In ihr ist ein RoboFrame-OutBuffer gekapselt, der das jeweilige Modell anderen Frameworkmodulen zugänglich macht. Die Methode *add* ermöglicht es, ein Model direkt in den Buffer zu schreiben.

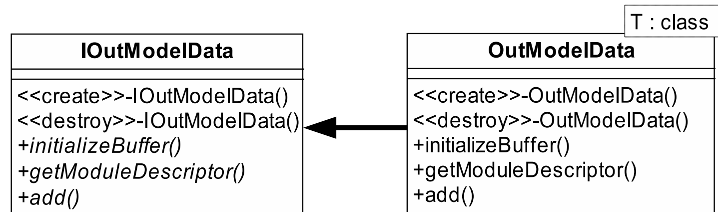

Abbildung 6.10: Ausgehende Modelle in der zentralen Datenverwaltung

Die für das RoboFrame benötigten Bufferinitialisierungen und -registrierungen werden von WorldData durch Aufruf der Methoden *initializeBuffer* und *getModuleDescriptor* vorgenommen. Die beiden Schnittstellen *IInModelData* und *IOutModelData* ermöglichen den Komponenten der Weltmodellierung, Modelldaten eines unbekannten Typs aus einem InBuffer zu laden beziehungsweise ein solches Objekt in einen OutBuffer zu schreiben.

6.1.3.3 Teammodelle

Kommunizierte Modelle eines anderen Roboters werden als konkrete, in der Weltmodellierung bekannte Modelle gespeichert. Da neben den reinen Modelldaten zusätzliche Informationen, wie beispielsweise die Roboternummer sowie Metainformationen über die einzelnen kommunizierten Attribute benötigt werden, wird dieses Modell in der Templateklasse *Team* gekapselt, die diese Informationen bereitstellt (siehe Abbildung 6.11). Aufgrund der allgemeinen Datenhaltung und der Anzeige der Modelle in der GUI ist auch hier eine Schnittstelle nötig (*ITeam*), die die allgemeinen Informationen des Teammodells zur Verfügung stellt, ohne den konkreten Typ des Objektes kennen zu müssen. Sie wurde von IStreamable abgeleitet, um über die Kommunikationsmechanismen von RoboFrame verschickt werden zu können.

Um ein kommuniziertes Modell aus ITeam auslesen zu können, wurde die *getBaseModel* Methode entwickelt, welche dieses Modell in Form der Basisklasse *Model* zurück gibt. Ein Modell des konkreten Typs lässt sich aus der Templateklasse Team auslesen, wofür die Methode *getModel* zur Verfügung steht. Mit *setModel* kann ein Modell in ein Objekt der Teamklasse gesetzt werden.

Da es bei der Kommunikation vorkommen kann, dass manche Modelle nur teilweise ausgetauscht werden, ist es möglich, dass Teammodelle Attribute besitzen, deren Werte nicht von der Kommunikation gesetzt wurden und sich somit im initialen Zustand befinden. Die Methode *isDummyAttribute* gibt diese Information zu einem Attributidentifier

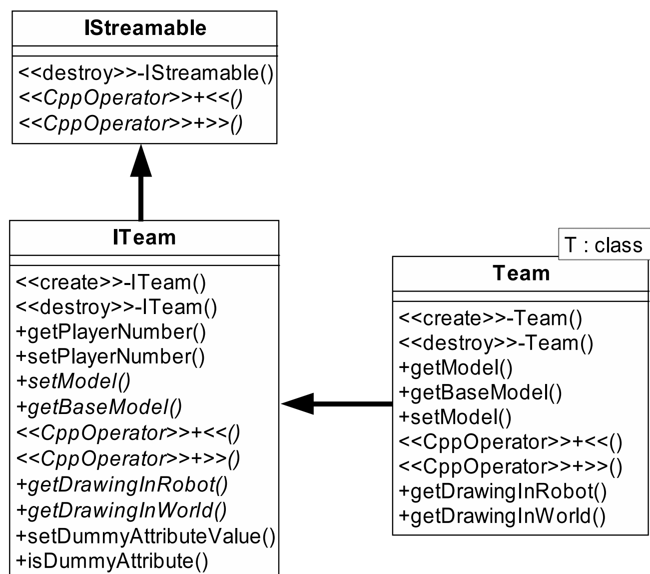

Abbildung 6.11: Teammodelle der Weltmodellierung

zurück, die zuvor über *setDummyAttributeValue* gesetzt werden konnte.

Um im Dialog der Weltmodellierung auch Teammodelle darstellen zu können, müssen analog zu Model die Methoden *getDrawingInRobot* und *getDrawingInWorld* überschrieben werden. Dafür werden die gleichnamigen Methoden des Modells aufgerufen und zusätzliche Informationen wie die Spielernummer hinzugefügt.

Das Konzept der ein- und ausgehenden Datenobjekte wurde hier fortgesetzt, sodass die Teammodelle in *In-* (siehe Abbildung 6.12) und *OutTeamModel*-Klassen (siehe Abbildung 6.13) gekapselt werden. Die für das RoboFrame benötigten Bufferinitialisierungen und -registrierungen werden auch hier von WorldData durch Aufruf der Methoden *initializeBuffer* und *getModuleDescriptor* vorgenommen.

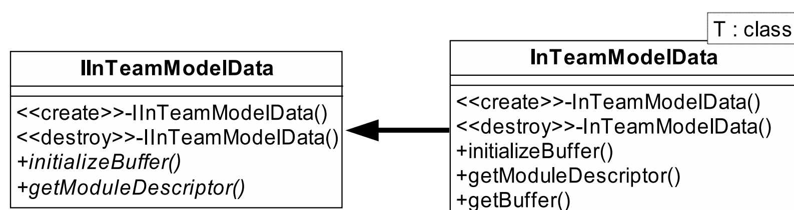

Abbildung 6.12: Eingehende Teammodelle in der zentralen Datenverwaltung

Die Templateklasse verfügt über einen RoboFrame-InBuffer des Templatetyps, welcher die Daten des Teammodells beinhaltet. Dieser InBuffer kann mittels *getBuffer* abgerufen werden, sodass analog zu den Modellen alle Funktionen des Buffers zur Verfügung stehen. Diese Funktionalität kann nur von einem konkreten InTeamModelData-Objekt genutzt werden, da dem allgemeinen Schnittstellenobjekt der Typ des Buffers

nicht bekannt ist.

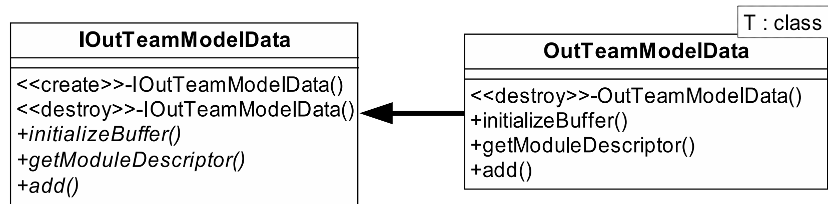

Abbildung 6.13: Ausgehende Teammodelle in der zentralen Datenverwaltung

Die OutTeamModelData-Objekte werden von der Kommunikation genutzt, um mittels der Funktion *add* die empfangenen Teammodelle in den jeweiligen OutBuffer zu schreiben. Hierdurch können die Modellierungen auf diese Teammodelle in Form eines InTeamModelData-Objektes zugreifen können.

6.1.4 Modellierungsverwaltung

Die in der Weltmodellierung hinzugefügten Modellierungen werden vom ModelingHandler (siehe Abbildung 6.14) verwaltet. Dieser steuert unter Verwendung des Template-Method-Pattern [9] den Lebenszyklus einer Modellierung und stellt ihr Metainformationen wie den Start- und Endezeitstempel der letzten Ausführung zur Verfügung.

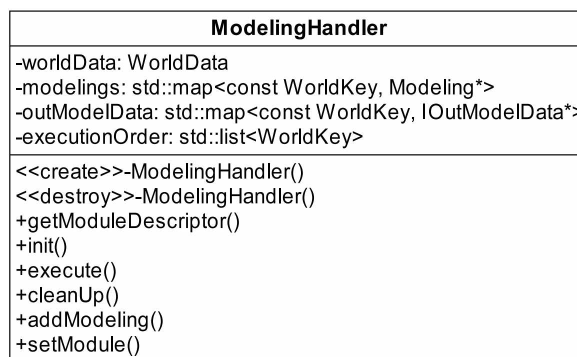

Abbildung 6.14: Verwaltung und Steuerung der Modellierungen

Bei der Registrierung einer Modellierung im ModelingHandler werden dieser neben der Referenz auf WorldData alle von der Modellierung genutzten Schlüssel als Konstruktorparameter übergeben. Anhand des von der Modellierung repräsentierten Schlüssels wird aus WorldData ein OutModelData Objekt geladen, welches dem ModelingHandler das Schreiben der von den Modellierungen berechneten Modelle in die jeweiligen OutBuffer ermöglicht. Nach dem Hinzufügen der Modellierung über die Methode

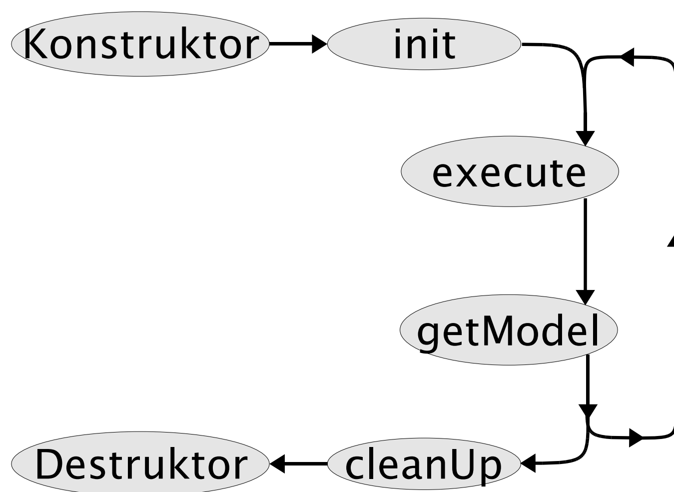

Abbildung 6.15: Lebenszyklus von Modellierungen

addModeling wird der Lebenszyklus durch den Konstruktor-Aufruf gestartet (siehe Abbildung 6.15).

In der *init*-Methode legt der ModelingHandler eine Aufrufreihenfolge anhand der von den Modellierungen zur Berechnung benötigten Modelle fest. Die Informationen, die zu dieser Berechnung nötig sind, können aus WorldData mit der Methode *getInputModels* abgerufen werden. Bei der Berechnung dieser Reihenfolge wird darauf geachtet, dass Modellierungen, die Modelle anderer Modellierungen in ihre Berechnungen einbeziehen, nach diesen Modellierungen aufgerufen werden, damit die benötigten Modelle den aktuellen Stand haben. Falls zwei Modellierungen das Modell der jeweils anderen benötigen, ist es nicht möglich, beiden Modellierungen das aktuelle Modell zu liefern und die zuerst dem ModelingHandler Hinzugefügte wird vor der anderen Modellierung aufgerufen. Diese konkurrierende Anfrage wird mittels eines Logeintrages des Levels *Warning* ausgegeben. In der nun feststehenden Aufrufreihenfolge werden die Initialisierung der Modellierungen über die Methode *init* vorgenommen.

Das zyklische Aufrufen der Modellierungen erfolgt in der durch *WorldModelModul* aufgerufenen Funktion *execute* des ModelingHandlers. Vor und nach jedem Aufruf der *execute*-Methode wird der aktuelle Zeitstempel gespeichert, mit welchem nach der Ausführung einer Modellierung deren Laufzeit berechnet werden kann. Diese Informationen werden in der Schnittstelle der Modellierung gespeichert und können zur dynamischen Optimierung genutzt werden. Nach jeder Ausführung einer Modellierung wird das berechnete Model mit der Methode *getModel* in das jeweilige OutModelData-Objekt gespeichert und damit den anderen Komponenten zur Verfügung gestellt.

Wird die Anwendung beendet, ruft RoboFrame vor der Zerstörung der Objekte durch ihren *Destruktor* die Methode *cleanUp* auf den Modulen auf. Über WorldModelModule wird die Methode *cleanUp* des ModelingHandlers aufgerufen, welcher die gleichnamige Methode auf den einzelnen Modellierungen in der Aufrufreihenfolge ausführt. Mit dem Aufruf der Destruktoren (ebenfalls in der berechneten Reihenfolge) endet der Le-

benszyklus der Modellierungen.

6.1.4.1 Modellierungen

Die Klasse *Modeling* ist die Basisklasse aller Modellierungen (siehe Abbildung 6.16). Sie beinhaltet alle von den Modellierung benötigten Objekte und Informationen und gibt die für den Lebenszyklus zu implementierenden Funktionen vor.

Modeling
<<create>>-Modeling() <<destroy>>-Modeling() +getResultKey() *+init()* *+execute()* *+cleanUp()* *+getModel()* +getName() +getModuleDescriptor() +setCurrentStart() +getCurrentStart() +setLastStart() +getLastStart() +setLastFinished() +getLastFinished()

Abbildung 6.16: Schnittstelle für Modellierungen

Jede Modellierung hat einen Schlüssel, der im Konstruktor übergeben wird und das berechnete Modell identifiziert. Die Modellierungsverwaltung kann diesen Schlüssel über die Methode *getResultKey* abfragen und das von der Klasse berechnete Modell zuordnen.

Die von der Modellierungsverwaltung gesetzten Zeitstempel repräsentieren den aktuellen Aufrufzeitpunkt sowie den letzen Aufruf- und Endezeitpunkt. Um den Modellierungen den Zugriff auf diese Informationen geben zu können, existieren Get- und Set-Methoden in der Klasse Modeling. Mittels dieser Information kann eine Modellierung dynamische Optimierungsverfahren nutzen, um die Berechnungsmethoden bezüglich ihrer Laufzeit zu variieren.

Die Laufzeit ist außerdem im Chronometer-Dialog der GUI anzeigbar, da die Modellierungsverwaltung die zu diesem Zweck bereitgestellten Makros aufruft. Der in diesem Dialog angezeigte Name wird über *getName* von den Modellierungen bereitgestellt. Um weitere Buffer in den Modellierungen registrieren zu können, die zu Debugzwecken in der GUI genutzt werden können, kann die Funktion *getModuleDescriptor* überschrieben werden.

In den folgenden Abschnitten werden Modellierungen beschrieben, die aufbauend auf dem neuen Framework im Rahmen dieser Arbeit entwickelt wurden.

6.1.4.1.1 Verhaltensmodellierung

Die Verhaltensmodellierung (*BehaviorModeling*) berechnet aus den Daten der kommunizierten Verhaltensmodelle, seinem eigenen *BallModel* und dem *TeamBallModel* seiner Mitspieler seine eigene Rolle bei einem 2-gegen-2-Fußballspiel (siehe Abbildung 6.17). Diese Information sowie die initiale Rolle des Spielers werden vom Verhalten genutzt, um den für diese Rolle relevanten Zustandsteilgraphen auszuführen.

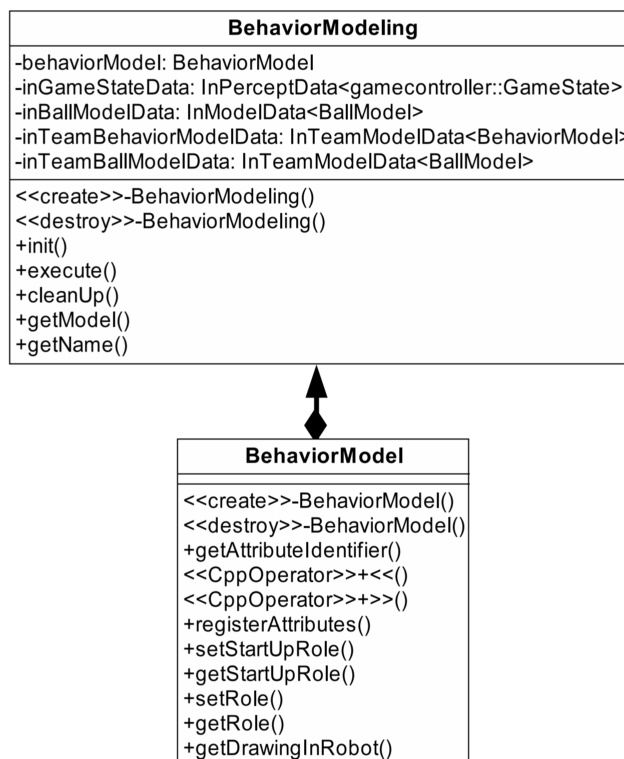

Abbildung 6.17: Verhaltensmodellierung zur Steuerung der Spielerrolle

Die in dieser Arbeit entwickelte Verhaltensmodellierung bietet eine Rollenwechselstrategie für Teams, bestehend aus einem Stürmer und wahlweise einem Torwart oder Abwehrspieler. Da das Rollenverhalten eines Fußballteams abhängig von der gewählten Strategie ist, muss diese Modellierung bei einer Strategieänderung oder bei einer Vergrößerung des Teams angepasst werden.

Es wird vorausgesetzt, dass das Verhalten eines Stürmers immer zum Ball strebt und sein Ziel das Schießen von Toren ist. Aus diesem Grund wird ein Roboter bei beiden Strategien immer Stürmer, sobald er nach einer gewissen Zeit (Timeout) keine Rolle seines Mitspielers empfangen hat. Dies stellt sicher, dass auch ein Torwart oder Abwehrspieler zum Stürmer wird, sobald die Kommunikation gestört oder er der einzige Spieler auf dem Feld ist. Da beide Rollenwechselstrategien vom kommunizierten und vom eigenen Ballmodell abhängig sind, werden diese vor ihrer Verwendung bezüglich

der Wahrscheinlichkeit geprüft, die angibt, ob diese Ballinformationen zu verwenden sind.

