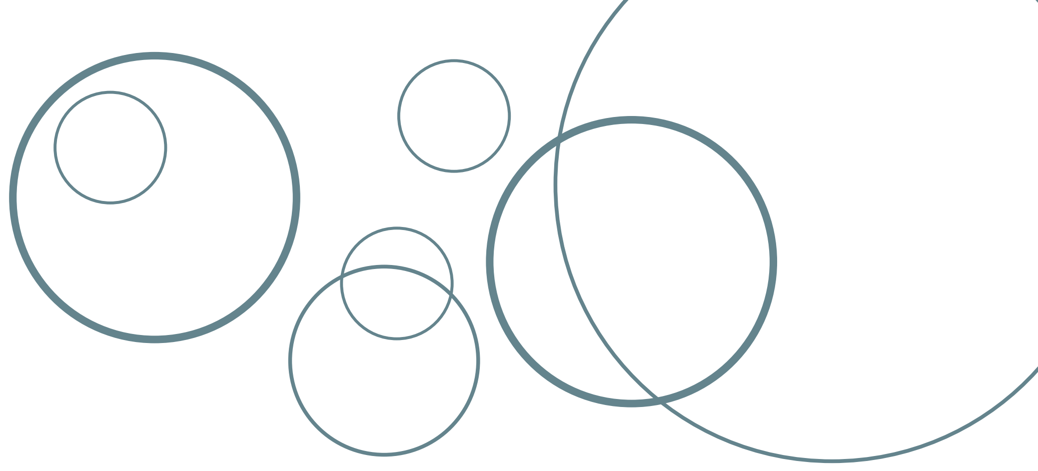


MARC SZYMANSKI

ENTWICKLUNGSUMGEBUNG FÜR ROBOTERSCHWÄRME

KIT Scientific Publishing

Marc Szymanski

Entwicklungsumgebung für Roboterschwärme

Entwicklungsumgebung für Roboterschwärme

von
Marc Szymanski

KIT Scientific Publishing

Dissertation, Karlsruher Institut für Technologie
Fakultät für Informatik
Tag der mündlichen Prüfung: 14. Dezember 2010

Impressum

Karlsruher Institut für Technologie (KIT)
KIT Scientific Publishing
Straße am Forum 2
D-76131 Karlsruhe
www.ksp.kit.edu

KIT – Universität des Landes Baden-Württemberg und nationales
Forschungszentrum in der Helmholtz-Gemeinschaft

Diese Veröffentlichung ist im Internet unter folgender Creative Commons-Lizenz
publiziert: http://creativecommons.org/licenses/by-nc-nd/3.0/de/

KIT Scientific Publishing 2011
Print on Demand

ISBN 978-3-86644-619-9

Entwicklungsumgebung für Roboterschwärme

Zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften
der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Dipl.-Inform. Marc Szymanski
aus Ratingen

Referent:	Prof. Dr.-Ing. Heinz Wörn
Koreferent:	Prof. Dr. rer. nat. habil. Paul Levi
Tag der Disputation:	14. Dezember 2010

Wehe! Es kommt die Zeit,
wo der Mensch nicht mehr den Pfeil
seiner Sehnsucht über den Menschen hinaus wirft,
und die Sehne seines Bogens verlernt hat, zu schwirren!

Also sprach Zarathustra,
Zarathustras Vorrede, S. 18.
Friedrich Nietzsche

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Prozessrechentechnik, Automation und Robotik (IPR) des Karlsruher Instituts für Technologie (KIT) in den EU-Projekten I-SWARM, Symbrion und Replicator.

Mein besonderer Dank gilt dem Leiter des Instituts, Herrn Prof. Dr.-Ing. Heinz Wörn für seine Unterstützung und Ermöglichung dieser Arbeit, sowie für das in mich gesetzte Vertrauen und die große Freiheit bei der Gestaltung meiner Projekte.

Darüber hinaus möchte ich Herrn Prof. Dr. rer. nat. habil. Paul Levi für sein Interesse an meiner Arbeit und die Übernahme des Korreferats, sowie für die langjährige und fruchtbare Zusammenarbeit in diversen Projekten danken.

Allen Mitarbeitern des IPR danke ich herzlich für das freundliche Arbeitsklima und die große Hilfsbereitschaft, die zum Gelingen dieser Arbeit beigetragen hat. Besonders bedanken möchte ich mich bei Alexander Kettler für die fruchtbare Zusammenarbeit bei der Entwicklung des Wanda-Roboters, und Jens Liedke und Dirk Göger für die stets hilfreiche Unterstützung bei mechanischen und elektrotechnischen Fragestellungen. Ebenfalls danke ich Lutz Winkler und Rene Matthias für die erfolgreiche Zusammenarbeit in den verschiedenen EU-Projekten, die wir zusammen bestritten ha-

ben. Den langjährigen Mitarbeitern Margit Pfitzer, Frank Linder und Hartmut Regner möchte ich für die schnelle Umsetzung meiner Idee ausdrücklich danken.

Ebenfalls zu erwähnen sind die wertvollen Beiträge zahlreicher Studenten, die im Rahmen von Diplo- und Studienarbeiten an dieser Arbeit mitgewirkt haben. Ein besonderer Dank gilt hier Mathias Lüdtke, der mir mit sehr großem Engagement bei allen Unwägbarkeiten mit dem Jasmine- und Wanda-Roboter tatkräftig zur Seite gestanden hat.

Stellvertretend für die zahlreichen internationalen Partnern, mit denen ich während meiner Tätigkeit in dem I-SWARM-, Symbrion- und Replicator-Projekt zusammenarbeiten durfte, möchte ich Serge Kernbach danken, der mich stets als Freund durch diese Projekte hindurch begleitet hat.

Besonders herzlich möchte ich meinem Vater für die Korrektur meiner Arbeit, und bei meiner Mutter und meinem Vater für ihre Unterstützung in allen Belangen und für das uneingeschränkte Vertrauen in meinen Weg bedanken.

Ein ganz besonderer Dank geht an Sarah Wehebrink, die mir eine so wunderbare Tochter geschenkt hat.

Karlsruhe, im Dezember 2010 Marc Szymanski

für Lina

Inhaltsverzeichnis

1 Einführung **1**

1.1 Motivation - Vereinfachte Handhabung von Miniatur- und Mikroroboterschwärmen . . . 1

1.2 Zielsetzung . . . 3

1.2.1 Begriffserklärung . . . 3

1.2.2 Thesen . . . 9

1.2.3 Zielsetzung im Detail . . . 9

1.3 Methodik . . . 11

1.4 Gliederung der Arbeit . . . 12

2 Verwandte Arbeiten **13**

2.1 Miniaturroboter . . . 14

2.1.1 Khepera . . . 14

2.1.2 Alice . . . 15

2.1.3 E-Puck . . . 16

2.2 Mikroroboter . . . 18

2.2.1 MINIMAN III . . . 18

2.2.2 MiCRoN . . . 18

2.3 Entwicklungsumgebungen und Softwarearchitekturen . . . 19

2.4 Schwarmrobotik Projekte 23
2.4.1 Swarm-Bots und Swarmanoid 23
2.4.2 Symbrion und Replicator 24
2.5 Zusammenfassung 25

3 Mikro- und Miniaturroboter 27
3.1 I-SWARM 28
3.1.1 Prozessor 29
3.1.2 Sensorik 31
3.1.3 Aktuatorik 31
3.1.4 Besonderheiten 31
3.2 Jasmine 33
3.2.1 Prozessor 33
3.2.2 Sensorik 36
3.2.3 Aktuatorik 36
3.2.4 Besonderheit 36
3.3 Wanda 37
3.3.1 Aufbau 40
3.3.2 Prozessor 42
3.3.3 Sensorik 42
3.3.4 Aktuatorik 45
3.3.5 Besonderheit 45
3.4 Zusammenfassung 45

4 Entwicklungsumgebung 47
4.1 Anforderungsanalyse für die Steuersoftware 48
4.1.1 Anforderungen aus der Darstellung von Algorithmen 48
4.1.2 Anforderungen durch Evolutionary Computation . . 49
4.1.3 Anforderungen aus der Hardware 50
4.1.4 Zusammenfassung und Analyse der Anforderungen 51
4.2 Die extended Motion Description Language 2 54
4.2.1 Formale Beschreibung der MDL2ϵ 55
4.2.2 MDL2ϵ in der eXtensible Markup Language 64
4.2.3 Entwicklung der MDL2ϵ Software 68
4.2.4 MDL2ϵ Ausführungsstapel 70
4.2.5 Darstellung von MDL2ϵ als C- und als Bytecode . . 72
4.2.6 MDL2ϵ in der Robotersteuerung 76
4.3 Simulationsumgebung 82
4.3.1 Integration von Breve in die MDL2ϵ-Applikation . . 83

4.3.2 Integration von Stage mit MDL2ϵ 85
4.4 Interaktive Dynamische Arena 85
4.4.1 Konzept 86
4.4.2 ODeM 91
4.4.3 Arena-Steuerung 92
4.4.4 Drahtlose Kommunikation basierend auf ZigBee Hardware 93
4.5 Zusammenfassung 94

5 Genetische Programmierung 97
5.1 Softwarearchitektur 99
5.1.1 Anbindung von GP an MDL2ϵ 100
5.1.2 Verteilte Master-Slave GP Architektur 100
5.2 Initiale Population 104
5.3 Genetische Operatoren 106
5.3.1 Reproduktion 106
5.3.2 Mutation 106
5.3.3 Crossover 107
5.4 Selektionsstrategien 108
5.4.1 Fitnessproportionale Selektion 109
5.4.2 Rangbasierte Selektion 109
5.4.3 Turnierselektion 110
5.5 Pruning 110
5.6 Diversität von MDL2ϵ-Plänen 111
5.6.1 Paarweise Ähnlichkeit 111
5.6.2 Ähnlichkeit der Population 115
5.6.3 Diversität der Population 116
5.7 Zusammenfassung 116

6 Experimente und Bewertung 117
6.1 Entwicklungsumgebung 117
6.1.1 Stigmergie 118
6.1.2 Sammeln von Ressourcen unter energetischen Gesichtspunkten 125
6.2 Genetische Programmierung 145
6.2.1 Kollisionsvermeidung 146
6.2.2 Aggregation 158
6.2.3 Experimente mit Wanda 167
6.2.4 Bewertung 170

6.3 Zusammenfassung 172

7 Zusammenfassung und Ausblick 175

7.1 Ergebnis und Beitrag der Arbeit 176

7.2 Ausblick 177

A MDL2ϵ XML Beschreibung in DTD 179

B MDL2ϵ-Pläne zu den durchgeführten Experimenten 183

B.1 Stigmergie Experiment 183

B.1.1 Zufällige Suche von Ressourcen 183

B.1.2 Pheromonbasierte Suche von Ressourcen 186

B.2 Sammeln von Ressourcen unter energetischen Gesichtspunkten 190

Akronyme 201

Literaturverzeichnis 204

1. Einführung

1.1 Motivation - Vereinfachte Handhabung von Miniatur- und Mikroroboterschwärmen

Das *Institut für Prozessrechentechnik, Automation und Robotik (IPR)* am *Karlsruher Institut für Technologie (KIT)* beschäftigt sich seit 1993 mit der Entwicklung mobiler Mikroroboter zur Handhabung von Mikroobjekten sowie zur Mikromontage. Basierend auf diese langjährigen Erfahrung entstand am IPR, zusammen mit anderen europäischen Partnerinstituten, die Idee, die Mikrorobotik mit der Schwarmrobotik in dem *Intelligent Small World Autonomous Robots for Micro Manipulation (I-SWARM)*-Projekt zu vereinen. Dies erscheint sinnvoll, da beide Bereiche stark von einander profitieren können.

Mit zunehmender Miniaturisierung von Robotern werden die Fähigkeiten des einzelnen Roboters immer stärker eingeschränkt. Dies betrifft zum einen die Manipulation von Objekten sowie die auf den Robotern vorhandene Rechenleistung. Durch die Kombination der Mikrorobotik mit der Schwarmrobotik können die durch die Miniaturisierung des Roboters entstehenden Probleme teilweise egalisiert werden. So basieren die meisten Algorithmen in der Schwarmrobotik auf sehr einfachen reaktiven Verhal-

ten der Roboter, die nicht viel Rechenleistung beanspruchen, die aber in der Summe des Kollektives zu dem gewünschten Ergebnis führen. Des Weiteren ist die Redundanz, Skalierbarkeit und Robustheit der Algorithmen aus der Schwarmrobotik wichtig, da durch die Miniaturisierung der Roboter die Fehleranfälligkeit des Roboters zunimmt, was zu einem Ausfall von Robotern führen kann.

Die Schwarmrobotik hingegen erfährt durch die Miniaturisierung der Roboter eine Reduktion der Kosten des einzelnen Roboters, da Methoden der Massenfabrikation verwendet werden können. Dies führt dazu, dass die betrachtete Größe von Roboterschwärmen von 10–30 Individuen auf bis zu 1.000 Roboter stark ansteigt. Deshalb war ein wichtiges Anliegen des I-SWARM-Projektes die Entwicklung eines Roboters möglichst auf einem Wafer.

Um solche großen Schwärme zu entwickeln und zu untersuchen, ist es notwendig einen Rahmen zu schaffen, der dies auch zulässt. Können Experimente mit kleinen Schwärmen von bis zu zehn Robotern noch von Hand programmiert und überwacht werden, so stellen größere Schwärme ein deutliches logistisches Problem dar. Die Handhabung solcher Roboterschwärme verlangt eine Entwicklungsumgebung, die von Grund auf an die Bedürfnisse solcher großer Schwärme angepasst ist. Dies zieht sich von der Steuerung des einzelnen Roboters, über die Simulation bis hin zur verwendeten Arena für Experimente durch.

Im Besonderen ist es gerade für stark miniaturisierte Systeme, die sich mit herkömmlichen Softwarewerkzeugen zur Fehlerbeseitigung nicht in den Griff bekommen lassen, wichtig, dass die entsprechende Software auf einem anderen System ohne große Veränderungen verifiziert und korrigiert werden kann. Dies führt zu einer schnelleren und stabileren Softwareentwicklung.

