Reinforcement Learning für Laufroboter

Diplomarbeit zur Erlangung des akademischen Grades
Diplom Informatiker (FH)

von
Markus Schneider

Juli 2007

Betreuender Prüfer:	Prof. Dr. rer. nat. Wolfgang Ertel
Zweitgutachter:	Prof. Dr.-Ing. Holger Voos
Abteilung:	Angewandte Informatik
Tag der Abgabe:	25.07.2007

Eidesstattliche Versicherung

DER AUTOR BESTÄTIGT MIT SEINER UNTERSCHRIFT, DASS ER DIE VORLIEGENDE DIPLOMARBEIT OHNE UNZULÄSSIGE FREMDE HILFE VERFASST UND KEINE ANDEREN ALS DIE ANGEGEBEN QUELLEN UND HILFSMITTEL BENUTZT HAT.

IN ALLEN FÄLLEN IN DENEN EINE EINVERSTÄNDNISERKLÄRUNG DES RECHTSINHABERS BENÖTIGT WURDE, WURDE DIESE EINGEHOLT ODER LAG SCHON VOR. DER AUTOR VERSICHERT DAS MIT DIESER ARBEIT KEINE COPYRIGHT-VERLETZUNGEN BEGANGEN WURDEN.

Unterschrift des Verfassers

Danksagung

Hiermit möchte ich mich bei meinem Betreuer *Professor Dr. Wolfgang Ertel* dafür bedanken, dass er mir diese interessante Diplomarbeit ermöglicht hat und für seine Unterstützung während der gesamten Zeit.

Ich bedanke mich auch bei *Joachim Feßler*, *Achim Feucht*, *Arne Usadel* und *Michel Tokic*, die mir bei Fragen zur Seite standen und von denen ich viel lernen konnte. Dies gilt auch für alle anderen Mitglieder des Robocup-Teams. Es war eine schöne Zeit und es hat sehr viel spaß gemacht im Labor zu arbeiten.

Special thanks goes to *Sonia Seoane Puga* for the last 5 month. This thesis would have never been possible without her great work and her continuing support.

Nicht zuletzt bedanke ich mich auch bei meiner Familie, meiner Freundin und meinen Freunden für ihr Verständnis und ihre Unterstützung währen meiner Arbeit.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Übersicht

Ziel dieser Arbeit ist es einen mehrbeinigen Laufroboter zu entwickeln der lernt sich selbst vorwärts zu bewegen. Der Roboter soll in der Lage sein dieses Verhalten vollkommen autonom und ohne Hilfe eines Lehrers zu erlernen. Dazu sollen zunächst mehrere Algorithmen anhand eines einfacheren Roboters getestet werden. Insbesondere ist dabei zu beachten, dass der Agent zu Beginn keinerlei Wissen über seine Umwelt oder deren Reaktion auf sein Verhalten besitzen soll. Er soll diese Kenntnis lediglich durch Interaktion mit seinem Umfeld erlangen und daraus lernen.

Das maschinelle Lernverfahren zur Steuerung realer Roboter erfolgreich einsetzbar sind zeigte bereits die Diplomarbeit von Michel Tokic (Tokic [2006]). In ihr lernte ein einbeiniger Krabbelroboter mit Hilfe von Wert-Iteration das Laufen. Für die Tests neuer Algorithmen wird ein Roboter mit analogem Aussehen und Funktionalität verwendet.

1.2 Gliederung der Arbeit

Kapitel 2, *"Bioloid Robot Kit"*: Für die Hardware des Roboters wird ein Bausatz der Firma Robotis genutzt. Nach einer kurzen Vorstellung des Roboterbausatzes wird etwas näher auf die hier entwickelte Software zur Steuerung eingegangen. Diese kapselt die komplette Funktionalität der Robotermodule in Klassen und sorgt für eine zuverlässige Kommunikation zwischen Computer und Roboter. In Zusammenarbeit mit Sonia Seoane Puga [Puga, 2007] entstand ein Mikrokontroller, der es ermöglicht beliebige Sensoren in den Bausatz zu integrieren. Dieser ist für die Messung der zurückgelegten Distanz nötig und wird in diesem Kapitel ebenfalls kurz erläutert.

Kapitel 3, *"Lernverfahren"*: Da Tests an einem mehrbeinigen Roboter zu Zeitintensiv sind werden in diesem Kapitel verschiedene Algorithmen aus dem Gebiet des reinforcement learning anhand eines einbeinigen Krabbelroboters getestet. Für diese Tests wurde ein Simulator entwickelt und der Roboter wurde ebenfalls in Hardware gebaut. In den Tests geht es vor allem darum, einen geeigneten Algorithmus zu

finden, der auch für komplexere Roboter beziehungsweise Problemstellungen gut funktioniert. Wichtig hierbei ist vor allem eine geringe Laufzeit, da jedes Mal von Grund auf gelernt wird und es keine seperate Trainingsphase gibt.

Kapitel 4, *"Der vierbeinige Laufroboter"*: Die im vorherigen Kapitel gewonnenen Erkenntnisse werden nun genutzt um zu zeigen, dass reinforcement learning auch für schwierigere Aufgaben geeignet ist. Dazu wird Q-learning mit Planen auf einem mehrbeinigen Laufroboter implementiert.

Kapitel 5, *"Software"*: Die wichtigsten Softwaremodule, die währen dieser Arbeit entwickelt wurden werden in diesem Kapitel kurz vorgestellt. Es handelt sich dabei ausschließlich um Komponenten, die für die Lernalgorithmen von Bedeutung sind. Die Diagramme aller Klassen sind in Anhang F zu finden.

Kapitel 6, *"Zusammenfassung"*: Die Resultate der Arbeit werden kurz widergespiegelt und es werden mögliche Erweiterungen betrachtet.

Kapitel 2

Bioloid Robot Kit

Als Hardwareplattform für die Algorithmen dient der Roboterbausatz Bioloid der Firma Robotis[1]. Mit ihm ist es möglich Roboter in für die verschiedensten Zwecke in einer Vielzahl an Formen zu konstruieren. Dadurch entfallen die meist langen Entwicklungszeiten der Hardware und defekte Teile können mit wenig Aufwand ausgetauscht werden. Ein weiterer entscheidender Vorteil eines solchen Bausatzes ist es, dass Verfahren schnell an verschiedenen Robotern getestet werden können ohne jedes Mal von neuem mit dem Hardwaredesign beginnen zu müssen.

Mit zum Lieferumfang des Bioloid Robot Kit gehört auch eine Software die es erlaubt mittels Drag & Drop sehr schnell und einfach Bewegungsabläufe zu generieren. Da dies für KI Anwendungen nicht sehr von Nutzem ist, war ein wesentlicher Grund weshalb die Wahl auf diesen Bausatz viel, die Möglichkeit die Steuereinheit mittels C zu programmieren.

Der Bausatz besteht aus drei Hauptkomponenten, den Servomotoren (AX-12, Abschnitt 2.2), einem Sensormodul (AX-S, Abschnitt 2.3) und einer Steuereinheit (CM-5, Abschnitt 2.1). Sonia Seoane Puga entwickelte in ihrer Arbeit [Puga, 2007] ein Mikrokontrollerboard, an das weitere Sensoren angeschlossen werden können. Dieses Board wird eingesetzt um die, vom Roboter, zurückgelegte Distanz zu messen. Mehr Informationen zu den Bioloidkomponenten sind in den Handbüchern der Firma Robotis (Rob [2006c], Rob [2006a], Rob [2006b]) zu finden.

2.1 Steuerungseinheit CM-5

Im Inneren des CM-5 befindet sich ein Atmega128[2] der für die Steuerung des Roboters zuständig ist. Im Normalbetrieb werden die Bewegungsabläufe, die mit der Bioloid Software erstellt wurden, auf einen 128 Kilobyte Flash geschrieben. Beim Start wird dieses Programm vom Bootloader geladen und ausgeführt. Das Schreiben des Flashspeichers geschieht mittels der seriellen RS232 Schnittstelle des PCs.

Die Firmware, die sich standardmäßig auf dem CM-5 befindet, kann durch ein selbst geschriebenes C-Programm ersetzt werden. Kleinere, wenig speicherintensi-

[1] http://www.robotis.com
[2] http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf

ve, Algorithmen können so direkt auf dem CM-5 ausgeführt werden. Da jedoch für die meisten Fälle die 4 Kilobyte RAM nicht ausreichend sind müssen die Berechnungen von einem externen Computer ausgeführt werden. Für diesen Zweck wurde in dieser Arbeit ein Programm[3] entwickelt, welches die Steuerung der Motoren und Sensoren direkt über die serielle RS232 erlaubt. Der CM-5 dient von nun an nur noch als "Brücke" zwischen den einzelnen Systemen.

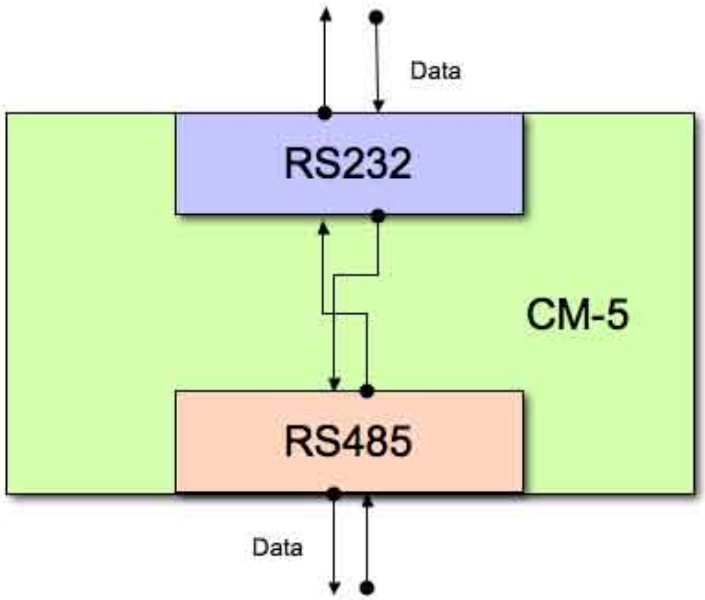

Abbildung 2.1: Mit dem Programm Toss.hex werden alle Pakete, die von der seriellen Schnittstelle des PCs eintreffen an die Dynamixel weitergeleitet (ebenso in der Entgegengesetzten Richtung)

2.2 Motor Modul AX-12

Der AX-12 ist ein sehr flexibler Servomotor der eine Vielzahl an Funktionen bietet. Er wird, wie alle Komponenten, über das serielle Bussystem mit einer Geschwindigkeit von 1.000.000 bps gesteuert. Dies ist ein Vielfaches von dem, was die meisten Servomotoren leisten. Der AX-12 kann durchgehend überwacht werden. Es stehen Informationen über die aktuelle Position, Temperatur, Kraft- und Stromverbrauch so wie Geschwindigkeit zur Verfügung. Zusätzlich wird ein kleines LED an der Seite als Statusanzeige verwendet. Besonders die Information über die aktuelle Position und ob der Motor noch in Bewegung ist macht ihn für komplexere Roboter und Aufgabenstellungen interessant.

Der Motor kann Positionen in einem 300 Gradwinkel mit einer Auflösung von 1024

[3]Das Programm ist unter dem Namen Toss.hex zu finden und ebenso das C-Programm als Sourcecode. Auf der Homepage der Firma Robotis ein Programm mit gleicher Funktionalität zu finden. Jedoch nur für den CM-2 und nur in kompilierter Form.

(etwa 0,3 Grad Genauigkeit) ansteuern. Ebenso kann die gewünschte Geschwindigkeit und verwendete Kraft vorgegeben werden.

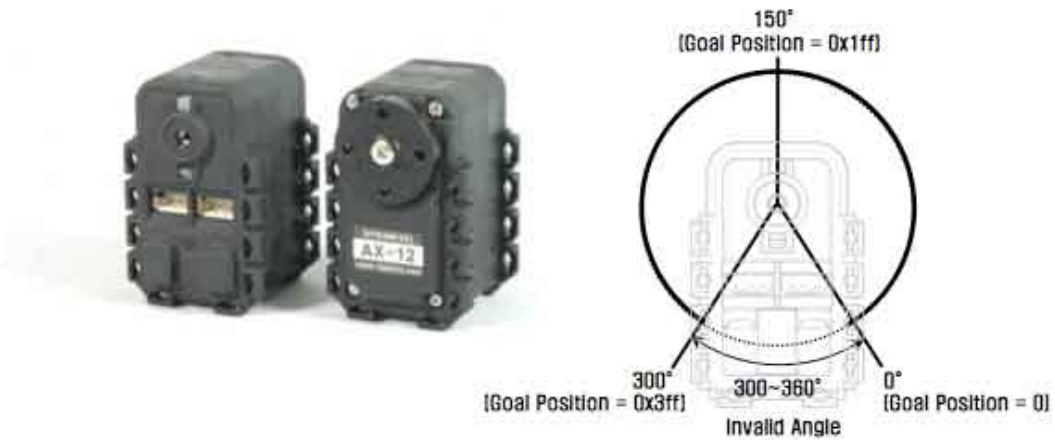

Abbildung 2.2: Der Servomotor AX-12

2.3 Sensor Modul AX-S

Das Dynamixel Sensor Module AX-S integriert eine Fülle an Funktionen. Es beinhaltet einen kleinen Buzzer, der Töne erzeugen kann, einen Abstandssensor und einen Sensor der auf Licht reagiert. Mit dem eingebauten geräuschempfindlichen Mikrofon können Geräusche registriert und gezählt werden. Er kann zusätzlich in Verbindung mit einer Infrarot-Fernbedienung genutzt werden um Informationen zu übertragen oder den Roboter zu steuern. Er kommuniziert wie der AX-12 auf dem 1.000.000 bps Bussystem.

Abbildung 2.3: Das Dynamixel Sensor Module AX-S

2.4 Distanzsensor

In Zusammenarbeit mit Sonia Seoane Puga [Puga, 2007] wurde ein zusätzliches Sensormodul entwickelt, das die zurückgelegte Distanz des Roboters messen kann. Das Modul besteht zunächst aus einem kleinen Mikrokontrollerboard, an das bis zu zwei Sensoren angeschlossen werden können. Auf dem Board befindet sich ein Atmega16, auf dem das von Bioloid benutzte Protokoll implementiert wurde. In Abbildung 2.4 ist auf der linken Seite der erste Prototyp des Board zu sehen. Auf ihm ist nur Platz für einen Sensor. In der finalen Version (rechts) wurde ein wesentlich kleinerer Mikrokontroller verwendet und Anschlüsse für 2 Sensoren angebracht.

Das Bioloid-Protokoll wurde implementiert, um das System homogen zu halten. So lässt sich das neue Sensormodul transparent in den bestehenden Bausatz einfügen. Dafür wurde ein Parser für die Bioloid Pakete in Form einer *Statemachine* realisiert. Dieser ist auf Geschwindigkeit optimiert um den hohen Anforderungen 1.000.000 bps des Bussystems gerecht zu werden. Das Programm für das Sensormodul ist unter dem Namen *sensormodul.c* zu finden.

Als Sensor für die Wegmessung wird ein optischer ME16-Encoder verwendet. Dieser wird in Abbildung 2.5 gezeigt. An ihn wurde ein Rad angebracht, welches am Boden mitläuft. Durch den Signal-Pegelwechsel des Encoders kann einfach festgestellt werden, wie weit und in welche Richtung sich das Rad bewegt hat. Am vierbeinigen Roboter werden 2 dieser Sensoren angebracht um auch drehende Bewegungen feststellen zu können.

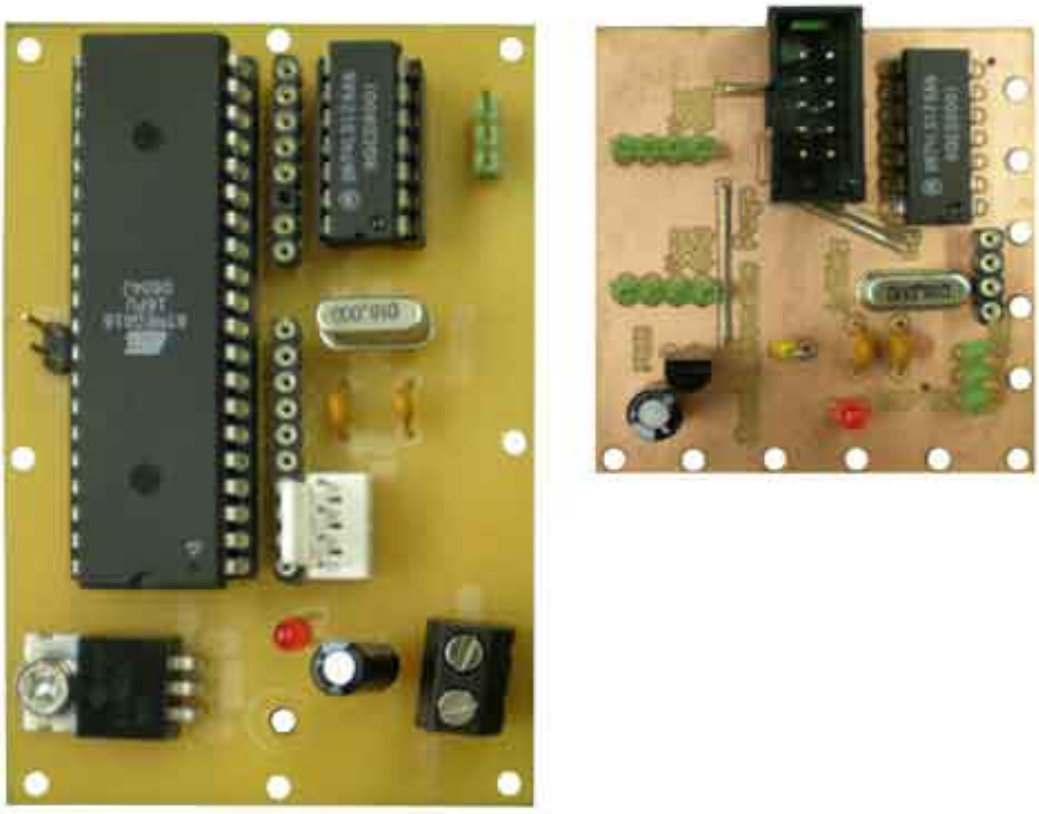

Abbildung 2.4: Das erste Board mit einem *Atmega16 PDIP* auf der linken Seite und das neue Board mit einem kleineren *Atmega16 TQFP* rechts.

Abbildung 2.5: Der Encoder für die Wegmessung.

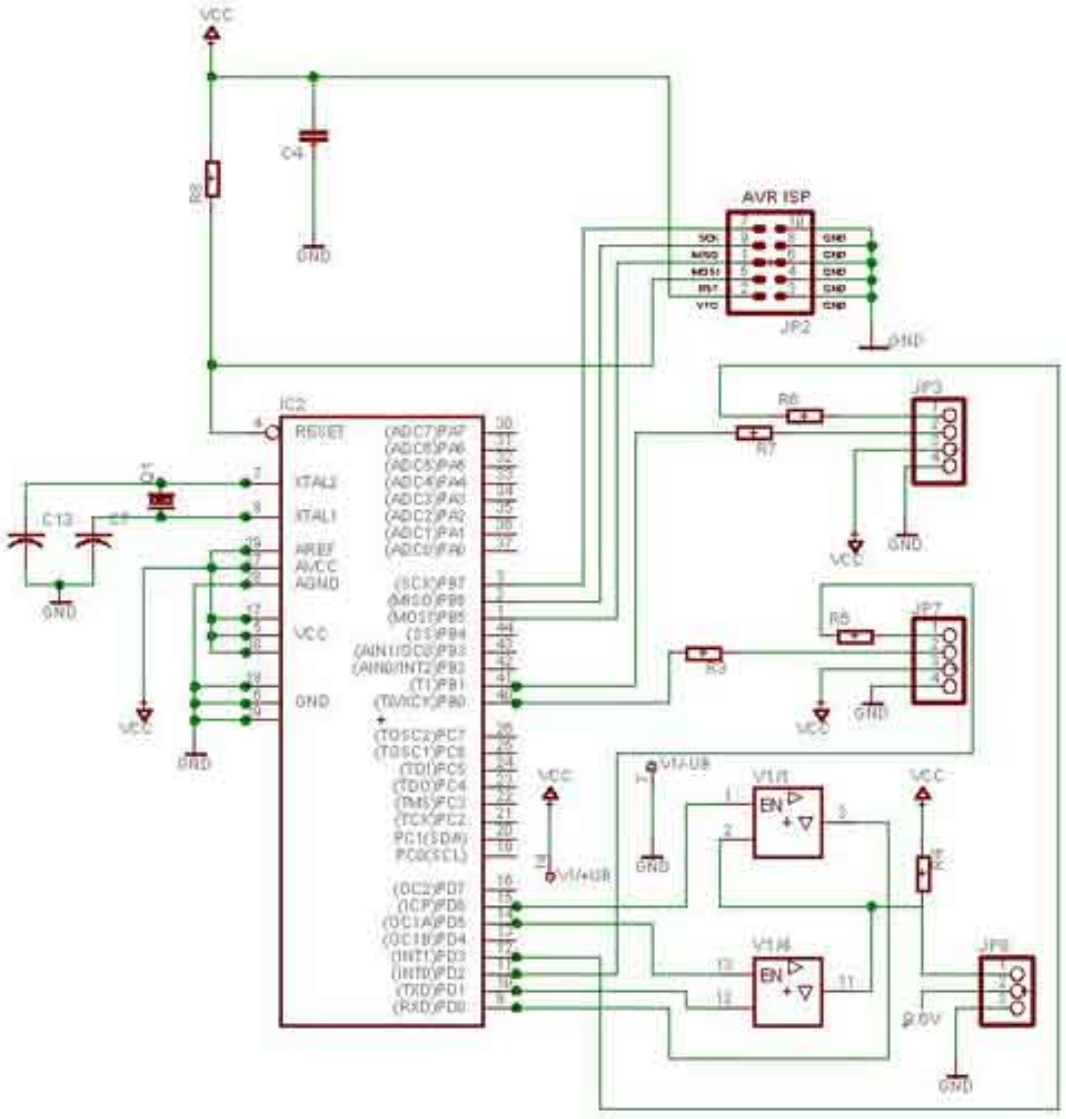

Abbildung 2.6: Die schematische Darstellung des neuen Boards.

