Freie Universität Berlin
Fachbereich Mathematik und Informatik
Institut für Informatik

Bachelorarbeit

Verhaltenssteuerung für autonome humanoide Fußballroboter mit XABSL

Rico Jonschkowski

Gutachter: Prof. Dr. Raúl Rojas
Betreuer: Hamid Reza Moballegh

22. Dezember 2010

Zusammenfassung

2010 war es nötig, eine neue Verhaltenssteuerung für die autonomen humanoiden Fußballroboter des RoboCup-Teams FUmanoids zu entwickeln. Die Aufgabe der Verhaltenssteuerung ist die Ableitung sinnvoller Aktionen aus den Wahrnehmungen des Roboters. Bisher lösten die Entwickler diese unter Verwendung der CSBP-Architektur. Mit zunehmender Komplexität des Verhaltens wurde die Implementierung dabei jedoch immer unübersichtlicher. Deshalb wurde mit XABSL ein anderes Verhaltenssteuerungssystem ausgewählt und in die Roboter-Software integriert. Mithilfe dieses auf hierarchischen endlichen Zustandsautomaten basierenden Systems entstand eine Verhaltenssteuerung, die beim RoboCup 2010 in Singapur erfolgreich zum Einsatz kam. Die FUmanoids belegten bei diesem Wettkampf den zweiten Platz in der RoboCup KidSize League und gewannen die Technical Challenge.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht im Rahmen eines anderen Prüfungsverfahrens eingereicht wurde.

Berlin, den 22. Dezember 2010

Inhaltsverzeichnis

1 Einleitung **5**

1.1 Robotik . . . 5
1.2 RoboCup Humanoid KidSize League . . . 6
1.3 FUmanoids . . . 7
1.4 Verhaltenssteuerung . . . 8
1.5 Ziel und Aufbau dieser Arbeit . . . 9

2 Verhaltenssteuerung 2009 **10**

2.1 Concurrent Scenario-Based Planning (CSBP) . . . 10
2.2 Die vier Ebenen . . . 11
2.3 Probleme . . . 13

3 Systeme zur Verhaltenssteuerung **14**

3.1 Anforderungen . . . 14
3.2 Verschiedene Ansätze . . . 14
3.2.1 Wissensbasierte Systeme . . . 14
3.2.2 Verhaltensbasierte Systeme . . . 14
3.3 Auswahl eines Systems . . . 16
3.4 XABSL . . . 17
3.4.1 Konzepte und Begriffe . . . 17
3.4.2 Beispiel XABSL-Code . . . 18
3.4.3 Ausführung des Beispiels . . . 21
3.4.4 Einbindung von XABSL in die FUmanoid-Software . . . 22

4 Verhaltenssteuerung 2010 **22**

4.1 Verwendete Grundlagen der FUmanoid-Software . . . 22
4.1.1 Statische und dynamische Bewegungen . . . 22
4.1.2 Das Laufen . . . 23
4.1.3 Das Weltmodell . . . 23
4.2 Andere Ansätze als 2009 . . . 24
4.2.1 Immer zum Tor ausrichten . . . 25
4.2.2 Raumdeckung . . . 26
4.2.3 Zentrale Strategieentscheidung . . . 27
4.3 Überblick . . . 28
4.4 Detaillierte Beschreibung . . . 29
4.4.1 Kopfsteuerung . . . 29
4.4.2 Bewegungssteuerung . . . 30

4.4.3 Fußballfähigkeiten 36
4.4.4 Der Feldspieler 40
4.4.5 Rollen 41
4.4.6 Rollenverteilung und Teamkoordination 44
4.4.7 Spielsteuerung 44

5 Fazit und Ausblick **45**

1 Einleitung

1.1 Robotik

Robotik und Künstliche Intelligenz sind zentrale Themengebiete in der Informatik. Sie beschäftigen sich mit der Entwicklung intelligenter Maschinen, die eigenständig Probleme lösen können.

Der Traum von intelligenten Werkzeugen ist aber schon sehr viel älter als die Informatik. Schon Aristoteles schrieb: „Wenn jedes Werkzeug auf Geheiß, oder auch vorausahnend, das ihm zukommende Werk verrichten könnte, [...] so bedürfte es weder für den Werkmeister der Gehilfen noch für die Herren der Sklaven“ [1].

Doch erst seit der Mitte des letzten Jahrhunderts werden solche Werkzeuge als Roboter bezeichnet. Der Begriff Roboter ist aus dem slawischen Wort für Arbeit *robota* abgeleitet und wurde vor allem von Isaac Asimov in seinen Science-Fiction-Romanen geprägt.

Etwa zur gleichen Zeit wurden die ersten wirklichen Roboter entwickelt. In den 1970er Jahren haben sie schließlich in Form von Industrierobotern Einzug in die Fabriken gehalten. Bis heute ist dies das größte Einsatzgebiet der Robotik.

An die Roboter der Zukunft werden jedoch höhere Anforderungen gestellt. Autonome mobile Roboter sollen sich selbstständig in ihrer Umwelt bewegen und agieren können, sowie in verschiedensten Anwendungsbereichen eingesetzt werden.

Das langfristige Ziel sind intelligente Roboter, die in der Lage sind, verschiedene komplexe Aufgaben zu lösen. In der nächsten Zeit wird es aber vor allem spezialisierte Roboter geben, die nur ein sehr begrenztes Spektrum an Aufgaben übernehmen können. So sind bereits heute diverse Putz-, Wach- und Unterhaltungsroboter erhältlich. Für letztere herrscht besonders in den asiatischen Ländern eine regelrechte Euphorie. Solche Roboter interagieren mit Menschen auf eine sehr intuitive Art und haben oft die Gestalt von Haustieren. Sie werden auch eingesetzt, um ältere, vereinsamte Menschen geistig fit zu halten.

In Zukunft werden Senioren eine große Zielgruppe der Robotik sein. Denn aufgrund der gegenwärtigen demografischen Entwicklung mit einer zunehmenden Überalterung unserer Gesellschaft wird es einen immer größer werdenden Bedarf an Pflege und Betreuung älterer Menschen geben. Roboter könnten Senioren dabei helfen, länger selbstständig zu bleiben und außerdem die PflegerInnen in den Heimen entlasten, indem sie verschiedene alltägliche Aufgaben übernehmen.

Ein anderes Szenario für den Einsatz von Robotern ist das Retten und Bergen von Menschen in Katastrophengebieten. Helfer begeben sich hierbei oft selbst in Gefahr, was durch den Einsatz von Robotern weitestgehend vermieden werden könnte.

Für die aufgeführten Anwendungen benötigen die mobilen Roboter allerdings mehr Intelligenz als ihre feinmotorischen Verwandten in den Fabriken. Dabei gibt es einige Kernprobleme, die unabhängig von der konkreten Anwendung zu lösen sind: Roboter müssen ihre Umwelt wahrnehmen, Objekte erkennen und sich in einer dynamisch verändernden Umgebung orientieren können. Aus diesen Informationen müssen sie ein intelligentes Verhalten ableiten und ihre Aktionen sinnvoll planen und ausführen.

Das Ziel der Forschung im Bereich Robotik besteht darin, diese Probleme zu lösen. Dabei wird auf ganz unterschiedliche Art und Weise vorgegangen. Ein außergewöhnlicher und sehr erfolgreicher Ansatz wird dabei vom *RoboCup* verfolgt.

1.2 RoboCup Humanoid KidSize League

Der RoboCup ist eine internationale Forschungsinitiative, die Fußball als Anwendung für die Lösung der elementaren Probleme der Robotik gewählt hat und in verschiedenen Ligen unterschiedliche Anforderungsschwerpunkte setzt. Der RoboCup will damit die Grundlagenforschung in der Robotik vorantreiben und Studenten gleichzeitig die Möglichkeit von projektorientiertem Lernen geben. Die wichtigste Veranstaltung dieser Initiative ist die seit 1997 jährlich stattfindende *RoboCup World Championship and Conference*. Hier messen sich Teams von Universitäten der ganzen Welt im Roboterfußball und tauschen sich über ihre neuesten Entwicklungsergebnisse aus.

Der RoboCup hat sich ein ehrgeiziges Ziel gesetzt: Bis 2050 soll ein Team menschenähnlicher Roboter entwickelt werden, das den menschlichen Fußballweltmeister besiegen kann. Dieses Ziel ist bewusst so gewählt, dass humanoide Roboter im Mittelpunkt stehen, weil langfristig betrachtet solche Roboter entwickelt werden, da diese per Definition optimal auf die für Menschen geschaffene Umwelt angepasst sind. Humanoide Roboter könnten dieselben Wege benutzen wie Menschen, sie könnten dieselben Autos fahren und Menschen könnten mit ihnen ganz intuitiv interagieren.

Daher kommt der Liga der humanoiden Roboter eine besondere Rolle zu. Beim Fußball spielen müssen die Roboter stabil auf zwei Beinen laufen und dürfen nur solche Sensoren benutzen, die der menschlichen Wahrnehmung entsprechen. Die *Humanoid League* ist nach Größe der Roboter in drei Kategorien unterteilt. In der kleinsten dieser Kategorien, der *KidSize*, spielen Roboter mit bis zu 60 Zentimeter Körpergröße auf einem vier mal sechs Meter großen Spielfeld mit drei Robotern je Team [20]. In den anderen Kategorien *TeenSize* und *AdultSize* sind die Roboter bis zu 120 bzw. 160 Zentimeter groß. Die zusätzliche Schwierigkeit in diesen Kategorien besteht in der stabilen Konstruktion der größeren Roboter. Wegen dieser Hürde sind die Spiele der Teen- und AdultSize bisher weniger dynamisch als die der KidSize. Außerdem gibt es in jeder Kategorie eine *Technical Challenge*, in der Fähigkeiten gezeigt werden, die von den wenigsten Teams schon im Spiel angewandt werden. In der Technical Challenge der

KidSize League müssen ein Einwurf, das Dribbeln um Hindernisse und Passspiel demonstriert werden.

1.3 FUmanoids

FUmanoids ist ein studentisches Projekt unter der Betreuung von Prof. Dr. Raúl Rojas, das 2006 als Teil der *Arbeitsgruppe Künstliche Intelligenz* am Institut für Informatik der Freien Universität Berlin gegründet wurde. 2007 nahmen die FUmanoids zum ersten Mal in der Humanoid KidSize League teil und belegten bei der RoboCup-Weltmeisterschaft in Atlanta den dritten Platz. Im darauf folgenden Jahr erreichten sie den zweiten Platz bei den RoboCup German Open. 2009 feierten sie ihren bis dahin größten Erfolg und wurden Vizeweltmeister beim RoboCup in Graz. Die in dieser Arbeit entwickelte Verhaltenssteuerung kam beim RoboCup 2010 in Singapur zum Einsatz, wo die FUmanoids erneut zum Vizeweltmeister und außerdem zum Sieger in der Technical Challenge gekürt wurden.

Roboter-Hardware

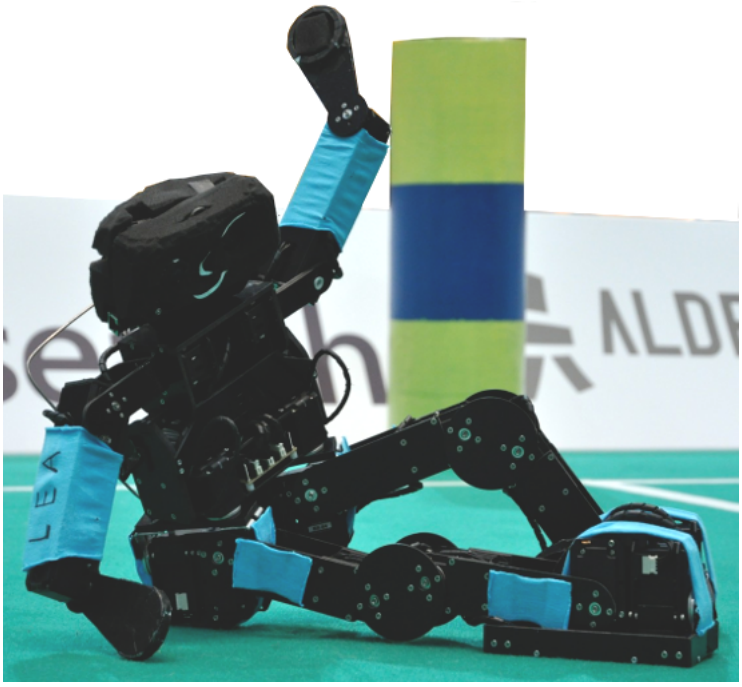


Abbildung 1: Lea - FUmanoids KidSize Roboter Version 2009

Diese Arbeit bezieht sich auf die Roboter aus dem Vorjahr (siehe Abbildung 1). Die Roboter Version 2009 sind 60 Zentimeter groß, besitzen 21 Servomotoren und wiegen viereinhalb Kilogramm. Die Sensorik umfasst eine Kamera mit VGA-Auflösung und Weitwinkelobjektiv, einen Inertialsensor, der Beschleunigungskräfte misst, sowie Druckschalter an den Füßen.

Die Informationen der Sensoren werden von einem Computer auf dem Roboter mit 600 MHz Taktrate verarbeitet. Die Roboter kommunizieren über eine WLAN-Verbindung.

1.4 Verhaltenssteuerung

Begriffsklärung

Die *Verhaltenssteuerung* stellt die Verbindung zwischen Wahrnehmung und Aktion der Roboter her. Sie wird aus verschiedenen *Verhalten* aufgebaut, die abhängig von der jeweiligen Situation ausgeführt werden. Die für die Implementierung der Verhaltenssteuerung benötigten Programmiersprachen, Bibliotheken, Übersetzer und andere Werkzeuge werden folgend unter dem Begriff *Verhaltenssteuerungssystem* zusammengefasst.

Einordnung in die FUmanoid-Software

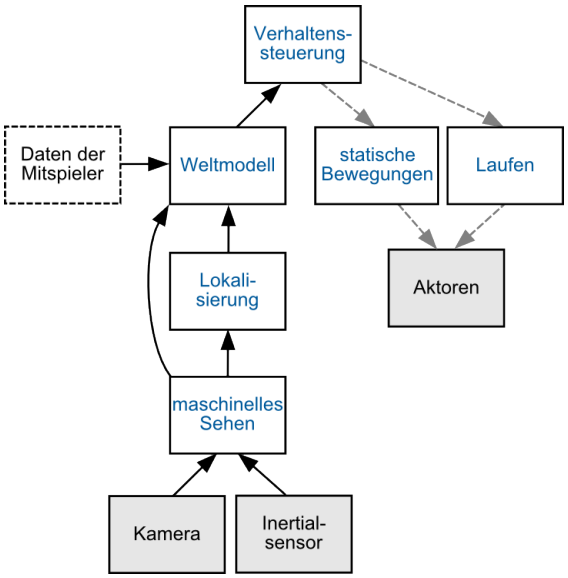


Abbildung 2: Schematischer Aufbau der FUmanoid Software (weiße Rechtecke stellen Softwaremodule dar; graue Rechtecke sind Teile der Hardware; schwarze Pfeile markieren den Informationsfluss, grau gestrichelte Pfeile den Aktivierungspfad)

Die Steuerung der Roboter wird von der *FUmanoid-Software* übernommen, welche in C++ geschrieben ist und unter Linux ausgeführt wird [22]. Diese Software ist in einzelne *Module* unterteilt (siehe Abbildung 2), die verschiedene Aufgaben übernehmen. Das Modul *Maschinelles Sehen* wertet die Bilder der Kamera aus und extrahiert daraus Objekte und die Linien des Spielfeldes. Dabei wird der Inertialsensor benutzt, um die Neigung der Kamera zu errechnen. Aus diesen Informationen bestimmt die *Selbstlokalisierung* die wahrscheinlichste Position des Roboters auf dem Feld. Zusammen mit den Daten, welche die anderen Roboter über das

Netzwerk verteilen, wird daraus das *Weltmodell* generiert. Die *Verhaltenssteuerung* bezieht Informationen aus diesem Modell und steuert abhängig davon die Bewegungen des Roboters. Für die Ausführung der Bewegungen existieren ebenfalls eigene Module.

