

Ich möchte Prof. Dr. W. Banzhaf für die interessante Themenstellung und Herrn Dipl.-Inform. Jens Ziegler für die aufmerksame Betreuung dieser Diplomarbeit danken.

Für die Diskussion zahlreicher Fragen des weitläufigen Themengebietes und diverse Anregungen gilt darüber hinaus besonderer Dank Dipl.-Inform. J. Busch, Dipl.-Inform. W. Kantschik und Dipl.-Inform. J. Niehaus.

Außerdem möchte ich der Firma Androtec, dem Distributor des EyeBot-Mikrocontrollerboards, und insbesondere Herrn Dipl.-Inform. Michael Kasper für den engagierten Support danken.

Inhaltsverzeichnis

1	Einleitung	1
2	Erweiterung der ZORC-Hardware	3
2.1	Die vorhandene Roboterarchitektur ZORC	3
2.2	Die Servopositions-Sensoren	5
2.3	Die Fußkontakt-Sensoren	8
2.4	Die Neigungs-Sensoren	8
2.5	Der Stromaufnahme-Sensor	10
2.6	Weitere Modifikationen	10
3	Vorbereitungen zum Einsatz der Simulationsumgebung	13
3.1	Das SIGEL-System	14
3.2	Modellierung von ZORC	15
3.3	Erweiterungen des SIGEL-Systems	19
3.3.1	Implementierung der Servomotoren	19
3.3.2	Implementierung der Pitch- und Roll-Sensoren	22
3.3.3	Implementierung des Kontakt-Sensors	23
3.3.4	Implementierung des Zweipunkt-Crossovers	24
3.3.5	Vorzeitiger Abbruch von Simulationen	25
3.3.6	Parallelisierung im Batchsystem	26
4	Die Evolution von Laufalgorithmen mit SIGEL	30
4.1	Vorstellung der eingesetzten Fitnessfunktionen	30
4.1.1	Die Fitnessfunktion “QuadHeight”	31
4.1.2	Die Fitnessfunktion “BigDistance”	32
4.1.3	Die Fitnessfunktion “BigDistance-2”	32
4.1.4	Die Fitnessfunktion “Stepper”	34
4.2	Ergebnisse mit dem simulierten ZORC-Roboter	35
4.2.1	Ergebnisse mit der Fitnessfunktion “QuadHeight”	37
4.2.2	Ergebnisse mit der Fitnessfunktion “BigDistance”	40
4.2.3	Ergebnisse mit der Fitnessfunktion “BigDistance-2”	43
4.2.4	Ergebnisse mit der Fitnessfunktion “Stepper”	46
4.2.5	Zusammenfassende Betrachtung der Simulationsergebnisse	48
5	Die Programmierung von ZORC	51
5.1	Überblick über die Implementierung von ZORCBRAIN	51

5.2	Implementierung des SIGEL-Subsystems	52
5.3	Implementierung der Hardwareabstraktionsklasse <code>CEyeBotLoLevel</code>	54
5.3.1	Ansteuerung der Servomotoren	55
5.3.2	Abfrage der Servoposition	57
5.3.3	Abfrage des Fußkontakts	57
5.3.4	Abfrage der Neigung	57
5.3.5	Abfrage der Stromaufnahme	58
5.3.6	Adressierung der Sensoren und Motoren	59
5.3.7	Die Klasse <code>ZORC_RS232Stream</code>	59
5.4	Die Klasse <code>CZorcBrain</code>	60
5.4.1	Die Testfunktionen von <code>CZorcBrain</code>	60
5.4.2	Ausführung von SIGEL-Programmen	60
5.4.3	Die Menüsteuerung	61
6	Übertragung und Evolution von Laufalgorithmen auf ZORC	63
6.1	Allgemeine Schwierigkeiten bei der Übertragung	63
6.2	Übertragbarkeit von SIGEL-Programmen auf ZORC	66
6.3	Übertragung der Simulationsergebnisse	67
6.4	Die fortgeführte Evolution auf ZORC	68
6.4.1	Motivation eines fortgeführten Lernprozesses	68
6.4.2	Steuerung von ZORC durch SIGEL	69
6.4.3	Die Fitnessfunktion “RemoteZorc”	69
6.4.4	Der “Laufstall”	75
6.4.5	Weitere technische Aspekte der Evolution auf ZORC	76
6.4.6	Die Evolution eines Laufalgorithmus auf ZORC	76
6.5	Vergleich der erlernten Laufalgorithmen	79
7	Ausblick	81
7.1	Die Zukunft von SIGEL	81
7.2	Die Zukunft von ZORC	83
A	Die Steckerbelegungen der EYEBOT-Ports	87
B	Der GP-Befehlssatz von SIGEL	89
B.1	LOAD: Register laden	89
B.2	COPY: Registerinhalte kopieren	89
B.3	ADD: Registerinhalte addieren	89
B.4	SUB: Registerinhalte subtrahieren	90
B.5	MUL: Registerinhalte multiplizieren	90
B.6	DIV: Registerinhalte dividieren	90
B.7	MOD: Modulo dividieren	90
B.8	MIN: Minimum bestimmen	90
B.9	MAX: Maximum bestimmen	90
B.10	CMP: Registerinhalte vergleichen	91
B.11	JMP: Bedingte Verzweigung	91

B.12	SENSE: Sensorwerte ermitteln	91
B.13	MOVE: Motoren bewegen	91
B.14	DELAY: Programmausführung verzögern	91
B.15	NOP: Leere Instruktion	91
C	Kurzanleitung zur SIGEL-Erweiterung	93
C.1	Motortyp SimpleServo	93
C.2	Sensortyp Pitch	94
C.3	Sensortyp Roll	94
C.4	Sensortyp Contact	95
D	Dokumentation der ZORCBRAIN-Klassen	97
D.1	ZorcBrain Class Documentation	98
D.1.1	CEyeBotLoLevel Class Reference	98
D.1.2	CZorcBrain Class Reference	106
D.1.3	tSensorTrait Struct Reference	111
D.1.4	tServoCtrlTabEntry Struct Reference	112
D.1.5	tServoTrait Struct Reference	113
D.1.6	ZORC_RS232Stream Class Reference	114
D.2	ZorcBrain File Documentation	116
D.2.1	CEyeBotLoLevel.h File Reference	117
D.2.2	CZorcBrain.h File Reference	120
D.2.3	ZORC_RS232Stream.h File Reference	120
	Literaturverzeichnis	121

Abbildungsverzeichnis

2.1	Frontansicht des ZORC-Roboters	4
2.2	Rückansicht des ZORC-Roboters	4
2.3	Geöffneter Servomotor	5
2.4	Aufbau einer Einheit des Relais-Routers	7
2.5	Schematische Darstellung des vollständigen Relais-Routers	8
2.6	Ein Fußelement von ZORC	9
2.7	Der Kopf mit den Neigungssensoren	9
2.8	Die Modifikation der Fußbasis	11
2.9	Die Akkulagerung auf ZORC	11
3.1	Bildschirmfoto des sigel_slave mit dem ZORC-Robotermodell.	16
3.2	Funktionsweise des Motortyps “SimpleServo”	21
3.3	Die Orientierung der Pitch- & Roll-Achsen am ZORC-Robotermodell	21
3.4	Klassendiagramm der neuen Sensorklassen	23
3.5	Sequenzdiagramm eines dynamischen SIGEL-Klienten	27
4.1	Bewegungsabläufe von zufällig erzeugten Programmen	36
4.2	Fitnessdiagramm für das “QuadHeight”-Experiment	37
4.3	Bewegungsablauf des besten “QuadHeight”-Individuums	39
4.4	Fitnessdiagramm für das “BigDistance”-Experiment	41
4.5	Instruktionsauswertung des besten “BigDistance”-Individuums	41
4.6	Der erfolgreiche Laufalgorithmus der Fitnessfunktion “BigDistance”	42
4.7	Fitnessdiagramm für das “BigDistance-2”-Experiment	45
4.8	Das beste Individuum des Experiments “BigDistance-2”	46
4.9	Fitnessdiagramm des “Stepper”-Experiments	48
4.10	Das beste Individuum der “Stepper”-Fitnessfunktion	49
5.1	UML-Diagramm des Programmpaketes ZORCBRAIN	52
5.2	Foto des EYEBOT mit dem ZORCBRAIN Programm	58
5.3	Diagramm der ZORCBRAIN-Menüstruktur	62
6.1	Quellcode eines SIGEL-Programms für den aufrechten Stand	66
6.2	Das Programm aus Abb. 6.1 in der Simulation	67
6.3	Das Programm aus Abb. 6.1 ausgeführt auf ZORC	67
6.4	Bildschirmfoto der GP-Parametereinstellungen in SIGEL	70
6.5	Der “Laufstall” mit Webcam	72

6.6	Testbild der in Abb. 6.5 gezeigten Kamera	73
6.7	Die “Laufstall”-Vorrichtung mit eingespanntem ZORC.	75
6.8	Das beste Individuum der Evolution auf ZORC	77
6.9	Instruktionsauswertung des “realen” Laufalgorithmus’	78

Tabellenverzeichnis

3.1	Dauer einer Fitnessberechnung	15
3.2	Übersicht über Dichte und Masse der ZORC-Glieder	18
4.1	Koza-Tableau des “QuadHeight”-Experiments	38
4.2	Koza-Tableau des “BigDistance”-Experiments	40
4.3	Koza-Tableau des “BigDistance-2”-Experiments	44
4.4	Koza-Tableau des “Stepper”-Experiments	47
A.1	Pinbelegung des EYEBOT-Digitalports	87
A.2	Pinbelegung des EYEBOT-Analogports	87
A.3	Pinbelegung der benutzten EYEBOT-Infrarotports	87

1 Einleitung

Die Programmierung von Laufrobotern stellt ein aufwändiges Unterfangen dar. Der klassische Ansatz zur Programmierung einer Bewegungssequenz erfordert die manuelle Spezifikation aller Teilschritte einer solchen Sequenz. Darüber hinaus ist es ggf. erforderlich, ein mathematisches Modell des Roboters zu entwerfen, um mittels inverser Kinematik die einzelnen Achsstellungen zu berechnen, die zum Erreichen einer Roboterposition erforderlich sind. Zudem müssen diese zeitintensiven Arbeitsschritte selbst für ein nur geringfügig modifiziertes Nachfolgemodell eines Roboters erneut durchgeführt werden. Der Aufwand einer manuellen Programmierung wird schließlich mit einer wachsenden Anzahl von Freiheitsgraden des Roboters zunehmend schwieriger und nicht zuletzt fehleranfälliger.

Im Rahmen dieser Diplomarbeit wurde daher die automatische Programmierung von Laufalgorithmen für eine humanoide Roboterarchitektur untersucht. Der Anteil des menschlichen Entwicklers an der Programmierung des Roboters beschränkt sich hierbei idealerweise auf die Angabe, welche Bewegung des Roboters erwünscht ist. Der Roboter erlernt anhand dieser Vorgaben selbständig, welche Programmierung zum gewünschten Ziel führt.

Die konkrete Aufgabenstellung bestand in der Entwicklung von Laufalgorithmen für den autonomen, humanoiden Roboter ZORC. Diese Roboterarchitektur entstand an der “Chalmers University of Technology” in Göteborg, Schweden. Dort war sie Gegenstand einer Diplomarbeit mit der Zielsetzung, eine autonome, an der humanoiden Anatomie orientierte Roboterarchitektur zu konstruieren [1][2]. Bei der Konstruktion des Roboters stand der Entwurf einer preisgünstigen Experimentierplattform im Vordergrund. Diese sollte der Erforschung von Steuerungsmechanismen dienen, die auf Genetischen Algorithmen [3] und insbesondere Genetischer Programmierung [4][5] basieren. Kapitel 2.1 geht auf die eingesetzte Roboterarchitektur ZORC näher ein.

Zunächst sollte der Roboter um Elemente erweitert werden, die für eine autonome Steuerung benötigt wurden. Im Zuge dieser Erweiterung waren insbesondere Sensoren für die aktuellen Gelenkstellungen, aber auch weitere Sensortypen zur Messung der Neigung im Raum und zur Erkennung von Bodenkontakt in die Roboterarchitektur zu integrieren. Diese Erweiterungen werden im Einzelnen in den Kapiteln 2.2 – 2.6 vorgestellt.

Der in dieser Form erweiterte Roboter diente schließlich als Vorlage bei der Erstellung eines maßstabsgetreuen Modells in einem CAD-Programm. Im weiteren Verlauf der Arbeit bildete dieses Modell die Grundlage für die Evolution von Laufroboteralgorithmen in einem Simulator. Hierzu sollte SIGEL eingesetzt werden [6][7], ein Simulationsprogramm für die Evolution von Laufrobotersteuerungsprogrammen mittels Genetischer Programmierung (GP). Das

1 Einleitung

Programm entstand als Ergebnis der Projektgruppe 369 am Lehrstuhl für Systemanalyse des Fachbereichs Informatik der Universität Dortmund.

Der ZORC-Roboter setzt einen Motortyp sowie mehrere Typen von Sensoren ein, die mit dem ursprünglichen SIGEL-System nicht modellierbar waren. Daher musste SIGEL für eine möglichst genaue Modellbildung um diese noch nicht vorhandenen Elemente erweitert werden. Diese Erweiterungen werden im Detail in Kapitel 3.3 beschrieben. Eine weitere Modifikation des ursprünglichen SIGEL-Systems war hinsichtlich der parallelisierten Fitnessberechnung auf dem Batchsystem am Lehrstuhl XI notwendig. Die Anpassung der Verteilung der Fitnessberechnungen an die automatische Rechenlastverteilung des Batchsystems beschreibt Kapitel 3.3.6.

Für die Evolution der Laufroboteralgorithmen wurden anschließend mehrere, auf die ZORC-Architektur zugeschnittene Fitnessfunktionen implementiert. Diese sind Gegenstand von Kapitel 4.1. Mit Hilfe dieser Fitnessfunktionen war das GP-Subsystem von SIGEL in der Lage, die simulierte Evolution zu steuern. Die mit den neuen Fitnessfunktionen durchgeführten Experimente und die damit erzielten Resultate werden in den Kapiteln 4.2.1 bis 4.2.5 vorgestellt.

Die in der Simulationsumgebung mit SIGEL evolvierten Laufalgorithmen sollten auf die reale Roboterarchitektur übertragen werden. Dazu wurden zunächst Teile des GP-Subsystems von SIGEL auf den Roboter portiert. Die Programmierung des Mikrocontrollers, der neben der unabhängigen Stromversorgung den autonomen Betrieb des Roboters ermöglicht, wird in Kapitel 5 beschrieben. Auf diese Weise wurde ZORC in die Lage versetzt, SIGEL GP-Programme auszuführen. Im Zuge dieser Ausführung konnten die GP-Programme die reale Roboterarchitektur steuern.

Die Resultate der schließlich auf den realen Roboter übertragenen Algorithmen werden in Kapitel 6.3 präsentiert. Im Anschluss daran geht Kapitel 6.4 auf eine fortgeführte Evolution der Laufalgorithmen auf der realen Roboterarchitektur ein.

Abschließend gibt Kapitel 7 einen Ausblick auf zukünftige Arbeiten.

2 Erweiterung der ZORC-Hardware

Die im Rahmen dieser Diplomarbeit eingesetzte Roboterarchitektur ZORC wird im folgenden Abschnitt zunächst vorgestellt. In den nachfolgenden Kapiteln werden die Erweiterungen des Roboters im Detail thematisiert.

2.1 Die vorhandene Roboterarchitektur ZORC

Bei ZORC handelt es sich um einen humanoiden Roboter, d. h. der Entwurf orientiert sich sowohl in den Dimensionen als auch in den Freiheitsgraden an der menschlichen Anatomie. Diese Architektur ist insofern von herausragender Bedeutung, da sie in einer an menschlichen Bedürfnissen orientierten Umgebung gegenüber den meisten anderen Roboterarchitekturen im Vorteil ist. Aber auch beim Einsatz in natürlichen Umgebungen, die in der Regel keine ebenmäßigen Oberflächen aufweisen, hätte ein humanoider Roboter Vorteile gegenüber Roboterarchitekturen, die sich beispielsweise auf Rädern fortbewegen.

Wie bereits in der Einleitung erwähnt wurde, entstand der Roboter im Rahmen einer Diplomarbeit an der Chalmers Universität, Göteborg [1]. Der zum Großteil aus PVC gefertigte Roboter hatte in der in den Experimenten eingesetzten Konfiguration¹ eine Größe von 37cm bei einem Gewicht von ca. 2000g. ZORC besitzt 13 Freiheitsgrade, angetrieben von 13 handelsüblichen Modellbau-Servomotoren. Eingesetzt wurden zwei Typen von Servomotoren. Während die stark belasteten Gelenke wie z. B. Knie und Fußgelenke entsprechend kräftige Servomotoren mit 78Ncm besitzen, verwenden die Arm und Schultergelenke² kleinere, weniger leistungsstarke Motoren mit 48Ncm Stellmoment [8]. Diese werden vom EYEBOT-Mikrocontrollerboard angesteuert. Mit Ausnahme einer an das EYEBOT-Board angeschlossenen Kamera, die bei den Experimenten im Rahmen der Diplomarbeit nicht eingesetzt wurde, verfügte der ursprüngliche Roboter über keinerlei Sensorik.

Der autonome Betrieb von ZORC wird durch das bereits erwähnte EYEBOT-Mikrocontrollerboard ermöglicht [9]. Es handelt sich dabei um ein speziell für den Einsatz in mobilen Robotern konzipiertes Mikrocontroller-Board. Dieses zeichnet sich neben einem einfachen Multitasking-Betriebssystem im ROM vor allem durch zahlreiche Schnittstellen aus. Außer digitalen und analogen Schnittstellen sind noch spezielle Anschlüsse für Gleichstrom- und

¹Durch die in 2.6 beschriebenen Modifikationen ist ZORC etwas größer als in der ursprünglichen Form.

²Die Beschreibung entspricht der letztendlich bei den Experimenten eingesetzten Konfiguration, d. h. nach den in Abschnitt 2.6 beschriebenen Modifikationen.

2 Erweiterung der ZORC-Hardware

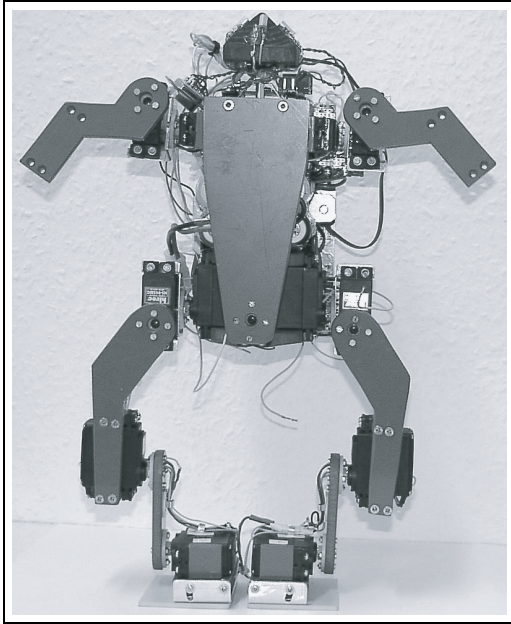


Abbildung 2.1: ZORC in der Frontansicht.

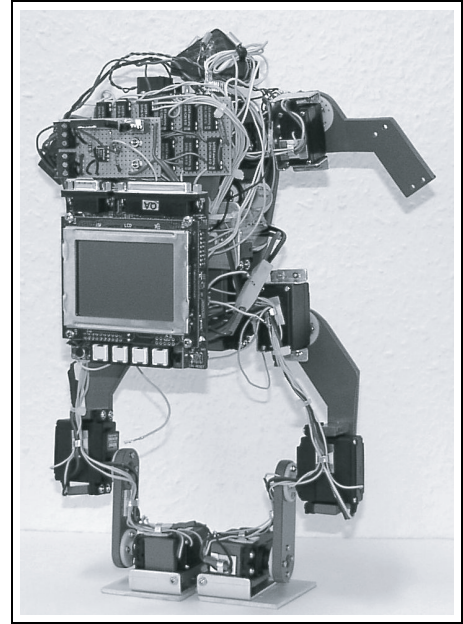


Abbildung 2.2: ZORC in der Rückansicht.

Servomotoren vorhanden. Zusätzlich sorgt eine Stromversorgung bestehend aus handelsüblichen Modellbauakkus für einen unabhängigen Betrieb von ZORC. Die autonome Energieversorgung kommt auch während der Experimente mit ZORC konsequent zum Einsatz, so dass der Roboter in jeder Hinsicht als autonom bezeichnet werden kann.

Über eine serielle Schnittstelle können Programme auf das EYEBOT-Board überspielt und dort ausgeführt werden. Die Programme für den Mikrocontroller werden in einer Entwicklungsumgebung bestehend aus einem Crosscompiler und diversen Werkzeugen auf einem LINUX-Rechner erzeugt. In den ersten Experimenten mit ZORC an der Universität Dortmund wurde eine eigens für den Roboter entwickelte Bibliothek mit diversen Funktionen zur Ansteuerung der Roboterhardware genutzt. Im Rahmen der Diplomarbeit wurde allerdings abgesehen von einer Servocontroller-Routine, die zum deutlich verlangsamten Anfahren der Servopositionen eingesetzt wird³, die Software zur Ansteuerung des Roboters von Grund auf neu entwickelt. Insbesondere wurde der Programmcode von C auf die objektorientierte Variante C++ umgestellt. Die Programmierung des EYEBOT ist Gegenstand von Kapitel 5.

Eine detaillierte Beschreibung der eingesetzten Roboterarchitektur in ihrer ursprünglichen Form findet sich unter anderem in [1] und [2].

³Das Anfahren der gewünschten Servoposition mit der maximal möglichen Geschwindigkeit führt durch die großen Beschleunigungen beim ruckartigen Anfahren und Abbremsen der Motoren oft zu instabilen Zuständen und zum Umfallen des Roboters. Aus diesem Grund fährt die Servocontroller-Routine die Endposition in Zwischenschritten an. Details zur Ansteuerung der Servomotoren erläutert Abschnitt 5.3.1.

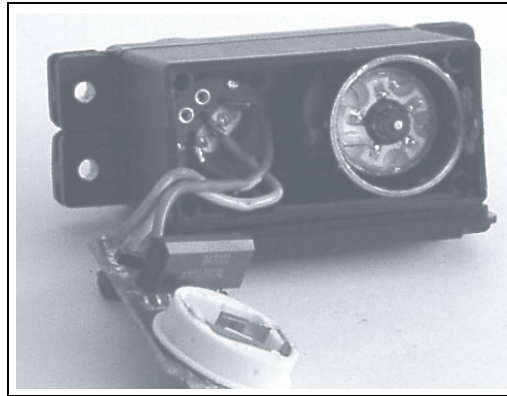


Abbildung 2.3: Blick ins Innere eines Servomotors. Das Potentiometer mit den 3 Anschlusskabeln ist in der linken Gehäusehälfte deutlich zu erkennen.

2.2 Die Servopositions-Sensoren

Die eingesetzten Servomotoren ermöglichten keine Rückmeldung über die aktuelle Stellung bzw. Position der Motorachse. Das hatte zur Folge, dass ein Fehlverhalten der Motoren vom Steuerungsprogramm nicht erkannt werden konnte. Für ein solches Fehlverhalten eines Servos können mehrere Faktoren eine Rolle spielen. Die Roboterglieder besitzen eine große Bewegungsfreiheit, so dass sie sich unter Umständen gegenseitig behindern können. Beispielsweise können sich bei entsprechenden Gelenkstellungen die Beine gegenseitig blockieren oder bei eingeknicktem Torso ein Arm die Bewegung eines Beines einschränken. Eine weitere Fehlerquelle sind fehlerhafte Kabelverbindungen. Während der Experimente mit ZORC trat wiederholt das Problem auf, dass einzelne Steuerleitungen der Servos sich von der Steckerleiste gelöst hatten und die betroffenen Servos keine Steuerbefehle mehr empfangen konnten.

Ohne eine Rückmeldung der Servoposition konnte der Roboter nur “blind” gesteuert werden. Die Software zur Steuerung des Roboters war daher lediglich in der Lage, vorgegebene Bewegungssequenzen abzuspielen und sich dabei einzig auf ein internes Modell des Roboters zu stützen, ohne die tatsächlichen Servopositionen zu kennen. Falls die befohlene Servoposition aus den erwähnten Gründen von der tatsächlichen Position eines Servos abwich, konnte daher nicht angemessen reagiert werden. In einer solchen Situation musste der Roboter zwangsläufig umfallen. In [1] konnten durch eine Kamera auf dem Roboter und mit Hilfe spezieller Umgebungsmarkierungen Rückschlüsse auf die Position des Roboters gezogen werden. Auf diese Weise sind jedoch kaum Rückschlüsse über die Stellungen der einzelnen Servomotoren möglich. Daher zählen die Sensoren zur Positionsmessung der Servomotoren zu den wichtigsten Erweiterungen der ZORC-Architektur.

Zur Winkelmessung der aktuellen Position der 13 Servomotoren wurden Potentiometer – Widerstände mit einem über eine Achse einstellbaren Widerstandswert – verwendet. Zunächst wurde geplant, die Potentiometer von Außen an den Servoachsen zu befestigen. Diese Variante hat sich jedoch nicht als praktikabel erwiesen, da sie mechanisch zu aufwändig gewesen wäre und zudem zu ungenaueren Messergebnissen geführt hätte als die letztendlich eingesetzte Methode. Die schließlich realisierte Positionsmessung setzt die Servomotoren selbst

2 Erweiterung der ZORC-Hardware

ein. Dabei wird die in jedem Servomotor vorhandene Steuerungselektronik genutzt, die zwei wesentliche Aufgaben erfüllt:

1. Dekodierung der über die Pulsbreite kodierten Stellbefehle (PWM-Kodierung)
2. Anfahren und Halten der Position

Die Steuerungselektronik setzt zur Positionskontrolle ein an die Motorachse gekoppeltes Potentiometer ein. Innerhalb des Servomotors wird diese Positionskontrolle für das Anfahren einer Position sowie für das aktive Halten dieser Position bei externer Belastung des Servomotors benötigt⁴.

Dieses bei allen eingesetzten Servomotoren vorhandene – und über die verschraubten Servo-Abdeckungen relativ problemlos (s. a. Abb. 2.3) zugängliche – Potentiometer wurde über ein zusätzliches Kabel an den A/D-Wandler des EYEBOT angeschlossen. Die auf diese Weise am Potentiometer gemessene Spannung steht im proportionalen Verhältnis zur aktuellen Stellung des Servomotors.

Bei der Messung der Spannungspegel an den auf die beschriebene Weise modifizierten Servomotoren trat in der Praxis allerdings ein Problem auf. Sobald die Spannung der Potentiometer abgefragt wurde, liefen die Servomotoren unruhig und hielten nicht länger die Position. Dieser Effekt lässt sich durch eine Beeinträchtigung der Servoelektronik durch die A/D-Wandlung erklären. Zur Spannungsmessung wird stets ein sehr geringer Strom von der zu messenden Spannungsquelle benötigt. Daher wird die Steuerungselektronik der Servos während der Messung geringfügig beeinflusst und somit in ihrer Funktion beeinträchtigt. Durch den Einsatz eines Operationsverstärkers⁵ vor jedem A/D-Kanal des EYEBOT konnte dieser Effekt kompensiert werden. Bei der eingesetzten Beschaltung mit einer Verstärkung vom Faktor 1 wurde der Spannungspegel des Servo-Potentiometers am Ausgang des Operationsverstärkers zur Verfügung gestellt. Bei einer A/D-Wandlung wurde dadurch anstelle der Servoelektronik der Operationsverstärker belastet, der ebenfalls über eine eigene Stromversorgung verfügt. Die Funktion der Servomotoren wurde somit nicht länger durch die Messungen beeinträchtigt.

Der auf dem EYEBOT eingesetzte AD-Wandler bietet lediglich 8 A/D-Kanäle, von denen zwei Kanäle bereits hardwareseitig fest verdrahtet sind⁶. Damit standen 6 Kanäle zur Verfügung, um die Messungen der 13 Servomotoren sowie weitere Sensorsignale zu verarbeiten. Um diese Limitation zu umgehen wurde eine Multiplexer-Schaltung eingesetzt, die im Folgenden als "Relais-Router" bezeichnet wird. Diese Erweiterung besteht aus 9 Reed-Relais, mit deren Hilfe sich insgesamt 12 Eingangssignale über lediglich 3 A/D-Kanäle messen lassen. Der Relais-Router setzt sich aus 3 identischen Einheiten zusammen, die jeweils 4 Eingangssignale auf

⁴Beispielsweise müssen die Servomotoren in den Knie-, Hüft- und Fußgelenken z. T. erhebliche Kräfte einsetzen, um die aktuelle Position zu halten. Insbesondere bei einseitiger Belastung tragen lediglich 2-3 Motoren das Gewicht des gesamten Roboters.

⁵Eingesetzt werden Präzisions FET-Input Operationsverstärker vom Typ AD820.

⁶Kanal 1 des Analogports ist fest verbunden mit dem eingebauten Miniatur-Mikrophon des EYEBOT, Kanal 2 überwacht die Batteriespannung.

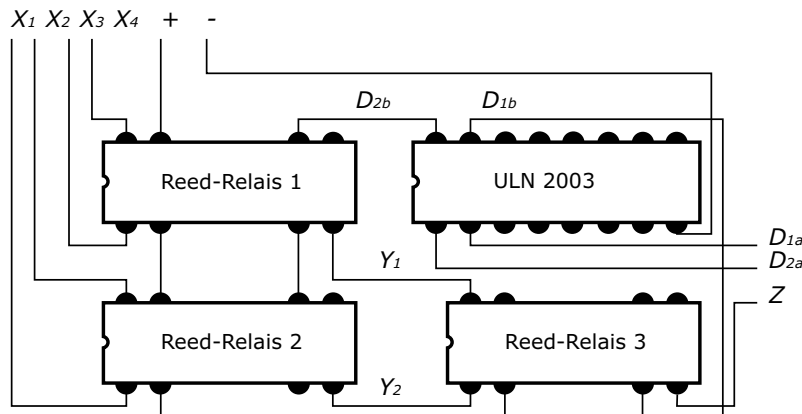


Abbildung 2.4: Schematische Darstellung einer Einheit des Relais-Routers. Mit einer solchen Einheit können 4 Eingangssignale einen Analogport des EYEBOT-Boards gemeinsam nutzen. Auf die Bezeichnungen sowie die Funktionsweise wird in Abschnitt 2.2 eingegangen.

einen A/D-Kanal lenken. Eine solche Einheit ist in Abbildung 2.4 schematisch dargestellt. Die Eingangssignale tragen in der Abbildung die Bezeichnungen $X_1 - X_4$. Kontrollsignale dienen der Steuerung der Relais und legen fest, welches der jeweils 2 Eingangssignale eines Relais weitergeleitet wird. Die Kontrollsignale tragen in der Abbildung die Bezeichnungen D_1 und D_2 (auf die Suffixe wird im folgenden Abschnitt eingegangen, für die Funktionsweise sind diese aber unerheblich). Die Relais 1 und 2 werden parallel über das Kontrollsignal D_2 geschaltet. Die auf diese Weise vorselektierten Signale Y_1 und Y_2 werden weiter zum Eingang von Relais 3 geleitet, an diesem wird schließlich mittels Kontrollsignal D_1 eines der Signale ausgewählt. Dieses in der Abbildung mit Z bezeichnete Signal ist direkt mit einem A/D-Kanal verbunden.

Der EYEBOT-Digitalport liefert die Kontrollsignale für die Relais-Router Einheiten. Da wie bereits erläutert je 2 Kontrollsignale pro Einheit benötigt werden, sind insgesamt 6 Kanäle des Digitalports für die Kontrolle des Relais-Routers zuständig. Die Belegung aller Pins des Digitalports ist in Tabelle A.1 aufgeführt, die Kontrollsignale für den Relais-Router tragen die Bezeichnungen $D_1 - D_6$. Da es sich bei den Reed-Relais um induktive Lasten handelt, die das direkt angeschlossene EYEBOT-Board unter Umständen schädigen können, wurden die Relais nicht direkt an den Digitalport angeschlossen. Stattdessen wurde eine integrierte Schaltung (IC) in der Funktion eines Treiber-Bausteins eingesetzt⁷. Die vom EYEBOT kommenden Steuersignale werden durch den IC an die Relais weitergeleitet. Durch diese Treiberstufe mit eigener Stromversorgung werden die Relais vom Digitalport entkoppelt. Jedes der 9 Reed-Relais benötigt ebenfalls einen Anschluss an die Stromversorgung, diese Anschlüsse sind in Abbildung 2.4 mit + und - ausgezeichnet.

In Abbildung 2.5 ist der vollständige Relais-Router dargestellt. Der Signalfluss der 12 über den Relais-Router geleiteten Signale ist dort im Überblick dargestellt. Sowohl die Kontrollsignale als auch die Stromversorgung wurden aus Gründen der Übersichtlichkeit nicht eingezeichnet. Die Abbildung zeigt außerdem, welche Kanäle des Digitalports die einzelnen Relais

⁷Es wurde ein Darlington-Array mit 7 Treibern vom Typ ULN2003 eingesetzt.

2 Erweiterung der ZORC-Hardware

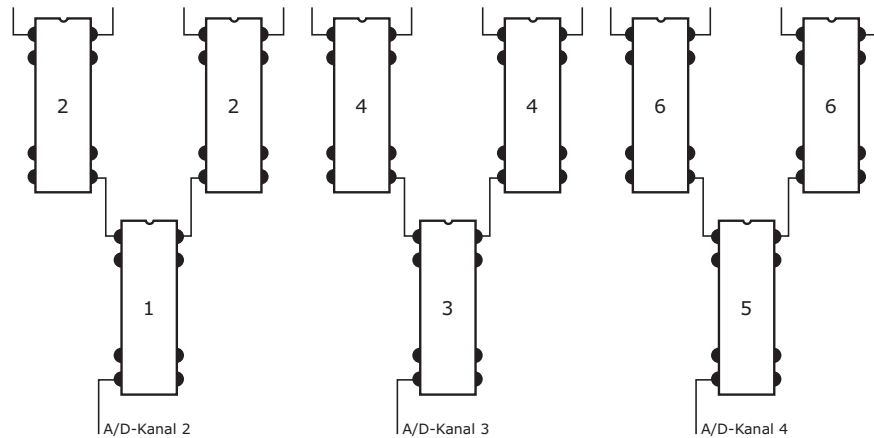


Abbildung 2.5: Der vollständige Relais-Router bestehend aus 3 der in Abb. 2.4 gezeigten Einheiten. Die vereinfachte Darstellung zeigt nur die Signalwege der 12 Eingangssignale mit den 3 benutzten A/D-Eingängen ohne die Kontrollsignale und Stromversorgung. Die Ziffern in den Relais bezeichnen das jeweilige Bit des Digitalports, welches das betreffende Relais ansteuert (s. a. Tabelle A.1).

schalten und in welche Kanäle des Analogports die Vierergruppen der Messsignale geleitet werden. Auf die Ansteuerung des Relais-Routers wird in Kapitel 5.3 eingegangen.

2.3 Die Fußkontakt-Sensoren

Ein weiterer Sensortyp wird eingesetzt, um festzustellen, ob ein Fuß gegenwärtig Bodenkontakt hat. Dazu wurden in die auf Abbildung 2.6 erkennbaren Hohlräume zwischen dem Fuß des Roboters und der beweglichen Fußklappe Miniaturtaster eingesetzt, die bei Belastung des Fußes einen Kontakt schließen. Diese Taster wurden mit dem Digitalport verbunden (s. a. Tabelle A.1) und setzen bei geschlossenem Kontakt die betreffenden Portbits auf 1. In Kapitel 5.3 wird erläutert, wie die Fußkontakt-Sensoren über die Software abgefragt werden können.

2.4 Die Neigungs-Sensoren

ZORC wurde mit der Zielsetzung entworfen, eine humanoide Roboterarchitektur zu konstruieren, die auf Grund ihrer Anatomie zu statischem Laufen befähigt ist. Dieses zeichnet sich im Gegensatz zum dynamischen Laufen dadurch aus, dass sich der Roboter stets in Balance befindet. Dazu muss sich der projizierte Schwerpunkt des Roboters immer innerhalb der Umriss des Fusses befinden, der Bodenkontakt hat. Der Neigungswinkel stellt aus diesem Grund insbesondere für humanoide Roboterarchitekturen einen kritischen Parameter dar. Demgegenüber unterliegen Roboterarchitekturen mit mehr als zwei Beinen weniger starken Beschränkungen bezüglich der Lage des Schwerpunktes, d. h. der Roboter muss weniger genau ausbalanciert werden um nicht zu Boden zu fallen. Für eine humanoide Roboterarchitektur könnte ein Sensor zur Erfassung des Neigungswinkels demzufolge ähnlich hilfreich

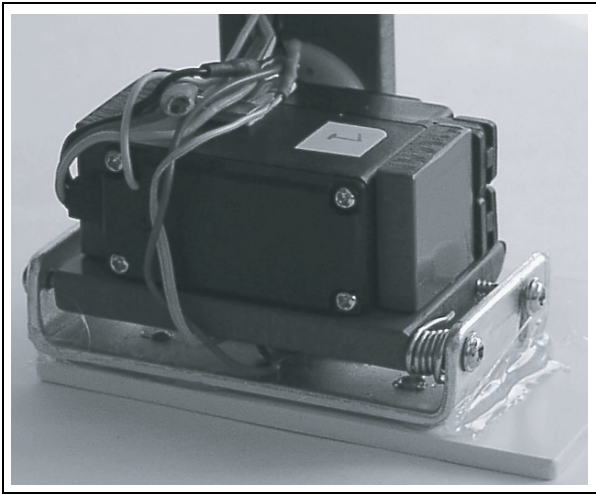


Abbildung 2.6: Ein Fußelement von ZORC. Zwischen beweglicher Fußklappe und Fuß befindet sich der Taster zur Erkennung von Bodenkontakt.

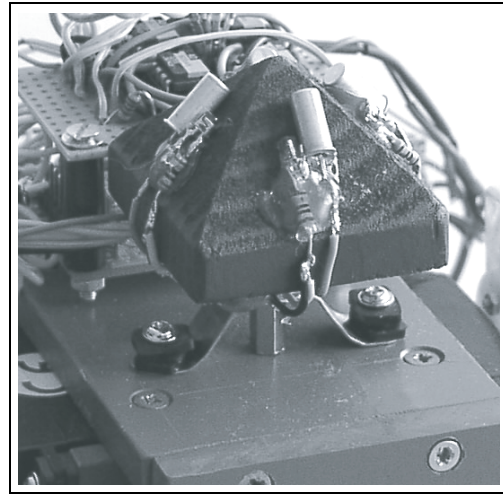


Abbildung 2.7: Der pyramidenförmige Kopf mit den Neigungssensoren.

sein wie der Gleichgewichtssinn beim Menschen.

Aus diesem Grund wurde der Einsatz eines Sensors zur Messung der Neigung des Roboters geprüft. Für eine solche Aufgabe sind Gyroskope bzw. Kreisel-Sensoren prädestiniert, die am Markt in zwei sehr unterschiedlichen Kategorien angeboten werden. Für den professionellen Einsatz gefertigte Gyroskope werden z. T. in hochwertigen mobilen Robotern wie [10] und [11] eingesetzt. Diese bieten eine hohe Präzision und sind auch in kleinen Dimensionen erhältlich, die den Einsatz auf ZORC zugelassen hätten. Eine Übersicht über diverse zum Zeitpunkt dieser Arbeit erhältliche Produkte mit Preisangaben gibt [12]. Da professionelle Gyroskope in einer Preisklasse von mehreren tausend Euro liegen, kamen diese für den Einsatz auf ZORC nicht in Betracht. Als zweite Kategorie standen Gyroskope für den Einsatz im Modellbau zur Wahl. Geräte dieser Hobby-Kategorie liegen in einer Preisklasse von mehreren Hundert Euro, allerdings hat auch dieser Gyroskoptyp zwei gravierende Nachteile. Alle erhältlichen Produkte sind für den direkten Anschluss von Servomotoren konzipiert, d. h. sie geben nur Steuerungssignale für die Motoren aus. Die gemessenen Neigungswinkel werden lediglich in der Elektronik der Geräte verarbeitet, sind aber nicht von außen zugänglich. Zudem besitzen die zu Hobbyzwecken produzierten Geräte eine mit der Zeit auftretende Messungenauigkeit (Drift), die um mehrere Größenordnungen über derjenigen der professionellen Gyroskope liegt.

Aus diesen Gründen wurde auf den Einsatz von Gyroskopen verzichtet. Stattdessen wurde eine äußerst einfach aufgebaute Vorrichtung konstruiert, die in Abbildung 2.7 dargestellt ist. Für diese wurden Neigungssensoren eingesetzt, die beim Überschreiten einer bestimmten Neigung einen elektrischen Kontakt schließen. Insgesamt 4 solcher Sensoren wurden auf den 4 Seitenflächen einer kleinen Pyramide angebracht. Die Seitenflächen wurden in einem Winkel von 45 Grad angeschrägt und auf die Front- und Seitenflächen des Roboters ausgerichtet. Das Bauteil wurde anstelle eines Kopfes auf dem Torso befestigt (s. a. Abb. 2.7). Auf diese

Weise schließt sich z. B. der auf der Rückseite befindliche Kontakt, sobald der Torso sich mit einem Winkel von mehr als 45 Grad nach vorne neigt. Die 4 Neigungssensoren liefern somit 4 binäre Messwerte, die rudimentäre Auskunft über die aktuelle Neigung des Torsos geben. Kapitel 5.3 geht auf die Behandlung der Neigungssensoren in der Software ein.

2.5 Der Stromaufnahme-Sensor

Mit den zum Zeitpunkt dieser Arbeit kommerziell verfügbaren Technologien zur unabhängigen Stromversorgung elektronischer Geräte ist das Problem der Kapazität noch immer unzureichend gelöst. Mobile Roboter sind davon in besonderem Maße betroffen: Einerseits soll eine möglichst lange Laufzeit erreicht werden, andererseits aber die Batterie so leicht wie möglich sein, da das zusätzliche Gewicht wiederum die Laufzeit verkürzt.

Der Einsatz von Genetischer Programmierung zur Evolution der Laufalgorithmen legt die Möglichkeit nahe, die Minimierung des Stromverbrauchs als Fitnesskriterium zu formulieren. Zu diesem Zweck wurde ein spezieller IC vom Typ MAX472 zur Präzisions-Strommessung eingesetzt, der den aktuellen Stromverbrauch des Roboters misst. Dieser gemessene Stromverbrauch wird vom IC als eine zum aufgenommenen Strom proportionale Spannung ausgegeben. Der Messbereich des MAX472 muss extern dimensioniert werden. Messungen des Stromverbrauchs von ZORC mittels Multimeter ergaben unter Situationen starker Belastung Spitzenwerte von bis zu 8 Ampere, daher wurde der Messbereich auf maximal 10 Ampere eingestellt. Bei dieser Beschaltung gibt der IC eine Spannung aus, die im Verhältnis von 0.2V pro Ampere steht. Diese Spannung wird über Kanal 6 des EYEBOT-A/D-Wandlers gemessen (s. a. Tab. A.2). Die Auswertung des Messwertes über die Software wird in Kapitel 5.3 beschrieben.

2.6 Weitere Modifikationen

Zusätzlich zu den beschriebenen Erweiterungen wurden einige kleinere Modifikationen an ZORC vorgenommen. In der ursprünglichen Konfiguration wurden für die Knie-Aktuatoren Servomotoren des Typs 48Ncm eingesetzt (s. a. Abschnitt 2.1). Unter Belastung konnten diese jedoch kaum die Last des Roboters tragen. Aus diesem Grund wurden die schwächeren Servomotoren gegen die Schulterservos des Typs 78Ncm ausgetauscht, da die an den Schultern zu erwartenden Belastungen als wesentlich geringer einzustufen sind.

Im Zuge dieses Austauschs wurden ebenfalls die Armelemente mit den Oberschenkeln des Roboters getauscht. Die ursprünglichen Oberschenkel erwiesen sich als zu schmal für die Befestigung der größeren Servomotoren des kräftigeren Typs. Die ehemaligen Armelemente waren im Vergleich zu den ursprünglichen Oberschenkeln um etwa 4cm länger. Aus diesem Grund ist die Größe von ZORC nach dem Austausch der Elemente um 4cm auf insgesamt 37cm angewachsen.

Die Fußelemente von ZORC wurden ebenfalls geringfügig modifiziert, um dem Roboter eine

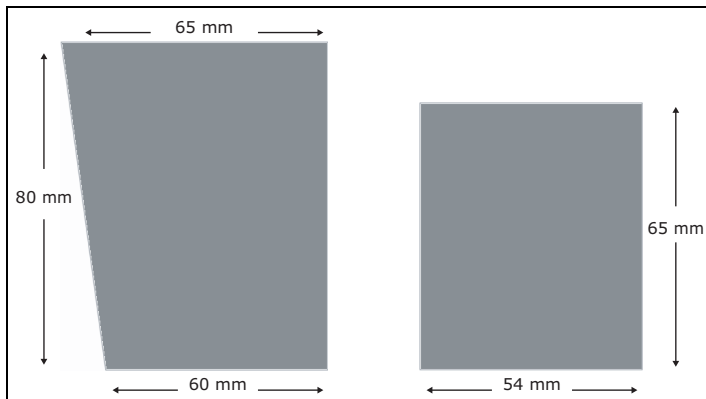


Abbildung 2.8: Die in Abschnitt 2.6 beschriebene Modifikation der Fußelemente. Links ist die neue und rechts ist die ursprüngliche Fußbasis zu sehen.