2.5 Das Kommunikationsprotokoll

Alle Dynamixelkomponenten werden über einen asynchronen seriellen Bus durch ein Protokoll gesteuert. Es existieren 2 verschiedene Arten von Paketen. Die Steuerpakete enthalten ein auszuführendes Kommando für eine Dynamixel Einheit. Als Reaktion auf ein solches Steuerpaket folgt ein Antwortpaket oder auch Statuspaket genannt. Da alle Komponente am selben Bus hängen, ist es wichtig, dass jeder Dynamixel eine unikale ID besitzt um ihn ansprechen zu können. Im Steuerpaket wird die ID der gewünschten Einheit hinterlegt, worauf nur diese das Paket beachten wird. Ebenso schreibt der Sender eines Antwortpakets seine eigene ID hinein um die Herkunft zu kennzeichnen.

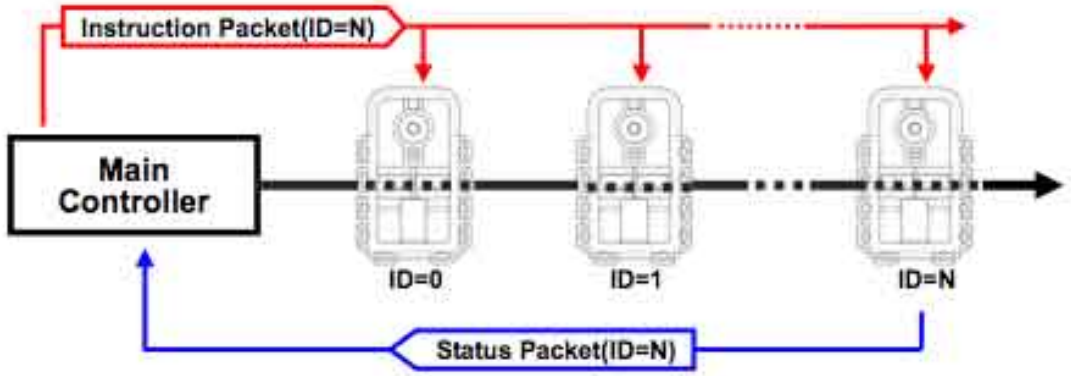

Abbildung 2.7: Das Kommunikationsprotokoll. Das Antwortpaket folgt als Reaktion auf ein Steuerpaket.

2.5.1 Das Steuerpaket

Das Steuerpaket ist an eine Dynamixel Einheit gerichtet und enthält einen in der Tabelle 2.1 zu sehenden Befehl, der ausgeführt werden soll. Das Paket hat folgenden Aufbau:

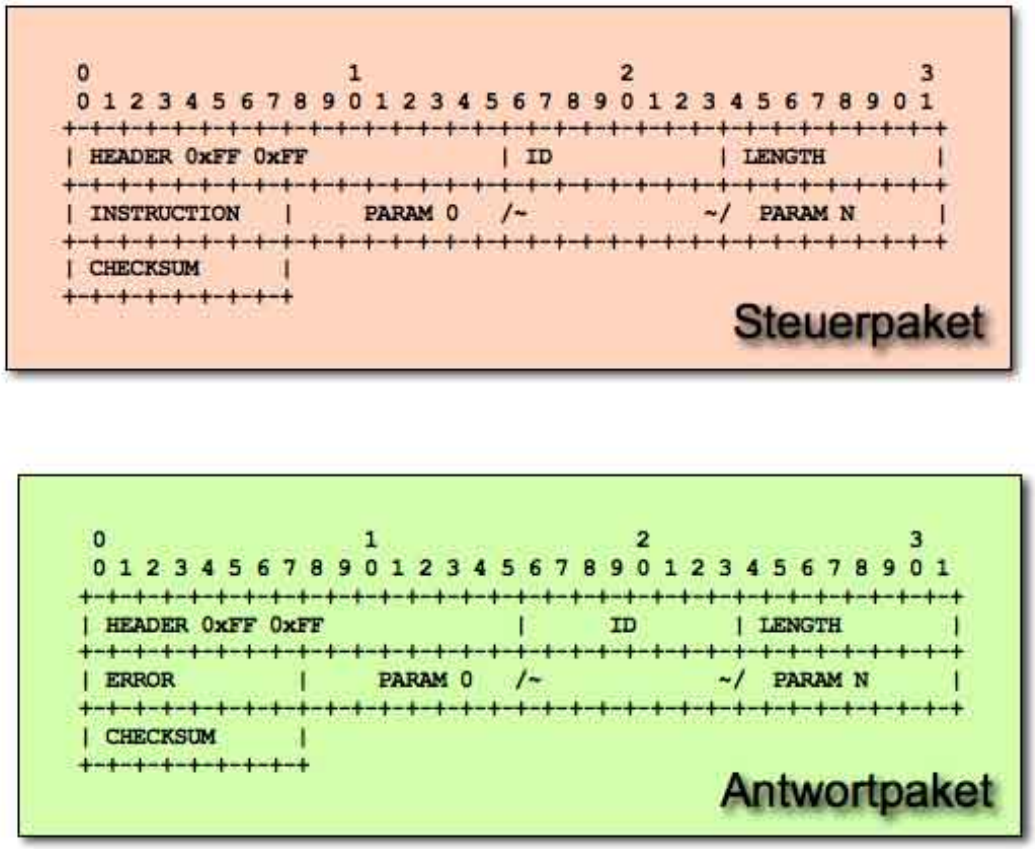

Abbildung 2.8: Steuer- und Antwortpaket

HEADER	Um den Start eines Paketes zu kennzeichnen werden 2 Byte mit 0xFF benutzt
ID	Die eindeutige ID des Dynamixel für den das Paket bestimmt ist
LENGTH	Da die Anzahl der Parameter variieren kann wird mit diesem Byte die Länge des Paketes angegeben. Die Länge wird durch Anzahl der Parameter + 2 angegeben
INSTRUCTION	Der Befehl, der von dem Dynamixel ausgeführt werden soll. Eine Beschreibung der möglichen Befehle ist in Tabelle 2.1 zu sehen
PARAMETER 0 – N	Für manche Befehle werden zusätzliche Parameter benötigt. Diese können hier nacheinander angegeben werden
CHECKSUM	Die Prüfsumme dient dazu um eventuelle Übertragungsfehler erkennen zu können. Sie wird wie folgt berechnet: $\sim (ID + Length + Instruction + Parameter1 + ... + ParameterN)$

Befehl	Beschreibung	Wert	Anzahl Parameter
PING	Auf dieses Paket folgt nur ein Statuspaket falls die Komponente aktiv ist.	0x01	0
READ DATA	Einen Wert aus der Kontrolltabelle lesen	0x02	2
WRITE DATA	Einen Wert in die Kontrolltabelle schreiben	0x03	2-N
REG WRITE	Gleich Funktion wie WRITE DATA, es wird jedoch auf ACTION gewartet bis der Wert tatsächlich geschrieben wird.	0x04	2-N
ACTION	Das REG WRITE Kommando wird nun geschrieben	0x05	0
RESET	Alle Werte werden zurückgesetzt	0x06	0
SYNC WRITE	Dieser Befehl wird benutzt um mehrere Dynamixel gleichzeitig zu steuern	0x83	4-N

Tabelle 2.1: Beschreibung der Befehle für ein Steuerepaket

2.5.2 Das Antwortpaket

Dieses Paket wird von einem Dynamixel als Reaktion auf ein Steuerpaket gesendet. Der Aufbau ist dem eines Steuerpaketes sehr ähnlich und ist ebenfalls in Abbildung 2.7 zu sehen.

HEADER	Um den Start eines Paketes zu kennzeichnen werden 2 Byte mit 0xFF benutzt
ID	Die eindeutige ID des Dynamixel von dem das Paket stammt
LENGTH	Die Länge wird durch Anzahl der Parameter + 2 angegeben.
ERROR	Falls ein Fehler aufgetreten ist, ist dieses Byte nicht 0
PARAMETER 0 – N	Hier könne zusätzlich benötigte Informationen gespeichert werden
CHECKSUM	Die Prüfsumme wird wie vorhin berechnet: $\sim (ID + Length + Error + Parameter1 + ... + ParameterN)$

2.6 Das Software Framework

In diesem Kapitel wird die Dynamixel Software-Bibliothek vorgestellt. Die Bibliothek entstand während dieser Diplomarbeit um eine möglichst einfache Bedienung der Dynamixel Komponenten zu gewährleisten. Sie ist komplett in C++ geschrieben und plattformunabhängig. Sie kapselt das im vorherigen Abschnitt beschriebene Protokoll in Klassen, erkennt automatisch Übertragungsfehler und definiert eine Schnittstelle zwischen PC und dem Bioloid-Bussystem. Durch Ihren modularen Aufbau ist es problemlos möglich neue Sensoren oder auch andere Komponenten transparent einzubinden. In den folgenden Abschnitten wird etwas detaillierter in die Bedienung und Handhabung der einzelnen Klassen eingegangen.

2.6.1 Designziele

Einfache Bedienung: Die Benuzung der Klassen soll so intuitiv wie möglich sein, um lange Einarbeitungszeit unnötig machen. Dies soll die Bedienung ohne Kenntnis des darunter liegenden Transportprotokolls gewährleisten.

Modularer Aufbau: Das System soll so modular wie möglich aufgebaut sein um einfach einzelne Komponente, wie etwa den Transportweg austauschen zu können.

Unempfindlich gegenüber Fehlern: Da verfälschte Befehle oder Sensordaten zu erheblichen Problemen führen können ist es wünschenswert, dass diese erkannt und korrigiert werden. Fehler bei der Übertragung sollen automatisch registriert und das entsprechende Paket neu versandt werden. Da es sich um ein Bussystem handelt, kann es auch vorkommen, dass Pakete in der falschen Reihenfolge eintreffen. In einem solchen Fall sollen diese gepuffert werden und anschließend richtig geordnet an das System ausgeliefert werden.

Plattformunabhängig: Die Bibliothek soll unabhängig vom verwendeten Betriebssystem einsetzbar sein.

Einfach erweiterbar: Es soll problemlos möglich sein neue Sensoren oder Motoren einzubinden und zu benutzen ohne das Framework verändern zu müssen.

Geschwindigkeit: Zwar sind die physischen Bewegungen der Motoren relativ langsam, doch für den Transport der Befehle und Daten ist eine schnelle Übertragung notwenig.

2.6.2 Dynamixel Klassen

Die Klasse *Dynamixel* stellt alle nötigen Funktionen zur Verfügung um Befehle an Dynamixel Module zu schicken, Daten zu Schreiben und zu Empfangen. Fehler bei der Übertragung werden erkannt und die Daten werden automatisch neu gesen-

det. Alle Klassen, die für die Kommunikation mit einer Dynamixel Komponente gedacht sind sollten von dieser Klasse abgeleitet werden. Ein Beispiel hierfür sind die Klassen *AX12* und *AXS* welche die volle Funktionalität für die gleichnamige Hardwareeinheit bereitstellen.

Da alle auf demselben Bus senden wird auch der Protokoll-Stapel gemeinsam genutzt. Es ist daher wichtig vor der ersten Instanzierung das ByteLayer-Objekt, welches genutzt werden soll, zu übergeben. Dies geschieht durch den statischen Methodenaufruf *Dynamixel::setByteLayer(ByteLayer*)*.

Ein Beispiel für die Verwendung dieser Klassen ist in Listing E.1 zu sehen.

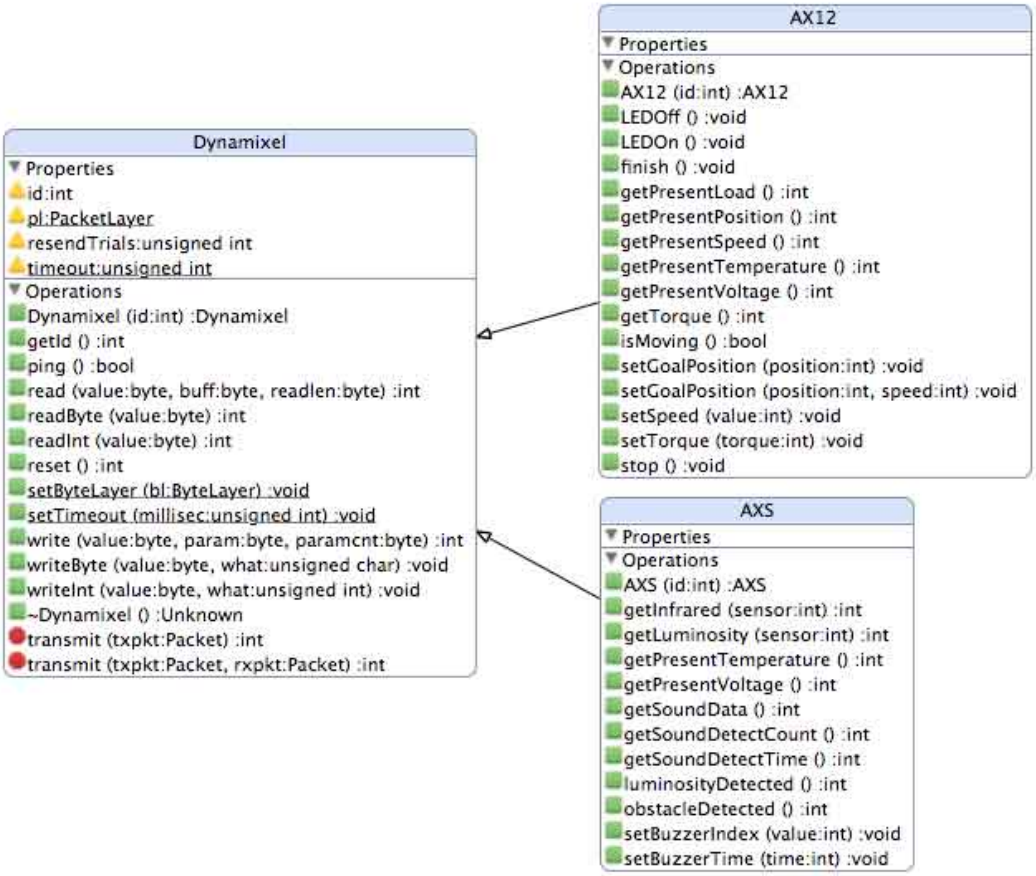

Abbildung 2.9: Klassendiagramm der Dynamixel Module

2.6.3 Der Protokoll-Stapel

Um eine flexible und zuverlässige Übertragung der Daten zu gewährleisten ist das, im vorherigen Abschnitt vorgestellte Protokoll in Form eines Schichtenmodells implementiert. Ein Paket wird durch die Klasse *Packet* repräsentiert, welche als Container für ein Steuer- oder Antwortpaket dient. Der Sender füllt eine solche Datenstruktur mit den gewünschten Informationen und übergibt sie dem *Packet-*

Layer, welcher wiederum die Dienste des *ByteLayer* in Anspruch nimmt um die Daten zu Versenden.

Eine der wichtigsten Klassen hierbei ist der *PacketParser*. Mit seiner Hilfe kann ein Paket aus einem *ByteStream* gelesen werden. Ein *ByteStream* muss lediglich die Methode *readByte()* implementieren, die das jeweils nächste zu parsenden Byte liefert. Dieses Modell ermöglicht es auch einen *PacketParser* einzusetzen um Beispielsweise empfangene Daten auf einem Mikrokontroller zu interpretieren.

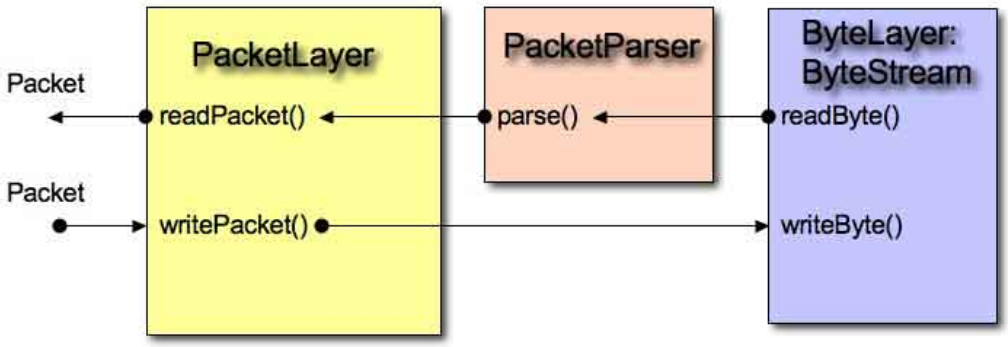

Abbildung 2.10: Die Kommunikation zwischen *PacketLayer* und *ByteLayer*

Im *PacketLayer* sind die Methoden *readPacket()* und *writePacket()* implementiert um ein Paket zu empfangen beziehungsweise zu senden. *writePacket()* zerlegt das Paket in einzelne Bytes und leitet es an den *ByteLayer* weiter. *readPacket()* bedient sich einem *PacketParser* um ein Packet aus dem *ByteLayer* zu empfangen. Der *ByteLayer* ist eine Unterklasse von *ByteStream* und kann deshalb direkt an den *PacketParser* übergeben werden.

Die Klasse *ByteLayer* ist für den eigentlichen Versand und Empfang der Daten verantwortlich. Derzeit ist nur die Verbindung über eine serielle Schnittstelle realisiert. Es kann jedoch jede Verbindung genutzt werden, die diese Schnittstelle implementiert. So könnten Beispielsweise auch Sockets für die Übertragung genutzt werden.

Derzeit existieren zwei Klassen, die als *ByteLayer* verwendet werden können. Die erste ist die plattformunabhängige Klasse *QTSerialPortBL*, die, die freie Bibliothek *QextSerialPort*[4] kapselt. Hier wird die *QextSerialPort Version 1.1* genutzt, welche kompatibel zu *Qt 4*[5] ist. Die Benutzer von *Qt 2 & 3* müssen die *QextSerialPort* in der Version 0.9 benutzen, wozu möglicherweise Änderungen an *QTSerialPortBL*

[4] http://qextserialport.sourceforge.net

[5] Qt ist eine Bibliothek der Firma Trolltech, die, für die plattformübergreifende Programmierung graphische Bedienoberfläche in C++ gedacht ist.
Homepage: http://trolltech.com/products/qt

nötig werden. Auf Kosten der Plattformunabhängigkeit wurde eine weitere Klasse für die Kommunikation mit einer seriellen Schnittstelle geschrieben, genannt *LightSerialPort*. Sie ist für POSIX Systeme gedacht und kann vollständig auf Threads, Signale und Interprozesskommunikation verzichten. Es wird auch kein riesiges Framework wie etwa *Qt* benötigt, was die Klasse sehr klein und schnell macht. Die Klasse *LightSerialPortBL* stellt alle benötigten Schnittstellen bereit um sie im beschriebenen Schichtenmodell einsetzen zu können.

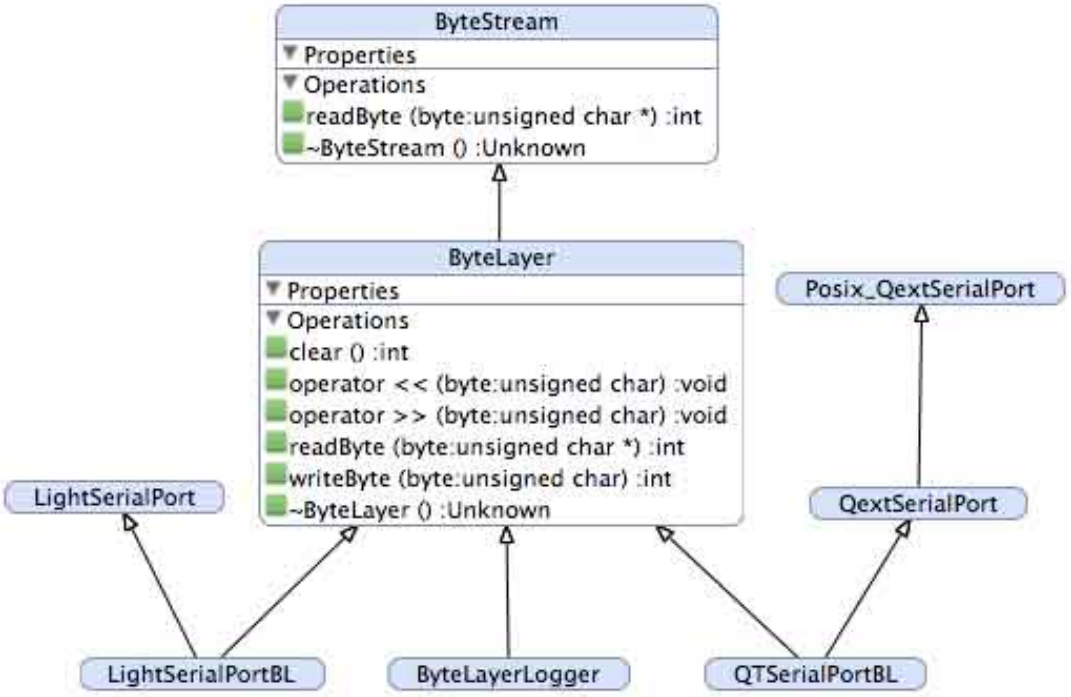

Abbildung 2.11: Klassendiagramm des ByteLayer

Kapitel 3

Lernverfahren

Dieses Kapitel soll zunächst eine kurze Einführung in die Materie des reinforcement learning bieten. Die hierbei verwendeten Definitionen und Darstellungen sind größtenteils aus Sutton and Barto [1998] entnommen. Anschließend folgt eine Vorstellung der Testumgebung, die aus einem einfachen Krabbelroboter[1] und einem Simulator für diesen besteht. Im darauf folgenden Abschnitt werden die verwendeten Algorithmen erläutert und deren Tests ausgewertet. Es werden hier jedoch nur Verfahren aus der Klasse der Temporal-Difference-Methoden verwendet, da diese für die Problemstellung bestens geeignet sind.