Schwierigkeiten bei der Entwicklung der Verhaltenssteuerung

Die Verhaltenssteuerung ist demnach das letzte Glied einer langen Kette von Modulen, die alle fehlerfrei funktionieren müssen, damit der Roboter richtig agiert. Ein Fehler an beliebiger Stelle dieser Kette kann zu einer Fehlfunktion im Verhalten führen. Tatsächlich werden viele Probleme der vor- und nachgelagerten Module erst beim Test der Verhaltenssteuerung deutlich, weil deren Zusammenspiel nur dabei unter realen Bedingungen getestet wird. Hierbei ist nicht immer sofort erkennbar, welche Module fehlerhaft sind. Außerdem ist das Verhalten auch vom Zustand der Hardware der einzelnen Roboter abhängig. Es sind immer Unterschiede bei den Motoren, der Kamera, der *Konfiguration* und der *Kalibrierung* vorhanden. Schon geringe Abweichungen können dazu führen, dass die Verhaltenssteuerung auf einem Roboter funktioniert, auf einem anderen hingegen nicht. Aus diesem Grund ist das Testen der Verhaltenssteuerung sehr aufwendig.

Hinzu kommt, dass die Verhaltenssteuerung am besten mit Erkenntnissen aus Spielen gegen andere Mannschaften weiterentwickelt werden kann. Spiele finden jedoch meistens erst in den Wettkämpfen statt, weshalb dort unter enormem Druck an der Verhaltenssteuerung gearbeitet wird. Oft müssen große Änderungen in wenigen Stunden durchgeführt und getestet werden. Die Arbeit unter diesen Bedingungen macht die Schwierigkeit der Entwicklung einer guten Verhaltenssteuerung aus.

1.5 Ziel und Aufbau dieser Arbeit

Das Ziel dieser Arbeit war es, ein neues System zur Verhaltenssteuerung in die FUmanoid-Software zu integrieren, um die Entwicklung in diesem Bereich zu vereinfachen. Außerdem sollte mit diesem System eine neue Verhaltenssteuerung entwickelt werden, um sie im RoboCup 2010 einzusetzen.

Dazu wird in Abschnitt 2 zunächst die Verhaltenssteuerung 2009 und deren Probleme beschrieben, durch welche die Verwendung eines anderen Verhaltenssteuerungssystems notwendig wurde. Unter Punkt 3 werden die Anforderungen an dieses spezifiziert und anhand dieser *XABSL* als neues System ausgewählt und beschrieben. Im Abschnitt 4 werden die Grundlagen der verwendeten Module aus der FUmanoid-Software erläutert und begründet, warum welche neuen Ansätze in der Verhaltenssteuerung 2010 verfolgt werden. Anschließend wird die in XABSL umgesetzte Verhaltenssteuerung 2010 im Detail beschrieben. Schließlich gibt es unter Punkt 5 ein Fazit, in dem die Verhaltenssteuerungen von 2009 und 2010 verglichen werden, und einen Ausblick auf die weitere Arbeit in diesem Bereich.

2 Verhaltenssteuerung 2009

2.1 Concurrent Scenario-Based Planning (CSBP)

Das 2009 eingesetzte Verhaltenssteuerungssystem ist eine Umsetzung der *Concurrent Scenario -Based Planning* (CSBP) Architektur von Hamid Moballegh [16].

Szenarien

Das Gesamtverhalten wird in diesem System in einzelne *Szenarien* unterteilt. Ein Szenario ist die Implementierung eines Teilverhaltens zur Lösung einer konkreten Aufgabe. Szenarien beenden sich *erfolgreich*, wenn sie die entsprechende Aufgabe bewältigt haben, oder *erfolglos*, wenn sie von etwas daran gehindert wurden.

Ein Beispiel dafür ist das *zum Ball laufen*. Dabei dreht der Roboter seinen Körper zum Ball und läuft vorwärts. Die Aufgabe ist erfüllt, wenn der Roboter nahe genug am Ball ist. In diesem Fall beendet sich das Szenario erfolgreich. Wenn der Roboter den Ball aus den Augen verliert oder stürzt, terminiert es erfolglos.

Ebenen

Die Szenarien werden auf unterschiedliche *Ebenen* aufgeteilt. Je höher die Ebene, desto abstrakter sind die ihr zugeordneten Szenarien. Je niedriger die Ebene, desto hardwarenäher und konkreter sind sie. Auf jeder Ebene ist von diesen Szenarien immer genau eines aktiv. Es werden daher, der Anzahl der Ebenen entsprechend, Szenarien parallel ausgeführt, wobei jedes Szenario bestimmt, welches auf der nächst niedrigeren Ebene aktiv ist (siehe Abbildung 3). Wenn Szenarien terminieren, geben sie an die nächst höhere Ebene Rückmeldung über Erfolg oder Misserfolg.

2.2 Die vier Ebenen

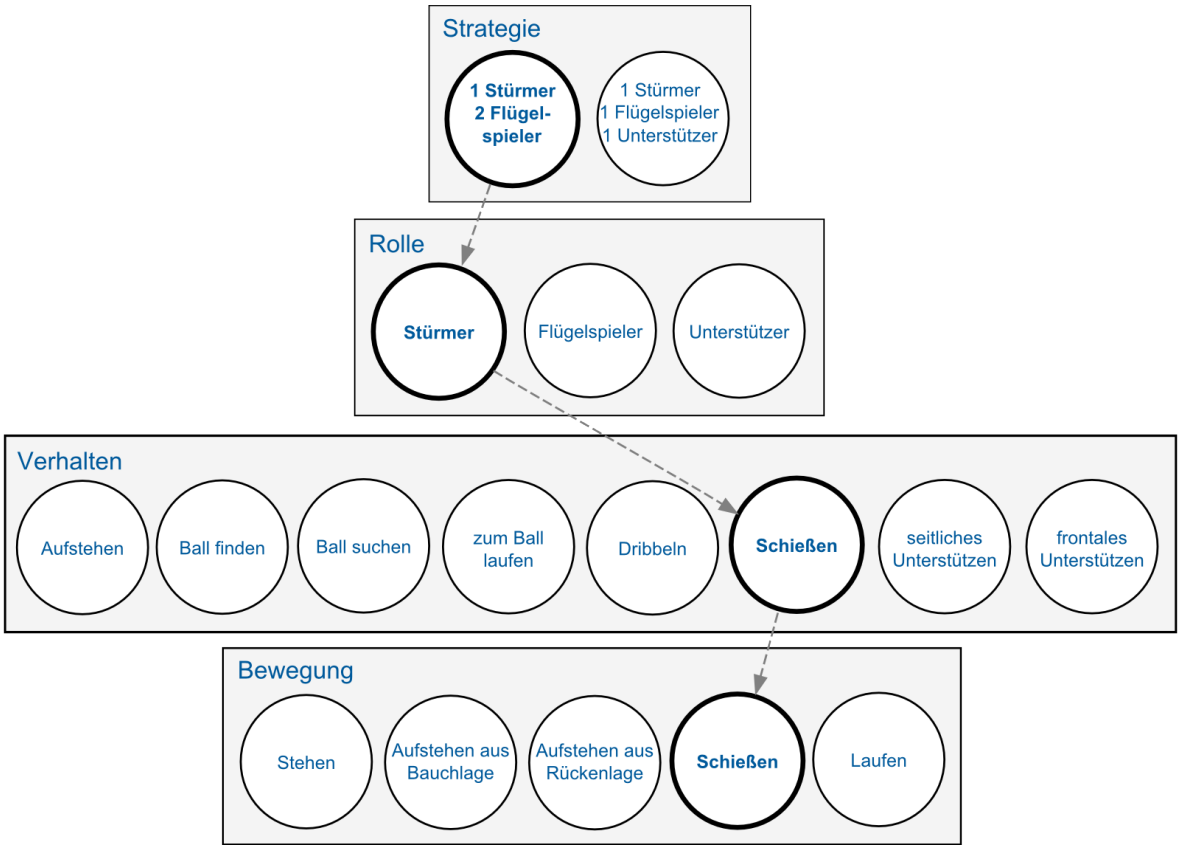


Abbildung 3: Beispiel für die Aktivierung der Szenarien in der Verhaltenssteuerung 2009

Die Verhaltenssteuerung 2009 wurde ausführlich von Gretta Hohl in ihrer Diplomarbeit [8] beschrieben und wird hier nur kurz umrissen. Sie basiert auf der CSBP Architektur und definiert vier Ebenen: Strategie, Rolle, Verhalten und Bewegung (siehe Abbildung 3). Szenarien werden dabei in C++ als Funktionen implementiert. Ebenen entsprechen Threads, in denen jeweils eine solche Funktion ausgeführt wird.

Strategie

Die *Strategie* ist die oberste Ebene der Verhaltenssteuerung. Beim Starten der FUmanoid-Software wird festgelegt, welches Szenario hier ausgeführt wird. Dieses entscheidet darüber, wie die Roboter sich als Team verhalten. Dazu hat die Strategieebene Kenntnis über alle Szenarien, die zu dem Zeitpunkt auf den anderen Ebenen aktiv sind. Da diese Daten über das Netzwerk unter den Robotern ausgetauscht werden, ist außerdem bekannt, welche Szenarien auf den anderen Robotern ausgeführt werden. Anhand dieser Informationen bestimmt die Strategie, welches Szenario auf der *Rollenebene* aktiv ist.

So wird beispielsweise dem Spieler, der zuerst erfolgreich zum Ball gelaufen ist, die Rolle des *Stürmers* zugeordnet, während seine Mitspieler die Anweisung bekommen *frontal* oder *seitlich zu unterstützen*. Sollte der Stürmer stürzen, reagiert das Strategieszenario eines anderen Roboters und lässt diesen die Rolle übernehmen.

Rolle

Die Szenarien auf der Rollenebene entsprechen den Positionen von menschlichen Fußballspielern in ihrer Mannschaft. Jede Rolle übernimmt bestimmte Aufgaben im Team.

Zum Beispiel ist es die Aufgabe des Stürmers zum Ball zu laufen und diesen in das gegnerische Tor zu dribbeln oder zu schießen, während die Unterstützer in der Nähe des Balls bleiben, um gegebenenfalls zu helfen.

Ein Rollenszenario entspricht dabei einem Zustandsautomaten, der je nach Zustand ein Szenario auf der *Verhaltensebene* aktiviert. Der Automat wechselt seinen Zustand abhängig davon, ob die von ihm aktivierten Szenarien mit oder ohne Erfolg terminieren.

Verhalten

Die Verhaltensebene beinhaltet den größten Umfang an Szenarien. In diesen werden verschiedene *Bewegungen* mit Hilfe der Sensordaten zu einfachen Verhalten zusammengefügt.

Ein Beispiel für ein Verhaltensszenario ist das *Schießen*, welches auch die Positionierung für den Schuss beinhaltet. Dabei entscheidet sich der Roboter zunächst für einen Fuß, mit dem er den Ball schießen wird. Anschließend bewegt er sich so, dass der Ball vor dem entsprechenden Fuß liegt. Dazu wird das *Bewegungsszenario Laufen* mit den passenden Parametern aktiviert. Sobald der Roboter die richtige Position erreicht hat, aktiviert er die Bewegung *Schießen* (siehe Abbildung 3). Nachdem der Schuss ausgeführt wurde, gibt die Bewegungsebene eine Rückmeldung an das Verhalten, welches sich dann beendet. Dies wird wiederum der Rollenebene mitgeteilt, die nun ihrerseits ein anderes Verhalten startet.

Bewegung

Die unterste Ebene der Verhaltenssteuerung ist die *Bewegung*. Nur Szenarien dieser Ebene können die Aktoren direkt ansteuern. Sie bilden die Grundbausteine für alle komplexeren Verhalten, wobei jedes Szenario einer Bewegung entspricht. Unterschieden wird zwischen den statischen Bewegungen *Stehen*, *Aufstehen* und *Schießen* sowie der dynamischen Bewegung *Laufen*. Die statischen Bewegungen sind als Folgen von Motorwerten realisiert, die eine Bewegung ergeben, wenn sie abgespielt werden. Das Laufen hingegen ist das Resultat einer komplizierten Regelung.

2.3 Probleme

Obwohl die Verhaltenssteuerung 2009 ihren Zweck erfüllte, wurden bei ihrer Entwicklung einige Probleme deutlich. Zum einen lassen sich mit der verwendete CSBP-Architektur nicht beliebig komplexe Verhalten übersichtlich implementieren, ohne die Anzahl der Ebenen zu erhöhen. Dies ist allerdings aufwendig und entspricht nicht den Anforderungen, schnell große Änderungen im Verhalten vornehmen zu können.

Zum anderen bildet die starre Architektur die Wirklichkeit nicht besonders gut ab. So passt beispielsweise die Überprüfung, ob der Roboter gestürzt ist, nirgendwo separat in die Architektur und muss daher von jedem Verhaltensszenario einzeln durchgeführt werden. Dadurch wird eine stärkere Modularisierung komplexer Verhalten unterbunden und Redundanz im Quellcode erzeugt. Deshalb wird die Verhaltenssteuerung mit zunehmender Größe immer schwerer zu verstehen, zu warten und zu erweitern.

Außerdem unterstützt diese Architektur parallele Verhalten nur, wenn diese auf verschiedenen Ebenen ausgeführt werden. Wenn zum Beispiel ein Verhalten existiert, das durch Bewegen der Kamera die Umgebung nach dem Ball absucht und ein anderes, das den Roboter zu einer bestimmten Position laufen lässt, dann können diese beiden Szenarien der Verhaltensebene nicht gleichzeitig aktiviert werden.

3 Systeme zur Verhaltenssteuerung

3.1 Anforderungen

Durch den Umstieg auf ein anderes Verhaltenssteuerungssystem sollen diese Probleme vermieden werden. Dazu sollte ein System ausgewählt werden, das folgende Anforderungen erfüllt:

Verhalten müssen sehr gut modularisierbar sein, damit die einzelnen Teilverhalten überschaubar bleiben und möglichst wenig Redundanz im Quellcode vorhanden ist. Außerdem sollen kontinuierliche Verhalten unterstützt werden, Verhaltensmodule also ineinander übergehen können. Weiterhin ist die Möglichkeit zur parallelen Ausführung solcher Module sehr wichtig, um beispielsweise den Roboter durch Bewegung seines Kopfes nach dem Ball suchen zu lassen, während er zurück zum eigenen Tor läuft.

Zudem sollte das neue System die Entwicklung der Verhaltenssteuerung erleichtern. Dies umfasst Syntax und Semantik der verwendeten Programmiersprache aber auch vorhandene Werkzeuge und Entwicklungsumgebungen. Schließlich musste die Integration des Verhaltenssteuerungssystems in die FUmanoid-Software angesichts der bis zum RoboCup 2010 kurzen verbleibenden Zeit zu bewältigten sein.

3.2 Verschiedene Ansätze

3.2.1 Wissensbasierte Systeme

Wissensbasierte Systeme bestehen aus einer *Wissensbasis* und einer *Inferenzmaschine*. Die Wissensbasis umfasst dabei eine Repräsentation der Umwelt in Form von *Fakten* und *Regeln*. Das Verhalten wird hier durch die Definition von Aufgaben impliziert. Die Inferenzmaschine leitet dann bei der Ausführung der Verhaltenssteuerung die passenden Aktionen zu einer gegebenen Aufgabe aus dem Fakten- und Regelwissen ab [21].

Beispiele für diesen Ansatz sind die Programmiersprachen *GOLOG* [11] und *The Planning Domain Definition Language* (PDDL) [7]. Diese sind jedoch nur bedingt geeignet um komplexe Verhaltenssteuerungen für Roboter zu entwickeln, da sie es schwierig machen, mit ungenauen Sensordaten und Unsicherheiten umzugehen (vgl. [18]).