Bei einem Mannschaftsspiel mit einem Abwehrspieler und einem Stürmer wird die Rolle zwischen den beiden Spieler abhängig von deren Ballentfernung gewechselt. Derjenige Spieler, der näher am Ball ist, wird zum Stürmer, der andere zum Abwehrspieler. Damit es bei einer leicht oszillierenden Ballentfernung und ähnlichen Entfernungen zweier Roboter zum Ball nicht zu einem ständigen Rollenwechsel kommt, wurde eine Hysterese entwickelt, bei der die Rolle höchstens alle zwei Sekunden gewechselt wird. Dies hat zur Folge, dass es kurzfristig zwei Stürmer im Spiel geben kann.

Bei dem Rollenwechsel zwischen einem Torwart und einem Stürmer wird die Rolle der Spieler erstmalig gewechselt, wenn der Torwart nach einem Timeout keine Rolle seines Mitspielers empfängt. In diesem Fall wird der Torwart zum Stürmer, damit die Möglichkeit erhalten bleibt, Tore zu schießen und die Gegner am Ball zu stören. Empfängt dieser Spieler wieder eine Rolle des ehemaligen Stürmers, bleibt er solange Stürmer, wie er näher am Ball ist. Damit sich die Teamspieler nicht gegenseitig in die Quere kommen, wird der wieder hinzugekommene, ehemalige Stürmer zum Abwehrspieler. Der nächste Rollenwechsel findet erst wieder statt, wenn der ehemalige Torwart weiter vom Ball entfernt ist als der ehemalige Stürmer. In dieser Situation wechseln die Rollen in den Ausgangszustand zurück, und jeder Spieler übernimmt wieder die initiale Rolle.

Um den Rollenwechsel einfach debuggen zu können, wird mittels der überschriebenen Methoden *getDrawingInRobot* die derzeitige Rolle zu der aktuellen Roboterposition im RoboGUI-Dialog *ModelViewer* dargestellt. Da sich in diesem Dialog auch die Rollen und alle anderen Modelle der Mitspieler anzeigen lassen, kann ein strategischer Rollenwechsel grafisch nachvollzogen werden.

6.1.4.1.2 Odometriemodellierung

Die Odometriemodellierung (*OdometryModeling*) bereitet die Odometriedaten, die in Form des Odometrieperzepts bereitgestellt werden, für die anderen Modellierungen auf. Dieses Perzept wird vom Bewegungsmodul bei jedem *Motion*-Prozessdurchlauf zur Verfügung gestellt. Dabei werden die Bewegungsdaten gespeichert, die aufgrund der Motorenbewegungen berechnet wurden. Diese Daten werden von den Modellierungen benötigt, um ihr Modell auch dann richtig positionieren zu können, wenn es im aktuellen Bild nicht erkannt wurde. In diesem Fall kann die Modellierung anhand der letzten Position und der vorliegenden Odometrie die aktuelle Position des Objektes berechnen.

Da die Modellierungen in einem anderen Prozess (*Cognitionprozess*) als das Bewegungsmodul (*Motionprozess*) laufen und letzterer eine geringere Frequenz besitzt, musste bisher jede Modellierung bei ihrem Aufruf mehrere Odometrieperzepte aufsummieren. Um diese Mehrfachberechnung zu vermeiden, wurde eine Odometriemodellierung entwickelt, die alle Odometrieperzepte eines Cognitionprozesses aufaddiert und über das *OdometryModel* bereitstellt (siehe Abbildung 6.18).

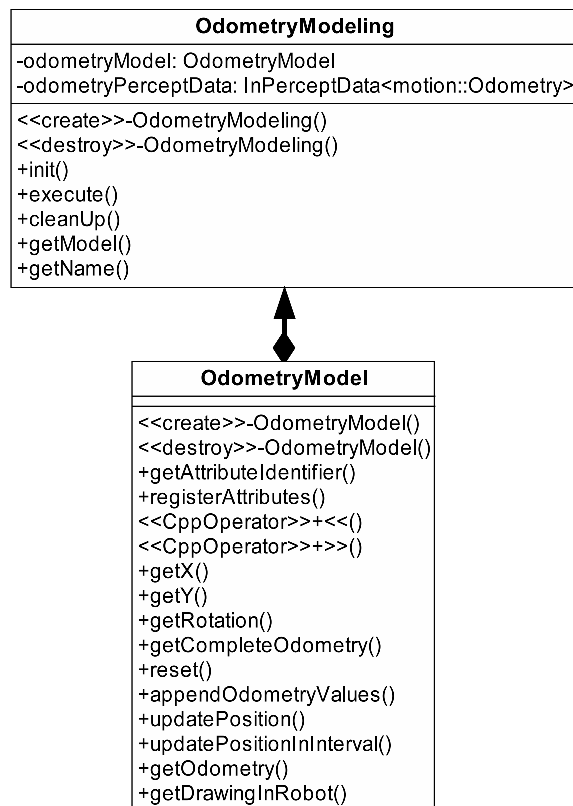

Abbildung 6.18: Odometriemodellierung

Das OdometryModel wird bei jedem Aufruf der Odometriemodellierung über die Methode *reset* zurückgesetzt, wodurch alle Perzeptinformationen des letzten Aufrufs gelöscht werden. Im zweiten Schritt werden anschließend über die Methode *appendOdometryValues* die aktuellen Odometrieperzepte einzeln dem Odometriemodell hinzugefügt. Bei jedem Aufruf wird das aktuelle Perzept zur bisherigen Odometriesumme addiert und zum aktuellen Perzept-Zeitstempel gespeichert. Diese Aufsummierung erfolgt vom “jüngsten“ bis zum “ältesten“ Perzept, sodass bei Abfrage der Odometrie ab einem bestimmten Zeitpunkt keine zusätzlichen Berechnungen seitens des Modells gemacht werden müssen.

Um die berechneten Werte abzufragen, stehen die Methoden *getX, getY und getRotation* bereit, denen jeweils ein Zeitstempel übergeben werden muss, damit diese die Odometry in X, Y und die Rotation ab diesem Zeitpunkt zurückgeben können. Um diese drei Informationen in Form eines einzigen Objekts über eine Methode abfragen zu können, werden die Methode *getOdometry*, die die Odometrie ab einem übergebenen Zeitpunkt zurückgibt, beziehungsweise *getCompleteOdometry*, die die aufsummierte Odometrie aller Perzepte liefert, bereitgestellt.

Um die von einer Modellierung benötigte Odometrie von dem OdometryModel aufaddieren zu lassen, stehen die polymorphen Methoden *updatePosition* und *updatePosi-*

tionInInterval zur Verfügung. Dabei fügt updatePosition der übergebenen Position die Odometrie bis zum aktuellen Wert hinzu, optional ist dies auch ab einem bestimmten Zeitstempel möglich. Die Methode updatePositionInInterval ist ähnlich aufgebaut, allerdings fügt sie nicht die Odometrie bis zum aktuellen, sondern bis zu dem übergebenen Zeitstempel hinzufügt. Auch hier kann optional die Entwicklung ab einem bestimmten, vorgegebenen Wert abgefragt werden.

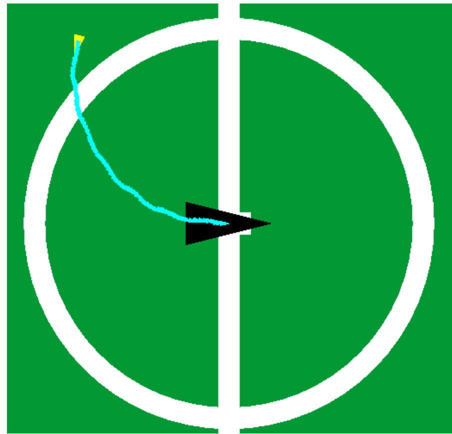

Abbildung 6.19: Darstellung des Odometriemodells in der GUI

Die komplette Odometrie eines Cognitionprozess-Durchlaufs wird mittels der dafür vorgesehenen Methode abgerufen und im Weltmodellierungsdialog gezeichnet (siehe Abbildung 6.19).

6.1.5 Modellierungsfunktionen

Um den Modellierungen Standardfunktionen zur Verfügungen stellen zu können, wurde die Klasse *ModelingOperations* entwickelt, welche ausschließlich aus statischen Methoden besteht. Diese können somit von den Modellierungen genutzt werden, ohne die Klasse instantiieren zu müssen (siehe Abbildung 6.20). Durch die Zentralisierung solcher Standardfunktionen wird doppelter Programmcode vermieden und die Fehlersuche erleichtert.

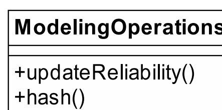

Abbildung 6.20: Statische Modellierungsfunktionen für Modellierungen

Die Erfahrungen bei der Entwicklung von Robotersteuerungsmodulen haben gezeigt, dass unterschiedliche Implementierungen der Wahrscheinlichkeit eines Modells, welche die Vertrauenswürdigkeit repräsentiert, zu Problemen bei der Verwendung im Verhalten führen. Um diese Unstimmigkeiten zu vermeiden sowie eine von der Taktrate

der Anwendung unabhängige Berechnung der Wahrscheinlichkeit eines Modells in der Modellierung zu ermöglichen, wurde die Funktion *updateReliability* entwickelt. Diese Methode berechnet anhand des Zeitstempels der letzten Wahrscheinlichkeitsaktualisierung, dem aktuellen Wahrscheinlichkeitswert sowie der Lebenszeit eines Modells den neuen Wahrscheinlichkeitswert. Ist die Lebenszeit eines Modells abgelaufen, beträgt die Wahrscheinlichkeit Null. Das Modell ist demnach nicht vertrauenswürdig und sollte von anderen Komponenten der Anwendung (wie dem Verhalten) nicht weiter in Entscheidungen einbezogen werden. Da die komplette Aktualisierung dieses Wertes auf den Zeitstempeln der Anwendung basiert, ist die Berechnung taktunabhängig.

Die Funktion *hash* berechnet anhand einer Zeichenkette einen Hashwert, welcher eine maximale, vom Aufrufer definierte Länge hat. Diese Funktion wird derzeit von der Kommunikation genutzt, um die kommunizierten Modelle anhand des Hashwerts eindeutig identifizieren zu können.

Es empfiehlt sich auch in Zukunft, den Programmcode, der in mehreren Modellierungen zum Einsatz kommt, an diese Stelle in eine statische Methode auszulagern.

6.1.6 GUI-ModelViewer

Um die Modelle als Ergebnisse der Berechnungen in den Modellierungen debuggen zu können, wurde der Dialog *ModelViewer* als Teil der RoboGui entwickelt (siehe Abbildung 6.21).

Dieser bestehende Dialog wurde auf die neue Weltmodellierung angepasst und mit neuen Funktionalitäten erweitert. Der ModelViewer bot bisher die Möglichkeit, Perzepte und Modelle auf einem Spielfeld anzuzeigen. Mit der neuen Weltmodellierung musste die Modellanzeige angepasst und der Dialog um die Teammodellvisualisierung erweitert werden.

Das Spielfeld gibt dem Entwickler einer Modellierung einen schnellen und einfachen Überblick über die berechneten Daten seines Modells und hilft der Verhaltensentwicklung, die Eigenschaften eines Modells in die eigene Entscheidungsfindung einfließen zu lassen. Werden alle Modelldaten der Weltmodellierung in diesem Dialog angezeigt, kann dies mit einer momentanen Kartografierung der Spielsituation verglichen werden.

Damit sich die Objekte (wie Modelle oder Teammodelle) in diesem Dialog anzeigen lassen, müssen die Methoden *getDrawingInRobot* beziehungsweise *getDrawingInWorld* überschrieben werden, was bereits im Abschnitt 6.1.3.2 für Modelle und im Abschnitt 6.1.3.3 für Teammodelle beschrieben wurde. Diese Methoden stellen die Modelle und Teammodelle in Roboter- beziehungsweise Weltkoordinaten dar. Die für ein Teammodell spezifischen Informationen (wie die Spielernummer) werden von der Teamschnittstelle beim Aufruf der jeweiligen Methode vor jedes Teammodell gezeichnet (siehe Abbildung 6.22).

Da die verschiedenen kommunizierten Modelle bei einer übermittelten Mitspielerposition (*ValidityPoseModel*) relativ zu dieser gezeichnet werden sollen, werden die Team-

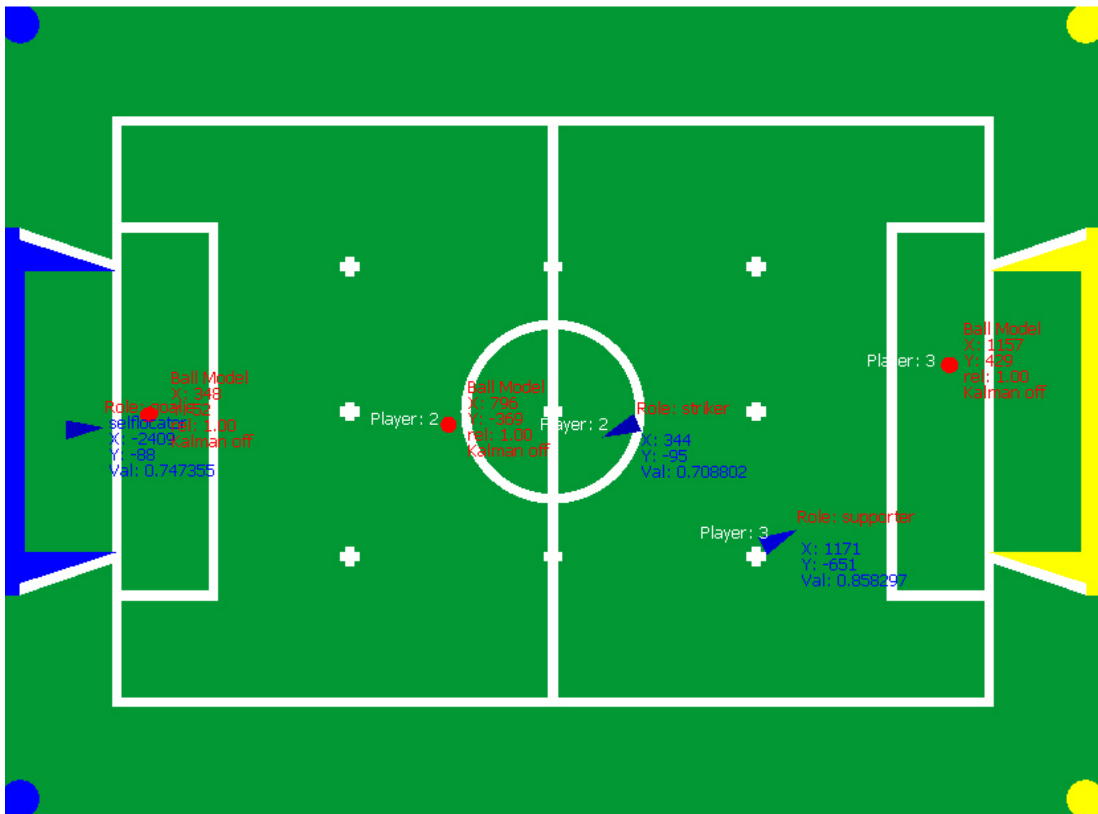

Abbildung 6.21: Anzeige von Modellen und Teammodellen im ModelViewer

modelle anhand der Spielernummer dem ValidityPoseModel zugeordnet und relativ zu diesem dargestellt.

Um die auf dem virtuellen Spielfeld anzuzeigenden Daten auswählen zu können, hat der Anwender die Möglichkeit, die auf der linken Seite getrennt nach Modellen (siehe Abbildung 6.23) und Teammodellen (siehe Abbildung 6.24) gruppierten Auswahlfelder zu selektieren.

Die Möglichkeit, Objekte anhand ihrer Elternklasse ordnen zu können, wird von RoboFrame bereitgestellt und für die Gruppierung der Daten in diesem Dialog genutzt. Die allgemeine Behandlung von Weltmodelldaten ermöglicht eine Erweiterbarkeit der Weltmodellierung, ohne den ModelViewer bezüglich der Anzeige von neuen Modell- beziehungsweise Teammodell-Objekten anpassen zu müssen.