Ein weiteres Problem ist die Implementierung und Optimierung von Robotersteuerungen, welche in der Regel sehr zeitaufwändig ist. Ein Traum jedes Ingenieurs ist die automatische Erzeugung und Optimierung eines Satzes von Teilprogrammen, die er beliebig in seine eigene Steuerung einbauen kann.

Aus diesen Überlegungen entstand die Idee einer Entwicklungsumgebung für Schwarmroboter. Die Entwicklungsumgebung sollte sowohl plattformunabhängig in Bezug auf die Roboter sein und die Bedürfnisse für Experimente mit größeren Roboterschwärmen erfüllen, als auch die Möglichkeit für die automatische Erzeugung von Unterprogrammen für Robotersteuerungen bieten.

Ein solches System wurde im Rahmen der vorliegenden Dissertation entwickelt, implementiert und auf drei verschiedenen Robotersystemen anhand von beispielhaften Experimenten evaluiert.

1.2 Zielsetzung

Die Arbeit verfolgt zwei wichtige Ziele. Ein Ziel der Arbeit ist es, ein System zu entwickeln, das zum einen eine plattformunabhängige Programmierung von Mikro- und Miniaturschwarmroboter ermöglicht, und zum anderen eine Umgebung zu Verfügung stellt, mit der Experimente mit größeren Roboterschwärmen durchgeführt werden können. Hierdurch soll der Aufwand für die Entwicklung von Schwarmrobotersystemen verringert werden. Ein weiteres Ziel der Arbeit ist die Entwicklung eines halbautomatischen Systems zur Erzeugung von Programmen zur Robotersteuerung mit Hilfe von Methoden der *Genetischen Programmierung (GP)*.

1.2.1 Begriffserklärung

Zur Präzisierung der Zielsetzung, soll zunächst die Verwendung bestimmter Begriffe genau bestimmt werden.

Mikrorobotik

Mikroroboter erweitern das bekannte Spektrum der Robotik durch Roboter, die zum einen klein sind und speziell für die Manipulation im μm-Bereich konstruiert worden sind. Laut Fatikow [32] bezieht sich die Definition des Mikroroboters auf die Eigenschaften von Robotern der Makrowelt:

> *Ein Mikroroboter ist ein durch programmierbares Verhalten, eine aufgabenspezifische Sensor- und Aktuatorausstattung und eine in der Regel uneingeschränkten Mobilität gekennzeichnet. Wie konventionelle Roboter besteht auch ein Mikrorobotersystem aus zwei im allgemeinen autonomen Teilsystemen, die das gesamte Bewegungspotential des Mikroroboters bestimmen: Aktuatoren zur Mikromanipulation mit Objekten ("Roboterarme und -hände") und Aktuatoren zum Transportieren der Roboterplattform ("Roboterbeine"). Das erste Teilsystem bestimmt dabei die Manipulationsfähigkeit und das zweite die Mobilität des Mikroroboters.*

Fatikow klassifiziert Mikroroboter nach ihrer Größe, ihrer Funktionalität und ihrer Aufgabe. Er unterscheidet somit nach der Größe zwischen Miniatur- und Mikrorobotern. Miniaturroboter haben eine Größe von einigen Kubikzentimetern und sind aus konventionellen miniaturisierten Komponenten zusammengesetzt. Mikroroboter hingegen sind einige μm^3 groß und bestehen aus eigens in *Mikrosystemtechnik (MST)*-Verfahren hergestellten, an den Roboter angepassten Komponenten.

Seyfried erweitert in [100] diese Definition und lässt auch Miniaturroboter als Mikroroboter zu, die im Milli-/Nanometerbereich mit eine Wiederholgenauigkeit von $10\mu m$ und besser manipulieren können.

Funktional unterscheidet er anhand ihrer Mobilität (vorhanden/nicht vorhanden), Autonomie (Energiequelle onboard/offboard) und Ansteuerungsart (kabellos/mit Kabel). Dabei führt er keine Klasse für die Autonomie der Roboter bezüglich ihrer Steuerung ein. Dies ist aber für den Begriff der Mikroschwarmrobotik sehr von Bedeutung. Daher erweitern wir Seyfrieds Klassifikation durch die Art der Steuerung (zentral gesteuert/autonom).

Die aufgabenspezifische Klassifikation von Mikrorobotern betrachtet das Verhältnis C zwischen den physikalischen Abmessungen des Roboters und seinem erzielbaren Bewegungsbereich. Hierbei gibt es auf der einen Seite der Klassifikationsskale $C \gg 1$ die stationären Mikromanipulationssysteme, welche zwar relativ groß sind aber sehr präzise im Millimeter- und Nanometer-Bereich manipulieren können. Auf der anderen Seite der Skala $C \ll 1$ stehen mikroskopisch kleine Mikroroboter. Dazwischen stehen mit $C \approx 1$ Robotersysteme, die in ungefähr mit den Abmessungen ihres Arbeitsbereichs übereinstimmen.

Vergleicht man die Mikro- mit der Makrorobotik, so scheint der Unterschied nicht sonderlich groß zu sein. Bei zunehmender Miniaturisierung verändert sich jedoch das physikalische Kräfteverhältnis, das für den Betrieb des Roboters und die Manipulation von Objekten entscheidend ist. So sinkt der Einfluss der Gravitation kubisch mit der Größe, die Oberflächenkräfte wie Adhäsion und Kohäsion sinken jedoch nur quadratisch. Somit ist es in der Mikrorobotik oftmals ein größeres Problem, ein gegriffenes Objekt loszuwerden, da es an dem Greifer haftet, als es zu greifen.

In Abbildungen 1.1 sieht man Beispiele für Mikroroboter am IPR. Ihre Klassifikation ist der Bildunterschrift zu entnehmen.

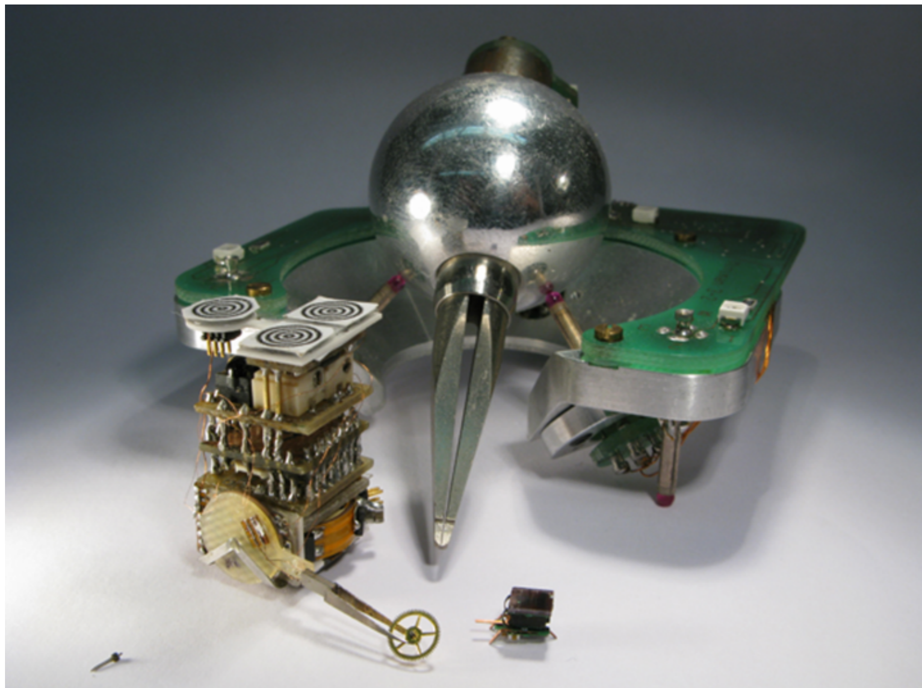


Abbildung 1.1: MINIMAN III (1989-2002) Klassifikation nach Fatikow: mobil, Energie offboard, mit Kabel, zentral gesteuert, $C \approx 1$ (*Mitte*); MiCRoN (2002-2004) Klassifik.: mobil, energieautonom, kabellos, zentral gesteuert, $C \approx 1$ (*links vorne*); I-SWARM (2004-2008)Klassifik.: mobil, energieautonom, kabellos, autonom, $C \approx 1$ (*rechts vorne*)

Schwarmrobotik

Die *Schwarmrobotik* wurde 1989 von Gerado Beni und Jing Wang in [4] als *Cellular Robotics* eingeführt[1]. Sie ist ein Ansatz, um eine große Anzahl von autonomen Robotern koordiniert zu steuern. Sie ist eine Unterart der Kollektiven Robotik, wobei dieser Ansatz inspiriert ist durch das Verhalten eusozialer Insekten wie Ameisen, Termiten, Wespen und Bienen, aber auch durch das Verhalten von Vogel-/Fischschwärmen sowie Herden von Säugetieren. Es ist erstaunlich zu beobachten, dass ein Schwarm von Insekten weit mehr erreichen kann als das einzelne Individuum.

Die Schwarmrobotik zieht ihre Motivation aus den Eigenschaften, die für *Multi-Roboter-Systeme (MRS)* wünschenswert wären und die man in biologischen Schwärmen beobachten kann. Diese Eigenschaften sind:

- **Robustheit**
  Trotz des fehlerhaften Verhaltens einzelner Individuen sollte es einem Schwarm möglich sein, die gegebenen Aufgabe weiterhin zu erfüllen.

- **Flexibilität**
  Das Schwarmrobotersystem soll fähig sein, modulare Lösungen für verschieden Aufgaben zu erfüllen, ohne die Konfiguration des Schwarms zu verändern.

- **Skalierbarkeit**
  Schwarmalgorithmen sollten möglichst unabhängig von der Anzahl der Individuen funktionieren.

Erol Sahin definiert in [94] die Schwarmrobotik als:

> *Swarm robotics is the study of how large numbers of relatively simple physically embodied agents can be designed such that a desired collective behavior emerges from the local interactions among agents and between the agents and the environment.*
>
> *(Die Schwarmrobotik untersucht, wie eine große Anzahl an relativ einfachen, physikalisch eingebetteten Agenten entworfen werden kann, so dass sich das erwünschte, kollektive Verhalten aus den lokalen Interaktionen zwischen den Agenten und den Agenten und der Umwelt herausbildet.)*

[1] Zeitgleich kreierten sie auch den Begriff der *Schwarmintelligenz*.

Um seine Definition zu spezifizieren und sich von verschieden Arten der Multi-Roboter Forschung zu unterscheiden, führt er weiter Kriterien auf, die ein Schwarmrobotersystem erfüllen sollte:

- **Autonome Roboter**
  Jeder Roboter im Schwarm sollte autonom sein. Eine zentrale Steuerung vieler Roboter ist kein Schwarm.

- **Große Anzahl an Roboter**
  Die Anzahl der Roboter sollte sich signifikant von einer Gruppe von Robotern unterscheiden. Sahin gibt hier eine Größe von 10-20 Robotern an, lässt aber auch weniger zu, wenn die Skalierbarkeit nicht ausser Acht gelassen wird.

- **Wenige homogene Gruppen von Robotern**
  Das untersuchte MRS sollte aus einer relativ geringen Anzahl verschiedener Gruppen ansonsten gleichförmiger Roboter bestehen. Die Anzahl der Roboter in einer Gruppe sollte relativ groß sein.

- **Relativ unfähige oder ineffiziente Roboter**
  Der einzelne Roboter sollte im Verhältnis zur Aufgabe entweder nicht fähig sein, diese alleine zu bewerkstelligen oder aber diese alleine nur sehr ineffizient ausführen können.

- **Roboter mit lokaler Wahrnehmung und Kommunikation**
  Die Sensorik der Roboter sollte lokal beschränkt sein und sich auf die direkte (lokale) Umgebung des Roboters beziehen.

Robotersteuerung

Eine Robotersteuerung ist ein System zum Steuern eines Roboters. Dabei ist der Begriff Steuerung gemäß DIN 19 226 Teil 1 wie folgt definiert:

> *Das Steuern, die Steuerung ist ein Vorgang in einem System, bei dem eine oder mehrere Größen als Eingangsgrößen andere Größen als Ausgangsgrößen aufgrund der dem System eigentümlichen Gesetzmäßigkeiten beeinflussen.*
>
> *Kennzeichen für das Steuern ist der offene Wirkungsweg oder ein geschlossener Wirkungsweg, bei dem die durch die Eingangsgrößen beeinflussten Ausgangsgrößen nicht fortlaufend und nicht wieder über dieselben Eingangsgrößen auf sich selbst wirken.*

Eine Roboterregelung unterscheidet sich somit von einer Robotersteuerung durch das Vorhandensein einer Stell- bzw. Regelgröße. Der Begriff der Regelung ist in DIN 19 226 Teil 1 wie folgt definiert:

> *Das Regeln, die Regelung ist ein Vorgang, bei dem fortlaufend eine Größe, die Regelgröße (die zu regelnde Größe), erfasst, mit einer anderen Größe, der Führungsgröße, verglichen und im Sinne einer Angleichung an die Führungsgröße beeinflusst wird. Kennzeichen für das Regeln ist der geschlossene Wirkungsablauf, bei dem die Regelgröße im Wirkungsweg des Regelkreises fortlaufend sich selbst beeinflusst.*

Genetische Programmierung

Die GP ist eine Art der *Evolutionary Computation (EC)*, bei der eine Menge an Lösungen (Population) generiert wird. Die einzelne Lösung bezeichnet man dabei als Individuum und deren Darstellung als Genom. Diese Lösungen werden anhand einer Metrik bzw. Fitnessfunktion bewertet. Auf dieser Bewertung findet eine Selektion von Lösungen statt, wobei mit höherer Wahrscheinlichkeit die besonders guten Lösungen selektiert werden. Aus den selektierten Lösungen wird eine neue Population durch Kreuzung und Mutation erzeugt. Dies wird wiederholt, bis die Güte der gefundenen Lösungen sich nicht mehr deutlich verbessert, oder ausreichend ist.

GP unterscheidet sich zu anderen Algorithmen der EC durch die Darstellung des verwendeten Genoms als Programm. Andere Spielarten von Genomen, die nicht der GP zufallen, sind z.B. die Gewichte eines *Künstlichen Neuronalen Netzes (KNN)* oder die Variablenbelegung einer mehrdimensionalen Funktion.