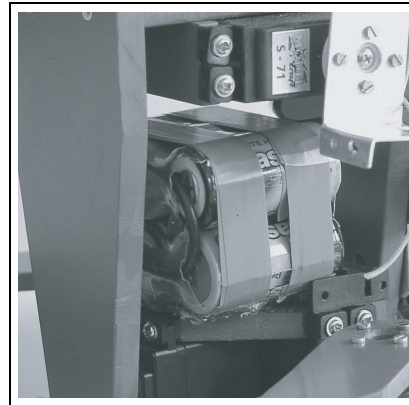


Abbildung 2.9: Die in Abschnitt 2.6 beschriebene Akkulagerung.

größere Standfläche zu geben und somit einen stabileren Stand zu ermöglichen. Wie Abbildung 2.8 illustriert, wurde die Fußbasis von ursprünglich $54 \times 65\text{mm}$ auf $65 \times 80\text{mm}$ vergrößert. In Anlehnung an die menschliche Anatomie verbreitert sich der neue Fuß zur Vorderseite hin geringfügig.

Am EYEBOT sind 12 Anschlüsse für die Ansteuerung von Servomotoren vorhanden. Da auf ZORC 13 Motoren eingesetzt werden, konnte die Ansteuerung aller Servomotoren nicht ausschließlich über das Mikrocontrollerboard erfolgen. Zur Ansteuerung der Arm- und Schultermotoren wird daher ein MINI-SSC II eingesetzt [13]. Dieses externe Modul wird über eine serielle Schnittstelle des EYEBOT kontrolliert⁸. Die Schnittstelle arbeitet dabei mit einer Variante des RS 232-Standard, dem so genannten “TTL-Level RS 232”. Diese Variante wird bevorzugt bei Geräten eingesetzt, denen keine negative Spannungsquelle zur Verfügung steht. Zu diesem Zweck werden Spannungspegel zwischen 0 und 5V zur Signalübertragung benutzt. Obwohl diese TTL-Level RS 232 Signalübertragung nicht offiziell genormt ist, hat sich bei den Herstellern ein “de facto”-Standard herausgebildet. Bei diesem werden die -12V des RS 232 Standard auf 0V und analog die 12V auf 5V abgebildet. Der EYEBOT nutzt allerdings für die seriellen Schnittstellen, die mit Spannungspegeln zwischen 0 und 5V arbeiten, eine dazu komplementäre Signalkodierung. Daher mussten für die Ansteuerung des Moduls über den EYEBOT die Signalleitungen zwischen der seriellen Schnittstelle SERIAL3 des EYEBOT und dem MINI-SSC II mit Hilfe einer integrierten Schaltung⁹ invertiert werden.

Bei den ersten Experimenten mit der ZORC-Architektur an der Universität Dortmund wurde der Akku provisorisch mit Klebeband an der Vorderseite des Torsos befestigt. Das zusätzliche Gewicht an der Vorderseite des Roboters trug zu einem gewissen Ausgleich des Schwerpunktes bei, da sich fast die gesamte Elektronik auf der Rückseite des Robotertorsos befindet. Allerdings ist diese Art der Befestigung nicht nur umständlich, sondern kann auch die Experimente wesentlich beeinflussen. Mangels Fixierungsmöglichkeit konnte der Akku nie an

⁸Details zur Software-Ansteuerung erläutert Kapitel 5.3.1

⁹Für die Invertierung der Signale wurde ein IC vom Typ CD4001 eingesetzt, der eine Schaltung mit 4 NOR-Gattern realisiert.

2 Erweiterung der ZORC-Hardware

exakt der gleichen Stelle befestigt werden, daher war die Lage des Schwerpunktes von ZORC deutlichen Schwankungen unterworfen. Da der Akku bei einem Gewicht von rund 340g mit knapp 20 % am Gesamtgewicht von ZORC beteiligt ist, ist zudem eine Lage wünschenswert, die den Schwerpunkt bei den Bewegungen des Roboters so wenig wie möglich verändert. Insbesondere sollte die beim Laufen notwendige Seitwärtsbewegung des Torsos immer noch eine Gewichtsverlagerung bewirken können. Das Akkupack sollte von dieser allerdings ausgenommen werden, da das Gewicht des Torsos mit den Armelementen hierfür ausreichend ist. Denn verschiebt sich der Schwerpunkt von ZORC bei der Bewegung des Torsos in zu starkem Maße, wird dadurch das Balancieren des Roboters erschwert. Der aus 6 in Reihe aneinandergelöteten Mono-Zellen bestehende Akkupack wurde zunächst umgebaut, so dass er aus 2 übereinander liegenden Reihen zu je 3 Zellen bestand. Dieser um die Hälfte verkürzte Akku konnte anschließend problemlos im Inneren des Torsos untergebracht werden, wobei die Befestigung vom Torso unabhängig auf dem Hüftelement erfolgte (s. a. Abb. 2.9). Eine Halterung stellte außerdem sicher, dass der Akku immer an der gleichen Position auf der Hüfte eingesetzt werden konnte und sich auch während der Experimente nicht verlagerte.

3 Vorbereitungen zum Einsatz der Simulationsumgebung

Der Einsatz einer Simulation war für die Zielsetzung, Laufalgorithmen für ZORC zu entwickeln, nicht zwingend erforderlich. In mehreren ähnlichen Projekten wurde der Lernprozess einzig auf den realen Robotern durchgeführt [1][14][15][16]. Es ergaben sich durch den Einsatz der Simulation im Rahmen dieser Arbeit zudem einige nicht-triviale Problemstellungen, die u. a. in den Kapiteln 3.2, 3.3.1 und 3.3.6 angesprochen werden. Diese waren vor allem darauf zurückzuführen, dass das vorhandene Simulationssystem nicht in jeder Hinsicht auf die Simulation von ZORC ausgelegt war. Dennoch wurde im Rahmen der Diplomarbeit relativ früh die Entscheidung getroffen, die Simulationsphase mit einzubeziehen und die Ergebnisse der Simulation anschließend auf ZORC zu übertragen (s. Kapitel 6). Denn trotz der Nachteile hat die Evolution von Laufalgorithmen in einer simulierten Umgebung zwei wesentliche Vorteile gegenüber ausschließlich auf dem realen Roboter durchgeführten Experimenten:

1. Die Hardware wird geschont
2. Der Lernprozess wird wesentlich beschleunigt

Der erste Faktor ist im wesentlichen ein Kostenfaktor. Auf ZORC werden, wie Kapitel 2.1 beschreibt, Servomotoren als Aktuatoren eingesetzt. Diese Motoren sind Verschleißteile und nutzen sich relativ schnell ab. Im Verlauf dieser Arbeit mussten insgesamt 2 defekte Servomotoren ausgetauscht werden. Zusätzlich zu den Kosten von rund 60 Euro pro Motor musste bei jedem Defekt eine Ausfallzeit des Roboters in Kauf genommen werden. Neben dem Einkauf und dem Austausch der Servomotoren mussten die in 2.2 beschriebenen Modifikationen an den neuen Motoren vorgenommen werden. Während dieser Zeitspanne waren keine Experimente mit dem Roboter möglich.

Von größerer Bedeutung im Rahmen einer zeitlich beschränkten Arbeit war jedoch der zweite Faktor. Durch den Einsatz der Simulation konnte die Evolution von Laufalgorithmen erheblich beschleunigt werden. Diese Beschleunigung ist zum einen möglich durch die Parallelisierung von Berechnungen (s. a. Kapitel 3.3.6), bei der mehrere evolvierte Programme parallel auf mehreren Rechnern simuliert werden können. Zusätzlich kann die Evolution von Laufalgorithmen in der Simulation ununterbrochen für mehrere Tage oder Wochen ablaufen. Hingegen können die Experimente auf dem realen Roboter nur wenige Stunden pro Tag durchgeführt werden, dabei wird die tatsächlich nutzbare Zeitspanne zusätzlich durch die in Kapitel 6.4.5 beschriebenen Faktoren wesentlich verkürzt. Betrachtet man außerdem, dass pro

3 Vorbereitungen zum Einsatz der Simulationsumgebung

Woche an maximal 5 Tagen manuelle Versuche mit ZORC durchgeführt wurden, so vergrößert sich der Vorsprung der Simulation noch einmal erheblich.

Betrachtet man allerdings zusätzlich die für den Einsatz der Simulation notwendigen Vorarbeiten, dann relativiert sich dieser Zeitvorteil durch den Einsatz von SIGEL. In diesem Fall muss zusätzlich der Zeitaufwand berücksichtigt werden, der für die Implementierung der Erweiterungen zu SIGEL (s. a. Kapitel 3.3) sowie für die Modellierung von ZORC (s. Kapitel 3.2) erforderlich war. Diese beiden Faktoren können jedoch mit gutem Gewissen vernachlässigt werden. Denn zum einen handelt es sich bei beiden Punkten um einen einmaligen Aufwand und damit um konstante Faktoren. Darüber hinaus sind sowohl das Modell von ZORC als auch die SIGEL-Erweiterungen über den Rahmen der Diplomarbeit hinaus wieder verwendbar und stehen sowohl für nachfolgende Experimente mit ZORC als auch mit beliebigen anderen Roboterarchitekturen zur Verfügung.

3.1 Das SIGEL-System

Als Simulationsumgebung kam SIGEL zum Einsatz [6][7]. Das Programm entstand im Rahmen der Projektgruppe 368 am Lehrstuhl für Systemanalyse des Fachbereichs Informatik an der Universität Dortmund. Das Akronym “SIGEL” steht für “Simulator für GP-evolierte Laufrobotersteuerungsprogramme”, d. h. das Programm erlaubt die automatische Programmierung von Laufalgorithmen für beliebige Roboterarchitekturen in einer simulierten Umgebung. Die eingesetzten Robotermodelle können aus beliebig geformten Gliedern bestehen, die durch mehrere Typen von Gelenken miteinander verbunden werden. Zusätzlich können die Roboter mit verschiedenen Typen von Sensoren und Motoren versehen werden. Die Modellierung des ZORC-Robotermodells wird in Kapitel 3.2 kurz beschrieben. Da nicht alle auf ZORC eingesetzten Sensor- und Motortypen mit dem ursprünglichen SIGEL-System modellierbar waren, musste dieses erweitert werden. Die Erweiterungen werden in den Kapiteln 3.3.1 – 3.3.3 beschrieben.

Für diese Robotermodelle können in SIGEL anschließend Laufalgorithmen mittels Genetischer Programmierung [4] evolviert werden. Zum Einsatz kommt dabei ein speziell für das Projekt entwickeltes Linear-GP-System (s. a. Kapitel 9 in [5]). Dazu werden die Robotermodelle zusammen mit einer Population von Programmen sowie diversen Parametern für das GP-System und die Simulation zu so genannten “Experimenten” zusammengefasst. Verschiedene Fitnessfunktionen stehen zur Steuerung der künstlichen Evolution zur Verfügung. Da sich diese allerdings aus den in Kapitel 4.1 erwähnten Gründen nicht für den Einsatz im Rahmen der Diplomarbeit eigneten, wurden zwei neue Fitnessfunktionen in SIGEL integriert. Diese werden im Detail ebenfalls im Kapitel 4.1 beschrieben.

Zur Evaluierung der Fitness von GP-Programmen (Individuen) werden diese von SIGEL in einem Interpreter ausgeführt. Zeitgleich wird durch die Simulationsbibliothek DYNAMECHS (s. a. Kapitel 3.4 in [6]) die physikalische Simulation des Robotermodells berechnet. Die Programme können mit dem simulierten Roboter mittels MOVE- und SENSE-Befehlen interagieren (s. a. Anhang B) und den Roboter auf diese Weise steuern. In den ersten Experimenten

BETRIEBSSYSTEM	RECHNERPLATTFORM (CPU / MHz)	BERECHNUNGSDAUER
Linux 2.4	AMD Athlon / 1550 MHz	6m 15s
Linux 2.4	AMD Athlon / 850 MHz	11m 15s
Solaris 5.7	Sun UltraSPARC / 170 MHz	74m 15s

Tabelle 3.1: Übersicht über die Dauer einer (vollständigen, s. a. Kapitel 3.3.5) Fitnessberechnung auf einigen der eingesetzten Rechnerplattformen. Die durchschnittliche Berechnungsdauer (gemittelt aus 4 Simulationsläufen) bezieht sich auf die 10-sekündige Simulation des besten Individuums des Experiments aus Kapitel 4.2.1 mit den in Tabelle 4.1 aufgeführten Parametern.

mit dem ZORC-Robotermodell in SIGEL zeigte sich bereits, dass die Fitnessevaluierungen sehr zeitaufwändig werden würden (Tabelle 3.1). Ursache dafür sind die aufwändigen Berechnungen, die für eine möglichst realistische physikalische Simulation der Robotermodelle notwendig sind. Daher wurde die in Kapitel 3.3.5 beschriebene Optimierung zum vorzeitigen Abbruch der Evaluierung schlechter Individuen ins System integriert. Außerdem konnte die schon angesprochene verteilte Fitnessberechnung genutzt werden, die SIGEL zur Parallelisierung der Berechnungen implementiert. Das System nutzt die PVM-Bibliothek zur selbständigen Verteilung dieser Berechnungen [17]. Auf Grund der in Kapitel 3.3.6 erläuterten Probleme wurde die dort beschriebene Erweiterung implementiert. Damit konnten schließlich die in Kapitel 4.2 vorgestellten Ergebnisse erzielt werden.

3.2 Modellierung von ZORC

Am Anfang der Arbeit mit SIGEL stand die Modellierung von ZORC. Als Grundlage für das Modell diente der bereits modifizierte Roboter mit den in Kapitel 2 beschriebenen Erweiterungen. Die Details der Erstellung von Robotermodellen für das Programm werden im Kapitel 6.3 in [6] behandelt. Die allgemeine Vorgehensweise sieht eine Definition der einzelnen Roboterglieder vor, d. h. eine separate Modellierung aller Elemente, die in sich starr sind und keine Gelenke beinhalten. Beispielsweise wurden bei der ZORC-Architektur die Oberschenkel oder Fußelemente als separate Roboterglieder definiert. Diese einzelnen Elemente werden anschließend in einer Textdatei, der so genannten "Roboterdefinitionsdatei", mittels Gelenken zusammengefügt. Zusätzlich können Sensoren definiert werden, mit denen z. B. die in Kapitel 2.2 beschriebenen Servopositionssensoren simuliert werden können.

Hinsichtlich der Übertragbarkeit der Simulationsergebnisse auf den realen Roboter war bei der Erstellung des ZORC-Modells eine möglichst genaue Modellierung notwendig. Daher wurde der Roboter in einem CAD-Programm "nachgebaut", indem die einzelnen Bauteile millimetergenau vermessen und aus elementaren Grundformen wie Quadern und Pyramiden im Maßstab 1:1 nachmodelliert wurden. Damit stellt das Modell von ZORC eine Abstraktion des wesentlich komplexer aufgebauten Roboters dar. Diese Vorgehensweise war allerdings aus zwei Gründen notwendig. Einerseits ergaben sich technische Probleme mit bestimmten CAD-Modellen, die in SIGEL importiert wurden. Sobald diese Robotermodelle

3 Vorbereitungen zum Einsatz der Simulationsumgebung

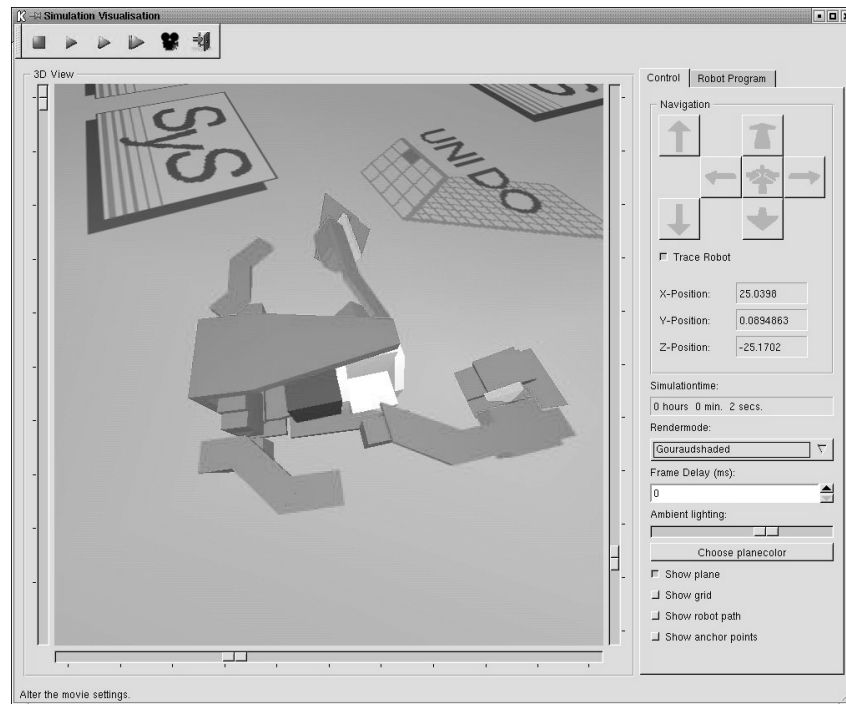


Abbildung 3.1: Bildschirmfoto des sigel_slave mit dem ZORC-Robotermodell.

zu komplex wurden, etwa aus zu vielen Polygonen bestanden, konnten diese nicht mehr in SIGEL dargestellt bzw. verarbeitet werden. Daher war ein gewisser Abstraktionsgrad schon aus Gründen der Durchführbarkeit der Simulation notwendig. Zum Anderen muss der Aufwand der Modellierung eines realen Roboters in einem sinnvollen Verhältnis zum Nutzen der Simulation stehen. Schon zu Beginn dieses Kapitels wurde erläutert, dass insbesondere beim Einsatz von Maschinellern Lernen die Verwendung einer Simulation vor allem einen Zeitvorteil mit sich bringen soll. Dieser fällt entsprechend geringer aus, je mehr Zeit in eine möglichst genaue Modellierung des Roboters investiert wird. Eine ungenauere Modellierung muss auf der anderen Seite ebenfalls mit einem höheren Zeitaufwand erkaufte werden, sobald die Ergebnisse der Simulation auf den realen Roboter übertragen werden. In diesem Fall muss zusätzliche Zeit in einen fortgesetzten Lernprozess auf dem Roboter investiert werden, um die Ergebnisse an die Differenzen zwischen Modell und Roboter anzupassen. Parker und Mills bringen diesen Zusammenhang in [18] im Rahmen eines ähnlich gelagerten Problems¹ auf den Punkt:

“All of these techniques require that we either put time into the model or into the training on the actual robot.”

Unter diesen Aspekten wurde ZORC so genau wie möglich modelliert, um die Simulation möglichst realistisch zu gestalten. Allerdings wurde bei der Modellierung überall dort, wo

¹In [18] wird ebenfalls die Übertragung von mittels Genetischen Algorithmen erzielten Simulationsergebnissen auf reale Roboter thematisiert.

es sinnvoll erschien, vom realen Roboter abstrahiert. Beispielsweise werden alle Servomotoren vereinfacht als Quader dargestellt. Außerdem wurden die Verbindungselemente und Achsen der Servomotoren als Pyramiden modelliert, deren Spitzen an genau der Stelle zusammentreffen, an der zwei Roboterglieder durch einen Servomotor – bzw. dessen Achse – miteinander verbunden sind. Das auf diese Weise entstandene Robotermodell von ZORC ist in Abbildung 3.1 dargestellt. Die in der Roboterdefinitionsdatei beschriebene Konfiguration des Roboters dient zudem als Startposition in der Simulation, d. h. die Winkel der Servomotoren zu Beginn der Simulationsphase entsprechen den in der Definitionsdatei angegebenen Werten. Das Modell von ZORC startet in einer aufrechten Position, die leicht nach vorne geneigt ist. Die Bewegungsrichtung des Roboters wurde von keiner der verwendeten Fitnessfunktionen (s. a. Kapitel 4.1) vorgegeben. Daher hat es sich über mehrere GP-Läufe als vorteilhaft erwiesen, wenn der Roboter einen gewissen Impuls in die “natürliche” Laufrichtung, d. h. nach vorne, erhält. Auf diese Weise konnte GP “motiviert” werden, den Roboter vorwärts zu bewegen. Bei vorangegangenen Versuchen mit dem vollständig aufgerichteten Robotermodell setzten sich häufig Programme durch, die den Roboter rückwärts fortbewegten.

Die in Kapitel 6.2 und 6.3 vorgestellten Resultate deuten darauf hin, dass das SIGEL-Robotermodell die reale Roboterarchitektur in akzeptabler Weise nachbildet.

Die auf diese Weise modellierten Glieder des Robotermodells wurden anschließend wie beschrieben in einer Roboterdefinitionsdatei zusammengefügt. Daneben wurden alle Sensoren von ZORC – mit Ausnahme des Stromaufnahmesensors (s. Kapitel 2.5) – im Modell definiert, d. h. 13 Sensoren zur Messung der Servoposition, 2 Fußkontakt-Sensoren sowie die 2 Neigungssensoren. Neben der Geometrie der Roboterglieder benötigt die Simulationsumgebung außerdem deren Masse, daher muss bei der Definition des Robotermodells jedem der Glieder eine Dichte zugeordnet werden. Dies geschieht in SIGEL – wie Kapitel 6.3 in [6] eingehend erläutert – über Materialien. Dazu werden den Robotergliedern beim Erstellen eines Robotermodells Materialien zugewiesen, mittels derer die Dichte dieser Bauteile spezifiziert wird. In Tabelle 3.2 sind für ZORC und dessen Robotermodell die Dichten zusammen mit den Massen der Roboterglieder zusammengefasst. Die einzelnen Roboterglieder bestehen nicht aus einem homogenen Material, für das die Dichte aus Tabellen wie beispielsweise in [19] hätte entnommen werden können. Statt dessen handelt es sich um ein Stoffgemisch aus diversen Materialien wie Metall, PVC-Formteilen und diversen Kunststoffen. Daher wurden die Dichten am realen Roboter gemessen und dazu zunächst die Komponenten des Roboters einzeln gewogen. Anschließend wurden zur Berechnung der Dichten die Volumina benutzt, die von SIGEL für die modellierten Roboterglieder berechnet wurden, um eventuelle Ungenauigkeiten der Modellierung zu berücksichtigen. Auf diese Weise war sicher gestellt, dass auch bei ggf. leicht abweichenden Geometrien des realen ZORC und dem Modell die Massen in der Simulation in jedem Fall denen des realen Roboters entsprachen. Als Besonderheit bleibt in diesem Zusammenhang zu erwähnen, dass SIGEL – bzw. die dort benutzte Dynamiksimulation DYNAMECHS – keine hohlen Körper kennt. Daher musste für den Torso des Robotermodells eine vergleichsweise kleine Dichte definiert werden. Bei der Berechnung der Dichte wurde die Luft im Inneren des hohlen Robotertorsos mit berücksichtigt.

Bei der Modellierung des Robotermodells wurde – wie auch anhand von Abbildung 3.3 erkennbar ist – zunächst der vollständige Roboter modelliert. Beim Zusammensetzen des Ro-

3 Vorbereitungen zum Einsatz der Simulationsumgebung

GLIED	DICHTE	MASSE	GLIED	DICHTE	MASSE
Fuß L,R	$1888.3 \frac{kg}{m^3}$	120g	Unterschenkel L,R	$1792.1 \frac{kg}{m^3}$	25g
Oberschenkel L,R	$2186.0 \frac{kg}{m^3}$	110g	Hüftgelenk L,R	$2175.5 \frac{kg}{m^3}$	60g
Hüfte	$2147.0 \frac{kg}{m^3}$	300g	Akkupack	$2780.0 \frac{kg}{m^3}$	340g
Schulter L,R	$1631.6 \frac{kg}{m^3}$	45g	Arm L,R	$1338.7 \frac{kg}{m^3}$	20g
Torso	$1791.9 \frac{kg}{m^3}$	600g			

Tabelle 3.2: Die Übersicht zeigt die Dichten und Massen der einzelnen Roboterglieder. Wegen der in Kapitel 3.2 erwähnten Gründe ist der Kopf nicht aufgeführt.

botermodells aus den einzelnen Gliedern in SIGEL trat allerdings das Problem auf, dass sich das Modell in der Simulation fehlerhaft verhielt, sobald der Kopf im Modell definiert wurde. Dieses Phänomen war lediglich auf das Vorhandensein des Kopfelements zurückzuführen und konnte auch mit der Hilfe mehrerer ehemaliger Teilnehmer der Projektgruppe 368 nicht aufgeklärt werden. Auf Grund der geringen Masse des Kopfelements, das mit etwa 25g insbesondere in Relation zu den benachbarten Torso- und Schulter-Elementen kaum ins Gewicht fällt, wurde der Kopf schließlich in den Simulationen nicht weiter berücksichtigt. Daher ist auch in den folgenden Abbildungen der Simulationsumgebung das ZORC-Robotermodell ohne Kopf abgebildet und wird dieser in Tabelle 3.2 ebenfalls nicht aufgeführt.

Das Hüftelement wurde als Torso-Link des Robotermodells definiert (s. a. Kapitel 6.3 in [6]). Der geometrische Mittelpunkt – nicht der Schwerpunkt – dieses Robotergliedes wurde daher für alle Messungen der Roboterposition im simulierten Raum als Bezugspunkt benutzt. Die zurückgelegte Distanz des Robotermodells und die Höhe des Roboters über dem Boden (s. a. Kapitel 4.1.1 bis 4.1.4) wird daher gemessen in Bezug auf den Mittelpunkt des Hüftelements.

Abschließend soll kurz darauf eingegangen werden, dass der Einsatz des ZORC-Robotermodells in SIGEL zunächst einige Probleme bereitete. Die Ursache dafür lag im Wesentlichen an den Voreinstellungen des Programms, die für die während der Projektgruppe eingesetzten Robotermodelle optimiert wurden. Es handelte sich hierbei im Speziellen um die voreingestellten Konstanten für die Dynamik-Simulation DYNAMECHS (Kapitel 3.4 in [6]) und die Schrittweite der Simulation (Kapitel 6.4 in [6]). Bei der Berechnung von Simulationen des ZORC-Modells brach diese entweder mit numerischen Fehlern ab oder zeigte ein fehlerhaftes Verhalten. Ein häufig auftretendes Fehlverhalten ließ beispielsweise die Modelle buchstäblich über den simulierten Boden “schweben”, ohne dass diese wieder auf den Boden zurückfielen. Die Ursache dieser unverträglichen Parameter liegt in den Eigenschaften der erwähnten Robotermodelle der Projektgruppe. Diese unterschieden sich in wesentlichen Punkten vom ZORC-Roboter. Zum einen waren die Roboter mit einer Höhe von einem bis mehreren Metern wesentlich größer als ZORC, der nur 37cm misst (bzw. ohne modellierten Kopf ungefähr 34cm). Darüber hinaus wurden für diese Robotermodelle Materialien mit unrealistischen Dichten definiert. Mit beispielsweise $0.5 \frac{kg}{m^3}$ betrugen diese ungefähr $\frac{1}{3}$ der Dichte von Sauerstoff beim Robotermodell walker der PG 368 [7]. Auch bei den “schwereren” Ro-

botermodellen – beispielsweise beim Robotermodell `twoBases` – bewegte sich die Dichte mit $2 \frac{kg}{m^3}$ in der Größenordnung von Propangas [19]. Stellt man diesen Werten die für das ZORC-Modell benutzten Materialdefinitionen gegenüber, die in Tabelle 3.2 zusammengefasst sind, fallen die deutlichen Abweichungen auf. Wieder in Relation gesetzt zu anderen Stoffen entspricht die errechnete Dichte der (inhomogenen) Roboterglieder ungefähr der doppelten Dichte von Eis und ist damit erwartungsgemäß wesentlich höher als bei den zuvor erwähnten Robotermodellen.

3.3 Erweiterungen des SIGEL-Systems

3.3.1 Implementierung der Servomotoren

In SIGEL waren zu Beginn dieser Diplomarbeit lediglich der Motortyp `Force` für die Simulation mit der `DYNAMECHS`-Bibliothek verfügbar. Es handelt sich hierbei um einen Motor, der bei der Ansteuerung durch die Instruktion `MOVE` (s. a. Anhang B) zwei Parameter erhält:

1. “*Motor-ID*” – Die Nummer des anzusteuernenden Motors
2. “*Kraft*” – Die auf das Gelenk, an dem der Motor befestigt ist, auszuübende Kraft

Diese Art der Ansteuerung durch die “Injektion einer Kraft” in das Robotergelenk orientiert sich sehr nahe an der Realisierung der Robotersimulation durch `DYNAMECHS`. Vom mechanischen Standpunkt betrachtet wirkt sich die vom Motor ausgeübte Kraft letztendlich am Gelenk aus; allerdings ist diese Form der Motoransteuerung für die Modellierung realer Motoren nicht realistisch. Ein Motor, der zur Ansteuerung ein Kommando der Form “15 Newton” akzeptiert, ist bei realen Robotern eher unüblich. Bei den in realen Robotern dagegen verbreitet eingesetzten Servomotoren dient der anzusteuernende Winkel als Ansteuerungsparameter für den Motor. Darüber hinaus können auch Schrittmotoren, deren Ansteuerung durch die Software sich nur marginal von der Ansteuerung eines Servomotors unterscheidet, als eine Variante winkelangesteuerter Motoren interpretiert werden. Damit konnte das ursprüngliche SIGEL-System zwei überaus häufig in der Robotik anzutreffende Motortypen nicht adäquat modellieren und simulieren.

Aus diesem Grund wurde der neue Motortyp `SimpleServo` implementiert. Dieser sollte in der Lage sein, die auf ZORC eingesetzten – und auch bei anderen real existierenden Robotern zu erwartenden – winkelangesteuerten Servomotoren in SIGEL zu simulieren. Diese Motorvariante wird ebenfalls durch eine `MOVE`-Instruktion angesteuert. Für einen `SimpleServo`-Motor wird dazu der 2. Parameter als anzusteuernender Winkel interpretiert. Die bei der Gelenkdefinition anzugebenden Minimal- und Maximalauslenkungen (Kap. 6.3 in [6]) werden bei der Ansteuerung berücksichtigt. Liegt der anzusteuernende Winkel außerhalb dieses zulässigen Bereiches, wird er vom `SimpleServo`-Motor je nach Drehrichtung auf eine dieser Auslenkungsgrenzen korrigiert. Die Kraft, mit der ein Motor vom Typ `SimpleServo` angesteuert wird, entspricht immer der Maximalkraft des Motors (Motor-Parameter “*maximalforce*” in der Definitionsdatei). In diesem Punkt orientiert sich der neue Motortyp am

3 Vorbereitungen zum Einsatz der Simulationsumgebung

realen Vorbild. Reale Servo- oder Schrittmotoren sehen keine Möglichkeit vor, bei der Ansteuerung die vom Motor auszuübende Kraft zu spezifizieren. Die Maximalkräfte, die von einem Motor ausgeübt werden können, sind allerdings ein Bestandteil der Motorspezifikation und stehen damit zur Modellierung und der daran anschließenden Simulation des Motors zur Verfügung. Aus diesem Grund wird die – aus Kompatibilitätsgründen mit den anderen SIGEL-Motortypen bei der Modellierung immer mit anzugebende – Minimalkraft in den Motorparametern ignoriert. Um Missverständnissen vorzubeugen, empfiehlt es sich, der Minimalkraft in der Roboterdefinitionsdatei den gleichen Wert wie der Maximalkraft zuzuweisen. Die Definition eines SimpleServo-Motors unterscheidet sich damit lediglich durch das Schlüsselwort “`simpleservo`” von der des kraftangesteuerten Motors vom Typ “`force`”. Anhang C beschreibt die Details zum Einsatz des Motors in Robotermodellen.

Die Realisierung des simulierten Servomotors in SIGEL wurde durch einige Besonderheiten der SIGEL-Implementierung erschwert. Die ursprünglich geplante Implementierung des Servomotors sollte sich am realen Vorbild orientieren. Bei diesem steuert die – bereits in 2.2 erwähnte – Steuerungselektronik im Servomotor einen Elektromotor und überwacht mittels Positionsrückmeldung dessen aktuelle Stellung. Die Software-Nachbildung dieser Steuerungselektronik wäre dank der bereits in SIGEL verfügbaren Winkelabfrage von Gelenken relativ problemlos zu realisieren gewesen. Eine solche Softwaresteuerung würde den Motor solange mit der Maximalkraft ansteuern, bis die gewünschte Endposition erreicht wäre. Durch die in SIGEL realisierte Form des Zusammenwirkens von Simulationsumgebung und GP-Interpreter war eine solche aktive Steuerung jedoch nicht möglich.

Während der Simulationsphase ruft die Klasse `SIG.Simulation` abwechselnd den Interpreter und die Simulationsumgebung auf. Dabei wird ähnlich einem präemptiven Prozess-Scheduler jeder der beiden Methoden nur eine sehr kleine Zeitspanne der Simulationszeit zur Verfügung gestellt. Für die Dauer dieser Zeitspanne führt der Interpreter solange Befehle aus, bis die Summe der einzelnen Befehlsdauern² die erlaubte Zeitspanne erreicht hat. Analog dazu wird in der Simulationsumgebung der Fortschritt für die spezifizierte Zeitspanne berechnet. Auf diese Weise laufen der Interpreter und die physikalische Simulation des Roboters praktisch simultan ab. Die Länge der Zeitspanne, währenddessen der Interpreter und die Simulation abwechselnd rechnen, wird durch den Simulationsparameter “`Stepsize`” bei der Erstellung eines SIGEL-Experimentes festgelegt.

Da Simulation und Interpreter also scheinbar parallel, tatsächlich jedoch in kleinen Zeitscheiben nacheinander ablaufen, war die Nachbildung einer Servo-Steuerungselektronik nicht zu realisieren. Bei dieser hätte die Kontrolle bis zum Erreichen der Endposition des Motors in der entsprechenden `MOVE`-Instruktion des Interpreters verbleiben müssen. Da die Simulationsumgebung während dieser Zeitspanne allerdings keine Rechenzeit erhält und damit in der Simulation kein Fortschritt erzielt wird, bewegt sich der simulierte Motor nicht, solange der Interpreter ausgeführt wird.

Die schließlich realisierte Implementierung setzt an die Stelle eines expliziten Steuerungsmechanismus’ für das Anfahren der gewünschten Servoposition die bereits im SIGEL-System vorhandene Begrenzung der Gelenkbewegung. Durch die bereits erwähnten Minimal- und

²Bei SIGEL-Experimenten werden Befehlsdauern in den Sprachparametern spezifiziert (Kap. 4.4.2 und 6.2 in [6]).

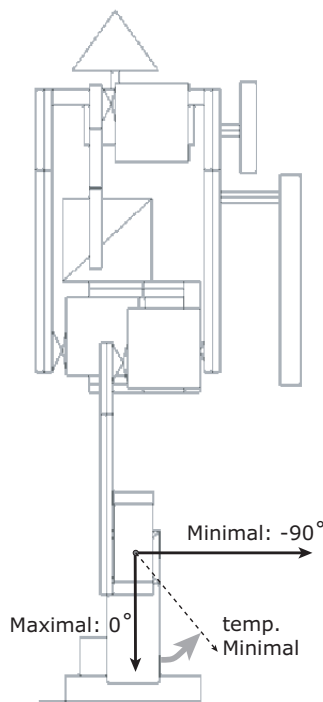


Abbildung 3.2: Manipulation der Gelenkauslenkungen durch den SIGEL-Servomotor (Kapitel 3.3.1)

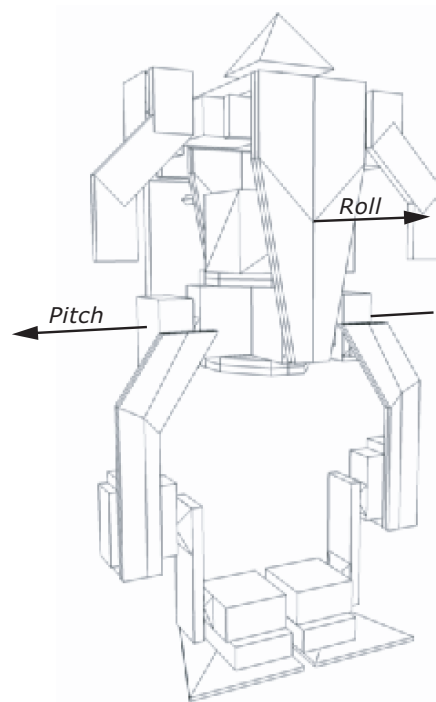


Abbildung 3.3: Die Grafik illustriert die Orientierung der Pitch- und Roll-Achsen anhand des ZORC-Robotermodells.

Maximalauslenkungen eines Gelenkes wird dessen Bewegungsfreiheit definiert. Beispielsweise hat das in Abbildung 3.2 gezeigte linke Knie von ZORC eine Minimalauslenkung von -90 und eine Maximalauslenkung von 0 Grad. Erreicht ein Gelenk im Verlauf der Simulation eine dieser Begrenzungen, wird es von DYNAMECHS automatisch an der Weiterbewegung gehindert. Da jeder Motor an einem Gelenk befestigt ist, lässt sich dieser Mechanismus zur Kontrolle der Motorbewegung einsetzen.

Um die Auslenkungsbegrenzungen für die Simulation des SimpleServo-Motors zu nutzen, werden die Begrenzungen des Gelenkes, an dem der simulierte Servomotor befestigt ist, in der Methode `moveDrive` der Klasse `SIG_CommandInterface` temporär geändert. Zu diesem Zweck wird zunächst die Drehrichtung berechnet, die abhängig von der aktuellen Servoposition zur gewünschten Servoposition führt. Die Begrenzung des Gelenkes, die in der Drehrichtung am Nächsten liegt, wird anschließend manipuliert. Zuvor wurde die tatsächliche Auslenkungsbeschränkung intern gesichert, so dass die Begrenzung zu einem späteren Zeitpunkt wieder auf den Originalwert gesetzt werden kann. Anschließend wird auf den Motor die (maximale) Motorkraft in Richtung der gewünschten Servoposition angewendet. In Abbildung 3.2 ist die Vorgehensweise am Beispiel des linken Knies von ZORC illustriert. Die minimale Auslenkungsbegrenzung wird auf „temp. Min“ herabgesetzt und der Motor aus der Anfangsstellung in Richtung der manipulierten Begrenzung in Bewegung gesetzt.

3 Vorbereitungen zum Einsatz der Simulationsumgebung

Sobald die Motorkraft durch den Aufruf der entsprechenden DynaMechs-Methode aktiviert wurde, wird der MOVE-Befehl verlassen. Gegebenenfalls werden bis zum Erreichen der Zeitschranke noch weitere Instruktionen abgearbeitet und der Interpreter daraufhin ebenfalls verlassen. Aus den bereits beschriebenen Gründen hat sich der Servomotor zu diesem Zeitpunkt noch nicht bewegt. Erst der dem Interpreteraufwurf folgende Aufruf der Simulation setzt den Motor in Bewegung. Durch die Manipulation der Auslenkungsbegrenzung wurde die Endabschaltung des Servomotors an DYNAMechs übertragen, da die Motorbewegung bei Erreichen dieser Begrenzung wie beschrieben gestoppt wird.

Genaugenommen wird der Motor zu diesem Zeitpunkt allerdings nicht abgeschaltet, sondern lediglich an der Weiterbewegung gehindert, während er weiterhin mit der maximalen Motorkraft in die gleiche Richtung dreht. Die von DYNAMechs zum Einhalten der Auslenkungsbegrenzung ausgeübte Gegenkraft hält den simulierten Servomotor aktiv an der gewünschten Position. Auf diese Weise wird SIGEL in die Lage versetzt, das aktive Halten einer Position durch den Servomotor zu simulieren. Diese Eigenschaft von realen Servomotoren erlaubt es beispielsweise, dass die Knie-Servomotoren von ZORC auch bei einer Belastung durch das Gewicht des Roboters ihre Position halten.

Die Geschwindigkeit, mit der die simulierten Servomotoren die Endposition anfahren, lässt sich auf Grund der Abgabe der Kontrolle an die Simulationsumgebung nicht beeinflussen. Allerdings wirkt sich die in SIGEL-Experimenten einstellbare Reibungskonstante “*Joint Friction*” auf die Geschwindigkeit aus, mit der sich der Motor bewegt³. Dadurch liegt die Drehgeschwindigkeit der simulierten Servomotoren im Rahmen der realen Vorbilder.

Die im Robotermodell definierten Auslenkungsbegrenzungen behalten weiterhin ihre Gültigkeit. Bei jeder MOVE-Instruktion zur Bewegung eines “SimpleServo”-Motors werden vor der beschriebenen Manipulation der Gelenkparameter die zuvor gesicherten Begrenzungen wiederhergestellt. Wenn das linke Knie aus dem vorangegangenen Beispiel wieder auf die Ausgangsposition bewegt werden soll, werden zunächst die in Abbildung 3.2 dargestellten Minimal- und Maximalauslenkungen wiederhergestellt. Anschließend werden die Begrenzungen für die erneute Bewegung des Servomotors auf die beschriebene Weise erneut eingestellt.

3.3.2 Implementierung der Pitch- und Roll-Sensoren

Zur Modellierung und Simulation der Neigungssensoren von ZORC (s. a. Kapitel 2.4) in der Simulationsumgebung ist SIGEL um die Sensortypen `Pitch` und `Roll` erweitert worden. Durch den sehr modularen und konsequent objektorientierten Aufbau von SIGEL fügten sich diese Erweiterungen relativ problemlos in das System ein. Zur Implementierung der neuen Sensortypen wurden dazu wie Abbildung 3.4 illustriert für jeden Sensortyp eine konkrete Klasse von der abstrakten Basisklasse `SIG_Sensor` abgeleitet.

Die Klasse `SIG_PitchRollSensor` implementiert die Sensoren zur Messung der Neigung der Pitch- und Roll-Achse. Gemäß Abbildung 3.3 ist die Pitch-Achse orthogonal zur Sagit-

³Die bei den Experimenten mit dem ZORC-Robotermodell verwendeten Reibungskonstanten werden in den Koza Tableaus der entsprechenden Experimente in Kapitel 4.2 angegeben.

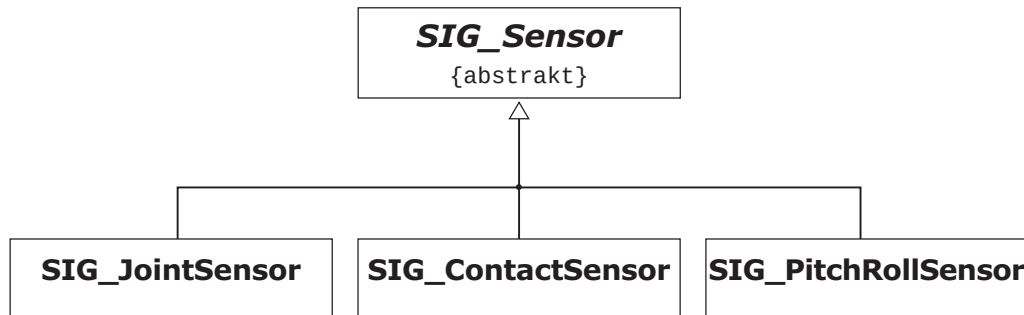


Abbildung 3.4: Dargestellt ist die abstrakte Sensor-Basisklasse `SIG_Sensor` mit den konkreten Implementierungen der neuen Sensortypen (s. a. Kapitel 3.3.2 und 3.3.3).

talebene der menschlichen Anatomie orientiert. Die Roll-Achse ist orthogonal zur Frontalebene orientiert. Da beide Sensoren eine fast identische Funktion erfüllen und sich nur in der zu messenden Achse unterscheiden, wurden beide Typen als Varianten eines Sensortyps zusammengefasst und dementsprechend implementiert. Dazu identifiziert innerhalb der `SIG_PitchRollSensor`-Klasse der Wert der Klassenattribute `isPitch` und `isRoll` die Sensorvariante, die von einer Instanz der Klasse repräsentiert wird. Abgesehen von der Implementierung werden beide Varianten in SIGEL jedoch als zwei separate Sensortypen gehandhabt, sowohl bei der Deklaration dieser Sensortypen als auch bei der Speicherung in Dateien.

Die Implementierung der Pitch- und Roll-Sensoren greift zur Ermittlung der Neigung auf die bereits erwähnte `DYNAMECHS`-Simulationsbibliothek zurück (Kapitel 3.4 in [6]). Wird innerhalb von SIGEL-GP-Programmen ein Sensor vom Typ `Pitch` oder `Roll` abgefragt (s. a. Anhang B), fordert die `SIG_PitchRollSensor`-Klasse zunächst von `DYNAMECHS` die Orientierung des betreffenden Robotergliedes an. Die Simulationsbibliothek liefert daraufhin den Rotationswinkel des Robotergliedes zurück, wie er zum Zeitpunkt der Abfrage, d. h. in der bis dahin verstrichenen Simulationszeit, gemessen wird. Der Rotationswinkel wird in Form einer Matrix zurückgeliefert und beinhaltet die Information über die aktuelle Rotation des Robotergliedes in allen 3 Achsen. Je nach der Belegung der erwähnten Klassenattribute wird schließlich der Wert einer dieser Achsen zurückgeliefert.