3.2.2 Verhaltensbasierte Systeme

Verhaltensbasierte Systeme setzen das gewünschte Verhalten aus einzelnen Teilverhalten zusammen. Die verschiedenen Ansätze unterscheiden sich dabei durch die Art der Komposition.

Hierarchische Schichten

Eine Möglichkeit der Modularisierung ist die Aufteilung der Verhaltenssteuerung auf verschiedene *Schichten*, wobei die Komplexität der Teilverhalten mit der Höhe der Schicht steigt.

Dieser Ansatz geht auf die von Brooks entwickelte *Subsumption-Architektur* [3] zurück. Das Verhalten des Roboters wird dabei in kleinere Verhaltensmodule aufgeteilt. Diese Module werden Schichten zugeteilt, die unterschiedliche Kompetenzen haben. Die Schichten werden nebenläufig ausgeführt, wobei Module höherer Schichten Module niedrigerer Schichten unterdrücken oder verhindern können.

Ebenfalls von Brooks stammt die *Behavior Language* [4]. Sie basiert auf der Subsumption Architektur und spezifiziert Verhaltensmodule durch Mengen von Regeln. Die einzelnen Module werden nebenläufig ausgeführt und können durch den Austausch von Nachrichten miteinander kommunizieren.

Die im Abschnitt 2.1 beschriebene, 2009 verwendete CSBP-Architektur ist ein weiteres Beispiel für diesen Ansatz, wobei die Begriffe Ebenen und Schichten synonym verwendet werden.

Kombination von Agenten

Eine andere Art die Verhaltenssteuerung aufzuteilen, basiert auf der *Social Agent Theory* [23]. Auch hier wird das Verhalten in einzelne Module aufgeteilt, die in dieser Sprache Agenten heißen. Komplexere Agenten entstehen rekursiv durch die Kombination einfacherer Agenten.

Dieser Ansatz wird in der Programmiersprache *Configuration Description Language* (CDL) [15] umgesetzt. Dabei können Agenten um die Ausführung konkurrieren und der auszuführende Agent nach einer Metrik ausgewählt werden. Weitere mögliche Formen der Kombination von Agenten sind die Kooperation, bei der die Resultate verschiedener Agenten zu einem Ergebnis zusammengefasst werden, und die sequenzielle Ausführung einzelner Agenten.

Hierarchische endliche Zustandsautomaten

Eine dritte Möglichkeit der Dekomposition der Verhaltenssteuerung ist die Verwendung hierarchischer endlicher Zustandsautomaten [9]. Dabei werden Zustandsautomaten in einer Hierarchie angeordnet und können einander aktivieren, deaktivieren und Informationen austauschen. Es können immer mehrere Automaten gleichzeitig aktiv sein, die parallel ausgeführt werden.

Eine Programmiersprache, die diesen Ansatz verfolgt ist *COLBERT* [10]. Automaten werden darin in einer C ähnlichen Syntax durch die Angabe von Prozeduren definiert.

Urbi [2] ist eine Softwareplattform zur Entwicklung von Verhaltenssteuerungen, in der ebenfalls implizit hierarchische Zustandsautomaten definiert werden. Dazu wird eine objektorientierte, ereignisgesteuerte und parallele Skriptsprache namens *UrbiScript* verwendet. Verhalten

werden darin spezifiziert, indem Funktionen und Ereignisse definiert und einander zugeordnet werden. Abhängig von eintretenden Ereignissen werden die entsprechenden Funktionen nebenläufig ausgeführt. Eine Gruppierung der Funktionen ist durch die Verwendung von Marken möglich, wodurch Funktionen einander deaktivieren können.

The Extensible Agent Behavior Specification Language (XABSL) [13, 12] ist eine weitere Sprache, mit der sich Verhalten von Robotern auf Basis von hierarchischen endlichen Automaten beschreiben lassen. Im Gegensatz zu COLBERT und UrbiScript, werden Zustandsautomaten in XABSL jedoch explizit durch die Angabe der Zustände, ihrer Aktionen sowie der Zustandsübergänge definiert.

3.3 Auswahl eines Systems

Max Risler vergleicht in seiner Doktorarbeit [18] XABSL mit vielen älteren Ansätzen, wie der Behavior Language, COLBERT und CDL. Er stellt dabei die Vorteile von XABSL dar, die vor allem in der sehr guten Modularisierbarkeit von komplexen Verhalten und in der Einfachheit der Sprache liegen. Außerdem wird XABSL von einigen RoboCup-Teams, wie dem *German Team* [5] und den *Darmstadt Dribblers* [6] eingesetzt. Beide Teams wurden mehrfach Weltmeister in ihrer Liga, was deutlich macht, dass XABSL für den Einsatz im RoboCup gut geeignet ist. Außerdem existieren sehr hilfreiche Werkzeuge für diese Sprache. So lässt sich aus dem Quellcode beispielsweise automatisch eine Dokumentation der Verhaltenssteuerung mit grafischer Darstellung der Zustandsautomaten generieren.

UrbiScript verfolgt einen sehr interessanten Ansatz, dessen Fokus vor allem auf Parallelität liegt. Die Sprache ist dabei deutlich komplexer als XABSL, wodurch dem Entwickler mehr Möglichkeiten gegeben werden. Fraglich ist dennoch die Skalierbarkeit auf sehr komplexe Verhalten. Durch die ständige Beachtung der Nebenläufigkeit aller Funktionen könnte die Entwicklung kompliziert und fehleranfällig sein. Um dies zu vereinfachen, bietet die Firma *Gostai*, von der Urbi entwickelt wird, die Entwicklungsumgebung *Gostai Studio* an. Darin können komplexe Verhalten durch die Spezifikation hierarchischer endlicher Automaten mit einer grafischen Benutzeroberfläche erstellt werden. Gostai Studio ist anders als die Sprache UrbiScript jedoch keine freie Software.

Nach intensiver Auseinandersetzung mit beiden Systemen fiel die Entscheidung auf XABSL. Ausschlaggebend dafür war die Unsicherheit bezüglich Urbi und dessen proprietärer Entwicklungsumgebung und der schwer abzuschätzende Aufwand für die Einbindung dieses deutlich größeren Systems in die FUmanoid-Software. Dahingegen sind die Werkzeuge für XABSL frei und die Sprache hat sich beim RoboCup bereits bewährt. Außerdem kooperieren die FUmanoids mit dem *Nao Team Humboldt* von der Humboldt-Universität Berlin, welches ebenfalls XABSL verwendet und Hilfe bei eventuellen Schwierigkeiten anbot.

3.4 XABSL

3.4.1 Konzepte und Begriffe

Verhaltensmodule heißen in XABSL *Optionen*. Sie werden in den folgenden Abbildungen als Rechtecke dargestellt. Optionen sind in einem kreisfreien, gerichteten Graphen, dem *Optionsgraphen*, angeordnet. Eine Kante von einer Option A zu einer Option B bedeutet dabei, dass Option A Option B aktivieren kann. Durch Auswahl einer Option aus dem Graphen als Wurzel entsteht der Optionsbaum. Die Wurzel des Baumes wird als *Agent* bezeichnet.

Bei der Auswertung wird vom Agenten ausgehend der *Aktivierungsbaum* erstellt, der alle in dieser Ausführung aktiven Optionen enthält. Hierzu werden einige der ausgehenden Kanten der aktuell betrachteten Option verfolgt und die Optionen, zu denen sie führen, aktiviert. Mit diesen wird analog fortgefahren, bis es keine entsprechenden Kanten mehr gibt.

Jede Option ist selbst ein *Zustandsautomat* mit einer endlichen Menge von *Zuständen*. Zustände sind in den nachfolgenden Abbildungen als Kreise dargestellt, wobei Endzustände doppelt umrandet und Startzustände durch zwei waagerechte Linien markiert sind.

Jeder Zustand wird durch zwei Teile definiert. Der erste ist die *Aktion*, die in diesem Zustand ausgeführt wird. In ihr können andere Optionen aktiviert und auf *Symbole* zugegriffen werden. Der zweite Teil ist die *Entscheidung*. Sie definiert die Übergänge zu anderen Zuständen mit Hilfe eines *Entscheidungsbaums*.

Die Schnittstelle zur übrigen auf dem Roboter ausgeführten Software wird durch Symbole bereitgestellt. Diese können auf Variablen oder Funktionen verweisen. Durch sie erhält die Verhaltenssteuerung Informationen über die Umwelt. Auch die Bewegungen des Roboters werden durch Manipulation der Symbole kontrolliert.

3.4.2 Beispiel XABSL-Code

In Abbildung 4 ist der Optionsbaum für ein simples Beispiel dargestellt. Er entspricht der Datei `agents.xabsl` (siehe Listing 1), in der Symbol- und Optionsdateien eingebunden und Agenten definiert werden.

```
1  include "Symbols/Symbols.xabsl";
2  include "Options/A.xabsl";
3  include "Options/C.xabsl";
4  include "Options/D.xabsl";
5  include "Options/B.xabsl";
6  include "Options/E.xabsl";
7  agent agentA("A", A);
```

Listing 1: `agents.xabsl`

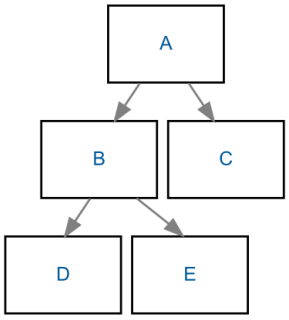


Abbildung 4: Optionsbaum

Die eingebundene Symboldatei `Symbols.xabsl` definiert in diesem Beispiel zwei Eingabe- und zwei Ausgabesymbole (siehe Listing 2).

```
1  namespace Symbols("") {
2    float input input1;
3    bool input input2;
4    float output output1;
5    float output output2;
6  }
```

Listing 2: `Symbols.xabsl`

In Abbildung 5 sind im Optionsbaum zusätzlich die Zustandsautomaten in den Optionen A und B zu sehen. In C, D und E gibt es ebenfalls Zustandsautomaten, die hier aber nicht betrachtet werden. Option A hat nur einen Zustand, in dem immer B und C aktiviert werden (siehe Listing 3). In Option B wird je nach Zustand eine der Optionen D oder E aktiviert (siehe Listing 4).

```
option A {
  initial state a1 {
    action {B(); C();}
  }
}
```

Listing 3: `A.xabsl`

```
option B {
  initial state b1 {
    decision {...}
    action {D();}
  }
  state b2 {
    decision {...}
    action {E();}
  }
}
```

Listing 4: `B.xabsl`

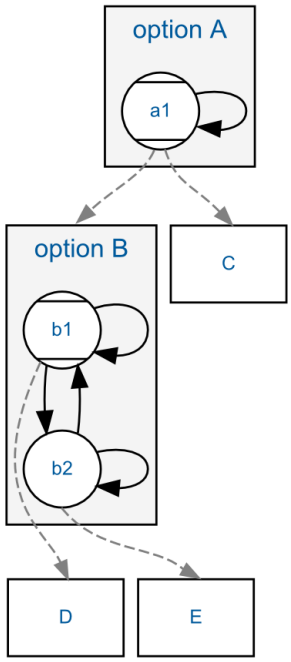


Abbildung 5: Optionsbaum

Dass außer der Aktivierung anderer Optionen auch Symbole manipuliert werden können, ist in Option C zu sehen (siehe Listing 5). In der Aktion von `c1` wird das Ausgabesymbol `output2` auf den Wert 100 gesetzt.

```
option C {
  initial state c1 {
    action {
      output2 = 100;
    }
  }
}
```

Listing 5: `C.xabsl`

Option B besitzt zwei Zustände und jeweils Übergänge zwischen diesen. Abhängig davon in welchem Zustand die Option ist, wird der entsprechende Entscheidungsbaum (siehe Abbildungen 6 und 7) ausgewertet und in den resultierenden Zustand gegangen. Die Entscheidungen hängen von den Symbolen `input1` und `input2` (siehe Listing 6 und 7) ab.

```
option B {
  initial state b1 {
    decision {
      /** input1 > 0? */
      if (input1() > 0)
        goto b2;
      else
        stay;
    }
    action {
      D();
    }
  }
  ...
}
```

Listing 6: Zustand b1 in `B.xabsl`

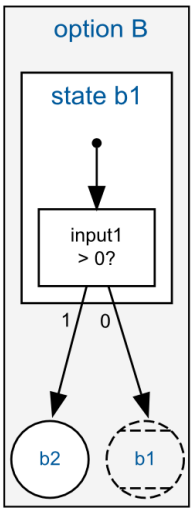


Abbildung 6: Entscheidungsbaum für b1

```
option B {
  ...
  state b2 {
    decision {
      /** input2 = true? */
      if (input2())
        goto b1;
      else
        stay;
    }
    action {
      E();
    }
  }
}
```

Listing 7: Zustand b2 in `B.xabsl`

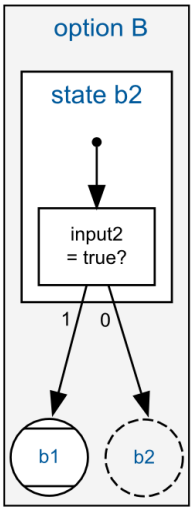
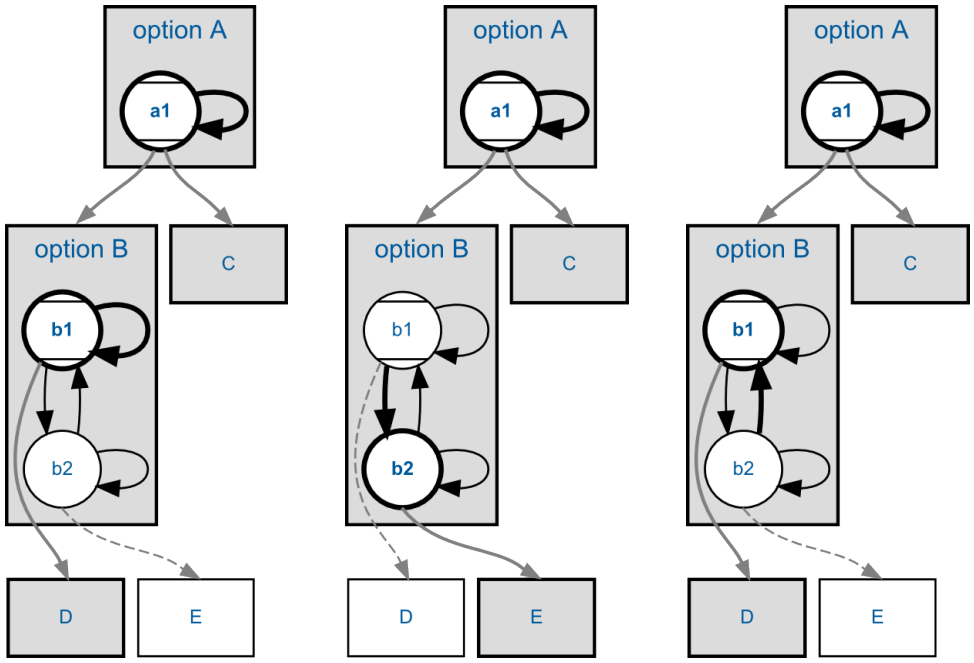


Abbildung 7: Entscheidungsbaum für b2

3.4.3 Ausführung des Beispiels



(a) (b) (c)

Abbildung 8: Aktivierung im XABSL-Beispiel (aktive Optionen sind grau ausgefüllt, aktuelle Übergänge und Zustände sind fett gedruckt)

Das Beispiel wird das erste Mal ausgeführt (siehe Abbildung 8a), indem mit dem Startzustand des Agenten A begonnen wird. Von diesem Zustand wird der einzige Übergang verfolgt, welcher wiederum auf sich selbst zeigt. Danach wird die Aktion von a1 ausgeführt, wodurch die Optionen B und C aktiviert werden. Die Option B ist zunächst ebenfalls in ihrem initialen Zustand b1. Sei `input1 <= 0`, dann behält auch B seinen Zustand bei, welcher die Option D aktiviert. Danach folgen die Auswertungen von D und C, die hier nicht weiter beschrieben werden.