Simulationsumgebung

Simulationsumgebungen sind ein wichtiger Bestandteil in der Robotik. Besonders bei der Verwendung von Gruppen/Schwärmen autonomer Roboter beschleunigt eine Simulation die Entwicklung und hilft Fehler in Algorithmen frühzeitig zu erkennen. Eine Simulation ist nach VDI 3633 wie folgt definiert:

> *Simulation ist das Nachbilden eines dynamischen Prozesses in einem System mit Hilfe eines experimentierfähigen Modells, um*

zu Erkenntnissen zu gelangen, die auf die Wirklichkeit übertragbar sind.

Anhand der Definition, welche den Begriff eines "experimentierfähigen Modells" aufgreift, erkennt man auch die klassischen Probleme einer Simulation. Die Güte des Modells ist ausschlaggebend für die Güte der Simulation. Die Güte wird hier an der Übertragbarkeit auf die Wirklichkeit gemessen. Gerade in der Robotik hat man hier Kompromisse einzugehen, da ein sehr genaues Modell die Rechenzeit erhöht und ein lineares Anwachsen im Detaillevel zu einem exponentiellen Wachstum der Zeit für einen Simulationsschritt führen kann. Gerade bei MRS kennt man das Problem, dass das Verhältnis zwischen der simulierten Zeit t_S und der benötigten Rechenzeit t_R, bei anwachsendem Schwarm und hinreichendem Detaillevel, schnell kleiner als eins ist ($\frac{t_S}{t_R} < 1$). Dies bedeutet, dass das Modell langsamer läuft als der reale Roboterschwarm. Simulationsumgebungen ersetzen daher keine Experimente auf echten Robotern.

1.2.2 Thesen

Behauptungen, die in dieser Arbeit belegt werden sollen sind:

- *Eine Entwicklungsumgebung für Roboterschwärme beschleunigt die Entwicklung und macht unabhängiger von Ergebnissen, die durch Simulationen gewonnen werden.*
- *Methoden der GP können eingesetzt werden, um Teile einer Schwarmrobotersteuerung halbautomatisch zu erzeugen, und ermöglichen im Gegensatz zu EC basierend auf KNNs eine einfachere Analyse der gefundenen Lösung.*

1.2.3 Zielsetzung im Detail

Um eine Entwicklungsumgebung für Miniatur-/Mikroschwarmroboter zu entwickeln, werden folgende Teilziele verfolgt:

- **Analyse der Anforderungen einer Schwarmrobotersteuerung**
  Es wird analysiert, welche Anforderungen an die Steuerung für Schwarmroboter gestellt werden. Hierbei bezieht sich die Analyse auf die zu Verfügung stehenden Schwarmroboter und die Anforderungen, die sich aus dieser Hardware ergeben. Durch die Auswahl

der Hardware wird ein großes Spektrum von verschieden Miniatur-/Mikrorobotersystemen gewährleistet. Des Weiteren soll die Analyse Anforderungen miteinbeziehen, die sich aus der Schwarmrobotik und der in der Schwarmrobotik verwendeten Algorithmen ergeben. Ein weiterer Punkt in der Analyse ist die Verwendbarkeit für Methoden der GP.

- **Definition einer allgemeinen Steuersprache**
  Nach der Analyse der Anforderungen ist es ein Ziel, eine geeignete Steuersprache zu entwickeln, die möglichst plattformunabhängig auf dem Roboter sowie auch in einer Simulationsumgebung eingesetzt werden kann. Dies beinhaltet auch die Implementierung verschiedener Roboterbetriebssysteme und die Einbindung der Steuersprache in diese zur Steuerung des Roboters.

- **Entwicklung einer interaktiven Arena für Schwarmroboterexperimente**
  Ziel ist die Entwicklung einer Arena, die im Besonderen auf Probleme eingeht, die sich zum einen durch große Roboterschwärme als auch durch die Verwendung von Miniatur-/Mikroschwarmrobotern ergeben. Diese Arena geht daher auf die Informationsakquise während der Durchführung eines Experimentes, als auch auf die Erweiterung der Robotersensoren durch einen aktiven Teil der Arena ein. Ein besonderes Augenmerk liegt auf der Reproduzierbarkeit von Experimenten und das Einbringen von dynamischen Veränderungen in die Umgebung.

- **Entwicklung eines Rahmenwerks für die Genetische Programmierung**
  Ziel ist die Implementierung und Evaluierung eines Rahmenwerks für GP. Mit Hilfe der GP können Teile der Robotersteuerung halbautomatisch erzeugt werden.

- **Modellierung der Roboter in einer Simulationsumgebung**
  Zum Testen der implementierten Software und als Basis für die GP ist die Modellierung eines Roboters in einer Simulationsumgebung unabdingbar. Hierbei werden verschiedene Simulationsumgebungen auf ihre Handhabbarkeit getestet und eine ausgewählt.

- **Evaluation des entwickelten Systems anhand ausgewählter Algorithmen**
  Ziel ist es, das gesamte System anhand von ausgewählten Algorithmen zu testen. Hierbei werden alle Teile des Systems einbezogen und evaluiert. Die Evaluation findet soweit möglich auf verschiedenen Robotertypen statt.

1.3 Methodik

Die in der Zielsetzung in Abschnitt 1.2.3 aufgegliederten Teilziele folgen einer Methodik, die in Abbildung 1.2 dargestellt ist. Kernpunkt der Methodik ist die Evaluation der Teilziele durch entsprechende Experimente zum einen in der Simulation und zum anderen auf den realen Robotern in der Arena. Durch dieses methodische Vorgehen ist gewährleistet, dass die Teilergebnisse funktionieren und gegebenenfalls anhand der Ergebnisse der Experimente angepasst und verbessert werden können. Eine nebenläufige Hardware- und Softwareentwicklung hilft frühzeitig Fehler zu erkennen und zu beheben.

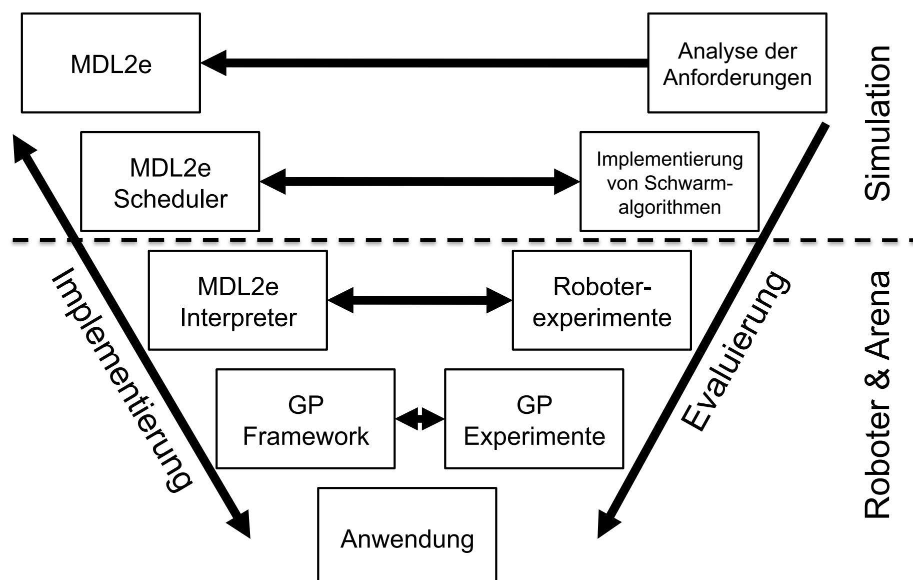


Abbildung 1.2: Die während der Arbeit angewendete Methodik.

1.4 Gliederung der Arbeit

In Kapitel 2 *Verwandte Arbeiten* werden zunächst bestehende Miniatur- und Mikroroboter und ihren Verwendung in der Schwarmrobotik beschrieben.

Kapitel 3 beschreibt die Hardware der verwendeten Roboter I-SWARM, Jasmine und Wanda.

In Kapitel 4 *Entwicklungsumgebung* werden die wichtigsten Teile der Entwicklungsumgebung aufgezeigt. Zuerst wird eine Anforderungsanalyse durchgeführt, an die sich die Beschreibung einer Steuersprache und deren Implementierung auf den Robotern angliedert.

Kapitel 5 *Genetische Programmierung* beschreibt das für die Evolution von Programmen zur Robotersteuerung entwickelte Framework for Learning and Self-Organisation.

Die in den Kapiteln 4 – 5 beschriebenen Systeme werden anhand ausgewählter Experimente in Kapitel 6 *Experimente und Bewertung* evaluiert. Zuerst wird auf die in Kapitel 4 beschriebene Entwicklungsumgebung und dann auf das in Kapitel 5 aufgezeigte *Framework for Learning and Self-Organisation (FLSO)* eingegangen.

Kapitel 7 gibt eine kurze Zusammenfassung der Arbeit, zeigt die wichtigsten Ergebnisse und Beiträge und schließt mit einem Ausblick für weiter Forschung basierend auf dem in der vorliegenden Arbeit diskutierten System.

2. Verwandte Arbeiten

In diesem Kapitel werden verwandte Arbeiten in den Bereichen Miniatur- und Mikroschwarmrobotik sowie verschiedenen Entwicklungsumgebungen dargestellt. Zunächst wird auf die wichtigsten bestehenden Roboter in den Bereichen Miniatur- und Mikroschwarmrobotik eingegangen. Hierbei liegt ein Fokus auf Experimente, die im Bereich der Evolution von Robotersteuerungen auf den beschriebenen Robotern durchgeführt worden sind, sowie auf den für den Roboter entwickelten System zur Überwachung von Experimenten.

Im Bereich der Schwarmrobotik werden viele Miniaturroboter eingesetzt, die zum Teil als Mikroroboter bezeichnet werden, die aber nicht der von Fatikow und Seyfried aufgestellten Einordnung als Mikroroboter standhalten und somit als Miniaturroboter zu bezeichnen sind. Ein Beispiel hierfür ist der Alice-Roboter der in Abschnitt 2.1 neben weiteren Miniaturrobotern beschrieben wird, die häufig in der Schwarmrobotik zur Anwendung kommen.

Da es eigentlich neben dem I-SWARM-Projekt keine weiteren echten Schwarmroboter-Projekte im Bereich der Mikrorobotik gibt, die der Einordnung durch Fatikow als Mikroroboter standhalten, wird in Abschnitt 2.2 eine Übersicht über bestehende Mikroroboter gegeben, die die direkten Vorgänger des I-SWARM-Roboters sind.

In Abschnitt 2.3 wird eine Übersicht über bestehende Entwicklungsumgebungen und Softwarearchitekturen gegeben, die in der mobilen und teilweise auch in der Kollektiven Robotik eingesetzt werden.

Im letzen Abschnitt 2.4 werden aktuelle Projekte in dem Bereich der Schwarmrobotik und Kollektiven Robotik beschrieben.

2.1 Miniaturroboter

2.1.1 Khepera

Der Khepera [80] ist ein kleiner (Durchmesser 55 mm) autonomer Roboter mit differentiellem Antrieb. Die Entwicklung des Khepera begann 1991 am *Microcomputing Laboratory (LAMI)* an der *Ecole Polytechnique Federale de Lausanne (EPFL)* (Lausanne, Schweiz) von Edo. Franzi, Francesco Mondada und André Guignard unter der Leitung von Prof. Jean-Daniel Nicoud. Das Design basiert auf einem Motorola 68331 mit 16 MHz Taktung 256 KB RAM und 512 KB EEPROM. Ausgestattet ist der Khepera-Roboter mit 8 *Infrarotlicht (IR)*-Näherungssensoren und einem Umgebungslicht-Sensor. Die durchschnittliche Fahrzeit beträgt ca. 45 Minuten. Für den Roboter waren Erweiterungen wie Greifer, Kamera und Funk erhältlich.

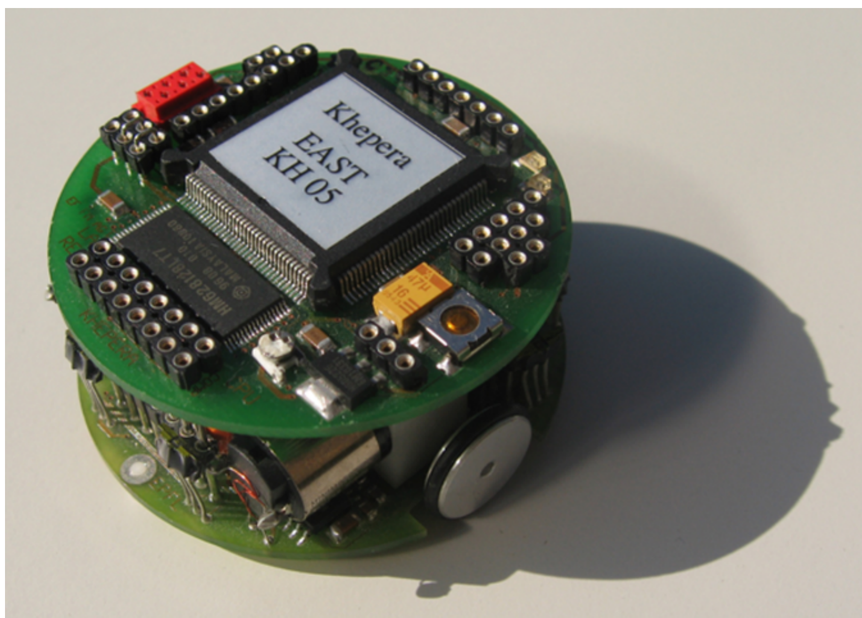


Abbildung 2.1: Erste Generation des Khepera von 1996; mit freundlicher Genehmigung von Stéphane Magnenat.

Obwohl der Khepera-Roboter nicht als Schwarmroboter entwickelt worden war, wurde er in vielen Experimenten zur Schwarmrobotik eingesetzt [73]. Grund dafür war, dass er über viele Jahre einer der wenigen kommerziell erhältlichen Roboter im unteren Preissegment war und dadurch interessant für die Schwarmrobotikforschung wurde. Der Khepera-Roboter ist mittlerweile in der dritten Generation erhältlich.