In der Roboterdefinitionsdatei werden diese Sensortypen über die Schlüsselwörter `pitch` und `roll` in Robotermodelle eingesetzt, gefolgt von der Angabe des zu messenden Robotergliedes. Die Anwendung dieser Sensoren sowie ein kurzes Beispiel erläutert das Addendum zum SIGEL-Handbuch in Anhang C.

3.3.3 Implementierung des Kontakt-Sensors

Für die Simulation von ZORC wurde ferner ein Sensortyp benötigt, mit dem sich die in Kapitel 2.3 beschriebenen Fußtaster modellieren ließen. Zu diesem Zweck realisiert die neue Klasse `SIG_ContactSensor` einen Kontaktsensor. Dieser ist in der Lage, den Kontakt des ihm zugeordneten Robotergliedes mit dem Boden festzustellen. Bei der Kontaktüberprüfung

3 Vorbereitungen zum Einsatz der Simulationsumgebung

werden im Gegensatz zum realen Fußtaster nicht nur eine Fläche, sondern sämtliche Kanten und Seitenflächen eines Robotergrundes berücksichtigt.

Die Implementierung des Kontaktsensors greift wiederum auf DYNAMECHS zurück. Im Zuge der physikalischen Simulation der Robotermodelle überprüft die Bibliothek den Bodenkontakt aller Glieder eines Roboters im Rahmen der Interaktion mit dem Boden. Auf diese Weise wird beispielsweise sichergestellt, dass Roboter nicht in den Boden einsinken – sofern dieses Verhalten durch eine entsprechende Parametrisierung der Umgebung erzwungen wird (s. a. Kap. 6.4.1 in [6]). Bei einer Abfrage des Sensors wertet die Klasse `SIG.ContactSensor` den Status des betroffenen Robotergrundes innerhalb von DYNAMECHS aus. Wurde von der Bibliothek ein Bodenkontakt erkannt, liefert der `SENSE`-Befehl den Wert 1 zurück, anderenfalls einen Wert von 0. Der in SIGEL implementierte Vergleichsbefehl `CMP` (s. a. Anhang B) bietet die Möglichkeit, auf beide Werte unterschiedlich zu reagieren, daher ist diese Art der Kodierung für das SIGEL-GP-System geeignet.

Wie bei den vorhergehenden Erweiterungen ist auch zum Kontaktsensor eine Anleitung für den Einsatz in Robotermodellen in Anhang C enthalten.

3.3.4 Implementierung des Zweipunkt-Crossovers

Der im SIGEL-System implementierte Rekombinations-Operator (Crossover-Operator) realisierte verschiedene Varianten des Einpunkt-Crossovers. Bei diesem werden beginnend ab einem zufällig gewählten Punkt im linearen Genom eines Individuums alle folgenden Instruktionen gegen die Instruktions-Sequenz eines zweiten Individuums getauscht, die auf die gleiche Weise gewählt wurde. Eine Rekombination dieses Typs zeichnet sich durch zum Großteil massiv destruktive Effekte aus, selbst im Vergleich zu den ohnehin meist destruktiven Auswirkungen der Rekombination (vgl. auch Kapitel 6 in [5]). Im Vergleich zur wesentlich weiter verbreiteten Variante von GP mit einer Repräsentation des Genoms als Baum bietet das lineare Genom diesem Operator eine wesentlich größere Angriffsfläche für massive Eingriffe in die Programmstruktur. Da die meisten Knoten eines Baums Blätter darstellen, ist die Wahrscheinlichkeit der Wahl eines inneren Knotens bei der zufälligen Auswahl eines Crossover-Punkts entsprechend gering. Zusätzlich werden bei der üblicherweise implementierten Rekombination nur Teilbäume getauscht, denen im linearen Genom – im Vergleich zur gesamten Programmlänge nur relativ kurze – Instruktions-Sequenzen entsprechen.

Aus diesen Gründen wurde ein neuer Rekombinations-Typ eingeführt, der ein Zweipunkt-Crossover implementiert. Der neue Operator orientiert sich dabei an dem Vorbild des Crossover-Operators für Baum-Genome und operiert dementsprechend auf vergleichsweise kurzen Instruktionssequenzen. Zunächst wird analog zum bereits vorhandenen Einpunkt-Crossover für jedes der beiden beteiligten Programme zufällig ein Punkt auf dem Genom gewählt. Anschließend werden, wieder für beide Genome getrennt, die Längen der auszutauschenden Sequenzen bestimmt. Die Sequenzlänge wird dabei zufällig aus dem Intervall $[1.. \frac{l}{10}]$ gewählt, wobei l für die Programmlänge des jeweils betrachteten Individuums steht. Die maximale Länge der getauschten Sequenzen wurde auf 10% der Programmlänge beschränkt, um ausschließlich den Tausch von kurzen Instruktionssequenzen zu ermöglichen. Auf diese Wei-

se soll die destruktive Wirkung des Operators eingeschränkt werden. Im Anschluss an die zufällige Wahl der Parameter der Rekombinations-Operation werden die gewählten Instruktionssequenzen beider Individuen schließlich getauscht.

Soll die Rekombination zur Erzeugung von neuen Individuen angewendet werden, dann kommt zufällig einer der drei derzeit in SIGEL implementierten Rekombinations-Operatoren zum Einsatz. Die beiden Einpunkt-Crossover Varianten werden mit einer Wahrscheinlichkeit von je 20% gewählt, das Zweipunkt-Crossover kommt mit einer Wahrscheinlichkeit von 60% zum Einsatz. Die höhere Wahrscheinlichkeit für letzteren Rekombinationstyp wurde bewusst gewählt, um die Wahrscheinlichkeit der beschriebenen destruktiven Auswirkungen einer Rekombination – über alle Rekombinations-Operatoren betrachtet – zu verringern.

3.3.5 Vorzeitiger Abbruch von Simulationen

Zufällig erzeugte Individuen sind mit hoher Wahrscheinlichkeit nicht in der Lage, den Roboter sinnvoll fortzubewegen oder auch nur in stehender Position zu halten. Abbildung 4.1 illustriert dieses anhand der Bewegungsabläufe einiger zufälliger Programme. Ähnliches trifft auch auf die meisten Programme zu, die mittels der Genetischen Operatoren Mutation und insbesondere Rekombination erzeugt werden (s. a. Kapitel 6.3 in [5]). Daher entfällt der überwiegende Anteil der Rechenzeit zur Evaluierung von Individuen auf wenig Erfolg versprechende Programme, d. h. Individuen mit geringen Fitnesswerten. Betrachtet man den Ablauf einer Simulation dieser “schlechten” Programme, fällt zudem auf, dass ein Großteil der Rechenzeit buchstäblich verschwendet wird. Beispielsweise verursachen viele dieser “schlechten” Programme typischerweise innerhalb der ersten Sekunde der Simulationszeit ein Umfallen des Roboters. Bei allen im Rahmen der Diplomarbeit durchgeführten GP-Läufen wurde ein Zeitraum von 10 Sekunden simuliert. Das bedeutet, die Simulation berechnet für die beschriebenen Programme während 90% der gesamten Simulationszeit einen möglichst realistisch am Boden liegenden Roboter. Hinsichtlich des erwünschten Verhaltens der Programme, einen Laufalgorithmus für die Fortbewegung auf zwei Beinen zu implementieren, ist das Verhalten der Individuen ab dem Zeitpunkt, ab dem der Roboter am Boden liegt, irrelevant. Die Durchführung der Evaluierung solcher Individuen bietet offensichtlich ein enormes Optimierungspotential.

Die Klasse `SIG_Simulation` wird in der SIGEL-Implementierung zur Berechnung der Simulationen eingesetzt und agiert als Schnittstelle zur `DYNAMECHS`-Bibliothek. Diese Klasse wurde um die virtuelle Methode `prematureTermination()` erweitert. Die Methode liefert eine Boolesche Variable zurück, von deren Belegung die Fortführung der Simulation des gerade evaluierten Individuums abhängig gemacht wird. Zu diesem Zweck wird diese Methode während der Berechnung der Simulation wiederholt abgefragt⁴. In dieser neuen Form wird die Simulation nun entweder bis zum Ablauf der voreingestellten Simulationszeit (Simulationsparameter “*Time to Simulate*”) durchgeführt oder bei Bedarf auf die beschriebene Weise vorzeitig abgebrochen. Diese Verfahrensweise zum vorzeitigen Simulationsabbruch bzw. allgemeiner zum vorzeitigen Abbruch einer Fitnessevaluierung wird in der Fachliteratur übli-

⁴Die Abfragen werden nach jedem Simulationsschritt der Länge “*Step Size*” durchgeführt.

3 Vorbereitungen zum Einsatz der Simulationsumgebung

cherweise als “premature termination” oder auch “limited error fitness” (s. a. Kapitel 10.1 in [5]) bezeichnet.

Für eine sinnvolle Nutzung dieser Optimierung müssen von `SIG_Simulation` abgeleitete Klassen die Methode `prematureTermination()` reimplementieren. Da das vorzeitige Terminierungskriterium sich an konkreten Aufgabenstellungen orientiert, ist eine allgemeingültige Implementierung nicht möglich. Daher liefert diese Methode in der Implementierung der Basisklasse `SIG_Simulation` einen konstanten Wert zurück, der nie zum Abbruch einer Simulation führt. Als Terminierungskriterium für den Einsatz mit dem ZORC-Robotermodell bot sich die Höhe des Hüftelements an. In der von `SIG_Simulation` abgeleiteten Klasse `SIG_EarlyRunTermSimulation` wird daher die Höhe des Hüftelements über dem Boden verfolgt. Fällt dessen Höhe auf weniger als $\frac{2}{3}$ der Starthöhe, wird die Simulation abgebrochen. Es hat sich experimentell bestätigt, dass kein aufrechtes Laufen mehr möglich ist, sobald das Hüftelement unter diese Höhe gesunken ist.

Durch den Einsatz dieser Optimierung konnte die durchschnittliche Rechenzeit für Fitness-evaluierungen drastisch reduziert werden. Für die beschriebenen “schlechten” Programme, die den überwiegenden Anteil einer intialen Population stellen, fällt die Optimierung am deutlichsten auf; bei diesen Programmen wird die Evaluierung um rund 900% beschleunigt. Mit einer wachsenden durchschnittlichen Fitness der Population im weiteren Verlauf der künstlichen Evolution werden diese Programme zunehmend durch bessere Individuen ersetzt. Damit steigt selbstverständlich die durchschnittliche Rechenzeit, die zur Evaluierung der Individuen notwendig ist, an. Hierbei ist allerdings zu beachten, dass diese Rechenzeit nun “sinnvoll” eingesetzt wird, d. h. zur Simulation von Programmen, die Bewegungsmuster erzeugen, die zunehmend menschlichem Laufen ähneln.

3.3.6 Parallelisierung im Batchsystem

Das SIGEL-System nutzt die frei verfügbare PVM-Bibliothek [17] zur Verteilung der Fitnessberechnungen von Individuen auf mehrere Rechner. Zu diesem Zweck wird ein Master-Slave-Paradigma implementiert: Die Master-Applikation “`sigel`” verwaltet das Experiment und berechnet die Tournaments, während die Slave-Applikation “`sigel_slave`” die Fitnessberechnungen durchführt. PVM verwaltet eine Liste von Rechnern, die für diese Berechnungen zur Verfügung stehen. Diese – in der PVM-Terminologie als “PVM-Hosts” bezeichnete – Liste wird bei der Definition eines Experiments angegeben.

Soll die Fitness eines Individuums der aktuellen GP-Population berechnet werden, so startet PVM auf einem dieser Rechner ein (dort installiertes) `sigel_slave` Programm. Die `sigel`-Applikation überträgt anschließend die zur Fitnessberechnung benötigten Daten – zu evaluierendes Individuum, Robotermodell, Umgebungsparameter – mittels von PVM zur Verfügung gestellten Funktionen an das `sigel_slave` Programm. Dieses berechnet pro Aufruf die Fitness genau eines Individuums. Nach erfolgter Berechnung wird der Fitnesswert des Individuums ebenfalls über PVM-Funktionen an `sigel` übertragen.

Dieser Mechanismus zur verteilten Fitnessberechnung war allerdings nicht geeignet für den Einsatz auf dem Batchsystem, wie es am Lehrstuhl für Systemanalyse eingesetzt wird. Letzte-

3.3 Erweiterungen des SIGEL-Systems

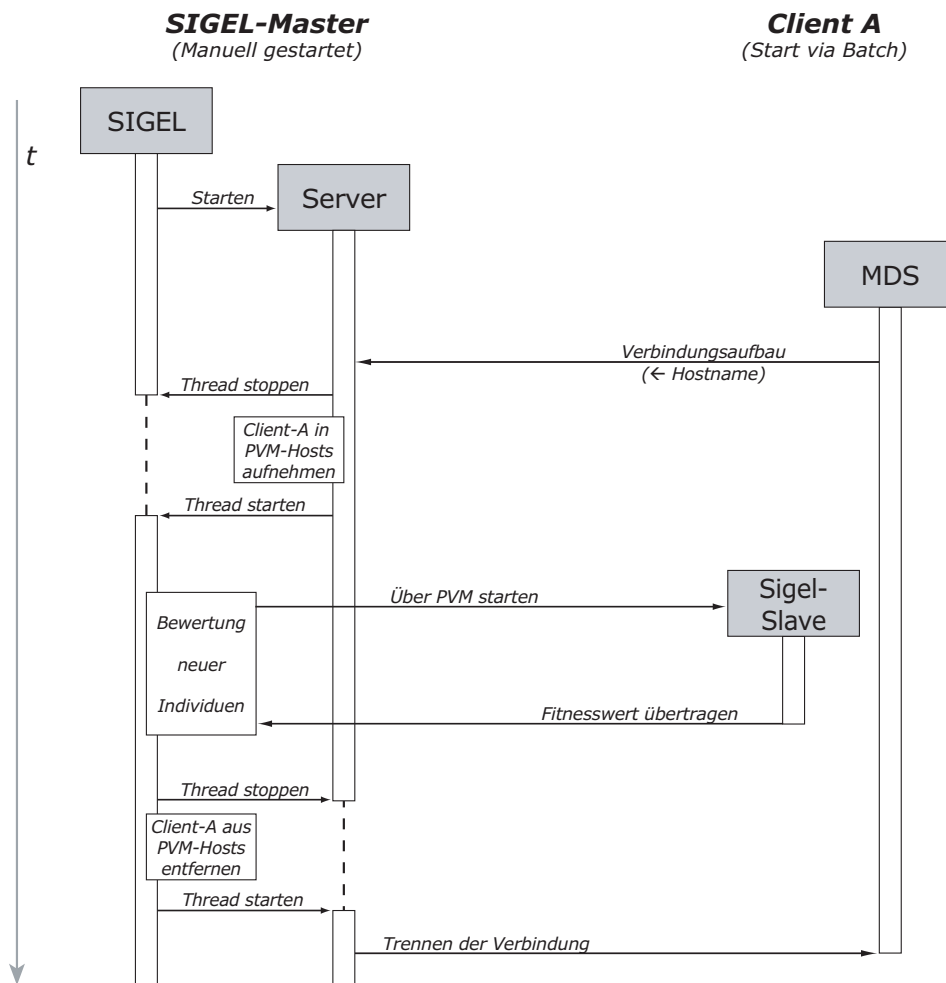


Abbildung 3.5: Das Sequenzdiagramm gibt den zeitlichen Ablauf von der Anmeldung bis zur Abmeldung eines Dynamic Slave wieder (s. a. Kapitel 3.3.6). “MDS” steht für die Applikation managedDynSlave, “Master” ist in diesem Fall der Rechner mit der Sigel-Applikation.

res dient der Parallelisierung beliebiger Berechnungen, indem die diese Berechnungen durchführenden Applikationen automatisch auf die am Lehrstuhl zur Verfügung stehenden Rechner verteilt werden. Dabei ist insbesondere nicht bekannt, welche Rechner zu einem bestimmten Zeitpunkt für die Berechnung verfügbar sind. Ein Rechner kann dabei nicht nur aus technischen Gründen – etwa durch Wartungsarbeiten – nicht verfügbar sein, sondern auch einem anderen Benutzer des Batchsystems zugeteilt worden sein. Für den Einsatz von SIGEL auf dem Batchsystem musste das Programm dahingehend abgeändert werden, dass statt der statischen PVM-Hosts-Liste eine dynamischen Liste von Rechnern benutzt wird. Diese Zuteilung der Rechner musste vom Batchsystem übernommen werden.

Zu diesem Zweck wurde der “Dynamic Slave” Modus implementiert⁵. In diesem Modus star-

⁵Um das Programm in diesem Modus zu starten, wird `sigel` in der Kommandozeile mit der Option “-de” gestartet. In der grafischen Oberfläche ist dieser Modus nicht verfügbar.

3 Vorbereitungen zum Einsatz der Simulationsumgebung

tet SIGEL einen Server-Thread, dessen Aufgabe darin besteht, eingehende Verbindungsanfragen von Rechnern anzunehmen, die vorübergehend an der Berechnung teilnehmen sollen. Die Hostnamen dieser Rechner werden anschließend der SIGEL-Applikation bekanntgegeben. Der Serverthread hält diese Verbindungen aufrecht, bis SIGEL diese nicht mehr benötigt und dieses dem Server signalisiert. Trifft dieses Signal ein, schickt der Serverthread eine Nachricht an die einzelnen Rechner und die Verbindungen werden getrennt. Auf diese Weise ist auch den temporär an den Berechnungen teilnehmenden Rechnern (Dynamic Slaves) bekannt, zu welchem Zeitpunkt sie nicht mehr benötigt werden. Denn je nach Implementierung, Experiment oder der Anzahl aktuell an der GP-Evolution teilnehmender Rechner wird `sigel_slave` mehrmals hintereinander auf einem Dynamic Slave gestartet, bis dieser freigegeben wird. Daher gibt es für den Dynamic Slave auch keine Möglichkeit, eigenständig festzustellen, zu welchem Zeitpunkt seine Aufgabe erfüllt ist. Wie im Folgenden erläutert wird, ist dieser Aspekt nicht zuletzt für den Einsatz im Batchsystem relevant.

Der Ablauf einer Berechnung im Batchsystem von der Anmeldung bis zur Abmeldung eines Dynamic Slave wird im Folgenden beschrieben. Dieser Ablauf ist zusätzlich in Abbildung 3.5 illustriert. Als zu startende Applikation wird dem Batchsystem ein Shell-Skript übergeben. Wird es bei erfolgter Zuteilung eines Dynamic Slaves auf diesem ausgeführt, installiert es zunächst die `sigel_slave`-Applikation sowie diverse zur Laufzeit benötigte Dateien in das lokale temporäre Verzeichnis. Anschließend startet es die Applikation `manageDynSlave`, die das Gegenstück zum oben erwähnten Server repräsentiert. Wird das Programm gestartet, kontaktiert es die Serverapplikation auf dem Master-Rechner und überträgt den Hostnamen des Dynamic Slaves. Dazu wurde der Name des Master-Rechners, auf dem die Applikation `sigel` ausgeführt wird, dem Shell-Skript beim Start übergeben. Nach der Übermittlung des Hostnamens des Dynamic Slaves wartet `manageDynSlave` auf die Nachricht des Servers, die Verbindung wieder zu trennen. Während dieses Zeitraums zwischen Anmeldung und Abmeldung ist der Dynamic Slave aus Sicht des Batchsystems belegt, auch wenn die von diesem gestartete Applikation keine Rechenlast erzeugt, sondern lediglich eine Reservierungsfunktion erfüllt. In diesem Zeitraum kann SIGEL mittels PVM auf die beschriebene Weise beliebig oft die `sigel_slave`-Applikation starten. Der Dynamic Slave wird somit für Berechnungen im Sinne des Batchsystems genutzt, auch wenn die rechenlast-verursachende Applikation nicht von diesem selbst gestartet wird. Nach den Berechnungen wird wie bereits erwähnt `manageDynSlave` auf dem Dynamic Slave benachrichtigt, dieses trennt die Verbindung und beendet sich selbst. Anschließend wird das Shell-Skript weiter abgearbeitet. Dieses löscht die temporär installierten Dateien und beendet sich ebenfalls, damit ist auch aus Sicht des Batchsystems die Berechnung auf dem Dynamic Slave beendet und der betreffende Rechner kann neu zugeteilt werden.

Der Master-Rechner, auf dem die `sigel`-Applikation ausgeführt wird, muss jederzeit von den Dynamic Slaves erreichbar sein. Zu diesem Zweck wurde eine Maschine eingesetzt, die nicht in das Batchsystem aufgenommen war und daher ausschließlich für SIGEL zur Verfügung stand. Um diesen Rechner effizienter einzusetzen, allerdings auch aus Gründen der internen Handhabung der Lastverteilung in SIGEL⁶, wurden auf diesem Rechner eben-

⁶Auch im dynamischen Modus muss mindestens ein Rechner statisch im Experiment definiert werden, da die Applikation `sigel` Berechnungen nicht selbständig durchführen kann. Zu diesem Zweck ruft `sigel` auf die

falls Fitnessberechnungen durchgeführt.

Die beschriebenen Zyklen, in denen ein Dynamic Slave an den Fitnessevaluierungen teilnimmt, sind in Relation zur gesamten Zeitspanne der GP-Evolution relativ kurz. In der im Rahmen der Diplomarbeit eingesetzten Implementierung wurden die Dynamic Slaves nach der Berechnung aller Begegnungen des aktuellen Turniers freigegeben. In zukünftigen Erweiterungen wäre es denkbar, dieses Schema flexibler zu handhaben, etwa durch zusätzliche Parameter in den Experimenten. Auf diese Weise wäre man bei weniger rechenaufwändigen Experimenten in der Lage, den Dynamic Slave optional statt nur an einem an mehreren Turnieren teilnehmen zu lassen. Dadurch würde der Overhead des Dispatchings durch das Batchsystem verringert, da auf jedem Dynamic Slave wie beschrieben einige Dateien installiert werden müssen. Ebenso sinnvoll wäre eine Verkürzung der Teilnahmeperiode bei solchen Experimenten, die eine längere Zeit für Berechnungen benötigen. In diesem Szenario könnte möglicherweise schon nach einer Fitnessberechnung der Rechner wieder freigegeben werden, d. h. die `sigel_slave` Applikation würde in diesem Fall nur einmal gestartet. Auf diese Weise wäre es möglich, die SIGEL-Berechnungen auf sehr viele Rechner zu verteilen, welche möglicherweise nur für einen relativ kurzen Zeitraum zur Verfügung stehen. Ähnlich wie bei den Projekten [20] und [21] könnte SIGEL auf diese Weise über das Internet “brachliegende” Rechenressourcen nutzen. Da in diesem Szenario insbesondere ungewiss ist, für welche Zeitdauer ein einzelner Rechner an der Berechnung teilnimmt, wäre eine Aufteilung der Rechenlast in einzelne Fitnessberechnungen hierbei besonders vorteilhaft. In Kapitel 7 wird die Thematik eingehender behandelt.

beschriebene Weise `sigel_slave` auf. Bei nur einem verfügbaren Rechner arbeitet `sigel_slave` auf dem gleichen Rechner wie die `sigel`-Applikation.

4 Die Evolution von Laufalgorithmen mit SIGEL

Nachdem die Implementierung der in den vorangegangenen Kapiteln beschriebenen Erweiterungen zum SIGEL-System und die Modellierung von ZORC abgeschlossen war, konnte mit der eigentlichen Evolution von Laufalgorithmen begonnen werden. Die künstliche Evolution, die durch das GP-System von SIGEL durchgeführt wird, muss durch eine Fitnessfunktion gesteuert werden (s. a. Kap. 4.6 in [6] und Kap. 5 in [5]). Diese bewertet durch die Vergabe von Fitnesswerten die Qualität der einzelnen Individuen. Zu diesem Zweck wird ein umso höherer Wert vergeben, je mehr das Verhalten des Individuums dem letztendlich gewünschten Verhalten entspricht. Bei den in SIGEL verwendeten Fitnessfunktionen wird dazu das Verhalten des Roboters in der Simulation bewertet, der von einem GP-Programm gesteuert wird.

Die im Rahmen dieser Arbeit eingesetzten Fitnessfunktionen werden zunächst in den Abschnitten 4.1.1 bis 4.1.4 vorgestellt. Die Ergebnisse, die mit diesen Fitnessfunktionen erzielt wurden, werden anschließend in den Abschnitten 4.2.1 bis 4.2.4 präsentiert.

4.1 Vorstellung der eingesetzten Fitnessfunktionen

Es wurde schon angesprochen, dass die bereits in SIGEL implementierten Fitnessfunktionen sich nicht für den Einsatz im Rahmen der Diplomarbeit eignen. Diese zeichnen sich beispielsweise durch im Quellcode fest vorgegebene Höhenangaben aus, mittels derer die Fitnessfunktionen an die im Laufe der Projektgruppe eingesetzten Robotermodelle angepasst wurden. Da ZORC wesentlich kleiner ist als diese Roboter, deren Höhe z. T. mehrere Meter betrug [7], ließen sich die Fitnessfunktionen nicht sinnvoll für die Evolution von Laufalgorithmen für ZORC einsetzen. Zudem war die in 3.3.5 beschriebene Optimierung vor allem für humanoide Roboterarchitekturen geeignet, die bislang nicht Gegenstand von SIGEL-Experimenten waren. Um die Optimierung in den nachfolgend beschriebenen Experimenten einsetzen zu können, mussten ohnehin neue Fitnessfunktionen¹ implementiert werden.

¹In SIGEL dient eine Fitnessfunktion nicht nur der Bewertung eines Programms, sie steuert zudem die Simulationsumgebung und den Interpreter. Durch die Wahl der `SIG_EarlyRunTermSimulation` Simulationsklasse in einer Fitnessfunktion kann daher die optimierte Simulation genutzt werden.

4.1.1 Die Fitnessfunktion “QuadHeight”

Die ersten Testläufe mit dem ZORC-Robotermodell in SIGEL zeigten bereits, dass für das System die Steuerung dieses Roboters wesentlich schwieriger war als für die im Laufe der Projektgruppe eingesetzten Robotermodelle. Als Ursache dafür ist neben den 13 Freiheitsgraden des Roboters vor allem die humanoide Architektur anzuführen, die von den Steuerungsprogrammen im Gleichgewicht zu halten war. Beide Faktoren führten zusammen genommen dazu, dass der Anteil an Erfolg versprechenden Programmen am gesamten Suchraum für die Experimente mit ZORC wesentlich kleiner als bei den Experimenten der Projektgruppe war.

Aus diesem Grund wurde zunächst eine Fitnessfunktion entworfen, die vor allem Programme “belohnt”, die den Roboter in aufrechter Position halten. In der Fitnessfunktion “QuadHeight” fließt daher die Höhe des Roboters (gemessen am Hüftelement, s. a. Kap. 3.2) quadratisch ein, so dass sich die Fitness Fit_I eines Individuums I berechnet als:

$$Fit_I = \frac{d}{t} \cdot h_{norm}^2 \quad \text{mit} \quad h_{norm} = \min(h, 1) \quad (4.1)$$

Die Variable d bezeichnet dabei die während der simulierten Zeitspanne t zurückgelegte Distanz des Roboters. Es wurde schon erwähnt, dass t bei den Experimenten 10 Sekunden betrug. Der erste Faktor bewertet demnach die Geschwindigkeit, mit der sich der Roboter fortbewegt. Mit h wird die durchschnittliche Höhe des Roboters über den Verlauf der Simulation bezeichnet, die aus den n während der Simulation gemessenen Höhenmessungen h_i wie folgt berechnet wird:

$$h = \frac{h_{Sum}}{n} \cdot \frac{1}{h_{Start}} \quad \text{mit} \quad h_{Sum} = \sum_{i=0}^n h_i \quad (4.2)$$

Der Faktor h_{norm} liegt demnach im Intervall $[0, 1]$, wobei ein Wert von 1.0 der Starthöhe h_{Start} des Roboters zu Simulationsbeginn entspricht und ein Wert von 0 der Bodenhöhe. Die Minumbildung stellt dabei sicher, dass h_{norm} auch durch unvorhergesehene Situationen in der Simulation nicht größer werden kann als 1. Da dieser Faktor quadratisch in die Fitnessberechnung eingeht, ist jedes Absinken der Roboterhöhe mit drastischen Fittesseinbußen verbunden.

Die Anzahl der Simulationsschritte n ergibt sich aus den Simulationsparametern “Time to Simulate” und “Stepsize”. Der zweite Parameter betrug bei allen Experimenten $5 \cdot 10^{-5}$, so dass sich z. B. für eine Simulationszeit von 10 Sekunden für n ein Wert von 2000 ergibt. Außerdem wird die in Kapitel 3.3.5 beschriebene Simulationsklasse `SIG_EarlyRunTermSimulation` eingesetzt. Diese wird auch von den folgenden, darauf aufbauenden Fitnessfunktionen benutzt. Da die Simulation beim Unterschreiten von $2/3$ der Starthöhe von dieser Simulationsklasse abgebrochen wird, nimmt die Fitnessfunktion für die verbleibende Zeit eine Höhe von 0 an. Individuen, die den Roboter zum Umfallen bringen, werden somit bestraft.

Abschnitt 4.2.1 behandelt die mit der Fitnessfunktion “QuadHeight” erzielten Ergebnisse.

4.1.2 Die Fitnessfunktion “BigDistance”

Im Anschluss an die Experimente mit der Fitnessfunktion “QuadHeight” wurde die im Folgenden beschriebene Fitnessfunktion “BigDistance” entworfen. Diese Fitnessfunktion bewertet die Fitness Fit_I eines Individuums I als:

$$Fit_I = \frac{100d}{t} \cdot h_{norm} \quad \text{mit} \quad h_{norm} = \min(h, 1) \quad (4.3)$$

Die durchschnittliche Höhe des Roboters h berechnet sich wieder nach Formel 4.2. Aus den in Abschnitt 4.2.1 beschriebenen Gründen wurde in der “BigDistance” Fitnessfunktion der Einfluss der Roboterhöhe auf die Fitness reduziert. Statt dessen erhielt die vom Roboter zurückgelegte Distanz einen stärkeren Einfluss. Die Gewichtung mit dem Faktor 100 wurde experimentell über mehrere GP-Läufe ermittelt. Ein zu großer Wert führte – analog zu den in 4.2.1 beschriebenen Beobachtungen – zu einer Überbewertung der Distanz. In diesen Szenarien wurde die Population von Programmen dominiert, die den Roboter auf möglichst riskante Art zu Fall brachten. Durch die dabei erreichten hohen Geschwindigkeiten kamen die Roboter relativ weit von der Ausgangsposition am Boden zur Ruhe und erreichten auf diese Weise größere Distanzen. Programme, die den Roboter aufrecht halten konnten, hatten mit dieser starken Gewichtung allerdings keine Überlebenschance in der GP-Population. Eine Gewichtung von 100 hält hingegen beide Faktoren – Distanz bzw. Geschwindigkeit und Höhe – in Waage und führte zu den in Kapitel 4.2.2 beschriebenen Ergebnissen.

Auf Grund der damit erzielten Resultate wurde die Fitnessfunktion in SIGEL unter dem Namen “Adaptive Walking” verfügbar gemacht. Die Adaptivität dieser Fitnessfunktion liegt in der flexiblen Behandlung der Roboterhöhe. Damit unterscheidet sie sich von den ursprünglich in SIGEL vorhandenen Fitnessfunktionen. Zur Fitnessberechnung wird kein fester Wert für die Höhe – wie beispielsweise “1 Meter” – genutzt. Stattdessen wird die prozentuale Höhe in Relation zur Startposition betrachtet, so dass die tatsächliche Größe des Robotermodells keinen Einfluss auf die Fitnessberechnung hat. Aus diesem Grund ist die Fitnessfunktion für eine Vielzahl von Roboterarchitekturen geeignet.

4.1.3 Die Fitnessfunktion “BigDistance-2”

Trotz der mit der Fitnessfunktion “BigDistance” erzielten Erfolge (s. Kapitel 4.2.2) haben sich in der Praxis einige Nachteile dieser Fitnessfunktion herausgestellt. Zum Einen traten gelegentlich Programme auf, die an die Simulationszeit überangepasst waren. Dieser Programmtyp hat das Robotermodell zu Beginn der Simulation zunächst stabilisiert, so dass dieses nicht umfiel. Im weiteren Simulationsverlauf wurde der Roboter typischerweise kaum bewegt. Erst kurz vor Ablauf der definierten Simulationszeit wurde eine Bewegung ausgeführt, die den Roboter aus dem Gleichgewicht und schließlich zu Fall brachte. Zeitlich ist eine solche Bewegungssequenz so abgestimmt, dass die durch den Fall zurückgelegte Distanz in die Wertung eingeht und dennoch die vollständige Zeitspanne simuliert wird. Die Simulation wird abgeschlossen, bevor das Terminationskriterium von `SIG_EarlyRunTermSimulation` zu einem

4.1 Vorstellung der eingesetzten Fitnessfunktionen

Abbruch der Simulation führen kann. Der Sturz wird folglich von der Fitnessfunktion nicht registriert und kann daher nicht durch die Zuweisung eines Höhenwertes von 0 bestraft werden. Dieses zeitliche abgestimmte bzw. “kontrollierte Umstürzen” des Robotermodells bereitet Schwierigkeiten während des Lernprozesses. Denn insbesondere bevor im Verlauf der künstlichen Evolution Laufprogramme in der Population auftreten, die “echte” Laufalgorithmen realisieren und auf diese Weise eine Distanz von mehr als 25cm zurücklegen, kann die Population von Programmen des beschriebenen Typs dominiert werden. Diese können einzig durch den Sturz des Roboters eine Distanz von etwa 25cm (gemessen am Hüftelement) erzielen.

Um die Überanpassung an die Länge der Simulationszeit zu erschweren, wird die Simulationsdauer variiert. Zu diesem Zweck wird die Simulationsdauer für jedes Individuum durch eine normalverteilte Zufallsvariable bestimmt, bei der die voreingestellte Simulationszeit – in der Regel 10 Sekunden – die höchste Wahrscheinlichkeit besitzt. Zusätzlich werden die Zufallsvariablen auf Werte zwischen 50% und 200% der voreingestellten Zeit beschränkt, d. h. im Rahmen der Experimente auf Werte von mindestens 5 und höchstens 20 Sekunden.

In SIGEL werden Individuen grundsätzlich nur nach ihrer Erzeugung evaluiert. Der dabei berechnete Fitnesswert wird beibehalten, bis das Individuum die Population verlässt. Falls daher bei Individuen eine Überanpassung an eine bestimmte Simulationsdauer auftritt, kann der auf diese Weise erzielte (vermeintlich gute) Fitnesswert nicht durch nachfolgende Evaluierungen mit einer unterschiedlichen Simulationsdauer relativiert werden. Durch die variierende Dauer ist es jedoch weniger wahrscheinlich, dass solche überangepassten Individuen die Population dominieren. Denn zumindest deren Nachkommen erhalten einen wesentlich schlechteren Fitnesswert, sobald sie mit abweichenden Simulationsdauern evaluiert werden.

Eine weitere Maßnahme dient dem Zweck, Individuen zu bestrafen, die den Roboter sehr ungleichmäßig fortbewegen. In diese Kategorie fallen insbesondere die zuvor erwähnten Laufprogramme, die den Roboter erst gegen Ende der Simulation bewegen. Neben der Strecke, die während der gesamten Simulationszeit zurückgelegt wird, betrachtet die Fitnessfunktion “BigDistance-2” die während jeder simulierten Sekunde zurückgelegten Distanzen. Eine gleichmäßige Fortbewegung des Roboters, die sich durch viele ähnliche Werte in diesen Einzelstrecken auszeichnet, galt es zu belohnen. Im Gegenzug musste die Fitnessfunktion sehr unterschiedliche Werte bestrafen. Daher wird die Standardabweichung σ dieser Einzelstrecken berechnet. Die Standardabweichung ergibt für gleichmäßige Bewegungen kleine Werte, während ungleichmäßige Bewegungen in großen Werten resultieren. Aus diesem Grund wird in der Fitnessfunktion der Kehrwert $1/\sigma$ verwendet, um die gewünschte Wirkung zu erzielen. Mit diesem Faktor wird die durchschnittliche Geschwindigkeit der Roboterbewegung gewichtet. Insgesamt wird die Fitnessfunktion aus Formel 4.3 für die Fitnessfunktion “BigDistance-2” erweitert zu:

$$Fit_I = \frac{1}{100\sigma + 1} \cdot \frac{100d}{t} \cdot h_{norm} \quad \text{mit} \quad h_{norm} = \min(h, 1) \quad (4.4)$$

Die Gewichtung von σ mit dem Faktor 100 hat sich experimentell als notwendig erwiesen, um der Standardabweichung einen signifikanten Einfluss auf den Fitnesswert zu geben. Zusätz-

lich bewirkt die Addition von 1, dass sich eine Standardabweichung von 0 neutral auswirkt, d. h. der Geschwindigkeitsfaktor zu 100% in den Fitnesswert einfließt.

Schließlich wird die durch den Einsatz von `SIG_EarlyRunTermSimulation` gewonnene Information genutzt, um ungeeignete Laufalgorithmen zu bestrafen. Falls die Simulation vorzeitig abgebrochen wird, da das Robotermodell in der Simulation zu Boden gefallen ist, wird die Fitness Fit_I des betreffenden Individuums halbiert.

In Kapitel 4.2.3 wird das Ergebnis des Experiments präsentiert, das mit der Fitnessfunktion “BigDistance-2” durchgeführt wurde.

4.1.4 Die Fitnessfunktion “Stepper”

Die mit der Fitnessfunktion “BigDistance” erzielten Ergebnisse zeichnen sich durch Programme aus, die den Roboter in kleinen Schritten vorwärts bewegen. Insbesondere der in Abbildung 4.6 dargestellte beste Laufalgorithmus des gezeigten “BigDistance”-Experiments benutzt nur sehr kleine Schrittweiten. Frühere Erfahrungen mit der (realen) ZORC-Architektur zeigen jedoch, dass diese in der Lage ist, sich mit größeren Schritten vorwärts zu bewegen [1]. Es ist anzunehmen, dass mit dem modifizierten ZORC durch die längeren Oberschenkel-Elemente noch größere Schrittweiten möglich sind. Dadurch würde dem Roboter neben einer größeren Geschwindigkeit vor allem zu einem Laufstil verholfen, der mehr Ähnlichkeit mit dem menschlichen Gang besitzt. Mit der Fitnessfunktion “Stepper” wurde die Absicht verfolgt, die Schrittweite der Laufalgorithmen zu maximieren. Die Schwierigkeit bestand zunächst darin, eine Formalisierung für den Begriff der “Schrittweite” zu finden. Dabei mussten die Möglichkeiten und Grenzen der DYNAMECHS-Simulationsumgebung beachtet werden. Allgemein lässt sich für eine zurückgelegte Distanz d die durchschnittliche Schrittweite s bestimmen, indem die Anzahl i der Schritte gezählt wird. Dann beträgt die durchschnittliche Schrittweite $s = \frac{d}{i}$.

In DYNAMECHS ist es zwar nicht möglich, die Anzahl der Schritte eines Robotermodells zu zählen. Da es keine Möglichkeit gibt, einzelne Glieder eines beliebigen Robotermodells als “Füße” auszuweisen, können diese nicht gesondert betrachtet werden. Es ist jedoch möglich, allgemein den Bodenkontakt aller Glieder des Robotermodells zu betrachten. Dieser Mechanismus von DYNAMECHS wurde im Zusammenhang mit der Implementierung der Kontaktsensoren in Kapitel 3.3.3 bereits erwähnt.

Zunächst wurde daher die Klasse `SIG_GPFullDataRecorder` erweitert. Diese zeichnet für jeden Simulationsschritt der Länge “Stepsize” diverse Daten des simulierten Robotermodells auf, z. B. die einzelnen Punkte der zurückgelegten Wegstrecke. Mit Hilfe dieser Daten werden beispielsweise die Individuen nach abgeschlossener Simulation von den Fitnessfunktionen bewertet. Durch die Erweiterung dieser Klasse wird in jedem Simulationsschritt zusätzlich die Anzahl der Roboterglieder aufgezeichnet, die momentan den Boden berühren.

Um nun die Schrittweite zu maximieren, muss die Anzahl der Roboterglieder, die während des simulierten Zeitraums den Boden berühren, minimiert werden. Diese Vorgehensweise soll Laufalgorithmen motivieren, welche die gesamte Wegstrecke mit möglichst wenig Schrit-

4.2 Ergebnisse mit dem simulierten ZORC-Roboter

ten zurücklegen. Die Fitnessfunktion “Stepper” berechnet die Fitness Fit_I eines Individuums I als:

$$Fit_I = \frac{d}{t} \cdot r \quad (4.5)$$

Der Faktor r bewertet die durchschnittliche Anzahl der Roboterglieder, die im Verlauf der simulierten Zeitspanne den Boden berührt haben. Dabei fällt r umso niedriger aus, je mehr Roboterglieder in Bodenkontakt getreten sind. Für einen stabilen Stand des Roboters wird zu jedem Zeitpunkt mindestens ein Fuß am Boden benötigt. Dieser Fall wirkt sich daher neutral auf die Fitness aus. Eine kurze “Flugphase” des Roboters, d. h. falls dieser keinen Bodenkontakt hat, wird ebenfalls so interpretiert, dass sich ein Roboterglied auf dem Boden befindet. Auf diese Weise sollen unerwünschte Nebeneffekte im Lernprozess vermieden werden. Damit ergibt sich für r folgende Formel:

$$r = \frac{r_{Sum}}{n} \quad \text{mit} \quad r_{Sum} = \sum_{i=0}^n \frac{1}{\max(1, r_i)} \quad (4.6)$$

Die Variable r_i bezeichnet für den Simulationsschritt i die Anzahl der Roboterglieder, die den Boden berühren. Mit n wird analog zu den vorangegangenen Gleichungen die Anzahl der Simulationsschritte bezeichnet. Steht z. B. der Roboter während der gesamten Simulationsdauer nur auf einem Fuß, hat r den Wert 1 und wirkt sich neutral auf den Fitnesswert aus.

Die Fitnessfunktion “Stepper” setzt – wie alle zuvor erwähnten Fitnessfunktionen – ebenfalls die Simulationsklasse `SIG_EarlyRunTermSimulation` zum vorzeitigen Abbruch der Evaluierung wenig erfolgsversprechender Individuen ein. Aus diesem Grund wird bei der Betrachtung des Bodenkontaktes in den meisten Situationen ein Fußelement des Robotermodells betrachtet. Da die Simulation abbricht, sobald die Höhe des Hüftelements unter $2/3$ der Startposition fällt, haben die anderen Roboterglieder in der Regel keine Gelegenheit, vorher in Kontakt mit dem Boden zu treten.

Falls die Simulation von `SIG_EarlyRunTermSimulation` frühzeitig beendet wird, fehlen Informationen über die Bodenkontakte für die nicht durchgeführten Schritte der Simulation. Die Klasse `SIG_GPFullDataRecorder` zeichnet die Anzahl der Bodenkontakte lediglich für die simulierten Schritte auf. Daher wird vereinfachend angenommen, dass ein am Boden liegender Roboter für die verbleibende Simulationszeit mit 3 Gliedern den Boden berührt.

Die mit dieser Fitnessfunktion erzielten Ergebnisse sind Gegenstand von Kapitel 4.2.4.