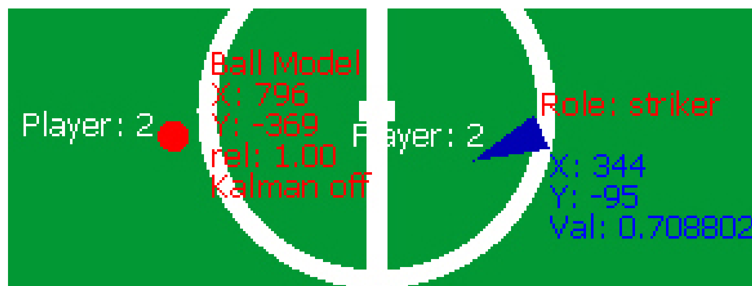

Abbildung 6.22: Anzeige eines Teammodells im ModelViewer

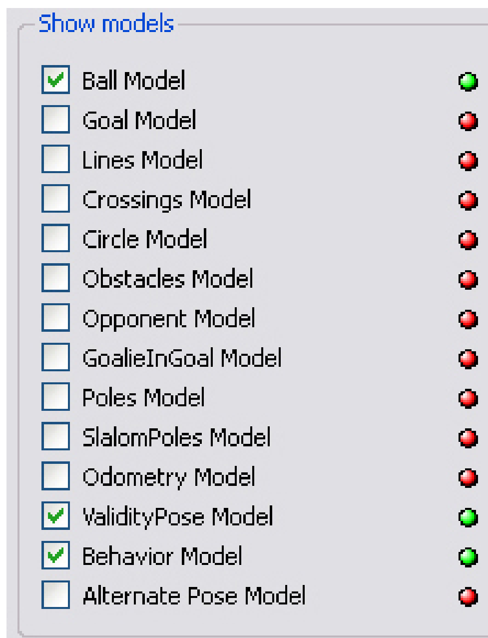

Abbildung 6.23: Auswahl der Anzeige der Modelle

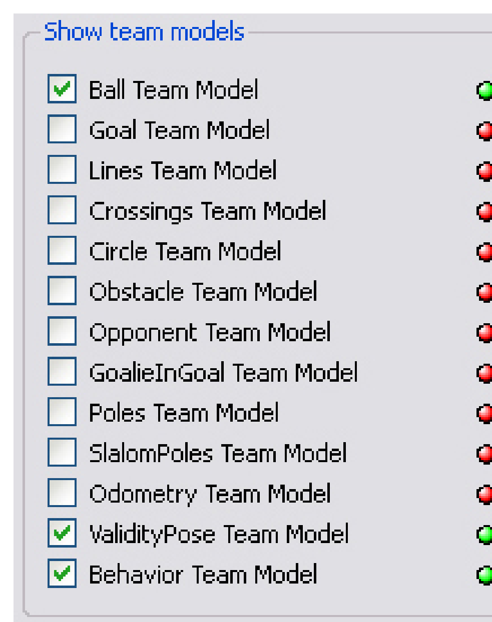

Abbildung 6.24: Auswahl der Anzeige der Teammodelle

6.2 Kommunikation

Bei der Definition der Ziele und der Entwicklung des Konzepts für die heterogene Kommunikation (Abschnitt 5.2) wurde gefordert, diese effizient und flexibel zu gestalten, sodass bei einer Erweiterung der Weltmodellierung durch neue Modelle keine Anpassungen erfolgen müssen. Hierfür wird jedes Modell mit einem Identifier registriert, welcher dieses Modell eindeutig und intuitiv beschreibt. Neben den Modell-Identifiern hat jedes Attribut eines Modells einen Identifier, welcher dieses innerhalb des Modells eindeutig kennzeichnet. Das daraus resultierende lokale Weltmodell wird mit anderen Robotern ausgetauscht, sodass ein globales Weltmodell gebildet werden kann. Bei diesem globalem Weltmodell kann es zu verschiedenen Kombinationen aus verwendeten Modellen und Attributen kommen. Die für die Kommunikation zu beachtenden Konstellationen sind:

1. Alle Modelle mit allen Attributen bilden das globale Schnittmodell (globales Weltmodell).
2. Es sind nicht alle Modelle der lokalen Weltmodellierung im globalen Weltmodell enthalten.
3. Es sind nicht alle Attribute eines lokalen Modells im globalen Weltmodell vorhanden.
4. Es ist ein Modell im globalen Schnittmodell enthalten, welches der lokalen Weltmodellierung unbekannt ist.
5. Es sind Attribute in einem Modell enthalten, welche dem lokalen Modell unbekannt sind.

Um diese Konstellationen abdecken zu können, müssen teilweise empfangene Modelle oder Attribute übersprungen werden, Attribute eigener Modelle im initialen Zustand gelassen und für unbekannte Attribute Platzhalterwerte gesendet werden. Die empfangenen (Teil-)Teammodelle werden den Modellierungen über die zentrale Datenverwaltung zur Verfügung gestellt.

Im Folgenden werden alle im Konzept erwähnten Komponenten der Kommunikation beschrieben, ihre Beziehungen untereinander dargestellt und wird detailliert auf die Implementierung eingegangen.

6.2.1 Kommunikationsverwaltung

Die Verwaltung der Kommunikation wird von der Klasse *Communication* übernommen, die von der Klasse *ICommunication* abgeleitet ist und somit deren vorgegebene, abstrakte Methode definiert (siehe Abbildung 6.25). Die Klasse ICommunication bildet ein Interface für mögliche konkrete Kommunikationsimplementierungen und ist als Klassenvariable in WorldModelModule eingebunden. Die erbende Klasse Communication muss wie die Modellierungen im Konstruktor von WorldModelModule07 erzeugt und der Communication-Variablen zugewiesen werden.

Bei der Instantiierung der Kommunikationsverwaltung muss eine Referenz auf die zentrale Datenklasse WorldData sowie die Roboter- und Teamnummer übergeben werden. Diese Parameter werden vom Konstruktor der Basisklasse ICommunication vorgegeben und in dieser gespeichert. Die ebenfalls im Konstruktor zu übergebenden RoboFrame-Schlüssel sind zum Empfang und Versand von Struktur und Daten über Buffer nötig.

Damit die Daten der Buffer über die vorgesehenen Ports verschickt und empfangen werden können, müssen sie in Form von Konnektoren, die über *getConnectors* abgerufen werden können, der Hauptanwendung hinzugefügt werden. Diese Konnektoren, die in der Klasse TeamConnector implementiert sind, bilden die Schnittstelle zur Netzwerkkommunikation.

Abbildung 6.25: Schnittstelle der Kommunikation

In der Methode *getModuleDescriptor* werden die Buffer der Kommunikation zur Applikation hinzugefügt und für alle registrierten Modellierungen InModelData- und OutTeamModelData-Objekte angelegt. Über diese Objekte ruft die Kommunikation die zu sendenden Modelle ab beziehungsweise stellt die empfangenen Teammodelle zur Verfügung. In der Methode *init* werden Initialisierungen von der Komponente vorgenommen.

Die eigentliche Logik der Kommunikation ist in den von der Kommunikationsschnittstelle vorgegebenen Methoden *send* und *receive* enthalten. Die Versendung der Daten ist zeitabhängig realisiert und wird von WorldModelModule alle 500 ms gestartet, wobei der Empfang in jedem Takt ausgeführt wird, damit den Modellierungen zu jeder Zeit die aktuellen Daten zur Verfügung stehen. Die genaue Funktionsweise der Methoden wird in Abschnitt 6.2.3 und in Abschnitt 6.2.4 detailliert beschrieben. Die für diese Vorgänge benötigten Informationen sind in eine kommunikationsspezifische Datenklasse (*CommunicationData*) ausgelagert, die im folgenden Abschnitt erläutert wird.

Um das standardmäßig aktive Empfangen und Senden der Daten zu Debugzwecken unterbinden zu können, kann das Streamable-Objekt *CommunicationParameters* genutzt werden. Damit die Anwendung in Testsituationen ohne den Einfluss von kom-

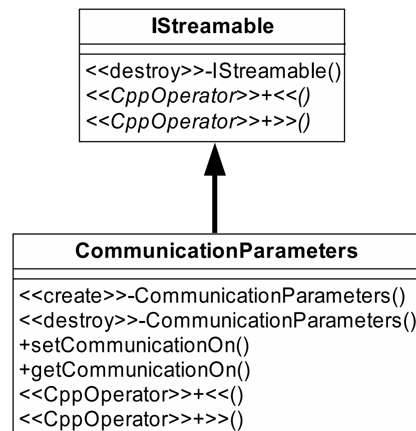

Abbildung 6.26: Streamable zur Steuerung der Kommunikation über die GUI

munizierten Modellen anderer Systeme getestet werden kann, lässt sich diese in der GUI (siehe Abschnitt 6.1.6) deaktivieren beziehungsweise zu einem späteren Zeitpunkt wieder aktivieren. Der in CommunicationParameters enthaltene boolsche Wert wird von WorldModelModule ausgelesen und der Aufruf der Kommunikation dementsprechend gesteuert.

6.2.2 Datenverwaltung

Die in diesem Abschnitt beschriebene Datenverwaltung *CommunicationData* (siehe Abbildung 6.27) speichert die für die Kommunikation relevanten Daten und bietet Methoden für zusätzliche Berechnungen und Auswertungen an. Zu den in CommunicationData gespeicherten Daten gehört die lokale Weltmodellstruktur, die durch das Hinzufügen der Modellidentifier aufgebaut wird. Dabei werden mit *addModel* Zeichenketten, die die Modelle und ihre Attribute eindeutig identifizieren, in einer Datenstruktur abgelegt. Diese Datenstruktur kann über die Methode *getWorldModel* abgerufen und über einen Konnektor verschickt werden.

Außer der eigenen Weltmodellstruktur werden auch Strukturen anderer Roboter in dieser Klasse gespeichert. Diese können über die Methoden *addPlayerStructure* hinzugefügt beziehungsweise über *deletePlayer* gelöscht werden. Um abfragen zu können, welche Strukturen der Mitspieler in das globale Modell integriert sind, steht die Methode *getConnectedPlayers* zur Verfügung. Eine solche Abfrage kann auch mittels *isCommunicatedPlayerKnown* für eine konkrete Spielernummer erfolgen.

Aus der eigenen sowie der Struktur der verbundenen Mitspieler wird in *calculateGlobalWorldModel* das globale Weltmodell berechnet, welches über die Methode *getGlobalWorldModel* zur Verfügung steht. Es besteht aus allen Attributen, die von mindestens zwei Robotersystem verstanden werden. Alle für die Kommunikation vorgesehenen Attribute eines Modells, die in der globalen Struktur nicht enthalten sind, werden in einer

CommunicationData
-ownModels: std::map<const std::string, std::vector<Model::ModelIdentifier> > -connectedPlayer: std::vector<CommunicatedPlayerModel> -mergedModel: std::map<const unsigned int, std::vector<Model::ModelIdentifier> >
<<create>>-CommunicationData() <<destroy>>-CommunicationData() +addModel() +getTypeIdentifier() +getAttributeIdentifier() +getOwnAttributeNumber() +getWorldModel() +getConnectedPlayer() +addPlayerStructure() +calculateGlobalWorldModel() +isCommunicatedPlayerKnown() +deletePlayer() +getGlobalWorldModel() +getNotCommunitatedAttributes() +getModelName() +getCommunicatedDummyAttributes()

Abbildung 6.27: CommunicationData

gesonderten Datenstruktur gespeichert und können über *getNotCommunitatedAttributes* abgerufen werden. Attribute des globalen Weltmodells, die einem Mitspieler unbekannt sind, werden in einer weiteren Datenstruktur für jeden Roboter gehalten. Diese Informationen werden über *getCommunicatedDummyAttributes* bereitgestellt und können von der Kommunikation beim Empfangen der Daten genutzt werden.

Das globale Modell wird für das Versenden und Empfangen der eigentlichen Weltdaten benötigt, da aufgrund dieses Modells das "Transportobjekt" CommunicationObject (siehe Abschnitt 6.2.4) zusammengestellt beziehungsweise ausgepackt wird. Für diese beiden Vorgänge wird die Methode *getAttributePosition* benötigt, da sie die Position eines bestimmten Attributs eines Modells in der Attributliste zurückgibt. Mithilfe dieser Position können Attribute eines Modells ausgelesen beziehungsweise in dieses gesetzt werden.

6.2.3 Kommunikation der Weltmodellstruktur

Die Kommunikation der Weltmodellstruktur wird mit dem Objekt *CommunicationStructure* realisiert (siehe Abbildung 6.28), die von IStreamable erbt. Die für den Versand des lokalen Weltmodells benötigten Daten werden aus der Datenverwaltung der Kommunikation geladen und in diesem Objekt gespeichert. Die betreffenden Informationen umfassen die eigene Spielernummer, die Spielernummern der bekannten Weltmodellstrukturen sowie die eigene Weltmodellstruktur (siehe Abbildung 6.29).

Dieses Streamable wird über den Konnektor versendet und kann von anderen Robotersystemen empfangen werden. Die beim Auslesen aus CommunicationStructure beschriebenen Eigenschaften werden in der für den Empfang zuständigen Methode *receive* geladen und ausgewertet. Die erhaltene Weltmodellstruktur wird gegebenenfalls

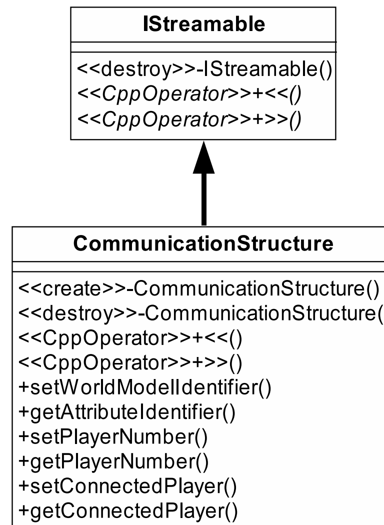

Abbildung 6.28: Streamable zur Kommunikation von Weltmodellstrukturen

	Strukturheader		Struktur eines Modells					
Header	Spielernr.	bekannte Spielernr.	Modell Identifier A	Attribut Identifier X	...	Attribut Identifier Z	...	Modell Identifier C

Abbildung 6.29: Datenpaket zur Kommunikation des lokalen Weltmodells

der Kommunikation über *addPlayerStructure* hinzugefügt.

In der Auswertung wird anhand der übermittelten Spielernummer überprüft, ob dessen Weltmodellstruktur im globalen Weltmodell enthalten ist. Ist seine Spielernummer nicht bekannt, wird die Struktur dieses Roboters der Datenverwaltung der Kommunikation hinzugefügt. Diese Struktur wird auch hinzugefügt, wenn die eigene Weltmodellstruktur dem durch die Spielernummer repräsentierten Roboter unbekannt ist. Dies ist insbesondere bei einer Einwechslung nötig, damit dessen Struktur die Struktur des ausgewechselten Spielers mit identischer Spielernummer überschreibt.

In Abbildung 6.30 ist beispielhaft ein Kommunikationsaufbau zwischen drei Robotern beschrieben. Roboter A und B tauschen zuerst ihre lokalen Strukturen aus und berechnen jeweils das globale Weltmodell AB, aufgrund dessen sie ihre Attributwerte austauschen können. Roboter C kommt zu einem späteren Zeitpunkt hinzu und versendet seine lokale Struktur. Da diese neue Struktur den Robotern A und B unbekannt ist, bilden diese das neue globale Weltmodell ABC und antworten mit ihrer lokalen Struktur. Nachdem auch Roboter C die Strukturen empfangen hat, kann er ebenfalls das globale Weltmodell berechnen, sodass ein Datenaustausch zwischen den drei Robotern stattfinden kann.

Die lokale Struktur wird alle zehn Sekunden versendet. Dieser Wert ist in der execute-Methode von WorldModelModule konfigurierbar. Empfängt der Roboter die Struktur

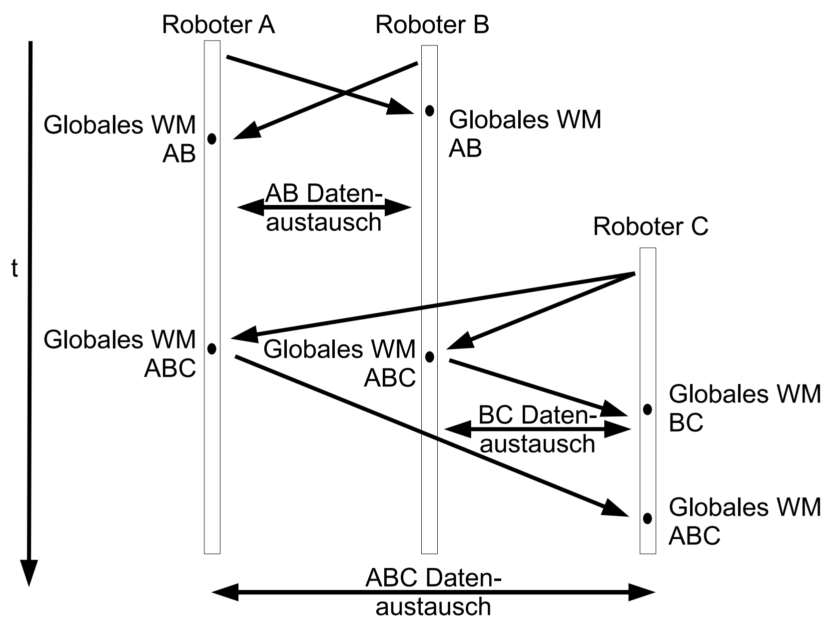

Abbildung 6.30: Ablauf bei der Kommunikation der Weltmodellstrukturen dreier Roboter

mindestens eines unbekannten Spielers, wird die eigene Struktur unmittelbar nach der Verarbeitung aller empfangenen Daten versendet. Das durch diese Verarbeitung entstandene globale Weltmodell dient als Grundlage für den Versand der Attributwerte und wird im folgenden Abschnitt näher beschrieben.

6.2.4 Kommunikation der Weltmodellattribute

Für die Kommunikation der Werte der Weltmodellattribute wird die Klasse *CommunicationObject* benötigt (siehe Abbildung 6.31). Einer Instanz dieser Klasse werden in *send* die zu versendenden Daten hinzugefügt, welche die eigene Spielernummer, die Attributwerte und die dazu gehörenden Attributtypen umfassen (siehe Abbildung 6.32). Diese Daten werden von CommunicationObject genutzt, um einen Stream aufzubauen, welcher mit der Spielernummer beginnt und dem anschließend abwechselnd ein Hashwert des Modells und die von diesem Modell zu sendenden Attributwerte hinzugefügt wird. Dieser Hashwert repräsentiert das Modell, sodass beim Empfang der Modelle deren Übereinstimmung mit dem globalen Weltmodell überprüft werden kann. Bei diesem Vorgang müssen alle fünf in Abschnitt 6.2.1 beschriebenen Fälle beachtet werden, damit eine heterogene Kommunikation ermöglicht wird.