Floreano et al. [34, 86, 79, 37, 36, 35, 85] haben mit dem Khepera Versuche im Bereich der EC mit einzelnen und mehreren Khepera Robotern durchgeführt. Hierfür waren die Roboter über ein Kabel mit einem Steuerrechner verbunden, der die Evolution der zur Steuerung verwendeten KNNs übernahm. Zur Bestimmung der Positionen für die Analyse der evolvierten Verhalten wurde ein laserbasiertes System verwendet. Versuche zur Evolution von Ausweichverhalten mit Hilfe von GP basierend auf einem einzelnen Khepera Roboter werden von Nordin und Banzhaf [88, 87] beschrieben.

Fricke et al. [39] beschreiben eine Arena für Experimente mit Schwärmen von Khepera-II-Robotern. Sie beschreiben ein System basierend auf zwei SICK LADAR Sensoren, mit denen die Position der Roboter während eines Experiments verfolgt werden kann. Über eine drahtlose Schnittstelle können in C implementierte Programme direkt auf den Roboter übertragen werde.

2.1.2 Alice

Der Alice-Roboter [21, 18, 20, 19] ist ein sehr kleiner nur $2 \times 2 \times 2cm^3$ großer Miniaturroboter mit differentiellem Antrieb. Er wurde am *Autonomous Systems Lab (ASL)* der EPFL zwischen 1998 und 2004 entwickelt. Ausgestattet ist der Roboter mit einem PIC16F877 *Microcontroller (μC)*, vier IR-Näherungssensoren und einem Fernbedienungsempfänger für omnidirektionale Kommunikation. Seine Fahrzeit beträgt bis zu zehn Stunden. Es wurden Erweiterungen wie Kamera, bidirektionaler Funk, taktile Sensoren und ZigBee für Alice entwickelt. Ziel der Entwicklung war ein möglichst kleiner und kostengünstiger Roboter für Forschung im Bereich der Schwarmrobotik.

Der Alice-Roboter wurde in einigen Schwarmexperimenten eingesetzt. Correll beschreibt in [26] ein Szenario zur Überwachung von Turbinen. Zur Evaluation des Szenarios wird ein Schwarm von Alice-Robotern mit einer verhaltensbasierten Steuerung eingesetzt. In [77] beschreibt Mermoud ein Experiment, in dem die Roboter eine kollektive Entscheidung zwischen ei-

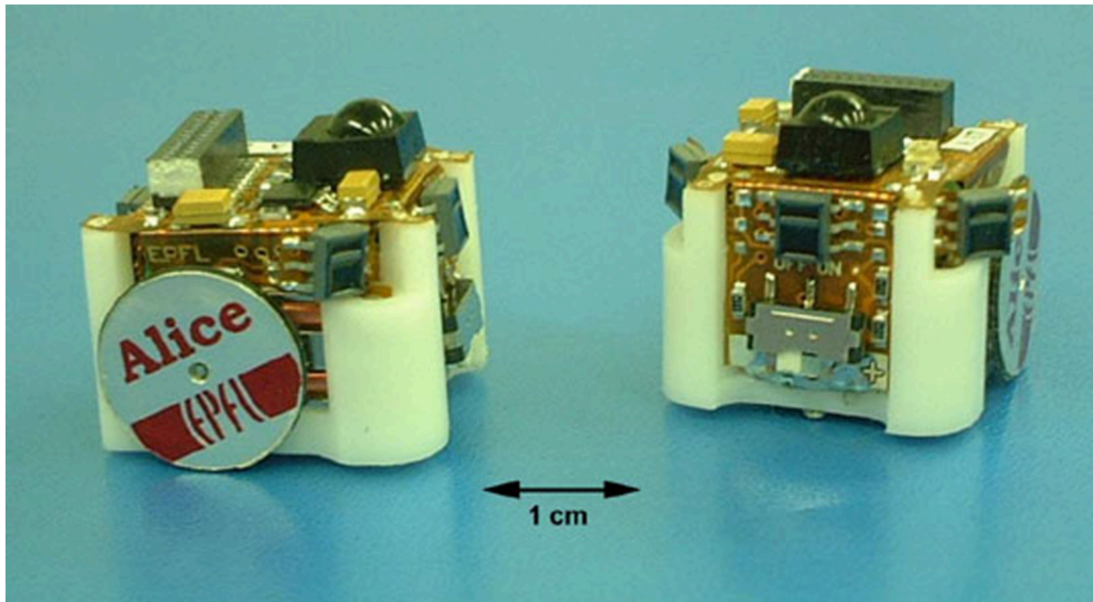


Abbildung 2.2: Der Alice Roboter; mit freundlicher Genehmigung von Gilles Caprari.

nem guten und einem schlechten Aufenthaltsort treffen sollen. Ziel ist es, in dem guten Ort zu aggregieren. Die Darstellung dieser Orte wird durch einen Projektor über der Arena realisiert. Die Roboter sind für die Wahrnehmung mit einer Photozelle ausgestattet.

Zum besseren Verständnis des Verhaltens von Kakerlaken, beschreiben Garnier und Jost [52, 42] einen Schwarm von Alice-Robotern, der sich qualitativ wie Kakerlaken verhält. Sie zeigen, dass sie das Aggregationsverhalten von Kakerlaken erfolgreich auf den Alice-Roboter übertragen konnten.

2.1.3 E-Puck

Der E-Puck-Roboter [84] ist ein mittelgroßer (Durchmesser 70 mm) autonomer mobiler Miniaturroboter ebenfalls basierend auf einem differentiellen Antrieb. Er wurde im Besonderen für die Ausbildung an Schulen und Universitäten von Michael Bonani und Francesco Mondada am ASL der EPFL Mitte des ersten Jahrzehnts des 21. Jahrhunderts entwickelt. Der E-Puck ist ausgestattet mit einem dsPic 30 CPU mit 30 MHz Taktung, 8 KB RAM, 144 KB Flash. Er hat acht IR-Näherungssensoren, eine Farbkamera, einen Ring mit acht *Licht Emittierende Dioden (LEDs)*, einen 3D-Beschleunigungsmesser, Bluetooth, drei Mikrophone und einen Lautsprecher. Erweiterung wie Bodensensoren, ZigBee oder 2D omnidirektionale

Kameras sind erhältlich. Der E-Puck wird seit seinem Erscheinen viel in der Kollektiven Robotik und der Evolutionären Robotik eingesetzt.

Abbildung 2.3: Der E-Puck Roboter; mit freundlicher Genehmigung von Francesco Mondada.

Seperati et al. beschreiben in [103] die Synthese von koordiniertem Gruppenverhalten mit EC auf KNNs. Das zuvor in der Simulation erzeugte Verhalten wurde dann auf drei echte E-Puck-Roboter übertragen. Correll et al. [27] übertragen hier den in [26] beschriebenen Versuch von Alice auf einen Schwarm von E-Puck-Robotern. Ziel ist die automatische Untersuchung einer Turbine mit Hilfe von Roboterschwärmen. Marcolino und Chaimowicz [71] beschreiben einen Algorithmus, mit dem sie einen Roboterschwarm um ein Objekt herum navigieren können. Für den auf Potentialfeldern basierenden Algorithmus benötigt jeder Roboter seine absolute Position. Im Experiment mit E-Puck-Robotern wird die Position der Roboter durch ein externes markerbasiertes System bestimmt und dann an die Roboter gesendet. Das gesamte System unterstützt bis zu 40 verschiedene Marker bei 25 Hz und einer Genauigkeit von 1 cm.

2.2 Mikroroboter

Zum Beginn der Entwicklung des I-SWARM-Roboters standen der Miniman-III- und der MiCRoN-Roboter. Diese Roboter waren nicht autonom und basierten auf einer zentralen Steuerung. Dennoch kann man sie als die direkten Urväter des I-SWARM-Roboters bezeichnen.

2.2.1 MINIMAN III

Der Miniman-III-Roboter [104, 120] ist ein handtellergroßer mobiler Mikroroboter, vgl. Abbildung 1.1. Er wurde zwischen 1998 und 2002 im von der *Europäischen Komission (EK)* finanzierten MINIMAN-Projekt, in Kooperation mit mehreren europäischen Partnerinstituten unter der Leitung von Heinz Wörn (IPR) entwickelt und gebaut. Der Miniman Roboter setzt sich im Wesentlichen aus zwei Hauptkomponenten zusammen, der Roboterplattform, die als holonome Positionierungseinheit [72] dient, und der austauschbaren Manipulationseinheit. Mit dem Miniman ist es möglich, Objekte im Nanometer Bereich zu manipulieren. Er wurde am IPR für die Mikromontage von mechanischen und optischen Systemen im Millimeterbereich, aber auch für die Manipulation von biologischen Zellen im Mikrometerbereich, eingesetzt. Abbildungen 1.1 zeigt den Miniman Roboter mit einem einfachen Zangengreifer.

Der Miniman-Roboter wurde in der Regel von einem Operator oder von einer zentralen Steuerung aus betrieben. Er arbeitete zusammen mit dem MiCRoN-Roboter im Team. Aufgrund der kabelbasierten Steuerung war der Roboter trotz holonomer Plattform in seinen Bewegungen relativ eingeschränkt.

2.2.2 MiCRoN

Der MiCRoN-Roboter in Abbildung 1.1 entstand zwischen 2002 und 2004 im gleichen europäischen Team ebenfalls unter der Leitung von Heinz Wörn. Ziel des MiCRoN Projektes war zum einen eine weitere Miniaturisierung der bestehenden Mikrorobotersysteme, und zum anderen sollte der Grad an Autonomie des Roboters stark erhöht werden. Ziel war ein kabelloser und damit in der Bewegung sehr flexibler Roboter. Dies ermöglicht ein weiters Ziel des Projektes, den Aufbau von kleinen Clustern von bis zu fünf Robotern [15], die zusammen, verbunden durch eine zentrale Steuerung, Mani-

pulationsaufgaben durchführen sollten. Die Erreichung dieses Ziels war ein wichtiger Schritt von kleinen Clustern von Mikrorobotern hin zu größeren Schwärmen, wie es in [121] von Wörn und Seyfried beschrieben wird.

2.3 Entwicklungsumgebungen und Softwarearchitekturen

In diesem Abschnitt werden bestehenden Entwicklungsumgebungen und Softwarearchitekturen zur Programmierung mobiler Roboter beschrieben, die in der Steuerung von mobilen Robotern zur Anwendung kommen. Die aufgeführten Softwarearchitekturen sind in Tabelle 2.1 bezüglich ihrer Eigenschaften zusammengefasst.

URBI [2] ist eine parallele, event-basierte Skriptsprache mit einem Interface zu C++-Objekten. Es ermöglicht Designern von Steuerungen, für mobile Roboter in kurzer Zeit komplexe Verhalten zu implementieren. Dabei unterstützt URBI einige verschiedene Roboter, wie z.B. Aibo, Lego Mindstroms NXT und iRobot Create. URBI-Programme werden jedoch nicht direkt auf dem Roboter ausgeführt. Dies schränkt die Nutzbarkeit im Bereich der MRS stark ein. Durch die benötigte Kommunikation zwischen Client und Server kann es zu Verzögerungen kommen. Die Größe des zu steuernden Schwarms ist stark abhängig von der zur Verfügung stehenden Bandbreite, worunter die Skalierbarkeit leidet. Die Form der Skriptsprache macht eine Anwendung von GP schwierig.

Pyro [6] ist ein auf Python basierendes Rahmenwerk zur Programmierung von plattformunabhängigen Robotersteuerungen. Es zielt direkt auf die Ausbildung von Schülern und Studenten. Pyro basiert auf einer Client/Server-Architektur, bei der ein minimaler Server auf dem Roboter läuft und die Robotersteuerung als Client auf einem externen Rechner. Es ermöglicht ebenso wie URBI in kurzer Zeit komplexe Verhalten zu implementieren. Pyro kann mit Player (s.u.) auf allen unter Player verfügbaren Robotern zugreifen. Direkten Zugriff ermöglicht Pyro auf den Pioneer-, Khepera-, Aibo- und Roomba-Roboter. Pyro ist aufgrund der Client/Server-Architektur den gleichen Beschränkungen ausgesetzt wie URBI. Die Pyro-Bibliothek enthält ein Modul

für genetische Algorithmen, was eine automatische Erzeugung von wiederverwendbaren Verhalten möglich macht.

Player [43] ist eine weit verbreitete Middleware für die Steuerung von mobilen Robotern. Wie URBI und Pyro basiert es auf einer Client/Server-Architektur. Auf Sensoren und Aktuatoren kann ein Client, der auf einem externen Rechner läuft, zugreifen und so den Roboter steuern. Hierbei stellt Player eine große Vielfalt an Schnittstellen für verschieden Programmiersprachen zur Verfügung. Treiber für viele verschiedene Sensoren und Roboter werden ebenfalls bereitgestellt. Wie bei URBI und Pyro läuft auch hier die Steuerung nicht direkt auf dem Roboter. Player ist daher denselben Beschränkungen ausgesetzt.

LEGO MindStorms' Lego stellt mit den LEGO MindStorms' NXT eine schnelle und kompakte Lösung, um in kurzer Zeit Roboterprotoypen zu bauen. Die geringe Anzahl an verwendbaren Sensoren schränkt jedoch die Anwendbarkeit auf sehr einfache Probleme ein [59]. Zur Programmierung von Robotersteuerungen stellt LEGO eine graphische Benutzerschnittstelle zur Verfügung, in der Sensoren mit Aktuatoren per *drag-and-drop* miteinander verbunden werden können. Zusätzlich gibt es eine C-ähnliche Programmiersprache *Not eXactly C* (NXC). Da der NXT mit einem rechenstarken ARM7 μC und mit relativ viel RAM (64 KB) ausgestattet ist gibt es sogar eine vereinfachte virtuelle Maschine für Java. Es stehen jedoch keine Möglichkeiten zum Debuggen zur Verfügung und die Plattform an sich stellt nicht die benötigte lokale Kommunikation bereit, die für die Kollektive Robotik oder Schwarmrobotik benötigt wird.

Das CubeSystem [5] ist eine modulare Plattform zum Bauen von mobilen Robotern. Das CubeSystem stellt sowohl Hard- und Softwarekomponenten zur Verfügung mit denen sich in kurzer Zeit mobile Roboter aufbauen lassen. Das CubeSystem besteht aus dem RoboCube (der Elektronik), dem CubeOS (dem Betriebssystem) und der RoboLib einer Bibliothek, die einige bekannte Algorithmen aus der Robotik, wie z.B. dead-reckoning, umfasst. Das CubeSystem stellt Treiber für diverse Sensoren bereit. Das gesamte CubeSystem wurde speziell für den RoboCube entworfen und ist nicht Open-Source, was eine Anwendung im Bereich der Forschung erschwert.