4.2 Ergebnisse mit dem simulierten ZORC-Roboter

Die folgenden Abschnitte präsentieren die Ergebnisse der Evolution von Laufalgorithmen in der Simulationsumgebung. Für alle Experimente wird das Koza-Tableau mit allen wesentlichen Experiment-Parametern angegeben. Aus Platzgründen sind jedoch nur diejenigen Pa-

4 Die Evolution von Laufalgorithmen mit SIGEL

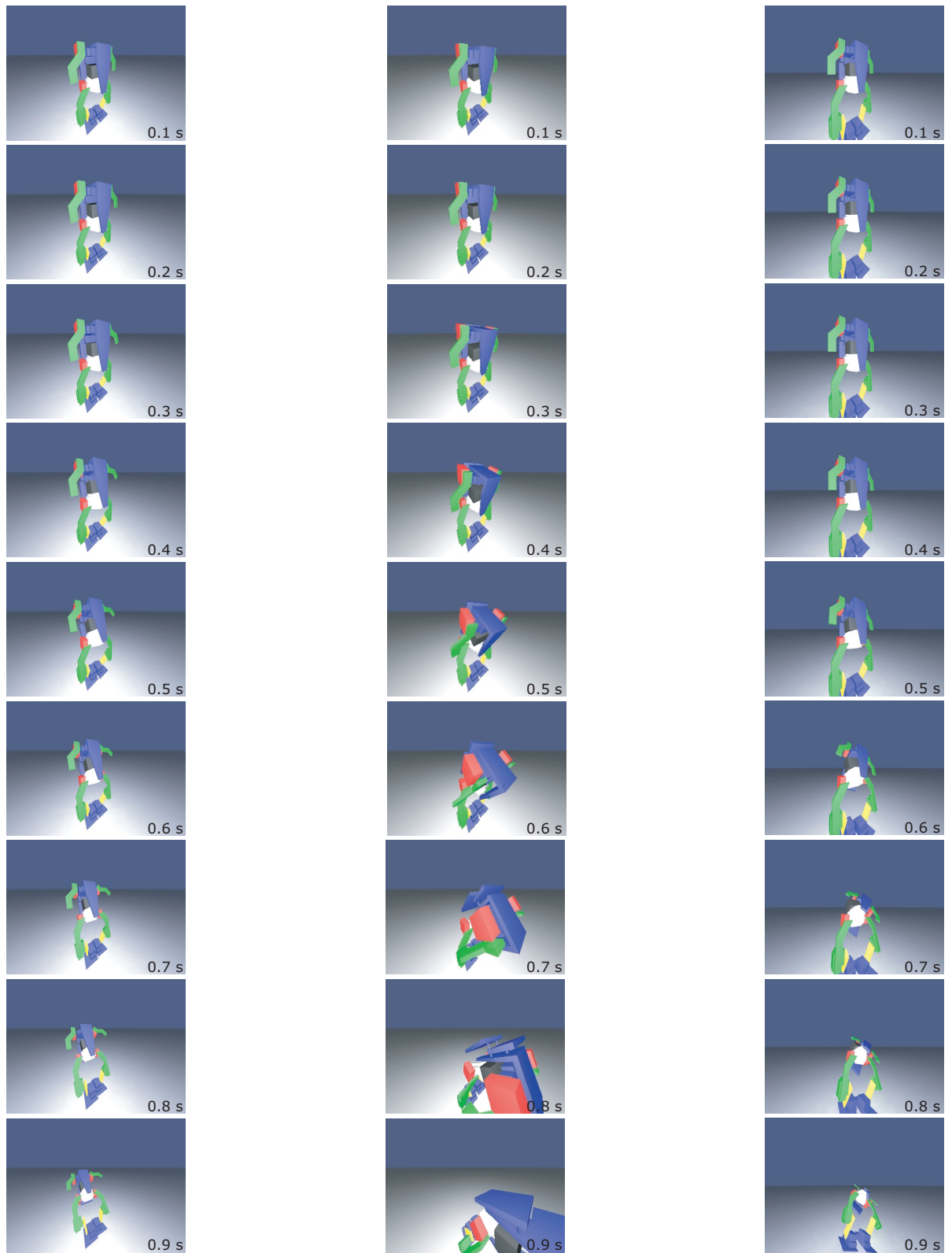


Abbildung 4.1: Die simulierten Bewegungsabläufe von drei zufällig generierten Individuen. Die Einzelbilder der Sequenzen wurden im Abstand von 1/10 Sekunden aufgezeichnet.

4.2 Ergebnisse mit dem simulierten ZORC-Roboter

parameter notiert, die von den Voreinstellungen für SIGEL-Experimente abweichen. Für eine vollständige Parameterübersicht ist daher auf den Endbericht der Projektgruppe SIGEL [6] bzw. die SIGEL-Applikation verwiesen [7]. Darüber hinaus werden alle dem GP-System zur Verfügung stehenden Instruktionen in Anhang B detailliert beschrieben.

In Abbildung 4.1 sind zunächst einige typische Bewegungsabläufe von zufällig erzeugten Programmen dargestellt. Da zu Beginn der Simulation alle simulierten Servomotoren ausgeschaltet sind, fällt der Roboter meist innerhalb kurzer Zeit auf Grund des eigenen Gewichts zu Boden. Diesen initialen Laufprogrammen werden im Folgenden die besten Individuen der einzelnen Experimente gegenübergestellt. Für die Darstellung der Bewegungssequenzen der jeweils besten Individuen sind in den meisten Fällen Einzelbilder im Abstand von 1/10 Sekunden aufgenommen worden. Für jedes Individuum wurden dazu die aussagekräftigsten Zeitabschnitte der Bewegungssequenz ausgewählt und der Zeitindex jeweils in der rechten, unteren Bildecke angegeben.

4.2.1 Ergebnisse mit der Fitnessfunktion “QuadHeight”

Für das vorgestellte Experiment mit der Fitnessfunktion “QuadHeight” fasst Tabelle 4.1 alle wesentlichen Parameter zusammen. An dieser Stelle ist erwähnenswert, dass SIGEL zwar mit einem Tournament-Selektionsschema arbeitet, durch den Parameter “*Tournaments per Generation*” jedoch einen Generationsbegriff einführt. Daher wird sowohl in den Koza-Tableaus als auch in den Fitness-Diagrammen die Anzahl der Generationen verwendet.

Das Verhalten des besten Individuums, das aus diesem Experiment hervorgegangen ist, zeigt Abbildung 4.3. Die Bewegungssequenz lässt erkennen, dass der Roboter sich nicht weit

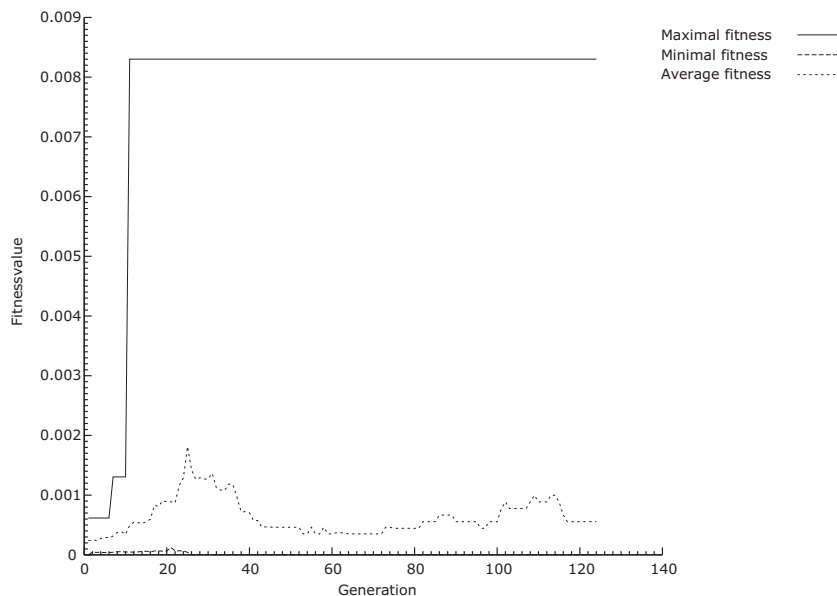


Abbildung 4.2: Das Fitnessdiagramm des Experiments mit der Fitnessfunktion “QuadHeight”

4 Die Evolution von Laufalgorithmen mit SIGEL

PARAMETER	WERT
Zielsetzung	Evolution eines Laufalgorithmus'
Populationsgröße	75
Initialisierungsmethode	zufällig
Mutations-Wahrscheinlichkeit	85 %
Rekombinations-Wahrscheinlichkeit	10 %
Reproduktions-Wahrscheinlichkeit	5 %
Instruktions-Set	ADD, SUB, DIV, MUL, MIN, MAX, MOD, CMP, JMP, COPY, LOAD, SENSE, MOVE, DELAY, NOP
Terminal-Set	Ganzzahlige Konstanten $\in [-2^{31}, 2^{31} - 1]$
Selektionsmethode	Tournament-Selektion
Turniergröße	2 (Mutation, Reproduktion) bzw. 4 (Rekombination)
Tournaments pro Generation	38
Generationen	124
Register	8 Register à 32 Bit
Befehlsdauer	1/1000 s
Max. Dauer von DELAY	5s
Schrittgröße "Step Size"	0.00005
"Random Seeds" (Simulation, GP)	1, 0
Simulationszeit	10s
Joint-Konstanten (Spring, Damper, Friction)	1000, 0, 0.38
Damper-Konstanten (Planar, Normal)	1000, 1000
Roboter-Startposition (X, Y, Z)	25, 0.2225, -25

Tabelle 4.1: Koza-Tableau des "QuadHeight"-Experiments

von der Startposition entfernt. Das Verhalten dieses Laufprogramms lässt sich im Wesentlichen als ein "auf der Stelle treten" zusammenfassen, dabei rotiert der Roboter langsam um die eigene Achse. Durch diese Drehbewegung entfernt sich der Roboter minimal, d. h. lediglich 2-3 cm von der Startposition. Damit ist dieses Individuum den anderen Individuen dieses Experiments, die den Roboter ebenfalls in aufrechter Position halten, minimal überlegen.

Im Verlauf des Lernprozesses wurde vom GP-System relativ früh gelernt, den Roboter in aufrechter Position zu halten. Offensichtlich führt jedoch die übermäßige Bewertung der Höhe bei dieser Fitnessfunktion dazu, dass alle weiteren Versuche, einen Laufalgorithmus zu entwickeln, entmutigt werden. Auf dem Weg zu einem erfolgreichen Laufalgorithmus für ZORC haben Individuen eine hohe Wahrscheinlichkeit, den Roboter zu Fall zu bringen. Der Fitnessgewinn durch eine größere Distanz ist bei der Fitnessfunktion "QuadHeight" offenbar zu gering in Relation zu dem Lernaufwand und dem Risiko, den das Erlernen eines aufrechten Ganges mit sich bringt.

Die im Fitnessdiagramm in Abbildung 4.2 dargestellten minimalen, durchschnittlichen und

4.2 Ergebnisse mit dem simulierten ZORC-Roboter

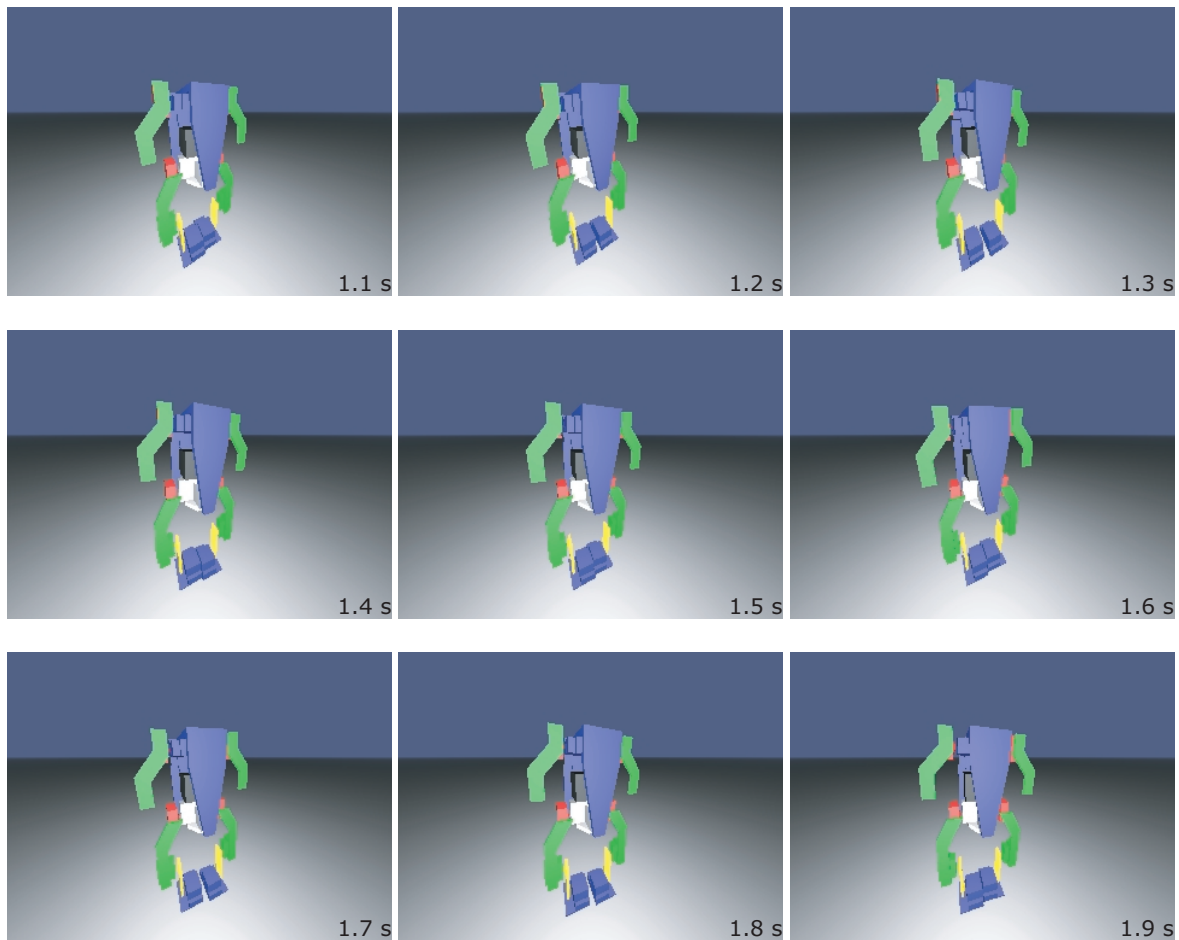


Abbildung 4.3: Bewegungsablauf des besten Programms des “QuadHeight”-Experiments (Kap. 4.2.1)

maximalen Fitnesswerte der Population lassen erkennen, dass die künstliche Evolution mit hoher Wahrscheinlichkeit zum Erliegen gekommen ist. Neben diesem aus dem Diagramm ersichtlichen fehlenden Fortschritt in der maximalen Fitness deutet zudem die (nicht abgebildete) geringe Diversität in der Population auf das Erliegen der Evolution hin. Gegen Ende der Evolution wird die Population von Programmkopien dominiert, die sich durch eine geringe phänotypische Diversität auszeichnen.

Die Resultate des vorgestellten sowie weiterer Experimente mit der Fitnessfunktion “QuadHeight” führten zu der in 4.1.2 beschriebenen Fitnessfunktion “BigDistance”. Bei dieser Fitnessfunktion ist der Einfluss der Roboterhöhe über dem Boden auf den Fitnesswert deutlich kleiner.

4 Die Evolution von Laufalgorithmen mit SIGEL

PARAMETER	WERT
Zielsetzung	Evolution eines Laufalgorithmus'
Populationsgröße	100
Initialisierungsmethode	zufällig
Mutations-Wahrscheinlichkeit	73 %
Rekombinations-Wahrscheinlichkeit	25 %
Reproduktions-Wahrscheinlichkeit	2 %
Instruktions-Set	ADD, SUB, DIV, MUL, MIN, MAX, MOD, CMP, JMP, COPY, LOAD, SENSE, MOVE, DELAY, NOP
Terminal-Set	Ganzzahlige Konstanten $\in [-2^{31}, 2^{31} - 1]$
Selektionsmethode	Tournament-Selektion
Turniergröße	2 (Mutation, Reproduktion) bzw. 4 (Rekombination)
Tournaments pro Generation	2
Generationen	7066
Register	8 Register à 32 Bit
Befehlsdauer	1/1000 s
Max. Dauer von DELAY	5s
Schrittgröße "Step Size"	0.00005
"Random Seeds" (Simulation, GP)	0, 0
Simulationszeit	10s
Joint-Konstanten (Spring, Damper, Friction)	1000, 0, 0.38
Roboter-Startposition (X, Y, Z)	25, 0.2225, -25

Tabelle 4.2: Koza-Tableau des "BigDistance"-Experiments

4.2.2 Ergebnisse mit der Fitnessfunktion "BigDistance"

Das hier gezeigte Experiment mit der Fitnessfunktion "BigDistance" führte zu dem ersten erfolgreichen Laufprogramm im Rahmen dieser Diplomarbeit. Der in Abbildung 4.6 gezeigte Bewegungsablauf dieses Individuums ähnelt in gewisser Weise dem menschlichen Gang. Der Laufstil zeichnet sich durch sehr kleine Schritte aus, die in Einzelbildern nur schlecht wiedergegeben werden können. Trotz dieser kleinen Schrittweite legt das beste Individuum dieses Experiments eine Entfernung von rund 70cm zurück. In der Bildsequenz ist gut zu erkennen, dass der simulierte Roboter sich schon im Verlauf einer Sekunde deutlich von der Startposition entfernt. Diese Geschwindigkeit erreicht das Programm durch eine sehr schnelle Schrittfolge. Aus dem Koza-Tableau dieses Experiments in Tabelle 4.2 ist ersichtlich, dass eine Instruktionsdauer von 1/1000 Sekunde benutzt wird. Diese gilt für alle Instruktionen einschließlich des MOVE-Befehls. Aus diesem Grund war es dem GP-System möglich, einen Laufalgorithmus zu entwickeln, der den simulierten Roboter mit dieser vergleichsweise hohen Geschwindigkeit vorwärts bewegt.

Das beste Individuum hat eine Länge von 145 Instruktionen. Die maximal erlaubte Pro-

4.2 Ergebnisse mit dem simulierten ZORC-Roboter

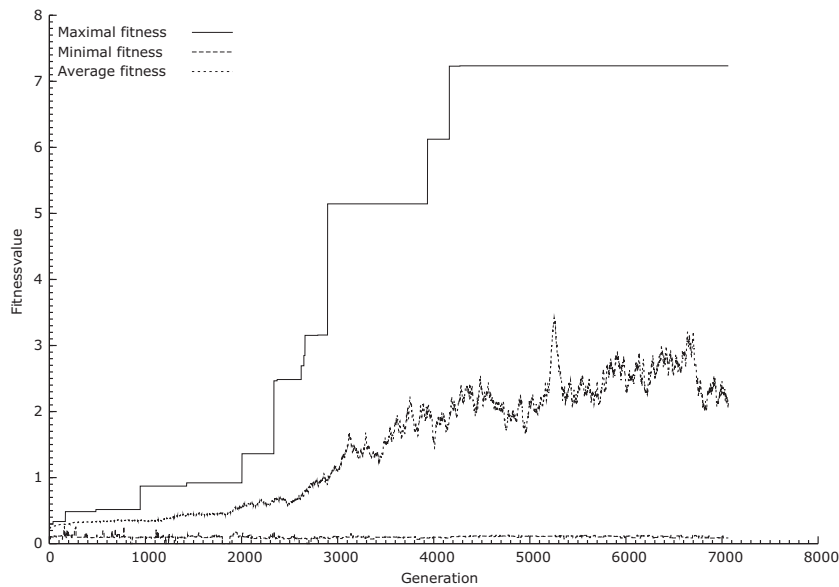


Abbildung 4.4: Das Fitnessdiagramm des Experiments mit der Fitnessfunktion “BigDistance”.

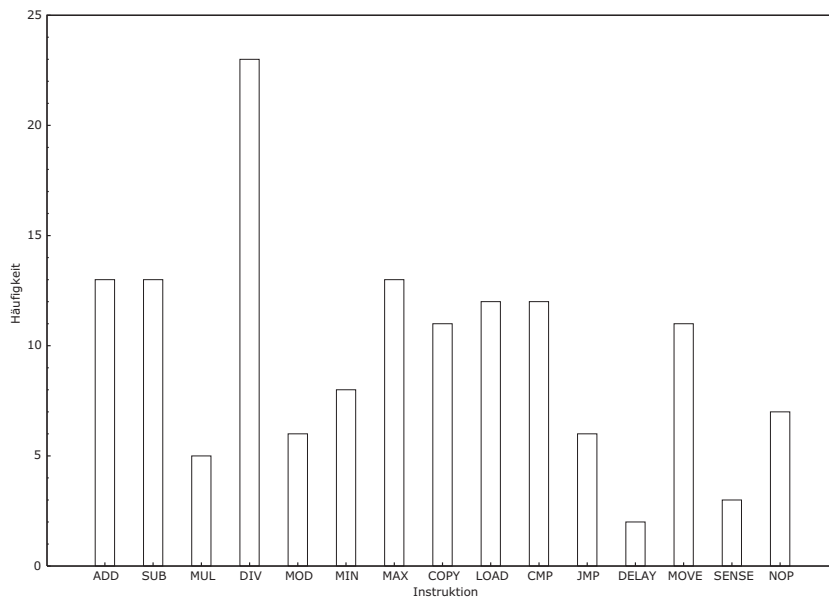


Abbildung 4.5: Eine Auswertung der Häufigkeit der einzelnen Instruktionen für das beste Individuum des “BigDistance”-Experiments.

grammlänge war in diesem wie allen anderen Experimenten durch 1024 Instruktionen beschränkt. In der Statistik in Abbildung 4.5 wurde die Häufigkeit der einzelnen Instruktionen in diesem Programm ausgewertet. Die MOVE-Instruktion hat im Vergleich zu den anderen Instruktionen eine durchschnittliche Häufigkeit. Dagegen ist überraschend, dass lediglich 3 Instruktionen vom Typ SENSE auftreten.

4 Die Evolution von Laufalgorithmen mit SIGEL

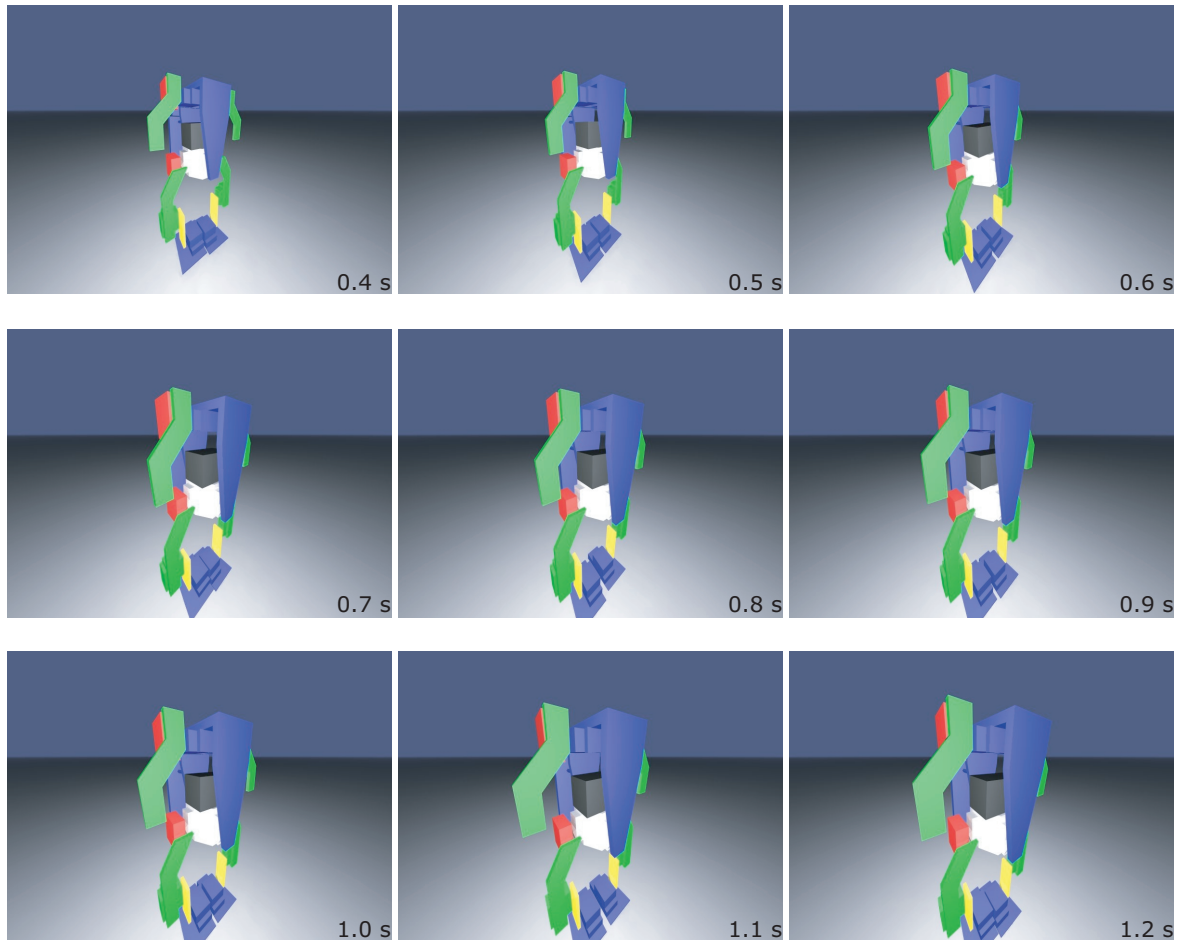


Abbildung 4.6: Der Bewegungsablauf des ersten Laufprogramms, das mit der ‘BigDistance’-Fitnessfunktion gelernt wurde und den Roboter in einem aufrechten Gang vorwärts bewegt (Kap. 4.2.2).

Für diese sowie die in Abbildung 6.4.6 gezeigte Statistik wurde auf eine Analyse der Programme hinsichtlich Introncode verzichtet. Beispielsweise können durch den Einsatz des Sprungbefehls `JMP` große Programmsequenzen übersprungen werden. Instruktionen innerhalb solcher Sequenzen kämen demzufolge nie zur Ausführung. Die gezeigten Statistiken sind daher nur eingeschränkt aussagekräftig.

Die Fitnessentwicklung des Experiments ist in Abbildung 4.4 dargestellt. Die X-Achse ist in den von SIGEL generierten Fitnessdiagrammen mit der Anzahl der Generationen beschriftet. Wie schon erwähnt wurde, kann diese Größe frei gewählt werden. Daher sollten die individuellen Parametereinstellungen bei Vergleichen der Diagramme von den in diesem Kapitel vorgestellten Experimenten ebenfalls in Betracht gezogen werden. Beim Vergleich mit dem Fitnessdiagramm des ‘QuadHeight’-Experiments in Abbildung 4.2 fällt jedoch der deutliche Unterschied in der Fitnessentwicklung, insbesondere der maximalen Fitness auf. Im erstgenannten Experiment tritt schon nach rund 600 Turnieren (bei 38 Turnieren pro Generation) keine weitere Verbesserung der maximalen Fitness ein. Für das gezeigte Experiment mit der

4.2 Ergebnisse mit dem simulierten ZORC-Roboter

Fitnessfunktion “BigDistance” findet jedoch erst nach rund 9000 Turnieren (bei 2 Turnieren pro Generation) keine weitere Verbesserung der maximalen Fitness mehr statt. Es kann daher angenommen werden, dass die Fitnessfunktion “BigDistance” sich wesentlich besser zur Steuerung der Evolution von Laufalgorithmen mit dem ZORC-Robotermodell eignet.

Die Ergebnisse dieses Experiments konnten leider nicht auf ZORC übertragen werden. Die verwendeten Sprachparameter nutzen die initialen SIGEL-Einstellungen und damit Instruktionslängen von 0.001 Sekunden. Die Implementierung von ZORCBRAIN fand zu diesem Zeitpunkt zeitgleich mit der Durchführung dieses Experiments statt. Daher war die in Kapitel 5.2 beschriebene Timing-Beschränkung des EYEBOT-Betriebssystem noch nicht bekannt. Eine Anpassung der Instruktionslängen auf 0.01 Sekunden führte dazu, dass die zuvor erfolgreichen Laufalgorithmen mit den veränderten Sprachparametern das Robotermodell in der Simulation zu Fall brachten. Aus diesem Grund war auch die Übertragung des evolvierten Laufalgorithmus’ auf ZORC wenig sinnvoll. Eine weitergeführte Evolution mit den angepassten Sprachparametern schien ebensowenig sinnvoll. Denn die ursprünglich besten Programme verhielten sich mit den um den Faktor 10 langsameren Instruktionen nicht länger sinnvoller als zufällig erzeugte Programme. Daher wurden neue Experimente mit der Fitnessfunktion “BigDistance-2” und angepassten Sprachparametern gestartet. Das hier vorgestellte Simulationsergebnis dieses Experiments hat jedoch bewiesen, dass SIGEL in der Lage ist, Laufalgorithmen für das ZORC-Robotermodell mittels Genetischer Programmierung zu evolvieren. Dieses Resultat wird in einer abschließenden Betrachtung der Simulationsphase in Kapitel 4.2.5 diskutiert.

4.2.3 Ergebnisse mit der Fitnessfunktion “BigDistance-2”

Das hier gezeigte Experiment ist das letzte aus einer Reihe von Experimenten, die “realistische” Instruktionsdauern verwendeten, d. h. Befehlslängen, die auf den EYEBOT übertragbar waren. Es wurden dabei zunächst Experimente mit der Fitnessfunktion “BigDistance” und im späteren Verlauf mit dem Nachfolger “BigDistance-2” durchgeführt. Das Koza-Tableau des hier vorgestellten Experiments in Tabelle 4.3 zeigt, dass überwiegend Instruktionsdauern von 1/100 Sekunden verwendet wurden. Lediglich die Instruktionen SENSE und MOVE nutzen Instruktionsdauern von 5/100 Sekunden, um die Reaktionszeiten der Roboterhardware zu berücksichtigen. Bei einer Sensormessung muss der Relais-Router umgestellt und zusätzlich eine gewisse Zeit abgewartet werden, bis sich der korrekte Messwert an den A/D-Eingängen des EYEBOT eingestellt hat. Bei einer durch eine MOVE-Instruktion ausgelösten Motorbewegung wird der betreffende Servomotor von der interruptgesteuerten Servocontroller-Routine von ZORCBRAIN angesteuert. Dieser Vorgang nimmt ebenfalls eine gewisse Zeit in Anspruch. Im Vergleich zu den Experimenten aus den Kapiteln 4.2.1 und 4.2.2 dauern die hier eingesetzten Instruktionen um den Faktor 10-50 länger, d. h. die Programme laufen wesentlich langsamer ab.

Die Startpopulation dieses Experiments wurde im Gegensatz zu allen anderen Experimenten, die im Rahmen dieser Arbeit durchgeführt wurden, nur teilweise von zufällig erzeugten Programmen gestellt. Von den 50 Individuen der initialen Population wurden lediglich 10 Programme zufällig von SIGEL erzeugt. Für die restlichen Individuen wurden die besten

4 Die Evolution von Laufalgorithmen mit SIGEL

PARAMETER	WERT
Zielsetzung	Evolution eines Laufalgorithmus'
Populationsgröße	50
Initialisierungsmethode	<i>siehe Text</i>
Mutations-Wahrscheinlichkeit	80 %
Rekombinations-Wahrscheinlichkeit	20 %
Reproduktions-Wahrscheinlichkeit	–
Instruktions-Set	ADD, SUB, DIV, MUL, MIN, MAX, MOD, CMP, JMP, COPY, LOAD, SENSE, MOVE, DELAY, NOP
Terminal-Set	Ganzzahlige Konstanten $\in [-2^{15}, 2^{15} - 1]$
Selektionsmethode	Tournament-Selektion
Turniergröße	2 (Mutation, Reproduktion) bzw. 4 (Rekombination)
Tournaments pro Generation	20
Generationen	176
Register	24 Register à 16 Bit
Befehlsdauer	1/100 s, MOVE und SENSE: 5/100 s
Max. Dauer von DELAY	0.25s
Schrittgröße "Step Size"	0.00005
"Random Seeds" (Simulation, GP)	6872, 701
Simulationszeit	5 - 20s (<i>siehe Text</i>)
Joint-Konstanten (Spring, Damper, Friction)	1000, 0, 0.38
Roboter-Startposition (X, Y, Z)	25, 0.2225, -25

Tabelle 4.3: Koza-Tableau des "BigDistance-2"-Experiments

Programme ausgewählt, die aus den bisher durchgeführten Experimenten hervorgegangen waren. Auf diese Weise sollte es SIGEL ermöglicht werden, auf bereits erzielten Lernerfolgen aufzubauen und den Lernprozess nicht ausschließlich mit zufällig erzeugten Programmen, d. h. "bei null" zu starten. Neben den hier vorgestellten Experimenten wurden auch aus weiteren, hier nicht gezeigten Experimenten Individuen für die Startpopulation ausgewählt. Die in der Regel unterschiedlichen Sprachparameter der einzelnen Experimente führten dazu, dass die meisten der importierten Individuen zu abweichenden Bewegungsabläufen führten, sobald sie in diesem Experiment ausgeführt wurden. Die Ursachen für dieses Verhalten werden in Kapitel 6.1 näher erläutert. Da für die Startpopulation dieses Experiments jedoch in erster Linie "vielversprechendes" genetisches Material erwünscht war, rückte dieser Faktor in den Hintergrund. Zusätzlich zu den besten Individuen der einzelnen Experimente wurde das handgeschriebene Programm aus Kapitel 6.2 in die Startpopulation aufgenommen.

Das beste aus diesem Experiment hervorgegangene Individuum ist in Abbildung 4.8 dargestellt. Es ist unschwer zu erkennen, dass es sich bei diesem Programm um keinen Laufalgorithmus handelt, d. h. das Robotermodell wird nicht in einem kontinuierlichen Bewegungsablauf fortbewegt. Der Roboter bewegt sich in geringem Umfang auf der Stelle, verliert jedoch

4.2 Ergebnisse mit dem simulierten ZORC-Roboter

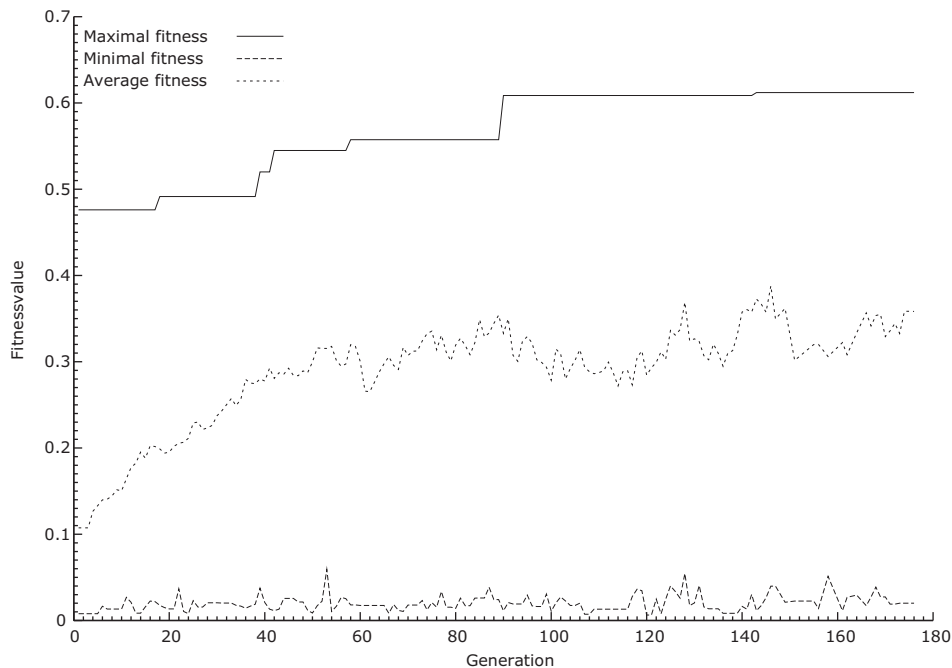


Abbildung 4.7: Das Fitnessdiagramm des Experiments mit der Fitnessfunktion “BigDistance-2”

relativ früh das Gleichgewicht und fällt zu Boden. Im Gegensatz zu den bereits gezeigten Programmen, deren Bewegungsabläufe sich in sehr kurzen Zyklen wiederholen, erstreckt sich der hier gezeigte Bewegungsablauf über mehrere Sekunden. Aus diesem Grund wurden für die Einzelbilder Abstände von 0.5 Sekunden gewählt. In der Anfangsphase schwingt der Torso des Robotermodells zunächst vor und zurück. Diese Bewegung verläuft relativ langsam und ist in den Bildern 1 bis 6 zu erkennen. Im Anschluss daran werden diverse Arm- und Bein-Motoren bewegt und der Roboter fällt nach ca. 6 Sekunden auf den Rücken. Dieses für den menschlichen Betrachter kaum sinnvolle Ergebnis ist typisch für alle Experimente, die mit den hier verwendeten Sprachparametern durchgeführt wurden. Viele dieser Programme fielen nach einer anfangs stabilen Phase zu Boden, sobald zufällige Motorbewegungen ausgeführt wurden, die den Roboter destabilisierten. Die wahrscheinliche Ursache für diese Resultate wird im Vergleich mit den Laufalgorithmen, die auf der realen ZORC-Architektur evolviert wurden, in Kapitel 6.5 diskutiert.

Bei diesem wenig erfolgreichen Algorithmus lässt sich zumindest positiv vermerken, dass das handgeschriebene Programm aus Kapitel 6.2 einen positiven Einfluss auf den Lernprozess gehabt hat. Der hier nicht abgebildete Programmcode des besten Individuums weist deutliche Ähnlichkeiten zu diesem Programm auf und lässt sich damit als ein direkter Nachfahre dieses “Stehprogramms” identifizieren. Ausgehend von der Standposition wurde das Programm um die in der Abbildung ersichtlichen Motorbewegungen ergänzt, so dass zumindest ein gewisser Lernerfolg zu beobachten ist. Aus dem Fitnessdiagramm dieses Experiments in Abbildung 4.7 lässt sich im Gegensatz zu den anderen gezeigten Experimenten weniger eindeutig schließen, dass die Evolution zum Erliegen gekommen ist. Möglicherwei-

4 Die Evolution von Laufalgorithmen mit SIGEL

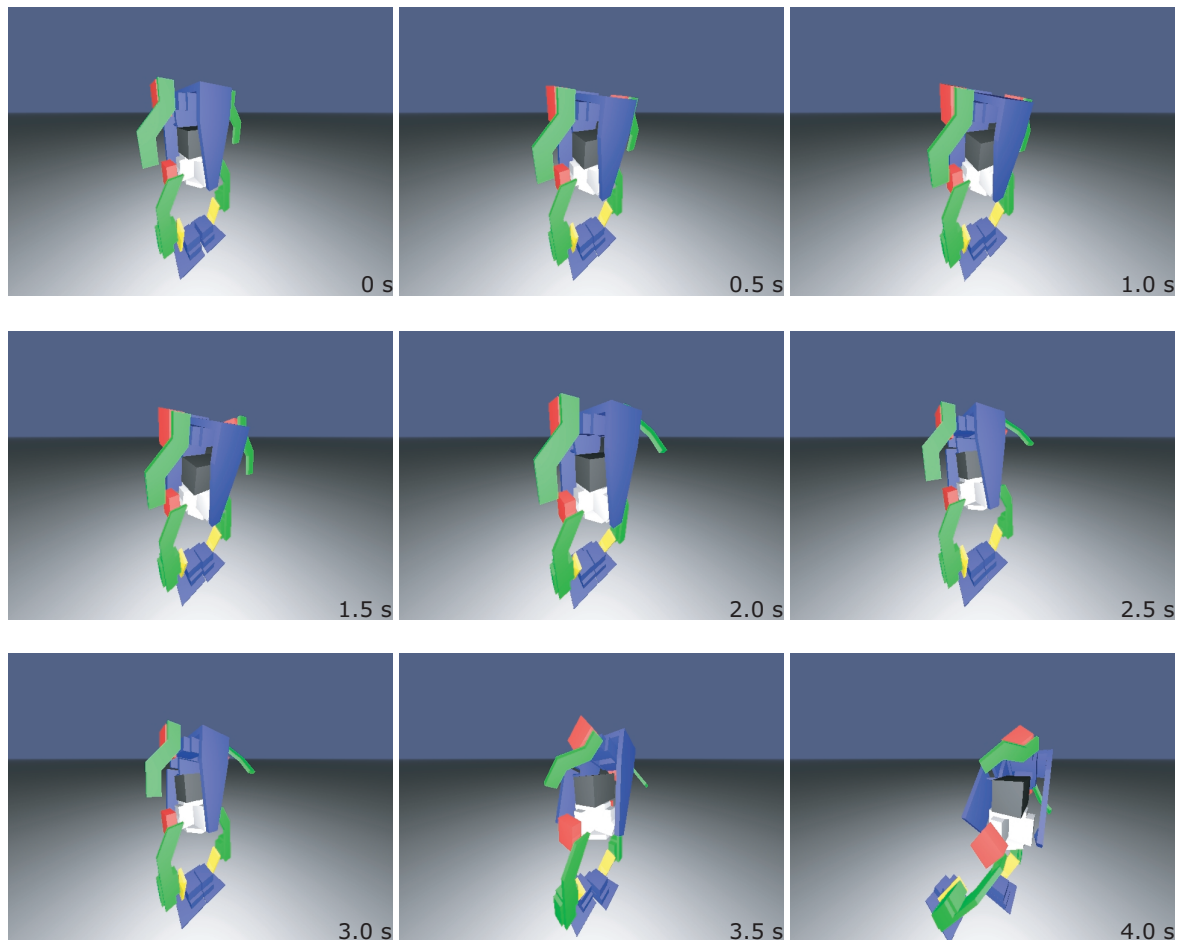


Abbildung 4.8: Der Bewegungsablauf des besten Individuums, dass aus dem gezeigten Experiment mit der Fitnessfunktion “BigDistance-2” hervorgegangen ist. Auf Grund des vergleichsweise langsamen Bewegungsablaufes wurden die Einzelbilder in Abständen von 0.5 Sekunden aufgenommen.

se hätte eine fortgeführte Evolution zu besseren Ergebnissen geführt, aus Zeitmangel wurde dieses Experiment in der Simulationsumgebung jedoch nicht weiter verfolgt.

Stattdessen wurden die erzielten Ergebnisse als Ausgangspunkt für die Evolution von Laufalgorithmen auf der realen Roboterarchitektur verwendet. Die Fortführung dieses Experimentes mit ZORC an Stelle der Simulationsumgebung wird in Kapitel 6.4.6 beschrieben.

4.2.4 Ergebnisse mit der Fitnessfunktion “Stepper”

Die Bildsequenz in Abbildung 4.10 zeigt den Bewegungsablauf des besten Individuums, das aus dem hier gezeigten Experiment mit der Fitnessfunktion “Stepper” hervorgegangen ist. Es ist deutlich zu erkennen, dass der evolvierte Laufalgorithmus den Roboter bereits innerhalb der ersten Sekunde der Simulationszeit zu Fall bringt. Das Fitnessdiagramm in Abbildung

4.2 Ergebnisse mit dem simulierten ZORC-Roboter

PARAMETER	WERT
Zielsetzung	Evolution eines Laufalgorithmus'
Populationsgröße	50
Initialisierungsmethode	zufällig
Mutations-Wahrscheinlichkeit	50 %
Rekombinations-Wahrscheinlichkeit	48 %
Reproduktions-Wahrscheinlichkeit	2 %
Instruktions-Set	ADD, SUB, DIV, MUL, MIN, MAX, MOD, CMP, JMP, COPY, LOAD, SENSE, MOVE, DELAY, NOP
Terminal-Set	Ganzzahlige Konstanten $\in [-2^{15}, 2^{15} - 1]$
Selektionsmethode	Tournament-Selektion
Turniergröße	2 (Mutation, Reproduktion) bzw. 4 (Rekombination)
Tournaments pro Generation	5
Generationen	2276
Register	24 Register à 16 Bit
Befehlsdauer	1/100 s
Max. Dauer von DELAY	0.5s
Schrittgröße "Step Size"	0.00005
"Random Seeds" (Simulation, GP)	77, 77
Simulationszeit	10s
Roboter-Startposition (X, Y, Z)	25, 0.225, -25
Joint-Konstanten (Spring, Damper, Friction)	1000, 0, 0.38

Tabelle 4.4: Koza-Tableau des "Stepper"-Experiments

4.9 zeigt, dass nach anfänglichen Verbesserungen der besten und maximalen Fitnesswerte die Evolution zum Erliegen gekommen ist. Für diese Deutung spricht ebenfalls die geringe phänotypische Diversität in der Population, gemessen an den Fitnesswerten der Individuen. Fast die gesamte Population wird durch Kopien weniger Individuen mit identischen Fitnesswerten dominiert.

Neben dem hier gezeigten Experiment wurden mehrere Experimente mit der "Stepper"-Fitnessfunktion durchgeführt. Bei diesen wurden leicht abweichende Parameter definiert, insbesondere unterschiedliche Wahrscheinlichkeiten für die genetischen Operatoren und abweichende "Seeds" für GP-System und Simulationsumgebung. Die Ergebnisse glichen im Wesentlichen den Resultaten des hier gezeigten Experiments, so dass davon ausgegangen werden kann, dass die Fitnessfunktion in ihrer aktuellen Form ungeeignet ist, die Evolution von Laufalgorithmen zu steuern. Die abgebildete Bewegungssequenz lässt erkennen, dass der Laufalgorithmus offensichtlich bemüht ist, den Roboter nur mit einem Fuß den Boden berühren zu lassen. Der "Sprung" des Roboters lässt sich als Versuch deuten, die Distanz zu maximieren, bevor die Evaluierung auf Grund des Unterschreitens der erforderlichen Höhe von 2/3 der Ausgangshöhe abgebrochen wird. Damit erfüllt das gezeigte Laufprogramm bei-

4 Die Evolution von Laufalgorithmen mit SIGEL

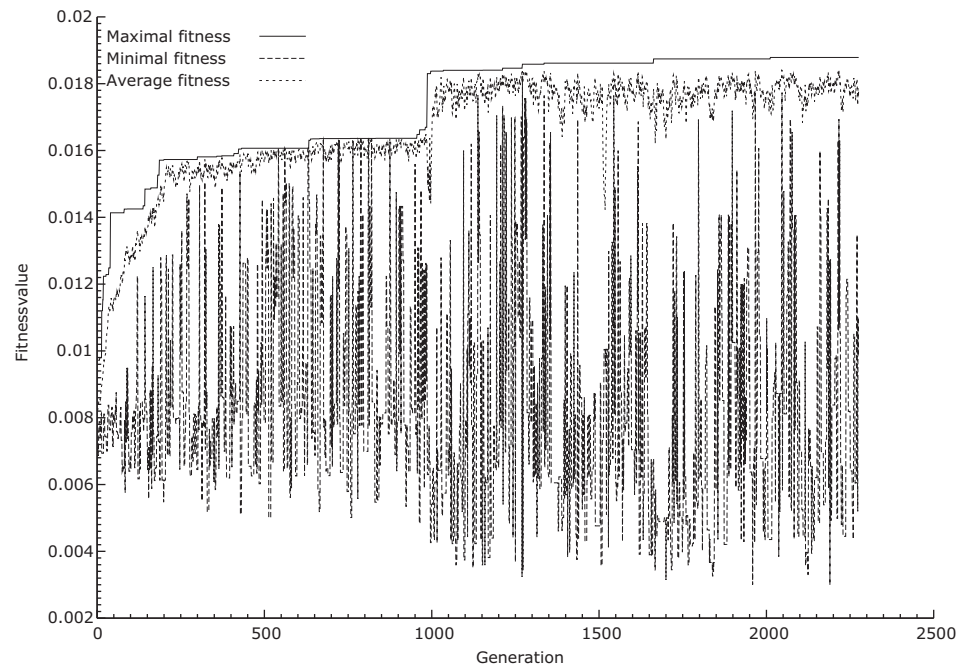


Abbildung 4.9: Das Fitnessdiagramm des Experiments mit der Fitnessfunktion “Stepper”

de Kriterien der Fitnessfunktion, ohne jedoch den Roboter im Gleichgewicht zu halten. Offensichtlich ist der Höhen-Faktor h bzw. h_{norm} der vorangegangenen Fitnessfunktionen auch in der Fitnessfunktion “Stepper” notwendig, um die Höhe des Roboters und damit das Gleichgewicht zu stabilisieren.