Bei jeder weiteren Ausführung ist das Vorgehen analog, bis auf die Tatsache, dass die Optionen ihren letzten Zustand beibehalten. Nur wenn eine Option in einem Durchlauf nicht aktiviert wird, kehrt sie zum Startzustand zurück.

In der zweiten Ausführung (siehe Abbildung 8b) geschieht zunächst das gleiche wie in der ersten: A bleibt im Zustand a1 und aktiviert B und C. Sei jetzt `input1 > 0`, dann geht die Option B vom Zustand b1 in den Zustand b2. Durch die Ausführung der Aktion von b2 wird E aktiviert. Schließlich werden die Optionen E und dann C ausgewertet.

Auch die dritte Ausführung des Beispiels (siehe Abbildung 8c) beginnt in A, dessen Zustand a1 bleibt, wodurch die Optionen B und C aktiviert werden. Die Option B ist wegen des letzten Durchlaufes zunächst im Zustand b2. Darum wird der Entscheidungsbaum von b2 für den Zustandsübergang ausgewertet. Wenn `input2 == true` ist, wechselt B zurück in den Zustand b1. In dessen Aktion wird D aktiviert. Zuletzt werden D und C ausgewertet.

3.4.4 Einbindung von XABSL in die FUmanoid-Software

Um XABSL auf den FUmanoid-Robotern zu verwenden, musste die *XABSL-Engine* in die übrige Software integriert werden. Dazu wurde die entsprechende C-Bibliothek eingebunden. Als nächstes mussten die Schnittstelle in der `symbols.xabsl` spezifiziert und alle benötigten Variablen und Funktionen der FUmanoid-Software bei der XABSL-Engine registriert werden.

Nun lässt sich Verhaltenssteuerung ausführen, indem der XABSL-Code mit dem *XABSL-Compiler* in einen Zwischencode übersetzt und auf den Roboter geladen wird. Die FUmanoid-Software lädt diesen Code in die XABSL-Engine, welche daraus den Optionsbaum aufbaut. Aus der iterativen Auswertung dieses Baumes entsteht das Verhalten des Roboters.

4 Verhaltenssteuerung 2010

4.1 Verwendete Grundlagen der FUmanoid-Software

4.1.1 Statische und dynamische Bewegungen

Diese Arbeit baut im Bereich Bewegungen auf den bestehenden Strukturen der untersten Ebene der Verhaltenssteuerung von 2009 auf. Funktionen zum Starten der verschiedenen Bewegungen existieren bereits im FUmanoid-Programm und müssen in der Verhaltenssteuerung nur aufgerufen werden. Dabei wird zwischen statischen und dynamischen Bewegungen unterschieden.

Statische Bewegungen entsprechen einer Folge von Motorpositionen, die nacheinander innerhalb einer bestimmten Zeit abgespielt werden. Diese Bewegungen werden aufgenommen, indem die Motorwerte gelesen werden, nachdem der Roboter manuell in die gewünschte Position gebracht wurde. Durch das Abspielen mehrerer solcher Posen entsteht eine flüssige Bewegung. Die Bewegungen Stehen, Aufstehen, Schießen und zwei Formen der Torwartabwehr sind mit statischen Bewegungen bereits realisiert.

In diesen Bewegungen findet jedoch keine Anpassung auf Grund von äußeren Bedingungen statt. Damit ist keine aktive Stabilisierung oder Parametrisierung möglich. Komplexere Bewegungen, die dies realisieren, heißen *dynamische Bewegungen*. Sie entsprechen einem Algorithmus, der sich zeitlich ändernde Motorpositionen aus verschiedenen Parametern berechnet.

4.1.2 Das Laufen

Die einzige in dieser Arbeit verwendete dynamische Bewegung ist ein omnidirektionales Laufen, welches von Hamid Moballegh entwickelt wurde [17]. Diese Bewegung erlaubt dem Roboter sich gleichzeitig vorwärts bzw. rückwärts und seitwärts zu bewegen, sowie sich um *seine* eigene Achse zu drehen.

Des Weiteren wird auch eine *Hindernisvermeidung* zur Verfügung gestellt. Solange diese von der Verhaltenssteuerung aktiviert wird, verändert sie die Laufrichtung des Roboters, sodass dieser automatisch einen Bogen um vorhandene Hindernisse macht.

4.1.3 Das Weltmodell

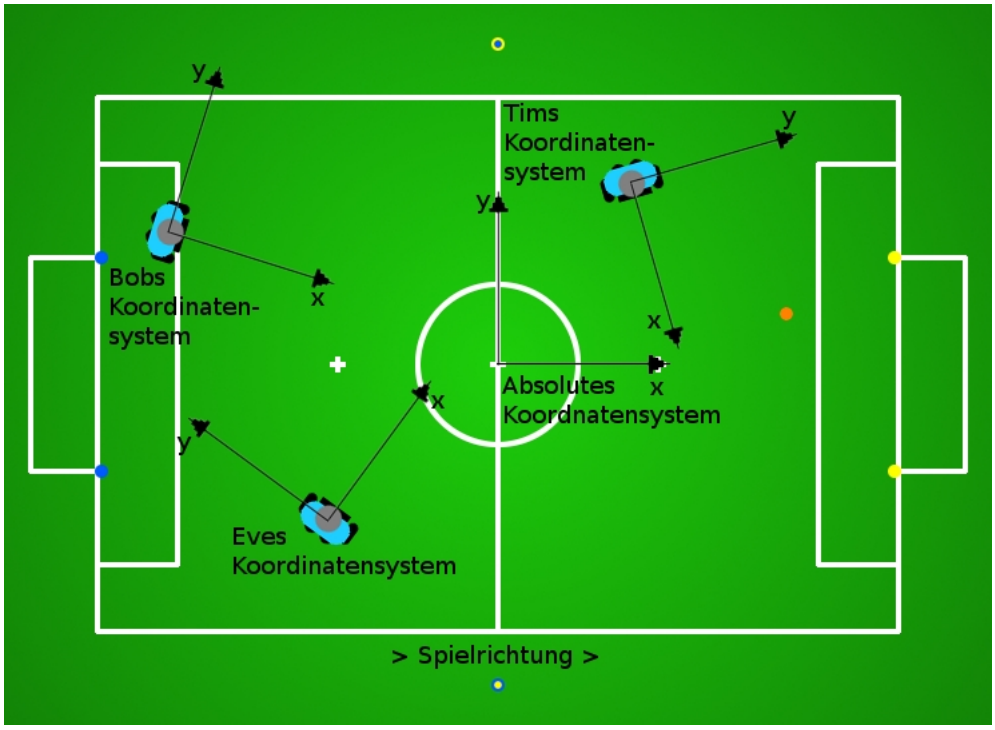


Abbildung 9: Relatives und absolutes Koordinatensystem

Wie bereits in der Einleitung erläutert, hat die Verhaltenssteuerung keinen direkten Zugriff auf die Sensorik, sondern erhält die Daten bereits aufbereitet in einem Weltmodell. Die meisten Informationen darin beziehen sich auf Positionen von Objekten auf dem Spielfeld. Die wichtigsten Fragen, welche durch das Weltmodell beantwortet werden müssen, sind: Wo befindet sich der Ball, wo das Tor? Wie ist meine eigene Position und Orientierung? Wo sind meine Mit- und Gegenspieler? Die Antworten auf diese Fragen können in relativen oder absoluten Koordinaten gegeben werden (siehe Abbildung 9).

Jeder Roboter ist der Ursprung seines eigenen *relativen Koordinatensystems*. Die x-Achse dessen zeigt dabei in Richtung der Orientierung des Roboters. Die y-Achse zeigt vom Roboter aus nach links. Da alle Bewegungen des Roboters abhängig von seiner jeweiligen Ausgangsposition sind, sind es relative Bewegungen. Um auf direktem Weg zum Ball zu laufen, sind deshalb ausschließlich die relative Koordinaten des Balls interessant. Um den Ball ins Tor zu schießen, reicht es aus, zusätzlich die relative Position des Tores zu wissen. Mehr Informationen werden für ein sehr rudimentäres Fußballverhalten nicht benötigt.

Für komplexere und kooperative Verhalten ist jedoch ein *absolutes Koordinatensystem* notwendig. Dieses wurde so definiert, dass es seinen Ursprung im Zentrum des Spielfeldes hat und die x-Achse zum gegnerischen Tor zeigt, während die y-Achse um 90 Grad gegen den Uhrzeigersinn gedreht ist. Absolute Koordinaten geben die Position eines Objektes auf dem Spielfeld an. Insbesondere die eigene Position und Orientierung ist dabei von Interesse. Mit Hilfe dieser Informationen lassen sich Punkte aus einem Koordinatensystem in das jeweils andere transformieren.

Wenn ein Roboter also weiß, wo er sich auf dem Feld befindet, kann er einen absoluten Zielpunkt in relative Koordinaten umwandeln und somit jede beliebige Position auf dem Spielfeld ansteuern. Außerdem kann er die relativen Koordinaten von Objekten wie dem Ball in absolute Koordinaten transformieren und die gewonnenen Informationen seinen Mitspielern mitteilen. Diese können dann, wenn sie ebenfalls gut lokalisiert sind, die entsprechende Rücktransformation durchführen und beispielsweise zu einem Ball laufen, den sie selbst nicht sehen.

4.2 Andere Ansätze als 2009

In dieser Arbeit wurden die Bewegungen aus der alten Verhaltenssteuerung übernommen und einige Ideen, die sich bewährt haben, wieder implementiert. Dennoch beinhaltet sie weit mehr als die Implementierung der alten Verhaltenssteuerung in einer neuen Sprache. Die Verhaltenssteuerung wurde von Grund auf neu entwickelt, wobei andere Ansätze als im Vorjahr verfolgt wurden.

4.2.1 Immer zum Tor ausrichten

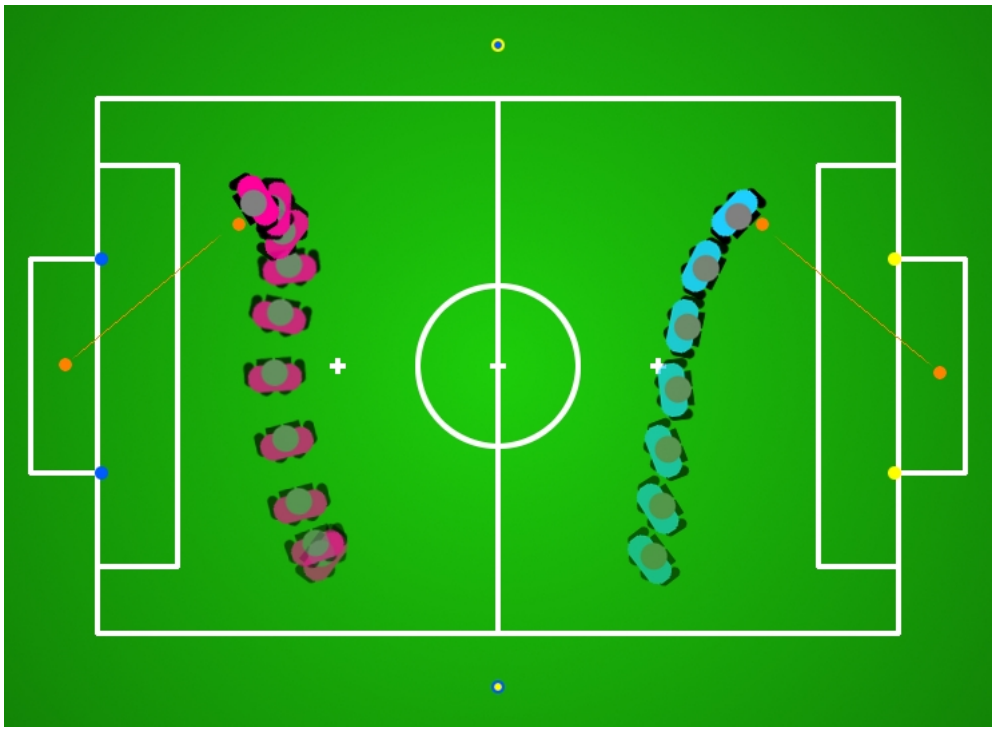


Abbildung 10: Gegenüberstellung: Orientierung in Laufrichtung (magenta), Orientierung zum Tor (cyan)

Wie im Abschnitt 4.1.2 erläutert, ist der auf den Robotern implementierte Laufalgorithmus sehr vielseitig und ermöglicht ein omnidirektionales Laufen. Die maximale Laufgeschwindigkeit nach vorn und zur Seite ist dabei ähnlich hoch. Dadurch kann sich ein Roboter gleichzeitig in eine Richtung drehen und in eine andere laufen. Dies wird genutzt, um eine sehr einfache, aber effektive Idee umzusetzen: Die Vorderseite des Roboters soll immer in Richtung des gegnerischen Tores zeigen. Dazu wird die Drehung des Roboters so geregelt, dass dieser immer darauf ausgerichtet bleibt. Mit diesem Vorgehen ändert der Roboters seine Richtung nur sehr wenig, während er vorwärts und seitwärts läuft. Die Vermeidung schneller Drehungen bringt Geschwindigkeits- und Stabilitätsvorteile beim Laufen.

Außerdem kann sich der Roboter durch gleichzeitiges Laufen zum Ball und Drehen zum Tor schnell in eine gute Schussposition bringen. Weil er immer zum Tor gedreht ist, kommt er automatisch in eine solche Position, sobald er an den Ball gelangt. Damit entfällt eine komplizierte Positionierung hinter dem Ball, wie sie viele andere Teams implementiert haben (siehe Abbildung 10).

4.2.2 Raumdeckung

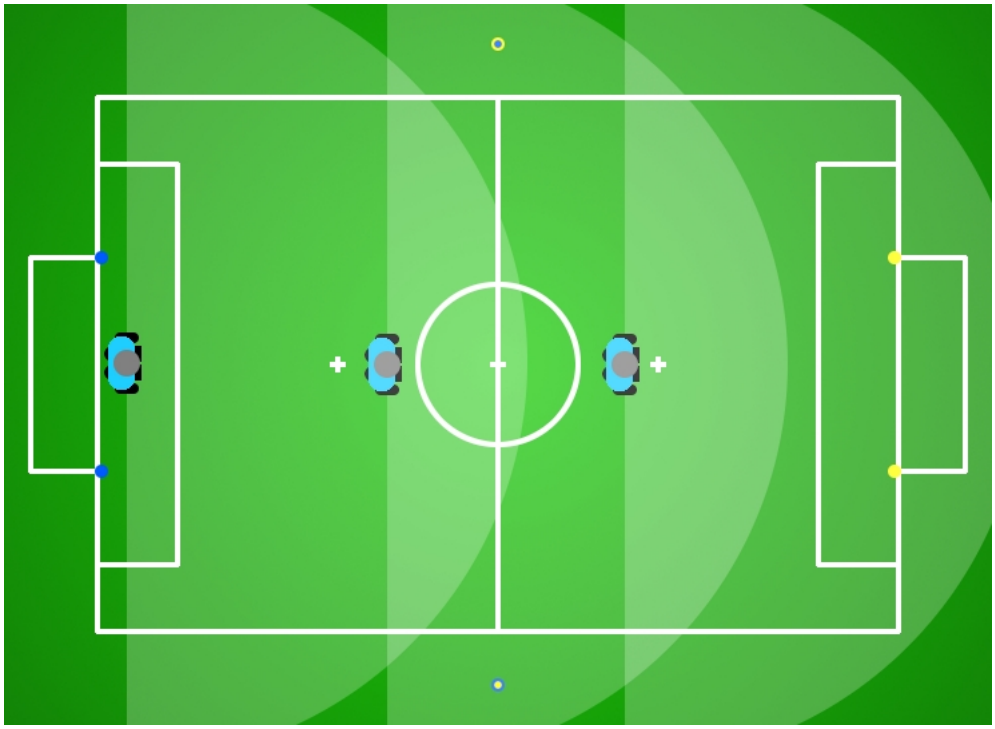


Abbildung 11: Die Roboter auf ihren Positionen: Verteidiger/Torwart, Unterstützer, Stürmer. Im Bild ist ein drei Meter weiter Sichtkegel hervorgehoben. In diesem Bereich ist es den Robotern möglich den Ball zu erkennen.