3.1 Grundlagen

Ziel des reinforcement learning ist es, einem Agenten (dies kann zum Beispiel eine Simulation, ein Roboter oder ein beliebig anderes autonomes System sein) die Möglichkeiten zu geben, aus seiner Interaktion mit der Umwelt zu lernen. Normalerweise ist das Verhalten eines Agenten durch ein fest vorgegebenes Programm geregelt. Dies kann zwar mittels Sensoren dynamisch auf Veränderungen reagieren, jedoch müssen diese Reaktionen vorher definiert werden. In manchen Fällen ist es sehr schwer oder unmöglich vorherzusagen, in welchen Situationen sich der Agent später befinden wird. Beim reinforcement learning versucht der Agent selbst durch ausprobieren ein optimales Verhalten für seine Situation zu finden. Nach ein paar Schritten in seiner Umgebung weiß der Agent welche Zustände beziehungsweise welche Aktionen für ihn von Vorteil sind und welche besser vermieden werden. Der große Vorteil besteht darin, nicht mehr jeden einzelnen Schritt genau vorgeben zu müssen, sondern nur noch das zu erreichende Ziel. Hierfür wird für gute Zustände oder gute Aktionen eine Belohnung[2] definiert. Diese Belohnung muss nicht unmittelbar erfolgen, sondern kann auch das Resultat einer längeren Aktionsfolge sein. Ziel des Agenten beim verstärkten Lernen ist es diese Belohnungen auf lange Sicht (die Lebensdauer des Agenten) zu maximieren.

Einer der größten Unterschiede zu anderen Verfahren aus dem Bereich des maschinellen Lernens besteht darin, dass es hier keinen Lehrer gibt, der die Aktionen des Agenten bewertet. Es werden auch keine Daten als Beispiele für den Agenten

[1] Eine detaillierte Beschreibung der verwendeten Hardware ist in Kapitel 2 zu finden.

[2] In der Literatur wird auch manchmal von einer Kostenfunktion gesprochen, die es zu minimieren gibt.

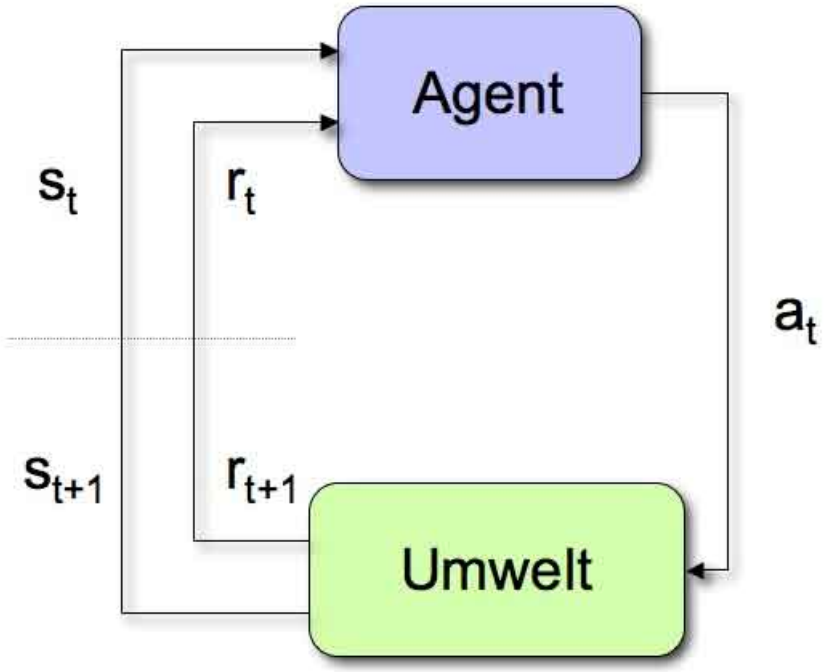

Abbildung 3.1: Der Agent in Interaktion mit seiner Umwelt. [siehe Sutton and Barto, 1998, Kapitel 3]

bereitgestellt, anhand derer er lernen oder sein derzeitiges Verhalten evaluieren könnte. Alle Informationen werden einzig aus getätigten Aktionen und der Reaktion der Umgebung auf diese gewonnen.

3.1.1 Zustände und Aktionen

Formal gesprochen befindet sich der Agent zu einem Zeitpunkt t in einem Zustand s_t und führt die Aktion a_t aus. Durch diese Aktion wechselt der Agent in einen neuen Zustand s_{t+1} und erhält eine Belohnung r_{t+1} (Siehe Abbildung 3.1). $\mathcal{S}$ ist hierbei die Menge aller möglichen Zustände und $\mathcal{A}(s_t)$ die Menge aller möglichen Aktionen im Zustand s_t. Das Belohnungssignal r ist ein numerischer Wert, bei dem, wenn er negativ ist auch von einer Bestrafung gesprochen wird. Bei den empfangen Signalen muss es sich nicht um die tatsächlichen Werte handeln. So kann es Beispielsweise vorkommen, dass durch fehlerhafte Sensordaten ein falscher Zustand angenommen wird.

3.1.2 Die Policy

Die Auswahl der Aktionen trifft der Agent anhand seiner aktuellen policy π_t. Diese policy ist eine Abbildung von Zuständen auf Aktionen. Genauer gesagt bestimmt

sie die Wahrscheinlichkeit in der eine Aktion a in einem Zustand s gewählt wird.

$$\pi(s,a) \quad = \quad Pr\{a = a_i \mid s = s_t\} \tag{3.1}$$

Eine solche policy kann im einfachsten Fall durch eine Tabelle repräsentiert werden, es kann sich aber auch um eine komplexe Funktion handeln, wie etwa ein neuronales Netz oder ein Entscheidungsbaum. In Kapitel Abschnitt 3.2 wird näher auf die Realisierung von policies eingegangen.

3.1.3 Der Markovsche Entscheidungsprozess

Ist die Wahrscheinlichkeit für das Erreichen eines bestimmten Nachfolgezustandes nur vom aktuellen Zustand und der gewählten Aktion abhängig, so spricht man von einem Markov Entscheidungsprozess (engl. Markov Decicion Process MDP). So können Entscheidungen aufgrund des aktuellen Zustandes getroffen werden und es müssen nicht alle vorherigen Zustände und Aktionen berücksichtigt werden. Dies ist zum Beispiel bei Backgammon oder Schach der Fall. Alle relevanten Informationen sind durch die aktuelle Position der Spielfiguren beschrieben. Es spielt hierbei keine Rolle durch welche vorherigen Züge der Spielstand erreicht wurde.

Ein Markov Entscheidungsprozess liegt vor, wenn die folgende Gleichung erfüllt ist:

$$Pr\{s_{t+1} = s', r_{t+1} = r \mid T(s_{0:t}, a_{0:t})\} \tag{3.2}$$
$$= Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\} \tag{3.3}$$

Wobei $T(s_{0:t}, a_{0:t})$ den bisherigen Verlauf von Beginn bis zum Zeitpunkt t definiert. Ist der Zustands- und Aktionsraum zusätzlich endlich, so handelt es sich um einen finiten Markov Entscheidungsprozess.

Ein Markov Entscheidungsprozess wird durch die Menge der Zustände $\mathcal{S}$, die Menge der Aktionen $\mathcal{A}$(s) und durch die Übergangswahrscheinlichkeiten zwischen den Zuständen

$$\mathcal{P}^a_{ss'} = Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\} \tag{3.4}$$

Zusammen mit dem Erwartungswert der Belohnung bei diesen Übergängen

$$\mathcal{R}^a_{ss'} = E\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\} \tag{3.5}$$

vollständig beschrieben.

Die Markov Eigenschaft ist nicht immer gegeben, wird jedoch häufig angenommen, da dies meist eine entscheidende Vorraussetzung für die reinforcement learning Algorithmen ist.

3.1.4 Die Belohnung

Wie schon erwähnt ist das Ziel des Agenten seine Belohnung auf lange Sicht zu maximieren. Es genügt also nicht nur die Aktion zu wählen, die im nächsten Schritt die größte Belohnung erzielt, sondern es müssen auch Belohungen in betracht gezogen werden, die weiter in der Zukunft liegen.

Für episodische Aufgaben, dies sind Aufgaben die einen oder mehrere Endzustände besitzen, lässt sich die langfristige Belohnung einfach durch die Summe aller folgenden Belohnungen definieren. So ist die langfristige Belohnung R_t zum Zeitpunkt t gegeben durch:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T \tag{3.6}$$

Dies ist jedoch für kontinuierliche Aufgaben nicht möglich. Kontinuierliche Aufgaben haben im Gegensatz zu episodischen Aufgaben keinen Endzustand[3]. Somit würde die Belohnung immer ins Unendliche wachsen. Aus diesem Grund werden die Belohnungen für Aktionen die weiter in der Zukunft liegen durch einen Parameter γ, $0 \leq \gamma \leq 1$, abgeschwächt.

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{3.7}$$

Der Parameter γ wird oft auch als *discount rate* bezeichnet und sorgt dafür, dass die Belohnung endlich bleibt. Die beiden Definitionen können in der folgenden Gleichung zusammengefasst werden:

[3]Bei dem in dieser Arbeit behandelte Problem handelt es sich ebenfalls um eine kontinuierliche Aufgaben, da der Roboter sich ständig Vorwärtsbewegen soll.

$$R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1} \tag{3.8}$$

Hier ist $T = \infty$ für kontinuierliche Aufgaben und $\gamma = 1$ für den episodischen Fall.

3.1.5 Wertefunktionen

Beim reinforcement learning wird meist versucht der Wert eines Zustandes oder der Wert einer Aktion abzuschätzen. Das heißt zu bewerten wie gut es ist, sich in einem Zustand zu befinden oder in einem Zustand eine bestimmte Aktion auszuführen. Der Wert wird hierbei meist als die zu erwartende Belohnung, wie im vorherigen Abschnitt beschrieben, definiert. Diese Belohnung ist natürlich von der aktuell verwendeten policy abhängig. Deshalb ist die *Zustands-Wertefunktion* (*state-value function*) für die policy π wie folgt beschrieben:

$$V^\pi(s) = E_\pi\{R_t \mid s_t = s\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s \right\} \tag{3.9}$$

E_π ist hier der erwartete Wert, wenn zu jedem Zeitpunkt die policy π befolgt wird. Ähnlich sieht die Definition einer *Aktions-Wertefunktion* (*action-value function*) aus:

$$\begin{aligned} Q^\pi(s,a) &= E_\pi\{R_t \mid s_t = s, a_t = a\} && (3.10) \\ &= E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \right\} && (3.11) \end{aligned}$$

Eine der wichtigsten Erkenntnisse für reinforcement learning ist der Zusammenhang zwischen einem Zustand und seinem Nachfolger. Dieser Zusammenhang wird durch die Bellmann Gleichung beschrieben, die besagt, dass für jede policy π und jeden Zustand s gilt[4]:

[4]Für den vollständigen Beiweis siehe [Sutton and Barto, 1998, Kapitel 3.7]

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma V^{\pi}(s')\right] \tag{3.12}$$

Für eine Aktions-Wertefunktion sieht die Gleichung wie folgt aus:

$$Q^{\pi}(s,a) = \sum_{s'} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma \sum_{a'} \pi(s',a') Q^{\pi}(s',a')\right] \tag{3.13}$$

Mittels $\mathcal{P}^{a}_{ss'}$, $\mathcal{R}^{a}_{ss'}$ und π wird hierbei über alle Möglichkeiten gemittelt und mit der Wahrscheinlichkeit ihres Auftretens gewichtet.

3.1.6 Evaluation und Verbesserung

Mit den zuvor definierten Wertefunktionen lassen sich auch policies vergleichen. So ist eine policy π genauso gut oder besser als eine andere policy π' falls für alle Zustände s gilt: $V^{\pi}(s) \geq V^{\pi'}(s)$. Eine optimale policy $\pi^{\star}$ ist mindestens genauso gut wie alle anderen policies[5]. Deshalb gilt:

$$V^{\pi^{\star}}(s) = \max_{\pi} V^{\pi}(s) \quad \forall s \tag{3.14}$$

Die Wertefunktion einer optimalen policy wird der besseren Lesbarkeit halber mit $V^{\star}$ bezeichnet. Weil $V^{\star}$ eine Wertefunktion wie jede andere ist, muss sie ebenfalls die Bellmann Gleichung erfüllen. Für den Spezialfall einer optimalen Wertefunktion kann diese Gleichung zur *Bellmann-Optimalitätsgleichung*[6] umgeformt werden, die nun keinen Bezug mehr zu einer bestimmten policy besitzt.

$$V^{\star}(s) = \max_{a} \sum_{s'} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma V^{\star}(s')\right] \tag{3.15}$$

[5] Es existiert immer mindestens eine solche policy, es muss jedoch nicht die einzige sein.
[6] Beiweis siehe [Sutton and Barto, 1998, Kapitel 3.8]

Somit ist der Wert eines Zustandes unter einer optimalen policy die Belohnung, die aus der besten Aktion folgt. Daraus lässt sich ebenfalls schließen, dass eine optimale policy in jedem Zustand die beste Aktion wählt.

Durch diese Erkenntnis und aus der Gleichung 3.15 ergibt sich nun eine Iterationsvorschrift um $V^\star$ zu berechnen:

$$V_{k+1}(s) = \max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V_k(s')\right] \tag{3.16}$$

Es kann gezeigt werden, dass diese Anpassung immer gegen die optimale Wertefunktion $V^\star$ konvergiert. Dieses Verfahren ist auch unter dem Namen *Value Iteration* bekannt.

3.1.7 Temporal-Difference-Methoden

Temporal-Difference Methoden (oder kurz TD-Methoden) kommen, anders als Value-Iteration, ohne die Zustandsübergangsfunktion $\mathcal{P}^a_{ss'}$ und die Belohnungsfunktion $\mathcal{R}^a_{ss'}$ aus. TD-learning ist also ein modellfreier Ansatz. Die TD-Methoden basieren ebenfalls auf der Bellmanngleichung. Sie verwenden allerdings nicht die Erwartungswerte für Nachfolgezustände und Belohnungen, stattdessen wird zuerst ein Aktion ausgeführt und die Berechnungen erfolgen dann auf den beobachteten Folgen dieser Aktion, die aus der Belohnung und dem Nachfolgezustand bestehen. Die Aktualisierungsregel sieht für Zustandswertefunktionen wie folgt aus:

$$V(s_t) \leftarrow V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \tag{3.17}$$

Der Ausdruck $r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ wird in der Literatur meist als *TD-Error* bezeichnet. Da die Belohnung r_{t+1} und der Nachfolgezustand $V(s_{t+1})$ zum Zeitpunkt t noch nicht bekannt sind, wird zuerst eine Aktion a_t ausgeführt und die Aktualisierung zum Zeitpunkt $t+1$ vorgenommen. Der Parameter α $(0 < \alpha \leq 1)$ ist die so genannte Lernrate. Sie bestimmt wie stark die aktuelle Beobachtung in der neuen Schätzung gewichtet wird. Die Lernrate sollte von Schritt zu Schritt abnehmen. Dies ist notwenig um eine Konvergenz zu garantieren. Dazu muss die Lernrate die folgenden Bedingungen erfüllen:

$$\sum_{k=1}^{\infty} \alpha_k(a) = \infty \quad \text{und} \quad \sum_{k=1}^{\infty} \alpha_k^2(a) < \infty \tag{3.18}$$

```
1  Initialize Q(s,a) arbitrarily, π to the current policy
2  Repeat (for each episode)
3    Initialize s
4    Repeat (for each step of episode):
5      a ← action given by π for s
6      Take action a, observe r and next state s'
7      V(s) ← V(s) + α[r + γV(s') − V(s)]
8      s ← s'
9    until s is terminal
```

Listing 3.1: TD-learning Pseudocode

```
1  Initialize Q(s,a) arbitrarily
2  Repeat (for each episode)
3    Initialize s
4    Choose a from s using policy derived from Q
5    Repeat (for each step of episode):
6      Take action a, observe r, s'
7      Choose a' from s' using policy derived from Q
8      Q(s,a) ← Q(s,a) + α[r + γQ(s',a') − Q(s,a)]
9      s ← s'
10     a ← a'
11   until s is terminal
```

Listing 3.2: SARSA Pseudocode

Dabei ist $\alpha_k(a)$ die Lernrate nachdem die Aktion a zum k-ten mal ausgeführt wurde. Die Bedingung ist zum Beispiel für den Mittelwert $\alpha_k(a) = \frac{1}{k}$ erfüllt.

Analog zu Zustands-Wertefunktionen lassen sich auch Aktions-Wertefunktionen mit Temporal-Difference Methoden lernen. Der SARSA-Algorithmus benutzt folgende Aktualisierungsregel für seine Schätzungen:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{3.19}$$

Der Name SARSA stammt von den 5 Werten $\mathbf{s}_t, \mathbf{a}_t, \mathbf{r}_{t+1}, \mathbf{s}_{t+1}, \mathbf{a}_{t+1}$, die bei jedem Schritt benötigt werden.

3.2 Exploration und Exploitation

Wie schon im Unterabschnitt 3.1.2 erwähnt, ist eine policy eine Abbildung von Zuständen auf Aktionen. Es gibt jedoch Situationen, in denen es nicht sinnvoll ist, immer nur die beste Aktion zu wählen. Dies kann zum Beispiel in einem frühen Stadium des Lernens der Fall sein, in dem nur wenige Informationen über die Umwelt bekannt sind und die beste Aktion noch gar nicht fest steht. Dann ist es notwendig zuerst möglichst viele unterschiedliche Aktionen auszuprobieren um überhaupt gute Aktionen finden zu können. Es ist nicht ganz einfach zu entscheiden, ab wann genug ausprobiert wurde und mit der Ausnutzung des gewonnenen Wissens begonnen werden kann. Man spricht hierbei vom so genannten *Exploration-Exploitation Dilemma*.

Es werden normalerweise drei Arten von policies für Q-Funktionen verwendet:

- Die greedy policy: Keine Exploration. Es wird immer die beste Aktion gewält.
- Die epsilon greedy policy: Mit einer Wahrscheinlichkeit von ϵ wird eine Zufallsaktion gewählt, mit der Wahrscheinlichkeit von $1 - \epsilon$ die beste Aktion.

$$P(s, a_i) = \begin{cases} 1 - \epsilon + \frac{1}{|A_s|} & wenn \quad a_i = \text{argmax}_{a'} Q(s, a') \\ \frac{1}{|A_s|} & sonst \end{cases} \tag{3.20}$$

- Die soft-max policy[7]: Diese policy verteilt die Wahrscheinlichkeit für eine Aktion gewählt zu werden gleichmäßig anhand des Q-Wertes.

$$P(s, a_i) = \frac{\exp(\beta \cdot Q(s, a_i))}{\sum_{j=0}^{|A_s|} \exp(\beta \cdot Q(s, a_j))} \tag{3.21}$$

  Somit haben auch zwei Aktionen mit ähnlichem Wert eine ähnliche Chance genommen zu werden. Der Parameter β regelt hier das Verhältnis von Exploration und Exploitation. Für $\beta \to \infty$ wird diese polcy zu einer greedy policy. Dieses vorgehen erzielt etwas bessere Ergebnisse als die epsilon greedy policy, es ist jedoch wesentlich schwerer den Parameter β richtig zu wählen, da dieser stark von den Q-Werten abhängig ist.

Die Wahl einer rein zufälligen Aktion für die Exploration ist nicht optimal. Man spricht hierbei auch von einer *ungerichteten Exploration*. Whitehead konnte in [Whitehead, 1991a,b] zeigen, dass eine ungerichtete Exploration selbst in einer deterministischen Welt mit diskreten Zuständen und Aktionen ineffizient ist. Die

[7] Diese Funktion ist auch Boltzmann Verteilung bekannt.

Komlexität wächst im worst-case exponentiell zur Anzahl der Zustände. [Thrun, 1992] und [Wyatt, 1997] haben sich mit diesem Problem auseinandergesetzt und nutzen eine *gerichtete Exploration*. Diese nutzt weitere Informationen um die Exploration effektiver zu machen. [Thrun, 1992] bewies in seiner Arbeit, dass die Komplexität einer gerichteten counter-basierenden Exploration im worst-case höchstens nur noch polynomiell mit der Anzahl der Zustände wächst.