Bei der Verwendung einer zunehmenden Anzahl von Fitnesskriterien in der Fitnessfunktion wird die Gewichtung und Abstimmung der einzelnen Kriterien zunehmend schwieriger. Die Erfahrungen mit den hier vorgestellten Fitnessfunktionen, die im Rahmen der Diplomarbeit implementiert wurden, zeigen, dass dies insbesondere auf so gegensätzliche Faktoren wie Distanz und Roboterhöhe zutrifft. Gerade beim Einsatz von SIGEL ist diese Justierung der einzelnen Fitnesskriterien durch den Einsatz der rechenaufwändigen physikalischen Simulation ein sehr langwieriger Prozess. Da mit den Ergebnissen der “BigDistance”-Experimente bereits erfolgreich Laufalgorithmen evolviert werden konnten und damit für die Übertragung auf die reale Roboterarchitektur zur Verfügung standen, wurde die Optimierung und Weiterentwicklung der Fitnessfunktion “Stepper” nicht weiter verfolgt.

4.2.5 Zusammenfassende Betrachtung der Simulationsergebnisse

In diesem Kapitel konnte gezeigt werden, dass das SIGEL-System mit den im Rahmen der Diplomarbeit implementierten Erweiterungen in der Lage ist, Laufalgorithmen für humanoide Roboterarchitekturen mittels Genetischer Programmierung zu evolvierten.

Dieses Ergebnis ist um so bemerkenswerter, stellt man es den Resultaten vergleichbarer Un-

4.2 Ergebnisse mit dem simulierten ZORC-Roboter

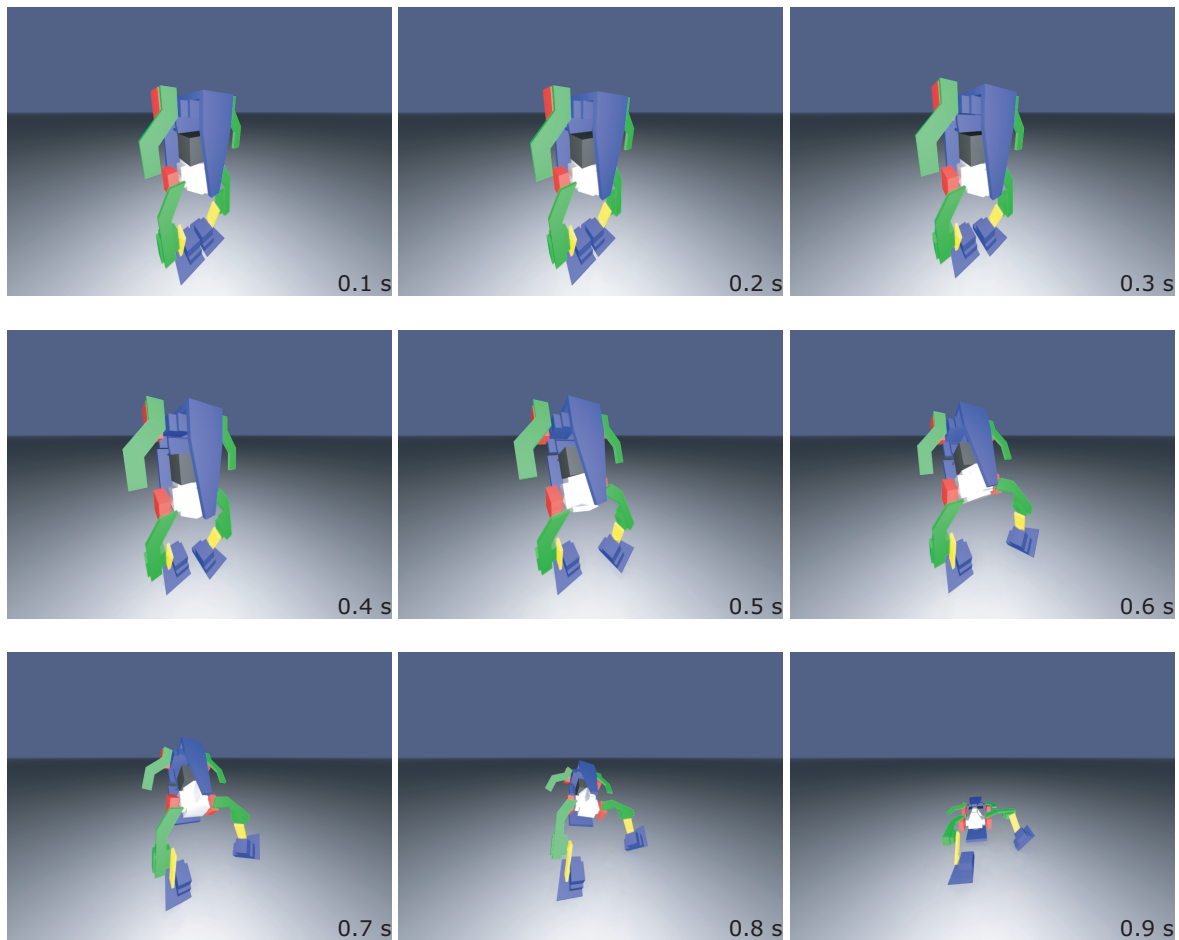


Abbildung 4.10: Die Grafik zeigt den Bewegungsablauf des besten Individuums des Experiments mit der “Stepper”-Fitnessfunktion

tersuchungen gegenüber. Beispielsweise berichten Parker und Mills in [18], wie für einen sechsbeinigen, insektenartigen Roboter ein Laufalgorithmus entwickelt wurde. Im Vergleich zu ZORC besitzt der dort beschriebene Roboter eine wesentlich stabilere Architektur, die praktisch nicht umfallen kann. Der beschriebene Ansatz nutzt dennoch ein stark vereinfachtes Modell zur Steuerung des Roboters. Dieser Roboter, der ebenfalls zunächst in einer Simulation trainiert wurde, wird durch Binärstrings gesteuert. Der Inhalt einer bestimmten Position im String legt fest, ob z. B. ein Bein gehoben bzw. in die vordere oder hintere Position bewegt werden soll. Eine Sequenz der Binärstrings, die zusammengekommen die Bewegung des Roboters kodieren, wurde zur Fortbewegung des Roboters “abgespielt”. Dieser Ansatz nutzt also nicht nur eine Kodierung, bei der die Bewegung des Roboters impliziter Bestandteil ist. Zusätzlich wird durch die stark vereinfachte Steuerung der Beine (auf / ab bzw. vorne / hinten) die Aufgabenstellung für den Genetischen Algorithmus wesentlich vereinfacht.

Verglichen mit Parker und Mills besitzt der im Rahmen dieser Arbeit eingesetzte Genetische Algorithmus in Form von Genetischer Programmierung ungleich mehr Freiheitsgrade. So

4 Die Evolution von Laufalgorithmen mit SIGEL

wird beispielsweise keine Bewegung durch die Art der Kodierung der GP-Programme implizit vorgegeben. Die MOVE-Instruktionen müssen vom GP-System selbständig eingesetzt werden. Außerdem wird keine starre Bewegungs- bzw. Befehlssequenz vorgegeben. In SIGEL werden Individuen zwar ebenfalls zyklisch ausgeführt, durch den Einsatz von Vergleichen und Sprungbefehlen sowie Verzögerungen sind diese Programme jedoch ungleich flexibler in ihrer Ausführung. Selbst die implizite Information über das Robotermodell, die Parker und Mills in gewissem Maße in ihrem Ansatz nutzen, ist bei SIGEL nicht gegeben. Beim erstgenannten Ansatz beschreibt jeder Eintrag des Binärstrings eine Beinposition, zudem kodieren ungerade Indizes die linken und gerade Indizes die rechten Beine. In SIGEL muss der Lernprozess erst “herausfinden”, welche Motoradressen einen Motor bewegen, dessen Bewegung gerade sinnvoll ist. Darüber hinaus müssen die Stellwinkel gelernt werden, die eine sinnvolle Bewegung des Roboters erlauben. Allein die theoretische Einschränkung der anzufahrenden Winkel auf 2 für das Laufen sinnvolle Motorstellungen würde für das SIGEL-System den Lernprozess erheblich vereinfachen, da der Suchraum für das GP-System wesentlich verkleinert werden würde. Dabei ist es zunächst nicht relevant, ob mit einer solch starken Einschränkung überhaupt noch Laufalgorithmen realisiert werden können. Die in SIGEL realisierbaren Freiheitsgrade stellen vielmehr eine Stärke des hier vorgestellten Ansatzes dar. Durch möglichst wenig Vorgaben des Benutzers bzw. Experimentators stehen dem GP-System wesentlich mehr Wege offen, eine effektive Lösung zu finden, als es bei stark eingeschränkten Lösungen möglich wäre. Betrachtet man alle genannten Faktoren, dann wird jedoch auch deutlich, um wie viel schwieriger sich der Lernprozess in der eingesetzten Form verglichen mit anderen Ansätzen gestaltet.

Im Vergleich zu den bislang mit der ZORC-Architektur erzielten Ergebnissen stellen die hier vorgestellten Resultate einen wesentlichen Fortschritt dar. In [1] wurde eine Evolutionäre Strategie eingesetzt, um einen von Hand geschriebenen Laufalgorithmus zu optimieren. Die dabei eingesetzte Kodierung ähnelt der von Parker und Mills eingesetzten Variante. Ein Vektor aus Integerkonstanten kodiert die Geschwindigkeit und Positionen der Servomotoren sowie Verzögerungen. Auch hier sind vom Lernprozess wesentlich weniger Freiheitsgrade zu bewältigen gewesen. Die Evolutionäre Strategie wurde in [1] zwar direkt auf den realen Roboter angewandt und dadurch die Problematik der Umsetzung der Simulationsergebnisse auf den realen Roboter umgangen. Der notwendige Aufwand, den die manuelle Kodierung eines Laufalgorithmus’ als Ausgangspunkt für den Lernprozess bedeutet, ist jedoch nicht zu unterschätzen. Zudem schränkt die verwendete Kodierung die Anzahl möglicher Steuerungsprogramme, die durch die Integer-Strings beschrieben werden können, ein. Durch die Wahl der Kodierung wird in gewisser Hinsicht eine Vorauswahl unter den realisierbaren Laufalgorithmen getroffen und mit dieser Verkleinerung des Suchraumes möglicherweise effizientere Laufalgorithmen von vornherein ausgeschlossen. Bei der Verwendung von Genetischer Programmierung wird diese potentielle Fehlerquelle ausgeschlossen, da die vollständige Menge der Programme, die mit dem verwendeten GP-System realisierbar sind, den Suchraum bildet. Der Einsatz einer Simulation im Rahmen der Diplomarbeit erlaubte es zudem, wesentlich mehr Programme des Suchraumes “auszuprobieren”, als es allein auf der realen Roboterarchitektur in einem realistischen Zeitraum möglich gewesen wäre.

5 Die Programmierung von ZORC

In Kapitel 2 wurde bereits erläutert, dass der autonome Betrieb von ZORC durch das EYEBOT-Mikrocontrollerboard ermöglicht wird. In diesem Kapitel wird die Programmierung des EYEBOT und das auf dem Mikrocontroller ausgeführte Programm ZORCBRAIN beschrieben. Dadurch wird ZORC in die Lage versetzt, die von SIGEL evolvierten GP-Programme selbständig auszuführen.

5.1 Überblick über die Implementierung von ZORCBRAIN

Bei der Wahl von SIGEL als Simulationsumgebung wurde nicht zuletzt der Schritt von der Simulation auf die reale Roboterarchitektur in Betracht gezogen. Die portable Programmierung von SIGEL erfolgte in der Programmiersprache C++ unter Verwendung der frei erhältlichen GNU Compiler Collection (GCC) [22]. Der Compiler ist für zahlreiche Rechnerplattformen verfügbar, unter anderem auch für den MC68332¹. Daher war die Umsetzung von SIGEL auf den EYEBOT-Mikrocontroller prinzipiell ohne großen Aufwand möglich. Einschränkungen bei der Umsetzung ergaben sich lediglich bei den Komponenten des SIGEL-Systems, die zusätzliche Ressourcen benötigen. Darunter fallen beispielsweise die grafische Oberfläche, der Zugriff auf Dateien oder die Netzwerk-Schnittstelle. Die Anpassungen des GP-Subsystems von SIGEL für den Einsatz auf ZORC werden in Kapitel 5.2 geschildert. Mit diesen Anpassungen war es schließlich möglich, GP-Programme auf den realen Roboter zu importieren und dort auszuführen.

Kapitel 5.3 beschreibt die eingesetzte Hardwareabstraktionsklasse `CEyeBotLoLevel`. Durch diese wird die Ansteuerung der Aktuatoren und die Abfrage der Sensoren von ZORC implementiert sowie der Zugriff auf die diversen Schnittstellen des EYEBOT ermöglicht, ohne auf hardwarenahe Routinen des EYEBOT-Betriebssystems zurückgreifen zu müssen. In diese Kategorie fallen u. a. die seriellen Schnittstellen und die Abfrage der Taster des EYEBOT (s. a. Abb. 5.2). Die in diesem Kontext ebenfalls beschriebene Klasse `ZORC_RS232Stream` vereinfacht den Zugriff auf die serielle Schnittstelle des EYEBOT.

Zentrale Klasse des ZORCBRAIN-Programms ist `CZorcBrain`. Sie steuert die Interaktion von ZORC und dem Benutzer bzw. die Fernsteuerung des Roboters durch SIGEL. Die einzelnen Programmkomponenten von ZORCBRAIN – z. B. Testfunktionen oder der Interpreter – werden durch diese Klasse aufgerufen. `CZorcBrain` ist Gegenstand von Kapitel 5.4. Das Zusam-

¹Der Motorola MC68332-Mikrocontroller wird auf dem EYEBOT eingesetzt.

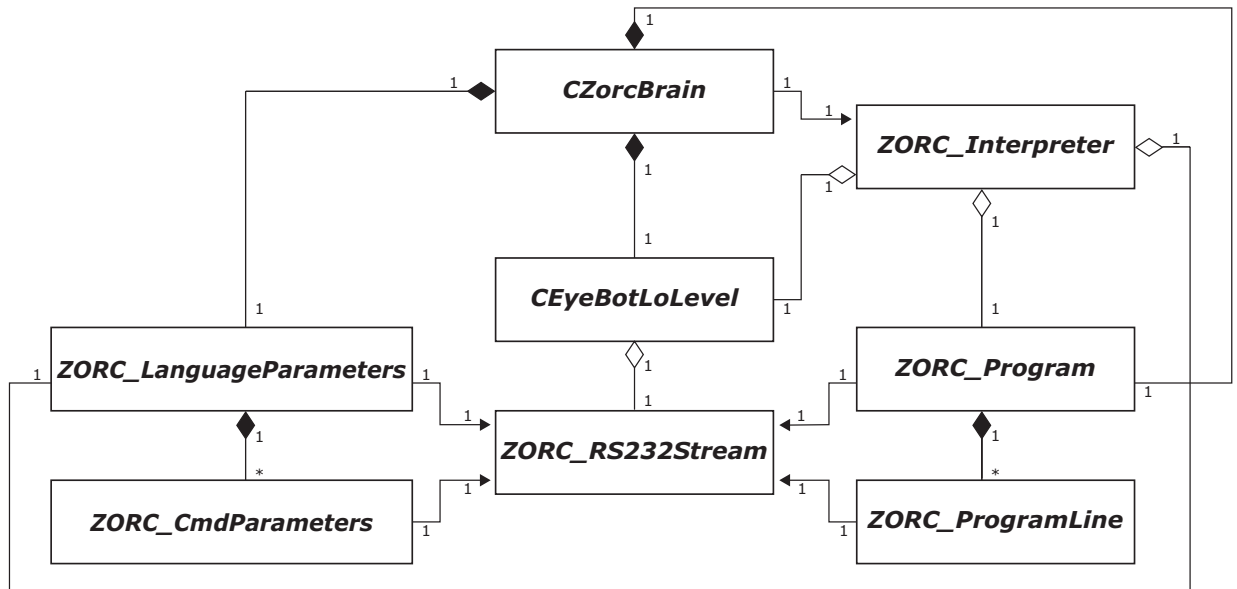


Abbildung 5.1: Das UML-Diagramm stellt das in diesem Kapitel behandelte Programmpaket ZORCBRAIN mit allen beteiligten Klassen und Abhängigkeiten dar.

Beispiel aller Klassen des ZORCBRAIN-Programms ist in dem UML-Diagramm in Abbildung 5.1 dargestellt.

5.2 Implementierung des SIGEL-Subsystems

ZORC sollte durch das ZORCBRAIN-Programm in die Lage versetzt werden, SIGEL GP-Programme auszuführen. Die Programme sollten i. d. R. zuvor in SIGEL evolviert und anschließend auf den Roboter übertragen werden. Zu diesem Zweck sind folgende Klassen der SIGEL-Implementierung auf den EYEBOT-Mikrocontroller portiert und angepasst worden:

- ZORC_CmdParameters (Portierung von *SIG.CommandParameters*)
- ZORC_Interpreter (Portierung von *SIG.Interpreter*)
- ZORC_LanguageParameters (Portierung von *SIG.LanguageParameters*)
- ZORC_Program (Portierung von *SIG.Program*)
- ZORC_ProgramLine (Portierung von *SIG.ProgramLine*)
- ZORC_Register (Portierung von *SIG.Register*)

Von diesen Klassen haben insbesondere ZORC_LanguageParameters, ZORC_Program und ZORC_Interpreter eine zentrale Bedeutung für den Einsatz auf ZORC. Zur Repräsentation der Sprachparameter (Language Parameter) wird die Klasse ZORC_LanguageParameters

benötigt. Die Sprachparameter beinhalten

- Eigenschaften von GP-Instruktionen (z. B. Befehlsdauer)
- Eigenschaften der durch das GP-System simulierten CPU (Registerbreite und Anzahl der verfügbaren Register)

Einzelheiten zu den Sprachparametern finden sich in den Kapiteln 4.4.2 und 6.2 in [6]. Die Repräsentation von GP-Programmen wird von der Klasse `ZORC_Program` übernommen. Die Klasse `CZorcBrain` besitzt zur Repräsentation dieser Daten je eine Instanz beider Klassen. Die Klasse `ZORC_Interpreter` ist von zentraler Bedeutung für die Ausführung von GP-Programmen, d. h. Instanzen der Klasse `ZORC_Program`. Zur Ausführung des aktuell geladenen GP-Programms wird von `CZorcBrain` eine Instanz der Klasse `ZORC_Interpreter` erzeugt. Alle weiteren aufgeführten Klassen werden von diesen drei zentralen Klassen intern benötigt, wie das UML-Klassendiagramm in Abbildung 5.1 verdeutlicht.

Im Zuge der Portierung auf den EYEBOT-Mikrocontroller waren einige Änderungen nötig. Zum Einen stehen auf dem Mikrocontroller keine Dateien zur Verfügung. Daher mussten alle Methoden mit Dateizugriff eliminiert werden oder, wenn sie für die Programmausführung auf ZORC unabdingbar benötigt wurden, umgeschrieben werden. Das Laden von Programmen bzw. Programmzeilen und der Sprachparameter wurde abgeändert, so dass alle vormals aus Dateien geladenen Daten über die serielle RS232-Schnittstelle des EYEBOT empfangen werden können. Die Dateiformate blieben von dieser Änderung unberührt. Daher können z. B. SIGEL-Programme mit einem einfachen Terminalprogramm als simpler Text² an ZORC übertragen werden. Die Fernsteuerung von ZORC durch SIGEL gestaltet sich auf diese Weise ebenfalls unkompliziert (s. a. Kapitel 6.4.2). Zur Übertragung der GP-Programme an ZORC sendet SIGEL die gleichen Daten, die zuvor in einer Datei gespeichert wurden, an die serielle Schnittstelle.

Eine weitere Änderung, die an nahezu allen verwendeten SIGEL-Klassen im Zuge der Portierung vorgenommen werden musste, war die Umstellung von QT (Kap. 3.6 in [6]) auf STL (Teil III in [23]). Die Klassenbibliothek QT wurde in SIGEL nicht nur für die grafische Benutzeroberfläche eingesetzt. Die von der Bibliothek zur Verfügung gestellten Implementierungen für z. B. Listen und Vektoren wurden auch in Klassen eingesetzt, die nicht an der Benutzeroberfläche beteiligt waren. Daher nutzten fast alle der portierten Klassen etliche Hilfsklassen der QT-Bibliothek. Eine Portierung von QT – das ebenfalls im Quelltext vorlag – kam aus Platzgründen nicht infrage, da der auf dem Mikrocontroller verfügbare Speicher mit hoher Wahrscheinlichkeit nicht ausgereicht hätte. Außerdem wäre der Codeumfang von ZORC-BRAIN durch die komplexe Bibliothek unverhältnismäßig angestiegen. Alle eingesetzten QT-Klassen wurden aus diesem Grund durch Klassen der zur C++ – Spezifikation gehörenden STL-Bibliothek ersetzt. Diese Bibliothek stand bereits für den MC68332 zur Verfügung, da sie im GCC-Compiler enthalten war. Alle verwendeten QT-Klassen besaßen bis auf kleinere Abweichungen äquivalente STL-Implementierungen. Aus diesem Grund verlief die Umstellung relativ problemlos und nur wenige Methoden mussten vollständig umgeschrieben werden.

²Die Exportfunktion von SIGEL speichert GP-Programme als ASCII-Text.

5 Die Programmierung von ZORC

Eine wesentliche Änderung betraf schließlich die Klasse `ZORC_LanguageParameters` und die davon eingesetzte Klasse `ZORC_CmdParameters`. Letztere dienen der Spezifikation unterschiedlicher Parameter einer SIGEL GP-Instruktion. Darunter fällt u. a. die Länge, die eine Instruktion bei ihrer Ausführung auf der simulierten CPU benötigt. Die CPU wird von der Klasse `ZORC_Interpreter` – bzw. in SIGEL von der Klasse `SIGEL_Interpreter` – simuliert. In SIGEL ist die während der Ausführung des Befehls real verstrichene Zeit nicht relevant, da auch die Simulation nicht in Echtzeit abläuft. Auf ZORC hingegen wird die Rolle der Simulation vom realen Roboter übernommen, d. h. die Bewegung des Roboters läuft in Echtzeit ab. Das bedeutet, dass auch die Ausführung der Befehle durch den Interpreter in Echtzeit ablaufen muss. Eine GP-Instruktion darf demzufolge nicht mehr Zeit im Interpreter in Anspruch nehmen, als die in den `ZORC_CmdParameters` festgelegte Instruktionsdauer – aber auch nicht weniger.

In `ZORC_Interpreter` wird daher wie folgt verfahren:

1. Eine GP-Instruktion wird durch den Interpreter ausgeführt,
2. Die (simulierte) CPU-Zeit wird um die Dauer der Instruktion heraufgezählt,
3. Solange in einer Warteschleife pausiert, bis die seit dem letzten Befehl real verstrichene Systemzeit auf dem EYEBOT der definierten Instruktionsdauer entspricht

Die für den 3. Schritt notwendige Abfrage des EYEBOT-Timers ist durch Limitationen des EYEBOT-Betriebssystems auf eine Genauigkeit von $\frac{1}{100}$ Sekunde beschränkt. Daher können Befehle mit einer Dauer von weniger als 0.01 Sekunden nicht zuverlässig von der Klasse `ZORC_Interpreter` behandelt werden. Für auf ZORC übertragene SIGEL-Programme gilt daher die Beschränkung, dass die Befehlsdauer der zum Programm gehörigen Sprachparameter mindestens 0.01 Sekunden betragen muss. Kürzere Befehlsdauern werden beim Laden der Sprachparameter auf ZORC mit einer Fehlermeldung zurückgewiesen.

5.3 Implementierung der Hardwareabstraktionsklasse `CEyeBotLoLevel`

In der Klasse `CEyeBotLoLevel` wurden sämtliche Funktionen zur Interaktion mit der Roboterhardware gekapselt. Alle weiteren an ZORCBRAIN beteiligten Klassen greifen auf die Hardware von ZORC ausschließlich über diese Klasse zu. Durch den Einsatz dieser Abstraktionsebene wird zudem eine eventuelle spätere Portierung des ZORCBRAIN-Programms auf eine andere Hardwareplattform erleichtert.

Im Überblick bietet diese Klasse Methoden zur

- Ansteuerung der Servomotoren
- Abfrage der Servopositions-, Neigungs-, Fußtaster- & Stromaufnahme-Sensoren

5.3 Implementierung der Hardwareabstraktionsklasse *CEyeBotLoLevel*

- RS232-Kommunikation
- Abfrage der Taster des EYEBOT
- Interaktion mit dem Anwender

Die detaillierte Dokumentation dieser umfangreichen Klasse mit allen Methoden und Datenstrukturen befindet sich in Anhang D. Die folgenden Kapitel behandeln die in Kapitel 2 beschriebenen Erweiterungen von ZORC aus Sicht der Software, erläutern einige Details zur Ansteuerung der Servomotoren und der Adressierung von Servomotoren und Sensoren. Außerdem wird kurz auf die Klasse `ZORC_RS232Stream` eingegangen, die einen unkomplizierten Gebrauch der RS232-spezifischen Methoden von `CEyeBotLoLevel` ermöglicht.

5.3.1 Ansteuerung der Servomotoren

Die 13 auf ZORC eingesetzten Servomotoren sind zum Teil direkt am EYEBOT angeschlossen, teilweise werden die Motoren von dem in 2.6 beschriebenen externen Servocontroller MINI-SSC II angesteuert. Diese externe Ansteuerung betrifft die vier Arm- und Schultermotoren. Die Ansteuerung aller Motoren übernimmt einheitlich die Methode `setServo()`. Dem Anwender bleibt dabei verborgen, ob ein externer oder interner Servomotor angesteuert wird. Die grundverschiedene Ansteuerung beider Typen wird von `CEyeBotLoLevel` gehandhabt. Anhand der ID, die jeden Servomotor identifiziert, wird unterschieden, ob es sich um einen internen oder externen Motor handelt. Im ersten Fall wird eine Routine des EYEBOT-Betriebssystems zur Ansteuerung des Servomotors eingesetzt. Im zweiten Fall wird über die serielle Schnittstelle, an die der externe Servocontroller angeschlossen ist, das Kommando zum Bewegen des entsprechenden Servomotors gesandt. Das hierbei benutzte Protokoll wird in [13] beschrieben.

Die schon zuvor erwähnte, interruptgesteuerte Servocontroller-Methode wird ebenfalls in dieser Klasse implementiert. Durch den Aufruf einer Routine des EYEBOT-Betriebssystems wird die Methode `servoController()` als Interrupt installiert. Die hierbei verwendete Konstante `kServoCtrlQuantum` (s. Anhang D) bestimmt, mit welcher Frequenz der Interrupt aufgerufen wird. Die Frequenz berechnet sich als $kServoCtrlQuantum / 100$ Hertz, d. h. ein Interrupt kann maximal 100 mal pro Sekunde aufgerufen werden. In der Methode `servoController()` werden die Servomotoren, die aktuell in Bewegung sind, in kleinen Schritten auf die gewünschte Endposition gebracht. Die Anzahl der Zwischenschritte wird durch die Konstanten `kServoSlowSpeed` und `kServoFastSpeed` (s. Anhang D) bestimmt. Die Bewegung der Servomotoren ist umso ruckhafter, je kleiner diese Konstante gewählt ist und je weniger Schritte damit zwischen der aktuellen und der anzusteuernenden Servoposition liegen. Die durch die Methode `setServo()` gesteuerte Servobewegung, die u. a. bei der Interpretation einer MOVE-Instruktion zum Einsatz kommt, nutzt die Konstante `kServoFastSpeed`. Die durch diese Konstante definierte Geschwindigkeit wurde so gewählt, dass der Roboter durch die Bewegung der Motoren nicht durch zu große Beschleunigungen destabilisiert wird, die Bewegung des Roboters auf der anderen Seite dennoch nicht zu träge wird.

5 Die Programmierung von ZORC

Der Servocontroller-Interrupt wird durch `CEyeBotLoLevel` gesteuert. Zu diesem Zweck teilen `servoController()` und die Klasse `CEyeBotLoLevel` die gemeinsame Datenstruktur `ellServoTrait`. Sobald die Methode `setServo()` aufgerufen wird, manipuliert sie die Einträge dieser Datenstruktur. Die Belegung von `ellServoTrait` steuert wiederum den parallel zum Hauptprogramm ablaufenden `servoController()`. Da dieser im Interrupt abläuft, müssen Betriebssystem-Routinen des EYEBOT genutzt werden, um den Multitasking-betrieb während der Manipulation dieser kritischen Datenstruktur zu stoppen. Anderenfalls könnte die Servoansteuerung durch den gleichzeitigen Zugriff auf diese Datenstruktur in einen undefinierten Zustand geraten.

Das EYEBOT-Betriebssystem wie auch der externe MINI-SSC II erwarten, dass die anzufahrende Servoposition durch einen Wert von 0-255 angegeben wird. Dabei entspricht der Wert 0 der minimalen Servoposition und 255 analog dazu der maximalen Servoposition, die vom Motor angefahren werden kann. Der externe MINI-SSC II-Servocontroller stellt diesen anfahrbaren Bereich zwischen minimaler und maximaler Servoposition fix auf 180 Grad ein. In diesem Fall entspricht ein Ansteuerungswert von 255 einer Servostellung von 180 Grad. Beim EYEBOT können zudem die Bereiche, in denen ein Servomotor bewegt werden darf, in der sogenannten Hardware Description Table (HDT) festgelegt werden (s. a. [9] und [2]). Daher kann dieser Bereich für jeden Servo unterschiedlich ausfallen und demzufolge ein Ansteuerungswert von beispielsweise 60 bei jedem Servo in einen anderen Winkel umgesetzt werden. Diese hardwarenahe Steuerung wird von der Klasse `CEyeBotLoLevel` komplett vor dem Anwender verborgen. Stattdessen wird für Servomotoren die anzufahrende Position über den Aufruf der Methode `setServo()` in 0-360 Grad angegeben. Um diese anwenderfreundlichere und nachvollziehbarere Steuerung zu realisieren, mussten alle vom EYEBOT kontrollierten Servomotoren ebenfalls auf einen anfahrbaren Bereich von 180 Grad justiert werden. Durch die identische Bereichseinstellung bei allen Servomotoren führt daher ein Wert von beispielsweise 127 bei jedem Motor zu einer Position von 90 Grad. Auf diese Weise konnte in `setServo()` eine Ansteuerung realisiert werden, bei der die Servostellung als anzusteuern-der Winkel übergeben wird.

Die Klasse `CEyeBotLoLevel` stellt darüber hinaus einen Mechanismus zur Verfügung, mit dem die Bewegungsfreiheit eines Servomotors definiert und eingeschränkt werden kann. Zu diesem Zweck wurde die Methode `EnsureAngleWithinServoLimits()` implementiert. Ähnlich der Vorgehensweise bei der Definition eines Robotermodells in SIGEL werden in ZORCBRAIN für jeden Servomotor dessen Minimal- und Maximalauslenkungen in der Datenstruktur `tServoTrait` definiert³. Diese Werte werden beispielsweise bei der Ausführung eines MOVE-Befehls durch den GP-Interpreter eingesetzt, um die Einhaltung der Minimal- und Maximalauslenkungen der Servomotoren zu gewährleisten. Zunächst wird die Methode `EnsureAngleWithinServoLimits()` mit den Parametern Servo-ID und dem anzusteuern- den Winkel aufgerufen. Der übergebene Winkel wird von der Methode automatisch korrigiert, falls er außerhalb der in der `tServoTrait`-Struktur definierten Auslenkungsbegrenzungen liegt. Damit ist sichergestellt, dass sich die Bewegungsfreiheit von ZORC nicht von der seines Robotermodells in der SIGEL-Simulation unterscheidet.

Schließlich enthält die `tServoTrait`-Datenstruktur auch die initialen Servopositionen. Auf

³Die Initialisierungswerte für die Servomotoren werden im Konstruktor von `CEyeBotLoLevel` festgelegt.

5.3 Implementierung der Hardwareabstraktionsklasse *CEyeBotLoLevel*

diese Weise kann die Ausgangsposition aller Motoren spezifiziert werden, so dass diese identisch ist mit der Startposition des Robotermodells in der SIGEL-Simulation.

5.3.2 Abfrage der Servoposition

Die Methode `getSensorDeg()` dient dazu, die aktuelle Position eines Servomotors in Grad zu ermitteln. Die Nummer S des abzufragenden Servomotors wird als Parameter übergeben. Die internen Nummern der Servomotoren (auch als “Servo-ID” bezeichnet) werden in der Dokumentation der Klasse *CEyeBotLoLevel* in Anhang D aufgeführt.

Nach erfolgreichem Aufruf stellt `getSensorDeg()` zunächst den Relais-Router über den Digital-Ausgang ein. Dazu werden die gemäß Abbildung 2.5 beteiligten Ausgangsleitungen des Digitalports so geschaltet, dass die betroffene Einheit des Relais-Router das Signal des zu messenden Servomotors direkt mit einem A/D-Eingang verbindet (s. a. Kapitel 2.2).

Anschließend wird der aktuelle Messwert U_{Raw_S} abgefragt und in eine Gradposition umgerechnet. Hierfür werden die während der Initialisierungsprozedur ermittelten Parameter genutzt.

Die Initialisierungsprozedur wird bei jedem Start von ZORCBRAIN durchgeführt. Dazu wird zunächst jeder Servomotor auf eine Stellung von 0 Grad gebracht. Dort wird eine Messung vorgenommen und der Messwert U_{Ini_S} der Nullposition gespeichert. Anschließend wird der Servomotor um 10 Grad bewegt und eine weitere Messung durchgeführt. Somit ist für den Motor die Spannungsdifferenz U_{Δ_S} pro Grad bekannt und die Servoposition lässt sich als $(U_{Raw_S} - U_{Ini_S})/U_{\Delta_S}$ berechnen.

5.3.3 Abfrage des Fußkontakts

Die Methode `getSensorValue()` kann zur Ermittlung des Status’ der Fußkontaktsensoren mit den Parametern `kFootSensL` bzw. `kFootSensR` aufgerufen werden (s. a. Konstanten-deklaration in der Klasse *CEyeBotLoLevel* in Anhang D). Bei geschlossenen Kontakten – d. h. aufgesetztem Fuß – wird ein Wert von 1, anderenfalls 0 zurückgeliefert.

5.3.4 Abfrage der Neigung

Die 4 Neigungssensoren von ZORC werden abgefragt durch die Methoden `getPitchDeg()` und `getRollDeg()`. Zu diesem Zweck werden die gegenüberliegenden Sensorpaare zur Pitch- und Roll-Achse zusammengefasst. Diese Achsen wurden bereits in Kapitel 3.3.2 erläutert und in Abbildung 3.3 auf S. 21 anhand des ZORC-Robotermodells illustriert.

Beide Methoden geben je nach Status der jeweiligen beiden Neigungsschalter einen der folgenden Werte zurück:

5 Die Programmierung von ZORC



Abbildung 5.2: Das Foto zeigt den EYEBOT-Controller, der das Programm ZORCBRAIN ausführt. Die im LCD sichtbaren Menüpunkte können über die darunter gelegenen Menütaster ausgewählt werden.

- 45 : linker (Roll) bzw. hinterer (Pitch) Neigungsschalter aktiv
- 0 : kein Schalter aktiv
- 45 : rechter (Roll) bzw. vorderer (Pitch) Neigungsschalter aktiv

5.3.5 Abfrage der Stromaufnahme

Die A/D-Wandler des EYEBOT arbeiten mit einer Auflösung von 10 Bit. Damit stehen $2^{10} = 1024$ Stufen zur Verfügung, um die anliegende Spannung zu digitalisieren. Da die Wandler in einem Intervall von 0..5V arbeiten, entspricht das einer Auflösung von ca. 205 Stufen pro 1V-Schritt. Der MAX472-IC gibt in der eingesetzten Konfiguration die gemessene Stromstärke in einem Verhältnis von 0.2V pro gemessenem Ampere aus. Damit ergibt sich für die am EYEBOT gemessene Spannung ein Verhältnis von 41 Stufen pro gemessenem Ampere. Die aktuelle Stromaufnahme berechnet sich somit unter Verwendung der A/D-Messung an Kanal AD_6 des EYEBOT als $\frac{AD_6}{41}$ Ampere.

Die Stromaufnahme wird über die Methode `getCurrentDraw()` ermittelt, die den aktuellen Wert in 1/10 Ampere zurückliefert. Ein Rückgabewert dieser Methode von z. B. 50 entspricht daher 0.5 Ampere.

5.3.6 Adressierung der Sensoren und Motoren

Das ZORCBRAIN-Programm nutzt die in Anhang D aufgeführten Konstanten zur Adressierung der Servomotoren und Sensoren. Die Adresse eines Sensors ist identisch mit dem ihm zugeordneten Servomotor, jedenfalls soweit es sich um Sensoren handelt, die zur Messung des Servowinkels eingesetzt werden. So wird z. B. die Konstante `kRKnee` an die Methode `getSensorDeg()` übergeben, um den Winkel des rechten Knies zu ermitteln, oder an die Methode `setServo()`, um den rechten Knie-Servo zu bewegen.

In SIGEL GP-Programmen werden Sensoren und Motoren ebenfalls über Adressen angesprochen. Leider lassen sich die Adressen der Sensoren und Motoren bei der Definition eines Robotermodells nicht beeinflussen, sondern werden durch interne Faktoren beim Einlesen des Robotermodells auf unvorhersehbare Weise festgelegt⁴. Da die Adressierung für das Robotermodell von ZORC nicht beeinflusst werden konnte, wurden die Adressen des fertigen Robotermodells im ZORCBRAIN-Programm umgerechnet. Zwei Methoden definieren diese Permutationen: `S2Z_Sensor()` setzt eine Sensoradresse von SIGEL in die Adresse des entsprechenden Sensors auf ZORC um, analog dazu setzt `S2Z_Servo()` die Adressen von Servomotoren um. Beide Methoden werden bei der Interpretation der `MOVE` und `SENSE` Instruktionen zur Umsetzung von Adressen aufgerufen. Damit ist sichergestellt, dass GP-Programme in der SIGEL-Simulation und auf der realen Roboterarchitektur die gleichen Motoren und Sensoren adressieren.

5.3.7 Die Klasse `ZORC_RS232Stream`

Um das Lesen und Schreiben von Daten von der RS232-Schnittstelle zu vereinfachen, wurde die `ZORC_RS232Stream`-Klasse implementiert. Sie bietet spezialisierte Methoden für das Lesen unterschiedlicher Datentypen und bietet einen Zugriff auf die RS232-Schnittstelle, die dem Zugriff auf Streams ähnelt (s. Kap. 21 in [23]). Wird beispielsweise im RS232-Datenstrom als nächstes Datum eine Gleitkommazahl vom Typ `float` erwartet, so kann diese mit der Methode `getFloat()` der Klasse `ZORC_RS232Stream` gelesen werden. Bei der direkten Verwendung von `CEyeBotLoLevel` hätten zunächst die Zeichen vom RS232-Datenstrom bis zu einem geeigneten Trennzeichen eingelesen werden müssen. Diese Textfolge hätte anschließend als Gleitkommazahl interpretiert und schließlich in eine geeignete Variable konvertiert werden müssen.

Die Methode `Build_RS232Stream_Obj()` der Klasse `CEyeBotLoLevel` dient der Erzeugung eines Objekts vom Typ `ZORC_RS232Stream`. Die eingesetzten Portierungen der SIGEL-Klassen nutzen `ZORC_RS232Stream` intensiv zum Datentransfer mit der seriellen Schnittstelle des EYEBOT.

⁴Die Sensor- und Motoradressen ändern sich nicht mehr, sobald die Struktur des Roboters durch die Roboterdefinitionsdatei festgelegt wurde. Auch beim Export und anschließenden Import der Roboter – als Dateien mit der Endung “.crb” – bleibt diese Nummerierung konstant.

5.4 Die Klasse CZorcBrain

Das ZORCBRAIN-Programm wird durch die zentrale Klasse CZorcBrain gesteuert. Nach dem Start des Programms auf dem EYEBOT-Mikrocontrollerboard werden zunächst diverse Initialisierungen vorgenommen. Dazu zählen das Einstellen der Servoparameter, Installieren des Servocontroller-Interrupts, Initialisieren der seriellen Schnittstellen und die Prozedur zur Sensorkalibrierung. Anschließend geht die Kontrolle des Programmablaufs an eine Instanz von CZorcBrain über. Mittels der in Kapitel 5.4.3 beschriebenen Menüsteuerung können nun

- Programme und Sprachparameter geladen werden
- Diverse Testfunktionen aufgerufen werden
- Programme auf ZORC ausgeführt werden

Das Laden von Programmen über die RS232-Schnittstelle, an die z. B. ein Desktop-Rechner angeschlossen werden kann, bedient sich der bereits besprochenen Klassen und wird nicht näher erläutert.

Die folgenden Kapitel gehen kurz auf die zur Verfügung stehenden Testfunktionen und das Ausführen von SIGEL GP-Programmen auf ZORC ein. Zusätzlich wird in Kapitel 5.4.3 die zum Einsatz kommende Menüsteuerung beschrieben.

5.4.1 Die Testfunktionen von CZorcBrain

Unter dem Menüpunkt “Opt” stehen verschiedene Optionen zum Test der Roboterhardware und Programmfunktion zur Verfügung. Neben einer Anzeige der aktuellen Sensormesswerte auf dem LC-Display des EYEBOT können diese in Form eines SIGEL-Programms über die serielle Schnittstelle an den daran angeschlossenen Rechner gesendet werden. Dieses Programm kann anschließend wahlweise in SIGEL importiert und dort visualisiert werden oder an ZORC zurückgesendet werden, um die aktuelle Stellung der Servomotoren zu reproduzieren. Der Inhalt des aktuell in einer CZorcBrain-Instanz gespeicherten Programms kann durch Aufruf der “Check Program” Funktion über die serielle Schnittstelle ausgegeben werden.

Die Funktion “RS232 Ping” wird von SIGEL benötigt, um die Verbindung zum Roboter zu testen. Nach dem Aufruf der Funktion wird der Empfang einer Ziffer n über die serielle Schnittstelle abgewartet. Wurde eine solche Ziffer empfangen, wird der Wert $2n$ als Antwort zurückgesandt.

5.4.2 Ausführung von SIGEL-Programmen

Ein auf ZORC übertragenes GP-Programm kann über den Menüpunkt “Run” ausgeführt werden (s. Abb. 5.2 und 5.3). Die Übertragung der Programme läuft bei der Evolution von Laufalgorithmen auf der realen Roboterarchitektur automatisch ab (s. a. Kapitel 6.4.2). Um daher

die Ausführung des GP-Programms auf ZORC zu einem definierten Zeitpunkt zu starten, wird der Interpreter erst aufgerufen, sobald die Fußtaster einen Bodenkontakt feststellen. Die Programmausführung kann demzufolge erst beginnen, wenn die Person, welche das Experiment durchführt, ZORC auf die Experimentierfläche stellt. Zuvor werden die Servomotoren in die Ausgangsstellung bewegt, die wie beschrieben in der `ellServoTrait`-Datenstruktur festgelegt ist. Die daran anschließende Ausführung von GP-Programmen durch eine Instanz der Klasse `ZORC_Interpreter` wurde in Kapitel 5.2 beschrieben.

In der Simulation von Robotersteuerungsprogrammen durch SIGEL wird die Länge der zu simulierenden Zeitspanne im zugehörigen Experiment festgelegt. Eine ähnliche Funktionalität wurde in ZORCBRAIN implementiert. Während `CZorcBrain` auf den Kontakt der Fußtaster wartet, kann eine Ziffernfolge an ZORC übertragen werden. Diese wird als Zeitangabe in Sekunden interpretiert, die der Simulationszeit in SIGEL entspricht. Nach Ablauf dieser Zeitspanne wird die Programmausführung im Interpreter gestoppt. Aus Sicht der Fernsteuerung über die serielle Schnittstelle (s. a. Kapitel 5.4.3) kann z. B. eine Laufzeit von 10 Sekunden festgelegt werden, indem nach dem Aufruf des Menüpunktes "Run" die Ziffer "10" gesendet wird, d. h. durch die Übertragung der Ziffernfolge: 2 - 1 - 0.