In der bisherigen Verhaltenssteuerung der FUmanoids positionierten sich meist alle Roboter relativ zum Ball. Abgesehen vom ballführenden Roboter bewegten sich auch andere Roboter in die Nähe des Balls, um den stürmenden Spieler zu unterstützen oder bei Ballverlust für ihn einzuspringen (siehe Abschnitt 2.2). Dieses Vorgehen hat den Nachteil, dass sich alle Roboter in einem kleinen Bereich des Spielfeldes aufhalten. Wenn sie den Ball verlieren, ist es schwierig diesen wiederzufinden. Insbesondere wenn die Roboter von ihren Gegenspielern überlaufen werden, kann ein gegnerischer Roboter leicht zum Torschuss kommen.

Dass dennoch diese Strategie implementiert wurde, lag zum Teil daran, dass ein solches Verhalten keiner sehr zuverlässigen Berechnung der eigenen Position auf dem Spielfeld bedurfte. Durch die Verbesserung der Selbstlokalisierung wurde der Verhaltenssteuerung mehr Spielraum für die Umsetzung neuer Ideen eingeräumt.

In der neuen Verhaltenssteuerung werden die Roboter, die im Moment nicht am Ball sind, anders eingesetzt. Sie sollen dabei eine der über das Spielfeld verteilten Positionen einnehmen (siehe Abbildung 11), damit der Ball in jeder Situation möglichst schnell gefunden werden kann, die Roboter nicht überlaufen werden und sie sich nicht im Weg stehen. In der Abbildung

wird deutlich, wie effektiv die Abdeckung des gesamten Feldes funktioniert. Der Ball wird fast auf dem gesamten Feld von mindestens einem Roboter gesehen, solange alle auf ihren Positionen sind.

4.2.3 Zentrale Strategieentscheidung

Für die Zuweisung dieser Positionen und damit nicht alle Roboter gleichzeitig zum Ball laufen und sich behindern, ist eine Absprache der Roboter untereinander nötig. Dazu müssen zunächst alle relevanten Informationen, wie die Entfernung zum Ball und die eigene Position auf dem Feld, kommuniziert und dann eine Entscheidung getroffen werden.

Der erste Teil wird realisiert, indem jeder Roboter regelmäßig Statuspakete an seine Mitspieler sendet. Diese speichern die aktuellsten Informationen im Weltmodell, wo sie von der Verhaltenssteuerung ausgewertet werden.

Bisher folgte dann eine verteilte Entscheidungsfindung, in der jeder Roboter seine eigene Rolle im Team bestimmte und damit entschied, ob er selbst zum Ball gehen oder nur unterstützen sollte. Dies funktioniert aber nur dann problemlos, wenn alle Roboter ihre Entscheidungen auf Basis der selben Daten fällen. Tatsächlich ist ein Roboter aber nicht im Besitz der aktuellen Informationen seiner Mitspieler, sondern nur der des zuletzt empfangenen Statuspakets. Das W-LAN ist außerdem bei Wettkämpfen häufig überlastet, weshalb Pakete bei der Übertragung verloren gehen können. Im Ergebnis entstehen Missverständnisse zwischen den Robotern, sodass beispielsweise mehrere Roboter gleichzeitig die Stürmerrolle übernehmen, zum Ball laufen und sich dabei gegenseitig zu Fall bringen.

Zur Vermeidung dieser Effekte sollen in der neuen Version alle Entscheidungen von einer Instanz, dem Kapitän, gefällt werden. Damit wird garantiert, dass jede Rollenverteilung zumindest in sich selbst widerspruchsfrei ist. Weiterhin muss das Verfahren aber ermöglichen, dass bei Ausfall des Kapitäns schnell ein anderer dessen Aufgabe übernimmt.

Die sehr einfache Umsetzung, die hier verwendet wird, funktioniert so, dass jeder Roboter eine eindeutige Nummer erhält, die seinen Rang unter den Kapitänen bestimmt. Alle Roboter sind demnach Kapitäne, erstellen immer eine Rollenverteilung für das gesamte Team und schicken diese ihren Mitspielern als Teil des Statuspakets. Jeder Roboter besitzt somit die Entscheidungen aller Mitspieler, gehorcht aber nur der Entscheidung des Spielers, der in der Rangfolge am höchsten steht. Fällt der Kapitän aus, dann wird seine Entscheidung nach kurzer Zeit als veraltet bewertet. Alle Roboter hören dann auf den zweiten Kapitän. Dadurch ist dieses System sehr robust gegenüber Roboterausfällen.

4.3 Überblick

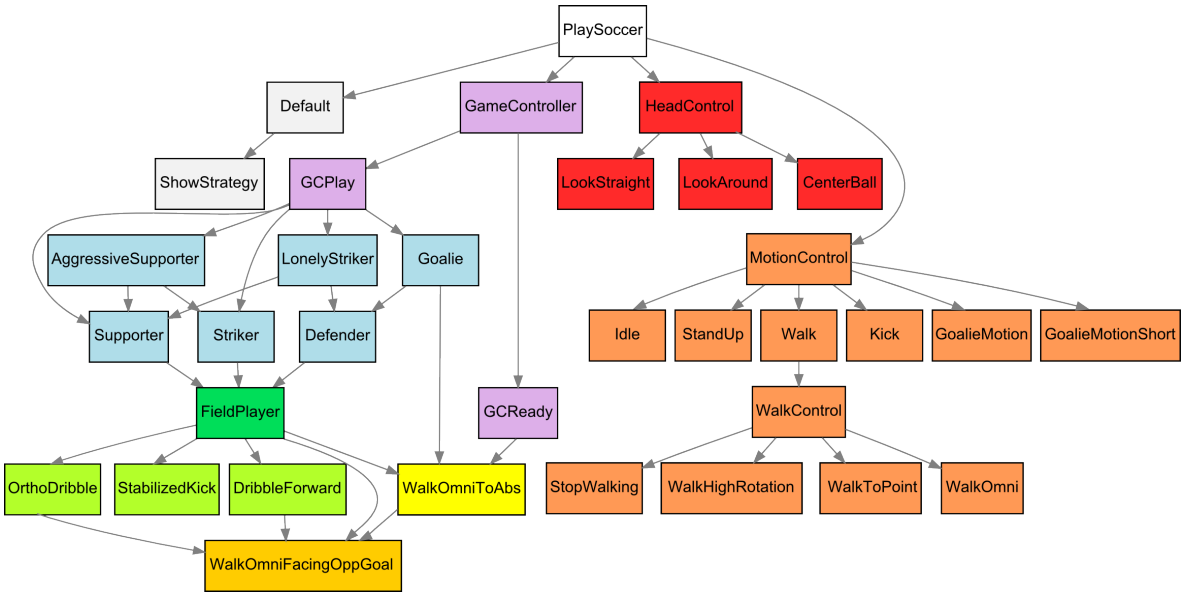


Abbildung 12: Optionsbaum der Verhaltenssteuerung

In Abbildung 12 ist der Optionsbaum der Verhaltenssteuerung mit allen verwendeten Optionen dargestellt. In der Struktur ist erkennbar, wie komplexe Verhalten, weiter oben im Baum, durch die Kombination einfacherer Verhalten, unten im Baum, entstehen. Im Folgenden wird der Baum von einfachen zu komplexen Verhalten durchlaufen, und im Detail beschrieben. Zunächst wird hier aber ein kurzer Überblick über die Aufgaben der verschiedenen Gruppen von Optionen gegeben. Dazu sind diese farblich einheitlich zusammengefasst.

Die Hilfsmodule (grau) weisen Symbolen Standardwerte zu und generieren Ausgaben für die Entwickler. Die Steuerung des Kopfes (rot) und die Bewegungssteuerung (orange) bilden die Grundfähigkeiten des Roboters. Sie laufen parallel zum restlichen Verhalten und werden durch Manipulation interner Symbole gesteuert. Ein weiterer Grundbaustein der Verhaltenssteuerung ist das omnidirektionale Laufen zu einer Position in relativen Koordinaten (gelborange). Aufbauend darauf ist das Laufen zu einer Position auf dem Feld in absoluten Koordinaten implementiert (gelb). Die nächste Gruppe (hellgrün) definiert verschiedene Fußballfähigkeiten, wie das Dribbeln und Schießen. Diese werden vom Herzstück der Verhaltenssteuerung, dem Feldspieler-Verhalten (dunkelgrün), verbunden. Verschiedene Rollen (hellblau) parametrisieren und erweitern den Feldspieler. Schließlich startet die Spielsteuerung (lila) das Verhalten entsprechend des Spielzustandes und entscheidet darüber, welche Rolle gespielt wird. Die Wurzel (weiß) ist der Startpunkt der Verhaltenssteuerung.

4.4 Detaillierte Beschreibung

4.4.1 Kopfsteuerung

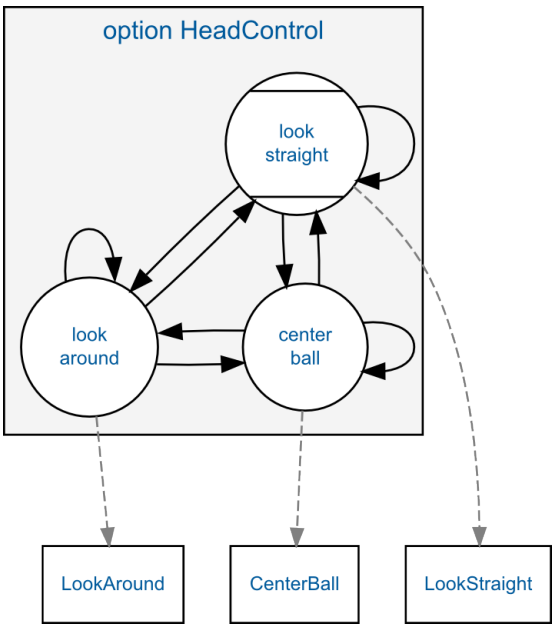


Abbildung 13: Kopfsteuerung

Die Steuerung des Kopfes müsste naturgemäß unten in der Hierarchie der Verhaltenssteuerung zu finden sein, da sie ein sehr konkretes Verhalten beschreibt, auf dem andere Verhalten aufbauen. Tatsächlich ist dies aber nicht der Fall (siehe Abbildung 12 (rot)).

Stattdessen läuft sie parallel zu den komplexeren Verhalten und wird von ihnen durch Schreiben eines gemeinsamen Symbols `head.state` beeinflusst. Dieses kann die drei Werte `LookStraight`, `LookAround` und `CenterBall` annehmen, die jeweils einem Zustand der Option `HeadControl` (siehe Abbildung 13) entsprechen. Die Kopfsteuerung passt ihren Zustand dem Wert des Symbols an und aktiviert dabei die entsprechende Option, welche die gewünschte Kopfbewegung schließlich ausführt.

Die Verwendung dieser Architektur vermeidet ein Problem, das auftritt, wenn die Kopfsteuerung direkt von den Verhaltensmodulen aktiviert und parametrisiert wird. Weil in XABSL die Mehrfachaktivierung einer Option nicht erlaubt ist, müsste in diesem Fall sichergestellt werden, dass es bei keiner Konfiguration des Baumes dazu kommt, dass die Kopfsteuerung von mehreren Verhaltensmodulen gleichzeitig aktiviert wird.

Mit dem hier gewählten Ansatz tritt dieses Problem nicht auf. Dem Symbol können in einem Durchlauf des Baumes nacheinander verschiedene Werte zugewiesen werden. Die Kopfsteuerung liest nur den zuletzt geschriebenen Wert und bestimmt daraus, welche Bewegung tat-

sächlich ausgeführt wird. Dies wird unter anderem genutzt, um dem Symbol `head.state` in der Option `Default` (siehe Abbildung 12 (grau)) den Wert `LookStraight` zuzuweisen.

Dadurch schaut der Roboter standardmäßig geradeaus, was dem Verhalten der gleichnamigen Option `LookStraight` entspricht. Darin wird eine C-Funktion aufgerufen, die den Kopf dazu veranlasst, sich nach vorn zu drehen. Der Kopf behält diese Position, solange die Option aktiv bleibt.

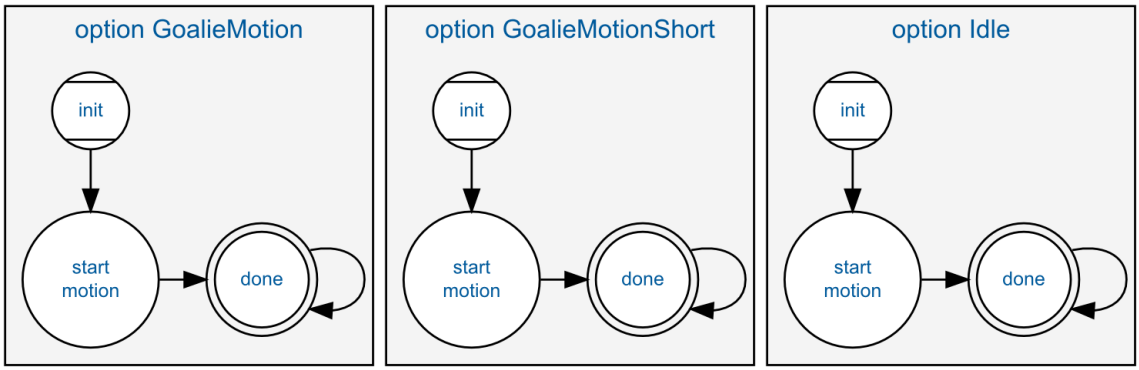
Das Umsehen, wie es in der Option `LookAround` implementiert wurde, ist eine sequenzielle Ausführung mehrerer Kopfbewegungen, die dazu dienen soll, den Ball zu finden, wenn dieser außerhalb des eigenen Sichtfeldes liegt. Dazu wird der Kopf zunächst nach links und dann nach rechts gedreht. Während der Bewegungen kann eine andere Option diesen Vorgang durch Schreiben von `head.state` unterbrechen, wenn zum Beispiel der Ball gefunden wurde.

Der Kopf kann den Bewegungen des Balls auch automatisch folgen, solange sich dieser in seinem Sichtfeld befindet. Dazu muss die Option `CenterBall` aktiviert werden. Diese versucht den Ball in jedem Durchlauf durch eine entsprechende Drehung des Kopfes zu zentrieren.

4.4.2 Bewegungssteuerung

Die Bewegungssteuerung (siehe Abbildung 12 (orange)) läuft wie die Kopfsteuerung parallel zu den anderen Verhaltensmodulen. Auch hier findet die Kommunikation durch Nutzung eines gemeinsamen Symbols `motion.state` statt. Die Bewegungssteuerung überprüft jedoch in jedem Durchlauf zunächst, ob der Roboter hingefallen ist. Ist dies der Fall, dann wechselt sie in den Zustand `stand_up` und aktiviert die Option `StandUp`, um sich wieder aufzurichten. Sonst liest sie das Symbol `motion.state`. Entspricht dieses einem der Werte `GoalieMotion`, `GoalieMotionShort`, `Kick` oder `Idle`, dann gibt es einen Übergang in den entsprechenden Zustand, was zur Aktivierung der gleichnamigen Option führt, welche schließlich die statische Bewegung ausführt. Wenn `motion.state` dagegen den Wert `StopWalking`, `WalkHighRotation`, `WalkToPoint` oder `WalkOmni` enthält, wird im Zustand `walk` die Option `Walk` aktiviert, wodurch wiederum die Laufsteuerung aktiv wird.