Um die gerichtete Exploration mit den oben genannten policies nutzen zu können, erfolgt die Aktionswahl nun nicht mehr anhand der Werte der Q-Funktion, sondern sie wird von nun an von einer neuen Funktion *Eval* abhängig gemacht.

$$Eval(s,a) = \Gamma Q(s,a) + (1-\Gamma)\xi\Psi(s,a) \tag{3.22}$$

$Q(s,a)$ ist die bekannte Q-Funktion und $\Psi(s,a)$ ist der so genannte Explorationsterm. Je höher dieser Term ist, desto wichtiger ist es diese Aktion für einen Explorationsschritt zu wählen. Der Parameter ξ skaliert diesen Term. Mittels Γ ($0 \leq \Gamma \leq 1$) kann die Gewichtung von Q-Wert und Explorationsterm eingestellt werden.

3.2.1 Counter-basierende Exploration

Für jede Aktion wird ein Zähler $c(s,a)$ mitgeführt, der angibt, wie oft eine Aktion bereits ausgeführt wurde[8]. Für diesen einfachen Fall kann einfach

$$\Psi(s,a) = -c(s,a) \quad \text{oder} \tag{3.23}$$
$$= -c(s,a)/f(c(s)) \tag{3.24}$$

gesetzt werden. Die Funktion f in Gleichung 3.24 dient dazu, den Wert in Abhängigkeit der anderen Zähler zu skalieren. Somit werden die Aktionen bevorzugt, die in der Vergangenheit nicht so häufig ausgeführt wurden. Dies ist das einfachste und zugleich effektivste Verfahren um zielgerichtet Aktionen auszuführen. Deshalb wird es auch für alle späteren Anwendungen dieser Arbeit benutz.

In Abbildung 3.2 ist ein Vergleich zwischen einer ungerichteten Exploration mit zufälligen Aktionen (oben) und der Counter-basierenden Version (unten) zu se-

[8] In [Thrun, 1992] wird nur das Verfahren für Zustands-Wertefunktionen beschrieben. Dies wurde in dieser Thesis leicht abgeändert um es auf Aktions-Wertefunktion anwendbar zu machen. Es wird dadurch auch einfacher, da keine Nachfolgezustände betrachtet werden müssen.

hen[9]. Auf der Y-Achse sind die Aktualisierungen der Aktionswerte zu sehen. Man sieht sehr deutlich, dass die Q-Funktion im unteren Diagramm sehr viel schneller konvergiert und die Schätzungen sich schneller an den wahren Wert annähern.

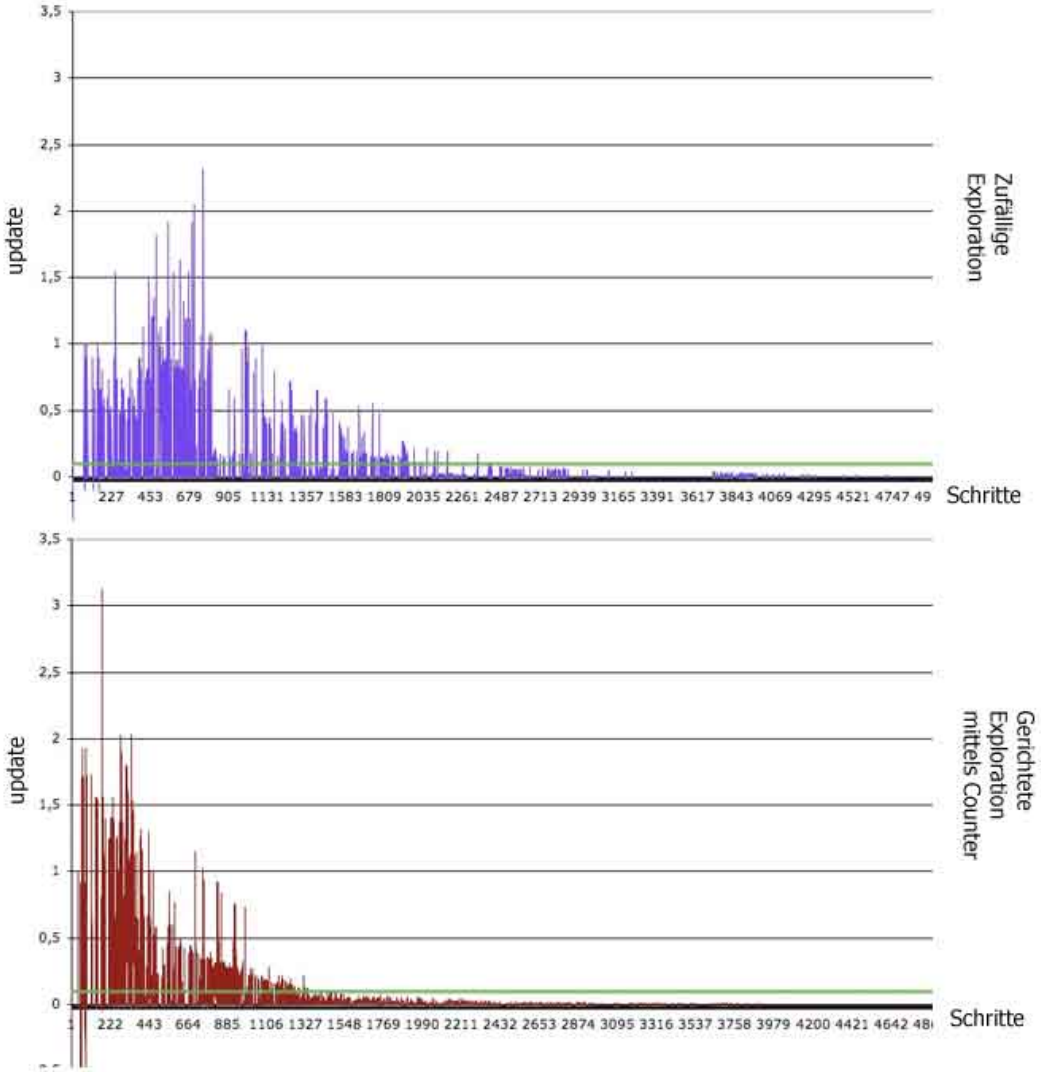

Abbildung 3.2: Vergleich ungerichtete Exploration gegenüber der Counter-basierenden Exploration

3.2.2 Zeitlich-basierende Exploration

Dieses Verfahren misst die Zeit, die seit der letzten Ausführung einer Aktion vergangen ist. Es werden Aktionen bevorzugt, die weiter in der Vergangenheit ausgeführt wurden. In [Sutton, 1990] wird die Wurzel der Zeit $p(s, a)$, die seit der letzten Ausführung vergangen ist, als Explorationsterm benutzt.

[9] Als Versuch dient eine deterministische 5x5 Welt und es wird normales Q-learning als Algorithmus verwendet.

$$\Psi(s,a) = \sqrt{p(s,a)} \tag{3.25}$$

3.2.3 Fehler-basierende Exploration

Dieser Ansatz basiert darauf, dass Zustände oder Aktionen mit großen Änderungen in der Wertefunktion nochmals besucht, beziehungsweise ausgeführt werden sollten. Hierzu kann einfach die letzte Aktualisierung oder der Mittelwert der letzten n Aktualisierungen benutzt werden:

$$\Psi(s,a) = n^{-1} \sum_{i=t-n}^{t} \Delta Q(s,a)_i \tag{3.26}$$

Nähere Informationen und Implementationen sind in [Thrun and Möller, 1991], [Schmidhuber, 1991] und [Thrun and Möller, 1992].

3.3 Die Testumgebung

Um eine visuelle Darstellung der Aktions-Wertefunktionen zu ermöglichen dient als Testumgebung ein Roboter mit lediglich 2 Freiheitsgraden wie in [Tokic, 2006]. Der verwendete Roboter ist in Abbildung 3.4 zu sehen. Er kann sich vorwärts bewegen in dem er mit seinem Arm nach vorn greift und sich anschließend zieht. Es ist ihm möglich seinen Körper auf der Seite des Armes ein wenig vom Boden zu heben.

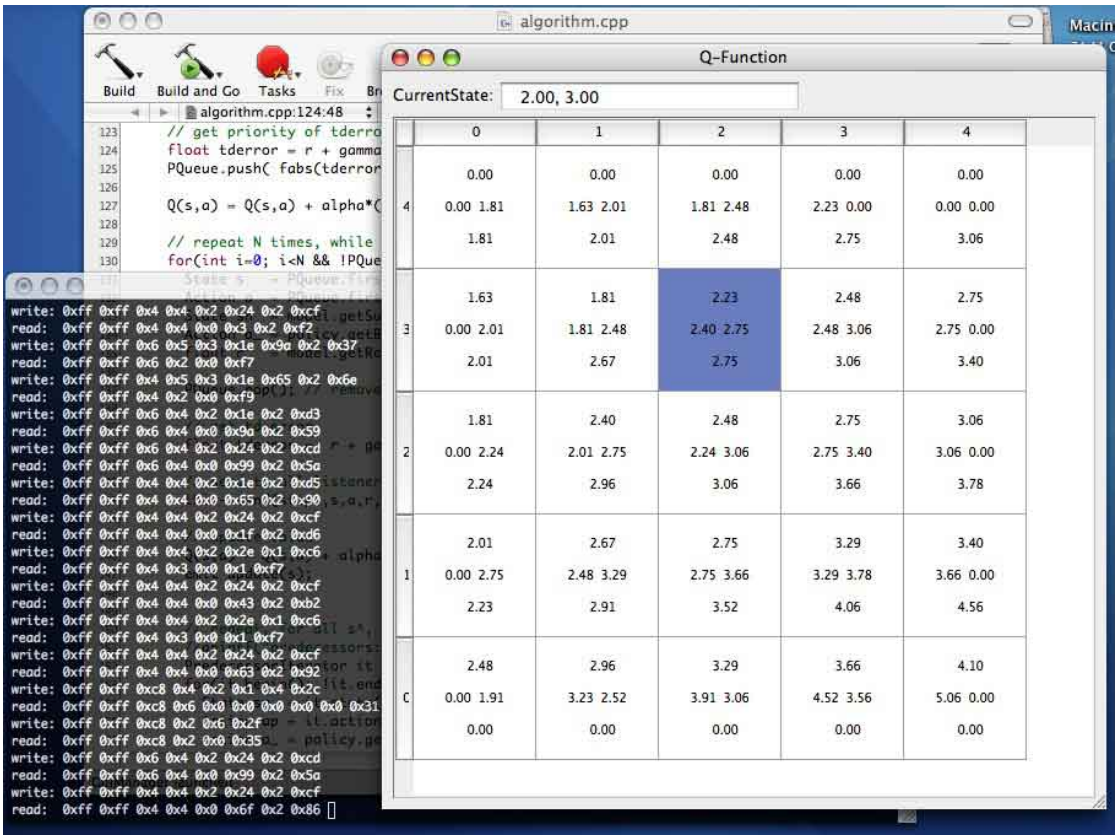

Abbildung 3.3: Der Simulator für den einbeiniger Roboter. Das blaue Feld kennzeichnet die aktuelle Position der Spitze des Beins.

In Abbildung 3.3 ist der Simulator zu sehen. Jedes Rechteck stellt einen Zustand dar. Die vier Zahlen in jedem Zustand repräsentieren den jeweiligen Wert für die Aktionen *oben*, *unten*, *links* und *rechts*. Das blaue Feld markiert den aktuellen Zustand der Spitze des Roboterarms. Der kontinuierliche Zustandsraum wurde gleichmäßig in eine 5x5 Welt diskretisiert.

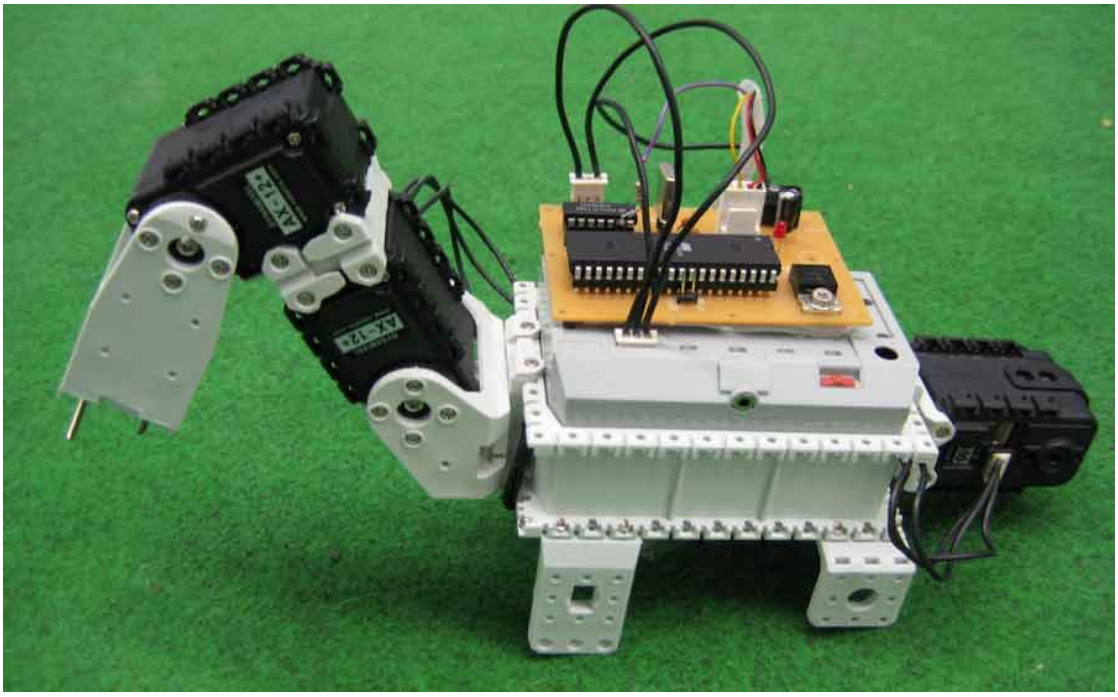

Abbildung 3.4: Der Krabbelroboter mit einem Bein.

3.4 Q-learning

Der erste hier getestete Algorithmus ist Q-learning mit einer Counter-basierenden policy. Mittels Q-learning können Aktions-Wertefunktionen gelernt werden, ohne, dass der Agent die Gegebenheiten seiner Umwelt kennen muss. Anders als bei Value-Iteration bei der die Zustandsübergangsfunktion $\mathcal{P}^a_{ss'}$ und die Belohnungsfunktion $\mathcal{R}^a_{ss'}$ bekannt sein müssen. Der Pseudocode ist in Listing 3.3 zu sehen. Q-learning hat den Vorteil gegenüber SARSA[10], das es *off-policy* arbeitet. *Off-policy* Methoden schenken Explorationsschritten keine Bedeutung, da diese meist nicht optimal im gegenwärtigen Zustand sind und gehen bei ihren Berechnungen immer von einer greedy-policy aus.

```
1 Initialize Q(s,a) arbitrarily
2 Repeat (for each episode)
3   Initialize s
4   Repeat (for each step of episode):
5     Choose a from s using policy derived from Q
6     Take action a, observe r, s'
7     Q(s,a) ← Q(s,a) + α [r + γ max_a' Q(s',a') − Q(s,a)]
8     s ← s'
9   until s is terminal
```

Listing 3.3: Q-learning Pseudocode

In Abbildung 3.5 ist die Konvergenzkurve von Q-learning gemittelt über 20 Läufe zu sehen. Auf der X-Achse ist die Höhe der noch ausstehenden Updates zusehen, auf der Y-Achse die Anzahl der ausgeführten Schritte. Erst ab Schritt 1670 sinkt der Wert unter 0.1 und ab Schritt 3840 unter 0.001.

[10] SARSA ist ein *on-policy* Algorithmus um Aktions-Wertefunktionen zu lernen. Das bedeutet dieser Algorithmus ist von der gegenwärtig verfolgten policy abhängig. Für weitere Informationen siehe Sutton and Barto [1998, Kapitel 6.4]

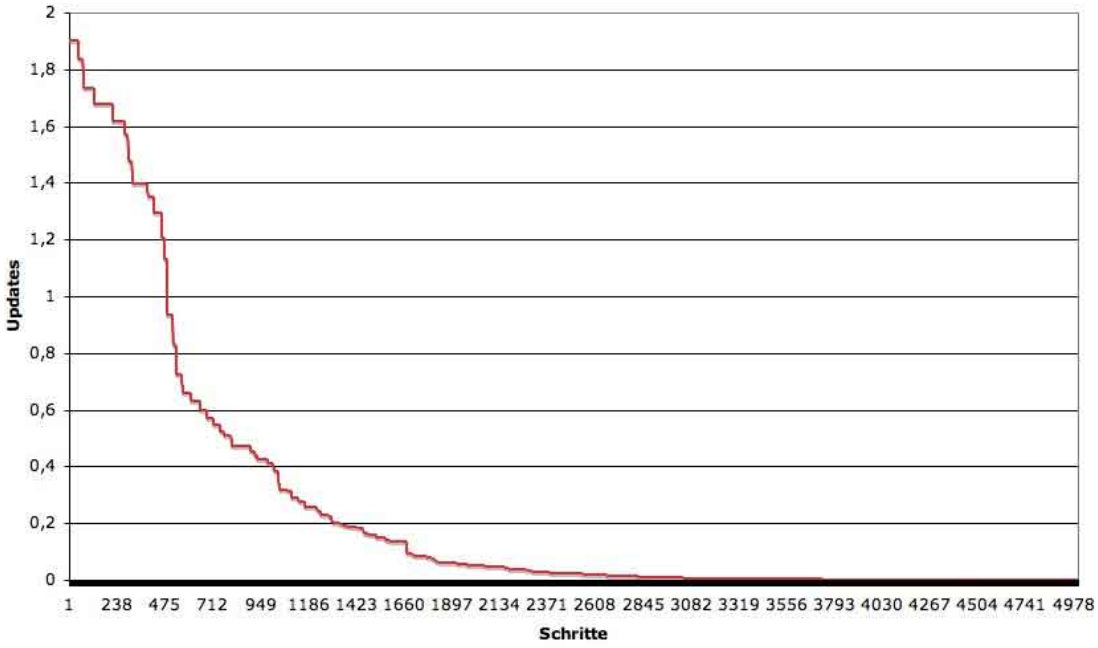

Abbildung 3.5: Konvergenzkurve für Q-learning in einer 5x5 Welt mit $\gamma = 0.9$

3.5 Q(λ)-learning

Normalerweise basiert ein TD-Update auf dem augenblicklichen Zustand und der Belohnung, die durch eine ausgeführte Aktion in diesem Zustand entsteht. Normalerweise sind jedoch auch Zustände und Aktionen, die den Agenten in diesen Zustand gebracht haben für diese Belohnung verantwortlich. Sie sollten also auch an der Belohnung teilhaben. Dies wird mit so genannten *eligibility traces (e-traces)* realisiert. Dabei bestimmt ein e-trace $e(s, a)$ wie hoch der Beitrag von Aktion a in Zustand s zur aktuellen Belohnung war. Das aktuelle TD-Update wird nun für jede Aktion mit ihrem e-trace gewichtet:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot tderror \cdot e(s, a) \quad \text{für alle} \quad s, a \tag{3.27}$$

Alle e-traces werden nach jedem Schritt um einem Faktor $\lambda\,[0, 1]$ abgeschwächt, da eine Aktion die weiter in der Vergangenheit ausgeführt wurde einen weniger hohen Beitrag geleistet hat, als eine erst kürzlich ausgeführte. Wird $\lambda = 0$ gesetzt, hat dies denselben Effekt, als würde man normales TD-learning verwenden. Man spricht daher bei SARSA und Q-learning ohne e-traces auch von $TD(0)$ Algorithmen. Der Parameter λ bestimmt ob sich der Algorithmus mehr wie normale Temporal-Difference Methoden ($\lambda = 0$) oder mehr wie Monte-Carlo Methoden ($\lambda = 1$) verhalten soll. Monte-Carlo Methoden werden in dieser Arbeit nicht behandelt, da sie das Ende einer Episode abwarten müssen. Dies ist bei einer kontinuierlichen Aufgabe, wie sie hier vorliegt, nicht möglich. Eine Beschreibung des Monte-Carlo Algorithmus ist in [Sutton and Barto, 1998, Kapitel 5] zu finden.

Es wird dabei zwischen 2 Arten von e-traces unterschieden:

- Akkumulierende e-traces:

$$e_t(s, a) = \begin{cases} \lambda\gamma e_{t-1}(s, a) + 1 & wenn \quad s = s_t, a = a_t \\ \lambda\gamma e_{t-1}(s, a) & sonst \end{cases} \tag{3.28}$$

- Ersetzende e-traces:

$$e_t(s, a) = \begin{cases} 1 & wenn \quad s = s_t, a = a_t \\ \lambda\gamma e_{t-1}(s, a) & sonst \end{cases} \tag{3.29}$$

Es gibt keine generelle Aussage darüber unter welchen Bedingungen welcher Ansatz bessere Resultate erzielt.

Die Implementation von e-traces für Q-learning gestaltet sich ein wenig aufwendiger als SARSA(λ). Die im vorherigen Kapitel erwähnte Eigenschaft, dass Q-

```
Initialize Q(s,a) arbitrarily and e(s,a) = 0, for all s, a
Repeat (for each episode)
  Initialize s, a
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q
    δ ← r + γQ(s',a') − Q(s,a)
    e(s,a) ← e(s,a) + 1
    For all s, a:
      Q(s,a) ← Q(s,a) + αδe(s,a)
      e(s,a) ← γλe(s,a)
    s ← s'; a ← a'
  until s is terminal
```

Listing 3.4: $SARSA(\lambda)$ Algorithmus

```
Initialize Q(s,a) arbitrarily and e(s,a) = 0, for all s, a
Repeat (for each episode)
  Initialize s, a
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q
    a* ← argmax_b Q(s',b)
    δ ← r + γQ(s',a*) − Q(s,a)
    e(s,a) ← e(s,a) + 1
    For all s, a:
      Q(s,a) ← Q(s,a) + αδe(s,a)
      If a' = a*, then e(s,a) ← γλe(s,a)
                  else e(s,a) ← 0
    s ← s'; a ← a'
  until s is terminal
```

Listing 3.5: Watkins's $Q(\lambda)$ Algorithmus

learning *off-policy* arbeitet bringt nun Probleme mit sich. Es ist nicht klar, was mit den e-traces nach einem Explorationsschritt geschehen soll. Watkins [Watkins and Dayan, 1992, Watkins, 1989] schlägt vor, alle e-traces nach einem solchen Explorationsschritt wieder auf 0 zurückzusetzen. Dieses Vorgehen ist das gebräuchlichste. Der Pseudocode zu Watkins's Q(λ) ist in Listing 3.5 zu sehen.