Ein Schutzmechanismus des ZORCBRAIN-Programms überwacht die Pausen zwischen zwei aufeinander folgenden Aufrufen des Interpreters. Um die Servomotoren vor Überhitzung und damit irreparablen Defekten zu schützen, wird eine Pause von 20 Sekunden⁵ erzwungen. Diese Limitierung bezieht sich jedoch nicht auf die Länge einer Programmausführung durch den Interpreter; diese wird nicht beschränkt. Falls der Interpreter vor Ablauf der Frist erneut aufgerufen wird, pausiert ZORCBRAIN bis die 20 Sekunden verstrichen sind.

5.4.3 Die Menüsteuerung

Das ZORCBRAIN-Programm bietet zahlreiche Funktionen, die dem Anwender für eine effektive Nutzung über eine Menüsteuerung zugänglich gemacht wurden. Die minimale Benutzerschnittstelle von ZORC – bzw. dem EYEBOT – besteht aus den in Abbildung 5.2 gezeigten Tastern. Die vier Bedienelemente sind direkt unterhalb des LC-Displays angebracht und können durch einen kurzen Text im Display beschriftet werden.

Mit dieser Benutzerschnittstelle wurde eine hierarchische Menüstruktur entworfen, mit der das ZORCBRAIN-Programm gesteuert werden kann. Auf oberster Ebene kann zwischen den drei Teilbereiche des Programms gewählt werden, d. h.

1. Laden von GP-Programmen und -Sprachparametern
2. Ausführen des geladenen GP-Programms
3. Test- und Hilfsfunktionen

⁵Die Länge der einzuhaltenden Pause zwischen zwei Interpreteraufrufen wird durch die Belegung der Variable `zbESPWaitSecs` im Konstruktor der Klasse `CZorcBrain` definiert.

5 Die Programmierung von ZORC

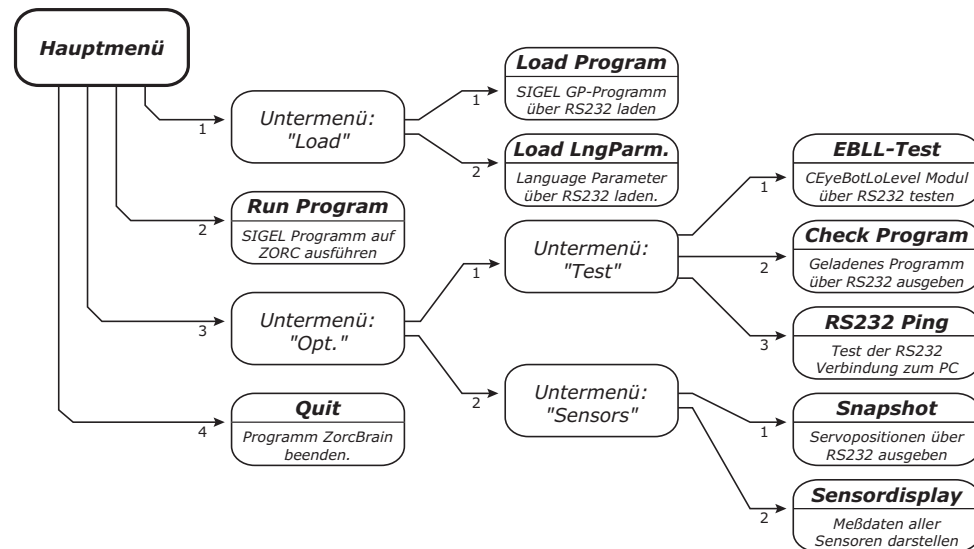


Abbildung 5.3: Das Diagramm stellt die ZORCBRAIN-Menüstruktur dar. Die jedem Menüpunkt vorangestellten Zahlen geben an, welche Taste bzw. welche über die serielle Schnittstelle empfangene Ziffer der Anwahl des entsprechenden Menüpunktes dient.

Die detaillierte Menüstruktur ist im Diagramm in Abbildung 5.3 wiedergegeben. Auf jeder Menüebene bietet Taste 4 die Möglichkeit, auf die nächsthöhere Menüebene zurückzukehren.

In Kapitel 5.1 wurde bereits erwähnt, dass ZORC für eine effiziente und benutzerfreundliche Durchführung der Experimente optional von einem externen Rechner ferngesteuert werden soll. Die Anbindung an diesen Rechner erfolgt über die externe RS232-Schnittstelle des EYEBOT (s. a. 6.4.2). Aus diesem Grund wurde eine optionale Steuerung des ZORCBRAIN-Menüs über die RS232-Schnittstelle realisiert. Ein Menüpunkt kann auf zwei verschiedene Weisen gewählt werden:

1. Wahl eines Tasters am EYEBOT
2. Senden einer Ziffer über RS232 an den EYEBOT

Das zuerst eintreffende Signal – “Taste gedrückt” oder “RS232 Datum” – bestimmt die Wahl des Menüpunktes. Die RS232-Steuerung setzt dabei eine einfache Kodierung ein, bei der die Tasten 1-4 durch die Ziffern “1” bis “4” repräsentiert werden. ZORC kann auf diese Weise auch manuell mittels eines Terminalprogramms ferngesteuert werden. Um über die RS232-Menüsteuerung beispielsweise das “Snapshot” aller Servopositionen von ZORC in Form eines SIGEL-Programmsegmentes zu veranlassen, muss gemäß Abbildung 5.1 die Ziffernfolge 3 - 2 - 1 übertragen werden.

6 Übertragung und Evolution von Laufalgorithmen auf ZORC

In diesem Kapitel wird die Übertragung von Laufalgorithmen, die mittels Genetischer Programmierung in der SIGEL-Simulationsumgebung “gelernt” wurden, auf die reale Roboterarchitektur geschildert. Neben der Übertragbarkeit der Ergebnisse aus der Simulation auf ZORC wird auf die fortgeführte Evolution dieser Laufalgorithmen auf dem realen Roboter und die auf diese Weise erzielten Ergebnisse eingegangen.

6.1 Allgemeine Schwierigkeiten bei der Übertragung

Das auf dem EYEBOT ablaufende Programm ZORCBRAIN, das eingehend in Kapitel 5 beschrieben wird, ermöglicht ZORC die Ausführung von SIGEL GP-Programmen. Hierdurch wird der reale Roboter von den Individuen gesteuert. Die Ausführung der GP-Programme in der Simulation und auf dem realen Roboter wird von den – im Wesentlichen – identischen Klassen `SIGEL_Interpreter` und `ZORC_Interpreter` übernommen. Dennoch kann es dabei zu unterschiedlichen Ergebnissen kommen, betrachtet man die resultierenden Bewegungsmuster in der SIGEL-Simulation und auf ZORC. Die Ursachen dafür liegen vor allem in

1. Unterschiedlichen Sprachparametern
2. “Kritischen” SENSE-Instruktionen
3. Abweichungen zwischen ZORC und dessen Robotermodell

Die Ausführung von Programmen durch den Interpreter wird wesentlich von den verwendeten Sprachparametern beeinflusst. Zu diesen zählt die simulierte Dauer der GP-Instruktionen (Command Duration), die aus den in Kapitel 5.2 erläuterten Gründen auf ZORC eine Dauer von 0.01 Sekunden nicht überschreiten darf. Haben die Programme während der Evolution in der SIGEL-Simulation eine kürzere Instruktionsdauer genutzt, dann resultieren diese Programme auf ZORC mit hoher Wahrscheinlichkeit in unterschiedlichen Bewegungsmustern, wie das Beispiel des ersten “BigDistance”-Experiments in Kapitel 4.2.2 gezeigt hat. Eine Übertragbarkeit der evolvierten Laufalgorithmen ist demzufolge nur mit identischen Instruktionslängen garantiert.

6 Übertragung und Evolution von Laufalgorithmen auf ZORC

Einen maßgeblichen Einfluss auf die Programmausführung besitzen außerdem die in den Sprachparametern definierten Eigenschaften der virtuellen CPU. Zu diesen zählen die Anzahl der Register und die Registerbreite. Die CPU wird durch den Interpreter simuliert, wenn dieser GP-Programme ausführt. Bei der Interpretation einer GP-Instruktion haben Registeranzahl und -breite einen wesentlichen Einfluss auf das Verhalten des Programms. Beispielsweise lässt sich betrachten, wie die Instruktion `“MOVE 10”` interpretiert wird. Gemäß Anhang B bezeichnet der Parameter `“10”` die Adresse eines Registers. Dessen Inhalt wird als Nummer des zu bewegenden Servomotors interpretiert. Als Schutzmaßnahme vor Überläufen wird allerdings der Inhalt des Registers mit der Adresse `“10 % Registeranzahl”` betrachtet. Das `“%”` Zeichen steht für die Operation `“Rest der Division”` bzw. die modulo-Division, wie beispielsweise auch in der Programmiersprache C (s. Kapitel 2.3.1 in [23]). Offensichtlich kommt man bei unterschiedlicher Anzahl der Register zu unterschiedlichen Ergebnissen, z. B. resultiert eine Registeranzahl von 8 in der Adresse 2 und eine Registeranzahl von 16 in der Adresse 10. Auf ähnliche Weise hat die Registerbreite einen Einfluss auf die letztendlich resultierende Motoradresse. Zusammenfassend kann man feststellen, dass nur mit identischen Sprachparametern in der Simulation und auf ZORC das (weitgehend) identische Verhalten eines Laufalgorithmus’ in beiden Umgebungen gewährleistet ist.

Auch wenn die Sprachparameter in der Simulation und auf dem realen Roboter übereinstimmen, können sich GP-Programme, die in der Simulationsumgebung erfolgreich waren, auf ZORC vollkommen unterschiedlich verhalten. Die Ursache dafür liegt vor allem in der Natur einer Simulation. Die physikalische Simulation des ZORC-Robotermodells durch die DYNAMECHS-Bibliothek ist `“zu perfekt”` verglichen mit dem realen Roboter. Bei jeder Simulation des Robotermodells weicht dessen Ausgangsposition in der simulierten Umgebung nicht einmal um 1/1000 Millimeter von der Ausgangsposition tausender weiterer Simulationsläufe ab. Jedes GP-Programm findet das Robotermodell in absolut identischer Lage vor. Eine identische Instruktionssequenz überführt das Robotermodell in die identische Konfiguration, d. h. alle Gelenke besitzen anschließend den exakt gleichen Winkel. Wird nun der Winkel eines solchen Gelenkes mittels einer `SENSE`-Instruktion abgefragt (s. a. Anhang B), erhält diese Instruktion folglich bei jeder Ausführung des Programms den gleichen Messwert. Falls dieser Messwert im weiteren Programmverlauf zu Vergleichen, z. B. der Form `“Wenn linkes-Knie > 90°”`, herangezogen wird, ist diese Instruktion für den Programmablauf wenig problematisch. Auch wenn das Ergebnis der Abfrage im Folgenden ignoriert wird, d. h. das Register, in dem der abgefragte Winkel gespeichert wurde, nicht ausgelesen wird, beeinflusst diese Instruktion den Programmablauf nicht. Eine `SENSE`-Anweisung kann allerdings zu einer `“kritischen Instruktion”` werden, falls das Ergebnis der Abfrage zur Adressierung eingesetzt wird. Betrachtet man z. B. die folgende Instruktionssequenz, wird die Problematik deutlich:

1. Eine `SENSE`-Anweisung schreibt den gemessenen Wert in Register n
2. Eine `MOVE`-Anweisung interpretiert den Registerinhalt von n als Adresse eines zu bewegenden Motors

Diese Kombination von Anweisungen wird auf der realen Roboterarchitektur mit einer hohen Wahrscheinlichkeit zu einem vollkommen anderen Ergebnis führen als in der Simulation.

6.2 Übertragbarkeit von SIGEL-Programmen auf ZORC

Denn es ist höchst unwahrscheinlich, dass der gemessene Winkel auf ZORC mit dem in der Simulation gemessenen Winkel exakt übereinstimmt. Es ist zudem unwahrscheinlich, dass selbst auf ZORC mehrmals hintereinander der exakt gleiche Winkel gemessen wird, da auf Grund der – konstruktions- und materialbedingten – geringen Präzision von ZORC eine solche Genauigkeit kaum zu erzielen ist. Eine solche Verwendungsart der SENSE-Instruktion führt daher zu Programmen, die unter realistischen Bedingungen nicht robust funktionieren. Eine Abweichung im gemessenen Winkel von $\pm 1^\circ$ führt im Beispiel dazu, dass ein anderer Motor bewegt wird. Verglichen mit dem Gebrauch der Sensorabfrage in Form eines Größer-Als-Vergleiches gibt es bei der Verwendung in Form einer Adressierung keinen “Spielraum” für den Wert, den der Sensor zurückliefert.

Eine Möglichkeit, diesen Effekt mit Sicherheit zu vermeiden, bestünde in der Sperrung der SENSE-Instruktion. Damit wäre es dem GP-System untersagt, diesen Befehl in GP-Programmen zu verwenden. Beispielsweise wurde bei dem (handgeschriebenen) Programm aus Kapitel 6.2 auf diese Instruktion verzichtet. Nur auf diese Weise konnte sichergestellt werden, dass die Ergebnisse der Simulation uneingeschränkt übertragbar waren. Der Nachteil dieser Vorgehensweise liegt auf der Hand: Durch den Verzicht auf Sensorabfragen wäre der alte Zustand der “blinden Steuerung” mit allen in Kapitel 2.2 erwähnten Nachteilen wieder hergestellt. Die aufwändige Erweiterung von ZORC um die beschriebene Sensorik wäre zudem nutzlos gewesen. In Kapitel 7 wird im Zusammenhang mit dieser Problematik eine Erweiterung des SIGEL-Systems vorgeschlagen. Diese könnte das System möglicherweise dazu veranlassen, Programme zu evolvieren, die sich hinsichtlich der SENSE-Problematik robuster verhalten.

Bei der im Rahmen der Arbeit eingesetzten Form des SIGEL-Systems führte diese Problematik dazu, dass die Übertragung von Simulationsergebnissen auf die reale Roboterarchitektur in gewisser Weise einem Glücksspiel ähnelte. Bei allen Experimente wurde die SENSE-Instruktion uneingeschränkt zugelassen. Die Programme, deren Bewegungsablauf in der Simulationsumgebung sowie auf ZORC im Detail verglichen wurde, fielen jedoch nicht durch deutlich abweichende Bewegungsmuster auf.

Die Unterschiede zwischen ZORC und seinem Robotermodell bilden letztendlich den dritten Faktor, der einen Einfluss auf die Übertragbarkeit der Simulationsergebnisse auf ZORC besitzt. Bei der Modellierung von ZORC wurden die in Kapitel 3.2 beschriebenen Vereinfachungen vorgenommen. Die bei der Modellierung und durch die Vereinfachungen entstandenen Differenzen zwischen ZORC und seinem Modell können bei der Ausführung der übertragenen Programme auf dem realen Roboter zu einem abweichenden Verhalten führen. Der folgende Abschnitt belegt anhand eines einfachen SIGEL-Programms, dass zumindest die Übertragbarkeit von simplen Programmen aus der Simulationsumgebung auf die reale Roboterarchitektur gewährleistet ist.

6.2 Übertragbarkeit von SIGEL-Programmen auf ZORC

Das in Abbildung 6.1 im Quellcode wiedergegebene SIGEL GP-Programm wurde handgeschrieben und zunächst in der Simulationsumgebung am ZORC-Robotermodell getestet. In Anhang B werden die zur Verfügung stehenden Instruktionen der SIGEL-Individuen eingehend erläutert. Durch das gezeigte Programm werden alle 13 Motoren des Roboters aktiviert und eine statische Position angefahren, d. h. ZORC verharrt in der gezeigten Position.

In ein SIGEL-Experiment importiert bewegt das Programm das Robotermodell von ZORC in eine aufrecht stehende Position. In Abbildung 6.2 ist die resultierende Endposition des Robotermodells in der Simulationsumgebung dargestellt. Das gleiche Programm wurde anschließend auf den realen ZORC übertragen und von dem in Kapitel 5 beschriebenen ZORCBRAIN-Programm ausgeführt. Abbildung 6.3 illustriert, dass es auf dem Roboter in einer identischen Endposition resultiert.

Dieses Beispiel belegt, dass Programme, die in der Simulationsumgebung von SIGEL das Robotermodell von ZORC kontrollieren, auf den realen Roboter übertragbar sind. Dort können die Programme das gleiche Verhalten bewirken, wobei die Übertragbarkeit der Programme durch die im vorangegangenen Abschnitt diskutierten Faktoren eingeschränkt werden kann. Identische Sprachparameter und der Verzicht auf die Instruktion SENSE garantieren bei dem gezeigten Beispiel das identische Verhalten in der Simulationsumgebung und auf ZORC. Das gezeigte Programm fährt lediglich eine statische Position an und kann daher nicht als Lau-

1	LOAD 1, -128	22	LOAD 1, -121
2	LOAD 0, 20	23	LOAD 0, -15
3	MOVE 1	24	MOVE 1
4	LOAD 1, -127	25	LOAD 1, -120
5	LOAD 0, -15	26	LOAD 0, -15
6	MOVE 1	27	MOVE 1
7	LOAD 1, -126	28	LOAD 1, -119
8	LOAD 0, -15	29	LOAD 0, -45
9	MOVE 1	30	MOVE 1
10	LOAD 1, -125	31	LOAD 1, -118
11	LOAD 0, -20	32	LOAD 0, 45
12	MOVE 1	33	MOVE 1
13	LOAD 1, -124	34	LOAD 1, -117
14	LOAD 0, -20	35	LOAD 0, -5
15	MOVE 1	36	MOVE 1
16	LOAD 1, -123	37	LOAD 1, -116
17	LOAD 0, 0	38	LOAD 0, -5
18	MOVE 1	39	MOVE 1
19	LOAD 1, -122		
20	LOAD 0, 0		
21	MOVE 1		

Abbildung 6.1: Der Quellcode eines handgeschriebenen SIGEL-Programms, das ZORC in stehender Position hält. Das Programm benötigt Sprachparameter mit 24 Registern und einer Registerbreite von 8 Bit.

6.3 Übertragung der Simulationsergebnisse

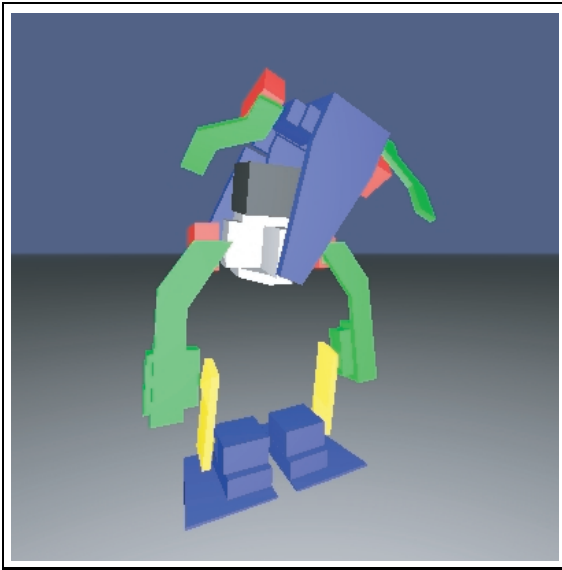


Abbildung 6.2: Das Programm aus Abb. 6.1 ausgeführt in der SIGEL-Simulation.

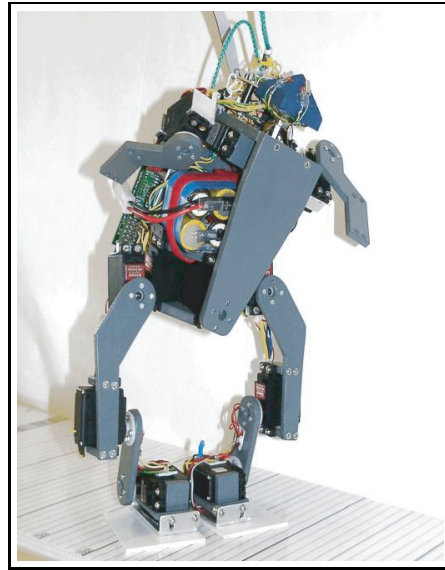


Abbildung 6.3: ZORC bei der Ausführung des gleichen Programms.

falgorithmus im engeren Sinn bezeichnet werden. Die erfolgreiche Übertragung dieses Programms deutet jedoch darauf hin, dass die reale Roboterarchitektur – bis zu einem gewissen Grad – adäquat vom ZORC-Robotermodell nachgebildet wird.

6.3 Übertragung der Simulationsergebnisse

Die Motivation für die in Kapitel 4 beschriebene Evolution von Laufalgorithmen in der Simulationsumgebung lag in der anschließenden Übertragung erfolgreicher Individuen auf ZORC. Im Rahmen des Kapitels wurde bereits erläutert, dass sich lediglich die Ergebnisse des Experimentes mit der “BigDistance-2”-Fitnessfunktion durch zumindest bedingt erfolgreiche und zugleich auf ZORC übertragbare Ergebnisse auszeichneten.

Zunächst wurden daher die Individuen dieses Experiments auf die reale Roboterarchitektur übertragen. Hierzu wurden alle 50 Individuen auf ZORC ausgeführt und deren Bewegungsabläufe begutachtet. Wie jedoch nach den Ergebnissen aus der Simulationsumgebung nicht anders zu erwarten war, ließen die auf dem realen Roboter ausgeführten Programme keine sinnvollen Bewegungsabläufe erkennen. Analog zu dem in Abbildung 4.8 dargestellten Bewegungsablauf resultierte das beste Individuum der Population auch auf ZORC in einer ähnlichen Bewegung, die den Roboter nach hinten fallen liess.

Ausgehend von dieser vor-evolierten Population wurde daher der Lernprozess auf der realen Roboterarchitektur fortgesetzt. Diese Weiterführung der künstlichen Evolution ist Gegenstand von Kapitel 6.4.6.

6.4 Die fortgeführte Evolution auf ZORC

Lassen sich Laufalgorithmen, die in der Simulationsumgebung erfolgreich gelernt werden konnten, auf die reale Roboterarchitektur übertragen und verhalten sich die übertragenen Programme auf dem realen Roboter zudem genau – oder zumindest sehr ähnlich – wie in der Simulation, dann wurde die Zielsetzung erreicht. Anderenfalls kann versucht werden, den Lernprozess auf der realen Roboterarchitektur fortzusetzen, um auf diese Weise zum gewünschten Ergebnis zu gelangen.

6.4.1 Motivation eines fortgeführten Lernprozesses

Die in Kapitel 6.1 erläuterten Faktoren können dazu führen, dass die Laufalgorithmen sich auf dem realen Roboter anders verhalten als in der Simulationsumgebung. Von den dort angesprochenen, potentiellen Problemquellen stellen die Sprachparameter kein grundlegendes Problem dar, solange die erwähnten Einschränkungen in der Instruktionsdauer beachtet werden. Die in der Simulationsumgebung verwendeten Sprachparameter, mit denen ein Programm evolviert wurde, können zusammen mit dem Individuum auf ZORC übertragen werden.

Problematischer ist jedoch ein unterschiedliches Verhalten des realen Roboters, das durch Abweichungen im Robotermodell oder durch den Einsatz kritischer SENSE-Instruktionen bedingt ist. In diesem Fall müssen die Individuen selbst modifiziert werden. Eine manuelle Anpassung evolvierter Programme ist möglich, da exportierte SIGEL-Individuen in Textform vorliegen. In dieser Form (vgl. Abbildung 6.3) sind die Programme leicht editierbar. Die störenden Instruktionen könnten daher z. B. lokalisiert und aus den Programmen entfernt werden. Eine solche Vorgehensweise wäre jedoch umständlich und zeitaufwändig. Es bietet sich daher an, diese Anpassungen mittels Genetischer Programmierung automatisiert durchführen zu lassen.

Neben einer solchen Weiterführung der Evolution zur Anpassung von Individuen, die bereits erfolgreich einen Laufalgorithmus implementieren, kann ein fortgesetzter Lernprozess der allgemeinen Optimierung von Individuen dienen. Die nur zum Teil erfolgreichen Ergebnisse der Experimente, deren Resultate auf ZORC übertragbar waren, haben im Rahmen dieser Arbeit eine Optimierung der Individuen auf ZORC motiviert. Hierbei traten die eigentlichen Laufalgorithmen erst auf der realen Roboterarchitektur auf. Dieser Erfolg wurde jedoch sehr wahrscheinlich nur durch die bereits geleistete “Vorarbeit”, d. h. in der Simulationsumgebung vor-evolvierte Individuen, ermöglicht.

Bei beiden Ausgangssituationen wird jedoch die identische Zielsetzung verfolgt: Eine Weiterentwicklung der bereits vor-evolvierten Individuen auf dem realen Roboter im Rahmen eines Lernprozesses. Durch die Bewertung des Verhaltens dieser Programme auf ZORC wird die Absicht verfolgt, einen robusteren Bewegungsablauf auf dem realen Roboter zu erreichen. Die nachfolgenden Kapitel beschreiben im Detail, wie sich dieser fortgeführte Lernprozess auf ZORC gestaltet.

6.4.2 Steuerung von ZORC durch SIGEL

Zur Steuerung des Lernprozesses auf der realen ZORC-Architektur kommt erneut das SIGEL-System zum Einsatz. Auf Grund des modularen Aufbaus von SIGEL ist die Fitness-evaluierung nicht zwangsläufig an die Simulationsumgebung gebunden. Das GP-Subsystem erwartet lediglich, dass die Fitnessfunktion eine Bewertung der Individuen vornimmt.

Im “klassischen” SIGEL-System steuern die Individuen den Roboter in der Simulationsumgebung und werden anhand des Verhaltens des simulierten Roboters bewertet. Für den Lernprozess auf der realen Roboterarchitektur werden die Individuen über eine serielle Verbindung auf ZORC heruntergeladen und dort ausgeführt. Um die Fitness dieser auf den Roboter übertragenen Individuen zu berechnen, wird analog zur Simulation das Verhalten des realen ZORC bewertet. Im folgenden Abschnitt wird die Fitnessfunktion beschrieben, die – unterstützt durch das ZORCBRAIN-Programm auf dem Roboter – eine Fitness-evaluierung von SIGEL-Individuen auf ZORC ermöglicht.

6.4.3 Die Fitnessfunktion “RemoteZorc”

Der fortgeführte Lernprozess auf ZORC sollte mit einem möglichst unveränderten SIGEL-Programm durchgeführt werden. Ebenso wäre eine Spezialversion des Programms möglich gewesen, die einzig für die Steuerung der Evolution auf dem realen Roboter geeignet gewesen wäre. Stattdessen wurde jedoch diese Funktionalität in Form einer neuen Fitnessfunktion in SIGEL integriert. Damit steht sie jedem Anwender zur Verfügung und die Evolution von Laufalgorithmen auf ZORC kann ggf. von einer Weiterentwicklung des Programms profitieren. Diese neue Fitnessfunktion trägt in der SIGEL-Applikation die Bezeichnung “RemoteZorc”. In Abbildung 6.4 ist die Applikation mit den Parametereinstellungen für das GP-Subsystem dargestellt, in denen die Fitnessfunktion ausgewählt werden kann.

Die Fitnessfunktion “RemoteZorc” dient zum einen der Übertragung von Individuen, die evaluiert werden sollen, an den Roboter. Außerdem übermittelt sie die resultierenden Fitnesswerte an SIGEL. Die Funktionsweise dieser Fitnessfunktion wird im Folgenden anhand einer exemplarischen Fitnessauswertung beschrieben.

Wenn die Fitness eines Individuums durch dessen Ausführung auf ZORC evaluiert werden soll, testet die Fitnessfunktion zunächst die Verbindung zum Roboter. Als Gerätedatei für die Kommunikation mit dem Roboter wird “/dev/modem” verwendet¹. Konnte die serielle Schnittstelle des Rechners, der die SIGEL-Applikation ausführt, erfolgreich geöffnet werden, wird der in Kapitel 5.4.1 beschriebene RS232-Ping von ZORCBRAIN aufgerufen. Die Funktion wird gemäß Abbildung 5.3 durch die Fernsteuerung von ZORCBRAIN über die serielle Schnittstelle aufgerufen. Anschließend wird eine einstellige Zahl übertragen und die Antwort von ZORC empfangen. Bei einer korrekten Antwort ist die Verbindung zu ZORC sichergestellt und die Evaluierung wird fortgesetzt, anderenfalls wird eine Fehlermeldung ausgegeben.

¹Diese Gerätedatei ist unter dem Linux-Betriebssystem üblicherweise ein symbolischer Link. Durch Anpassen des Verweises auf eine reale Gerätedatei (z. B. “/dev/ttyS0”) kann die serielle Schnittstelle angegeben werden, an der ZORC angeschlossen ist.

6 Übertragung und Evolution von Laufalgorithmen auf ZORC

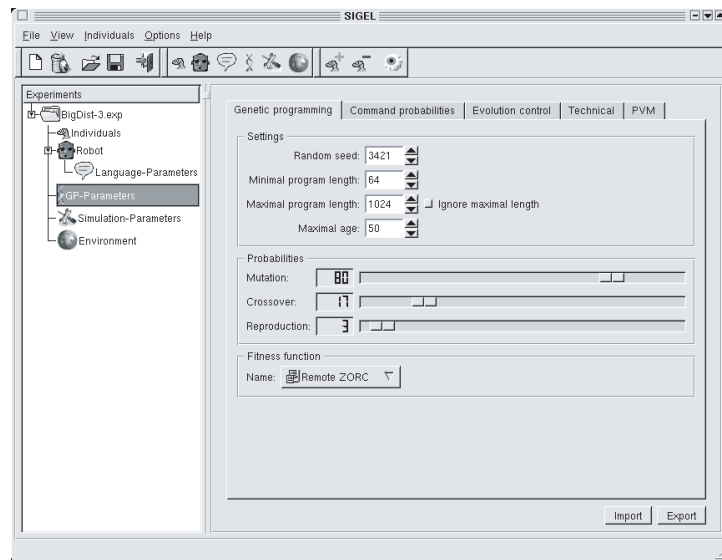


Abbildung 6.4: Bildschirmfoto von SIGEL mit den das GP-System betreffenden Parametereinstellungen und der angewählten “RemoteZorc”-Fitnessfunktion.

Im Anschluss daran wird das zu evaluierende Programm an den Roboter übertragen. Wenn die Fitnessfunktion über die “Visualize”-Funktion der SIGEL-Applikation gestartet wurde, ist dies das vom Anwender in der Individuen-Liste ausgewählte Individuum. Im Zuge eines Turniers handelt es sich hierbei ansonsten um eines der durch die Genetischen Operatoren neu erzeugten Individuen. In jedem Fall wird dieses Programm durch den Aufruf der Methode `printToString()` der Klasse `SIG.Program` in einen Text umgewandelt. Mittels der Fernsteuerung von ZORCBRAIN wird der Programmpunkt zum Download eines Individuums angewählt und ZORC damit in Empfangsbereitschaft versetzt. Danach wird das Programm in Textform zum Roboter übertragen. Der Empfang des Programms wird von ZORCBRAIN mit der Übermittlung der Anzahl empfangener Programmzeilen (bzw. Instruktionen) quittiert. Diese wird von SIGEL mit der Anzahl der Instruktionen in der SIGEL-internen Kopie des Programms verglichen. Durch diesen simplen Test kann die Übertragung des Programms auf grobe Fehler überprüft werden. Die Sprachparameter werden auf die gleiche Weise übertragen, indem zunächst die Parameter in Textform gebracht und anschließend an ZORC gesendet werden. Bei der Übertragung der Sprachparameter wird keine Überprüfung auf Fehler vorgenommen.

Nachdem die zur Programmausführung notwendigen Daten an ZORC übertragen wurden, wird die Ausführung des geladenen GP-Programms initialisiert. Hierzu wird der entsprechende Programmpunkt aus dem ZORCBRAIN-Menü ferngesteuert ausgewählt. Direkt nach der Anwahl des Menüpunktes wird die Anzahl von Sekunden übertragen, nach der die Programmausführung automatisch beendet werden soll. Dieser in Kapitel 5.4.2 beschriebene Mechanismus erhält den im SIGEL-Experiment definierten Parameter “*Time to Simulate*” als Eingabe. Die Programmausführung wird im Anschluss daran gestartet, sobald ZORC wie in Kapitel 5.4.2 geschildert auf die Experimentierfläche gestellt wird.

Zur Berechnung der Fitness von Individuen, die auf ZORC übertragen und dort ausgeführt werden, sind zwei Methoden untersucht worden. Diese werden in den folgenden Abschnitten beschrieben.

6.4.3.1 Die kamera-basierte Fitnessbewertung

In der ursprünglich geplanten Form sollte die Fitnessbewertung nahezu vollständig automatisch durchgeführt werden. Mittels einer am Linux-Rechner angeschlossenen digitalen Kamera bzw. Webcam sollte die Bewegung von ZORC aufgezeichnet werden, während die Programme auf dem Roboter ausgeführt werden. Zu diesem Zweck sollte eine Vorrichtung ähnlich des in Kapitel 6.7 beschriebenen “Laufstalls” benutzt werden. Mit einem relativ trivialen Algorithmus wäre es dann möglich gewesen, den mit der Kamera aufgenommenen Roboter zu lokalisieren. Die von ZORC zurückgelegte Wegstrecke hätte so automatisch bestimmt werden können. Darüber hinaus enthalten solche Kamerabilder nicht nur Informationen über die zurückgelegte Distanz, sondern zusätzlich die Information über die Höhe des Roboters. Einer Fitnessfunktion ständen demzufolge mit Distanz und Höhe des Roboters die gleichen Informationen wie in der Simulationsumgebung zur Verfügung. In diesem Szenario hätte lediglich das Zurücksetzen von ZORC auf die Startposition manuell durchgeführt werden müssen.

Zur Realisierung dieser kamera-basierten Fitnessauswertung sollte auf dem Hüftelement des Roboters eine Infrarot-Leuchtdiode (IR-LED) angebracht werden. Die in digitalen Kameras verwendeten Bildsensoren reagieren äußerst empfindlich auf Licht im infraroten Spektrum. Der Algorithmus zur Lokalisierung des Roboters in den von der Kamera gelieferten Bildern sollte daher den hellsten Punkt in den Bilddaten auffinden. Bei normaler (künstlicher) Raumbeleuchtung, d. h. bei relativ geringem Anteil an Tageslicht und insbesondere ohne direkte Sonneneinstrahlung, hätte sich die IR-LED auf Grund der Empfindlichkeit der Bildsensoren als hellster Punkt auf den Bildern bemerkbar gemacht.

Eine Testimplementierung, die auf den Bildern der Webcam den hellsten Punkt sucht und dessen Weg im Verlauf der Zeit verfolgt, ist in Abbildung 6.6 gezeigt. Den dazu verwendeten Versuchsaufbau zeigt Abbildung 6.5. Auf dem Bildschirmfoto in 6.6 sind zwei eingezeichnete Kreise zu erkennen. Diese markieren den hellsten vom Programm erkannten Bildpunkt, der sich im Mittelpunkt der Kreise befindet. Die Position dieses Punktes im Zeitverlauf ist durch eingezeichnete Linien gekennzeichnet. Eine Kalibrierungsroutine zu Beginn einer Versuchsreihe hätte ausgereicht, um die Entfernung in Bildpunkten in Relation zu setzen zu der in Millimetern bzw. Zentimetern zurückgelegten Entfernung von ZORC. Diese Daten hätten anschließend zur Berechnung der Fitness des beobachteten Individuums eingesetzt werden können.

Dieser Ansatz konnte jedoch nicht mit den zur Verfügung stehenden Mitteln realisiert werden. Der Grund dafür liegt in der verwendeten Kamera. Für die Versuche wurde eine am Lehrstuhl für Systemanalyse zur Verfügung stehende Webcam eingesetzt. Diese nutzt einen Bildsensor mit integriertem IR-Filter. Da sich bei normalen Anwendungen die übermäßige Empfindlichkeit einer Webcam gegenüber infrarotem Licht eher störend auswirkt, wird bei den meisten am Markt befindlichen Kameras ein solcher Filter eingesetzt. In der Regel be-

6 Übertragung und Evolution von Laufalgorithmen auf ZORC

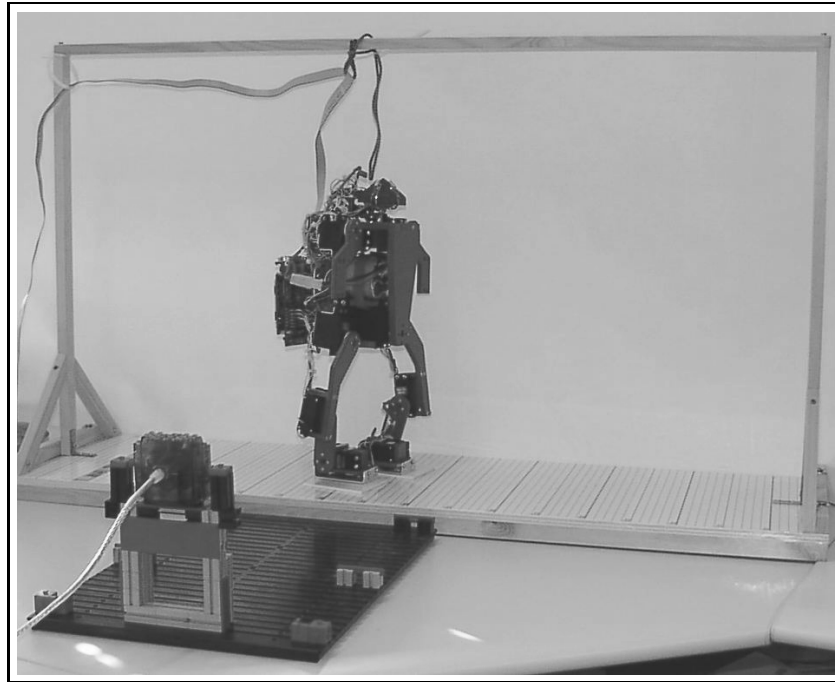


Abbildung 6.5: Der “Laufstall” mit Kamera. Das aufgenommene Kamerabild zeigt Abbildung 6.6.

stehen diese Filter aus einer einfachen Folie und sind problemlos zu entfernen. Bei der eingesetzten Kamera handelte es sich dagegen um ein fest mit dem Bildsensor verbundenes Filterelement. Daher konnte der Filter nicht entfernt werden, ohne dass dabei die Kameraelektronik zerstört worden wäre. Der IR-Filter arbeitete zudem sehr effektiv, so dass eine auf dem Hüftelement von ZORC angebrachte IR-LED auf den von der Kamera aufgenommenen Bildern nicht als Lichtquelle erkennbar war².

Ein alternativer Ansatz hätte darin bestanden, die Position des Roboters mit Algorithmen der Bildverarbeitung aus den Bilddaten zu extrahieren. Eine umfassende Übersicht über dieses Themengebiet geben z. B. [24] und [25]. In Abbildung 6.6 ist jedoch deutlich zu erkennen, dass die Kamera nur eine relativ schlechte Bildqualität bietet. Exakte Informationen über die Position von ZORC hätten daher nur durch eine aufwändige Bildverarbeitung aus diesen Bildern gewonnen werden können. Auf Grund der zeitaufwändigen Implementierung solcher Algorithmen wurde deshalb dieser Ansatz nicht weiter verfolgt.

Im Verlauf der Versuche mit der Webcam stellten sich zudem weitere Nachteile heraus. Die Kamera führte z. B. eine Kalibrierung durch, sobald die Digitalisierung der Bilder durch das Programm gestartet wurde. Diese automatische Einstellung auf die Beleuchtungsverhältnisse wurde von der Kameraelektronik selbständig durchgeführt und dauerte bis zu 10 Sekunden. Während dieses Zeitraumes waren die von der Kamera gelieferten Bilder auf Grund starker Helligkeitsschwankungen und z. T. extrem kontrastarmer Bilder nicht brauchbar. Diese

²Die in Abb. 6.6 markierten Bildpunkte befinden sich daher nur zufällig innerhalb der Umrissse von ZORC. In der Regel wurde ein Bildpunkt mit zufälliger Position als hellster Punkt auf den Kamerabildern markiert.

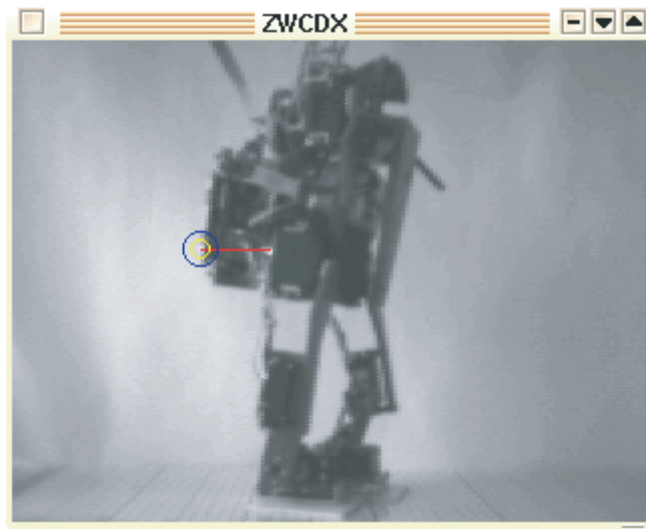


Abbildung 6.6: Ein Bildschirmfoto der Testimplementation aus Kapitel 6.4.3.1. Der hellste erkannte Bildpunkt befindet sich im Mittelpunkt der Kreise. Die eingezeichneten Linien zeigen den Weg dieses Punktes im Zeitverlauf an.

Kalibrierungs-Prozedur wäre vor jeder Fitnessevaluierung durchgeführt worden und hätte daher zu deutlichen Verzögerungen bei der Durchführung der Experimente geführt. Darüber hinaus bot die eingesetzte Kamera mit ca. 5 Bildern pro Sekunde eine relativ niedrige Bildrate. Mit dieser groben zeitlichen Auflösung wären z. B. kaum Unterschiede feststellbar gewesen in der Zeitspanne, die unterschiedliche Individuen den Roboter auf den Beinen zu halten vermochten. Gerade in der Anfangsphase der künstlichen Evolution sind diese feinen Unterschiede zwischen Individuen jedoch wichtig für die Steuerung des Lernprozesses.

Außer der verwendeten Webcam stand keine weitere Kamera zur Verfügung, die für den eingesetzten Linux-Rechner geeignet war. Auf Grund der beschriebenen Probleme mit diesem Modell wurde daher auf eine kamera-basierte Fitnessevaluierung verzichtet und die im Folgenden beschriebene Vorgehensweise gewählt.

6.4.3.2 Die halb-automatische Fitnessbewertung

Die schließlich realisierte Fitnessbewertung von Individuen, die auf ZORC ausgeführt wurden, bediente sich eines halb-automatischen Ansatzes. Die zurückgelegte Distanz des Roboters musste von der Person, welche die Versuche durchführte, manuell ermittelt werden. Die Zeitspanne, die das Individuum den Roboter aufrecht halten konnte bevor dieser ggf. umfiel, wurde vom ZORCBRAIN-Programm auf dem EYEBOT mit Unterstützung des Experimentators ermittelt. Dieser Wert wurde nach Abschluss einer Evaluierung an den Rechner übermittelt, auf dem die SIGEL-Applikation ausgeführt wurde.

Die Versuche mit ZORC wurden in einer Vorrichtung durchgeführt, die im Detail im nachfolgenden Kapitel 6.4.4 beschrieben wird. Zu Beginn jeder Fitnessevaluierung wurde ZORC

6 Übertragung und Evolution von Laufalgorithmen auf ZORC

auf die dort angebrachte Startmarkierung gesetzt. Sobald die Evaluierung abgeschlossen war, wurde die zurückgelegte Strecke auf einer Zentimeter-Skala abgelesen. Dieser Entfernungswert wurde anschließend manuell an SIGEL übergeben. Zu diesem Zweck öffnet die Fitnessfunktion “RemoteZorc” ein Dialogfenster, sobald die Evaluierung abgeschlossen wurde. In dieses wurde die von ZORC zurückgelegte Entfernung manuell eingegeben.