Bei den Fällen eins bis drei ist das globale Weltmodell vollständig im lokalen Modell enthalten, wodurch alle für die Kommunikation relevanten Daten im eigenen Weltmodell enthalten sind. Das resultierende Kommunikationsobjekt wird anhand der Attributinformationen aus dem Schnittmodell gebildet, indem anhand dieser Identifier die Attributinformationen aus den jeweiligen Modellen geladen werden. Im vierten Fall wird das

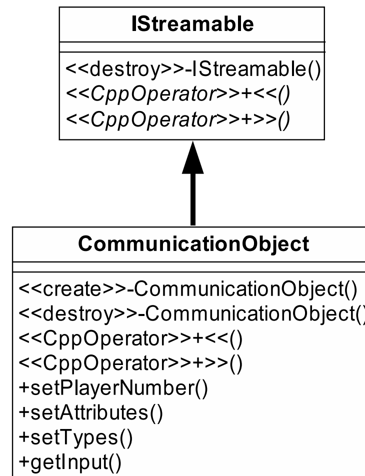

Abbildung 6.31: Streamable zur Kommunikation von Weltmodellen

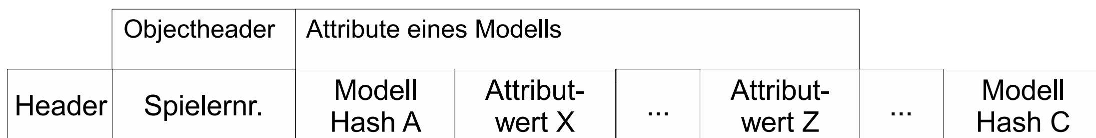

Abbildung 6.32: Datenpaket zur Kommunikation der Modelldaten des globalen Weltmodells

der lokalen Struktur unbekannte Modell bei der Zusammenstellung des Kommunikationsobjekts nicht berücksichtigt, sodass hierfür keine Platzhalterwerte gesendet werden. Dies hat den Vorteil, dass keine Modelle mit reinen Dummy-Attributen empfangen werden müssen und damit die Erstellung eines informationslosen Modells vermieden wird. Die im fünften Fall beschriebene Konstellation erfordert die Versendung eines Defaultwerts für ein unbekanntes Attribut, sodass bei der Erzeugung von CommunicationObject an den Stellen dieser Attribute Dummywerte eingefügt werden müssen.

Um die zusammengestellten Daten wieder auslesen und die daraus resultierenden Teammodelle erstellen zu können, müssen auch in der *receive*-Methode die verschiedenen Konstellationen beachtet werden. Aus der Methoden *getInput* von CommunicationObject werden die empfangenen Rohdaten in Form eines InputStreams von der Kommunikationsverwaltung geladen.

Tritt der erste in Abschnitt 6.2.1 beschriebene Fall ein, können diese Daten vollständig eingelesen und alle von der Kommunikation vorgesehenen Teammodelle erzeugt werden. Im zweiten Fall kann das empfangene Datenpaket ebenfalls komplett eingelesen werden, da jedoch nicht das ganze lokale Weltmodell kommuniziert wird, ist es nicht möglich, alle Teammodell zu erzeugen. Wie im dritten Fall beschrieben, kann es vorkommen, dass ein Modell empfangen wird, welches nicht alle der Kommunikation bekannten Attribute besitzt. Dies führt dazu, dass die Kommunikationsverwaltung für diese Attribute einen Dummywert in das entsprechende Teammodell speichert. Damit

eine Modellierung feststellen kann, welche Werte eines solchen Modells Dummy-Werte sind, wird diese Information zusätzlich in den Teammodellen bereitgestellt. In den Fällen vier und fünf werden sowohl Modelle als auch Attribute empfangen, die der lokalen Weltmodellstruktur unbekannt sind. Beim Einlesen werden diese überschüssigen Daten übersprungen.

In allen beschriebenen Konstellationen werden die Modelle in Form ihrer Attributwerte gesendet, womit die daraus resultierende Datenmenge so gering wie möglich gehalten wird.

6.3 Verhalten

Das Verhalten wurde wie die neue Weltmodellierung in ein eigenständiges Projekt verschoben. Dabei ergibt sich eine logische Unterteilung in zwei Bereiche. Einerseits in die allgemeinen Klassen, die über verschiedene RoboCup-Zyklen unverändert bleiben, und anderseits in die Klassen, die für den jeweiligen RoboCup spezifisch sind. Zu den allgemeinen Klassen gehört die XabslEngine sowie das Framework XabslControl. Die spezifischen Klassen beinhalten die Eingabeschnittstellen, im Folgenden *Symbol*-Klassen genannt (siehe Abschnitt 6.3.2), die Ausgabeschnittstellen, die als *BasicBehaviors* bezeichnet werden (siehe Abschnitt 6.3.1), sowie den Zustandsgraph.

Die Umstellung der Verhaltensverwaltung erwies sich als erforderlich, da vorher lediglich eine große Symbol- und eine BasicBehavior-Klasse bestanden und diese entsprechend umfangreich und unübersichtlich wurden, was vor allem die Lesbarkeit und Fehlersuche erschwerte. Da das neu entwickelte Weltmodell aus einzelnen Modellen besteht, bietet sich die Abbildung dieser Modelle auf einzelne Symbol-Klassen an, sodass jedes Modellattribut durch ein eigenes Symbol repräsentiert wird. Zudem wird dieser Ansatz von XabslControl mit dem Konzept der *BasicBehaviorProvidern* und *Symbol-Providern* unterstützt.

Als oberste Instanz dieser neuen Verhaltenssteuerung wurde die Klasse *BehaviorControl* entwickelt (siehe Abbildung 6.33). Sie beinhaltet als einzige Methode die Funktion *getXabslControl*, die als Rückgabewert ein XabslControl-Objekt liefert. Dieser XabslControl-Instanz werden die SymbolProvider und die BasicBehaviorProvider hinzugefügt. Des Weiteren übernimmt sie die komplette Initialisierung der XabslEngine sowie die Registrierung der Symbole und BasicBehaviors. Außerdem bekommt sie den für die Initialisierung der XabslEngine notwendigen Pfad zur Datei mit dem Intermediatecode des kompilierten Zustandsautomaten übergeben.

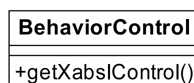

Abbildung 6.33: Verhaltenssteuerung mit XabslControl

Dieser Zustandsgraph ist in der Programmiersprache Xabsl geschrieben und besteht aus verschiedenen Optionen (*Options*), die wiederum aus verschiedenen Zuständen (den *States*) zusammengesetzt sind. Diese Optionen nutzen die Symbols für die Entscheidungsfindung und rufen BasicBehaviors auf, um Bewegungssteuerungsbefehle zu generieren. Die Realisierung dieser Schnittstellen wird in den folgenden Abschnitten näher beschreiben.

6.3.1 Symbole

Um in den Optionen zur Verfügung zu stehen, müssen die Symbole, die wie beschrieben die Eingabeschnittstelle zum Verhalten bilden, in Form eines SymbolProviders implementiert werden und die von dieser Klasse vorgegebenen abstrakten Methoden definieren. Dies wird hier am Beispiel der *BallSymbols* beschrieben (siehe Abbildung 6.34).

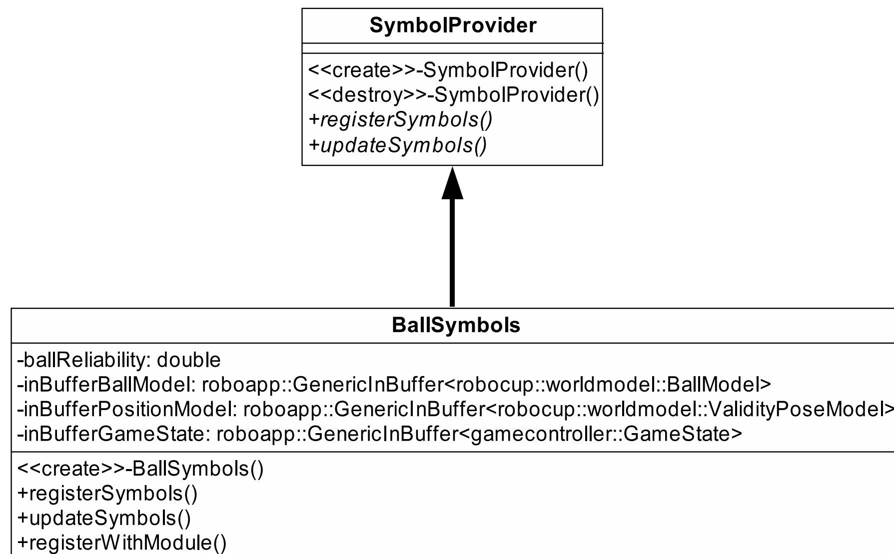

Abbildung 6.34: BallSymbol als Verhaltenssymbol

Von der Basisklasse SymbolProvider und deren Elternklasse werden drei Methoden vorgegeben, und zwar *registerSymbols, updateSymbols* und *registerWithModule*. In der Methode registerWithModule werden die für die Symbolklasse benötigten Modelldaten registriert. Im dargestellten Beispiel sind dies das BallModel, das RobotPoseModel und der GameState. Damit die daraus berechneten Symbole dem Zustandsautomaten zur Verfügung stehen, werden sie in registerSymbols, die bei der Initialisierung von Xabsl-Control aufgerufen wird, auf der XabslEngine registriert. Diese Symbole werden mittels privater Membervariablen in den Symbol-Klassen gehalten, bei dem dargestellten Ball-Symbol ist ein solches Attribut die *ballReliability*. Damit die Symbole kontinuierlich auf den aktuellen Stand gehalten werden können, werden sie bei jedem Aufruf von Xabsl-Control über updateSymbols neu berechnet. Dies geschieht bevor der Zustandsautomat

verarbeitet wird, damit bei der Entscheidungsfindung immer das aktuelle Weltmodell zur Verfügung steht.

6.3.2 BasicBehaviors

Die Ausgabeschnittstelle der Zustandsmaschine bilden die BasicBehaviors. Sie werden in einem Endzustand des Automaten ausgeführt und generieren ein Bewegungskommando, auch *MotionRequest* genannt, welches an die Bewegungssteuerung verschickt wird. Die BasicBehaviors werden über eine zusätzliche Verwaltungsklasse (dem *MotionProvider*) der XabslControl-Instanz hinzugefügt (siehe Abbildung 6.35).

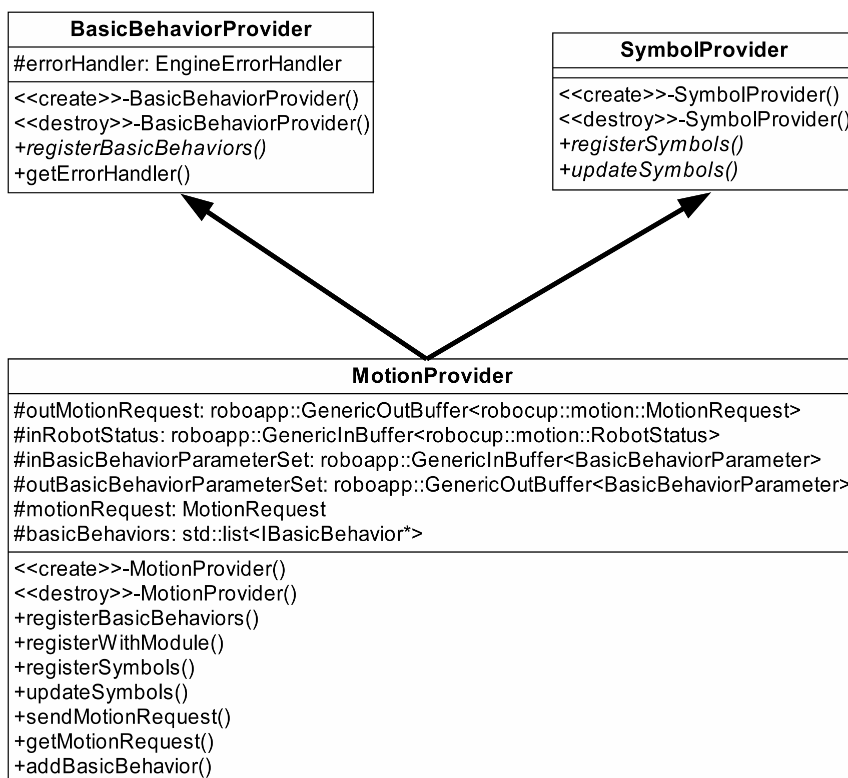

Abbildung 6.35: MotionProvider zur Ansteuerung der Bewegungsbefehle

Der MotionProvider wird genutzt, um Kontrolle über die zu verschickenden Bewegungskommandos zu gewinnen und diese gegebenenfalls anpassen zu können. Eine solche Anpassung ist nötig, wenn die Lagesensoren signalisieren, dass der Roboter umgefallen ist. In diesem Fall werden die Bewegungskommandos der Zustandsmaschine unterbrochen und stattdessen Aufstehbewegungen generiert, was in der Methode *sendMotionRequest* entschieden wird. Der MotionProvider implementiert die Schnittstelle BasicBehaviorProvider und kann dadurch dem XabslControl-Objekt hinzugefügt werden.

Bei der Initialisierung des MotionProvider werden die BasicBehaviors, die von der

einheitlichen Schnittstelle *IBasicBehaviors* erben, über *addBasicBehavior* hinzugefügt und in allgemeiner Form gehalten. Der Aufruf der vom MotionProvider zu implementierenden Methoden *registerBasicBehaviors* und *registerWithModule*, die vom BasicBehaviorProvider vorgegeben werden, kann somit an die IBasicBehaviors (siehe Abbildung 6.36) weitergeleitet werden. Diese werden analog zu den Symbol-Klassen über *registerBasicBehaviors* auf der XabslEngine registriert. Mittels *registerWithModule* können die für die Generierung der Bewegungskommandos benötigten Eingabedaten registriert werden. Des Weiteren bietet der MotionProvider über eine Reihe von Methoden (wie *createWalkRequest*) die Möglichkeit, von ihm vordefinierte Bewegungsanweisungen anzufragen und auszuführen.

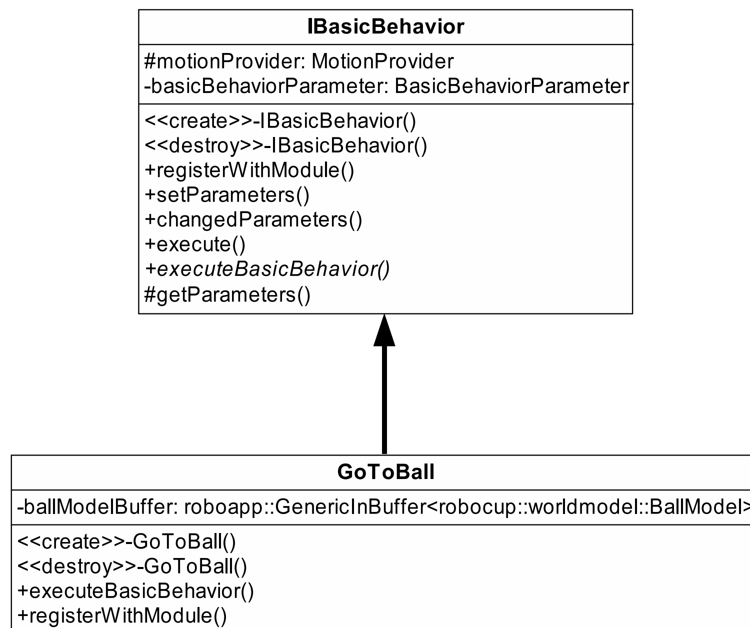

Abbildung 6.36: GoToBall als BasicBehavior

Die BasicBehaviors, hier am Beispiel *GoToBall* beschrieben, müssen die Methode *executeBasicBehavior* definieren, in der die Logik für die Bewegungsgenerierung enthalten ist. Der Aufruf wird von der XabslEngine und dem Interface *IBasicBehavior* über die Methode *execute* gesteuert. Wird von der XabslEngine ein BasicBehavior aufgerufen, erfolgt die Ausführung von executeBasicBehavior, und das dabei berechnete Bewegungskommando wird dem MotionProvider über sendMotionRequest übergeben.

7 Ergebnisse

In den folgenden Abschnitten werden die realisierten Komponenten dieser Arbeit anhand der Qualitätssicherungsmerkmale (siehe Kapitel 4) analysiert. Dabei wird die Korrektheit in einfachen Fällen durch Komponententests gezeigt, wofür das Testprojekt "WorldModelTest" angelegt wurde. Dieses Projekt wurde zusammen mit dem RoboTest von RoboFrame in die Testsolution "RoboCup07Test" integriert, sodass alle Tests der Robotersteuerungssoftware an einer Stelle automatisiert aufgerufen werden können. In komplizierteren Fällen wurde auf einen vollständigen Beweis durch Testfälle bewusst verzichtet, da dies den Rahmen dieser Arbeit überschritten hätte. Hier wurde die Korrektheit durch Anwendungstests mithilfe der dafür vorgesehenen Dialoge sichergestellt.

Bei der Entwicklung der Komponenten wurde darauf geachtet, die Plattformunabhängigkeit von RoboFrame beizubehalten. Dies wurde durch ausschließliche Verwendung von Standardbibliotheken sowie den RoboFrame-spezifischen Bibliotheken erreicht. Die Adaptierbarkeit und Portabilität ist auf RoboFrame beschränkt, da diese Arbeit nur zusammen mit RoboFrame eingesetzt werden kann. Um die Weltmodellierung und die Kommunikation in anderen Robotersteuerungsanwendungen einsetzen zu können, müssten die RoboFrame-spezifischen Verwaltungseinheiten ersetzt werden.

Alle in dieser Arbeit erstellten Klassen und Methoden sind kommentiert und vollständig objektorientiert umgesetzt. Um eine gute Lesbarkeit zu erreichen, wurde darauf geachtet, Berechnungen innerhalb einer Klasse in einzelne Methoden zu gliedern und Klassen bezüglich ihrer Zeilenanzahl in einem übersichtlichen Rahmen zu halten. Durch diesen Programmierstil und den konsequenten Einsatz von Schnittstellen, an denen die Entwicklung weiterer Module (wie Modellierungen mit Modellen) anknüpft, wurde eine gute Erlernbarkeit und Wartbarkeit dieser Komponenten erreicht.