Eine Alternative zu Watkins's Q(λ) ist Peng's Q(λ). Es ist jedoch eine Mischung aus *on-policy* und *off-policy* Verfahren und es gibt keine Konvergenzgarantie für nicht greedy policies. Für weitere Informationen siehe [Peng and Williams, 1991, Sutton and Barto, 1998]

In Abbildung 3.6 ist das Ergebnis für Watkins's Q(λ) mit $\lambda = 0.7$, $\lambda = 0.8$ und $\lambda = 0.9$ zu sehen. Für das hier genutzte Testszenario scheint ein Wert von $\lambda = 0.8$ am besten zu sein.

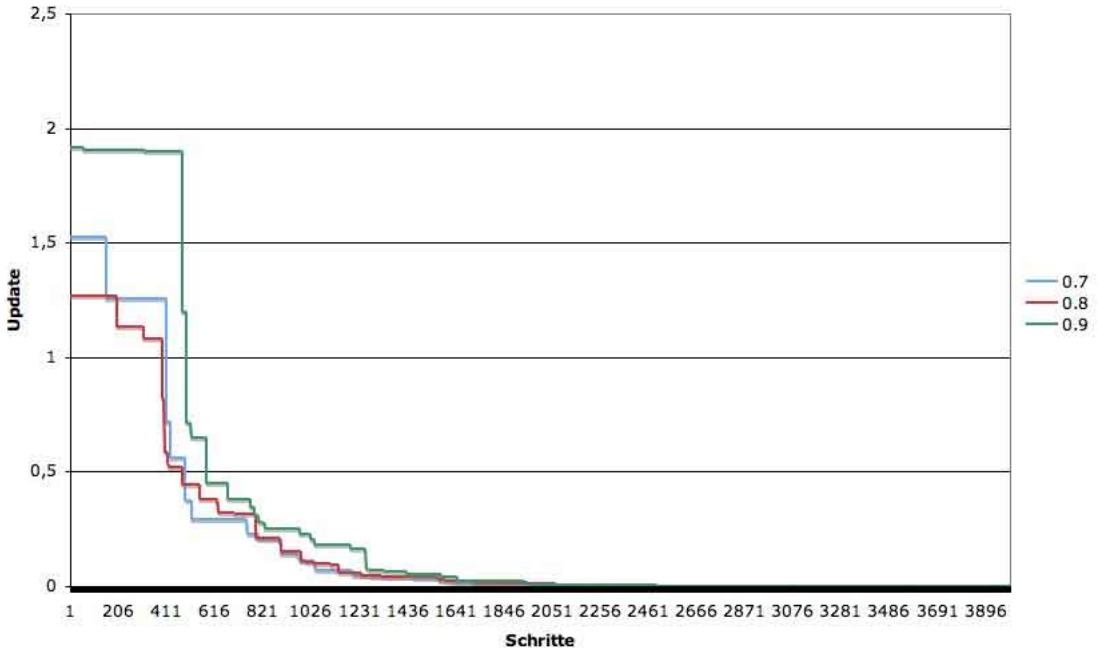

Abbildung 3.6: Konvergenzkurve für Watkins's Q(λ) mit $\lambda = 0.7$, $\lambda = 0.8$ und $\lambda = 0.9$

3.6 Q-learning mit Prioritized Sweeping

Prioritized Sweeping (PS) [Moore and Atkeson, 1993, Peng and Williams, 1993] nutzt anders als die anderen Algorithmen ein Modell seiner Umgebung. Dieses Modell entsteht nach und nach und wird durch die reale Interaktion mit der Umwelt aufgebaut. Nach jedem Schritt merkt sich der Agent die erhaltene Belohung, den Nachfolgezustand und den Vorgänger. Die Aktualisierung der Wertefunktion kann dann Anhand des Modells vorgenommen werden. So muss der Agent nicht bis zum nächsten Besuch des Zustandes warten, sondern kann den Wert anhand einer Simulation anpassen.

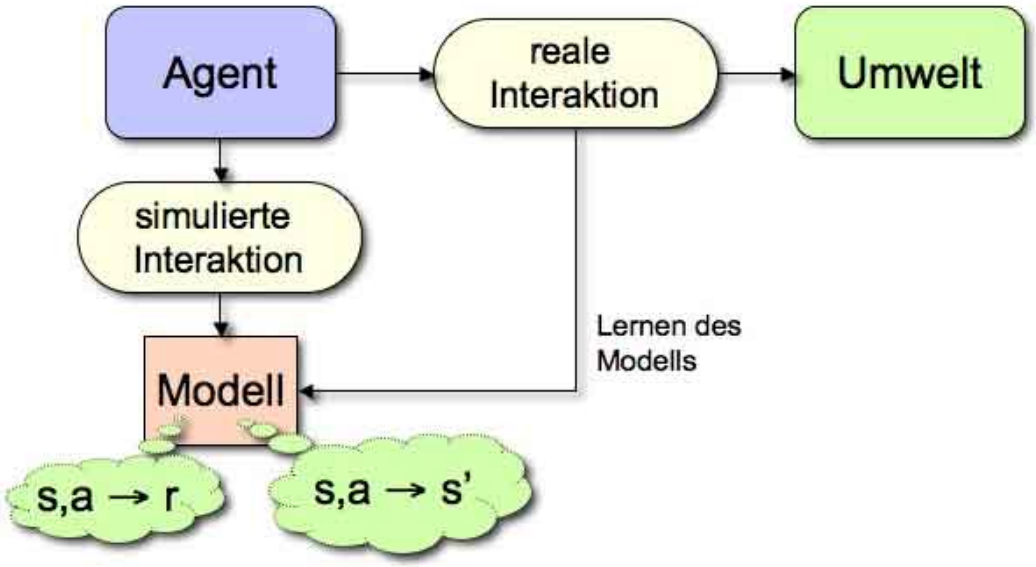

Abbildung 3.7: Der Agent und sein Modell der Welt

Der Algorithmus ist "prioritized", weil nicht alle Zustände nacheinander mit Hilfe der Simulation aktualisiert werden, sondern dies nach Dringlichkeit geschieht. Ändert sich der Wert eines Zustandes oder einer Aktion erheblich, so hat die Aktualisierung dessen Vorgänger Vorrang.

Nach jedem Schritt wird die Priorität p mit $p \leftarrow |Q(s,a)_t - Q(s,a)_{t+1}|$ ermittelt und das Zustands-Aktionspaar $< s, a >$ in eine Warteschlange eingefügt. Als nächstes wird das Paar mit der höchsten Priorität aus der Warteschlange entfernt und deren Wert aktualisiert. Anschließend werden auch alle Werte von Zustands-Aktionspaaren $< \bar{s}_i, \bar{a}_i >$ bearbeitet, die zu dem gerade aktualisierten Zustand führen. Dies geschieht Anhand des Modells des Agenten. Überschreiten die Änderungen der Vorgänger ein bestimmtes Limit δ, so werden auch diese Paare in die Warteschlange eingefügt.

In Listing 3.6 ist der Algorithmus für Prioritized Sweeping aus [Sutton and Barto, 1998, Kapitel 9.4] zu sehen. Diese Implementation ist jedoch nicht ganz opti-

```
1  Initialize Q(s,a), Model(s,a), for all s, a and PQueue to empty
2  Do forever:
3    s ← current (nonterminal) state
4    a ← policy(s,Q)
5    Take action a, observe r, s'
6    Model(s,a) ← s' and r
7    p ← |r + γ max_a' Q(s',a') − Q(s,a)|
8    if p > θ then insert s, a into PQueue with priority p
9    Repeat N times, while PQueue is not empty
10     s, a ← first(PQueue)
11     s', r ← Model(s,a)
12     Q(s,a) ← Q(s,a) + α[r + γ max_a' Q(s',a') − Q(s,a)]
13     Repeat, for all s̄, ā predicted to lead to s:
14       r̄ ← predicted reward
15       p ← |r̄ + γ max_a Q(s,a) − Q(s̄,ā)|
16       if p > θ then insert s̄, ā into PQueue with priority p
```

Listing 3.6: Q-learning mit Prioritized Sweeping (*Original Version*)

mal. So wird in Zeile 7 der *TD-Error* berechnet, jedoch nicht für eine Aktualisierung verwendet, sonder nur um die Priorität zu bestimmen. Die Aktualisierung erfolgt erst zu einem späteren Zeitpunkt (Zeile 12). So stützen sich Berechnungen der Simulation auf Aktionswerte, die noch nicht aktualisiert sind. Dieser "Fehler" fließt auch in die Berechnungen der Vorgängerzustände ein. Somit müssen mehr Aktualisierungen stattfinden als eigentlich notwendig wären. Deshalb wurde der Algorithmus in dieser Arbeit leicht abgeändert, wie in Listing 3.7 zu sehen. Die Aktionswerte werden sofort auf den neuesten Stand gebracht und erst danach in die Warteschlange eingefügt. Dies ist nicht nur für real ausgeführte Schritte der Fall, sondern auch für die Simulation. Dadurch wurde der *TD-Error* im Schnitt kleiner und es befanden sich weniger Elemente in der Warteschlange. Dies hatte eine kürzere Laufzeit und bessere Leistung zur Folge.

Der Algorithmus braucht nun zwar annähernd gleichviel Schritte wie das normale Q-Learning, jedoch fallen die simulierten Schritte im Model nicht ins Gewicht. Die tatsächliche Interaktion mit der realen Welt wurde deutlich reduziert. Denn in der Zeit, in der eine Aktion tatsächlich ausgeführt wird, können hunderte Schritte simuliert werden. In diesem Fall liegt der Engpass bei den Motorenbewegungen und nicht bei der Rechenleistung.

In Abbildung 3.8 ist das Verhalten des Algorithmus für $N = 5$, $N = 25$ und $N = 50$ zu sehen. Insbesondere gegen Ende erzielt die Variante mit 50 simulierten Schritten bessere Ergebnisse. Im Vergleich zu den anderen Algorithmen ist der *TD-Error* schon ab Schritt 150 unter 0.1 und ab Schritt 530 unter 0.001.

```
1  Initialize Q(s,a), Model(s,a), for all s, a and PQueue to empty
2  Do forever:
3    s ← current (nonterminal) state
4    a ← policy(s,Q)
5    Take action a, observe r, s'
6    Model(s,a) ← s' and r
7    δ ← r + γ max_a' Q(s',a') − Q(s,a)
8    Q(s,a) ← Q(s,a) + αδ
9    p ← |δ|
10   if p > θ, then insert s, a into PQueue with priority p
11   Repeat N times, while PQueue is not empty
12     s, a ← first(PQueue)
13     s', r ← Model(s,a)
14     Q(s,a) ← Q(s,a) + α[r + γ max_a' Q(s',a') − Q(s,a)]
15     Repeat, for all s̄, ā predicted to lead to s:
16       r̄ ← predicted reward
17       δ ← r̄ + γ max_a Q(s,a) − Q(s̄,ā)
18       Q(s̄,ā) ← Q(s̄,ā) + αδ
19       p ← |δ|
20       if p > θ then insert s̄, ā into PQueue with priority p
```

Listing 3.7: Q-learning mit Prioritized Sweeping (*Modifizierte Version*)

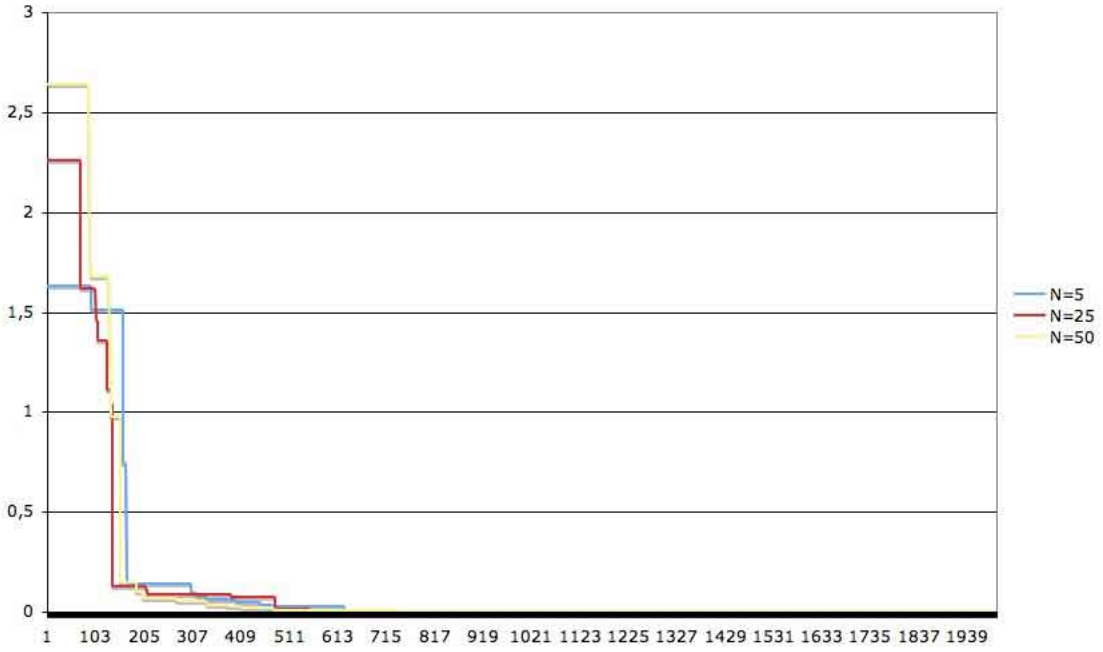

Abbildung 3.8: Konvergenzkurve für Q-learning mit Prioritized Sweeping für 5, 25 und 50 simulierten Schritte

Kapitel 4

Der vierbeinige Laufroboter

Als Lernverfahren auf dem vierbeinigen Roboter wird Q-learning mit Prioritized Sweeping eingesetzt, da dieses mit Abstand das beste hier getestete Verfahren ist. Der Roboter besitzt 4 Beine, wobei jedes Bein aus 3 der AX-12 Servos besteht (siehe Abbildung 4.1, Abbildung 4.2). Motor 1 ist für die Positionierung auf der horizontalen Achse verantwortlich. Die Motoren 2 und 3 platzieren das Bein auf der vertikalen Achse, werden jedoch als eine Einheit angesprochen. Ein einzelnes Bein kann somit durch die Eingabe X- und Y-Koordinaten positioniert werden.

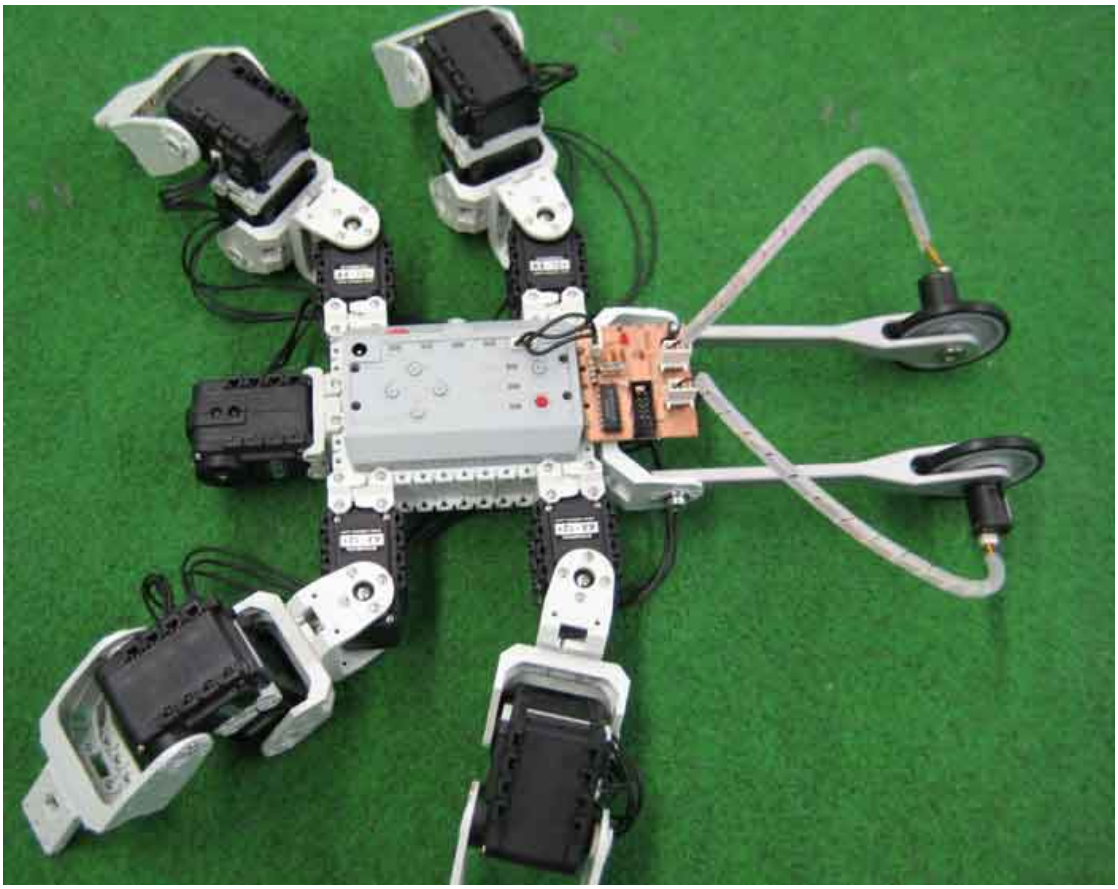

Abbildung 4.1: Der vierbeinige Laufroboter. Auf der linken Seite sind die beiden Sensoren für die Wegmessung zu sehen.

Ein Zustand des Roboters wird durch den Vektor

$$\vec{s_t} = ((x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4))^T \tag{4.1}$$

bestimmt, der die Position jedes einzelnen Beins repräsentiert. Eine Aktion ist analog dazu die nächste Position, die angesteuert werden soll.

$$\vec{a_t} = \vec{s_{t+1}} = ((x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4))^T \tag{4.2}$$

Die hier verwendeten Motoren könnten theoretisch 1024 unterschiedliche Positionen anfahren. Dies ergibt einen maximalen Zustandsraum von $1024^8 = 1.2089258196 \cdot 10^{24}$ unterschiedlichen Zuständen. Gewährt man dem Agenten maximale Flexibilität und lässt ihn aus einem Zustand heraus einen beliebigen anderen Zustand ansteuern, so ergeben sich $1.4615016373 \cdot 10^{48}$ unterschiedliche Zustands-Aktionspaare.

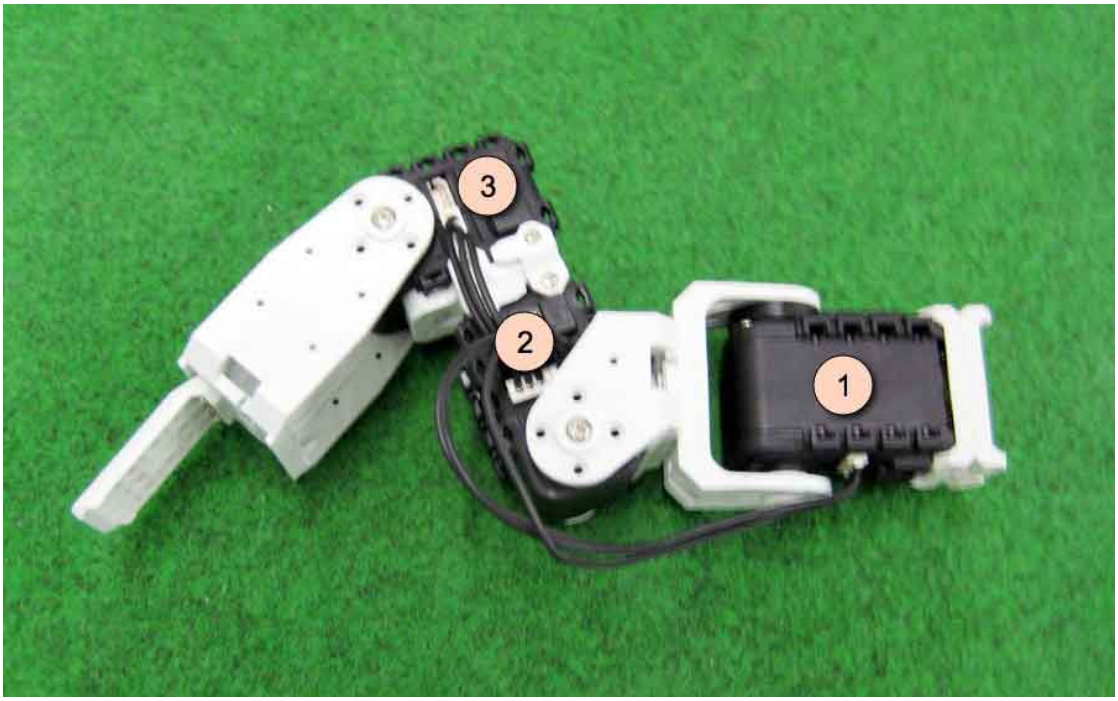

Abbildung 4.2: Ein einzelnes Bein mit 3 AX-12 Servos

In dieser Arbeit werden jedoch 3 Zustände und 3 Aktionen pro Bein verwendet. Das ergibt insgesamt $3^4 = 81$ Zustände und ebenfalls 81 mögliche Aktionen. Da in jedem Zustand jede der 81 Aktionen ausgeführt werden kann, macht das Zusammen 6561 Zustands-Aktionspaare. Beträgt die Ausführungszeit einer Bewegung 0,5 Sekunden, so braucht der Roboter etwa 55 Minuten um jede Bewegung einmal durchzuführen. Für eine deterministische Welt ist dies also die untere Schranke. Die Testresultate im vorherigen Kapitel zeigten jedoch, dass es nicht ausreicht, jede Aktion nur einmal auszuführen.