ASEBA [68] ist eine event-basierte eingebettete Middleware zur verteilten Steuerung von Robotern. Es wurde mit Blick auf die Kollektive Robotik entwickelt, und erfüllt somit viele Anforderungen, die an ein solches System gestellt werden. Es basiert auf einer interpretierten Skriptsprache, die Ähnlichkeiten mit der in Matlab verwendeten Programmiersprache hat. Eine mitgelieferte IDE ermöglicht das Debuggen von Robotersteuerungen während der Laufzeit. Beim Entwurf wurde auf geringen Speicherverbrauch (10 KB Flash, 4 KB RAM) geachtet. Zur Zeit unterstützt ASEBA den E-Puck-Roboter. ASEBA integriert den Roboter jedoch nicht in eine dynamische, interaktive Arena. Es liegen keine Arbeiten zu GP mit ASEBA vor.

Mathwork Matlab[1] und National Instruments LabView[2] sind beide sehr gute Entwicklungsumgebungen mit einer großen Palette an Möglichkeiten. Sie stellen auf einer einfachen Programmiersprache basierend eine Vielzahl an Funktionen zur Verarbeitung von Sensoreingaben und zur schnellen Berechnung von Ausgaben bereit. Eine Programmierschnittstelle zu USB, RS232, usw. ermöglicht es direkt auf Hardware zuzugreifen. Eine graphische Benutzeroberfläche sowie Möglichkeiten zum Debuggen sind vorhanden. Beide Programme sind jedoch nicht frei erhältlich, und erlauben es auch nicht, eigenständige Programme zu erzeugen, die auf einem μC laufen. Für Matlab existieren sowohl Werkzeuge für GP als auch andere genetische Algorithmen.

IDEs für die Entwicklung Eingebetteter Systeme stehen vielfältig auf dem Markt zur Verfügung. Es gibt viele kommerzielle Lösungen (z.B. Keil, CodeSourcery, etc.) aber auch einige freie Open-Source Lösungen (Eclipse, CodeSourcery Lite, GNU Toolchain, etc.). Die meisten Hersteller von μC stellen solche IDEs bereit oder verweisen auf Anbieter, mit denen sie zusammenarbeiten. Alle diese IDEs stellen Werkzeuge zum Debuggen und oft auch zur Simulation bereit. Sie gehen jedoch nicht auf die Bedürfnisse der Robotik und somit auch nicht auf die Bedürfnisse der Kollektiven Robotik ein. Es wird zumeist eine Kabelverbindung, zwischen dem μC und dem Rechner auf dem programmiert wird, benötigt.

[1] http://www.mathworks.com
[2] http://www.ni.com/labview

	IDE	Debugger vorhanden	leichte Bedienbarkeit	Sprache	Event-basiert	eingebettet	HW unabhängig	dynamische Arena	plattformunabhängig	Open-Source	Genetische Programmierung
MDL2ϵ	✓	✓	✓	speziell	✓	✓	✓	✓	✓	✓	✓
ASEBA	✓	✓	✓	speziell	✓	✓	✓	–	✓	✓	–
MindStorm	✓	–	✓	beliebig	–	✓	–	–	–	✓	–
Player	–	–	–	beliebig	–	–	✓	–	teilweise	✓	–
Urbi	–	–	✓	speziell	–	–	✓	–	✓	✓	–
Pyro	–	✓	✓	Python	–	–	✓	–	✓	✓	✓
Cubesystem	–	–	–	C/C++	–	✓	–	–	✓	–	–
Matlab/ LabView	✓	✓	✓	speziell	–	–	✓	–	✓	–	✓
IDE für Eingebettete Systeme	✓	✓	–	C/C++	k.A.	✓	–	–	✓	–	–

Tabelle 2.1: Vergleich von Softwarearchitekturen und Entwicklungsumgebungen für μC-basierte mobile Roboter. MDL2ϵ wird innerhalb der vorliegenden Arbeit beschrieben. Der Vergleich basiert auf [69].

Bibliotheken zur Programmierung von Robotern Es gibt viele Bibliotheken, wie OROCOS (Open Robot Control Software) [16] und YARP (Yet Another Robot Platform) [78], die eine saubere, modulare Architektur für die Programmierung von Robotern bereitstellen. Sie basieren zumeist auf einer Client/Server-Architektur die ein Interface für das Schicken und Empfangen von Nachrichten zwischen den Teilmodulen des Roboters bereitstellen. Aufgrund ihres Umfangs und ihrer Client/Server-Architektur laufen diese Bibliotheken nur auf externen Rechnern oder auf eingebetteten, leistungsstarken Rechnern, was die Anwendung zusammen mit Mikro- und Minaturrobotern ohne leistungsstarke Mikrocomputer nicht zulässt.

2.4 Schwarmrobotik Projekte

2.4.1 Swarm-Bots und Swarmanoid

Unter den Roboterschwärmen, die für die Forschung entwickelt worden sind, ist wohl das von der EK innerhalb der *Future and Emerging Technologies (FET)* geförderte Swarm-Bots Projekt [82, 83] eins der am meisten beachteten Projekte. Das von Marco Dorigo geleitete Swarm-Bots Projekt hatte eine Laufzeit von 42 Monaten und wurde Anfang 2005 beendet. In dem Projekt wurde ein Schwarmroboter – der S-Bot – vom EPFL unter der Leitung von Francesco Mondada entwickelt, der unter anderem die Fähigkeit hat, sich mit Hilfe von Greifern an andere Roboter anzuhängen [81]. Diese physikalische Verbindung ermöglichte es, dass der Roboter Hindernisse im Verbund überwinden konnte, die er als Einzelner zu überwinden nicht in der Lage gewesen wäre.

Abbildung 2.4: Schwarm von S-Bots; mit freundlicher Genehmigung von Francesco Mondada.

Der Roboter ist in zwei funktionale Teile unterteilt, die mobile Plattform mit differentiellem Kettenantrieb und den oben aufgesetzten Aktuatorteil mit Greifern, omnidirektionaler Kamera, RGB-LED-Ring und Hauptprozessor. Beide Teile sind voneinander drehbar gelagert. Eine reaktive, verteilte Steuerung des Roboters, war nach Aussagen des führenden Entwicklers F. Mondada, hauptsächlich erst durch einen, zwischen den zwei Hauptteilen eingebauten, Kraft-Momenten-Sensor möglich. Dies erleichterte Aufgaben im Bereich des kollektiven Transports [46, 89] und andere Verhalten, wie das Überwinden eines Hindernisses [112], welche einen Verbund der Roboter benötigen.

Der S-Bot wurde auch für Forschung im Bereich der Evolutionären Schwarm-/Kollektiven Robotik eingesetzt. Hierbei wurde untersucht, wie Phototaxis[3]-Verhalten [24] aber auch Kommunikation in Schwärmen [1] durch Methoden der EC evolviert werden können. Die Evolution basierte hier auf dynamischen KNNs, deren Gewichte im Laufe der Evolution angepasst werden.

Das Swarmanoid-Projekt ist seit Ende 2006 der direkte Nachfolger des Swarm-Bots Projektes, in dem die Forschung des Swarm-Bots-Projektes weitergeführt wird. Es endet im Oktober 2010. Ziel des Swarmanoid Projektes ist es zu untersuchen, wie ein heterogener Schwarm gebaut und gesteuert werden kann, der in einer 3D-Umgebung mit Menschen zusammen arbeitet. Hierfür werden drei verschieden Arten von Robotern gebaut.

Der Foot-Bot ist eine mobiler Roboter ausgestattet mit vielen Sensoren und untereinander andockbar. Er ist für den Transport von Objekten oder anderen Robotern vorgesehen. Der Eye-Bot ist ein fliegender Roboter, der speziell für das Überwachen und Analysieren einer Szene aus erhöhter Position entworfen wurde. Der Hand-Bot ist ein Roboter, der speziell zwischen den Bereichen von Foot- und Eye-Bot eingesetzt werden soll. Er soll sich mittels eines Seils innerhalb diese Bereiches fortbewegen.

2.4.2 Symbrion und Replicator

Das von der EK geförderte Symbrion- und das ebenfalls EK-geförderte Replicator-Projekt (2008 – 2013) [56], unter der Leitung von Paul Levi (*Institut für Parallele und Verteilte Systeme der Universität Stuttgart (IPVS)*), geht noch einen Schritt weiter als das Swarm-Bots und das Swarmanoid-

[3] Phototaxis ist ein Verhalten, bei dem die Fortbewegung des Individuums durch einen Lichtgradienten positiv wie negative beeinflusst wird.

Projekt. Sind in dem Swarm-Bot und Swarmanoid Projekt die Roboter noch lose gekoppelt, so vermischen das Symbrion- und das Replicator-Projekt den Bereich der Schwarmrobotik mit der selbst-rekonfigurierbaren Robotik. Sie stellen damit den aktuellsten Stand der Technik dar.

Beide Projekte teilen sich einen heterogenen Roboterschwarm bestehend aus zwei unterschiedlichen Roboterplattformen [55]. Gehen die Roboter untereinander eine Bindung ein, so ist diese zum einen mechanisch steif und beinhaltet zum anderen einen Energie- und Datenbus. Mit diesem Bus kann Energie zwischen den Robotern ausgetauscht werden. Hierüber findet auch eine auf Ethernet basierende Kommunikation statt, bei der jeder Roboter als Switch fungiert. Dies ermöglicht die massiveparallele Rechenkapazität[4], die in einem gekoppelten Organismus vorhanden ist, sinnvoll auszuschöpfen.

Zur Programmierung der Roboter wird auf Methoden der EC gesetzt [98], aber auch Verfahren, wie künstliche Immunnetzwerke [110], werden in den Projekten untersucht und angewendet.

Abbildung 2.5 zeigt den aktuellen Stand der Roboterentwicklung in den beiden Projekten.

2.5 Zusammenfassung

Aus dem Vergleich verwandter Arbeiten geht hervor, dass es verschieden Ansätze für Entwicklungsumgebungen für Miniaturroboter gibt. Bis auf ASEBA gibt es jedoch keinen Ansatz der sich direkt auf die Schwarmrobotik bezieht und die Entwicklung hierfür vereinfachen möchte. Gerade im Bereich einer integrierten Entwicklungsumgebung für Roboterschwärme, die auch eine interaktive, dynamische Arena und die umfangreiche Erfassung von Daten mit einbezieht, ist nichts vorhanden. Systeme mit denen die Position der Roboter berechnet werden kann, sind durch externe Systeme realisiert, die dann in Einzelfällen die Position den Robotern übermitteln. Lösungen, in denen der Roboter seine Position selbst berechnet gibt es nicht. Dies ist aber im Blick auf die Skalierbarkeit sehr wichtig.

[4]Jeder Roboter ist mit einem Dual-Core Blackfin BF-561 mit bis zu 500 MHz ausgestattet

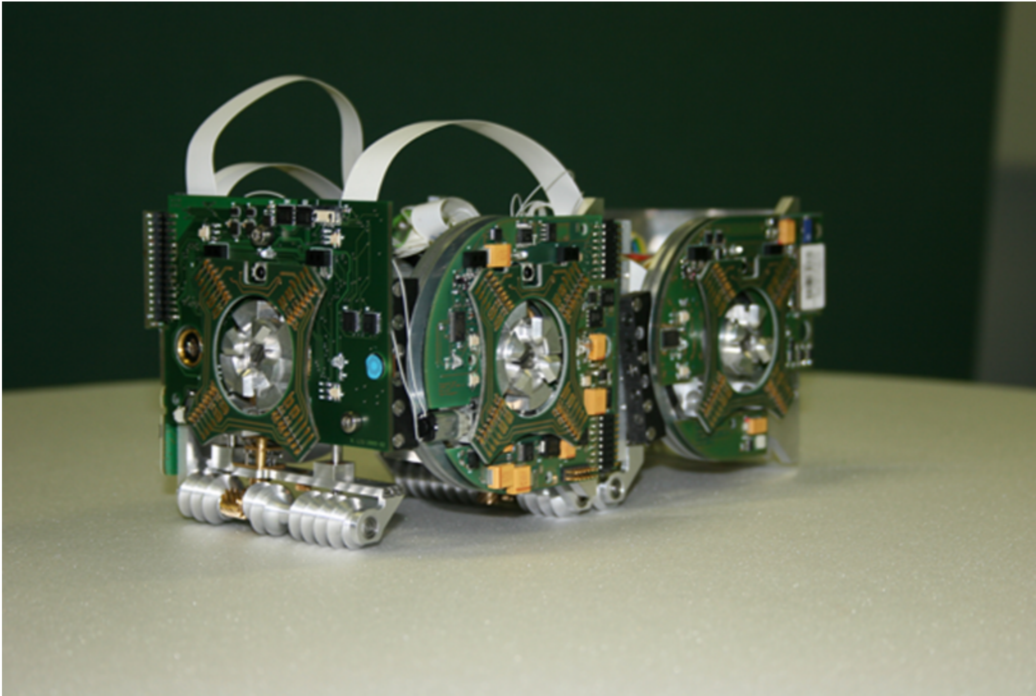


Abbildung 2.5: Zwei Prototypen des Symbrion-/Replicator-Roboters der Collective and Micro Robotics-Gruppe am IPR mit einen fast holonomen Schneckenantrieb im angedockten Zustand.

Arbeiten zu EC gibt es viele, die von Arbeiten mit einzelnen Robotern bis hin zu Arbeiten mit Schwärmen von Roboter reichen. Diese arbeiten jedoch auf KNNs und benutzen keine Genetische Programmierung. Besonders im Bereich der Schwarmrobotik mit Mikrorobotern gibt es keine schon vorhandenen Ansätze.