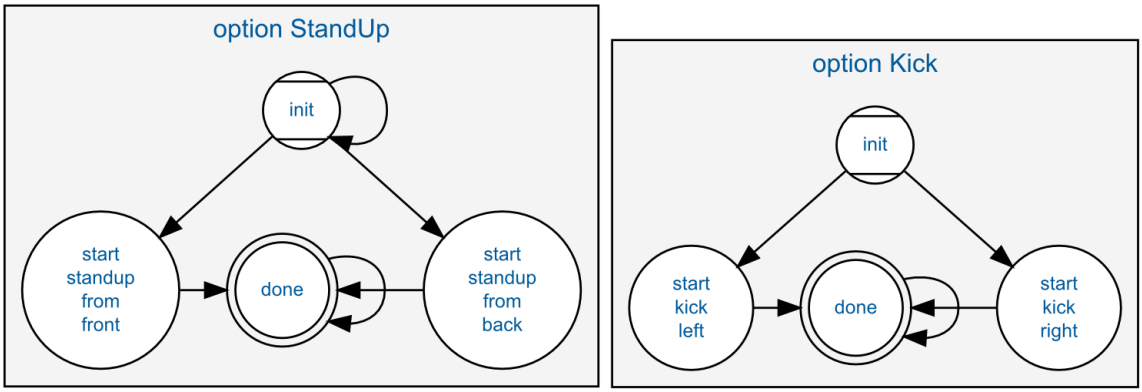
Ausführung statischer Bewegungen



(a) (b) (c)

Abbildung 14: Ausführung einfacher statischer Bewegungen

Das Ausführen statischer Bewegungen folgt immer dem gleichen Schema (siehe Abbildung 14a-14c.) Es gibt einen Initialzustand `init` mit nur einem Übergang zu dem Zustand `start_motion`, in dem die Bewegung gestartet wird und einen Endzustand `done`, der verhindert, dass dieselbe Bewegung mehrfach nacheinander ausgeführt wird.



(a) (b)

Abbildung 15: Ausführung statischer Bewegungen mit zwei Varianten

Die Ausführung der Bewegungen Aufstehen und Schießen funktioniert ähnlich. Es existieren jedoch jeweils zwei Varianten der Bewegung, zwischen denen sich die Bewegungssteuerung situationsabhängig entscheiden muss. Wenn der Roboter hingefallen ist, prüft sie im Weltmodell, ob er sich in Bauch- oder Rückenlage befindet, und führt die passende Aufstehbewegung aus (siehe 15a). Auch vor der Schussbewegung stellt sie zunächst fest, vor welchem Fuß der Ball liegt um mit dem richtigen Bein auszuholen (siehe 15b).

Laufsteuerung

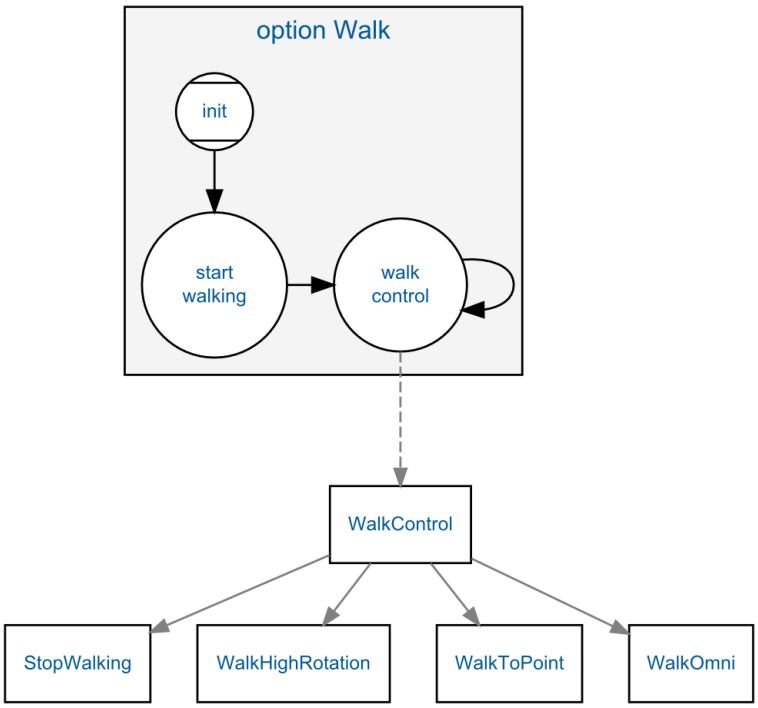


Abbildung 16: Laufsteuerung

Der Laufalgorithmus muss einmal gestartet werden und lässt sich danach mithilfe von Parametern für die Vorwärts- und Seitwärtsgeschwindigkeit, sowie für die Drehung steuern (siehe Abbildung 16).

Die Option `WalkControl` funktioniert analog zur Option `HeadControl`. Für jede implementierte Laufvariante gibt es einen Zustand in der Option `WalkControl`. Dieser Zustand wird entsprechend des Wertes von `motion.state` gewechselt, wodurch die gleichnamige Option aktiviert wird, welche die passende Laufvariante startet. Dabei werden gegebenfalls die internen Symbole `walker.target.x`, `walker.target.y` und `walker.target.rot` ausgewertet, um den Laufalgorithmus entsprechend zu parametrisieren.

Die einfachste der implementierten Varianten ist `StopWalking` (siehe Abbildung 17c), welche die entsprechende Funktion für das Stoppen des Laufens aufruft.

Die Option `WalkHighRotation` (siehe Abbildung 17a) lässt den Roboter schnell auf der Stelle um seine eigene Achse drehen.

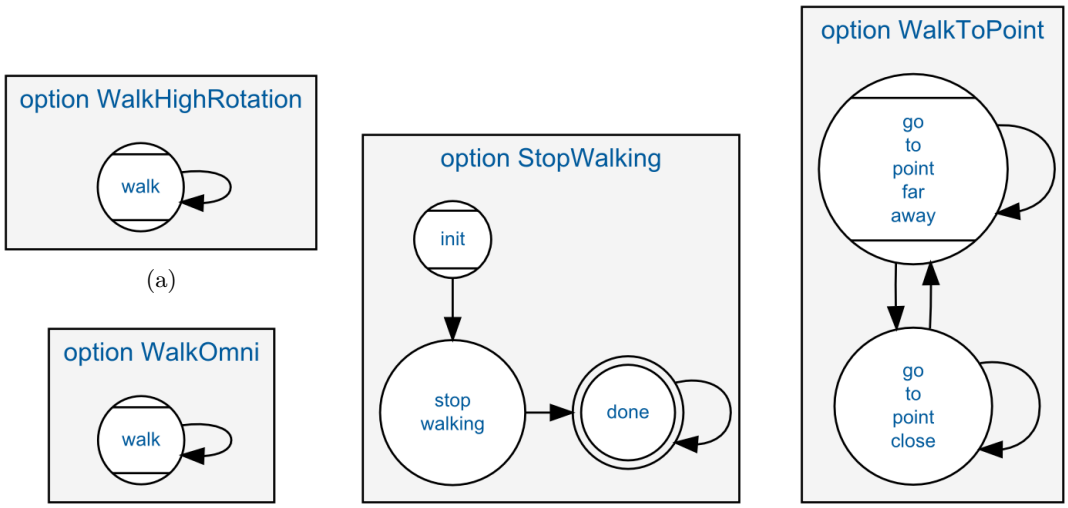


Abbildung 17: Laufsteuerung

Die Option `WalkToPoint` (siehe Abbildung 17d) ermöglicht es dem Roboter zu einem Punkt in relativen Koordinaten zu laufen und sich dabei in eine bestimmte Richtung zu drehen. Das Verhalten entscheidet sich in Abhängigkeit der Entfernung zum Zielpunkt für eine der beiden folgenden Vorgehensweisen:

Wenn die Entfernung zum Ziel groß ist, wird die gewünschte Drehrichtung zunächst vernachlässigt. Der Roboter richtet sich stattdessen zum Ziel aus und läuft vorwärts auf dieses zu (siehe Listing 8).

```
1  forward = walker.target.x/2;
2  sideward = 0;
3  rotation = walker.target.y/2;
4  walker.setSpeed(forward, sideward, rotation);
```

Listing 8: `WalkToPoint` bei großen Entfernungen

Wird sein Abstand zum Zielpunkt kleiner als 60 Zentimeter, dann dreht er sich in die vorgegebene Richtung und läuft gleichzeitig den Rest der Strecke zum Ziel (siehe Listing 9).

```
1  forward = walker.target.x/2;
2  sideward = walker.target.y/2;
3  rotation = walker.target.rot/2;
4  walker.setSpeed(forward, sideward, rotation);
```

Listing 9: `WalkToPoint` bei geringen Entfernungen

Währenddessen läuft der Roboter nicht mehr nur geradeaus sondern auch seitwärts auf sein Ziel zu. Dieses omnidirektionale Laufen mit gleichzeitiger Drehung entspricht der Laufform `WalkOmni` (siehe Abbildung 17b). Tatsächlich ist dies nur eine Vereinfachung der letzten Laufform. Die Option `WalkOmni` verhält sich immer so wie `WalkToPoint`, wenn der Zielpunkt nahe ist.

Omnidirektionales Laufen mit Blick zum Tor

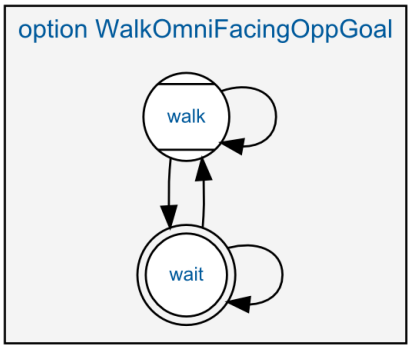


Abbildung 18: Omnidirektionales Laufen mit Blick zum Tor

In der Option `WalkOmniFacingOppGoal` (siehe Abbildung 12 (gelborange)) wird der Ansatz umgesetzt, dass der Roboter immer zum Tor sehen soll (siehe Abschnitt 4.2.1).

Diese Option erfüllt aber noch weitere Aufgaben. Zum einen soll der Roboter stehen bleiben, wenn er seinem Ziel nahe genug ist. Deshalb besitzt die Option den Parameter `eps` (Epsilon) für die Toleranz bezüglich des Zielpunkts. Wenn der Abstand in Zentimeter zum Zielpunkt kleiner als Epsilon ist, dann geht die Option in den Zustand `wait` (siehe Abbildung 18). In diesem Zustand tut sie nichts, geht aber wieder zurück in den aktiven Zustand `walk`, wenn der Abstand zum Ziel größer als zwei Epsilon ist.

Weil sich die Roboter in ihrer Mechanik und Konfiguration nicht komplett gleichen, sind manche Roboter mehr in eine Richtung geneigt als andere. Diese Neigung hat insbesondere bei niedrigen Geschwindigkeiten einen starken Einfluss auf das Laufverhalten. In diesen Fällen kann es vorkommen, dass die Roboter auf der Stelle stehen bleiben oder sich sogar in die falsche Richtung bewegen. Diese Situationen soll der Roboter bemerken und stärker in die gewünschte Richtung steuern, um den Fehler zu beheben.

Um dies zu ermöglichen, wurde ein Proportional-Integral-Regler eingesetzt. Bei größeren Entfernungen ist nur der Proportionalteil aktiv. Bei geringen Entfernungen kommt ein Integralanteil hinzu. Dieser summiert den Fehler zum Ziel über die Zeit. Solange der Roboter den Zielpunkt also nicht erreicht, wird das Integral stetig größer. Dadurch wird nach kurzer Zeit die Geschwindigkeit in die gewünschte Richtung erhöht und damit der Fehler behoben. Um ein

Übersteuern zu vermeiden, ist der Integralteil außerdem auf einen Maximalwert begrenzt und wird zurückgesetzt, sobald das Integral und der momentane Fehler gegensätzliche Vorzeichen haben.

Die Ergebnisse dieser Option sind die durch den Regler korrigierten Werte für den Zielpunkt und die benötigte Drehung, um sich zum Tor auszurichten. Diese werden in den internen Symbolen `target.x`, `target.y` und `target.rot` gespeichert, um die Laufsteuerung zu parametrisieren. Als Laufform wird meist `WalkOmni` verwendet, außer wenn die gewünschte Drehung sehr groß ist, der Roboter also komplett in die falsche Richtung schaut, dann wird stattdessen `WalkHighRotation` verwendet, um die richtige Orientierung möglichst schnell wieder herzustellen.

Zu einem beliebigen Punkt auf dem Feld laufen

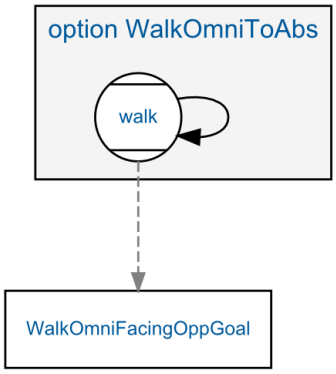


Abbildung 19: Zu einem beliebigen Punkt auf dem Feld laufen

Die Option `WalkOmniToAbs` (siehe Abbildung 12 (gelb)) benutzt die soeben beschriebene Steuerung, um den Roboter zu einem beliebigen Punkt auf dem Spielfeld laufen zu lassen. Dazu projiziert sie den Zielpunkt in absoluten Spielfeldkoordinaten in das relative Koordinatensystem des Roboters und aktiviert mit diesen Werten die Option `WalkOmniFacingOppGoal` (siehe Abbildung 19).

4.4.3 Fußballfähigkeiten

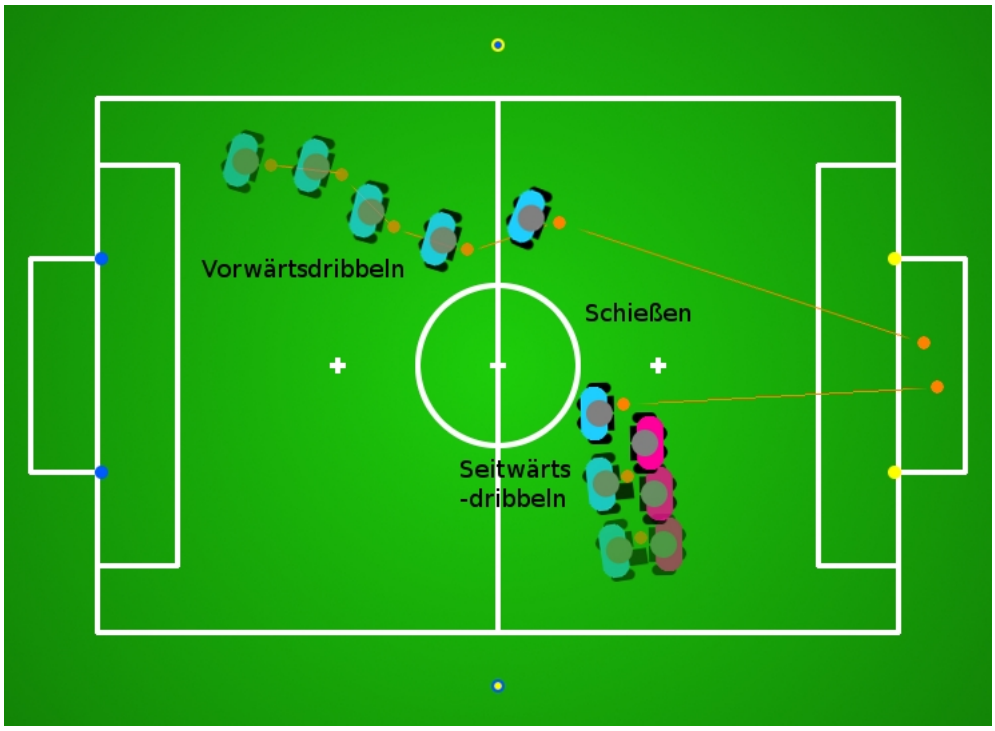


Abbildung 20: Fußballfähigkeiten in der Ausführung: Vorwärtsdribbeln, Seitwärtsdribbeln und Schießen

Aufbauend auf den bereits beschriebenen Bewegungen lassen sich die für ein Fußballspiel nötigen Fähigkeiten (siehe Abbildung 12 (hellgrün)) erstellen. Diese Fähigkeiten sind in Abbildung 20 dargestellt. Ihre Umsetzung wird im Folgenden erläutert.