4.1 Resultate

Abbildung 4.3 zeigt die Fortschritte während des Lernprozesses. Am Anfang bewegt sich der Roboter kaum vorwärts und es werden zufällig Aktionen ausgeführt. Der Algorithmus konvergiert nach und nach zu einem Gang, bei dem immer die über kreuz liegenden Beine die gleiche Position einnehmen. Dieser Kreuzgang ist in Abbildung 4.4 dreidimensional dargestellt. Bein 1 und 4 sowie Bein 2 und 3 haben immer zur selben Zeit die gleiche Position. In Abbildung 4.5 sind die Positionen der einzelnen Motoren für diesen Gang eingezeichnet. In einem Diagramm sind die Motoren für die Position auf der X- und Y-Achse eines Beins eingetragen.

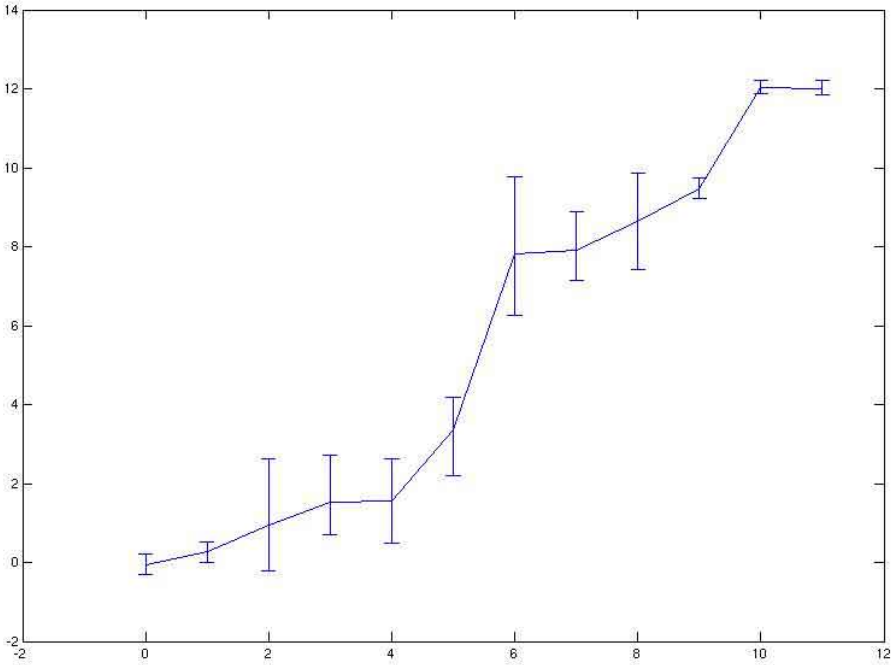

Abbildung 4.3: Durchschnittlich zurückgelegte Strecke des Roboters pro Schritt im Laufe des Lernprozesses.

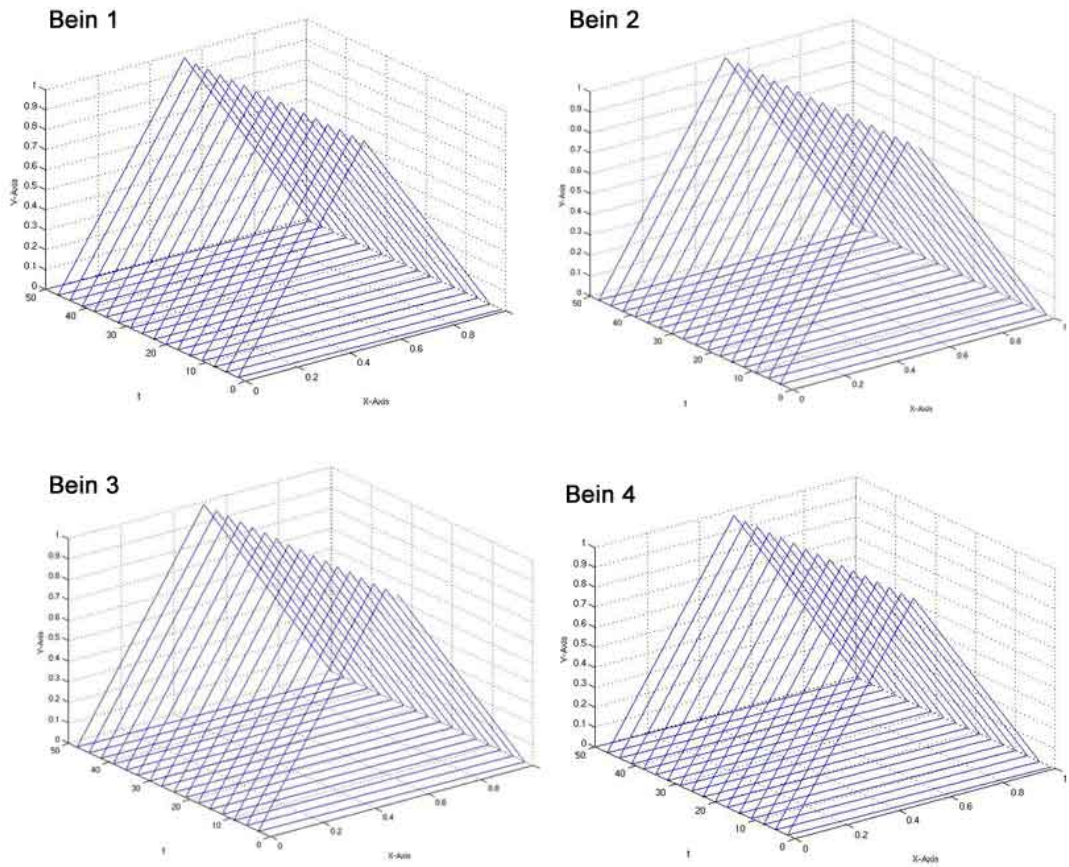

Abbildung 4.4: 3D-Visualisierung des Kreuzgangs

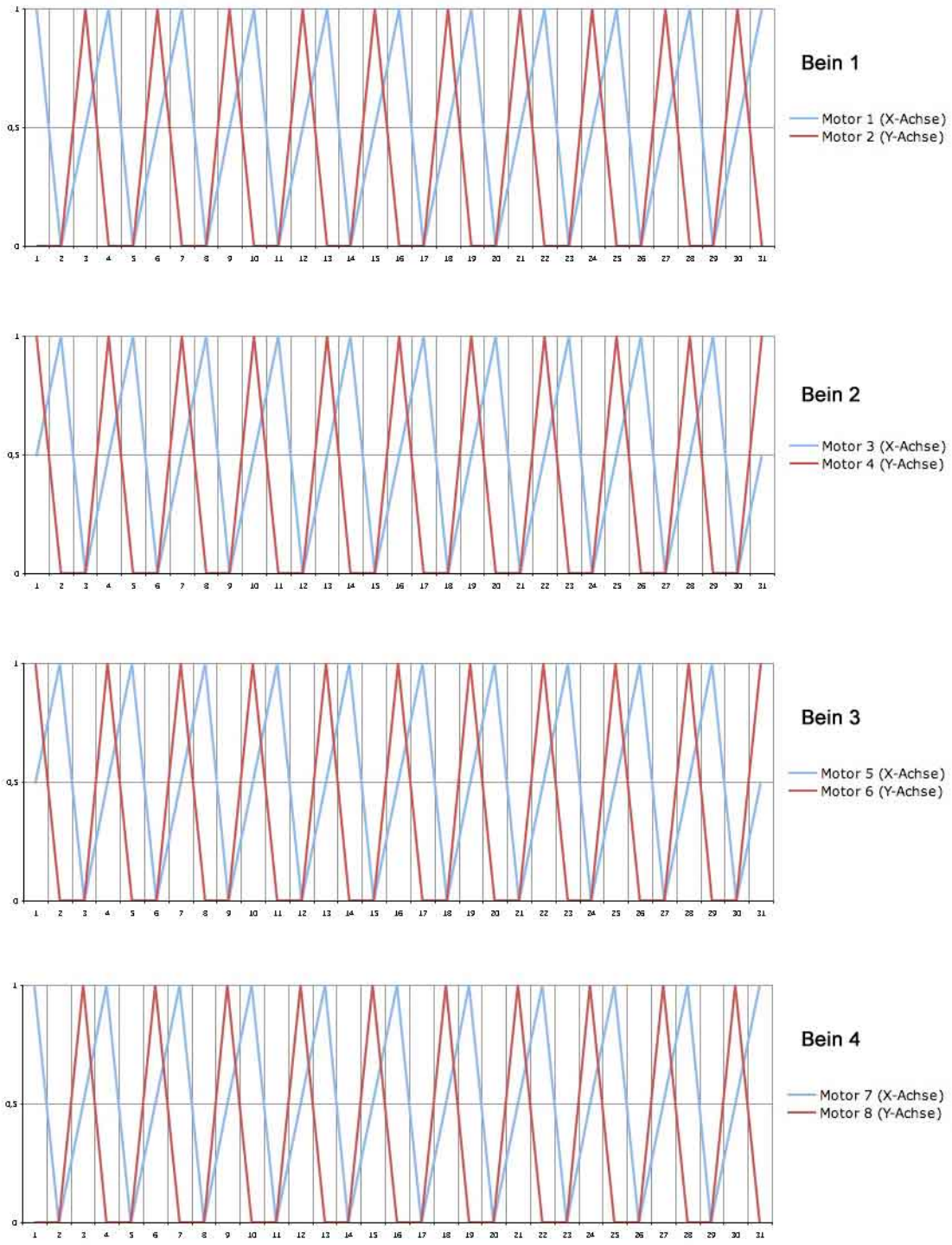

Abbildung 4.5: Der Kreuzgang mit 4 Beinen. Auf X-Achse sind die einzelnen Zeitschritte aufgetragen, auf der Y-Achse die Motorposition im Intervall $[0, 1]$

Kapitel 5

Software

5.1 Zustände und Aktionen

Zustände und Aktionen werden durch die Klassen *State* und *Action* repräsentiert. Beide Klassen sind Vektoren und können durch Rechenoperationen manipuliert werden.

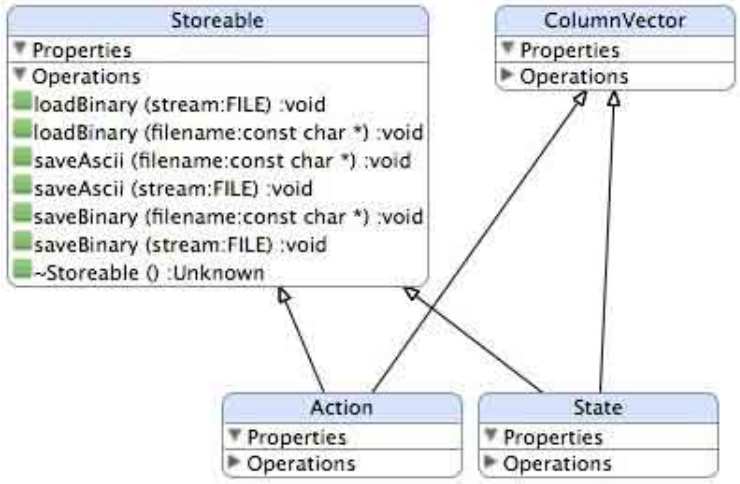

Abbildung 5.1: Zustände und Aktionen

5.1.1 ActionSet

Im diskreten Fall steht einem Agenten nur eine bestimmte Menge an möglichen Aktionen zur Verfügung. Diese Menge kann in einem *ActionSet* definiert werden. Alle Aktionen die der Agent ausführen kann müssen hier zunächst festgelegt werden. Ein *AktionSet* wird beispielsweise von einer *Policy* benötigt um eine passende Aktion zu finden.

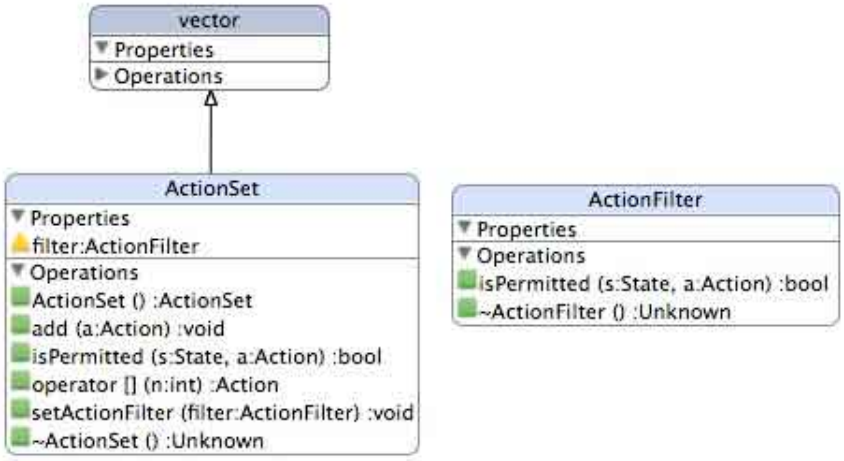

Abbildung 5.2: ActionSet

5.1.2 ActionFilter

In manchen Zuständen sind möglicherweise nicht alle Aktionen aus dem *ActionSet* erlaubt oder verfügbar. Dies ist beispielsweise der Fall, wenn sich der Arm des Agenten in der untersten Position befindet. In diesem Zustand ist es nicht möglich, den Arm noch weiter nach unten zu bewegen. Für diesen Fall kann dem *ActionSet* ein *ActionFilter* übergeben werden. Die Methode *ActionFilter::isPermitted(State, Action)* legt für jedes Zustands-Aktionspaar fest, ob es erlaubt ist oder nicht. Wird kein *ActionFilter* übergeben, so sind standardmäßig in jedem Zustand alle Aktionen erlaubt.

5.2 Die Umwelt

Die Umgebung des Agenten wird durch das Interface *Environment* spezifiziert. Ziel war es, eine universelle Schnittstelle für den Lernalgorithmus zu schaffen. Deshalb muss jede verwendete Umgebung dieses Interface implementieren. Für den Algorithmus spielt es anschließend keine Rolle mehr, ob es sich dabei um einen Simulator oder einen realen Roboter handelt.

Der Agent kann eine Aktion ausführen, die Belohung für diese Aktion abfragen und seinen aktuellen Zustand bestimmen. Belohnung und Zustand müssen nicht den tatsächlichen Werten entsprechen, sondern können fehlerbehaftet oder auch komplett falsch sein. Dies hängt von den Sensoren des Agenten ab.

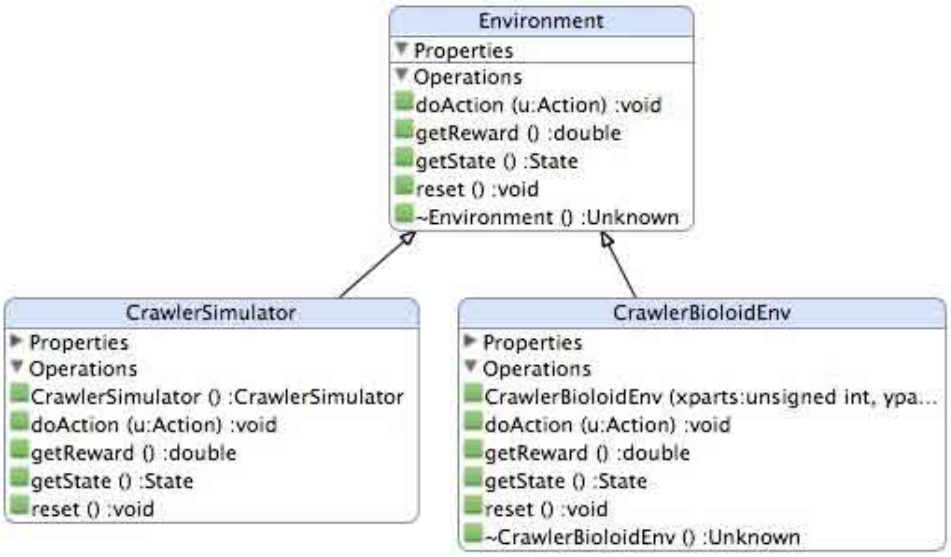

Abbildung 5.3: Die Umwelt

5.3 Die Wertefunktionen

Die Wertefunktionen sollten möglichst intuitiv zu benutzen sein. Dies wurde durch operator overloading realisiert. Sie sind als schneller assoziativer Speicher mit einer Hashmap realisiert.

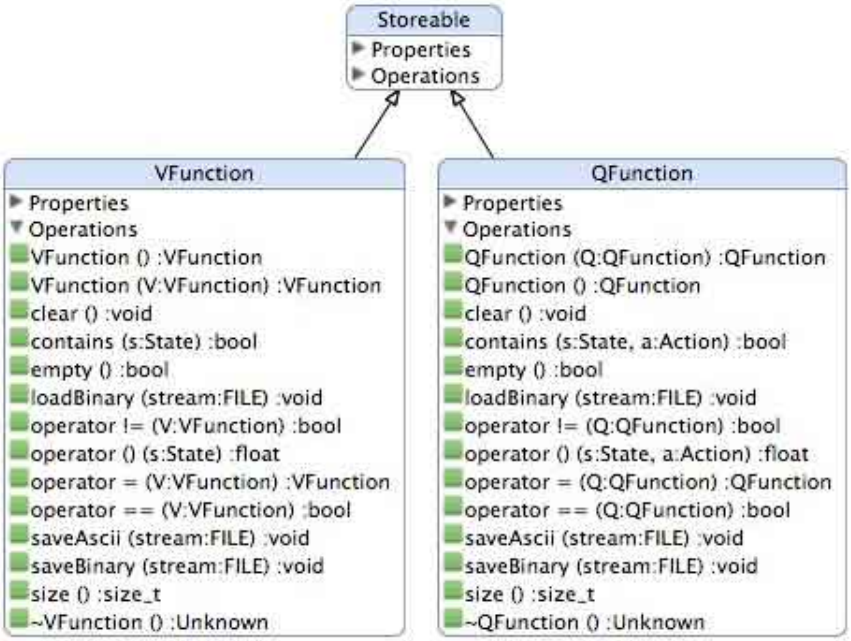

Abbildung 5.4: Die Wertefunktionen

```
1  State s;       // current state
2  Action a;      // current action
3  QFunction Q;   // action-value function
4  VFunction V;   // state-value function
5
6  // assign values
7  Q(s,a) = 10.0;
8  V(s)   = -5.2;
9
10 // print values
11 cout << Q(s,a);
12 cout << V(s);
```

Listing 5.1: Handhabung der Wertefunktionen

5.4 Die Policy

Die Aufgabe einer Policy ist es, in jedem Zustand eine Aktion zu wählen, die ausgeführt werden soll. Es existiert eine Schnittstelle *Policy*, von der jede neu definierte Policy abgeleitet werden muss. Außerdem muss die Methode *getNextAction()* überschrieben werden. Die Methode *getBestAction()* dieser Klasse liefert die Aktion mit dem höchsten Q-Wert in diesem Zustand. Existieren mehrere Aktionen mit gleich hohem Q-Wert, so wird zufällig eine aus diesen Aktionen gewählt.

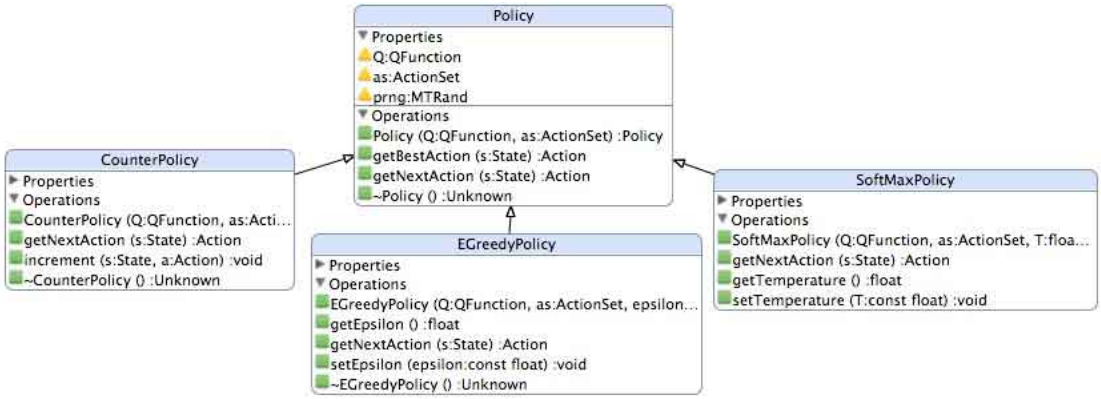

Abbildung 5.5: Die Policy

5.5 Eligibility Traces

Die eligibility traces (e-traces) sind in Form einer Liste implementiert. Es wäre jedoch Verschwendung von Rechenkapazität, wenn bei jeder Aktualisierung der Wertefunktion durch alle Zustände und Aktionen iteriert werden müsste, wie es in Listing 3.4 beschrieben ist. Deshalb werden in dieser Liste nur Einträge gespeichert,

```
Etrace e(0.01);  // create ETrace with threshold=0.01
e(s,a) = e(s,a) + 1; // increment trace for <s,a> by 1

// ... some code ...

// update all Q-Values
ETraceIterator it = e.iterator();
for(it.begin(); !it.end(); it++ ){
  State  s = it.state();
  Action a = it.action();
  Q(s,a) = Q(s,a) + alpha * gamma * e(s,a);
}

// discount etrace (traces < threshold will be deleted)
e *= lambda;
```

Listing 5.2: Handhabung der Eligibility Traces

die größer als ein minimaler Schwellwert sind. Einträge die nach der Abschwächung mit λ unter diesen Schwellwert fallen werden automatisch aus der Liste gelöscht. Die Liste kann mit einem *ETraceIterator* durchlaufen werden.