3. Mikro- und Miniaturroboter

Im Rahmen der Arbeit kamen mehrere Roboter zur Anwendung. Diese sind der Jasmine-, I-SWARM- und Wanda-Roboter. Sie unterscheiden sich in ihrer Größe, den verwendeten μCs, der Sensorik und der Aktuatorik, die auf den Robotern verwendet werden. Die unterschiedlichen Architekturen machen die Roboter sehr interessant für einen Vergleich der implementierten Software und zeigen deren vielseitige Anwendbarkeit. Interessant ist hierbei, dass sich die Software auf verschieden Arten von Informationsflüssen und Abläufen bezieht. Bei dem I-SWARM-Roboter wird aufgrund der entwickelten anwendungsspezifischen integrierte Schaltung (engl. *Application Specific Integrated Circuit (ASIC)*) jede Information über Hardware-Interrupts an die Steuersoftware weitergeleitet. Beim Jasmine-Roboter geschieht die Informationsakquise meist auf die typische Sense, Plan, Act Art, da die Sensordaten gepollt werden müssen. Bei dem Wanda-Roboter ist, aufgrund seines leistungsstarken μC, die Informationsakquise in ein Echtzeitbetriebssystem eingebettet. Auf die Einbettung der Steuersoftware in diese verschieden Abläufe wird in Abschnitt 4.2.6 genauer eingegangen.

3.1 I-SWARM

Der I-SWARM-Roboter wurde innerhalb des von Heinz Wörn geleiteten europäischen Projektes I-SWARM [101] entwickelt. Das Projekt lief von Anfang 2004 bis Mitte 2008. Ziel des Projektes war es, einen Schwarm aus bis zu 1.000 $3 \times 3 \times 3\ mm^3$ kleinen Mikrorobotern aufzubauen, der die Möglichkeit bot, zum ersten Mal Experimente mit sehr großen Roboterschwärmen durchzuführen. Ein weiteres Ziel war es, die Roboter mit Hilfe von Massenproduktionsverfahren herzustellen.

Massenproduktion ist für die Schwarmrobotik ein bisher viel zu selten beachteter wichtiger Punkt. Viele Wissenschaftler postulieren, dass die Schwarmrobotik mit einfachen und günstigen Robotern arbeitet. Betrachtet man jedoch die verwendeten Roboter genauer, so sind diese weder einfach noch preislich günstig und liegen wie z.B. der S-Bot im Bereich von einigen tausend Euro pro Stück. Bei Robotern, die in Massenproduktion günstig hergestellt werden können, ist es unerheblich, wenn ein gewisser Prozentsatz des Schwarms nicht funktioniert oder während des Betriebs defekt wird.

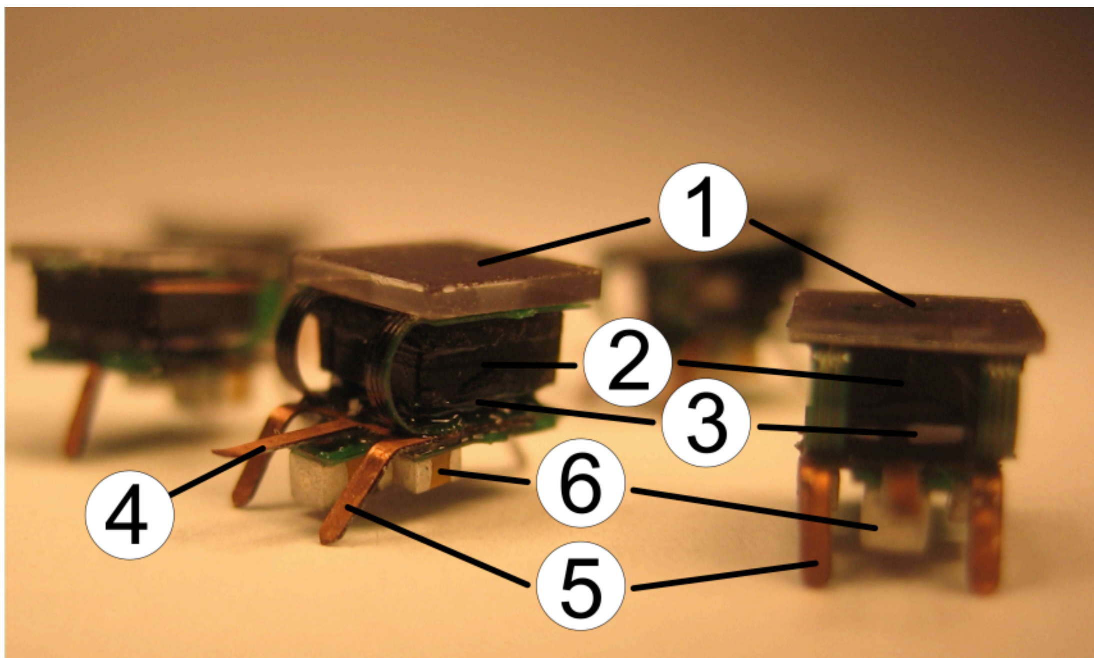


Abbildung 3.1: Der I-SWARM-Roboter. 1. Solarzelle; 2. Kommunikationsmodul; 3. ASIC; 4. Berührungssensor; 5. Beinchen; 6. Energiespeicher

Im Folgenden wird der Aufbau des I-SWARM-Roboters betrachtet. Abbildung 3.1 zeigt einen I-SWARM-Roboter. Der Aufbau des Roboters ist von oben nach unten:

①	Solarzellen
②	Kommunikationsmodul (IR)
③	ASIC
④	Berührungssensor (vibrierende Nadel)
⑤	Beinchen für die Bewegung des Roboters
⑥	Kondensatoren als Energiespeicher

Abbildung 3.2 zeigt schematisch den Aufbau des ASICs mit dem verwendeten DW8051 μC Kern und der Peripherie für die Ansteuerung der Sensoren sowie der Aktuatoren.

3.1.1 Prozessor

Wie schon im vorherigen Abschnitt beschrieben, wurde von der Universitat de Barcelona eigens ein ASIC zur Steuerung des I-SWARM-Roboters entworfen [22, 23]. In Abbildung 3.2 sieht man die einzelnen peripheren Steuereinheiten, die mittels von SFRs über den DW8051-Kern angesprochen werden können. Der DW8051 ist an verschiedene Speicher angeschlossen, die acht KB für Programme und zwei KB als *Random Access Memory (RAM)* zur Verfügung stellen. Alle Speicher sind aus Kostengründen als flüchtiger *Static Random Access Memory (SRAM)* Speicher angelegt. Dies hat den Nachteil, dass bei einem Neustart der Roboter die Steuersoftware immer wieder neu in den Speicher geladen werden muss.

Der verwendete Dallas DW8051-Prozessor-Kern ist ein 8-Bit-Prozessor mit modifizierter Harvard Architektur ohne Gleitkommaberechnungseinheit. Er kann in mehreren Stufen von 1,46 kHz bis 12 MHz getacktet werden. Ein extra Zeitgeber (engl. *timer*) ermöglicht es, den gesamten Prozessor schlafen zu legen, um Energie zu sparen. Der Prozessor wird dann nach Ablauf der vorher eingestellten Zeit wieder aufgeweckt, oder wenn ein Interrupt durch eine der anderen Peripherieeinheiten ausgelöst wurde.

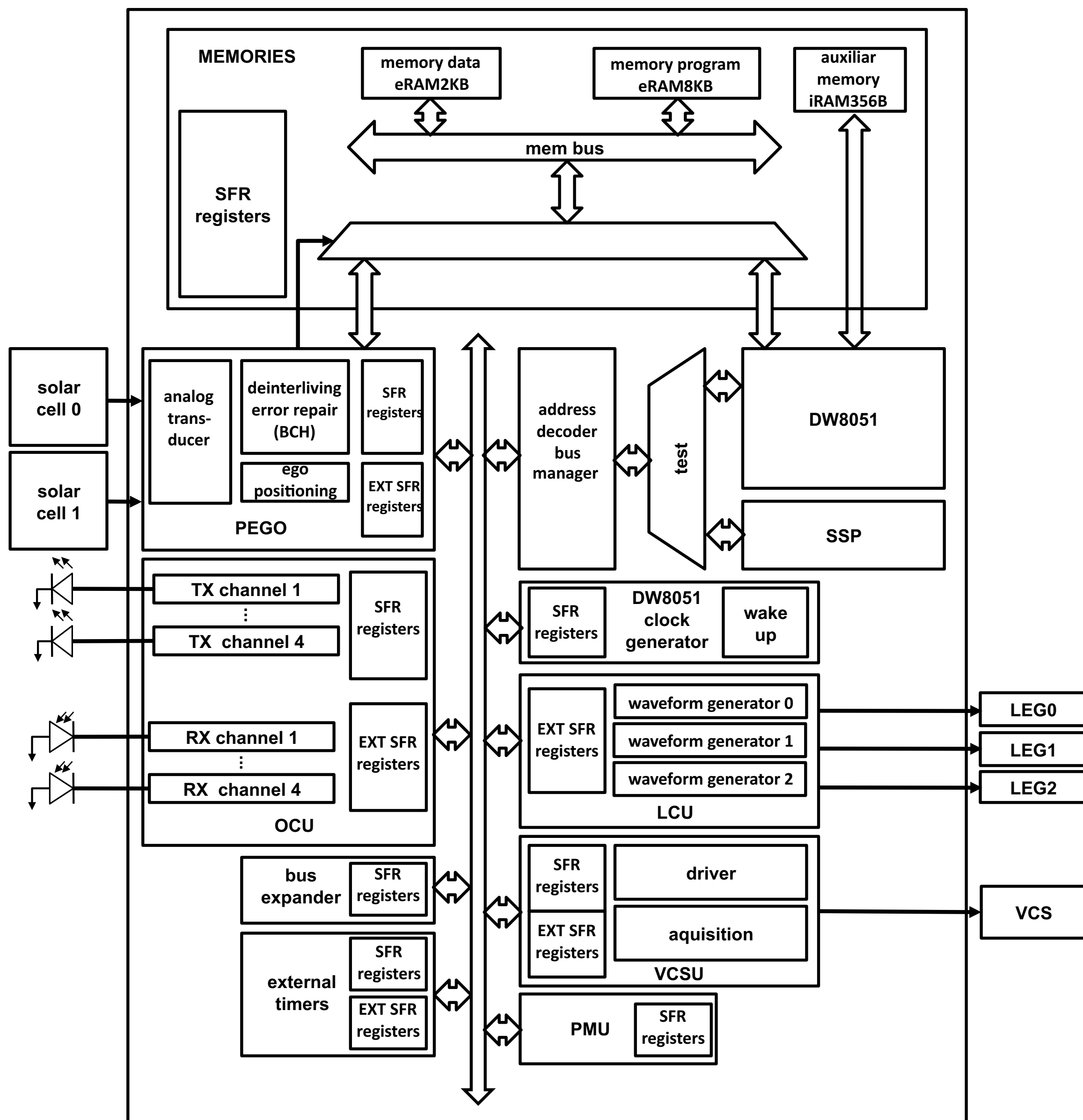


Abbildung 3.2: Blockdiagramm des I-SWARM ASIC mit der über die SFRs an den DW8051 angeschlossenen Module.

3.1.2 Sensorik

Die Sensorik des Roboters ist aufgrund der Größe des Systems eher simpel. Insgesamt hat er drei verschiedene Arten von Sensoren.

Vier um jeweils 90° versetzte Kommunikationskanäle [25], die auf IR basieren mit einer Kommunikationsreichweite von 7 mm. Die selben Sensoren werden auch für Kollisionsvermeidung benutzt, wobei ein einfaches Kommunikationsprotokoll hilft, zwischen Wand und Roboter zu unterscheiden.

Ein weiterer Sensor ist eine in Eigenschwingung versetzte Nadel, die als Berührungssensor eingesetzt werden kann. Dabei wird eine Berührung anhand der Veränderung der Eigenschwingung detektiert.

Der wohl mächtigste Sensor des Roboters sind zwei von der Solarzelle abgetrennte Photozellen, die zur Übermittlung von Daten mit Hilfe eines Projektors benutzt werden können [7]. Dies gibt die Möglichkeit, fehlende Sensoren im System zu ersetzen. So können absolute Positionsdaten übermittelt werden, die zur Abstands- und Winkelberechnung zwischen Robotern oder zwischen Objekten und Robotern herangezogen werden können. Des Weiteren lassen sich auch auf diese Art Umweltmuster, wie Lichtintensität, Futtermenge, Verschmutzungsgrad usw., in den Arbeitsbereich des Roboters projizieren. Hiermit sind viele verschiedene interessante Szenarien durchführbar.

3.1.3 Aktuatorik

Die Aktuatorik des Roboters besteht aus der mobilen Plattform, die durch drei vibrierende Beine mit mehreren Lagen von piezoresistiven Polymeren aufgebaut wurde [10]. Die Beinchen ermöglichen Vorwärts- und Rückwärtsbewegung des Roboters, sowie das Laufen auf einer durch die Konstruktion festgelegten Kreisbahn. Mit Hilfe einer interruptbasierten Softwarelösung lässt sich auch eine Rotation des Roboters auf der Stelle ermöglichen.

3.1.4 Besonderheiten

Für den I-SWARM-Roboter wurde eine Arena entwickelt, die ihn zum einen mit genügend Licht versorgen soll, so dass er über seine Solarzellen ausreichend Energie zur Verfügung hat, und zum anderen können mittels der Arena über einen Projektor verschiedene Daten, wie z.B. die Position des Roboters, in die Arena projiziert werden. Da der I-SWARM-Roboter keine

physische Schnittstelle zur Programmierung hat, wird er ebenfalls über den Projektor programmiert. Diese Art der Programmierung benötigt 45 Minuten für ein acht KB großes Programm, was die Entwicklung und Anpassung von Steueralgorithmen auf dem Roboter stark erschwert. Das Schreiben des Speichers kann in Blöcken von 256 KB stattfinden, was bei kleinen Programmen die benötigte Zeit erheblich reduziert. Abbildung 3.3 zeigt die vom *Fraunhofer-Institut für Biomedizinische Technik (IBMT)* entwickelte Arena.