Vorwärtsdribbeln

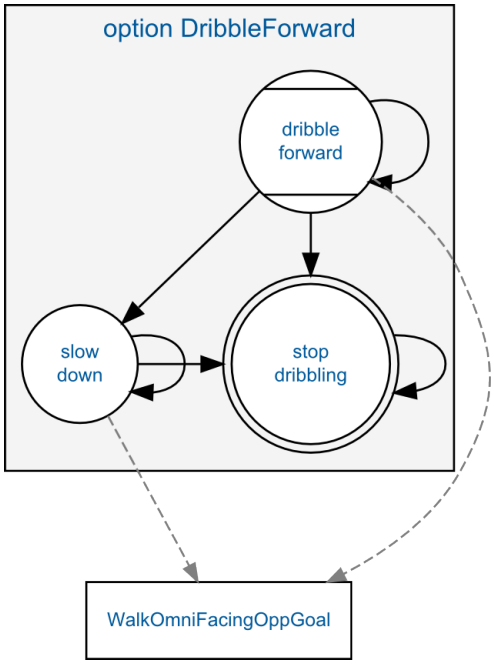


Abbildung 21: Vorwärtsdribbeln

Das Dribbeln ist in der Option `DribbleForward` (siehe Abbildung 21) implementiert. Solange der Startzustand `dribble_forward` aktiv ist, peilt der Roboter einen Punkt hinter dem Ball als Zielpunkt an und bewegt sich in dessen Richtung. Dabei läuft der Roboter gegen den Ball und stößt ihn vorwärts. Durch seine Ausrichtung auf das gegnerische Tor wird der Ball automatisch in diese Richtung befördert. Rollt der Ball währenddessen weiter als 30 Zentimeter vom Roboter weg, so wird das Dribbeln beendet, um danach einen Torschuss zu versuchen. Dazu wird zunächst der Zustand `slow_down` aktiviert. In diesem Zustand reduziert der Roboter seine Geschwindigkeit zwei Sekunden lang. Diese Verlangsamung ist nötig, um sich danach mit moderater Geschwindigkeit in eine Schussposition zu bringen. Ohne ein solches Abbremsen würde der Roboter bei der erneuten Annäherung an den Ball zu schnell sein und diesen noch einmal wegstoßen. Nach kurzer Zeit gibt es einen Übergang zum Endzustand `stop_dribbling`. Dieser markiert die Beendigung dieses Verhaltens und wird auch dann aktiviert, wenn der Ball in irgendeinem der Zustände verloren geht.

Seitwärtsdribbeln

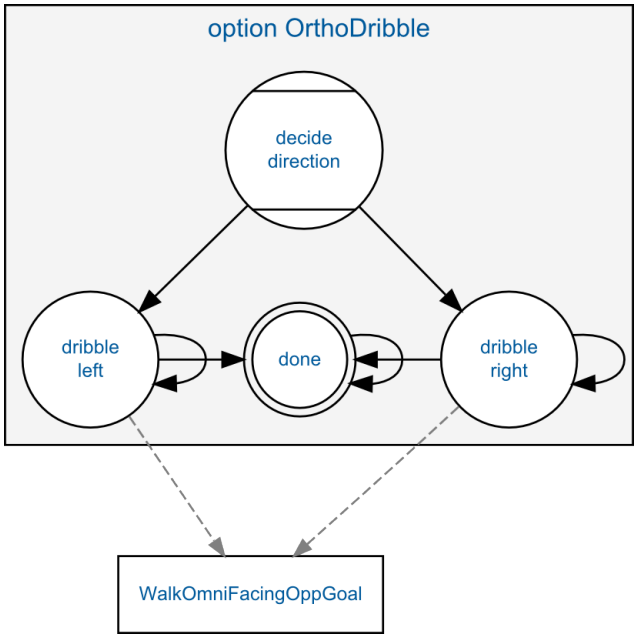


Abbildung 22: Seitwärtsdribbeln

Mit dem Verhalten `OrthoDribble` (siehe Abbildung 22) sollen Zweikämpfe, in denen sich der Ball zwischen einem eigenen und einem gegnerischen Roboter befindet, gewonnen werden. Zunächst wird im Zustand `decide_direction` die Richtung festgelegt: Befindet sich der Ball vor dem linken Fuß, dann wird nach links gedribbelt (Zustand `dribble_left)`, sonst nach rechts (Zustand `dribble_right`). Dabei wird der Fuß, vor dem sich der Ball nicht befindet, nach vorn gezogen und der Roboter läuft in die entsprechende Richtung, wobei er den Ball mit der Fußinnenseite des vorgezogenen Fußes mitführt. Wenn der Ball bei diesem Vorgang verloren geht oder das Ziel erreicht und der Gegner ausgespielt ist, dann wird der Endzustand `done` aktiviert und damit das Dribbeln beendet.

Schießen

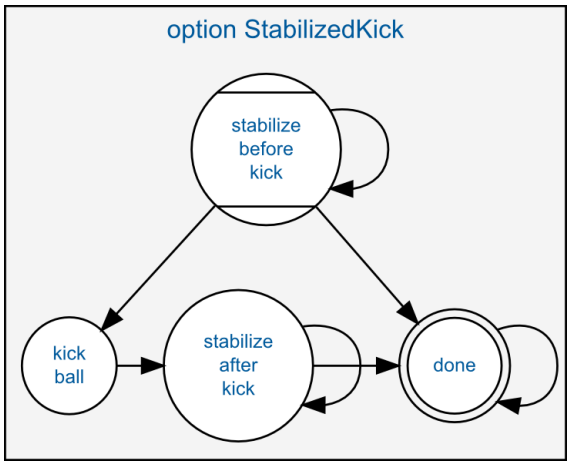


Abbildung 23: Schießen

Die Option `StabilizedKick` (siehe Abbildung 23) kann direkt aus dem Laufen heraus aktiviert werden, um einen geraden Schuss auszuführen. Dabei wird im Zustand `stabilize_before_kick` zunächst das Laufen angehalten, indem der Wert von `motion.state` auf `StopWalking` gesetzt wird. Um dem Roboter Zeit zu geben auszuschwingen, bleibt dieser Zustand zunächst weiter aktiv. Nach Ablauf einer Sekunde wird geprüft, ob der Ball noch in einer Position liegt, die von einem der beiden Füße erreichbar ist. Wenn dies nicht der Fall ist, weil der Ball etwa versehentlich angestoßen wurde und weggerollt ist, wird mit dem Übergang in den Zustand `done` das Verhalten beendet. Andernfalls führt der Fuß, auf dessen Seite sich der Ball befindet, nach vorheriger Auswahl die Schussbewegung aus.

```
1  action {
2    kick.leftLeg = ball.posRel.y > 0;
3    motion.state = Kick;
4  }
```

Im Anschluss daran gibt es einen Zustandsübergang zu `stabilize_after_kick`. Hier wartet die Option dreieinhalb Sekunden, in denen der Roboter den Schuss ausführt und wieder ruhig zum stehen kommt, um danach erneut loslaufen zu können. Schließlich gibt es einen Übergang zum Endzustand `done`.

4.4.4 Der Feldspieler

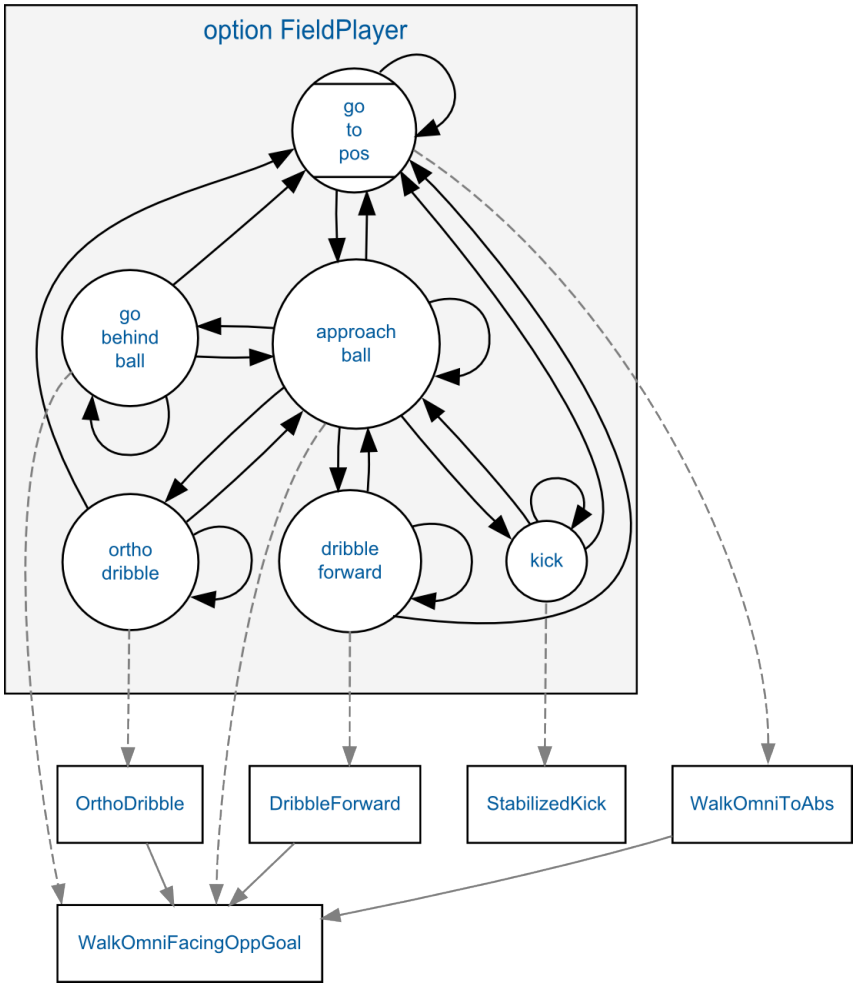


Abbildung 24: Der Feldspieler

Die Option `FieldPlayer` (siehe Abbildung 12 (dunkelgrün)) verknüpft die verschiedenen Fußballfähigkeiten sinnvoll zu einem Verhalten, das den Roboter Fußball spielen lässt.

Die Option beginnt im Zustand `go_to_pos` (siehe Abbildung 24), in dem der Roboter auf die ihm zugeteilte Position geht. Wenn er diese erreicht hat, bleibt er stehen und wartet. Dieser Zustand wird nur dann verlassen, wenn der Roboter weiß, wo sich der Ball befindet und vom Kapitän die Erlaubnis hat, zum Ball zu gehen.

In diesem Fall geht das Verhalten in den Zustand `approach_ball` über und der Roboter läuft auf den Ball zu. Schon dabei entscheidet sich dieser, mit welchem Fuß er zunächst versucht, in Schussposition zu kommen. Der Roboter steuert dann einen Punkt an, so dass der Ball vor dem entsprechenden Fuß liegt, wenn er sein Ziel erreicht. Sobald sich der Ball schließlich günstig vor einem der Füße befindet, wird der Zustand `kick` aktiv, welcher die Option `StabilizedKick`

aktiviert, um den Ball zu schießen. Nach Ausführung dieser Bewegung geht der Roboter wieder auf seine Position oder läuft erneut zum Ball.

Wenn der Ball dabei nicht vor dem Roboter liegt, muss die Zielposition anders gewählt werden, damit der Roboter nicht gegen den Ball stößt, während er versucht sich in Schussposition zu bringen. Dies passiert im Zustand `go_behind_ball`, indem der Roboter rückwärts läuft, bis er den Ball vor sich sieht.

Ist der Roboter länger als 10 Sekunden in der Nähe des Balls, ohne es zu schaffen, diesen zu schießen, dann ändert das Verhalten seinen Zustand in `dribble_forward` und der Roboter beginnt den Ball nach vorn zu dribbeln, um diese Situation aufzulösen. Dabei wird genau solange gedribbelt, bis die Option `DribbleForward` ihren Endzustand erreicht hat. Danach wird abhängig von der Situation ein neuer Versuch unternommen, zum Ball zu laufen, oder auf die eigene Position zurückgekehrt.

Sollte er stattdessen von einem Gegner frontal blockiert werden, wird der Zustand `ortho_dribble` aktiv, in welchem der Roboter versucht seinen Gegenspieler auszuspielen indem er seitwärts dribbelt. Auch dieses Verhalten wird so lange ausgeführt bis die Option `OrthoDribble` sich selbst beendet, indem sie in ihren finalen Zustand geht.

4.4.5 Rollen

Einfache Rollen: Verteidiger, Unterstützer und Stürmer

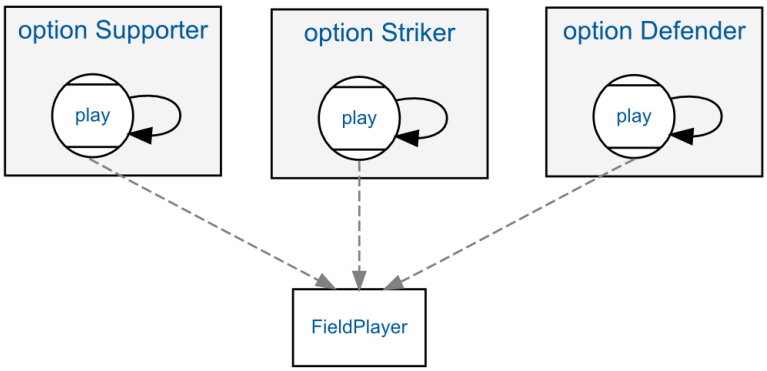


Abbildung 25: Einfache Rollen

Die einfachen Rollen sind eine Umsetzung des Ansatzes der Raumdeckung (siehe Abbildung 11). Sie verhalten sich wie der Feldspieler, laufen jedoch zu verschiedenen Positionen auf dem Feld, wenn sie nicht zum Ball gehen. Bei der Aktivierung des Feldspielers durch eine Rolle, wird diesem die entsprechende Position durch die Parameter `pos.x` und `pos.y` zugewiesen. Die Position des Verteidigers (`Defender`) befindet sich im eigenen Strafraum, die des Unterstützers (`Supporter`) vor dem Mittelkreis und die des Stürmers (`Striker`) hinter dem Mittelkreis.

Kombinierte Rollen: Aggressiver Unterstützer und einsamer Stürmer

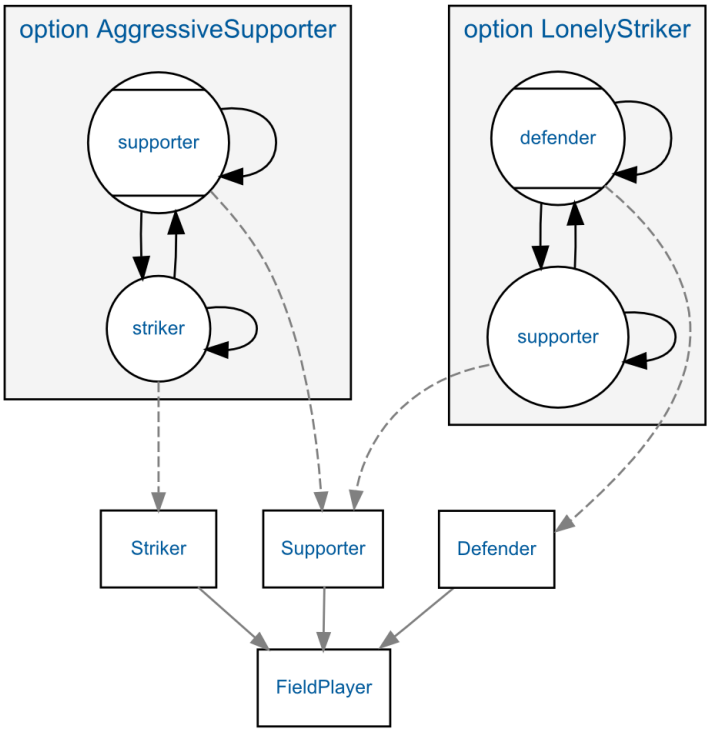


Abbildung 26: Kombinierte Rollen

Um die drei Positionen (siehe Abbildung 11) möglichst auch dann zu besetzen, wenn sich nur ein oder zwei Roboter auf dem Spielfeld befinden, wurden neue Rollen aus den bereits bestehenden kombiniert. In diesen wechseln die Roboter im 30-Sekunden-Takt ihre Rolle im Team. Wenn sie dabei nicht am Ball sind, ändern sie somit auch ihre Position auf dem Feld. Der Aggressive Unterstützer (`AggressiveSupporter`) wechselt zwischen Unterstützer- und Stürmerposition, während der einsame Stürmer (`LonelyStriker`) zwischen Verteidiger- und Unterstützerposition pendelt.