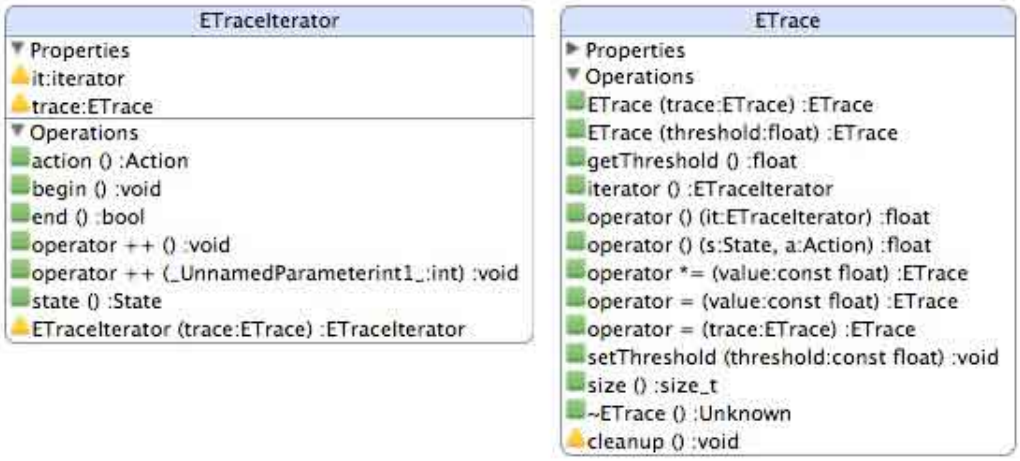

Abbildung 5.6: Eligibility Traces

5.6 Das Modell

Für Q-learning mit Prioritized Sweeping muss der Agent ein Modell seiner Welt aufbauen. Die Erfahrungen des Agenten werden in der Klasse *Model* gespeichert. Nach jedem Schritt des Agenten in seiner Umwelt, wird durch die Methode class-

Model::addExperience(State, Action, reward, nextState) das Weltenmodell erweitert. Dabei werden im Wesentlichen 3 Dinge gespeichert:

- Der Nachfolgezustand von *State* und *Action* ist *nextState*.
- Die Belohung, die auf diesen Zustandsübergang folgt.
- Ein Vorgänger von *nextState* ist *Action* in Zustand *nextState*. Dabei ist zu beachten, dass es mehrere Zustands-Aktionspaare gibt, die in denselben Zustand führen (Siehe Unterabschnitt 5.6.1).

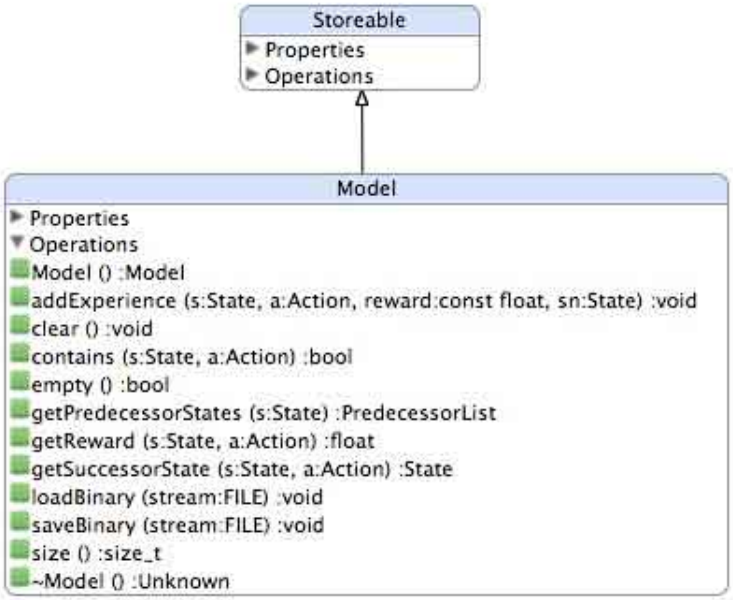

Abbildung 5.7: Das Modell

5.6.1 PredecessorList

In dieser Liste werden für einen Zustand s alle Zustands-Aktionspaare gespeichert, auf die dieser Zustand folgt.

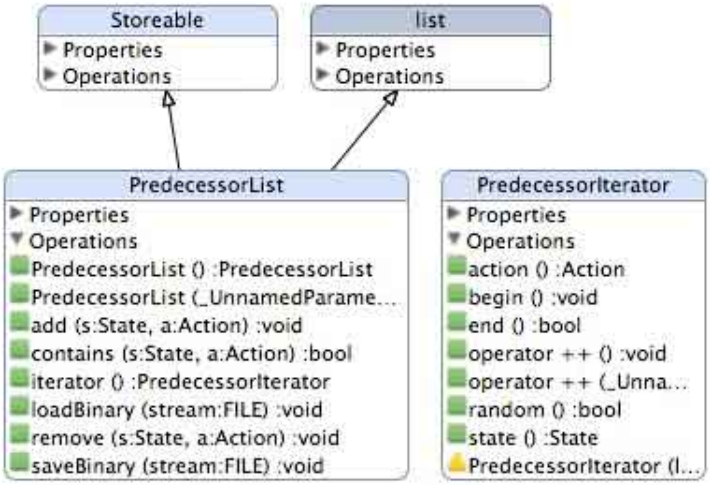

Abbildung 5.8: PredecessorList

```
1  // the world model
2  Model model;
3
4  // get state, perform action, observe reward and nextstate
5  // add the information to the world model
6  model.addExperience(s, a, reward, sn);
7
8  // query stored information
9  model.getReward(s,a);          // query reward
10 model.getSuccessorState(s,a) // query nextState
11
12 // get a list of <s,a> pairs leading to sn
13 PredecessorList list = model.getPredecessorStates(sn);
14 PredecessorIterator it = list.iterator();
15
16 // iterate through this list
17 // result will be <s,a>
18 for(it.begin(); !it.end(); it++){
19   cout << it.state();
20   cout << it.action();
21 }
```

Listing 5.3: Das Weltenmodell

5.7 Warteschlange mit Priorität

Die Klasse *Unique_PQueue* ist eine Liste von Zustands-Aktionspaaren, die nach ihrer Priorität sortiert ist. Wird ein Paar ein zweites Mal in die Liste eingefügt, so wird standardmäßig die Priorität des Eintrags auf den höheren der beiden Werte gesetzt. Das Zustands-Aktionspaar wird jedoch nicht nochmals in die Liste aufgenommen. Dieses Verhalten kann durch einen selbst definierten Functor überschrieben werden.

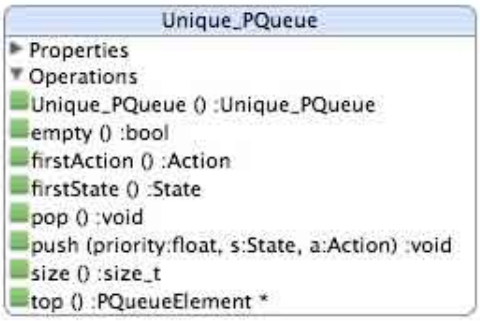

Abbildung 5.9: Warteschlange mit Priorität

Kapitel 6

Zusammenfassung

Wie in anderen Arbeiten gezeigt erzielt reinforcement learning für einige Aufgaben sehr beeindruckende Ergebnisse. Dies war Beispielsweise bei den Experimenten mit dem einbeinigen Krabbelroboter der Fall. Die Lernerfolge waren schon nach wenigen Schritten sichtbar und der Agent konnte sich binnen weniger Sekunden an neue Gegebenheiten in der Umwelt anpassen. Dies ist auch für die Standardbenchmarks wie *pole-balancing* oder *Gridworld* Aufgaben der Fall. Die Laufzeit erhöht sich jedoch dramatisch sobald auch nur etwas größere Zustandsräume benötigt werden. Die Situation wird noch wesentlich verschlechtert, sobald nicht mehr mit Simulationen sondern mit Hardware gearbeitet wird. Die real ausgeführten Bewegungen kosten sehr viel Zeit und die Sensordaten sind meistens verrauscht, wodurch noch mehr Interaktion benötigt wird. Dies macht eine schnelle Anpassung an die Umwelt nahezu unmöglich, zumindest mit den bisherigen Algorithmen. RL könnte jedoch sehr gut zur Optimierung bestimmter Bewegungsabläufe eingesetzt werden, sofern genug Trainingszeit vorhanden ist. Ein solches Verfahren wurde in [Kohl and Stone, 2004] benutzt um den Gang eines Sony Aibo Roboters mittels RL zu verbessern. Innerhalb von 3 Stunden war dieser schneller als jeder bis dahin von Menschenhand programmierte Gang. Es währe auch durchaus denkbar, zuerst in einer Simulation zu lernen und das Gelernte anschließend auf einen Roboter zu übertragen. Eine Simulation wird zwar nie den tatsächlichen Gegebenheiten entsprechen, jedoch kann sie helfen gute Startwerte für die policy zu finden. Die Erfahrungen aus der Simulation könnten dann auf dem realen Agenten weiter verbessert werden.

In den letzten Jahren gab es einige viel versprechende neue Entwicklungen von Seiten der Neuroinformatik und auch neue Ansätze für Policy Gradienten Methoden. Diese konnten leider aus Zeitmangel nicht mehr während dieser Arbeit getestet werden. Sie zeigen jedoch, dass wir das Potential auf diesem Gebiet noch lange nicht ausgeschöpft haben.

6.1 Ausblick

6.1.1 Funktionsapproximation

Um mehr Zustände und Aktionen möglich zu machen ist es notwendig mit Funktionsapproximation zu arbeiten. So könnte zwischen den einzelnen Zuständen interpoliert werden. Es könnte zum Beispiel lineare Funktionsapproximation wie etwa

Coarse Coding [Sutton, 1996], *Radial Basis Network* oder *Gaussian Softmax Basis Function Networks* [Doya, 1997] eingesetzt werden. Der Vorteil dieser Verfahren ist, dass sie nur lokalen Einfluss haben. Das bedeutet, das eine Aktualisierung der Wertefunktion von s nur die Werte der näheren Nachbarzustände ändert. Jedoch steigt die Anzahl der Möglichen Zustands-Aktionspaare weiterhin exponentiell mit der Anzahl der Zustände[1].

Es könnten auch *Feed Forward Neural Networks* [Coulom, 2002a,b] eingesetzt werden. Diese gehören zu den nichtlinearen Methoden und können daher auch mit sehr großem Zustandsraum umgehen. Sie werden jedoch nur selten für RL Aufgaben eingesetzt, weil es nur sehr wenige Konvergenzgarantien gibt. Da *Feed Forward Neural Networks* globalen Einfluss haben, kann eine Änderung eines Zustandswertes auch alle anderen Werte verändern, was meistens nicht wünschenswert ist. Weiterhin besteht die Gefahr, dass der Algorithmus in einem lokalen Minimum hängen bleibt.

6.1.2 Policy Gradienten Methoden

Die Klasse der Policy Gradienten Methoden arbeiten etwas anders als die bisher verwendeten Algorithmen. Normalerweise wird versucht eine Wertefunktion zu schätzen und anschließend eine policy[2] aus dieser abzuleiten.

Der Vorteil gegenüber anderen Algorithmen ist, dass die Policy Gradienten Methoden sehr gut mit stetigen Zustands- und Aktionsräumen klar kommen. Für Algorithmen, die Funktionsapproximation in solchen hochdimensionalen Räumen einzusetzen, gibt es nur sehr wenig Konvergenzgarantien. Es können sogar sehr triviale Beispiele mit nur 2 Zuständen gefunden werden, bei denen der Algorithmus oszilliert oder auch divergiert. Policy Gradienten hingegen haben, selbst wenn sie in Verbindung mit Funktionsapproximation genutzt werden, stärkere Konvergenzgarantien.

Die Policy Gradienten Methoden verzichten gänzlich auf eine Wertefunktion und versuchen die policy direkt darzustellen. Die policy π wird dabei durch einen Parametervektor $\vec{\theta}$ gesteuert. $J(\theta)$ dabei ein Maß für die Performanz der aktuellen policy π_θ

$$J(\theta) = E_\pi \left\{ (1-\gamma) \sum_{t=0}^{\infty} \gamma^t r_t \mid \theta \right\} \tag{6.1}$$

$$= \int_{\mathcal{S}} (1-\gamma) \sum_{t=0}^{\infty} \gamma^t Pr\{s_t = s\} \int_{\mathcal{A}} \pi(a \mid s) r(s,a) \mathrm{d}s\mathrm{d}a \tag{6.2}$$

Es wird versucht den Gradienten $\nabla_\theta J(\theta)$ zu schätzen und anschließend den Parametervektor $\vec{\theta}$ anhand dessen zu ändern. Eine ganze Gruppe von Algorithmen

[1] Im Englischen spricht man bei diesem Problem von *"the curse of dimensionality"*.
[2] Zum Beispiel benötigt eine greedy-policy die Q-Funktion um die beste Aktion zu wählen.

ist unter dem Namen REINFORCE bekannt. Sie wurden erstmals Williams [Williams, 1992] vorgestellt. Baxter entwickelte den Algorithmus GPOMDP, der den Gradienten für *partial observable Markov decision process (POMDP)* schätzt. Andrew Y. Ng und Michael Jordan nutzten ihren Algorithmus PEGASUS[3] [Ng and Jordan] um einen Helikopter auf dem Kopf zu steuern.

Ein sehr viel versprechender neuer Algorithmus, Natural Actor-Critic, wurde Peters, Vijayakumar und Schaal in [Peters et al., 2005] vorgestellt. Er verwendet nicht mehr den normalen Gradienten sondern ändert den Parametervektor anhand des natürlichen Gradienten. Dieser wird mit Hilfe der *Fisher information matrix* $G(\theta)$ berechnet:

$$\tilde{\nabla}_\theta J(\theta) = G^{-1}(\theta)\nabla_\theta J(\theta) \tag{6.3}$$

Durch die Verwendung des natürlichen Gradienten wird eine sehr viel schnellere Konvergenz erreicht.

6.1.3 Embedded System

Die Verbindung von Roboter und externem Rechner ist etwas störend. Sie ist jedoch im Moment noch notwendig, da Mikrokontroller des CM-5 zu wenig Speicher und Rechenleistung besitzt. Wünschenswert währe ein kleines Embedded System, wie es beispielsweise von der Firma *gumstix*[4] angeboten wird. Dieses könnte auf dem Roboter montiert werden und die Rolle des externen Rechners übernehmen.

6.1.4 Wireless

Eine wireless Verbindung könnte als Ersatz für das bisherige serielle Kabel benutzt werden. Sollte ein Embedded System als Rechner dienen könnte die kabellose Verbindung für Überwachungszecke und Visualisierung an einem externen Gerät eingesetzt werden.

6.1.5 Sensoren

Es müssten weitere Sensoren an den Roboter angebracht werden. Zum Beispiel währe es sinnvoll, auch die Seitwärtsbewegungen messen zu können. Möglicherweise sind auch Sensoren für Neigungswinkel und Beschleunigung für den Algorithmus von Nutzem.

[3] PEGASUS steht für *Policy Evaluation-of-Goodness And Search Using Scenarios*
[4] `http://www.gumstix.com`

Anhang A

Literaturverzeichnis

D. Andre, N. Friedman, and R. Parr. Generalized prioritized sweeping. NIPS, 1997.

Jonathan Baxter, Andrew Trigdell, and Lex Weaver. Knightcap: a chess program that learns by combining TD(λ) with game-tree search. In *Proc. 15th International Conf. on Machine Learning*, pages 28–36. Morgan Kaufmann, San Francisco, CA, 1998. URL citeseer.ist.psu.edu/article/baxter97knightcap.html.

Rémi Coulom. *Reinforcement Learning Using Neural Networks, with Applications to Motor Control*. PhD thesis, Institut National Polytechnique de Grenoble, 2002a. URL citeseer.ist.psu.edu/coulom02reinforcement.html.

Rémi Coulom. Feedforward neural networks in reinforcement learning applied to high-dimensional motor control. In Masayuki Numao Nicoló Cesa-Bianchi and Ruediger Reischuk, editors, *Proceedings of the 13th International Conference on Algorithmic Learning Theory*, pages 402–413. Springer, 2002b. URL citeseer.ist.psu.edu/coulom02feedforward.html.

Kenji Doya. Efficient nonlinear control with actor-tutor architecture. In Michael C. Mozer, Michael I. Jordan, and Thomas Petsche, editors, *Advances in Neural Information Processing Systems*, volume 9, page 1012. The MIT Press, 1997. URL citeseer.ist.psu.edu/doya97efficient.html.

Tommi Jaakkola, Satinder P. Singh, and Michael I. Jordan. Reinforcement learning algorithm for partially observable Markov decision problems. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 345–352. The MIT Press, 1995. URL citeseer.ist.psu.edu/jaakkola95reinforcement.html.

Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996. URL citeseer.ist.psu.edu/article/kaelbling96reinforcement.html.

N. Kohl and P. Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion, 2004. URL citeseer.ist.psu.edu/kohl04policy.html.

Andrew W. Moore and Christopher G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13:103–130, 1993.

URL citeseer.ist.psu.edu/moore93prioritized.html.

Andrew Y. Ng and Michael Jordan. PEGASUS:A policy search method for large MDPs and POMDPs. pages 406–415. URL citeseer.ist.psu.edu/ng00pegasus.html.

J. Peng and R. J. Williams. Efficient learning and planning within the dyna framework. In *Proceedings of the 2nd International Conference on Simulation of Adaptive Behavior, Hawaii*, 1993. URL citeseer.ist.psu.edu/peng93efficient.html.

Jing Peng and Ronald J. Williams. Incremental multi-step q-learning. Technical report, College of Engineering, University of California, Riverside, CA 92521, May 1991.

Jan Peters, Sethu Vijayakumar, and Stefan Schaal. Reinforcement learning for humanoid robotics. *Third IEEE-RAS International Conference on Humanoid Robots*, 2003.

Jan Peters, Sethu Vijayakumar, and Stefan Schaal. Natural actor-critic. In *the Proceedings of the 16th European Conference on Machine Learning*, 2005.

Doina Precup, Richard S. Sutton, and Satinder Singh. Eligibility traces for off-policy policy evaluation. In *Proc. 17th International Conf. on Machine Learning*, pages 759–766. Morgan Kaufmann, San Francisco, CA, 2000. URL citeseer.ist.psu.edu/precup00eligibility.html.

Sonia Seoane Puga. Development of a sensor module for a four-legged robot that learns to walk. Master's thesis, HS-Weingarten, 2007.

Dynamixel AX-12. Robotis, 2006a. URL http://www.tribotix.info/Downloads/Robotis/Dynamixels/AX-12(english).pdf.

Dynamixel AX-S. Robotis, 2006b. URL http://www.tribotix.info/Downloads/Robotis/Bioloid/AX-S1(english).pdf.

Bioloid User's Guide. Robotis, 2006c. URL http://www.tribotix.info/Downloads/Robotis/Bioloid/Bioloid%20User%27s%20Guide.pdf.

Jurgen H. Schmidhuber. Adaptive con dence and adaptive curiosity. Technical report, Technische Universitat Munchen, 1991.

Satinder P. Singh and Richard S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22(1–3):123–158, 1996. URL citeseer.ist.psu.edu/singh96reinforcement.html.

Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, 1990.

Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 1038–1044. The MIT Press, 1996. URL citeseer.ist.psu.edu/sutton96generalization.html.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning. An Introduction*. MIT Press, Cambridge, Massachusetts, 1998.

Sebastian B. Thrun. Efficient exploration in reinforcement learning. Technical Report CMU-CS-92-102, Carnegie Mellon University, Computer Science Department, Pittsburgh, January 1992.

Sebastian B. Thrun and Knut Möller. On planning and exploration in non-discrete environments. Technical report, GMD, Sankt Augustin, FRG, 1991.

Sebastian B. Thrun and Knut Möller. Active exploration in dynamic environments. Advances in Neural Information Processing Systems 4, 1992.

Michel Tokic. Entwicklung eines lernenden laufroboters. Master's thesis, HS-Weingarten, 2006.

Christoper Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, May 1989.

Christopher Watkins and Peter Dayan. Q-learning, 1992.

Steven D. Whitehead. Complexity and cooperation in q-learning, 1991a.

Steven D. Whitehead. A study of cooperative mechanisms for faster reinforcement learning. Technical Report 365, University of Rochester, Computer Science Department, Rochester, NY, March 1991b.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992. URL citeseer.ist.psu.edu/williams92simple.html.

Jeremy Wyatt. *Exploration and Inference in Learning from Reinforcement*. PhD thesis, University of Edinburgh, 1997.