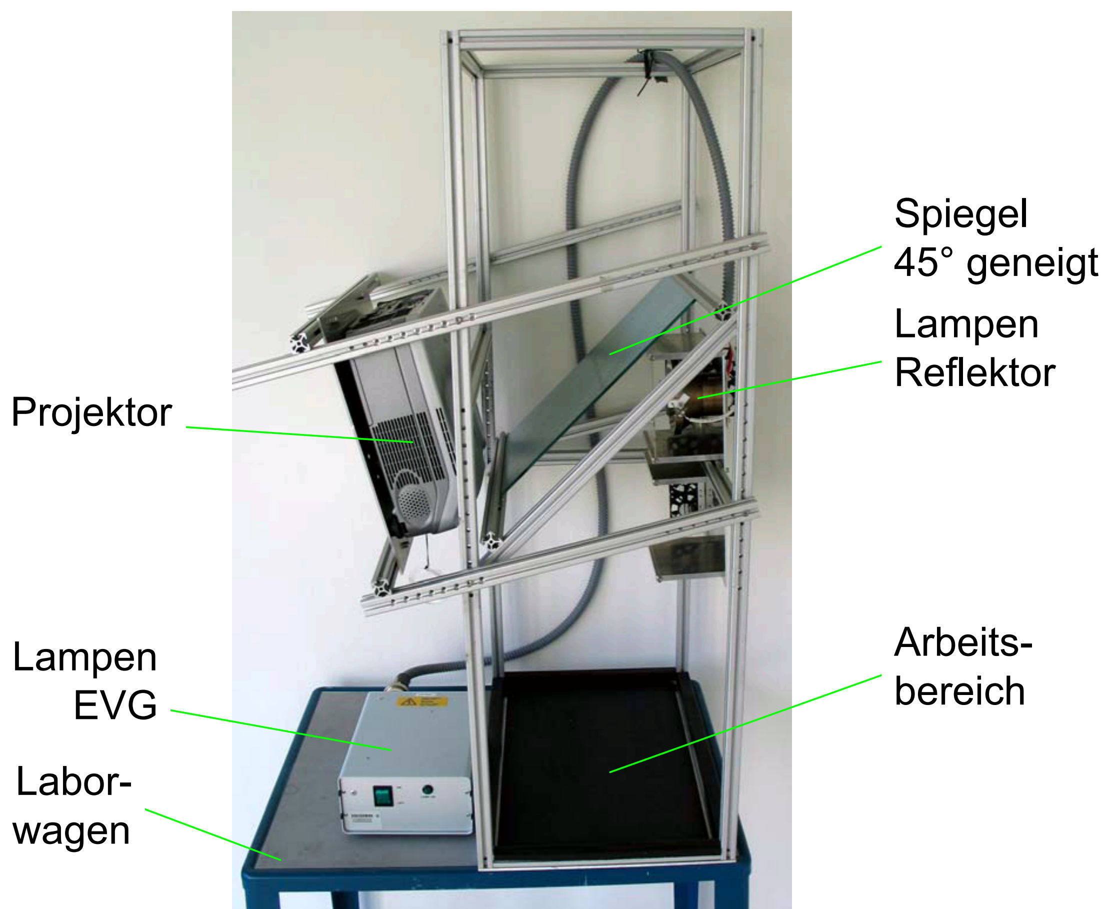


Abbildung 3.3: Die am IBMT entwickelte I-SWARM-Arena mit Projektor, Beleuchtung für die Energieversorgung und dem Arbeitsbereich in der Größe DIN A4.

Der I-SWARM-Roboter war bis zur Anfertigung der Dissertation nicht funktionsfähig. Jegliche Software, die sich auf den I-SWARM-Roboter bezieht, wurde mit Hilfe einer *Field Programmable Gate Array (FPGA)*-Entwicklungsumgebung portiert, implementiert und debugged. Der FPGA enthielt die gesamten digitalen Teil des ASIC, den DW8051-Kern eingeschlos-

sen. Alle bekannten Fehler im Silizium des ASIC wurden in der Software berücksichtigt. Das implementierte *I-SWARM* MDL**2**ϵ *Operating System (IsMOS)* wurden auf dem ASIC getestet.

3.2 Jasmine

Der Jasmine-Roboter ist ein 34 × 34 × 34 mm^3 großer Roboter. Er wurde vom IPVS in Zusammenarbeit mit dem IPR im Rahmen des I-SWARM-Projektes als Entwicklungs- und Testplattform entwickelt. Es wurde gefordert, den Roboter so ähnlich wie möglich dem I-SWARM-Roboter aufzubauen, und mit ähnlichen Sensoren auszustatten. Abweichungen von der Sensorik entstanden zum einen durch die damaligen Spezifikationen des I-SWARM-Roboters und zum anderen durch die verfügbare Technologie, welche möglichst günstig sein sollte. Abbildung 3.4 zeigt einen "Jasmine III+"-Roboter ausgestattet mit dem *Optical Data transmission system for Micro robots (ODeM)*-Sensor, auf den in Abschnitt 4.4.2 genauer eingegangen wird. Der Aufbau des Roboters ist von oben nach unten:

①	ODeM Board	–	ODeM-Sensor und ZigBee
②	Controller Board	–	AtMega168, Kommunikations-/Abstandsmessungsmodul (IR) und IR-Fernsteuerungsempfänger
③	LiPoly Board	–	Energiespeicher, mit Laderegler und Ladekontakten
④	Motor Board	–	AtMega88, inkrementfreie Drehzahlregelung, Berührungssensor und Motorsteuerung

3.2.1 Prozessor

Der Jasmine-Roboter ist mit einem AtMega168[1] μC mit 8-Bit AVR *Reduced Instruction Set Computing (RISC)*-Prozessor ausgestattet. Er bietet 16 KB Flash Programmspeicher, einem KB SRAM und 512 Bytes *Electrically Erasable Programmable Read-Only Memory (EEPROM)*. Der μC

[1] http://www.atmel.com

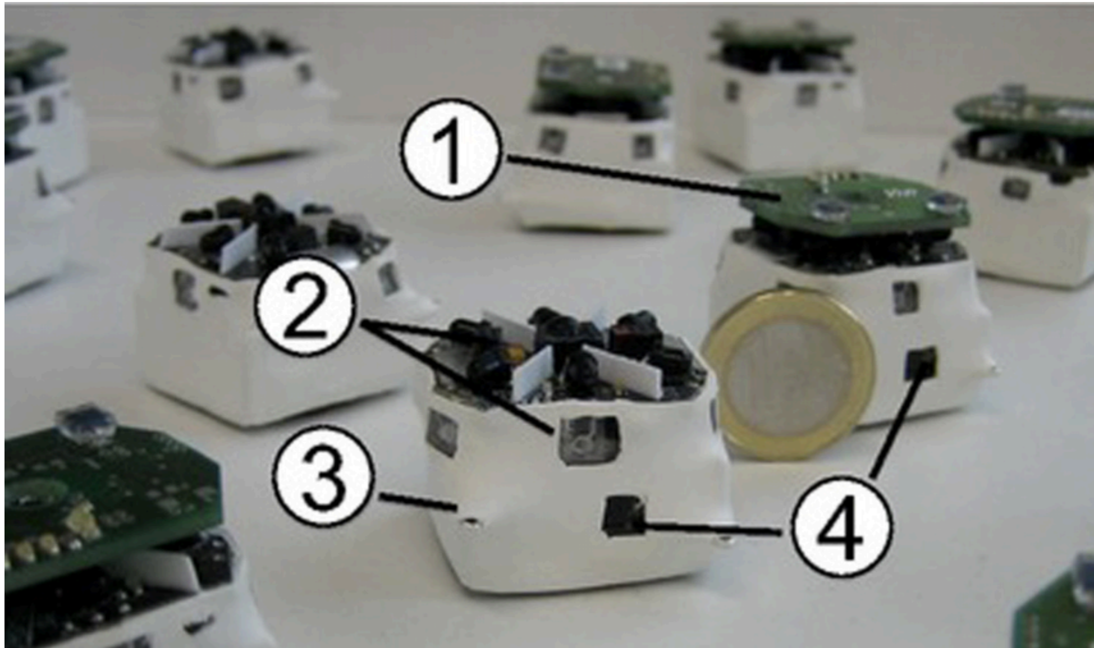


Abbildung 3.4: Der Jasmine-III+-Roboter mit und ohne ODeM-Sensor. 1. ODeM; 2. Controller Board; 3. Ladekontakt, LiPoly; 4. Berührungssensor auf Motor Board

wird durch einen internen 8-MHz-Takt versorgt. Zum Anschließen von Aktuatoren und Sensoren stehen zum einen mehrere *Analog Digital Converters (ADCs)*, sowie folgende Schnittstellen zur Verfügung: *Two-Wire Interface (TWI)*, *Serial Peripheral Interface (SPI)* und *Universal Asynchronous Receiver Transmitter (UART)*. Zusätzlich werden auch die *General Purpose Input Output Ports (GPIOs)* und *Pulsweitenmodulation (PWM)*-Kanäle verwendet.

Der AtMega168 Prozessor ist mit ungefähr einer *Millionen Instruktionen pro Sekunde (MIPS)* pro MHz von der Rechenleistung her ähnlich wie der DW8051. Der größere Programmspeicher wird aufgrund der Tatsache, dass alle Sensordaten gepollt und danach durch extra Treiber verarbeitet werden müssen, relativiert. Im Gegensatz zum I-SWARM-Roboter muss der Jasmine-Roboter aktiv auf eine Nachricht lauschen. Dies wird beim I-SWARM-Roboter durch ein extra Kommunikationsmodul übernommen. Daher wird ein größerer Teil des Programmspeichers beim Jasmine-Roboter durch Treibersoftware belegt.

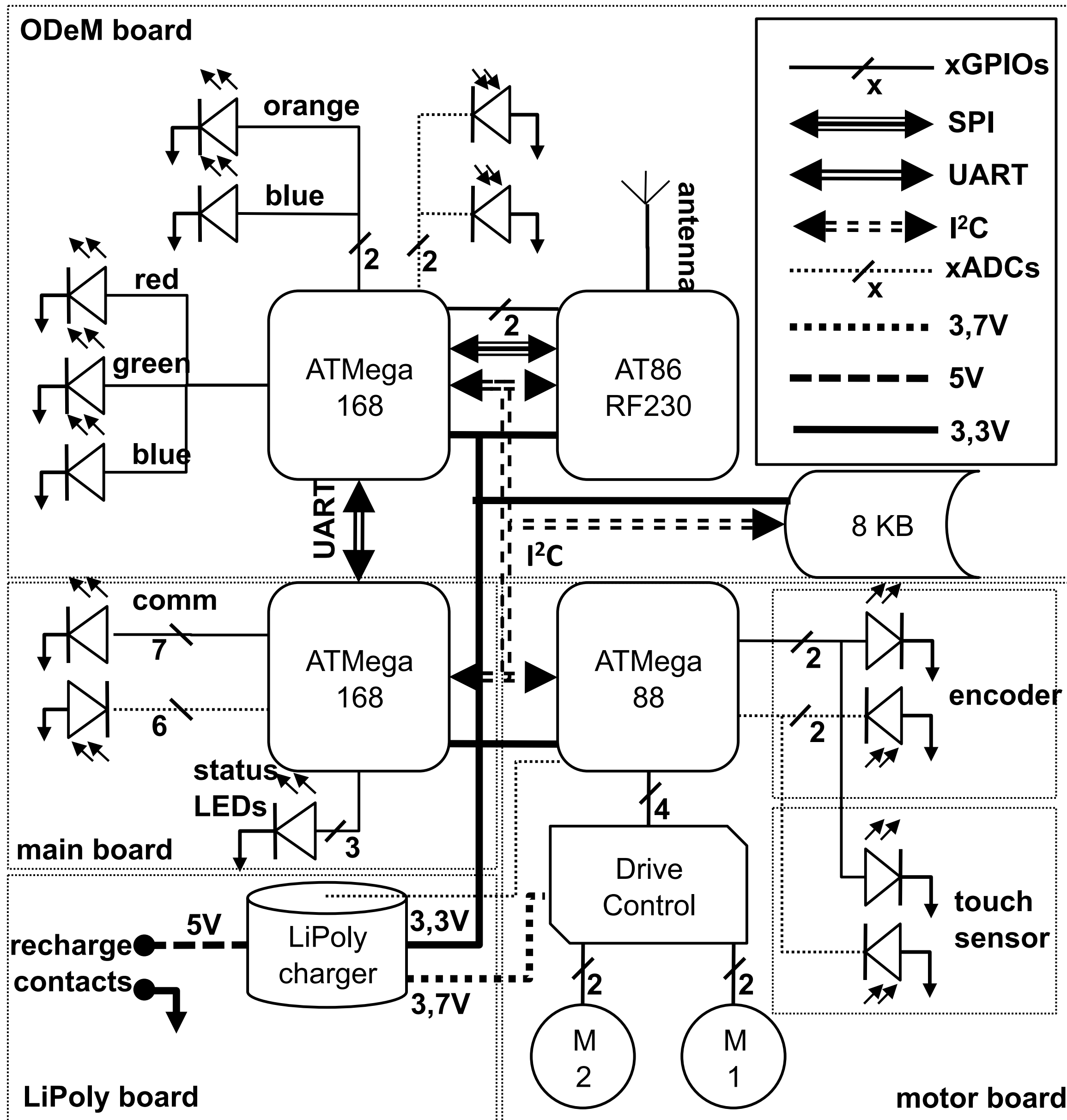


Abbildung 3.5: Blockdiagramm des Jasmine-III+-Roboters.

3.2.2 Sensorik

Der Jasmine-Roboter hat zunächst sechs um 60° um die Hochachse des Roboters versetzte IR-Sensoren (optisch getrennter Empfänger und Sender), die zum einen für die lokale Kommunikation mit anderen Robotern, und zum anderen zur Abstandsmessung verwendet werden. Für die Abstandsmessung steht noch ein weiterer IR-Sender zur Verfügung, der einen schmaleren Abstrahlwinkel als die Sender für die Kommunikation aufweist. Das zur Abstandsmessung angewendete Prinzip basiert auf dem Aussenden eines IR-Impulses und dem Messen des zurückkommenden reflektierten Lichtes. Dieser Ansatz ist sehr anfällig auf den IR-Anteil im Umgebungslicht. Gerade bei Sonnenlicht ist der Roboter nicht mehr einsetzbar.

Ein weiterer Sensor ist der ebenfalls auf IR basierende Berührungssensor, der eine maximale Reichweite von zehn Millimetern hat.