Neben der detaillierten Analyse der Weltmodellierung und Kommunikation anhand der Qualitätssicherungsmerkmale wird in diesem Kapitel die Verhaltenssteuerung lediglich bezüglich der Verwendbarkeit der Schnittstelle zur Weltmodellierung betrachtet.

7.1 Weltmodellierung

In den folgenden Abschnitten wird zunächst auf die entstandene Lösung der Datenintegration eingegangen sowie die Verwendung der Daten in der Weltmodellierung beschrieben. Es folgt eine Analyse der realisierten Modellierungsschnittstelle in Bezug auf die Einbindung und Verwaltung neuer Modellierungen. Nach einer Betrachtung der Effizienz der Weltmodellierung wird im letzten Abschnitt die Stabilität des entwickelten

Frameworks untersucht, wobei die dafür getroffenen Maßnahmen erläutert werden.

7.1.1 Daten

Wie im Konzept der Weltmodellierung beschrieben wurde (siehe Abschnitt 5.1), sollte eine zentrale Datenverwaltung entwickelt werden, die alle für die Weltmodellierungen benötigten Daten verwaltet. Um den Komponenten der Weltmodellierung die Daten einfach zur Verfügung stellen zu können, wurden intuitive Schnittstellen für das Hinzufügen und Abrufen der Perzepte, Modelle und Teammodelle benötigt.

Diese Funktionalität wurde in Form der WorldData-Komponente entwickelt, welche die benötigten Daten mittels der Datenklassen Percept-, Model- und TeamModelData verwaltet. Sie kapseln die In- und OutBuffer des RoboFrame, wodurch jeder Komponente weiterhin eine eigene Sicht auf die enthaltenen Daten ermöglicht wird. Dies hat den Vorteil, dass die bereits vorhandenen und getesteten Strukturen und Kommunikationswege des Frameworks zum Einsatz kommen und problemlos ein Verbindungsaufbau zur GUI möglich ist. Die neue Datenverwaltungskomponente ermöglicht es, der Weltmodellierung Informationen, die beim Abruf von Daten gewonnen werden, bereit zu stellen, die zur Berechnung der Aufrufreihenfolge der Modellierungen genutzt werden. Durch die zentrale Verwaltung der Modelle und Teammodelle in Form ihrer Basisklassen wurde die Entwicklung einer Kommunikation ermöglicht, die alle verfügbaren Modelle abrufen und alle empfangenen Modelle als Teammodelle hinzufügen kann. Diese flexible Behandlung der Daten bringt den Vorteil, dass bei einer Erweiterung der Weltmodellierung die Kommunikation nicht angepasst werden muss.

Der in WorldData enthaltene KeyManager ermöglicht eine gute Übersicht über die in der Weltmodellierung zur Verfügung stehenden Daten, da alle Percepte, Modelle und Teammodelle in dieser Klasse registriert werden müssen. Durch die getrennte Registrierung dieser Daten über die jeweiligen Schnittstellen ist es möglich, einem Datentyp und seinem RoboApp-Schlüssel einen Weltmodellschlüssel zuzuordnen. Dies hat den Vorteil, dass ein Weltobjekt, welches von mehreren Sensoren erkannt wird, über einen Weltmodellschlüssel den Modellierungen zur Verfügung steht, da Perzepte verschiedener Sensoren dem selben Weltmodellschlüssel zugeordnet werden können.

Die hinzugefügten Daten können mittels separater Methoden für Perzepte, Modelle und Teammodelle anhand der zugeordneten Weltmodellschlüssel geladen werden. Ein Perzeptdatenobjekt enthält alle ihm zugeordneten Perzepte, die anhand des Sensorschlüssels unterschieden werden können, sodass Perzepte, die das selbe Weltobjekt repräsentieren, nicht einzeln abgerufen werden müssen. Die Modellierungen können hierdurch die Perzepte mithilfe eines für den Sensor vergebenen Schlüssels bewerten und zu einem Modell fusionieren.

7.1.2 Integration von Modellierungen

Um den wachsenden Anforderungen an die Robotersteuerungsanwendungen gerecht werden zu können, ist eine ständige Erweiterung der Weltmodellierung durch Modellierungen zu berücksichtigen. Dieser Aspekt wurde durch die Konzipierung der Modellierungen als Module realisiert, sodass diese der Weltmodellierung einzeln hinzugefügt werden können. Durch eine einheitliche Basisklasse werden alle Methoden vorgegeben, die für die Weltmodellierung von Bedeutung sind. Da jede Art von Datenaustausch zwischen der Weltmodellierung und ihren Modellierungen über diese Methoden stattfindet, ist ihre Steuerung unabhängig von der speziellen Implementierung. Die dadurch gewonnene Modularität bietet den Vorteil, eine individuelle Weltmodellierung zu gestalten. Außerdem ist durch die einfache Registrierungsschnittstelle ein unkomplizierter Austausch gleichartiger Modellierungen möglich, sodass diese bezüglich ihrer Laufzeit und der Qualität des berechneten Modells verglichen werden können.

Durch das Registrieren der Modellierungen in der Modellierungsverwaltung wurde ermöglicht, die berechneten Modelle nach dem Aufruf der jeweiligen Modellierung automatisch abzurufen und in die zentrale Datenverwaltung zu schreiben. Durch diesen Automatismus wird den Modellierung der für das Bereitstellen der Daten benötigte Verwaltungsaufwand abgenommen und sichergestellt, dass bei jedem Takt ein Modell zur Verfügung steht und dieses in der zentralen Datenverwaltung gespeichert wird. Da die Modellierungen anhand ihres Weltmodellschlüssels in Form ihrer Basisklasse verwaltet werden, ist ein gezielter Zugriff auf die einzelnen Modellierungen möglich, sodass sie individuell aufgerufen werden können. Dies wird genutzt, um die anhand der Eingabedaten berechnete Aufrufreihenfolge zu realisieren.

7.1.3 Effizienz

Das Weltmodellierungs-Framework ist effizient konzipiert, da der größte Teil der Rechenzeit in die Modulinitialisierung fällt. Dort finden alle Modellierungsregistrierungen und -initialisierungen sowie die Bufferregistrierungen auf dem RoboFrame statt. Sobald diese Initialisierungsphase abgeschlossen ist und der Execute-Zyklus der Module beginnt, finden im Weltmodellierungs-Framework keine Berechnungen mehr statt. Ab diesem Zeitpunkt wird die komplette Laufzeit der Weltmodellierung von den Modellierungen bestimmt. Zur dynamischen Optimierung der Laufzeit werden den Modellierungen die Zeitstempel von vor und nach dem letzten Aufruf zur Verfügung gestellt.

7.1.4 Stabilität

Wie in Kapitel 4 beschrieben ist die Weltmodellierung ein zentraler Bestandteil der Robotersteuerungssoftware, sodass großer Wert auf ihre Stabilität gelegt werden muss. Eine gute Stabilität setzt sich zusammen aus Korrektheit, Robustheit und Zuverlässigkeit. Für das in dieser Arbeit entwickelte Framework wurden Robustheit und Zuverläs-

sigkeit differenziert betrachtet, da an einigen Stellen bewusst auf eine größtmögliche Robustheit verzichtet wurde. Unterschieden wurde dabei zwischen Benutzerfehlern, die die Anwendung in ihrer Korrektheit beeinflussen, und Fehlern, die weitestgehend unabhängig vom weiteren Anwendungsverlauf sind.

Im ersten Fall wurde bewusst eine hohe Robustheit vermieden und die Anwendung zum Absturz gebracht. Um diese Fehler gut lokalisieren und beheben zu können, wurden die Debugmöglichkeiten des RoboFrame genutzt und an den entsprechenden Stellen Logeinträge erstellt. Ein Beispiel für diesen Fall ist ein Registrierungsfehler im KeyManager. Wurde dort ein von einer Modellierung angefragtes Perzept nicht registriert, gibt der KeyManager eine Fehlermeldung aus und die Anwendung stürzt ab. Dieser Fehler wird bewusst nicht abgefangen, da die weiteren Berechnungen mit einem nicht vorhandenen Perzept inkorrekt und der daraus resultierende Fehler schwerer zu lokalisieren wäre.

Im zweiten Fall wird die Robustheit aufrechterhalten, da die Anwendung weitestgehend fehlerfrei weiterlaufen kann. Um den Benutzer auf eine solche Unregelmäßigkeit aufmerksam zu machen, wird eine Warnmeldung ausgegeben. Ein Beispiel ergibt sich beim Berechnen der Aufrufreihenfolge der Modellierungen. Benötigt jeweils eine Modellierung das Ergebnis der anderen zur Berechnung des eigenen Modells, lässt sich keine eindeutige Lösung der Aufrufreihenfolge bestimmen. Dies ist nicht kritisch, sodass lediglich eine Warnung in die Log-Datei geschrieben wird und die Anwendung normal weiterläuft. Die gemäß der berechneten Reihenfolge zuerst aufgerufene Modellierung rechnet in diesem Fall jeweils mit dem Modell des vorangegangenen Taktes.

Um die Korrektheit der Weltmodellierung sicher zu stellen, wurden Komponenten- und Anwendungstests durchgeführt. Die für das Testen einzelner Methoden geschriebenen Komponententests lassen sich automatisiert in der angelegten Testsolution starten. Ein Beispiel hierfür ist die Aktualisierung der Wahrscheinlichkeit eines Modells in der Komponente ModelingOperations. Der hierfür angelegte Testfall überprüft kontinuierlich, ob der übergebene Wahrscheinlichkeitswert nach drei Sekunden von Eins auf Null gesunken ist und diesen Wert beibehält. Die komplexeren und komponentenübergreifenden Funktionalitäten wurden in Form von Anwendungstests mithilfe der GUI sowie der dafür erstellten Debugausgaben überprüft. Um beispielsweise die Korrektheit der Modellierungsaufrufreihenfolge sicher stellen zu können, wurden die bestehenden Debugmechanismen genutzt und die Berechnung mittels verschiedener Weltmodellierungskonstellationen getestet. Der erweiterte Dialog ModelViewer bietet den Entwicklern von Modellierungen die Möglichkeit, die berechneten Modelle sowie die dafür genutzten Eingabedaten anzeigen zu lassen. Durch diese Mechanismen wird eine gute Testbarkeit des Weltmodellierungs-Frameworks sowie der darin enthaltenen Modellierungen erreicht.

7.2 Kommunikation

Wie bereits in Abschnitt 7.1.1 beschrieben, kann die Kommunikation alle in der Weltmodellierung vorhanden Modelle aus der zentralen Datenverwaltung in Form der Basisklasse abrufen, wodurch eine allgemeine Datenhaltung ermöglicht wird. Aufgrund der in diesen Modellen enthaltenen Informationen über die zu kommunizierenden Attribute ist es der Kommunikation möglich, eine lokale Weltmodellstruktur zu erstellen und an andere Roboter zu versenden. Die im globalen Weltmodell für den Austausch festgelegten Attribute werden von der Kommunikation über Methoden der Basisklasse beim Senden aus dem jeweiligen Modell geladen und beim Empfang in ein neu erstelltes Teammodell gespeichert. Aufgrund der entwickelten Schnittstellen der Weltmodellierung ist dieser Mechanismus unabhängig von der konkreten Implementierung von Modellen, sodass die Kommunikation bei Änderungen der Modellierungskonstellation nicht angepasst werden muss.

Wie in Abschnitt 6.2.2 beschrieben, berechnet die Datenverwaltungskomponente der Kommunikation alle speziell für den Datenaustausch benötigten Informationen (wie beispielsweise das globale Weltmodell). Um die Funktionsfähigkeit dieser Berechnungen sicherzustellen, wurden die entsprechenden Methoden mittels Komponententests überprüft und können bei einer Änderung der Kommunikation automatisiert aufgerufen werden.

Im Folgenden werden die Ergebnisse der Kommunikation in Hinblick auf die Heterogenität dargelegt. Wie die Weltmodellierung wird auch die Kommunikation bezüglich Effizienz und Stabilität analysiert.

7.2.1 Heterogenität

Wie im Abschnitt 6.2 beschrieben, wurden alle für die Entwicklung einer heterogenen Kommunikation relevanten Fälle betrachtet, sodass die entstandene Kommunikation den Austausch beliebiger Modelle zwischen verschiedenen Weltmodellierungen ermöglicht. In den folgenden Abschnitten wird diese Heterogenität anhand der Kommunikation der Modelle als Summe der Attributwerte sowie anhand der Kommunikation der Weltmodelle als Summe der Modelle dargelegt.

7.2.1.1 Kommunikation heterogener Modelle

Ein Modell im globalen Weltmodell kann einerseits einen Teil der eigenen Attributwerte und andererseits unbekannte Attributwerte enthalten. Die Korrektheit solcher Konstellationen wurden mithilfe von Anwendungstests überprüft. Dafür wurden Applikationen mit Weltmodellen erstellt, deren Modelle sich in der Attributanzahl unterschieden, sodass einem Kommunikationsteilnehmer Attribute bekannt sind, die von anderen nicht verstanden werden. Trotzdem ist es diesem Mitspieler möglich, diese Teilmodelle zu empfangen und in der Weltmodellierung zu nutzen.

Das folgende Beispiel visualisiert die Kommunikation eines solchen Teilmodells mithilfe der GUI. Roboter 2 und 3 senden sich gegenseitig ihre Position und ihr Ballmodell (siehe Abbildung 7.1). Die Attributanzahl von Roboter 3 ist geringer, da die relative x-Position (relativeX) in seinem Ballmodell nicht enthalten ist.

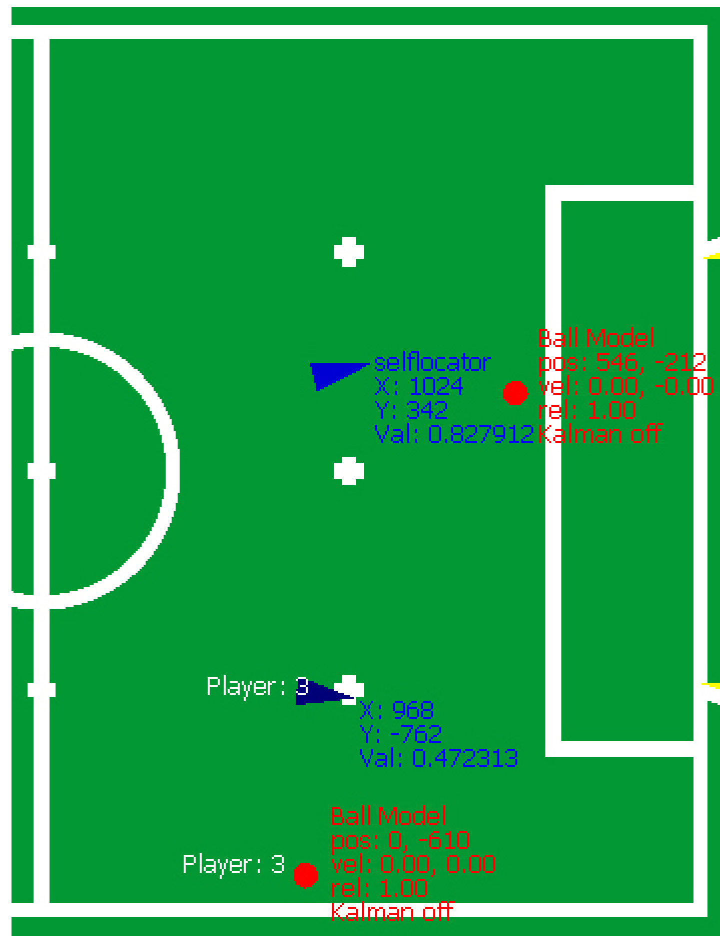

Abbildung 7.1: Visualisierung einer Kommunikation von heterogenen Modellen auf einem virtuellen Spielfeld

Dabei ist bei dem lokalen Ballmodell zu sehen, dass der Ball mit seiner relativen x- und y-Position dargestellt wird. Er hat die Koordinaten (x=546; y=-212) bezüglich der Roboterposition. Das gesendete Ballmodell von Roboter 3 wird nur auf der y-Achse der Roboterposition gezeichnet, da dieses Modell keine Informationen über einen relativen x-Wert besitzt. Beim Empfang dieses Teilmodells wird ein Teammodell erstellt, bei dem der fehlende Wert im initialen Zustand gelassen wird. In der Abbildung werden die Daten von Roboter 3 als Teammodelle der lokalen Weltmodellierung dargestellt. Die x-Position des Teamballmodells hat den initialen Wert 0, da sie wie beschrieben bei der Kommunikation des Modells, nicht enthalten ist. Die Abfrage, ob bestimmte Werte in den Teammodellen nicht empfangen wurden, ist mittels Komponententests überprüft und somit ihre Korrektheit sichergestellt worden. Dieser Test zeigt, dass auch Teile von Modellen gesendet, von dem empfangenden Roboter eingelesen und in der Weltmodel-

lierung genutzt werden können.

Aufgrund der Aufsplittung der Modelle in Attribute ist es möglich, Werte zu empfangen, die dem eigenen Modell unbekannt sind und deswegen von der Kommunikation verworfen werden. Die Korrektheit dieser Funktionalität wurde analog zu dem eben beschriebenen Beispiel getestet.

Außerdem kann es dazu kommen, dass in dem globalen Weltmodell Attribute enthalten sind, die dem lokalen Weltmodell unbekannt sind. In diesem Fall werden von der Kommunikation Platzhalter versendet, deren Abfrage durch Tests auf Korrektheit überprüft wurde. Bei diesen Testfällen wurden entsprechende Weltmodellstrukturen angelegt und bei der Berechnung des globalen Schnittmodells geprüft, ob die Platzhalter für jeden Roboter korrekt ermittelt werden.