Gemessen wurde im Gegensatz zu den simulierten Robotern nicht die zurückgelegte Distanz des Hüft-Elements, sondern – aus Gründen der Praktikabilität – die Position der Fußelemente. Hierdurch ergab sich jedoch ein entscheidender Vorteil gegenüber der Simulation. Dort konnte auch durch das Umfallen des Robotermodells eine gewisse Distanz erreicht werden und dadurch die künstliche Evolution in eine falsche Richtung gelenkt werden. Solche Effekte sind in der Fitnessfunktion “RemoteZorc” nicht zu erwarten, zumal ein menschlicher Versuchsleiter die Experimente durchführt. Indem also wenig sinnvolle Versuche durch entsprechend geringe Distanzen bewertet werden können, erhält die Fitnessfunktion in gewisser Hinsicht einen zusätzlichen subjektiven Faktor. Daher war es möglich, die Fitnessfunktion gegenüber den in der Simulationsumgebung eingesetzten Fitnessfunktionen zu vereinfachen. Die etwa in “BigDistance-2” eingesetzten Verfahren, etwa die Bewertung einer gleichmäßigen Fortbewegung, hatten einzig den Zweck, unerwünschte Nebeneffekte abzufangen. Diese werden in der “RemoteZorc”-Fitnessfunktion durch die subjektive Beurteilung der Laufprogramme kompensiert. Zudem wurde im Verlauf der Versuche diese manuelle Eingabemöglichkeit gezielt zur Steuerung der Evolution eingesetzt. Insbesondere zu Beginn der Versuche wurden Programme, die ein erfolgversprechendes Verhalten zeigten, “ermutigt”. Beispielsweise wurden vielversprechende Bewegungsabläufe eines Beines geringfügig besser bewertet als ein Programm, das den Roboter lediglich in eine Standposition bewegt hat. Beide Programmtypen bewegten den Roboter nicht von der Startposition, da allerdings die Bewegungssequenz einen Teilerfolg auf dem Weg zu einem Laufalgorithmus darstellte, wurde dieses Programm mit einer geringen Distanz “belohnt”.

Die Zeitspanne, die ein Laufprogramm ZORC im Gleichgewicht zu halten vermochte, wurde vom Experimentator gestoppt. Falls der Roboter vor Ablauf der vorgesehenen Zeitspanne zu Boden fiel, wurde dies über eine Tastatureingabe am Linux-Rechner signalisiert. Daraufhin wurde von der SIGEL-Applikation die Programmausführung auf ZORC über die serielle Schnittstelle gestoppt. Die Zeitspanne, die das Laufprogramm bis zu diesem Zeitpunkt im Interpreter von ZORCBRAIN ausgeführt wurde, ist anschließend über die serielle Verbindung an SIGEL übertragen worden. Für Individuen, die den Roboter während der Evaluierung nicht zu Fall brachten, wurde die zur Evaluierung vorgesehene Zeitspanne übertragen. Diese betrug wie in den meisten Experimenten in der Simulationsumgebung 10 Sekunden.

Für die Fitnessfunktion standen somit die tatsächlich evaluierte Zeitspanne t und die zurückgelegte Distanz d zur Verfügung. Die Fitness Fit_I eines Individuums I wird von der Fitnessfunktion “RemoteZorc” berechnet als:

$$Fit_I = d \cdot t \quad (6.1)$$

Es ist dabei erwähnenswert, dass der Faktor t in diesem Zusammenhang nicht die gleiche Bedeutung hat wie in den Fitnessfunktionen, die für die Simulationsumgebung eingesetzt

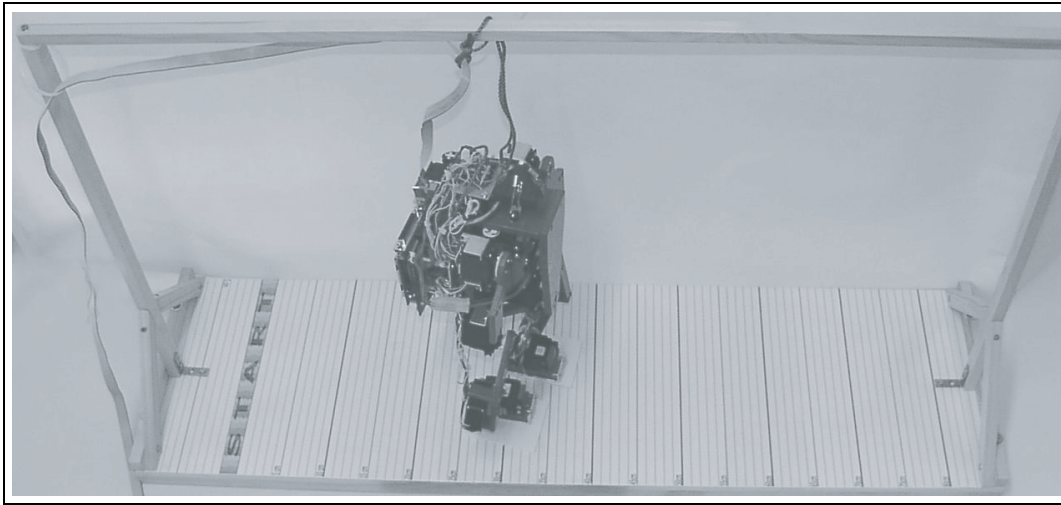


Abbildung 6.7: Die “Laufstall”-Vorrichtung mit eingespanntem ZORC.

wurden. Da t lediglich die tatsächlich evaluierte Zeitspanne umfasst, enthält der Faktor die implizite Information, ob das Laufprogramm den Roboter zu Fall gebracht hat. In den zuvor beschriebenen Fitnessfunktionen wird mit t stets die vollständige Zeitspanne bezeichnet, die ein Individuum in der Simulation evaluiert wird, d. h. der Simulationsparameter “*Time to Simulate*”.

6.4.4 Der “Laufstall”

Für die Durchführung der Experimente mit ZORC wurde eine Vorrichtung eingesetzt, die im Wesentlichen zwei Aufgaben zu erfüllen hatte. Wie im vorangegangenen Kapitel erläutert wurde, musste für die Fitnessauswertung die zurückgelegte Distanz manuell gemessen werden. Daher wurde die in Abbildung 6.7 dargestellte Basisplatte eingesetzt, auf der eine Zentimeter-Einteilung angebracht wurde. Anhand dieser war die vom Roboter beginnend von der “Start”-Markierung zurückgelegte Strecke leicht abzulesen. Diese Variante ist offensichtlich weitaus weniger fehleranfällig als beispielsweise eine Messung mit einem Lineal, da die Startposition fixiert ist und die Messeinrichtung sich während Fitnessauswertung nicht verschieben kann. Die Ausmaße der Basisplatte von $95 \times 25\text{cm}$ richteten sich nach den zu diesem Zeitpunkt bereits vorliegenden Simulationsergebnissen, bei denen die besten Individuen in 10 Sekunden eine Entfernung von rund 70cm zurücklegten (s. Kapitel 4.2.2). Auch die Resultate der bereits auf dieser Roboterarchitektur erprobten Laufalgorithmen legten ähnliche Werte nahe [1].

Der zweite Grund für den Einsatz dieser Vorrichtung bestand in der Schonung der Roboterhardware, insbesondere in Anbetracht der zahlreichen zu erwartenden Stürze während der Lernphase. Aus diesem Grund wurde über der erwähnten Basisplatte in etwa 50cm Höhe eine Holzschiene befestigt. An dieser wurde eine Nylonschnur angebracht, in die ZORC während der Experimente eingehängt wurde, wie Abbildung 6.7 veranschaulicht. Die Anbringung der

6 Übertragung und Evolution von Laufalgorithmen auf ZORC

Schiene parallel zur Laufstrecke ermöglicht es dem Roboter, die Nylonschnur mit sich zu ziehen. Schließlich wurde die Schnurlänge so gewählt, dass der Roboter sich ungehindert bewegen und einige Zentimeter zu beiden Seiten gehen kann, bei einem Fall allerdings aufgefangen wird. Auf diese Weise werden die Gelenke geschont und insbesondere verhindert, dass die Elektronik des Roboters – das EYEBOT-Mikrocontrollerboard und diverse Platinen für die in Kapitel 2 beschriebenen Erweiterungen – geschädigt werden kann. Dieser “Laufstall” für ZORC konnte zudem dazu genutzt werden, das am EYEBOT angeschlossene Verbindungskabel zum Desktoprechner über dem Kopf des Roboters zu montieren, damit es bei der Durchführung der Experimente nicht stören konnte.

6.4.5 Weitere technische Aspekte der Evolution auf ZORC

Bei der Durchführung der Experimente stellte ein schonungsvoller Umgang mit der Roboterhardware einen weiteren wichtigen Aspekt dar. Es wurde schon erwähnt, dass die Servomotoren in der ZORC-Architektur starken Belastungen ausgesetzt sind. Verglichen mit früheren Experimenten mit ZORC und ähnlichen Erfahrungen mit der Roboterarchitektur an der “Chalmers University of Technology” in Göteborg ist es überraschend, dass lediglich zwei Servomotoren während dieser Arbeit ausgetauscht werden mussten.

Hierzu hat der Umgang mit dem Roboter während der Experimente möglicherweise positiv beigetragen. Neben der durch ZORCBRAIN erzwungenen Pause von 20 Sekunden zwischen zwei aufeinander folgenden Programmevaluierungen (s. a. 5.4) wurden regelmässig längere Pausen eingelegt. Üblicherweise wurde pro Stunde 15-20 Minuten pausiert, so dass stündlich lediglich 40-50 Evaluierungen durchgeführt wurden. Bei durchschnittlich 8 Stunden, in denen täglich Experimente mit ZORC durchgeführt wurden, sind daher etwa 350 Fitnessauswertungen pro Tag vorgenommen worden.

Zusätzliche Pausen waren erforderlich, um regelmäßig das verbrauchte Akkupack zu wechseln. Außerdem musste selbst beim Einsatz von 3 Akkupacks, von denen bis zu zwei parallel aufgeladen wurden, regelmäßig längere Wartezeiten auf Grund verbrauchter Batterien eingelegt werden.

6.4.6 Die Evolution eines Laufalgorithmus auf ZORC

Ausgehend von den Ergebnissen des Experiments mit der Fitnessfunktion “BigDistance-2” aus Kapitel 4.2.3 wurde ein Experiment mit identischen Parametern und der Fitnessfunktion “RemoteZorc” gestartet. Zu diesem Zweck wurden wie zuvor erwähnt alle 50 Individuen des Experiments auf ZORC ausgeführt und evaluiert. Anschließend wurde die Population auf 20 Individuen reduziert, wobei mit Ausnahme des besten Individuums alle 20 Programme willkürlich ausgewählt wurden. Diese Reduktion der Population hat sich als notwendig erwiesen, da die Anzahl der Fitnessauswertungen, die mit manueller Unterstützung realisierbar sind, wesentlich geringer ist als die Anzahl der in der vollautomatischen Simulationsumgebung durchführbaren Evaluierungen. Im Unterschied zum “BigDistance-2”-Experiment wurde hinsichtlich der Experimentparameter lediglich die Anzahl der Turniere pro Generati-

6.4 Die fortgeführte Evolution auf ZORC

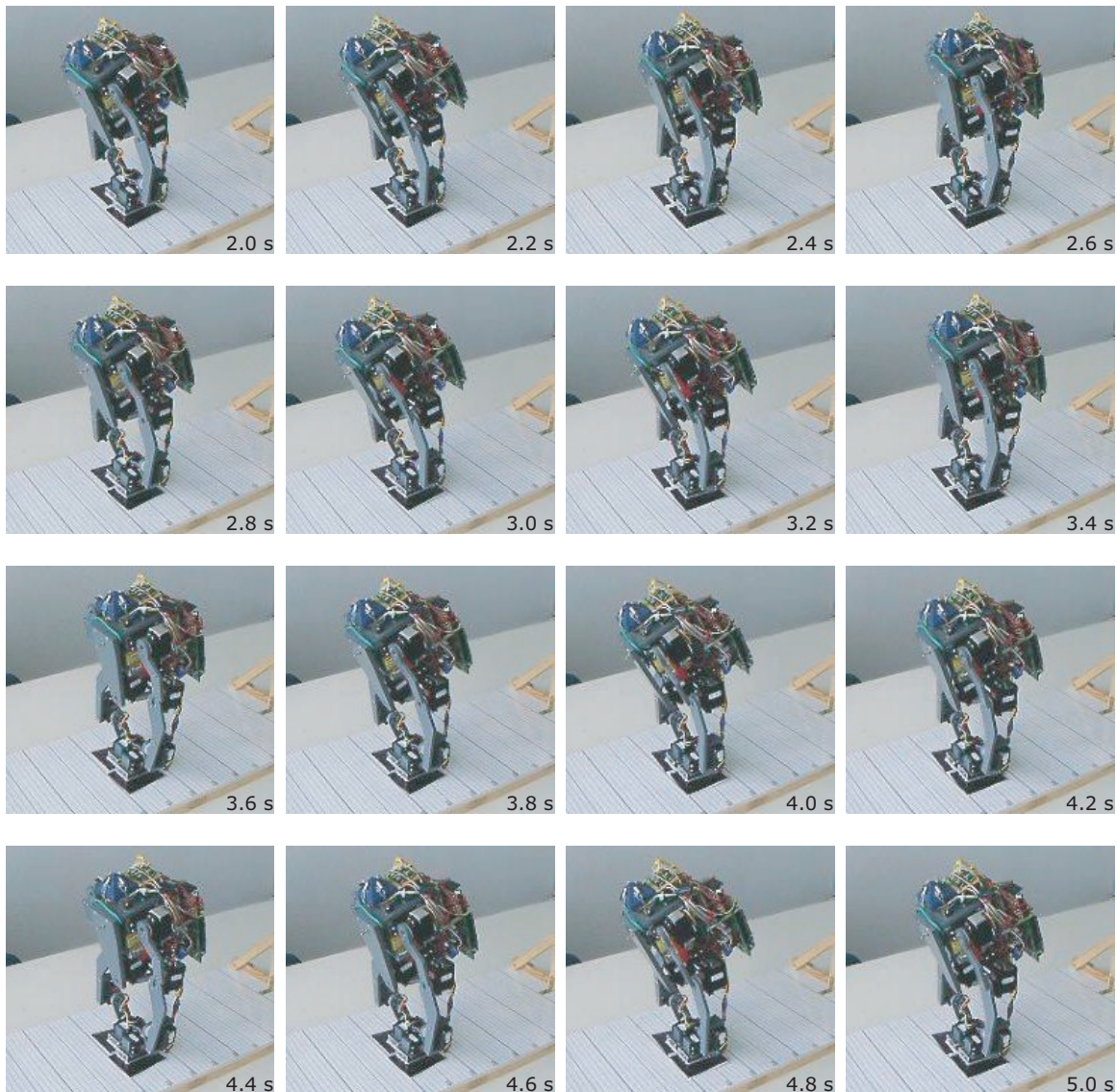


Abbildung 6.8: Dargestellt ist der Bewegungsablauf des besten Individuums, das während der Evolution auf der realen ZORC-Architektur mittels Genetischer Programmierung erlernt wurde.

on an die kleinere Population angepasst. Dieser Parameter wurde auf 5 Turniere pro Generation gesenkt. Die Simulationszeit betrug für das Experiment auf ZORC ebenfalls 10 Sekunden.

Im Sinne einer “interaktiven Evolution” wurde in den Lernprozess sporadisch eingegriffen, um den Nachteil der verhältnismäßig kleinen Population zu kompensieren. Immer dann, wenn im Verlauf dieses Prozesses neue Individuen auftauchten, die eine vielversprechende Neuerung darstellten, wurden diese manuell dupliziert. Jeweils 25 bis 50% der Population wurden durch das betreffende Individuum ersetzt. Diese Vorgehensweise hat sich im Verlauf des Experiments als erfolgreich erwiesen, da sich die neuen Verhaltensweisen schnell in der Population durchsetzen konnten. Es standen damit insbesondere mehr Möglichkeiten für das

6 Übertragung und Evolution von Laufalgorithmen auf ZORC

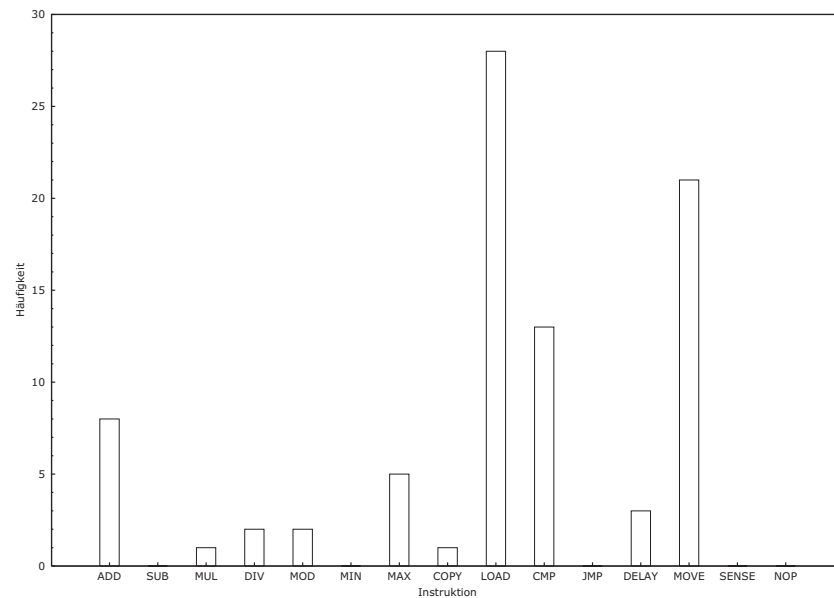


Abbildung 6.9: Eine Auswertung der Häufigkeit der einzelnen Instruktionen für den evolvierten Laufalgorithmus.

GP-System zur Verfügung, mit diesen neuen Verhaltensweise zu “experimentieren”. Bei diesen manuellen Eingriffen wurde dennoch auf eine möglichst hohe phänotypische Diversität innerhalb der Population geachtet, in dem z. B. nicht nur die schlechtesten Individuen durch Kopien des neuen Individuums ersetzt wurden.

Mit hoher Wahrscheinlichkeit hat diese Vorgehensweise zu dem raschen Erfolg dieses Experiments beigetragen. Innerhalb von 380 Generationen, d. h. 1900 Turnieren, hat das SIGEL-System auf der realen Roboterarchitektur einen Laufalgorithmus hervorgebracht, der den Roboter innerhalb von 10 Sekunden um etwa 4cm vorwärtsbewegt. Dieser in Abbildung 6.8 dargestellte Bewegungsablauf beinhaltet außerdem eine deutliche Seitwärtsbewegung, die jedoch nicht in die Fitnessbewertung eingeflossen ist. Zieht man diese Bewegung ebenfalls in Betracht, dann entfernt sich ZORC innerhalb von 10 Sekunden etwa 8-10cm von der Startposition. Der hierbei durch das GP-Programm realisierte Bewegungsablauf versetzt den Roboter zunächst in eine schwankende Bewegung. Diese Bewegung des Torsos ist in Abbildung 6.8 deutlich zu erkennen. Durch diese Gewichtsverlagerung lastet das Gewicht des Roboters abwechselnd auf der Vorderseite und der Rückseite der Fußelemente. Die auf diese Gewichtsverlagerung abgestimmte Bewegung beider Beine bewegt den Roboter schließlich vorwärts.

Der Programmcode des besten Individuums umfaßt 84 Instruktionen, wobei die maximal mögliche Programmlänge wieder 1024 Instruktionen betrug. Eine Auswertung der Häufigkeit der einzelnen Instruktionen dieses Individuums ist in Abbildung 6.4.6 dargestellt. Das nachfolgende Kapitel stellt u. a. die Struktur dieses Programms dem besten Laufprogramm gegenüber, das aus der Evolution in der Simulationsumgebung hervorgegangen ist. Neben der hohen Häufigkeit der Instruktionen MOVE und LOAD, die im Folgenden eingehender diskutiert wird, fällt vor allem das völlige Fehlen der SENSE- und JMP-Instruktionen auf. Das

Programm verzichtet demzufolge nicht nur auf jegliche Sensormessungen sondern gleichzeitig auf bedingte Verzweigungen. Das Vorkommen des Vergleichsbefehls `CMP` ist in diesem Zusammenhang nicht von Bedeutung, da dieser nur in Kombination mit dem Sprungbefehl `JMP` bedingte Verzweigungen realisieren kann. Diese "Sparsamkeit" lässt sich möglicherweise durch die lange Dauer einer `SENSE`-Instruktion begründen, die 0.05 Sekunden beträgt und damit die fünffache Zeit einer einfachen arithmetischen Operation in Anspruch nimmt. Gemäß dieser Betrachtungsweise kann der Verzicht auf die Anweisung `JMP` nur als konsequent bezeichnet werden. Ohne die Daten einer Sensormessung macht das bedingte Verzweigen in einem Programm, das beinahe ausschließlich aus `MOVE`-Instruktionen besteht, möglicherweise keinen Sinn mehr.

6.5 Vergleich der erlernten Laufalgorithmen

Im Rahmen dieser Arbeit konnten mittels Genetischer Programmierung zwei Laufalgorithmen erfolgreich gelernt werden. Zunächst wurde in der Simulationsumgebung der in Kapitel 4.2.2 gezeigte Laufalgorithmus erlernt. Die erwähnten Probleme bei der Übertragbarkeit dieser Resultate führten dazu, dass das im vorangegangenen Kapitel beschriebene Experiment als Fortsetzung der Ergebnisse aus Kapitel 4.2.3 durchgeführt wurde. Bei diesem wurde der Lernprozess auf `ZORC` mit einer vor-evolvierten Population gestartet, deren Individuen jedoch in der Simulationsumgebung noch keine erfolgversprechenden Bewegungsabläufe erlernt hatten.

Betrachtet man das Ergebnis aus Kapitel 4.2.3, dann fällt im Vergleich aller durchgeführten Experimente auf, dass durch die Verwendung von langsameren Instruktionen die Lernaufgabe offensichtlich signifikant erschwert wird. Die wahrscheinliche Ursache dieses Effekts lässt sich nachvollziehen, betrachtet man die Anfangsphase der Ausführung von Laufalgorithmen. Zu Beginn der Programmausführung sind sowohl in der Simulation als auch auf der realen Roboterarchitektur alle Motoren ausgeschaltet. Daher fällt der Roboter auf Grund der eigenen Schwerkraft sehr bald in sich zusammen und stürzt infolgedessen zu Boden, falls kein Motor durch geeignete `MOVE`-Instruktionen aktiviert wird. Um den Roboter daher zunächst in einer aufrechten Haltung zu stabilisieren, müssen relativ früh zu Beginn der Laufalgorithmen die Servomotoren aktiviert werden. Die hierzu notwendigen `MOVE`-Instruktionen müssen bei langsamer ablaufenden Programmen entsprechend früher im Programmcode auftauchen, damit die Motoren ausreichend schnell nach dem Programmstart eingeschaltet werden und der Roboter nicht hinfällt. Laufen die Programme schneller ab, dann können diese notwendigen `MOVE`-Instruktionen auch zu einem späteren Zeitpunkt im Programm auftauchen, werden jedoch durch die schnellere Programmausführung trotzdem "rechtzeitig" ausgeführt, um den Roboter zu stabilisieren. Im Vergleich der Sprachparameter der Experimente aus den Kapiteln 4.2.2 und 4.2.3 bedeutet dies, dass durch die langsameren Sprachparameter des "BigDistance-2"-Experiments der Bereich, in dem diese `MOVE`-Instruktionen im Programmcode auftreten müssen, um den Faktor 10-50 kleiner ist. Die Instruktionen müssen dort wesentlich präziser "positioniert" werden, um den Roboter lediglich in einer aufrechten Position zu halten. Dieser deutlich kleinere Anteil an erfolgversprechenden Programmen am Suchraum aller möglichen

6 Übertragung und Evolution von Laufalgorithmen auf ZORC

SIGEL-Programme führt offensichtlich dazu, dass der Schwierigkeitsgrad mit der Instruktionsdauer ansteigt. Aus dem gleichen Grund sind auch alle weiteren MOVE-Instruktionen, die für einen vollständigen Bewegungsablauf notwendig sind, wesentlich präziser im Programmcode zu positionieren.

Diese Vermutung wird gestützt durch einen Vergleich der jeweils besten Individuen der in der Simulation und auf ZORC durchgeführten Experimente. Offenbar gewinnen die MOVE-Instruktionen bei der Verwendung längerer Instruktionsdauern an Bedeutung. Betrachtet man die Instruktionsauswertung des besten Laufalgorithmus' mit der Fitnessfunktion "Big-Distance" aus Abbildung 4.5, lässt sich feststellen, dass der Anteil der MOVE-Instruktionen am Programmcode ca. 7.6% beträgt. Dahingegen bestehen beim besten Individuum des Experiments mit ZORC 25% des Programmcodes aus MOVE-Instruktionen. Lediglich die LOAD-Instruktionen haben bei diesem Individuum einen ähnlich hohen Anteil am Programmcode. Diese hängen mit hoher Wahrscheinlichkeit ebenfalls mit den Motorbewegungen zusammen, da der Motorwinkel nur über ein Register übergeben werden kann, dass dementsprechend häufig mittels LOAD neu geladen werden muss. Es kann also davon ausgegangen werden, dass in diesem Experiment die Instruktionen gewissermaßen "sorgfältiger ausgewählt" wurden. Die übrigen GP-Instruktionen sind für die konkrete Aufgabenstellung offenbar von untergeordneter Bedeutung, so dass diese im besten Individuum des ZORC-Experiments kaum vertreten sind.

Bei einer Betrachtung der resultierenden Bewegungsabläufe der besten Individuen beider Experimente fällt eine gemeinsame Eigenschaft der evolvierten Laufalgorithmen auf. In beiden Fällen wurden Variationen eines dynamischen Laufstils gelernt. Diese Resultate sind zunächst überraschend, da die ZORC-Architektur für einen statischen Gang entworfen wurde [1][2]. Wahrscheinlich war es für das GP-System von SIGEL im Verlauf der künstlichen Evolution jedoch einfacher, die beobachteten dynamischen Gangarten zu erlernen. Verglichen mit einem statischen Gang sind diese evolvierten Laufalgorithmen wesentlich simpler. Eine statische Gangart besteht beispielsweise aus wesentlich mehr Bewegungsphasen als die vorgestellten Resultate der künstlichen Evolution. Damit scheinen diese Ergebnisse die in Kapitel 4.2.5 getroffenen Vermutungen zu bestätigen, dass die Freiheitsgrade des hier eingesetzten Systems Lösungen hervorbringen können, die durch stark eingeschränkte Ansätze in dieser Form nicht realisierbar gewesen wären.

7 Ausblick

Die vorangegangenen Kapitel stellen die Ergebnisse dieser Arbeit vor. In Kapitel 4.2 werden die Resultate der Evolution von Laufalgorithmen in der Simulationsumgebung vorgestellt. Die Übertragung dieser Laufalgorithmen auf die reale Roboterarchitektur wird in Kapitel 5 behandelt. In diesem Kapitel sollen nun Aspekte angesprochen werden, die im Rahmen der Diplomarbeit nicht bearbeitet werden konnten und deren Untersuchung sich in Zukunft als interessant erweisen könnte. Im Rahmen einer Weiterentwicklung des SIGEL-Systems werden in Abschnitt 7.1 einige Ideen vorgestellt, die während der Arbeit mit dem ZORC-Robotermodell entstanden sind. Einen Ausblick für zukünftige Arbeiten mit der realen ZORC-Architektur gibt anschließend Abschnitt 7.2.

7.1 Die Zukunft von SIGEL

In Kapitel 3.3.6 wurde die Erweiterung von SIGEL für den Einsatz im Batchsystem behandelt. Der implementierte Mechanismus eignet sich dafür, die Berechnungen auf beliebige Rechner zu verteilen. Im Rahmen der Diplomarbeit wurden lediglich die Rechner des Lehrstuhls für Systemanalyse für SIGEL-Berechnungen genutzt. Prinzipiell wäre es mit dieser Erweiterung jedoch möglich, jeden Rechner an den Berechnungen zu beteiligen, der über einen Internetanschluss verfügt. Diese Rechner könnten sich dynamisch an den Berechnungen beteiligen, wenn sie beispielsweise nicht anderweitig genutzt werden. Allein die CPU-Ressourcen, die in vielen Unternehmen brachliegen, in denen Rechner oftmals auch außerhalb der Arbeitszeiten nicht ausgeschaltet werden, sind enorm. Analog zu Projekten wie SETI-AT-HOME [20] und QOOPY [21] könnte diese brachliegende Rechenleistung für SIGEL-Berechnungen eingesetzt werden. Die Rolle der Verteilung von Rechenressourcen durch das Batchsystem könnte dezentral durch die teilnehmenden Rechner selbst übernommen werden. Beispielsweise könnte ein einfaches Programm die Rechenlast überwachen und bei niedriger Last Kontakt zum Server aufnehmen, auf dem die SIGEL-Masterapplikation ausgeführt wird. Der Rechner würde anschließend für einen kurzen Zeitraum, beispielsweise für eine Fitnessberechnung, an einer aktuellen SIGEL-Berechnung teilnehmen. Die weitere Vorgehensweise verliefte in diesem Szenario analog zu dem in Kapitel 3.3.6 geschilderten Ablauf. Die zur Berechnung notwendigen Dateien und die `sigel_slave`-Applikation könnten einmalig auf den Rechnern installiert werden. Selbst die bei vielen derartigen Projekten integrierte "Belohnung" für die Eigentümer der teilnehmenden Rechner wäre mit wenig Mehraufwand realisierbar. Denkbar wäre z. B. die Visualisierung des aktuell besten Individuums der gesamten Population, die parallel zu der im Hintergrund ablaufenden Berechnung stattfinden könnte. Die auf diese

Weise gewonnene Rechenleistung ließe sich u. a. für die Evaluierung größerer Populationen und zur Simulation von längeren Zeitspannen einsetzen.

Die Problematik der in Kapitel 6.1 beschriebenen “kritischen” SENSE-Instruktionen könnte durch eine Erweiterung des SIGEL-Systems möglicherweise vollständig eliminiert werden. Verursacht durch eine perfekte Simulation des Robotermodells können die dort beschriebenen Programme entstehen, die in gewisser Hinsicht auf den immer gleichen Ablauf der Simulation überangepaßt sind. An dieser Stelle könnte ein Mechanismus ansetzen, der in die Simulationsumgebung Rauschen (Noise) einbringt. Aus Anwendersicht würde lediglich ein neuer Parameter in den Simulationseinstellungen auftauchen. Über einen solchen Noise-Parameter könnte bestimmt werden, wie stark das simulierte Rauschen die Simulation beeinflussen soll. Sinnvolle Werte für den Noise-Pegel N würden sich voraussichtlich im Rahmen von 0 (kein Rauschen) bis 10% (starkes Rauschen) bewegen. Dieses Rauschen kann das SIGEL-System auf mindestens drei verschiedene Weisen “stören” und damit die Evolution von robusteren Steuerungsprogrammen motivieren. Einerseits können Sensormessungen mit einem zufälligen Faktor von $0 - N\%$ verfälscht werden. Dieser zufällige Faktor sollte Störungen produzieren, die prozentual vom Messbereich des entsprechenden Sensors abhängen. Den zweiten Angriffspunkt für das künstliche Rauschen könnten die Motorbewegungen der Robotermodelle darstellen. Je nach Motortyp sollte die Ansteuerungskraft oder der Stellwinkel um $0 - N\%$ des zulässigen Bereiches von dem Wert abweichen, der durch eine MOVE-Instruktion befohlen wurde. Schließlich kann das künstliche Rauschen die Startposition des Roboters in der Simulation beeinflussen. Abhängig von der Größe des Robotermodells sollte der Roboter in der Ebene um $0 - N\%$ der Robotergröße verschoben werden. Um Kollisionen mit dem Boden zu vermeiden, sollte die Höhe über dem Boden nicht modifiziert werden. Die Konfiguration des Robotermodells könnte ebenfalls um $0 - N\%$ modifiziert werden, indem alle Gelenke um einen zufälligen Faktor innerhalb des zulässigen Bereiches bewegt werden. Schließlich kann das Robotermodell an der Startposition um einen zufälligen Wert von $0 - N\%$ von 360° um die Höhenachse (“Yaw”) rotiert werden. Damit würde die Richtung, in die der Roboter “losläuft”, minimal variiert. Die Beeinflussung dieser drei Faktoren – Sensormessung, Motorsteuerung und Startposition – der Simulation könnte die Evolution von realistischeren Laufalgorithmen zur Folge haben. Insbesondere für eine bessere Übertragbarkeit der Simulationsergebnisse auf real existierende Roboter wäre die Integration von Rauschen in das Simulationssystem äußerst hilfreich.

Eine weitere Möglichkeit, die Übertragbarkeit von Simulationsergebnissen zu verbessern, könnte sich an der Arbeit von Parker und Mills orientieren [18]. Dort wurde ebenfalls ein Laufalgorithmus für einen realen Roboter zunächst in der Simulation gelernt. Um die Übertragbarkeit zu gewährleisten, wurden jedoch schon während der Lernphase alle Algorithmen in regelmäßigen Abständen auf der realen Roboterarchitektur ausgeführt. Die bei dieser Ausführung gewonnenen Informationen wurde anschließend genutzt, um die Fitness aller Individuen der Population zu gewichten. Durch diese Berücksichtigung der Übertragbarkeit hatten gut übertragbare Individuen im weiteren Verlauf der Simulation bessere Chancen, selektiert zu werden. Eine solche fortlaufende Anpassung wäre in SIGEL realisierbar etwa als eine Kombination der Fitnessfunktionen “BigDistance” und “RemoteZorc”. Ein solcher Ansatz hätte sehr wahrscheinlich eine enorme Verbesserung der Übertragbarkeit von evolvierten

Laufalgorithmen auf reale Roboterarchitekturen zur Folge.

Schließlich kann eine Weiterentwicklung der eingesetzten Fitnessfunktionen der Gegenstand weiterer Arbeiten mit SIGEL sein. Im Rahmen dieser Arbeit hat sich angedeutet, dass insbesondere drei Aspekte in der Evolution von Laufrobotersteuerungsprogramme durch weiterentwickelte Fitnessfunktionen verbessert werden könnten: Höhen-Distanz-Gewichtung, Schrittweite und Geradlinigkeit. Die Experimente haben gezeigt, dass sowohl die Höhe des Roboters als auch die zurückgelegte Distanz als wesentliche Faktoren in den Fitnessfunktionen die Entwicklung von Laufalgorithmen steuern. Der Verlauf der Experimente deutete jedoch an, dass im Verlauf des Lernprozesses zunächst die Höhe bedeutsamer war. Nach dem Erlernen der aufrechten Haltung wirkte die Höhe sich eher hemmend aus und die Distanz steuerte im Wesentlichen den weiteren Verlauf des Lernprozesses. Für eine Optimierung des Lernprozesses würde es sich daher anbieten, eine dynamische Fitnessfunktion zu implementieren. Gesteuert durch geeignete Schwellenwerte in der Fitnessentwicklung der Population sollte die Fitnessberechnung zunächst die Roboterhöhe bevorzugt berücksichtigen. Im weiteren Verlauf – d. h. nach dem Erlernen einer Standposition – sollte die Bedeutung der Höhe ständig reduziert werden und der zurückgelegten Distanz eine stärkere Gewichtung eingeräumt werden.

In einer weitergehenden Form kann diese dynamische Variante der Fitnessfunktion bei erfolgreichen Laufalgorithmen zudem die Schrittweite integrieren. Analog zur zunehmenden Gewichtung der Distanz könnte die Schrittweite mit dem Erlernen eines Ganges eine zunehmende Bedeutung in der Fitnessberechnung erhalten.

Weitere Verbesserungen der Fitnessfunktionen könnten zum Ziel haben, die Laufalgorithmen zusätzlich anhand der Geradlinigkeit der Roboterbewegung zu bewerten. Dieser neue Aspekt in der Fitnessfunktion könnte der oft in Experimenten beobachteten Tendenz, Roboter auf einer bogenförmigen Bahn zu bewegen, entgegenwirken.

7.2 Die Zukunft von ZORC

Die ZORC-Architektur bietet Raum für diverse Verbesserungen. Zum Einen stellen die Aktuatoren von ZORC einen Schwachpunkt dar. Die eingesetzten Servomotoren bieten keinerlei Möglichkeit zur Geschwindigkeitsregelung, daher musste die beschriebene, interruptgesteuerte Servosteuerung implementiert werden. Einzelne Motoren müssen zudem häufig einen Großteil des Gewichts des Roboters tragen, indem entsprechend hohe Gegenkräfte aufgebracht werden. Dadurch wird die Lebensdauer der Motoren drastisch verkürzt und zudem viel Strom verbraucht. In der zukünftigen Arbeit mit ZORC könnten diese Probleme durch den Einsatz eines anderen Motortyps behoben werden. Die Servomotoren könnten z. B. durch Series Elastic Actuators (SEAs) ersetzt werden, die u. a. beim M2 Roboter eingesetzt wurden [11]. Der Einsatz dieses Motortyps würde zwar einen höheren mechanischen Aufwand bedeuten, dafür würden jedoch alle Haltekräfte vollständig von der Motormechanik aufgebracht. Der Motor selbst muss bei Series Elastic Actuators nur bei der Bewegung eines Gelenks in Aktion treten, daher brächte der Einsatz solcher Motoren auf ZORC voraussichtlich

7 Ausblick

eine beträchtliche Steigerung der Batterielaufzeit mit sich. Durch den Einsatz von Gleichstrommotoren ist außerdem die Geschwindigkeit der Motor- bzw. Gelenkbewegung über die Drehzahl frei regelbar.

In zukünftigen Untersuchungen könnte festgestellt werden, welche Rolle die Bewegung in der Roll-Achse für die Fußgelenke spielt und ob durch das Hinzufügen dieser zusätzlichen Freiheitsgrade für die ZORC-Architektur Vorteile entstehen könnten. Praktisch alle humanoiden Roboterarchitekturen, z. B. die Roboter M2 und Asimo [26], nutzen diesen zweiten Freiheitsgrad der Fußgelenke, um eine präzisere Kontrolle auf die Lage des Schwerpunktes auszuüben.

Im Zuge einer Umrüstung der Motortypen auf ZORC oder der Entwicklung eines Nachfolge-roboters könnte durch den Einsatz maschinengefertigter Teile eine wesentlich höhere Präzision erreicht werden. Durch eine Zusammenarbeit mit dem Fachbereich Maschinenbau könnten die an der Universität Dortmund ggf. zur Verfügung stehenden Ressourcen genutzt und Mehrkosten vermieden werden. Beispielsweise ließen sich durch den Einsatz von rechnergestützter Fertigung präzise Einzelteile fertigen. Bei der Erstellung des Robotermodells von ZORC wurden die einzelnen Roboterglieder bereits modelliert, so dass die notwendigen Geometriedaten für eine maschinelle Fertigung schon zur Verfügung stehen.

Die Erweiterung der Roboter-Elektronik resultierte in zahlreichen Platinen, die vergleichsweise provisorisch auf dem Roboter befestigt wurden. Die Unterbringung aller Komponenten auf einer eigens gefertigten Platine ist aus Gründen der Platzersparnis im Zuge einer Weiterentwicklung des Roboters empfehlenswert. Die Anbringung außerhalb des Robotertorsos führte zudem schon während der Versuche zu kleineren Defekten, da alle Kabel und die Platinen offen zugänglich waren. Zum Schutz der Elektronik wird zukünftig die Unterbringung dieser Komponenten in einem Gehäuse unumgänglich sein. Angelehnt an die momentane Befestigungsmethode würde sich z. B. eine Art "Rucksack" aus Kunststoff anbieten. Die Verwendung von leichten Materialien und eine Anbringung möglichst dicht am Robotertorso dürfte den Schwerpunkt des Roboters nicht wesentlich verlagern.

Bei der Evolution von Laufalgorithmen auf der realen ZORC-Architektur kann in zukünftigen Experimenten die Energieeffizienz im Vordergrund stehen. Die mit Hilfe des Stromaufnahmesensors gewonnenen Informationen können dabei genutzt werden, um automatisch Bewegungsmuster zu erlernen, die eine möglichst lange Batterielaufzeit gewährleisten. Dies gilt insbesondere für den Fall, dass weiterhin Servomotoren als Aktuatoren eingesetzt werden.

Eine auf diese Weise modifizierte ZORC-Architektur würde sich gemessen an dem internationalen Stand der universitären Forschung mit Sicherheit als konkurrenzfähig erweisen und eine solide Plattform für die weitergehende Erforschung von Laufalgorithmen bei humanoiden Roboterarchitekturen bieten.

ANHANG

A Die Steckerbelegungen der EYEBOT-Ports

PIN	BELEGUNG	PIN	BELEGUNG
1	N / C (Out 0)	9	Foot Switch, l (In 4)
2	D_1 (Out 1)	10	Foot Switch, r (In 5)
3	D_2 (Out 2)	11	N / C
4	D_3 (Out 3)	12	N / C
5	D_4 (Out 4)	13	VCC
6	D_5 (Out 5)	14	VCC
7	D_6 (Out 6)	15	GND
8	N / C (Out 7)	16	GND

Tabelle A.1: Pinbelegung des Digital-Ports auf ZORC, die logischen Bezeichner sind in den Klammern angegeben. Die Eingängsbits 0-3 sind fest mit den IR-PSDs (Tab. A.3) verdrahtet. ("N / C" = Not Connected)

PIN	BELEGUNG	PIN	BELEGUNG
1	VCC	6	Servo nn (In 5)
2	VCC	7	Strommessung (In 6)
3	RR-1 (In 2)	8	N / C (In 7)
4	RR-2 (In 3)	9	GND
5	RR-3 (In 4)	10	GND

Tabelle A.2: Pinbelegung des EYEBOT-MK4 Analog-Ports. Die Eingänge $In\ 0$ und $In\ 1$ sind fest mit dem Mikrophon (0) bzw. Batteriepegel (1) verbunden. „RR- n “ bezeichnet die Ausgänge 1-3 des Relais-Routers.

PSD	PINBELEGUNG 1	PINBELEGUNG 2	PINBELEGUNG 3	PINBELEGUNG 4
1	GND	V_{in} (Pulse)	VCC	Neigungssensor 1
2	GND	V_{in} (Pulse)	VCC	Neigungssensor 2
3	GND	V_{in} (Pulse)	VCC	Neigungssensor 3
4	GND	V_{in} (Pulse)	VCC	Neigungssensor 4

Tabelle A.3: Pinbelegung der 4 benutzten Infrarot-Ports (PSD 1-4) des EYEBOT-MK4 Controllers.

A Die Steckerbelegungen der EYEBOT-Ports

B Der GP-Befehlssatz von SIGEL

Die folgende Aufstellung erläutert kurz alle Befehle, die dem GP-System von SIGEL zur Verfügung stehen, um SIGEL-Programme zu evolvieren. Der gleiche Befehlsumfang steht SIGEL-Programmen zur Verfügung, die auf ZORC übertragen und dort ausgeführt werden. Die Befehlsdauer, d.h. die Zeitspanne, die bei der Ausführung einer Instruktion im Interpreter auf der durch den Interpreter simulierten CPU vergeht, wird für jeden Befehl individuell in den Sprachparametern festgelegt (s. Kapitel 5.2).

Um Überläufe zu verhindern, werden alle Argumente modulo der Anzahl der Register, des Wertebereichs der Register, der Anzahl der Sensoren oder Motoren sowie der Anzahl der Programmzeilen interpretiert. Nähere Einzelheiten zum Befehlssatz sowie zum Aufbau von Roboterprogrammen finden sich in Kapitel 4.3 in [6].

Im Folgenden wird mit $R(n)$ das Register mit dem Index n bezeichnet.

B.1 LOAD: Register laden

Syntax: LOAD X , Y

Beschreibung: Die Konstante Y wird in Register $R(X)$ geschrieben.

B.2 COPY: Registerinhalte kopieren

Syntax: COPY X , Y

Beschreibung: Der Inhalt von $R(Y)$ wird nach $R(X)$ kopiert.

B.3 ADD: Registerinhalte addieren

Syntax: ADD X , Y

Beschreibung: Die Inhalte der Register $R(X)$ und $R(Y)$ werden addiert. Das Ergebnis der Berechnung wird in $R(X)$ gespeichert.

B.4 SUB: Registerinhalte subtrahieren

Syntax: SUB X, Y

Beschreibung: Der Inhalt von Register $R(Y)$ wird vom Inhalt des Registers $R(X)$ subtrahiert. Anschließend wird das Ergebnis der Berechnung in $R(X)$ gespeichert.

B.5 MUL: Registerinhalte multiplizieren

Syntax: MUL X, Y

Beschreibung: Die Inhalte der Register $R(X)$ und $R(Y)$ werden multipliziert. Das Ergebnis der Berechnung wird in $R(X)$ gespeichert.

B.6 DIV: Registerinhalte dividieren

Syntax: DIV X, Y

Beschreibung: Der Inhalt von $R(X)$ wird durch den Inhalt von $R(Y)$ dividiert. Eine Division durch Null wird hierbei abgefangen und das Ergebnis in diesem Fall auf 0 gesetzt. Das Ergebnis der Berechnung wird in $R(X)$ gespeichert.

B.7 MOD: Modulo dividieren

Syntax: MOD X, Y

Beschreibung: Analog zur Division wird der Rest einer Division von $R(X)$ und $R(Y)$ bestimmt. Das Ergebnis der Berechnung wird in $R(X)$ gespeichert.

B.8 MIN: Minimum bestimmen

Syntax: MIN X, Y

Beschreibung: Das Minimum der Inhalte der Register $R(X)$ und $R(Y)$ wird in Register $R(X)$ geschrieben.

B.9 MAX: Maximum bestimmen

Syntax: MAX X, Y

Beschreibung: Der Maximum der Inhalte der Register $R(X)$ und $R(Y)$ wird in das Register $R(X)$ geschrieben.

B.10 CMP: Registerinhalte vergleichen

Syntax: CMP X , Y

Beschreibung: Falls der Inhalt des Registers $R(X)$ kleiner oder gleich dem Inhalt des Registers $R(Y)$ ist, wird das *Compare-Flag* gesetzt.