Anhang B

Abbildungsverzeichnis

Anhang C

Tabellenverzeichnis

Anhang D

Listingverzeichnis

Anhang E

Quellcode

E.1 Die Handhabung der Dynamixel Klassen

```
/* ↩
   ********************************************************** ↩

 *    $Id$
 *
 *    AUTHOR: Markus Schneider
 *    CREATE: 2007-05-23
 *
 ********************************************************** ↩
    */

#include <stdio.h>
#include "bytelayer.h"
#include "lightserialport.h"
#include "dynamixel.h"
#include "AX12.h"
#include "AXS.h"

int main (int argc, char * const argv[])
{
  /* create bytelayer with LightSerialPortBL instance */
  ByteLayer *bl = new LightSerialPortBL("/dev/cu. ↩
     usbserial-FTCXCJ5T");

  /* log all activity to "bytelayer.log.txt" */
  bl = new ByteLayerLogger(bl, "bytelayer.log.txt");

  /* add bytelayer to dynamixel protocol stack */
  Dynamixel::setByteLayer( bl );

  /*
   * create object of AX12 with ID 1
   * and sensormodule with ID 100
```

```
30     */
31   AX12 *servo  = new AX12(1);
32   AXS  *sensor = new AXS(100);
33
34   /* move AX12 to position 512 */
35   servo->setGoalPosition(512);
36
37   return 0;
38 }
```

Listing E.1: Die Handhabung der Dynamixel Klassen

E.2 Quellcode Prioritized Sweeping

```
1  /*
2   *
3   *  psweeping.cpp
4   *
5   *
6   *  Created by Markus Schneider on 8/16/07.
7   *
8   */
9
10 Environment *env = new SpiderEnvironment();
11
12 float alpha = 1;       // stepsize parameter
13 float gamma = 0.90;    // discount factor
14 float theta = 0.01;    // threshold for pqueue
15 const int N = 500;     // number of simulated steps
16
17 State s;               // state
18 State sn;              // next state
19 Action a;              // action
20 Action an;             // next action
21 Action a_;             // best next action
22 float r;               // reward
23 QFunction Q;           // the Q-Function
24 Model model;           // the agents model of the
       environment
25 SpiderActionSet as;    // the actionset
26 Policy *policy = Policy(Q,as); // the policy to use
27
28 // the prioritized queue
29 Unique_PQueue<priority_adjuster_highest> PQueue;
```

```
30
31 // repeat for each step of episode
32 for(unsigned int steps=0;;steps++){
33
34   // get current state
35   s = env->getState();
36
37   // choose a from s using policy derived from Q
38   a = policy->getNextAction(s);
39
40   // take action a, observe r, sn
41   env->doAction(a);
42   r  = env->getReward();
43   sn = env->getState();
44
45   // get best policy (arg max_an)
46   a_ = policy->getBestAction(sn);
47
48   // add exerience to model
49   model.addExperience(s,a,r,sn);
50
51   // get priority of tderror
52   float tderror = r + gamma*Q(sn,a_) - Q(s,a);
53
54   // add <s,a> pair to queue
55   PQueue.push( fabs(tderror) ,s,a);
56
57   // update Q-Values
58   Q(s,a) = Q(s,a) + alpha*(r + gamma*Q(sn,a_) - Q(s,a));
59
60   // repeat N times, while PQueue is not empty
61   for(int i=0; i<N && !PQueue.empty(); i++){
62     // get first state from list
63     State s  = PQueue.firstState();
64     // get first action from list
65     Action a  = PQueue.firstAction();
66     // get next state
67     State sn  = model.getSuccessorState(s,a);
68     // get best next action
69     Action a_ = policy->getBestAction(sn);
70     // get reward
71     float r  = model.getReward(s,a);
72
73     // remove first element
74     PQueue.pop();
75
76     // get td-error
```

```
77      float tderror =  r + gamma*Q(sn,a_) - Q(s,a) ;
78
79      // update value
80      Q(s,a) = Q(s,a) + alpha*( r + gamma*Q(sn,a_) - Q(s,a ↩
            ) );
81
82      // if p > threshold, then insert s,a into PQueue ↩
            with priority p
83      if( fabs(tderror) > theta ){
84        PQueue.push( fabs(tderror), s, a);
85      }
86
87      // repeat, for all s^, a^ predicted to lead to s
88      PredecessorIterator it = model.getPredecessorStates( ↩
            s).iterator();
89      for(it.begin(); !it.end(); it++, i++){
90        // get predecessor state
91        State sp  = it.state();
92
93        // get predecessor action
94        Action ap = it.action();
95
96        // get best next action
97        Action a_ = policy->getBestAction(s);
98
99        // get predecessor reward
100       float rp  = model.getReward(sp,ap);
101
102       // calculate td-error
103       float tderror   = rp + gamma*Q(s,a_) - Q(sp,ap);
104
105       // update Q-Function
106       Q(sp,ap) = Q(sp,ap) + alpha*( rp + gamma*Q(s,a_) - ↩
              Q(sp,ap) );
107
108       // if p > threshold, then insert s,a into PQueue ↩
              with priority p
109       if( fabs(tderror) > theta ){
110         PQueue.push( fabs(tderror), sp, ap);
111       }
112     }
113   }
114 }
```

Listing E.2: Q-learning mit Prioritized Sweeping

Anhang F

Klassendiagramme

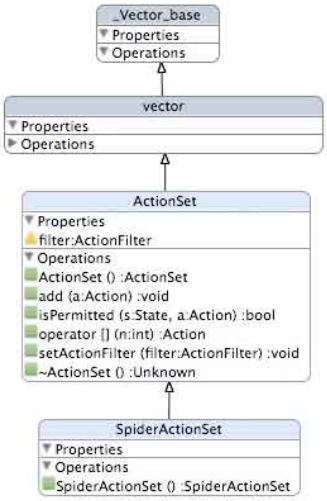

Abbildung F.1: ActionSet

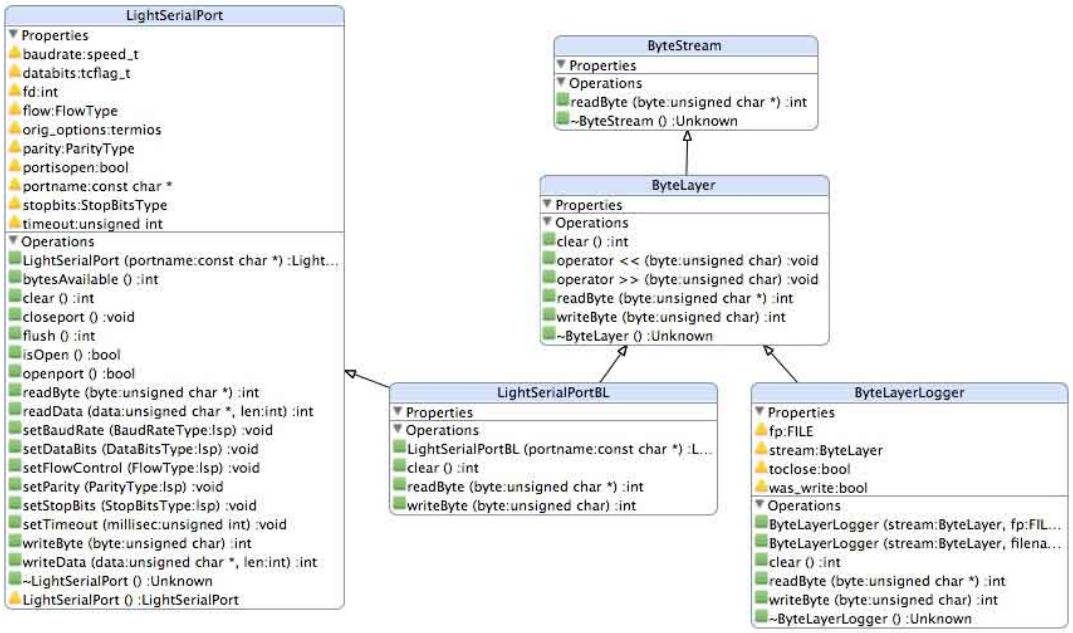

Abbildung F.2: ByteLayer

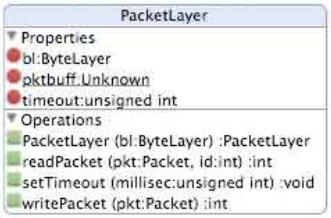

Abbildung F.3: PacketLayer

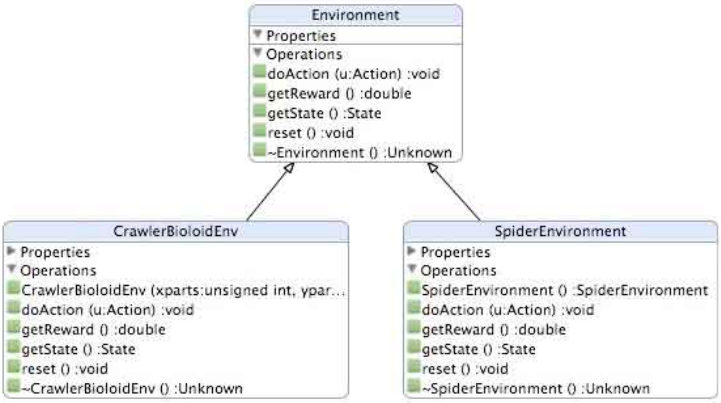

Abbildung F.4: Environment

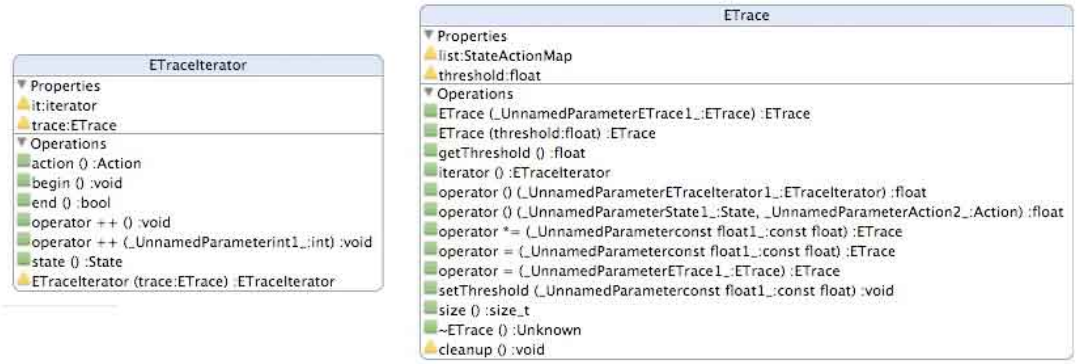

Abbildung F.5: Etrace

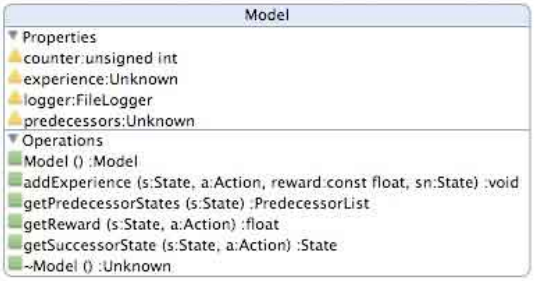

Abbildung F.6: Model

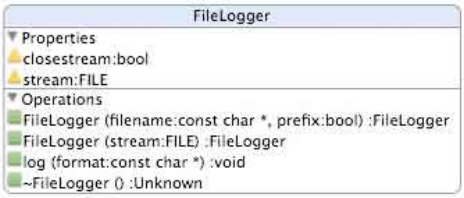

Abbildung F.7: FileLogger

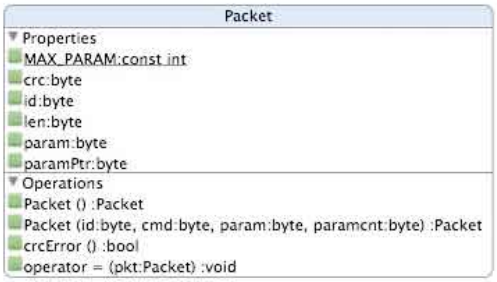

Abbildung F.8: Packet

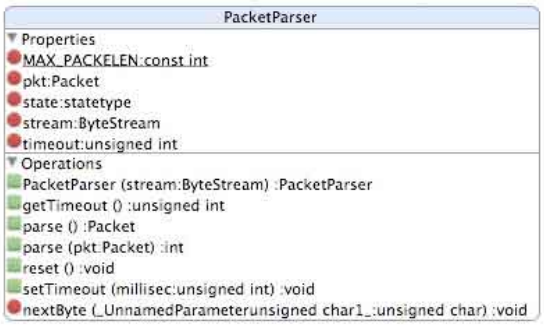

Abbildung F.9: PacketParser

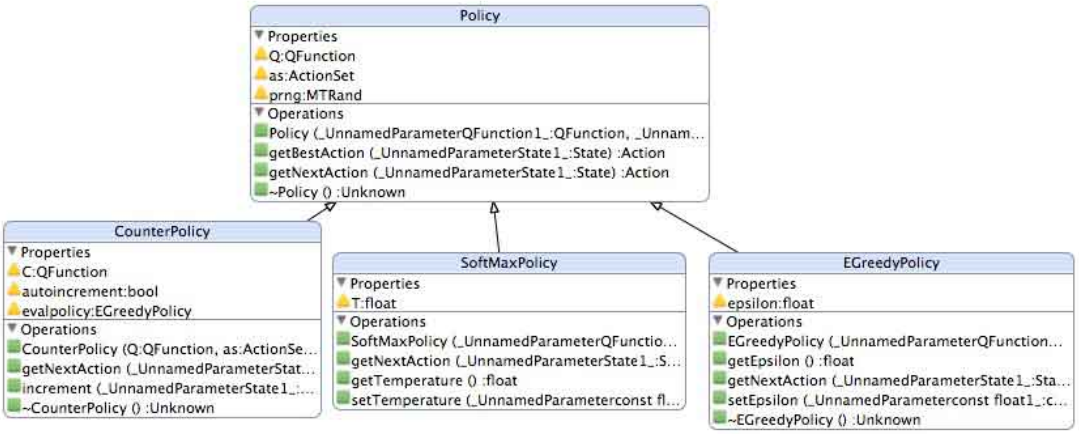

Abbildung F.10: Policy

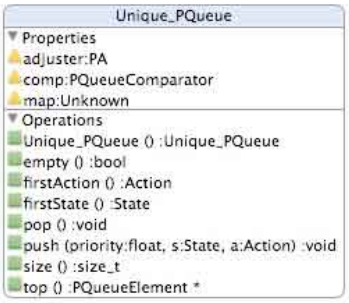

Abbildung F.11: UniquePQueue

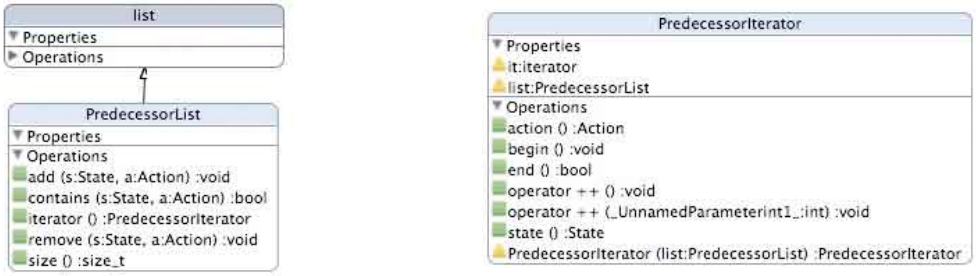

Abbildung F.12: Predecessor

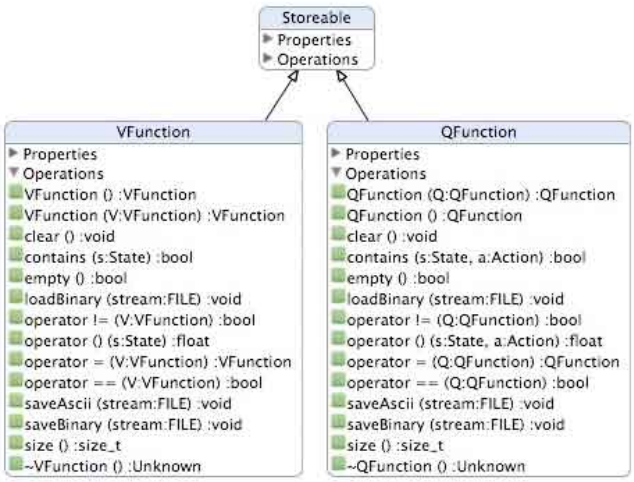

Abbildung F.13: QFunction

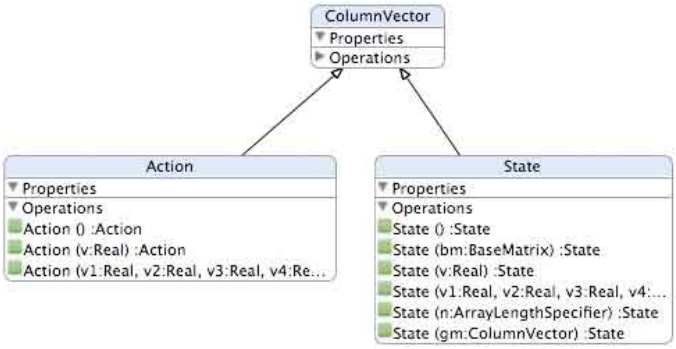

Abbildung F.14: State und Action

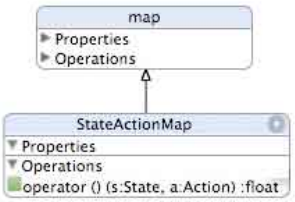

Abbildung F.15: StateActionMap

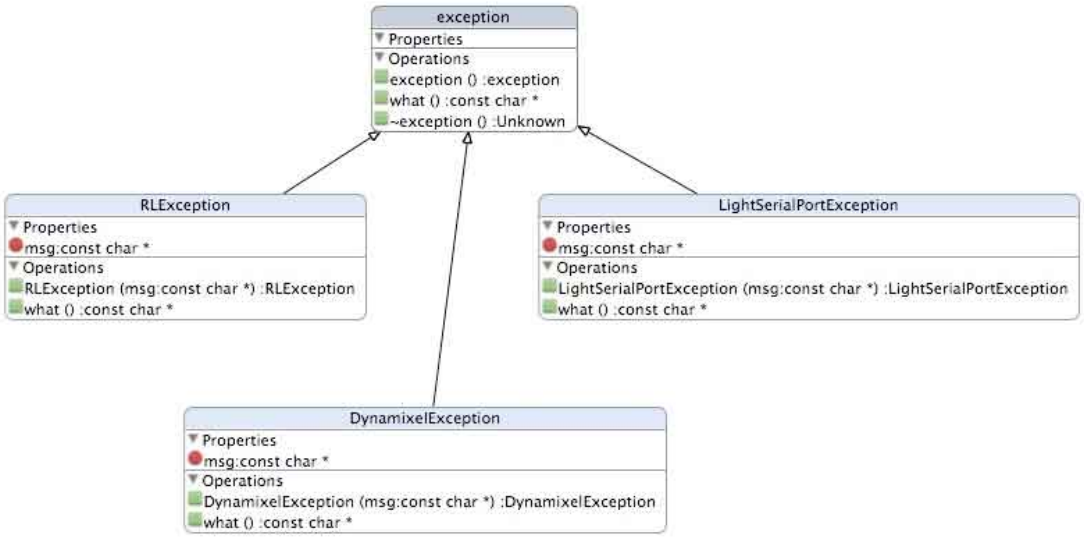

Abbildung F.16: Exceptions

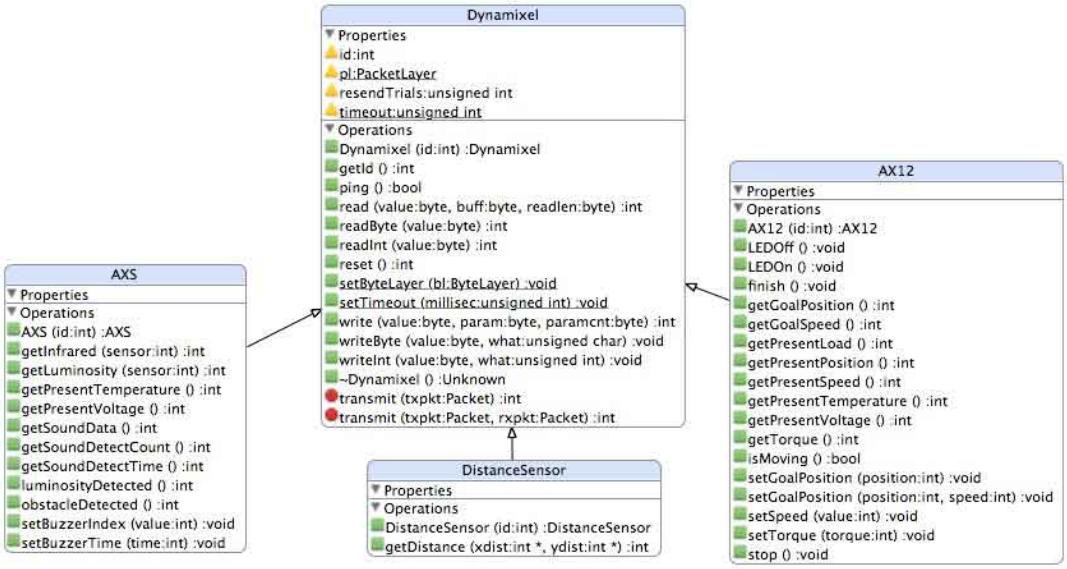

Abbildung F.17: Dynamixel