7.2.1.2 Kommunikation heterogener Weltmodelle

Das globale Weltmodell kann aus einer Teilmenge der eigenen Modelle sowie aus Modellen bestehen, die dem eigenen Weltmodell unbekannt sind. Die Funktionsfähigkeit der Kommunikation solcher heterogenen Weltmodelle wurde ebenfalls durch verschiedene Anwendungstests sichergestellt.

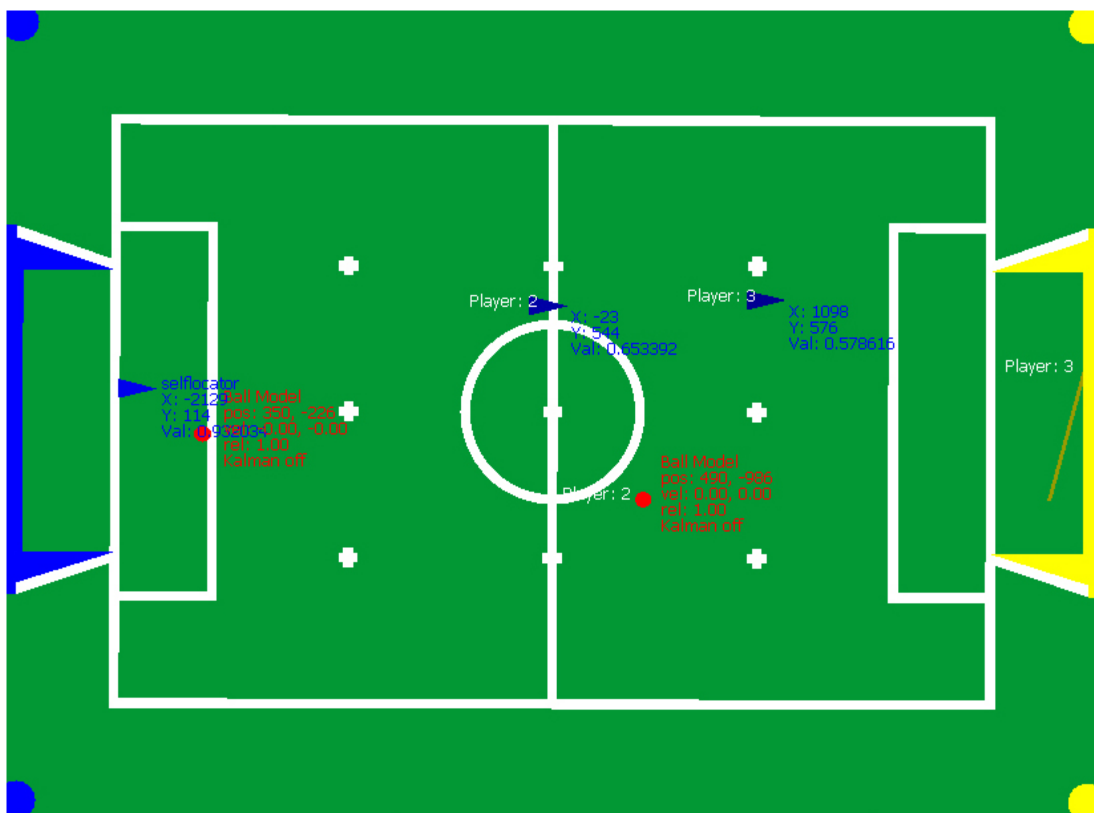

Abbildung 7.2: Visualisierung einer Kommunikation von heterogenen Weltmodellen auf einem virtuellen Spielfeld

Um zu zeigen, dass die Kommunikation auch dann noch stabil funktioniert, wenn Modelle gesendet werden, die nur einzelnen Robotern bekannt sind, wurden Tests durchgeführt, bei denen mindestens drei verschiedene Weltmodelle ausgetauscht werden. Diese Tests belegen, dass die Kommunikation unbekannte Modelle nicht versenden muss und beim Empfangen ignorieren kann.

Abbildung 7.2 illustriert eine dieser Konstellationen, bei der Roboter 1 die Modelle Ball, Tor und eigene Position, Roboter 2 nur den Ball und die eigene Position und Roboter 3 nur das Tor und die eigene Position versendet. Auf dem Spielfeld ist das Weltmodell von Roboter 1 zu sehen, das die empfangenen Modelle von Spieler 2 und 3 enthält. Die Ansicht dieses Spielers wurde gewählt, da er als einziger sämtliche Modelle seiner Mitspieler verarbeiten kann.

Abbildung 7.3: Ausschnitt der lokalen Weltmodelldaten

Zur besseren Übersicht werden im Folgenden die zum jeweiligen Roboter gehörenden Informationen dargestellt und ihre Korrektheit erläutert. In Abbildung 7.3 sind die lokalen Daten von Roboter 1 zu sehen, die den Ball sowie die eigene Position umfassen. Das Tor ist nicht sichtbar, da es außerhalb des Spielfeldes erkannt und gezeichnet wurde.

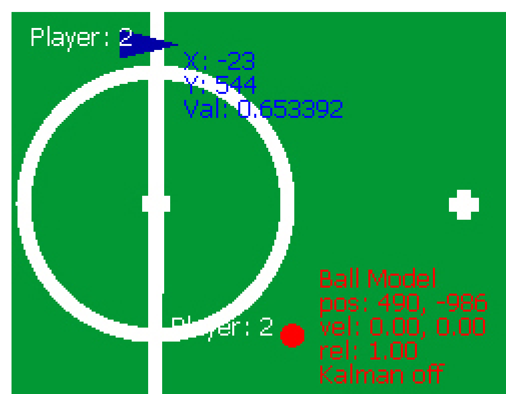

Abbildung 7.4: Teammodelle von Roboter 2

Roboter 1 empfängt von Roboter 2 das Ball- und Positionsmodell (siehe Abbildung 7.4). Wie in der Darstellung zu sehen ist, wird die Position des Roboters auf dem Spielfeld korrekt gezeichnet, was anhand der x und y Position zu erkennen ist. Des Weiteren

wird das Ballmodell relativ zur empfangenen Roboterposition dargestellt, wodurch sowohl die Funktionsfähigkeit der Visualisierung dieser Modelle als auch der korrekte Empfang bestätigt wird.

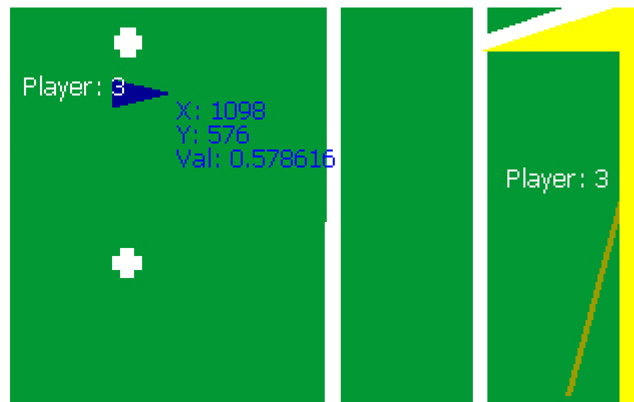

Abbildung 7.5: Teammodelle von Roboter 3

Die Modelle von Roboter 3 werden analog zu Roboter 2 als Teammodelle auf dem Spielfeld dargestellt (siehe Abbildung 7.5). Das Weltmodell dieses Mitspielers enthält wie zu sehen neben der eigenen Position kein Ballmodell, sondern ein Tormodell. Anhand dieser Darstellung wird gezeigt, dass Roboter 1 auch die Daten von Roboter 3 korrekt verarbeiten und anzeigen kann.

Der Test belegt, dass Roboter 1 und 2 neben der Roboterposition auch die Ballposition und Roboter 1 und 3 die Torposition austauschen. Da die Modelle jeweils über UDP-Broadcast versendet werden und damit alle Roboter immer sämtliche Modelle empfangen, müssen Roboter 2 und 3 Daten ignorieren, die ihnen unbekannt sind. Hierdurch wird die Funktionsfähigkeit der Kommunikation von heterogenen Weltmodellen gezeigt.

7.2.2 Effizienz

Bei der Konzeption der Kommunikation wurde auf einen effizienten Datenaustausch Wert gelegt, der sich in der Datenpaketgröße und in der Austauschfrequenz widerspiegelt. Um die Datenpaketgröße möglichst gering zu halten, wurden die Rohdaten von den Informationen getrennt, die das Weltmodell definieren. Anhand der Struktur ist es jedem Roboter möglich, das globale Weltmodell zu berechnen, sodass nur die Rohdaten verschickt werden, die mindestens zwei Robotern bekannt sind. Die Paketgröße wird hierdurch auf ein Minimum reduziert. Da die Struktur des Weltmodells lediglich zur Berechnung des globalen Schnittmodells benötigt wird, ist es nicht nötig, sie in gleicher Häufigkeit wie die Rohdaten auszutauschen. Sie wird daher in einem wesentlich größeren Intervall versendet, was ebenfalls zu einer wesentlich geringeren Netzauslastung führt.

Im folgenden Beispiel (siehe Abbildung 7.6) sind die Netzauslastung sowie die Größe

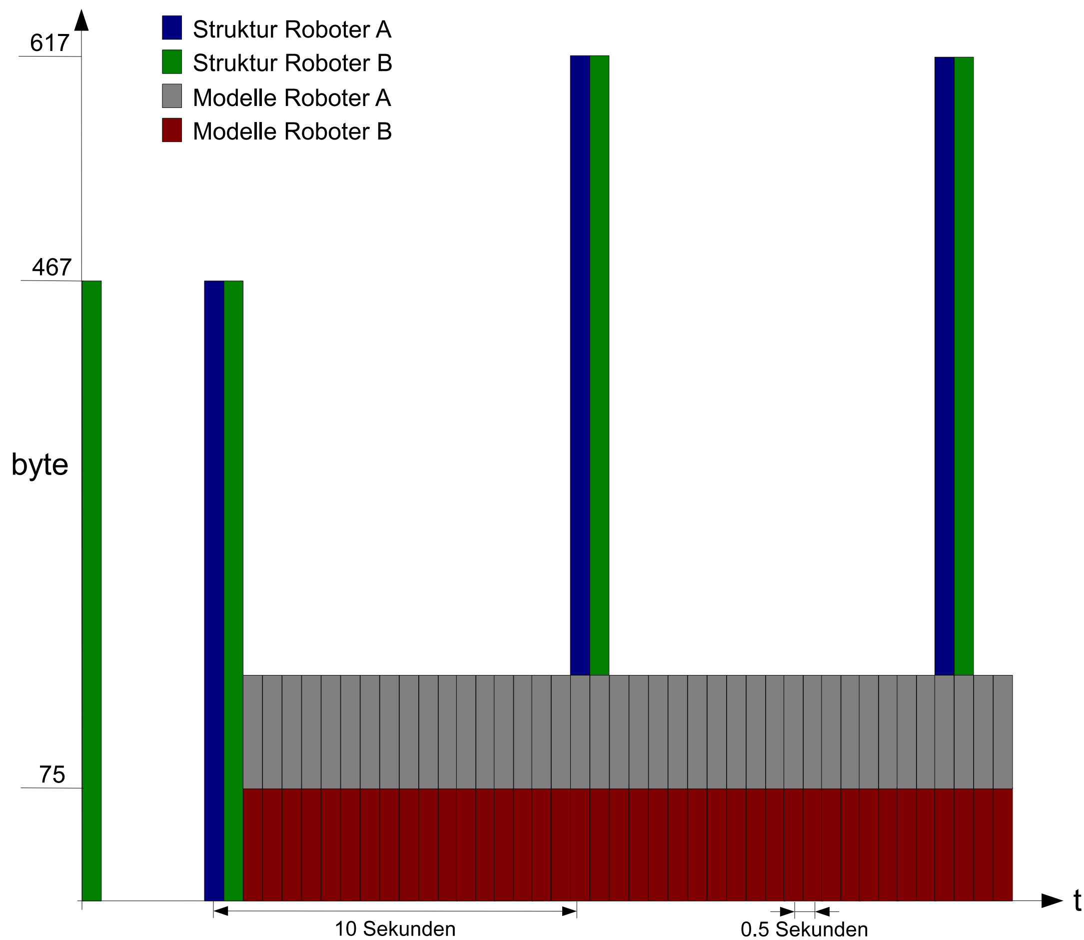

Abbildung 7.6: Beispielhafte Darstellung einer Netzwerkkommunikation der aktuellen RoboCup Anwendung sowie der daraus resultierenden Datenpaketgrößen

der Datenpakete der Kommunikation in der aktuellen RoboCup-Anwendung zu sehen. Es ist zu erkennen, dass zuerst Roboter B gestartet wird und seine lokale Weltmodellstruktur versendet. Sobald Roboter A gestartet wird, sendet dieser ebenfalls seine Weltmodellstruktur, welche von Roboter B empfangen wird, der mit seiner Struktur antwortet und das globale Weltmodell berechnet. Nachdem auch Roboter A das Schnittmodell gebildet hat, beginnt der Datenaustausch, der zyklisch jede halbe Sekunde durchgeführt wird. Damit bei einer Verbindungsunterbrechung oder Vergrößerung der Kommunikationsteilnehmeranzahl die Kommunikation wieder aufgenommen oder erweitert werden kann, wird die lokale Weltmodellstruktur alle zehn Sekunden von jedem Roboter gesendet. Wie in der Abbildung zu sehen ist, wird die Struktur mit einer Datenpaketgröße von 467 Byte im Vergleich zu den Rohdaten mit einer Paketgröße von 75 Byte wesentlich seltener gesendet.

7.2.3 Stabilität

Durch die Implementierung einer heterogenen Kommunikation wird eine hohe Robustheit und Zuverlässigkeit erlangt. Da Teilmodelle sowie unbekannte Modelle entweder eingelesen oder komplett ignoriert werden können, kommt es in diesen Fällen nicht zum Absturz der Anwendung. Auch bei der Kommunikation unterschiedlicher Versionen von bekannten Modelle wird die Anwendung nicht beeinträchtigt, da die Modelle serialisiert verschickt und von ihrem ursprünglichem Objekt losgelöst werden.

Wie bereits in den vorherigen Abschnitten beschrieben, wurde die Korrektheit sowohl mit Komponententests als auch mithilfe von Anwendungstests sichergestellt beziehungsweise überprüft. Die Komponententests können automatisiert innerhalb des angelegten Testprojekt angestoßen werden. Für Anwendungstests wurde der in dieser Arbeit erweiterte ModelViewer genutzt, der auch bei der Weiterentwicklung der Modellierungen zur Anzeige von Teammodellen verwendet werden kann.

7.3 Verhalten

Bei der Entwicklung des Verhaltens muss hervorgehoben werden, dass durch die Einbindung von XabslControl eine modulare Schnittstelle entstanden ist, da sich SymbolProvider und BasicBehaviorProvider ähnlich einem Baukastensystem einfach hinzufügen und entfernen lassen. Dadurch war es möglich, die Modelle auf eigene Symbolklassen abzubilden, was zu einer intuitive Verwendung der Modellinformationen im Verhalten führt.

Aufbauend auf diesen Informationen wurde das Verhalten für die Wettbewerbe von 2007 entwickelt. Die Symbole können über den zur GUI gehörenden Xabsl-Dialog mit den im ModelViewer angezeigten Modelldaten abgeglichen werden, wodurch sich die Korrektheit der Schnittstelle sowie das gesamte Verhalten testen lässt.

7.4 RoboCup Wettbewerbe 2007

Für die Teilnahme an den RoboCup Meisterschaften 2007 wurde eine neue RoboCup-Solution angelegt, in welche neben den bestehenden Projekten das neue Weltmodell, die Kommunikation und das Verhalten eingebunden wurden. Die bereits existierenden Modellierungen wurden in die neue Weltmodellierung integriert und weiterentwickelt. Außerdem wurde die Weltmodellierung durch die Odometrie- und Verhaltensmodellierung erweitert.

Die Odometriemodellierung ermöglicht eine effiziente Aktualisierung der Modelldaten, da die Odometriedaten für alle Modellierungen vorberechnet sind und über das Odometriemodell zur Verfügung stehen. Der von der Verhaltensmodellierung bestimmte Rollenwechsel war in den Spielen gut zu erkennen, da immer nur ein Mitspieler zum

Ball gegangen ist, während der andere auf Absicherung des Tores bedacht war. Der Torwart gab nur bei einem Ausfall des Angreifer seine verteidigende Rolle auf, wodurch entscheidende Tore erzielt wurden.

Das Verhalten dieser RoboCup-Anwendung wurde speziell für die Wettbewerbe 2007 neu entwickelt, um ein besseres Teamspiel möglich zu machen. Des Weiteren wurde in das Verhalten eine KickTable integriert, die passende Schüsse für übergebene Weiten und Winkel liefert. Dies gewährleistet den flexiblen Einsatz einer großen Anzahl unterschiedlicher Schüsse, die ein ansehnliches Fußballspiel ermöglichen.

Aufgrund dieser Entwicklungen konnte die Leistung der Roboterfußballmannschaft im Vergleich zu den Wettbewerben im vorangegangenen Jahr gesteigert werden.

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

Die im Rahmen dieser Diplomarbeit entwickelte Weltmodellierung dient zur einfachen Erstellung eines Weltmodells. Dabei wurde darauf Wert gelegt, dass einzelne Modellierungen genau ein Modell berechnen und dieses leicht in das Weltmodell integrierbar ist. Außerdem wurde eine zentrale Datenverwaltung realisiert, bei der die Datenbereitstellung über klar definierte und leicht verständliche Schnittstellen erfolgt. Diese zentrale Datenhaltung ermöglicht eine einfache Kartografierung der Umwelt durch den Roboter, da sich die in der Welt zentral vorhandenen Informationen durch eine Modellierung in wenigen Schritten zu einer Karte fusionieren lassen. Aufgrund der ebenfalls zentral vorliegenden Teaminformationen ist dies ebenfalls für heterogene Roboterteams möglich.