Erweiterte Rolle: Torwart

Der Torwart ist eine Erweiterung des Verteidigers, der sich ähnlich einem Eishockeytorwart zusätzlich in einen Spagat fallen lässt um den Ball abzuwehren. Diese Bewegung gibt es in zwei Varianten:

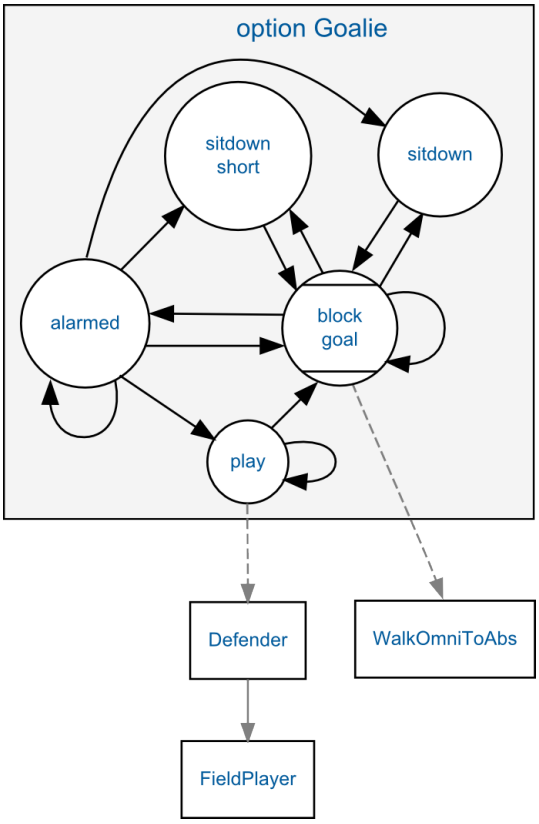


Abbildung 27: Torwart

Die Variante `GoalieMotionShort` wird im Zustand `sitdown_short` verwendet wenn ein Torschuss erkannt wird. Sie lässt den Roboter schnell wieder aufstehen, um auf weitere Angriffe reagieren zu können.

In der zweiten Variante `GoalieMotion` bleibt der Roboter dagegen so lange sitzen, wie dies nach den Regeln zulässig ist. Sie wird auf Verdacht im Zustand `sitdown` verwendet, der aktiv wird, wenn der Torwart einen Gegner mit Ball vor seinem Tor entdeckt.

Die Option `Goalie` (siehe Abbildung 27) startet zunächst im Zustand `block_goal`, in dem der Roboter zur Verteidigerposition im Strafraum läuft und dort stehen bleibt. Wenn währenddessen der Ball gesehen wird, geht die Option in den Zustand `alarmed` und der Roboter hält an seiner momentanen Position an, um besser auf einen Torschuss reagieren zu können. Wird ein solcher erkannt oder erwartet, dann führt er, wie bereits beschrieben, die passende Torwartabwehr aus. Ist dies nicht der Fall und hat der Torwart die Erlaubnis zum Ball zu gehen, so wird im Zustand `play` das Verteidigerverhalten aktiv. Der Roboter verhält sich in diesem Fall wie ein normaler Feldspieler, geht zum Ball und versucht diesen wegzuschießen.

4.4.6 Rollenverteilung und Teamkoordination

Die Ansätze des Teamkapitäns und der Raumdeckung (siehe Abschnitt 4.2) wurden in C++ umgesetzt, da die Schnittstelle zu den versendeten Netzwerkpaketen auf diese Art einfacher zu realisieren war.

Die Reihenfolge der Kapitäne ergibt sich aus der ID der Roboter. Alle fällen die folgenden Entscheidungen für das Team und teilen diese den anderen Robotern mit. Dabei werden den Robotern ihre Positionen zugewiesen und entschieden, welcher Roboter an den Ball gehen soll.

Die Zuteilung der Rollen mit ihren Positionen auf dem Feld ist zunächst von der Anzahl der Spieler abhängig, die sich auf dem Feld befinden. Gibt es nur einen Roboter, so spielt dieser immer die Rolle `LonelyStriker`. Bei zwei Robotern werden die Rollen `Goalie` und `AggressiveSupporter` verteilt, wobei der Roboter, der dem Tor am nächsten ist, Torwart wird. Der andere übernimmt die zweite Rolle. Wenn alle drei Roboter spielen, gibt es die Rollen `Goalie`, `Supporter` und `Striker`. Auch hier wird der dem eigenen Tor nächste Roboter Torwart, der zweitnächste wird Unterstützer und der dritte Stürmer. Damit die Rollen nicht ständig wechseln, falls zwei Roboter ähnlich weit von eigenen Tor entfernt sind, wird eine Hysterese verwendet. Damit erhält der Roboter, dem bisher die entsprechende Rolle zugewiesen wurde, bei jeder Rollenverteilung einen Bonus. Um einen anderen Torwart abzulösen, muss ein Roboter sich also deutlich näher am Tor befinden als dieser.

Damit nicht mehrere Roboter gleichzeitig zum Ball laufen und sich dabei gegenseitig behindern, entscheidet der Kapitän außerdem, welcher Roboter gerade zum Ball gehen darf. Diese Erlaubnis erhält immer nur der Roboter, der den geringsten Abstand zum Ball hat. Auch hier wurde eine Hysterese von 30 Zentimetern eingebaut.

4.4.7 Spielsteuerung

Der Begriff *GameController* bezeichnet ursprünglich eine Software, die während eines RoboCup-Spiels auf einem Computer ausgeführt und vom zweiten Schiedsrichter bedient wird, um den Robotern die Anweisungen des Schiedsrichters mitzuteilen. Die entsprechenden Informationen werden über WLAN an die Roboter gesendet und dort im Weltmodell aufgenommen.

Die Spielsteuerung (siehe Abbildung 12 (lila)) ist in der Option `GameController` implementiert. Sie wertet diese Informationen aus und aktiviert, solange das Spiel läuft, die Option `GCPlay`, welche die durch den Kapitän zugewiesene Rolle startet. Während einer Spielunterbrechung oder vor einem Anstoß wird stattdessen die Option `GCReady` aktiviert. Sie lässt die Roboter entsprechend ihrer Rollen und abhängig davon, welche Mannschaft Anstoß hat, auf ihre Positionen laufen.

5 Fazit und Ausblick

Die hier vorgestellte Verhaltenssteuerung fand ihren Einsatz beim RoboCup 2010 in Singapur. Dort haben die neuen Ansätze sehr gut funktioniert. Die Verwendung des omnidirektionalen Laufens, um immer zum Tor zu sehen, die Rollenverteilung durch den Teamkapitän und insbesondere die Raumdeckung haben sich bewährt und dafür gesorgt, dass das Ergebnis aus 2009 noch einmal verbessert werden konnte. So wurde nicht nur der zweite Platz in der Humanoid Kidsize League verteidigt, sondern zusätzlich der erste Platz in der Technical Challenge gewonnen.

Dies ist besonders beachtlich, da die Verhaltenssteuerung 2010 innerhalb weniger Wochen entwickelt wurde. Noch während des Wettbewerbs entstanden große Teile des Verhaltens, darunter das Vorwärts- und Seitwärtsdribbeln, die Rollenverteilung und der Torwart. Dabei wurde das Potenzial und die Flexibilität von XABSL deutlich.

Im Vergleich mit der CSBP Architektur von 2009 zeigen sich die Vorteile von XABSL. Die Programmiersprache hat eine einfachere, klarere Syntax, wodurch die Entwicklung einer Verhaltenssteuerung erleichtert wird. Die bessere Modularisierbarkeit und Parallelisierbarkeit von Verhalten vermindert zudem die Redundanz in der Implementierung. Diese Einschätzung lässt sich auch mit Fakten belegen (siehe Abbildung 28). In diesem Jahr wurden ein Viertel weniger Zeilen Code geschrieben als 2009. Die Anzahl der Zeichen verringerte sich sogar um zwei Fünftel.

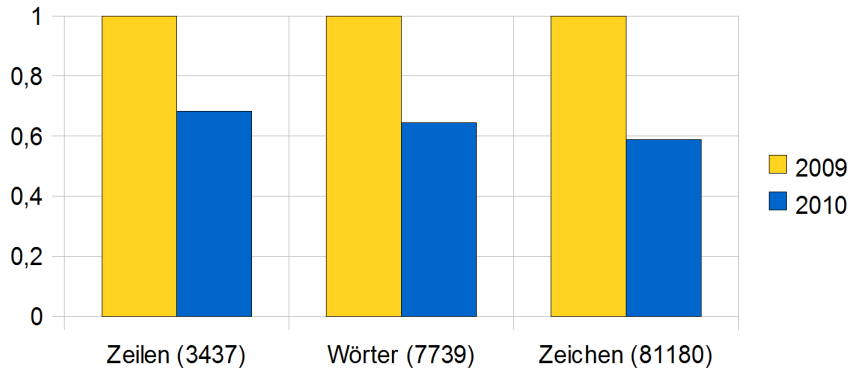


Abbildung 28: Vergleich des Quellcode-Umfangs der Verhaltenssteuerungen 2009 und 2010; normalisiert auf die Werte von 2009 (in Klammern angegeben)

XABSL wird auch im nächsten Jahr weiterhin eingesetzt werden. Aufbauend auf der Verhaltenssteuerung 2010 wird aktuell an komplexeren Strategien gearbeitet. Dabei sollen die Positionen der Roboter nicht mehr statisch sein, sondern sich der Spielsituation anpassen. So könnten Feldspieler die direkte Schusslinie zum Tor versperren oder eigenen Spielern Platz machen. Auch die Verschiebung der gesamten Mannschaft bei Angriff oder Verteidigung ist denkbar. Außerdem ist geplant, das Dribbeln zu verbessern und öfter als bisher einzusetzen, um vor einem Torschuss die Distanz zum Tor zu verringern.

Aber auch XABSL hat noch Potenzial zur Weiterentwicklung. Es enthält beispielsweise überholte und kaum noch verwendete Konzepte, wie *Basisverhalten*, die sich einfach durch Optionen ersetzen lassen, unterstützt keine eigenen Funktionsdefinitionen und besitzt nur einen globalen Namensraum für die Symbole. Auch eine grafische Benutzeroberfläche für die Erstellung der hierarchischen Zustandsautomaten wäre ein sehr hilfreiches Werkzeug, das die Entwicklung mit XABSL noch effektiver machen würde.

Literatur

[1] Aristoteles: *Politik*. – 1253b

[2] Baillie, Jean-Christophe ; Demaille, Akim ; Hocquet, Quentin ; Nottale, Matthieu ; Tardieu, Samuel: The Urbi Universal Platform for Robotics. In: *Workshop Proceedings of SIMPAR 2008*, 2008, S. 580–591

[3] Brooks, Rodney A.: A Robust Layered Control System for a Mobile Robot. In: *IEEE Journal of Robotics and Automation* 2 (1986), Nr. 10

[4] Brooks, Rodney A.: The Behavior Language; User's Guide. (1990)

[5] Dahm, I. ; Düffert, U. ; Hoffmann, J. ; Jüngel, M. ; Kallnik, M. ; Lötzsch, M. ; Risler, M. ; Röfer, T. ; Stelzer, M. ; Ziegle, J.: German Team 2003 / Humboldt-Universität Berlin, Universität Bremen, Technische Universität Darmstadt. 2003. – Team Description Paper. – 4 Seiten

[6] Friedmann, M. ; Hemker, T. ; Kohlbrecher, S. ; Petersen, K. ; Petters, S. ; Radkhah, K. ; Risler, M. ; Scholz, D. ; Thomas, D. ; Stryk, O. von: Darmstadt Dribblers - Team Description for Humanoid KidSize League of RoboCup 2010 / Technische Universität Darmstadt. 2010. – Forschungsbericht

[7] Ghallab, Malik ; Isi, Craig K. ; Penberthy, Scott ; Smith, David E. ; Sun, Ying ; Weld, Daniel: PDDL - The Planning Domain Definition Language / Yale Center for Computational Vision and Control. 1998. – Forschungsbericht

[8] Hohl, Gretta: *Kooperative Planung für autonome humanoide Fußballroboter*, Freie Universität Berlin, Institut für Informatik, Diplomarbeit, 2009

[9] Hopcroft, John E. ; Motwani, Rajeev ; Ullman, Jeffrey D.: *Introduction to Automata Theory, Languages, and Computation.* 3. Addison Wesley, 2006

[10] Konolige, Kurt ; Control, Motion: *COLBERT: A Language for Reactive Control in Sapphira.* 1997

[11] Levesque, Hector J. ; Reiter, Raymond ; Lesperance, Yves ; Lin, Fangzhen ; Scherl, Richard B.: GOLOG: A Logic Programming Language for Dynamic Domains. In: *J. Log. Program.* 31 (1997), Nr. 1-3, S. 59–83

[12] Lötzsch, Martin: *XABSL - A Behavior Engineering System for Autonomous Agents.* Diploma thesis. Humboldt-Universität zu Berlin, 2004

[13] Lötzsch, Martin ; Bach, Joscha ; Burkhard, Hans-Dieter ; Jüngel, Matthias: Designing Agent Behavior with the Extensible Agent Behavior Specification Language XABSL. In: Polani, Daniel (Hrsg.) ; Browning, Brett (Hrsg.) ; Bonarini, Andrea

(Hrsg.): *RoboCup 2003: Robot Soccer World Cup VII* Bd. 3020. Padova, Italy : Springer, 2004 (Lecture Notes in Artificial Intelligence), S. 114–124

[14] Lötzsch, Martin ; Risler, Max ; Jüngel, Matthias: XABSL - A Pragmatic Approach to Behavior Engineering. In: *Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS)*. Beijing, China, 2006, S. 5124–5129

[15] MacKenzie, Douglas C. ; Arkin, Ronald ; Cameron, Jonathan M.: Multiagent Mission Specification and Execution. In: *Autonomous Robots* 4 (1997), S. 29–52

[16] Moballegh, Hamid R.: *Concurrent Scenario-Based Planning, a New Approach to Implement Highly Cooperated Multi-Agent Systems.* – unveröffentlicht

[17] Moballegh, Hamid R. ; Mohajer, Mojgan ; Rojas, Raul: Increasing foot clearance in biped walking: Independence of body vibration amplitude from foot clearance / Freie Universität Berlin, Institut für Informatik. 2008. – Forschungsbericht

[18] Risler, Max: *Behavior Control for Single and Multiple Autonomous Agents Based on Hierarchical Finite State Machines*, Technische Universität Darmstadt, Fortschritt-Berichte VDI, May 15 2009

[19] Risler, Max ; Stryk, Oskar von: Formal Behavior Specification of Multi-Robot Systems Using Hierarchical State Machines in XABSL. In: *AAMAS08-Workshop on Formal Models and Methods for Multi-Robot Systems.* Estoril, Portugal, 2008

[20] RoboCup Humanoid League (Hrsg.): *RoboCup Soccer Humanoid League Rules and Setup.* RoboCup Humanoid League, 2010

[21] Russell, Stuart ; Norvig, Peter: *Artificial Intelligence: A Modern Approach.* 2nd edition. Prentice-Hall, Englewood Cliffs, NJ, 2003

[22] Seifert, Daniel: *Portierung der FUmanoid-Software*, Humboldt-Universität Berlin, Institut für Informatik, Studienarbeit, 2009

[23] Singh, Push: Examining the Society of Mind. In: *Computing and Informatics* 22 (2004), S. 521–543