B.11 JMP: Bedingte Verzweigung

Syntax: JMP X

Beschreibung: Bei gesetztem *Compare-Flag* wird der Programmzähler um den Inhalt des Registers $R(X)$ inkrementiert, bzw. bei negativem Registerinhalt dekrementiert.

B.12 SENSE: Sensorwerte ermitteln

Syntax: SENSE X

Beschreibung: Der Inhalt des Registers $R(X)$ wird als Nummer eines Sensors interpretiert, dessen aktueller Meßwert in Register $R(0)$ geschrieben wird.

B.13 MOVE: Motoren bewegen

Syntax: MOVE X

Beschreibung: Der Inhalt des Registers $R(X)$ wird als Nummer eines Motors interpretiert, der als Eingabe den Inhalt von $R(0)$ erhält. Je nach Motortyp wird diese Eingabe unterschiedlich interpretiert¹.

B.14 DELAY: Programmausführung verzögern

Syntax: DELAY X

Beschreibung: Der Inhalt von $R(X)$ wird als Anzahl an Millisekunden interpretiert, um welche die Programmausführung verzögert wird.

B.15 NOP: Leere Instruktion

Syntax: NOP

Beschreibung: Die Instruktion hat keine Auswirkung auf die Programmausführung.

¹Bei SIGEL-Programmen, die auf ZORC ausgeführt werden, sind nur Motoren vom Servotyp (s. Kapitel 3.3.1) erlaubt. Daher wird der Wert in Register 0 als Stellwinkel für den Servomotor interpretiert

B Der GP-Befehlssatz von SIGEL

C Kurzanleitung zur Verwendung der SIGEL-Erweiterungen

Dieses Kapitel faßt kurz alle im Rahmen der Diplomarbeit entstandenen Erweiterungen zum ursprünglichen SIGEL-System aus Sicht des Anwenders zusammen. Syntax und Funktion der neuen Elemente zur Erstellung von Robotermodellen werden kurz dargestellt, um deren Verwendung bei der Erstellung neuer Roboter zu erleichtern. Für weitergehende Erklärungen zu den einzelnen Erweiterungen ist auf die jeweiligen Kapitel verwiesen.

Dieses Kapitel beschäftigt sich nur mit den Erweiterungen zum ursprünglichen SIGEL-System und versteht sich als Ergänzung des Handbuchs zu SIGEL im SIGEL-Endbericht ([6], Kap. 6). Für die Grundlagen der Erstellung von Robotermodellen sei daher auf die dort enthaltenen, ausführlichen Informationen verwiesen.

C.1 Motortyp SimpleServo

Dieser Motortyp simuliert einen winkelangesteuerten Servomotor, der einen beliebigen Winkel innerhalb der festgelegten Minimal- und Maximalauslenkungen des Gelenks anfährt und diese Position hält. Servomotoren lassen sich nur auf Gelenke vom Typ “Rotational” anwenden. Die MOVE Instruktion interpretiert in diesem Fall den Inhalt von Register 0 als Stellwinkel für den simulierten Servomotor. Liegt der Winkel außerhalb der in der Roboterdefinitionsdatei (s.a. [6], Kap. 6.3) definierten Minimal- bzw. Maximalauslenkungen, wird der Stellwinkel auf die Minimal- bzw. Maximalauslenkung korrigiert. Die Einzelheiten zu diesem Motortyp werden in Kapitel 3.3.1 beschrieben.

Für die Anwendung eines SIGEL-Servomotors in Robotermodellen muss in der Motordefinition der Motortyp mit “`simpleservo`” angegeben werden. Zusätzlich muss analog zu den anderen Motortypen die minimale und maximale Ansteuerungskraft in der Motordefinition enthalten sein. Da Servomotoren nur mit der maximalen Kraft angesteuert werden können, wird der Minimalwert allerdings ignoriert. Um Missverständnisse und Fehler zu vermeiden, empfiehlt es sich daher, in Roboterdefinitionsdateien beide Einträge auf den gleichen Wert, d. h. die maximale Ansteuerungskraft, zu setzen.

Eine vollständige Definition eines Servomotors ist im Folgenden am Beispiel des linken Fuß-Servos von ZORC gegeben (der zugehörige Joint für das linke Fußgelenk trägt die Bezeichnung “`ankle_l`”):

```
drive ankle_l_drive
{
    simpleservo ankle_l;
    minimalforce 1.30;
    maximalforce 1.30;
}
```

C.2 Sensortyp Pitch

Ein Sensor vom Typ `Pitch` kann an ein Roboterglied – in der SIGEL-Terminologie als Link bezeichnet – angefügt werden, um dessen Ausrichtung in der Pitch-Achse¹ zu messen. Das Ergebnis wird dabei immer relativ zur Startposition bzw. Startorientierung des Roboters ausgegeben und als Abweichung in Grad in Register 0 gespeichert. In Kapitel 3.3.2 wird die Funktionsweise und Implementation dieses Sensors ausführlich erläutert.

Bei der Verwendung eines Pitch-Sensors muss der Sensortyp in der Roboterdefinitionsdatei mit dem Schlüsselwort “`pitch`” angegeben werden. Analog allen anderen Sensortypen muss dem Sensortyp die Bezeichnung des Links folgen, dessen Ausrichtung gemessen werden soll. Weitere Angaben sind nicht notwendig.

C.3 Sensortyp Roll

Ein Sensor des Typs `Roll` verhält sich ähnlich wie ein Sensor vom Typ `Pitch`. Anstelle der Pitch-Achse wird die Ausrichtung in der Roll-Achse² gemessen. Details zu diesem Sensortyp finden sich ebenfalls in Kapitel 3.3.2.

Um einen Roll-Sensor zu verwenden, muss in der Sensordefinition das Schlüsselwort “`roll`” angegeben werden, zusammen mit dem Link, dessen Ausrichtung gemessen werden soll. Das folgende Beispiel demonstriert die Anwendung beider Sensortypen:

```
sensor zorc_pitch_sensor
{
    pitch zorc_torso;
}
sensor zorc_roll_sensor
{
    roll zorc_torso;
}
```

¹Bei humanoiden Roboterarchitekturen wird mit der Bezeichnung “Pitch” die in die Gehrichtung zeigende Achse bezeichnet. Bewegungen auf dieser Achse kippen den Roboter dementsprechend nach links oder rechts (s.a. Abb. 3.3 auf S. 21).

²Mit “Roll” wird bei humanoiden Roboterarchitekturen gemäß Abb. 3.3 die Achse bezeichnet, die beide Hüften miteinander verbindet, d. h. das Kippen in Gehrichtung oder in die entgegengesetzte Richtung misst.

C.4 Sensortyp Contact

Ein Kontakt-Sensor überprüft, ob der ihm zugeordnete Link Bodenkontakt hat. Dabei ist es unerheblich, welcher Teil des Robotergrundes den Boden berührt. Im Gegensatz zu allen anderen Sensortypen in SIGEL kann ein Kontakt-Sensor nur zwei Werte zurückliefern; fällt die Kontaktüberprüfung positiv aus, wird der Wert 1 in Register 0 geschrieben, anderenfalls der Wert 0. Weitere Einzelheiten zu diesem Sensortyp finden sich in Kapitel 3.3.3.

Zur Verwendung eines Kontakt-Sensors in einem Robotermodell muss die Sensordeklaration das Schlüsselwort `“contact”` enthalten, gefolgt von der Bezeichnung des Links, dem der Sensor zugeordnet werden soll.

Das folgende Beispiel zeigt anhand des simulierten linken Fußtasters von ZORC die Verwendung von Kontaktsensoren in der Roboterdefinitionsdatei:

```
sensor foot_l_contact_sensor
{
    contact foot_l;
}
```

C Kurzanleitung zur SIGEL-Erweiterung

D Dokumentation der ZORCBRAIN-Klassen

Dieses Kapitel enthält die Dokumentation zu den drei zentralen Klassen, aus denen das ZORCBRAIN-Programm besteht. Das auf dem EYEBOT-Mikrocontroller ablaufende Programm, das im Detail in Kapitel 5 beschrieben wird, steuert den Roboter. Die Klassen

- `CZorcBrain`
- `CEyeBotLoLevel`
- `ZORC_RS232Stream`

werden in den folgenden Abschnitten dokumentiert. Die darüberhinaus im ZORCBRAIN-Programm genutzten Klassen

- `ZORC_CmdParameters`
- `ZORC_Interpreter`
- `ZORC_LanguageParameters`
- `ZORC_Program`
- `ZORC_ProgramLine`
- `ZORC_Register`

wurden nicht weiter berücksichtigt, da es sich um die Anpassungen der entsprechenden SIGEL-Klassen handelt, die sich meist nur zu einem geringen Anteil von den Originalimplementationen im Rahmen des SIGEL-Projektes unterscheiden. Für eingehendere Informationen hierzu ist auf den SIGEL-Endbericht verwiesen [6]. Die Details zum Design des ZORCBRAIN-Programms werden in Kapitel 5 behandelt.

Zur automatischen Generierung der Dokumentation aus den dokumentierten Quelldateien wurde DOXYGEN eingesetzt [27]. Die Quelldateien wurden in Anlehnung an die bereits vorhandene Dokumentation der SIGEL-Klassen in englischer Sprache dokumentiert.

D.1 ZorcBrain Class Documentation

D.1.1 CEyeBotLoLevel Class Reference

```
#include <CEyeBotLoLevel.h>
```

Public Methods

- CEyeBotLoLevel ()
- ~CEyeBotLoLevel ()
- void TestModule (void)
- void MyLCDClear ()
- void displayServoSensorSettings (bool doSensors)
- void DisplayLoop (void)
- void CalibrateSensors (void)
- class ZORC_RS232Stream* Build_RS232Stream_Obj (void)
- int sendStringRS232 (const char *sendstring)
- int readStringRS232 (char *received, int mSize)
- char readCharRS232 (void)
- void pingRS232 (void)
- bool InputPendingForRS232 (void)
- void FlushRS232Input (void)
- int magicMenu (char *inItem1, char *inItem2, char *inItem3, bool isTopLevel=false)
- void EnsureAngleWithinServoLimits (int &ioAngle, int &inServoID)
- int setServo (int servoID, int angle, int steps)
- void ResetServos (void)
- void servoController ()
- void AssignServoCtrlTimer (TimerHandle inSCT)
- int S2Z_Sensor (int inSigelSensorID)
- int S2Z_Servo (int inSigelServoID)
- int angleToServoPos (int inAngle)
- int getServoMaxPos (int servoID)
- int getServoMinPos (int servoID)
- int getServoIniPos (int servoID)
- int getCurrentDraw (void)
- int getPitchDeg (void)
- int getRollDeg (void)
- int getSensorValue (int inSens)

- int getSensorDeg (int inSens)
- bool getDig (int inDChn)
- void setDig (int inDChn, bool inVal)
- void clearDig (void)
- int getBatteryLevel (void)
- int getNumberOfSensors (void)
- int getNumberOfServos (void)
- bool WaitSerOrCancel (void)
- void WaitButton (void)
- void WaitSerialOrButton (void)
- void errMsg (char *inErrStr)

Data Fields

- tSensorTrait ellSensorTrait [kSensorNum]
- tServoTrait ellServoTrait [kServoNum]
- tServoCtrlTabEntry ellServoCtrlTab [kServoNum]
- TimerHandle ellServoCtrlTimer

Private Methods

- int privSetServo (int servo_id, int pos)
- void allocServo (int servo_id)
- void freeServo (int servo_id)

Private Attributes

- bool ellServoCtrlInstalled

D.1.1.1 Detailed Description

Class CEyeBotLoLevel handles low-level accesses to the EyeBot MCU. All sensor readings, servo commands and access to the serial ports of the EyeBot-HW.

Definition at line 237 of file CEyeBotLoLevel.h.

D.1.1.2 Constructor & Destructor Documentation

CEyeBotLoLevel::CEyeBotLoLevel ()

Constructor of the CEyeBotLoLevel() class. This method initializes the serial interfaces, various other variables and the EyeBot hardware; The servoController() interrupt WON'T be installed.

CEyeBotLoLevel::~~CEyeBotLoLevel ()

Destructor of the CEyeBotLoLevel() class. Releases the servos and detaches the servo controller task.

D.1.1.3 Member Function Documentation

void CEyeBotLoLevel::AssignServoCtrlTimer (TimerHandle inSCT)

Simple accessor setting the servocontrovertime member. Required to set the timer when the servo-controller method is installed via external C method as interrupt.

Called by the CZorcBrain() with identical name when the servocontroller has ben assigned from outside (C-Namespace).

class ZORC_RS232Stream * CEyeBotLoLevel::Build_RS232Stream_Obj (void)

Return an object for handling RS-232 data communication more comfortable using a much an instance of ZORC_RS232Stream.

void CEyeBotLoLevel::CalibrateSensors (void)

A method to calibrate the sensors. This method should be called directly after the servo controller has been installed. Because it relies on this interrupt-driven controller it can't be called from the constructor, which doesn't exist then.

Referenced by CZorcBrain::CalibrateSensors().

void CEyeBotLoLevel::DisplayLoop (void)

Display sensor values repeatedly until any key is pressed.

void CEyeBotLoLevel::EnsureAngleWithinServoLimits (int & ioAngle, int & inServoID)

This methods checks if the angle lies within the servo limits given by the appropriate tServoTrait entries and returns a corrected value if necessary.

void CEyeBotLoLevel::FlushRS232Input (void) [inline]

This method flushed all pending input from serial interface kSerRemote.

Definition at line 355 of file CEyeBotLoLevel.h.

bool CEyeBotLoLevel::InputPendingForRS232 (void) [inline]

Call this method to check if there's any input pending on the kSerRemote serial interface.

Definition at line 350 of file CEyeBotLoLevel.h.

void CEyeBotLoLevel::MyLCDClear ()

Clears the LCD screen. Works better than the ROBIOS OS-Call claiming to do the same job.

void CEyeBotLoLevel::ResetServos (void)

Reset the servomotors. Moves them to initial positions – zero degrees for most, 90 degrees e.g. for hip-spreaders – and turns all servos off again.

Mainly used to reset the robot for a new program execution.

int CEyeBotLoLevel::S2Z_Sensor (int inSigelSensorID)

This method maps SIGEL sensor numbers to ZORC sensors. It is required because defining the enumeration of sensors in SIGEL robot models is impossible and the resulting enumeration of a parsed robot model somewhat random.

This method only works for the iZORC.rrb model used throughout the master thesis and has to be adjusted for different models !

int CEyeBotLoLevel::S2Z_Servo (int inSigelServoID)

This method maps SIGEL servo IDs to the matching ZORC servo-IDs. It is required because defining the enumeration of servos in SIGEL robot models is impossible and the resulting enumeration of a parsed robot model somewhat random.

This method only works for the iZORC.rrb model used throughout the master thesis and has to be adjusted for different models !

void CEyeBotLoLevel::TestModule (void)

Test the EyeBot HW-access module. The method listens to commands from serial port n using "X Y\n" syntax to move servo 'X' to angle 'Y'.

void CEyeBotLoLevel::WaitButton (void)

Wait for a button on the EyeBot to be pressed.

bool CEyeBotLoLevel::WaitSerOrCancel (void)

Wait for serial data to arrive or button to cancel, Returns true if serial data is waiting on port "kSerRemote"

Required to wait for the begin of a download to avoid the serial terminates prematurely due to timeout.

void CEyeBotLoLevel::WaitSerialOrButton (void)

Wait for either a button to be pressed or serial data to arrive. The serial data isn't processed but flushed from the buffer, so you shouldn't send more than one byte.

void CEyeBotLoLevel::allocServo (int servo_id) [private]

Allocate a servo. This actually turns on the servo and initializes the Robios structures required to control the servo. Method freeServo() releases the servo, turning it off.

Called internally, use 'setServo()' and 'ResetServos()' in user programs !

int CEyeBotLoLevel::angleToServoPos (int inAngle)

This method converts any given angle to a position for SETServo() in the range 0..255.

void CEyeBotLoLevel::clearDig (void)

Clear the digital port. This turns off all relais on the relais-router device thereby saving some power.

void CEyeBotLoLevel::displayServoSensorSettings (bool doSensors)

Write all servo positions and sensor values the display; depending on 'doSensors' either motor or sensor values are written to the display.

Method does not get any measurements but relies on the member variables containing valid values, e.g. filled in by method DisplayLoop().

void CEyeBotLoLevel::errMsg (char * inErrStr)

Display an error message. Use the whole display to show the message, make some noise and wait for a button to be pressed. The error is also transmitted via RS-232 to simplify debugging.

Referenced by CZorcBrain::errMsg().

void CEyeBotLoLevel::freeServo (int servo_id) [private]

int CEyeBotLoLevel::getBatteryLevel (void)

This method returns the battery level in percent (0..100).

Following the ROBIOS scheme (also see HDT information) the empty battery has an OSGetAD(1) reading of ≤ 550 whereas a battery completely charged has a reading of about 850. These limits are used to compute the current level using the OSGetAD() system call.

int CEyeBotLoLevel::getCurrentDraw (void)

Return the current draw as measured by the MAX472 IC; the value returned is Ampere*10 to be displayed with an accuracy of 1/10 A

bool CEyeBotLoLevel::getDig (int inDChn)

Return the value of a particular bit of the digital port.

int CEyeBotLoLevel::getNumberOfSensors (void) [inline]

This method returns the number of sensors known to CEyeBotLoLevel.

Definition at line 526 of file CEyeBotLoLevel.h.

int CEyeBotLoLevel::getNumberOfServos (void) [inline]

This method returns the number of servos known to our class.

Definition at line 531 of file CEyeBotLoLevel.h.

int CEyeBotLoLevel::getPitchDeg (void)

Returns the measurement of the inclination-sensors. Using two binary switches per axis the method returns either -45/0/45 degrees. The measurement is artificially assembled using the inclination sensors kHeadSensF and kHeadSensB.

int CEyeBotLoLevel::getRollDeg (void)

Returns the measurement of the inclination-sensors. Using two binary switches per axis the method returns either -45/0/45 degrees. The measurement is artificially assembled using the inclination sensors kHeadSensL and kHeadSensR.

int CEyeBotLoLevel::getSensorDeg (int *inSens*)

Return sensor reading in degrees (0..360). This method vitally depends on the CalibrateSensors() method to be called at startup, which on the CalibrateSensors() required to convert raw sensor readings to degrees. Call only for sensors of servo-type !

int CEyeBotLoLevel::getSensorValue (int *inSens*)

This method returns the raw sensor-value. You will most likely not call this function for the servo-sensors but rather getSensorDeg() returning processed measurements.

int CEyeBotLoLevel::getServoIniPos (int *servoID*) [inline]

Accessor method for getting a servos initial position.

Definition at line 459 of file CEyeBotLoLevel.h.

int CEyeBotLoLevel::getServoMaxPos (int *servoID*) [inline]

Accessor method for getting a servos max. position limit.

Definition at line 449 of file CEyeBotLoLevel.h.

int CEyeBotLoLevel::getServoMinPos (int *servoID*) [inline]

Accessor method for getting a servos min. position limit.

Definition at line 454 of file CEyeBotLoLevel.h.

int CEyeBotLoLevel::magicMenu (char * *inItem1*, char * *inItem2*, char * *inItem3*, bool *isTopLevel* = false)

This method simplifys managing the user interaction by setting the menu-bar to the given three entries and waiting for either a key or a remote-command, finally clearing the menu-bar and returning the KEY pressed. Only the menu-bar will be affected, no other element of the current display, i.e. LCDClear()/ MyLCDClear() is not called.

Only the first three items can be used since KEY4 is always used for exiting the current sub-menu (or QUITing for 'isTopLevel=true', default value is false). All Strings must be specified, use " " for empty strings, but at least 1 space or Menuitem won't be cleared.

Remote commands to control this menu consist of a single number "n", where n=1..4 stands for key 1-4.

void CEyeBotLoLevel::pingRS232 (void)

Call this method to respond to a "ZORC ping", which waits for any digit on serial interface 'kSerRemote' and returns the double value (i.e. send '1', receive '2').

int CEyeBotLoLevel::privSetServo (int servo_id, int pos) [private]

Internal method called by the servoController(). This method sets servomotors 'servo_id' (0,...,kServoNum) to 'position' (0..255), You shouldn't call it under no circumstances, use setServo() instead ! Servo-ID is translated to the actual phys. port by this simple calculation:

if (servoID-4) is < 0 -> use external controller else use servo-port (servoID-4)

(Returnvalues: 0=ok, -1=servos_switched_off, -2=illegal_angle, -3=illegal_servo_id)

char CEyeBotLoLevel::readCharRS232 (void)

Read a character from serial interface 'kSerRemote'.

int CEyeBotLoLevel::readStringRS232 (char * received, int mSize)

Read a "\n"-terminated string from RS232 (kSerRemote). Returns the read string length (w/o "\0"), or errorcode -1

int CEyeBotLoLevel::sendStringRS232 (const char * sendstring)

Send a '\0'-terminated string via RS232 (kSerRemote) and return number of sent chars. Returns the number of transmitted characters – or -1 on error.

void CEyeBotLoLevel::servoController ()

This method is attached to the timer and called via interrupts periodically to smoothen the servo motion.

void CEyeBotLoLevel::setDig (int inDChn, bool inVal)

Set the dig. port channel/bit 'outDChn' to input value 'outVal'

int CEyeBotLoLevel::setServo (int servoID, int angle, int steps)

Method called to move servos. Supply servoID to identify the servo, angle for destination angle (0..360) and the number of steps to use for a smooth and slow movement.

Init and allocate the servo when it's called for the first time; thus servos have to be explicitly controlled to be activated, behaving the same as in the simulation.

D.1.1.4 Field Documentation

tSensorTrait CEyeBotLoLevel::ellSensorTrait

Array containing sensor related data, e.g. measurements. /** Since making measurements – especially when switching the Relais Router – takes some considerable amount of time, the values aren't filled in automatically but have to be explicitly filled in. E.g. DisplayLoop() requires this for interaction with displayServoSensorSettings().

Definition at line 262 of file CEyeBotLoLevel.h.

bool CEyeBotLoLevel::ellServoCtrlInstalled [private]

This variable is true if the servo-controller has been installed.

Definition at line 573 of file CEyeBotLoLevel.h.

tServoCtrlTabEntry CEyeBotLoLevel::ellServoCtrlTab

This data structure contains the information to steer the servos using method servoController(), which is called via interrupt.

THOU SHALT NEVER TOUCH THIS STRUCTURE, unless you're knowing EXACTLY what you're doing! All fields are modified only from interrupts, except under very special (defined) conditions in allocServo(), freeServo() and setServo()

Definition at line 278 of file CEyeBotLoLevel.h.

TimerHandle CEyeBotLoLevel::ellServoCtrlTimer

This member contains the TimeHandle to identify the interrupt driving the servoController. Must be assigned using method AssignServoCtrlTimer().

Definition at line 567 of file CEyeBotLoLevel.h.

tServoTrait CEyeBotLoLevel::ellServoTrait

The ellServoTrait member contains various traits for the servomotors.

Definition at line 267 of file CEyeBotLoLevel.h.

The documentation for this class was generated from the following file:

- CEyeBotLoLevel.h

D.1.2 CZorcBrain Class Reference

```
#include <CZorcBrain.h>
```

Public Methods

- CZorcBrain ()
- ~CZorcBrain ()
- void AssignServoCtrlTimer (TimerHandle inSCT)
- TimerHandle GetServoCtrlTimer (void)
- void CalibrateSensors (void)
- void ServoPosSnapshot (void)
- void ExecSigelProgram (void)
- void LoadProgramRS232 (void)
- void LoadLangParameterRS232 (void)
- void Menu_Load (void)
- void Menu_OptSensors (void)
- void Menu_OptTest (void)
- void Menu_Options (void)
- void ZorcBrainManager (void)
- void errMsg (char *inErrStr)

Data Fields

- ZORC_LanguageParameters* zbParms
- ZORC_Program* zbProg
- CEyeBotLoLevel* zbEBLL
- int zbTTR
- int zbESPWaitSecs
- int zbLastCallToESP

D.1.2.1 Detailed Description

The CZorcBrain class is the central link between SIGEL and ZORC. It manages the behaviour of the ZORC hardware and controls the interaction with either the user or the remote computer connected to ZORC via RS-232.

Members of this class contain the current program – if any has yet been downloaded to ZORC – as well as the language parameters required for executing the program in the interpreter and an instance of the CEyeBotLoLevel class required to manage accesses to the EyeBot hardware.

Definition at line 41 of file CZorcBrain.h.

D.1.2.2 Constructor & Destructor Documentation

CZorcBrain::CZorcBrain ()

The constructor creates the CEyeBotLoLevel object, required to managing access the EyeBot hardware.

CZorcBrain::~~CZorcBrain () [inline]

Destructor method. This one deletes the CEyeBotLoLevel and ZORC.LanguageParameters objects.

Definition at line 54 of file CZorcBrain.h.

D.1.2.3 Member Function Documentation

void CZorcBrain::AssignServoCtrlTimer (TimerHandle inSCT)

Simple accessor setting the servocontrollertimer member. Required to set the timer when the servo-controller method is installed via external C method as interrupt.

void CZorcBrain::CalibrateSensors (void) [inline]

Call the EBLs method to calibrate the sensors.

Definition at line 71 of file CZorcBrain.h.

void CZorcBrain::ExecSigelProgram (void)

Executes the SIGEL program by creating a ZORC.Interpreter object which interpretes the program currently loaded. ZORC is resetted before the run using ResetServos(). The run is terminated by either pushing a button on the EyeBot, any serial input or a timeout. Any data received before the robot is set down to start executing is interpreted as time to run, i.e. the ZORC-pendant of the time to simulate in SIGEL.

TimerHandle CZorcBrain::GetServoCtrlTimer (void)

Return the timerhandle associated with the servocontroller irq.

void CZorcBrain::LoadLangParameterRS232 (void)

This method is used to load a SIGEL Language-Parameter file over the serial interface. Like programs the parameters must be transmitted without any protocol over the RS-232 connection at 115kps/8N1, hardware flowcontrol (RTS/CTS).

void CZorcBrain::LoadProgramRS232 (void)

This method loads a SIGEL program over the serial interface. Any SIGEL program transferred using no protocol at 115kps/8N1 will work (e.g. a 'cat PROGRAM > /dev/modem'), but most terminal programs use more complex protocols resulting in transmission errors.

void CZorcBrain::Menu_Load (void)

The method managing the "LOAD" submenu to load SIGEL programs and language parameter files.

void CZorcBrain::Menu_OptSensors (void)

This method manages the "OPTIONS -> SENSORS" submenu. Either make a snapshot of the current sensor measurements or display the sensor measurements on the EyeBot display.

void CZorcBrain::Menu_OptTest (void)

This method manages the "TEST" submenu. The CEyeBotLoLevel class can be tested as well as the contents of the current program by sending it via text over RS-232.

void CZorcBrain::Menu_Options (void)

This method handles the <OPTIONS> submenu choices. Additionally some information about the robot is displayed, including the uptime of the EyeBot and the current battery level.

void CZorcBrain::ServoPosSnapshot (void)

This method takes a snapshot of the current servo positions, converts them to match the servo-addresses of the SIGEL-Robotmodel and sends them over the RS-232 connection in form of a SIGEL program segment.

Please mind that the program is only guaranteed to work with either ZORC or SIGEL when the Registerwidth of the LanguageParameters is identical.

void CZorcBrain::ZorcBrainManager (void)

The ZorcBrain manager calls the various modules controlling the robot. It is the central method to interact with ZORC.

void CZorcBrain::errMsg (char * inErrStr) [inline]

This method calls the CEyeBotLoLevel method 'errMsg()' to display an error message on the EyeBot LCD.

D Dokumentation der ZORCBRAIN-Klassen

Definition at line 148 of file CZorcBrain.h.

D.1.2.4 Field Documentation

CEyeBotLoLevel * CZorcBrain::zbEBLL

The object to manage all accesses and interaction with the ZORC hardware.

Definition at line 170 of file CZorcBrain.h.

int CZorcBrain::zbESPWaitSecs

This parameter defines the number of seconds ZORC must wait between two consecutive runs of the program. The default value is 60, causing the ExecSigelProgram() method to wait until 60 seconds are passed when the method was called recently before.

Since the servos are getting quite hot and these things are really expensive the 'forced-delay' shall help to make the servos last longer.

Definition at line 187 of file CZorcBrain.h.

int CZorcBrain::zbLastCallToESP

A parameter required for zbTTR to work, storing the systemtime of the ROBIOS when ExecSigelProgram() was last called.

Definition at line 193 of file CZorcBrain.h.

ZORC_LanguageParameters * CZorcBrain::zbParms

The ZORC_LanguageParameters instance contains information about the allowed command set and duration of commands required by the interpreter to execute programs. Be aware of the fact that the register width defined by this object significantly determines how any interpreted program will behave on ZORC ! The initial object contains default values similar to these in initial SIGEL experiments. It can also be downloaded via RS-232.

Definition at line 159 of file CZorcBrain.h.

ZORC_Program * CZorcBrain::zbProg

This program object stores the current SIGEL program for our ZORC; There's no initial program, it must always be downloaded via RS-232 first.

Definition at line 165 of file CZorcBrain.h.

int CZorcBrain::zbTTR

The Time-To-Run parameter causes the ExecSigelProgram() method to stop interpreting the program automatically if it is defined (i.e. >0).

Definition at line 176 of file CZorcBrain.h.

The documentation for this class was generated from the following file:

- CZorcBrain.h

D.1.3 tSensorTrait Struct Reference

```
#include <CEyeBotLoLevel.h>
```

Data Fields

- int stValue
- int stZeroDegVal
- double stStepVal

D.1.3.1 Detailed Description

A structure to store all sensor related data.

Definition at line 186 of file CEyeBotLoLevel.h.

D.1.3.2 Field Documentation

double tSensorTrait::stStepVal

The amount a A/D measurement changes for moving the servo by one degree

Definition at line 194 of file CEyeBotLoLevel.h.

int tSensorTrait::stValue

This field may contain the sensor-measurement.

Definition at line 188 of file CEyeBotLoLevel.h.

int tSensorTrait::stZeroDegVal

The A/D measurement for the initial servoposition.

Definition at line 191 of file CEyeBotLoLevel.h.

The documentation for this struct was generated from the following file:

- CEyeBotLoLevel.h

D.1.4 tServoCtrlTabEntry Struct Reference

```
#include <CEyeBotLoLevel.h>
```

Data Fields

- int act_position
- int aim_position
- int stepsize

D.1.4.1 Detailed Description

A structure storing information required by the servo controller to manage the servos from the IRQ task.

Definition at line 225 of file CEyeBotLoLevel.h.

D.1.4.2 Field Documentation

int tServoCtrlTabEntry::act_position

Definition at line 226 of file CEyeBotLoLevel.h.

int tServoCtrlTabEntry::aim_position

Definition at line 227 of file CEyeBotLoLevel.h.

int tServoCtrlTabEntry::stepsize

Definition at line 228 of file CEyeBotLoLevel.h.

The documentation for this struct was generated from the following file:

- CEyeBotLoLevel.h

D.1.5 tServoTrait Struct Reference

```
#include <CEyeBotLoLevel.h>
```

Data Fields

- int svtMinPos
- int svtMaxPos
- int svtIniPos
- bool svtIsRev
- bool svtIsIniServo
- ServoHandle svtHandle

D.1.5.1 Detailed Description

A structure storing all servo related information. All these values should match the SIGEL robot-model of ZORC.

Definition at line 201 of file CEyeBotLoLevel.h.

D.1.5.2 Field Documentation

ServoHandle tServoTrait::svtHandle

Contains a ROBIOS handle for the servo

Definition at line 218 of file CEyeBotLoLevel.h.

int tServoTrait::svtIniPos

Contains the initial position of a servo.

Definition at line 209 of file CEyeBotLoLevel.h.

bool tServoTrait::svtIsIniServo

This field is true, if the servo has been initiated. See allocServo(), freeServo()

Definition at line 215 of file CEyeBotLoLevel.h.

bool tServoTrait::svtIsRev

If the motor should be steered in a reverse manner, this field is true.

Definition at line 212 of file CEyeBotLoLevel.h.

int tServoTrait::svtMaxPos

Contains the maximum allowed position.

Definition at line 206 of file CEyeBotLoLevel.h.

int tServoTrait::svtMinPos

The min. allowed position for a servomotor.

Definition at line 203 of file CEyeBotLoLevel.h.

The documentation for this struct was generated from the following file:

- CEyeBotLoLevel.h

D.1.6 ZORC_RS232Stream Class Reference

```
#include <ZORC_RS232Stream.h>
```

Public Methods

- ZORC_RS232Stream (class CEyeBotLoLevel *inEBLL)
- ~ZORC_RS232Stream ()
- void getStrLine (char *outStr, int outMaxLen)
- void getString (char *outStr, int outMaxLen)
- void getInt (int *outInt)
- void getFloat (float *outFloat)
- void sendStrLine (const char *outStr)
- bool InputAvailable (void)

Private Methods

- ZORC_RS232Stream ()

Private Attributes

- class CEyeBotLoLevel* rs232EBLL

D.1.6.1 Detailed Description

A class simplifying access to the RS-232 interface by accessing it in a stream-like manner, but with dedicated accessor methods for reading and writing various datatypes.

Definition at line 17 of file ZORC_RS232Stream.h.

D.1.6.2 Constructor & Destructor Documentation

ZORC_RS232Stream::ZORC_RS232Stream (class CEyeBotLoLevel * *inEBLL*) [inline]

The Constructor (the empty constructor must not be used). Since we access the interface only via the CEyeBotLoLevel class a pointer to this object must be provided.

Definition at line 26 of file ZORC_RS232Stream.h.

ZORC_RS232Stream::~~ZORC_RS232Stream () [inline]

The destructor. Does nothing at all.

Definition at line 31 of file ZORC_RS232Stream.h.

ZORC_RS232Stream::ZORC_RS232Stream () [private]

Dummy constructor.

D.1.6.3 Member Function Documentation

bool ZORC_RS232Stream::InputAvailable (void)

Returns true if any input is pending for the RS-232 input.

D Dokumentation der ZORCBRAIN-Klassen

void ZORC_RS232Stream::getFloat (float * *outFloat*)

Reads a word from RS-232, convert the result to a floating-point variable.

void ZORC_RS232Stream::getInt (int * *outInt*)

Reads a word from RS-232 and converts it to an integer.

void ZORC_RS232Stream::getStrLine (char * *outStr*, int *outMaxLen*)

Reads a number of characters from RS-232 until a newline '
' is read.

void ZORC_RS232Stream::getString (char * *outStr*, int *outMaxLen*)

Reads the next word from RS-232 until a whitespace character is read; with '*outMaxLen*' defining the max. number of characters to be returned.

void ZORC_RS232Stream::sendStrLine (const char * *outStr*)

Send a line of text over RS-232.

D.1.6.4 Field Documentation

class CEyeBotLoLevel * ZORC_RS232Stream::rs232EBLL [private]

The EyebotLoLevel object for accessing RS-232

Definition at line 76 of file ZORC_RS232Stream.h.

The documentation for this class was generated from the following file:

- ZORC_RS232Stream.h

D.2 ZorcBrain File Documentation

D.2.1 CEyeBotLoLevel.h File Reference

Data Structures

- class CEyeBotLoLevel
- struct tSensorTrait
- struct tServoCtrlTabEntry
- struct tServoTrait

Defines

- `#define MIN(x, y) (x<=y) ? x : y`
- `#define MAX(x, y) (x>=y) ? x : y`
- `#define kSerRemote SERIAL1`
- `#define kBaudRemote SER115200`
- `#define kSerSrvCtrl SERIAL3`
- `#define kBaudSrvCtrl SER9600`
- `#define PI 3.14159265358979323846`
- `#define RAD2GRAD(x) (x*180.0)/PI`
- `#define GRAD2RAD(y) (y*PI)/180.0`

Enumerations

- enum tLinkServoConsts { kRShouldSpread = 0, kLShouldRoll = 1, kLShouldSpread = 2, kRShouldRoll = 3, kRKnee = 4, kTorsoBend = 5, kLAnkle = 6, kLHipSpread = 7, kLHipRoll = 8, kRHipRoll = 9, kLKnee = 10, kRHipSpread = 11, kRAnkle = 12, kPowerSens = 13, kFootSensL = 14, kFootSensR = 15, kHeadSensL = 16, kHeadSensR = 17, kHeadSensF = 18, kHeadSensB = 19, kVirtual_RollSens = 98, kVirtual_PitchSens = 99 }
- enum tRobCtrlConsts { kServoCtrlQuantum = 2, kServoSlowSpeed = 50, kServoFastSpeed = 10, kServoNum = 13, kSensorNum = 20 }

D.2.1.1 Define Documentation

#define GRAD2RAD(y) (y*PI)/180.0

Definition at line 64 of file CEyeBotLoLevel.h.

#define MAX(x, y) (x>=y) ? x : y

Definition at line 37 of file CEyeBotLoLevel.h.

D Dokumentation der ZORCBRAIN-Klassen

#define MIN(x, y) (x<=y) ? x : y

Definition at line 36 of file CEyeBotLoLevel.h.

#define PI 3.14159265358979323846

Some shortcuts for fast conversion degrees <-> radians.

Definition at line 62 of file CEyeBotLoLevel.h.

#define RAD2GRAD(x) (x*180.0)/PI

Definition at line 63 of file CEyeBotLoLevel.h.

#define kBaudRemote SER115200

Baudrate for the serial interface connected to the workstation. Other possible values include SER9600, SER19200, SER38400, SER57600.

Definition at line 51 of file CEyeBotLoLevel.h.

#define kBaudSrvCtrl SER9600

Constant defining the baud rate for the ext. servocontroller

Definition at line 57 of file CEyeBotLoLevel.h.

#define kSerRemote SERIAL1

Port connected to the workstation.

Definition at line 45 of file CEyeBotLoLevel.h.

#define kSerSrvCtrl SERIAL3

Constant defining the Port for the external servo controller.

Definition at line 54 of file CEyeBotLoLevel.h.

D.2.1.2 Enumeration Type Documentation

enum tLinkServoConsts

Constants referring to servos / sensors. This is the Servo and Sensor-Layout for ZORC:

(inclination-sensors, 4x)	+0+	
left shoulder roll	/o---o\	right shoulder roll
left shoulder spread	o o	right shoulder spread
	\ /	
hip bend	o	
left hip roll	/o-o\	right hip roll
left hip spread	o o	right hip spread
left knee	o o	right knee
left ankle	+o= =o+	right ankle
(plus feet sensors, 2x)		

The following constants are used to identify the ports servos and sensors are connected to; e.g. the right-knee is controlled via servo 4, the sensor can thus be read via 'getSensorValue(4)'

Enumeration values:

kRShouldSpread Right shoulder-spread joint

kLShouldRoll Right shoulder-roll joint

kLShouldSpread Left shoulder-spread joint

kRShouldRoll Left shoulder-roll joint

kRKnee Right knee joint

kTorsoBend Torso joint

kLAnkle Left ankle joint

kLHipSpread Left hip-spread joint

kLHipRoll Left hip-roll joint

kRHipRoll Right hip-roll joint

kLKnee Left knee joint

kRHipSpread Right hip-spread joint

kRAnkle Right ankle joint

kPowerSens The power sensor measuring current flow.

kFootSensL Left foot switch

kFootSensR Right foot switch

kHeadSensL Inclination sensor to the left

kHeadSensR Inclination sensor to the right

kHeadSensF Inclination sensor to the front

kHeadSensB Inclination sensor to the back

kVirtual.RollSens The virtual sensor-ID to represent requests to the roll-sensor.

kVirtual.PitchSens The second virtual sensor-ID used to identify the pitch-sensor.

Definition at line 87 of file CEyeBotLoLevel.h.

D Dokumentation der ZORCBRAIN-Klassen

enum tRobCtrlConsts

Various numerical constants required for controlling the robot.

Enumeration values:

- kServoCtrlQuantum*** Frequency 100/x Hz of calls to the servo controller
- kServoSlowSpeed*** Slow-moving constant for setServo()
- kServoFastSpeed*** Fast-moving constant (the regular speed used).
- kServoNum*** Number of servos installed on ZORC. 0-9: internal controller, 9-12: external controller.
- kSensorNum*** Number of sensors on ZORC.

Definition at line 164 of file CEyeBotLoLevel.h.

D.2.2 CZorcBrain.h File Reference

Data Structures

- class CZorcBrain

D.2.3 ZORC_RS232Stream.h File Reference

Data Structures

- class ZORC_RS232Stream

Literaturverzeichnis

- [1] Krister Wolff and Peter Nordin. Evolution of Efficient Gait with Autonomous Biped Robot Using Visual Feedback. In Erik D. Goodman, editor, *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 482–489, San Francisco, California, USA, 9–11 July 2001.
- [2] Jens Ziegler, Krister Wolff, Peter Nordin, and Wolfgang Banzhaf. Constructing a Small Humanoid Walking Robot as a Platform for the Genetic Evolution of Walking. In *Autonomous Minirobots for Research and Edutainment, AMiRE2001. Proceedings of the 5th International Heinz Nixdorf Symposium*, 2001.
- [3] J. H. Holland. *Adaption in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. University of Michigan Press, 1975.
- [4] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
- [5] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, dpunkt.verlag, January 1998.
- [6] Projektgruppe 368. *Simulator für GP-evolierte Laufrobotersteuerungsprogramme*. Lehrstuhl für Systemanalyse, Fachbereich Informatik der Universität Dortmund, 2001. Endbericht der PG 368.
- [7] Projektgruppe 368. Webseite des Projektgruppe 368, <http://ls11-www.cs.uni-dortmund.de/~sigel/>. (24.06.2002).
- [8] Webseite von Hitec RCD USA, Inc., <http://www.hitecrcd.com/>. (24.06.2002).
- [9] Thomas Bräunl. Online-Dokumentation des EyeBot-Mikrocontrollerboards, <http://www.ee.uwa.edu.au/~braunl/eyebot/>. Dept. of Electrical & Electronic Engineering, The University of Western Australia, (24.06.2002).
- [10] M. Gienger, K. Löffler, and F. Pfeiffer. Design and Sensor System of a Biped Robot. In *4th International Conference on Climbing and Walking Robots, CLAWAR2001*, 2001.
- [11] Daniel Joseph Paluska. Design of a Humanoid Biped for Walking Research. Master’s thesis, Department of Mechanical Engineering, Massachusetts Institute of Technology, 2000.
- [12] Johann Borenstein, H.R. Everett, and Liqiang Feng. *Where am I? Sensors and Methods for Mobile Robot Positioning*. University of Michigan, 1996.
- [13] Inc. Scott Edwards Electronics. The MiniSSC-II Servocontroller, <http://www.seetron.com/>. (24.06.2002).
- [14] Peter Dittrich, Andreas Burgel, and Wolfgang Banzhaf. Learning to Move a Robot with Random Morphology. In Phil Husbands and Jean-Arcady Meyer, editors, *Proceedings of the First European Workshop on Evolutionary Robotics*, volume 1468 of *LNCS*, pages 165–178, Paris, 16–17 April 1998. Springer-Verlag.
- [15] Peter Nordin and Wolfgang Banzhaf. Genetic Programming Controlling a Miniature Robot. In E. V. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 61–67, MIT, Cambridge, MA, USA, 10–12 November 1995. AAAI.
- [16] D. Floreano and F. Mondada. Evolution of homing navigation in a real mobile robot, <http://citeseer.nj.nec.com/floreano96evolution.html>. (24.06.2002).
- [17] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sundaram. *PVM: Parallel Virtual Machine, A Users’ Guide and Tutorial for Networked Parallel Computing*, <http://www.netlib.org/pvm3/book/pvm-book.html>. MIT Press, 1994.

Literaturverzeichnis

- [18] Gary B. Parker and Jonathan W. Mills. Adaptive Hexapod Gait Control Using Anytime Learning with Fitness Biasing, <http://citeseer.nj.nec.com/213645.html>. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 519–524, Orlando, Florida, USA, 13-17 1999. Morgan Kaufmann.
- [19] Anton Hammer and Karl Hammer. *Sammlung physikalischer Formeln und Konstanten*. Lindauer Verlag, München, 1985.
- [20] SETI@home: Search for Extraterrestrial Intelligence at Home, <http://setiathome.ssl.berkeley.edu/>. (24.06.2002).
- [21] goopy.net - Verteilte Evolution von Schachprogrammen, <http://goopy.cs.uni-dortmund.de/>. (24.06.2002).
- [22] Webseite des GCC-Compilers (GNU Project), <http://gcc.gnu.org/>. (24.06.2002).
- [23] Bjarne Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley, 1998.
- [24] Rafael Gonzalez and Richard Woods. *Digital Image Processing*. Addison Wesley Publishing Company, 2nd edition edition, 2002.
- [25] Haberäcker. *Praxis der Digitalen Bildverarbeitung und Mustererkennung*. Carl HanserVerlag, München / Wien, 1995.
- [26] Webseite der Honda Roboter Asimo & P2, <http://www.honda-p3.com/>. (24.06.2002).
- [27] Dimitri van Heesch. Webseite des Doxygen Dokumentations Systems, <http://www.stack.nl/~dimitri/doxygen/>. (24.06.2002).

Hiermit versichere ich, dass ich diese Arbeit selbständig verfaßt
und keine anderen als die angegebenen Quellen und Hilfsmittel
benutzt sowie Zitate kenntlich gemacht habe.

Dortmund, den 26. Juli 2002