Durch eine übersichtliche Strukturierung des Weltmodells ist leicht ersichtlich, welche Modellierungen existieren und welche Modelle daraus resultieren. Dies erleichtert das parallele Arbeiten an verschiedenen Modellierungen erheblich und bietet einfache Kontrollmöglichkeiten, sodass keine Perzepte, sondern nur durch Modellierungen gefilterte Modelle in der Verhaltenssteuerung genutzt werden. Aufgrund des konsequenten Einsatzes von Schnittstellen für die einzelnen Komponenten ist eine Erweiterung der Weltmodellierung durch neue Modellierungen vereinfacht worden, da die für die Weltmodellierung benötigten Funktionalitäten von den Basisklassen vorgegeben sind.

Die realisierte heterogene Kommunikation ist an die neue Weltmodellierung gekoppelt und dient dem Empfang und der Versendung des Weltmodells. Darüber hinaus kann jede Modellierung selbst entscheiden, welche Daten kommuniziert werden sollen, sodass die Kommunikation bei einer Änderung der zu kommunizierenden Daten nicht umgestellt werden muss.

Des Weiteren wurde der ModelViewer an die neue Weltmodellierung angepasst und um die neue Struktur der Teammodelle erweitert, wodurch eine gute Testbarkeit gewährleistet werden kann. Dieser Dialog bietet eine gute Übersicht über den aktuellen Zustand des Roboters und die in der Weltmodellierung enthaltenen Informationen, sodass sich die vom Verhalten durchgeführten Aktionen nachvollziehen lassen.

Darüber hinaus wurde die Schnittstelle von der Weltmodellierung zum Verhalten neu gestaltet, sodass die Integration neuer Modelle in die Entscheidungsfindung des Verhaltens leichter fällt. Im Zuge dieser Umstellungen wurden auch die Basisverhalten modular strukturiert, wodurch das Debuggen stark vereinfacht wird.

8.2 Mögliche Erweiterungen

Durch den zeitlich beschränkten Rahmen dieser Diplomarbeit war es nicht möglich, alle denkbaren Aspekte umzusetzen. Daraus ergeben sich einige Möglichkeiten, die Weltmodellierung und Kommunikation bezüglich ihrer Funktionalität zu erweitern. In den folgenden Abschnitten werden wir auf die uns wichtig erscheinenden Erweiterungen eingehen.

8.2.1 Anpassungen für das Spiel 3 gegen 3

Da in den RoboCup Wettbewerben 2008 jedes Team um einen Spieler von zwei auf drei erweitert wird, muss in erster Linie die Verhaltensmodellierung angepasst werden. In ihr wird die Taktik abhängig von der Spielsituation abgebildet. Derzeit ist sie nur auf zwei Spieler ausgelegt (siehe Abschnitt 6.1.4.1.1). Dabei wurden zwei verschiedene Spielerkonstellationen betrachtet, wobei immer mit einem Stürmer und wahlweise mit einem Abwehrspieler oder einem Torwart gespielt wurde, die situationsabhängig ihre Rollen wechseln.

Da das Team auf drei Spieler erweitert wird, ergeben sich bezüglich der Taktik und der Rollenwechselstrategie mehr Möglichkeiten. Beispielsweise kann mit einem Torwart, einem Abwehrspieler und einem Stürmer gespielt werden, wobei auch vorstellbar ist, abwechselnd mit zwei Abwehrspielern und einem Stürmer oder umgekehrt mit einem defensiveren und zwei offensiven Spielern aufzulaufen. Dieser Wechsel zwischen defensiver und offensiver Spielweise könnte auch abhängig von der Ballposition oder Spielstand dynamisch gewechselt werden. Auch im Spiel 3 gegen 3 sollte berücksichtigt werden, dass ein oder sogar zwei Spieler ausfallen könnten und die Taktik dementsprechend anzupassen ist. Wenn in diesem Fall der Stürmer wegfällt, müsste ein verbliebener Spieler dessen Part übernehmen, damit sich immer mindestens ein Akteur am Ball befindet.

Neben der Verhaltensmodellierung sollten auch alle anderen Modellierungen auf mögliche Anpassungen bezüglich der Spieleraufstockung überprüft werden, da dort ebenfalls die Möglichkeit besteht, dass von einer Teamgröße von zwei Mitspielern ausgegangen wurde. Die Kernkomponenten dieser Arbeit, das Framework der Weltmodellierung sowie die Kommunikation, sind bezüglich der Mitspielerzahl flexibel implementiert und müssen somit nicht angepasst werden.

8.2.2 Anpassung der Kommunikation auf allgemeine Datentypen

Da über die Kommunikationsschnittstelle derzeit ausschließlich primitive Datentypen ausgetauscht werden können, ist es sinnvoll, diese auf allgemeine, zusammengesetzte Typen zu erweitern. Um eine effiziente Lösung für diese Erweiterung zu realisieren,

ist eine Anpassung der Stream-Klassen von RoboFrame nötig. Da diese Anpassung im Rahmen dieser Diplomarbeit nicht möglich war, wurde auf das Versenden komplexer Datentypen verzichtet.

Die aktuellen Implementierungen dieser Stream-Klassen bieten keinerlei Informationen über Größe und Typ der von ihnen gekapselten Daten, sodass die Anzahl und Reihenfolge der einzelnen Attribute bekannt sein müsste, damit das Ein- und Auslesen von Objekten möglich ist. Da die in dieser Arbeit entwickelte heterogene Kommunikation jedoch auch vorsieht, Teile eines Objekts austauschen zu können, konnten die Stream-Klassen in der Realisierung nicht verwendet werden.

Nach Anpassung dieser Klassen könnte eine neue Kommunikationsverwaltung konzipiert werden, die ebenso wie die aktuelle Implementierung von der Schnittstelle Communication abzuleiten ist und mithilfe der neuen Streams den Austausch von zusammengesetzten und komplexen Datentypen ermöglicht. Die Datenverwaltung sowie die Kommunikation der lokalen Weltmodellstruktur sind von einer solchen Umstellung nicht betroffen und können von der neuen Kommunikation wiederverwendet werden.

8.2.3 Filterung der kommunizierten Modelle im ModelViewer

Das derzeitige Mannschaftsspiel der Humanoid League des RoboCup sieht ein Spiel mit 2 Spielern je Mannschaft vor. Demnach hat jeder Roboter lediglich einen Mitspieler, von dem er Daten empfangen kann. Der derzeit für Debugzwecke implementierte ModelViewer bietet keine Selektionsmöglichkeit zur Filterung der empfangenen Daten anhand der Mitspielernummern, sodass diese Ansicht bei Roboterteams mit vielen Mitspielern unübersichtlich werden kann.

Die GUI könnte die Information der Roboternummer aus den Teammodellen nutzen, um eine Auswahl zur Anzeige der Modelle einzelner Mitspieler zu ermöglichen. Bei der Implementierung ist darauf zu achten, dass die Stabilität der Kommunikation mit anderen Robotern schwanken kann und die Anzeige bezüglich der verbundenen Roboter dynamisch aktualisiert werden muss. Des Weiteren sollten nicht mehr verbundene Roboter automatisch aus dem ModelViewer entfernt werden.

8.2.4 Verwendung der Modelle in der Selbstlokalisierung

Die zur Berechnung der Roboterposition verwendete Selbstlokalisierung wurde in die Weltmodellierung integriert. Derzeit werden in dieser Modellierung ausschließlich ungeglättete Perzepte benutzt. Diese Daten werden in der Selbstlokalisierung gefiltert und gehen in die weitere Berechnung der Position ein. Die gesonderte Behandlung der Perzepte könnte in die dafür vorgesehenen Modellierungen ausgelagert und die daraus resultierenden Modelle anschließend von der Selbstlokalisierung genutzt werden. Dies würde eine Mehrfachberechnung der Modelle dieser Perzepte in einzelnen Modellierungen vermeiden und zu einer besseren Wartbarkeit und Testbarkeit führen.

8.2.5 Erweiterung der Funktionsbibliothek der Weltmodellierung

Bei der Berechnung der Modelle sollte darauf geachtet werden, dass Methoden, die von anderen Modellierungen ebenfalls zu verwenden sind, in die Weltmodellierungsbibliothek ModelingOperations ausgelagert werden. Durch diese Zentralisierung wird die Mehrfachimplementierung von solchen Methoden vermieden und die Entwicklung von automatischen Testfällen begünstigt, wodurch die Korrektheit dieser statischen Klasse sichergestellt werden kann. Es ist darauf zu achten, dass die darin ausgelagerten Methoden nicht taktabhängig, sondern wie bei der bereits implementierten Funktion zur Aktualisierung der Wahrscheinlichkeit eines Modells abhängig von der Ausführungszeit entwickelt werden, damit die Funktionsbibliothek auch auf Robotersystemen mit anderen Taktraten zum Einsatz kommen kann.

8.2.6 Transcoder für kommunizierte Modelle

Die Kommunikation ermöglicht einen Datenaustausch mit Robotern, deren Weltmodell sich bezüglich der darin enthaltenen Modelldaten unterscheiden kann. Da die Modellierung eines Weltobjektes hinsichtlich der Implementierung variieren kann, ist es möglich, dass heterogene Modelle entstehen, die sich im Konzept oder den gewählten Datentypen unterscheiden.

Ein Beispiel hierfür ist ein Roboter-Lokalisationsmodell, welches die Positionen der Objekte in der Welt mittels unterschiedlicher Datentypen repräsentiert. Die derzeitige Kommunikation würde diese bezüglich der Datentypen unterschiedlichen Attribute nicht in das globale Schnittmodell integrieren und somit nicht kommunizieren. Um diese Informationen nicht zu verlieren, könnten verschiedene Transcoder implementiert werden, die in solchen Fällen miteinander verwandte Datentypen aufeinander abbilden.

Ein weiterer Anwendungsfall eines Transcoders ist eine unterschiedliche Repräsentation kompletter Modelle in der Welt eines Roboters. Um beispielsweise ein metrisches mit einem topologischen Modell austauschen zu können, müsste ein Transcoder entwickelt werden, der das eigene Modell auf die Darstellung des verbundenen Roboters umwandeln kann. Hierbei sind zwei Szenarien denkbar. Einerseits könnte ein Roboter sein eigenes Modell vor dem Versenden auf die Repräsentation des verbundenen Roboters umwandeln. Damit würde sichergestellt, dass die kommunizierten Daten auch empfangen und genutzt werden können. Andererseits könnte eine Umwandlung eines kommunizierten Modells auf die interne Repräsentation erst beim empfangenden Roboter stattfinden.

Die Integration solcher Softwarekomponenten sollte modular und flexibel gestaltet werden, damit eine Erweiterung der verstandenen Modelle einfach möglich ist und die damit verbundene Modellbibliothek wachsen kann.

A Verwendung der Weltmodellierung

A.1 Definition neuer Welt- und Sensorschlüssel

Damit die Weltdaten von den verschiedenen Komponenten der Weltmodellierung angefordert werden können, muss in der Klasse *WorldModel07Keys.cpp* für ein neues Objekt ein Weltmodellschlüssel angeleget werden (siehe Abschnitt 6.1.3). Unter diesen Schlüsseln können *InPerceptData-, InModelData* und *InTeamModelData-Objekte* sowie *OutModelData* und *OutTeamModelData-Objekte* bei *WorldData* angefragt werden (beschrieben im Abschnitt 6.1.3). Im Codebeispiel (Listing A.1) sind die Weltmodellschlüssel Ball, Goal und Circle angelegt.

```
Beispiel WorldModel07Keys.cpp;

const robocup::worldmodel::WorldKey
  WorldModel07Keys::BALL = 1;
const robocup::worldmodel::WorldKey
  WorldModel07Keys::GOAL = 2;
const robocup::worldmodel::WorldKey
  WorldModel07Keys::CIRCLE = 3;

[...]

const robocup::worldmodel::SensorKey
  WorldModel07Keys::HEAD_CAMERA = 1;
const robocup::worldmodel::SensorKey
  WorldModel07Keys::UPPER_BODY_CAMERA = 2;
const robocup::worldmodel::SensorKey
  WorldModel07Keys::OTHER = 3;
```

Listing A.1: WorldModel07Keys.cpp Schlüsseldefinition

Neben den Weltmodellschlüsseln sind in dieser Klasse die Sensorschlüssel zu definieren. Diese werden benötigt, da WorldData alle zu einem Weltmodellschlüssel registrierten Perzepte bereitstellt. Damit diese Perzepte von der anfragenden Komponente unterschieden werden können, wird für jeden Sensor, der Perzepte bereitstellt, ein Sensorschlüssel festgelegt. Zu sehen ist dies am Beispiel der Kopfkamera, der Bauchkamera und einem allgemeinen Sensorschlüssel.

A.2 Registrierung neuer Perzepte

Damit ein neues Perzept in den verschiedenen Komponenten (wie Modellierungen oder der Kommunikation) benutzt werden kann, muss es neben den Registrierungen in RoboFrame auch in der Klasse *KeyManager07.cpp* eingetragen werden (siehe Listing A.2).

```
Beispiel KeyManager07.cpp;

KeyManager07::KeyManager07() {
  registerPerceptKeys();
}
KeyManager07::~KeyManager07() {
  cleanUpFactories();
}
[...]
void KeyManager07::registerPerceptKeys() {
  registerPerceptKey<BallPercept>(WorldModel07Keys::BALL,
   BALL_PERCEPT, WorldModel07Keys::HEAD_CAMERA);
  registerPerceptKey<BallPercept>(WorldModel07Keys::BALL,
   BALL_PERCEPT2, WorldModel07Keys::UPPER_BODY_CAMERA);
  registerPerceptKey<GoalPercept>(WorldModel07Keys::GOAL,
   GOAL_PERCEPT, WorldModel07Keys::UPPER_BODY_CAMERA);
  [...]
}
```

Listing A.2: KeyManager07.cpp Perzeptregistrierung

Die Registrierung erfolgt durch den Aufruf der Methode *registerPerceptKey*. Dies ist eine Template-Funktion, die neben dem Datentyp des Perzepts einen Weltmodellschlüssel, den Frameworkschlüssel sowie einen Sensorschlüssel übergeben bekommt. Damit erfolgt an dieser Stelle eine eindeutige Zuordnung der eben aufgezählten Schlüssel für das jeweilige Perzept.

A.3 Hinzufügen neuer Modellierungen

Um der Weltmodellierung eine neue Modellierung hinzufügen zu können, muss diese von der Klasse *Modeling* abgeleitet werden. Diese dient als Basisklasse für alle Modellierungen der Weltmodellierung (siehe Abschnitt 6.1.4.1). Des Weiteren muss die neue Modellierung in der Klasse *WorldModelModule07.cpp* eingetragen werden (siehe Listing A.3).

```
Beispiel WorldModelModule07.cpp;

WorldModelModule07::WorldModelModule07(
  KeyManager07* keyManager,
  const PlayerConfiguration& playerConfiguration,
  std::string parameterFilename,
  roboapp::Key imageInfo, roboapp::Key sampleSet,
  roboapp::Key selflocatorVariantsFromGui, roboapp::Key
  selflocatorVariantsToGui, roboapp::Key resetSamples,
  roboapp::Key kalmanParameters,
  roboapp::Key alternatePose, roboapp::Key useRobotOracle,
  roboapp::Key useBallOracle, roboapp::Key useKalman,
  roboapp::Key worldModelParameters,
  roboapp::Key communicationParameters,
  roboapp::Key inData, roboapp::Key outData,
  roboapp::Key inStructure, roboapp::Key outStructure)
    : WorldModelModule(keyManager,
      playerConfiguration.getTeamNumber(),
      playerConfiguration.getPlayerNumber(),
      worldModelParameters, communicationParameters,
      inData, outData, inStructure, outStructure) {

  modelingHandler.addModeling(new BallModeling(worldData,
    WorldModel07Keys::BALL, WorldModel07Keys::BALLORACLE,
    WorldModel07Keys::VALIDITYPOSE,
    WorldModel07Keys::ROBOTORACLE,
    WorldModel07Keys::ODOMETRY, WorldModel07Keys::FOOT,
    WorldModel07Keys::HEAD_CAMERA,
    WorldModel07Keys::UPPER_BODY_CAMERA,
    kalmanParameters, useRobotOracle,
    useBallOracle, useKalman));

  modelingHandler.addModeling(new OdometryModeling(
    worldData, WorldModel07Keys::ODOMETRY));

  modelingHandler.addModeling(new CircleModeling(
    worldData, WorldModel07Keys::CIRCLE));

  [...]
}
```

Listing A.3: WorldModelModule07.cpp Modellierungsregistrierung

Dazu muss im Konstruktor der Klasse die Methode *addModeling* auf dem ModelingHandler aufgerufen werden. Über diese Methode wird dem ModelingHandler eine Instanz der neuen Modellierung übergeben. Im Listing ist dies am Beispiel der Ball-, Odometrie- und Kreismodellierung zu sehen.

Im Konstruktor der Modellierungen werden alle zur Initialisierung benötigten Weltmodell- und Sensorschlüssel übergeben. Jede Modellierung benötigt dafür mindestens einen Weltmodellschlüssel, der das von ihm berechnete Ergebnismodell definiert und der Basisklasse weitergereicht wird (siehe Listing A.4). Ist dieser Weltmodellschlüssel noch nicht angelegt, muss dies in der Klasse WorldModelKeys07.cpp vorge-

nommen werden (siehe Abschnitt A.1)

```
Beispiel BallModeling.cpp;

BallModeling::BallModeling(
  WorldData& worldData,
  const WorldKey worldKey, const WorldKey odometryKey,
  const SensorKey headCam, const SensorKey bodyCam,
  [...]
: Modeling(worldKey),
  inBallPerceptData(worldData.getInPerceptData
    <vision::BallPercept>(worldKey)),
  odometryModelData(worldData.getInModelData
    <OdometryModel>(odometryKey)),
  inTeamBallModelData(worldData.getInTeamModelData
    <BallModel>(worldKey)),
  [...]
{
  [...]
}
```

Listing A.4: BallModeling.cpp Modellierungserstellung

Weitere Weltmodellschlüssel werden benötigt, um die für die Berechnung notwendigen Weltdaten aus WorldData anzufordern. Im Beispiel ist zu sehen, wie die Datenobjekte *inBallPerceptData, odometryModelData* und *inTeamBallModelData* mithilfe der Schlüssel initialisiert werden.

Damit die Bereitstellung der Modelle automatisiert ablaufen kann, müssen auch die aus den Modellierungen resultierenden Modelle in der Weltmodellierung registriert werden. Dies geschieht analog zur Perzeptregistrierung in der Klasse *KeyManager07.cpp* (siehe Listing A.5).

```
Beispiel KeyManager07.cpp;

KeyManager07::KeyManager07() {
  registerModelKeys();
}
KeyManager07::~KeyManager07() {
  cleanUpFactories();
}
[...]
void KeyManager07::registerModelKeys() {
  registerModelKey<BallModel>(WorldModel07Keys::BALL,
    BALL_MODEL, "Ball");
  registerModelKey<GoalModel>(WorldModel07Keys::GOAL,
    GOAL_MODEL, "Goal");
  registerModelKey<CircleModel>(WorldModel07Keys::CIRCLE,
    CIRCLE_MODEL, "Circle");
  [...]
}
```

Listing A.5: KeyManager07.cpp Modellregistrierung

Die Registrierung erfolgt über den Aufruf der Methode *registerModelKey*. Dies ist eine Template-Funktion, die neben dem Datentyp des Modells einen Weltmodellschlüssel, den Frameworkschlüssel sowie den Namen des Modells in Form einer Zeichenkette übergeben bekommt. Damit erfolgt an dieser Stelle eine eindeutige Zuordnung der eben aufgezählten Schlüssel für das jeweilige Modell.

A.4 Verwendung von Weltmodellfunktionen

In der Klasse *ModelingOperations.cpp* können allgemeine, für alle Modellierungen geltende Methoden implementiert werden. Die Methode *updateReliability* bietet die Möglichkeit, Wahrscheinlichkeiten taktunabhängig sinken zu lassen (siehe Listing A.6).

```
Beispiel: ModelingOperations.cpp;

void ModelingOperations::updateReliability(
    const roboapp::Timestamp& lastUpdate,
    unsigned long lifeMilliseconds, double& reliability) {
  double timeDifference = roboapp::SystemCall::
    getTimestamp().getMilliseconds() -
    lastUpdate.getMilliseconds();
  reliability = reliability -
    (timeDifference / lifeMilliseconds);
  if (reliability < 0) {
    reliability = 0;
  }
}
```

Listing A.6: ModelingOperations.cpp Modellierungsfunktionen

Als Übergabeparameter müssen der Methode der Zeitstempel des letzten Aufrufs, die Zeit, in der die Wahrscheinlichkeit von 1 auf 0 sinken soll, sowie die zu verändernde Wahrscheinlichkeit übergeben werden. Die neu berechnete Wahrscheinlichkeit steht dann in der übergebenen Referenz *reliabiliy* zur Verfügung.

A.5 Empfang und Verwendung von Teammodellen

Grundvoraussetzung für den Empfang von Teammodellen ist die Existenz eines entsprechenden Modells, da ein Teammodell auf einem bestehenden Modell aufbaut. Die Registrierung eines solchen Teammodells ist in Abschnitt 6.1.3.3 beschrieben. Wenn das entsprechende Modell besteht, muss lediglich das Teammodell in der Klasse *KeyManager07.cpp* registriert werden (siehe Listing A.7).

```
Beispiel KeyManager07.cpp;

KeyManager07::KeyManager07() {
  registerTeamModelKeys();
}
KeyManager07::~KeyManager07() {
  cleanUpFactories();
}
[...]
void KeyManager07::registerTeamModelKeys(){
  registerTeamModelKey<BallModel>(
    WorldModel07Keys::BALL, TEAM_BALL_MODEL);
  registerTeamModelKey<GoalModel>(
    WorldModel07Keys::GOAL, TEAM_GOAL_MODEL);
  registerTeamModelKey<CircleModel>(
    WorldModel07Keys::CIRCLE, TEAM_CIRCLE_MODEL);
  [...]
}
```

Listing A.7: KeyManager07.cpp Teammodellregistrierung

Die Registrierung erfolgt über den Aufruf der Methode *registerTeamModelKey*. Dies ist eine Template-Funktion, die neben dem Datentyp des Teammodells einen Weltmodellschlüssel und den Frameworkschlüssel übergeben bekommt. Damit erfolgt an dieser Stelle eine eindeutige Zuordnung der eben aufgezählten Schlüssel für das jeweilige Teammodell.

A.6 Versendung von Daten

Die für den Datenaustausch vorgesehenen Werte werden von den Modellen festgelegt. Gezeigt wird dies am Beispiel von *BallModel.cpp* (siehe Listing A.8)

```
Beispiel: BallModel.cpp

void BallModel::getAttributeIdentifier(
    std::vector<ModelIdentifier>& attributes) {
  setAttributeIdentifier(
    attributes, "relative.x", "int");
  setAttributeIdentifier(
    attributes, "relative.y", "int");
  setAttributeIdentifier(
    attributes, "absolute.x", "int");
  setAttributeIdentifier(
    attributes, "absolute.y", "int");
  setAttributeIdentifier(
    attributes, "reliability", "double");
  setAttributeIdentifier(
    attributes, "teamBall", "bool");
}
```

```
void BallModel::registerAttributes() {
  registerValue(&relative.x);
  registerValue(&relative.y);
  registerValue(&absolute.x);
  registerValue(&absolute.y);
  registerValue(&reliability);
  registerValue(&teamBall);
}
```

Listing A.8: BallModel.cpp Modellerstellung

Die zur Bildung der lokalen Struktur benötigten Attributidentifier werden von der Kommunikation über die Methode *getAttributeIdentifier* abgerufen. Um die eigenen Attribute dem übergebenen Vektor hinzufügen zu können, steht die Methode *setAttribute-Identifier* zur Verfügung, der der Vektor, der Identifier für das Attribut und der Identifier für den Attributtyp übergeben wird.

Damit es der Kommunikation möglich ist, empfangene Daten in ein neues Teammodell zu schreiben sowie die zu versendenden Daten aus einem bestehenden Modell abzufragen, müssen Pointer auf die entsprechenden Attribute in der Reihenfolge der hinzugefügten Identifier registriert werden. Dafür steht die Methode *registerValue* bereit, der die erwähnten Pointer übergeben werden müssen.

Registriert ein Modell keine Attribute, wird es von der Kommunikation für den Datenaustausch ignoriert.

A.7 Weitergabe der Daten an die Verhaltenssteuerung

Die Verhaltenssteuerung wird in oberster Ebene von der Klasse *BehaviorControl.cpp* übernommen. Hier werden die Unterkomponenten der Verhaltenssteuerung erzeugt und initialisiert. Die Klasse besteht aus einer einzigen statischen Methode *getXabslControl* (siehe Listing A.9). Sie hat als Rückgabewert das Objekt XabslControl, welches die Ansteuerung der Zustandsmaschine bereitstellt, was im Abschnitt 6.3 beschrieben wurde. Es wird am Ende des hier aufgeführten Beispiels instantiiert und bekommt unter anderem die beiden Maps *bbProviders* und *sProviders* übergeben, welche die für das Verhalten benötigten BasicBehavior- und SymbolProvider beinhalten.

```
Beispiel: BehaviorControl.cpp

XabslControl* BehaviorControl::getXabslControl(
  [...]
) {

  std::map<BasicBehaviorProvider*, bool>* bbProviders =
    new std::map<BasicBehaviorProvider*, bool>();

  [...]

  MotionProvider* motionProvider =
    new MotionProvider(outMotionRequestKey,
    robotStatusKey, outMotionRequestDrawingKey);

  motionProvider->addBasicBehavior(
    new DoNothing(motionProvider));
  motionProvider->addBasicBehavior(
    new DoSpecialAction(motionProvider));
  motionProvider->addBasicBehavior(
    new GoToBall(motionProvider, ballKey));

  [...]

  KickBall* bb_kickBall = new KickBall(motionProvider);
  motionProvider->addBasicBehavior(bb_kickBall);

  (*bbProviders)[motionProvider] = true;


  std::map<SymbolProvider*, bool>* sProviders =
     new std::map<SymbolProvider*, bool>();

  (*sProviders)[new HeadSymbols(
    inCameraMatrixKey, camera)] = true;
  (*sProviders)[new BallSymbols(
    ballKey, positionKey, gameStateKey)] = true;

  [...]

  return new XabslControl(
    "BehaviorControl", behaviorFilename,
    sProviders, bbProviders, eObservers,
    new XabslItemManipulator(inXabslExecuteKey),
    inIntermediateCode, agent
  );
}
```

Listing A.9: BehaviorControl.cpp Verhaltensanbindung

Der MotionProvider ist ein BasicBehaviorProvider, dem die verschiedenen Basisverhalten hinzugefügt werden. In diesem Beispiel sind dies *DoNothing, DoSpecialAction* und *GoToBall*. Wird ein neues Basisverhalten geschrieben, muss es lediglich an dieser

Stelle im MotionProvider über die Methode *addBasicBehavior* registriert werden und von der Klasse *IBasicBehavior.cpp* erben. Wenn neue Symbolklassen für das Verhalten geschrieben werden, müssen diese von der Klasse *SymbolProvider.cpp* abgeleitet werden und analog zu den *HeadSymbols* und *BallSymbols* der Map *sProviders* hinzugefügt werden.

B Abkürzungen

ADC Analog-Digital-Converter
Einheit, die analoge Eingangssignale in digitale Daten umwandelt.

Akku Akkumulator
Ein Speicher für elektrische Energie, meist in Form einer wiederaufladbaren Sekundär-Zelle.

API Application Programming Interface
Eine API bezeichnet die Schnittstelle, die ein Softwaresystem (zum Beispiel ein Betriebssystem) anderen Programmen zur Verfügung stellt. Die Schnittstelle besteht meist aus Routinen, Klassen und/oder Protokollen.

DD Darmstadt Dribblers[1]
RoboCup Humanoid League Mannschaft der Technischen Universität Darmstadt.

DDD Darmstadt Dribbling Dackels[2]
RoboCup Four-Legged League Mannschaft der Technischen Universität Darmstadt.

GT German Team[3]
Deutsche Nationalmannschaft der RoboCup Four-Legged League.

GUI Graphical User Interface
Eine grafische Benutzeroberfläche zur Visualisierung einzelner Softwarekomponenten.

IR Infrarot
Bezeichnung für elektromagnetische Wellen im Spektralbereich zwischen sichtbarem Licht und den langwelligeren Mikrowellen.

MCU Micro-Controller-Unit
Ein-Chip-Computersysteme, auf welchem nahezu alle benötigten Komponenten untergebracht sind.

MPU Micro-Processor-Unit
Schaltkreise, die keine eindeutig definierte Eingangs-Ausgangsrelation besitzen, sondern eine Abfolge von Operationsanweisungen abarbeiten.

---

[1] DD – Webseite: `http://www.dribblers.de`
[2] DDD – Webseite: `http://robocup.informatik.tu-darmstadt.de/`
[3] GT – Webseite: `http://www.germanteam.org`

MRS Multi-Roboter-Systeme
Roboter, die mit anderen Robotersystemen kooperativ Aufgaben lösen.

SIM Simulation, Systemoptimierung und Robotik[4]
Fachgebiet der Technischen Universität Darmstadt.

SVN Subversion[5]
Eine Open-Source-Software zur Versionsverwaltung von Dateien. Subversion wird häufig als Nachfolger von CVS bezeichnet.

TCP Transmission Control Protocol
Ein verbindungsorientiertes, zuverlässiges Transportprotokoll, welches eine Ende-zu-Ende-Verbindung und Teil der TCP/IP-Protokollfamilie ist.

UDP User Datagram Protocol
Ein verbindungsloses, nicht-zuverlässiges Netzwerkprotokoll, welches zur Transportschicht der TCP/IP-Protokollfamilie gehört.

XABSL Extensible Agent Behavior Specification Language
Auf Zustandsautomaten basierende Programmiersprache zur Entwicklung eines Verhaltens für autonome Systeme.

[4]SIM – Webseite: `http://www.sim.informatik.tu-darmstadt.de`
[5]SVN – Webseite: `http://subversion.tigris.org`

C Literaturverzeichnis

[1] DEFENSE ADVANCED RESEARCH PROJECTS AGENCY: *Grand Challenge der Defense Advanced Research Projects Agency Webseite:* `http://www.darpa.mil/grandchallenge/`. 2007

[2] DIGITAL-LOGIC AG: *Technical User's Manual For: PC/104 plus - MSM800SEV/SEL - Version 1.1*, 2006

[3] FACHGEBIET SIM DER TECHNISCHE UNIVERSITÄT DARMSTADT: *Darmstadt Dribblers Webseite:* `http://www.dribblers.de/`. 2007

[4] FACHGEBIET SIM DER TECHNISCHE UNIVERSITÄT DARMSTADT: *Fachgebiet SIM der Technische Universität Darmstadt Webseite:* `http://www.sim.informatik.tu-darmstadt.de/`. 2007

[5] FACHGEBIET SIMULATION UND SYSTEMOPTIMIERUNG: Mobile und sensorgeführte Robotiksysteme / Technische Universität Darmstadt. 2004. – Forschungsbericht

[6] FACHGEBIET SIMULATION UND SYSTEMOPTIMIERUNG: Vorgaben bei der Software-Entwicklung am Fachgebiet Simulation und Systemoptimierung - Version 0.3 / Technische Universität Darmstadt. 2007. – Forschungsbericht

[7] FRIEDMANN, Martin ; KIENER, Jutta ; PETTERS, Sebastian ; THOMAS, Dirk ; VON STRYK, Oskar: Team Description for Humanoid KidSize League of RoboCup 2007 / Department of Computer Science, Technische Universität Darmstadt. 2007. – Forschungsbericht

[8] FRIEDMANN, Martin ; PETTERS, Sebastian ; RISLER, Max ; SAKAMOTO, Hajime ; VON STRYK, Oskar ; THOMAS, Dirk: Proposal of an Autonomous Four-Legged Robot as a New Versatile, Modular and Affordable Platform for Research and Education / SIM - Technische Universität Darmstadt, Hajime Research Institue, Ltd. 2007. – Forschungsbericht

[9] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2002

[10] KIENER, Jutta: *Heterogene Teams kooperierender autonomer Roboter*, Technische Universität Darmstadt, Diss., 2006

[11] KIENER, Jutta ; PETTERS, Sebastian ; THOMAS, Dirk ; FRIEDMANN, Martin ; VON STRYK, Oskar: Architektur und Komponenten für ein heterogenes Team kooperierender, autonomer humanoider Roboter / SIM - Technische Universität

Darmstadt. 2005. – Forschungsbericht

[12] LÖTSCH, Martin ; BACH, Joscha ; BURKHARD, Hans-Dieter ; JÜNGEL, Matthias: *Designing Agent Behavior with the Extensible Agent Behavior Specification Language XABSL*. RoboCup 2003: Robot Soccer World Cup VII Bd. 3020. Padova, Italy, 2003

[13] MCKERROW, Phillip J.: *Introductions to Robotics*. Addison-Wesley Publishing Company, 1991

[14] MOBILEROBOTS: *MobileRobots Webseite: `http://www.activrobots.com/`*. 2007

[15] MURPHY, Robin R.: *Introduction to AI robotics S. 293 - 313*. Massachusetts Institute of Technology, 2000

[16] PETTERS, Sebastian ; THOMAS, Dirk: *RoboFrame - Softwareframework für mobile autonome Robotersysteme*, Technische Universität Darmstadt, Diplomarbeit, 2005

[17] RENESAS TECHNOLOGY: *Hardware Manual SH7125 Group, SH7124 Group - Rev. 2.00*, 2006

[18] RÖFER, Thomas ; BROSE, Jörg ; CARLS, Eike ; CARSTENS, Jan ; GÖHRING, Daniel ; JÜNGEL, Matthias ; LAUE, Tim ; OBERLIES, Tobias ; OESAU, Sven ; RISLER, Max ; SPRANGER, Michael ; WERNER, Christian ; ZIMMER, Jörg: GermanTeam 2006 - The German National RoboCup Team / Deutsches Forschungszentrum für Künstliche Intelligenz - Safe and Secure Cognitive Systems, Fachgebiet Simulation und Systemoptimierung - Fachbereich Informatik - Technische Universität Darmstadt, Fachbereich 3 - Mathematik / Informatik - Universität Bremen, Institut für Informatik, LFG Künstliche Intelligenz - Humboldt-Universität zu Berlin. 2006. – Forschungsbericht

[19] ROBOCUP: *RoboCup Humanoid League Webseite: `http://www.humanoidsoccer.org/`*. 2007

[20] ROBOCUP FEDERATION: *RoboCup Federation Call for Tenders: A Standard Robot Platform for Robot Soccer*. 2006

[21] TECHNOLOGIEZENTRUM INFORMATIK, TZI: *RoboCup - Four-Legged League Webseite: `http://www.tzi.de/4legged/`*. 2007

[22] TEMPLER, Simon: Kopfsteuerung für einen humanoiden Roboter / Technische Universität Darmstadt. 2006. – Forschungsbericht

[23] THRUN, S. ; BENNEWITZ, M. ; BURGARD, W. ; CREMERS, A. ; DELLAERT, F. ; FOX, D. ; HAHNEL, D. ; ROSENBERG, C. ; ROY, N. ; SCHULTE, J. ; SCHULZ, D.: *Minerva: a second-generation museum tour-guide robot*. IEEE International Conference on Robotics and Automation (ICRA) Bd. 3, 1999