Nicht sichtbar im Inneren des Roboters befinden sich Sensoren zur Messung der Radumdrehungen, die für die Odometrie des Roboters verwendet werden. Diese sind ebenfalls als IR-Sensoren implementiert. Die maximale Auflösung der Sensoren beträgt nur zwei Messpunkte pro Umdrehung, was die Genauigkeit der Drehzahlregelung sehr beschränkt und einen Schwachpunkt des Jasmine-Roboters darstellt.

Der Jasmine-Roboter ist mit einem Energiesensor der den Ladestand der *Lithium Polymer (LiPoly)*-Zelle angibt ausgestattet. Alle weiteren Sensoren, die sich auf der ODeM-Platine befinden, werden in Abschnitt 4.4.2 gesondert betrachtet, da sie eng mit dem Aufbau der Arena verbunden sind.

3.2.3 Aktuatorik

Die einzigen Aktuatoren, die der Jasmine-Roboter besitzt, sind zwei über PWM angesteuerte Motoren zur Bewegung des Roboters. Diese stellen einen klassischen differentiellen Antrieb dar, wie er oft bei mobilen Robotern zur Anwendung kommt. Die Geschwindigkeit des Roboters beträgt zwischen 50 mm/sek und 300 mm/sek.

3.2.4 Besonderheit

Eine Besonderheit des Roboters ist der integrierte Ladeschaltkreis für den LiPoly Akku. Dies ermöglicht es, dass der Roboter sich selbst wieder auflädt, sobald der Energiesensor ihm einen zu niedrigen Ladestand anzeigt.

3.3 Wanda

Der Wanda-Roboter ist 33 mm - 45 mm hoch und hat einen Durchmesser von ca. 50 mm. Er wurde vollständig im Rahmen der vorliegenden Dissertation und dem Graduiertenkolleg 1194 *Selbstorganisierende Sensor-Aktor-Netzwerke* am IPR entworfen und gebaut. Die Realisierung eines Schwarms von bis zu 125 Robotern wurde vom *KIT Collective Robotics Lab (KITCoRoL)* gefördert.

Das Konzept des Wanda-Roboters versucht Probleme des I-SWARM-Roboters und des Jasmine-Roboters zu lösen und dabei Vorteile der zwei anderen Roboter zu integrieren. Als Nachteile des Jasmine-Roboters haben sich die auf ADC basierte Kommunikation, die fast nicht vorhandene Odometrie, sowie der extrem hohe Anteil an Handarbeit bei der Fertigung des Roboters herausgestellt. Auch der recht leistungsschwache μC, welcher in Bezug auf den I-SWARM-Roboter ausgewählt wurde, wurde im Wanda-Design durch einen leistungsstärkeren ersetzt. Vorteile des Jasmine-Roboters, die in das Wanda Design übernommen wurden, waren die Möglichkeit zum Selbstaufladen, sowie die Aufteilung der Kommunikation in sechs Richtungen. Der für die Jasmine entwickelte ODeM-Sensor wurde mit leichten Anpassungen vollständig übernommen.

Bei der Programmierung des I-SWARM-Roboters hat sich die stark interruptbasierte Architektur mit viel Vorverarbeitung der Daten in der Hardwareperipheri des Roboters als sehr vorteilhaft dargestellt. Dieses interruptbasierte Konzept spiegelt sich in fast allen Sensoren des Wanda-Roboters wider. Hier kann sich der Benutzer entscheiden, ob die Daten aktiv abgefragt werden, oder ob die Sensoren so konfiguriert werden, dass sie bei bestimmten Ereignissen, Interrupts auslösen.

Allgemein war das Ziel beim Entwurf des Wanda-Roboters einen möglichst universellen Schwarmroboter mit möglichst viel Sensorik, guter und anpassbarer Inter-Roboter-Kommunikation, mit einem einfachen Aufbau für die Produktion des Roboters zu entwickeln. Aus diesem Grund wurde ein Design gewählt, das vollständig vom Rad bis zum Prozessorboard auf ein zweilagiges *Printed Circuit Board (PCB)* passt. Dies minimiert die Kosten sowie die Anzahl der von Hand zu lötenden Teile beim Zusammenbau des Roboters[2].

[2] Beim Zusammenbau müssen nur die Anschlüsse für die Motoren sowie der vordere Sensor von Hand eingelötet werden.

Durch das spezielle Design konnte der Anteil an Handmontagearbeit, inklusive Test aller Bauteile, von ca. 4-5 Mannstunden bei der Jasmine auf eine Mannstunde bei Wanda reduziert werden.

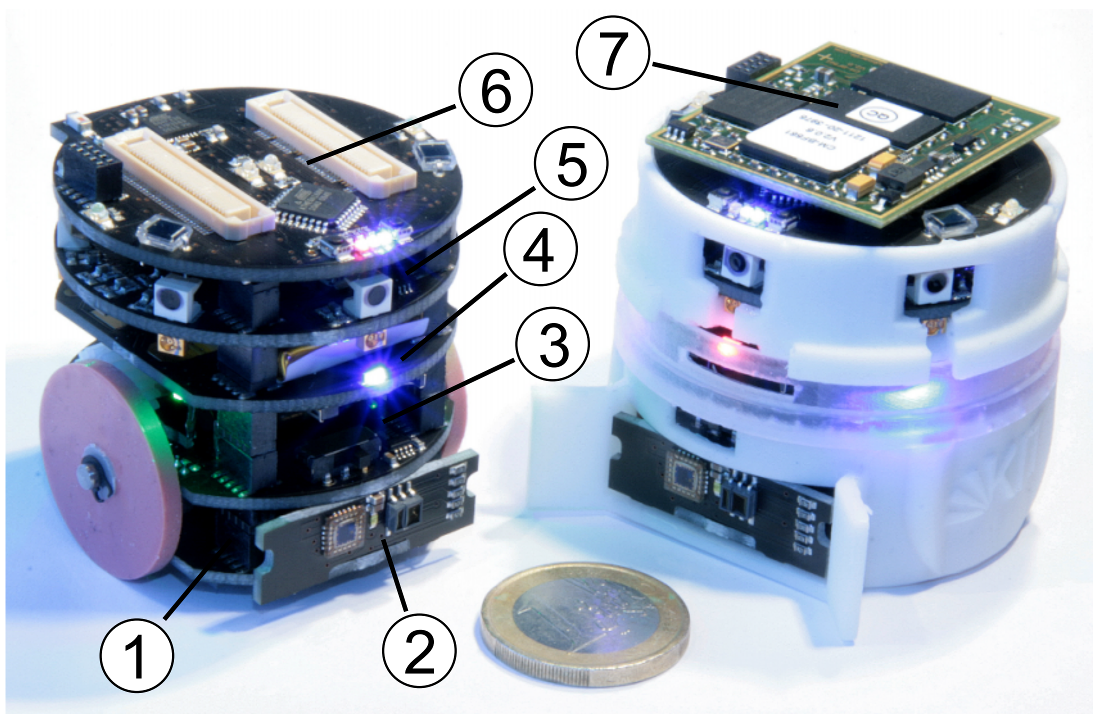


Abbildung 3.6: Wanda-Roboter mit und ohne Chassis. Der Roboter mit Chassis ist zusätzlich mit einem CM-BF-561 BlackFin-Board von Bluetechnix bestückt. 1. Bottom-Board; 2. RGB-Sensor; 3. Motor-Board; 4. Hilfsakku-Board; 5. Cortex-Board; 6. ODeM-Board; 7. BlackFin CM-BF-561

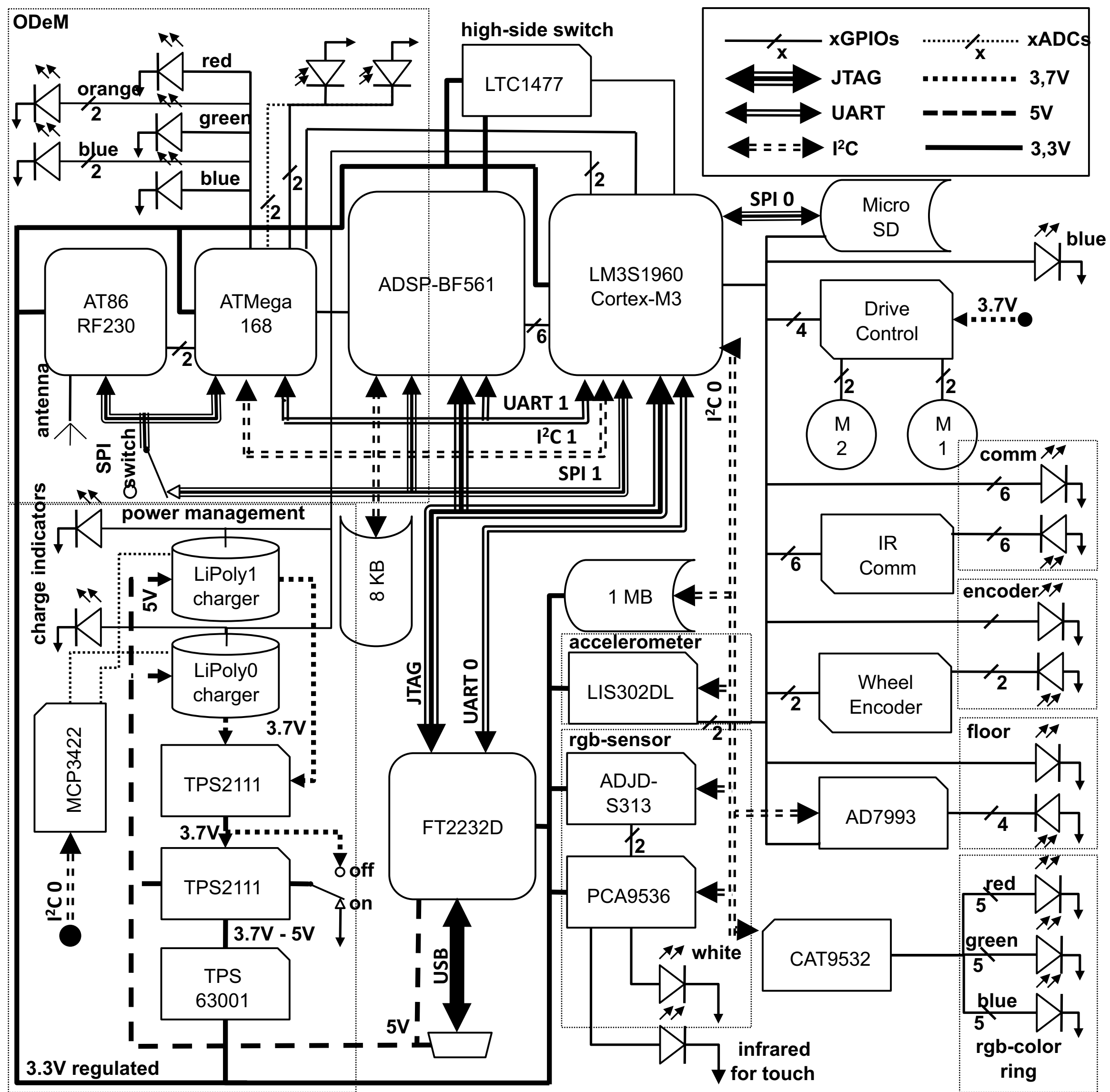


Abbildung 3.7: Blockdiagramm des Wanda-Roboters.

3.3.1 Aufbau

Der Roboter besteht aus bis zu sieben PCBs, welche in Abbildung 3.6 dargestellt sind. Sie sind von unten nach oben:

①	Bottom-Board	–	Bodensensoren, Beschleunigungssensor, LiPoly und USB (JTAG/UART)
②	RGB-Sensor	–	RGB- und Berührungssensor
③	Motor-Board	–	Motorsteuerung, Inkrementgeber und TWI-Speicher
④	Hilfsakku-Board	–	LiPoly, SD-Karte und fünf RGB-LEDs
⑤	Cortex-Board	–	Cortex μC, Infrarot Kommunikation und Ladestandsmessung
⑥	ODeM-Board	–	ODeM-Sensor und ZigBee
⑦	Blackfin-Board	–	Bluetechnix Dual Core BlackFin μC CM-BF-561[3]

Das unterste Modul enthält eine *Universal Serial Bus (USB)*-Schnittstelle. Diese Schnittstelle wird zum Laden des Roboters, für das Debuggen und Programmieren des Roboters mit Hilfe von *Joint Test Action Group (JTAG)*, sowie für eine UART-Schnittstelle verwendet. Ferner ist auf diesem PCB ein LiPoly-Akku mit 250 mA/h sowie eine Schnittstelle für ein nach vorne gerichtetes Sensorboard. Drei auf den Boden gerichtete Grauwertsensoren mit zugehörigen ADCs, zwischen denen Schleifkontakte zum selbständigen Laden sind, befinden sich auf der Unterseite. Ein dreiachsiger Beschleunigungssensor befindet sich unterhalb der Radachse aber ausserhalb des Drehpunktes.

Der zweite PCB ist für den differentiellen Antrieb des Roboters. Es enthält die Motortreiber, sowie auf IR basierte Inkrementgeber. Das Enkoderrad für die Inkremente ist in die Rädern als 18 freistehende Kupferflächen integriert. Ein Schmitttrigger erzeugt den Interrupt für jedes Inkrement. Zusätzlich ist auf diesem Board ein 1 MB großer Festspeicher integriert, der zum Speichern von roboterspezifischen Konfigurationsdaten genutzt wird.

[3] http://www.bluetechnix.com

Der dritte PCB ist für den Betrieb des Roboters im Allgemeinen nicht unbedingt notwendig. Es kann vollständig weggelassen werden, ohne die Fähigkeiten des Roboters stark zu beschränken. Dieses Board enthält eine zusätzliche 250 mA/h LiPoly Zelle, sowie die notwendige Elektronik, die für Laden des Akkus und für das Umschalten zwischen dem Akku auf dem untersten Board und der zusätzlichen Zelle notwendig ist. Diese Elektronik ermöglicht es, dass zuerst immer der Hilfsakku und dann erst der Hauptakku entladen wird. Geladen werden beide Akkus gleichzeitig. Dies verlängert die Lebensdauer der einzelnen Zellen. Zusätzlich sind auf diesem PCB fünf RGB-LEDs, welche über eine CAT9532[4] gesteuert werden. Dieses ermöglicht zusammen mit einem RGB-Sensor, der auf dem vorderen Sensor untergebracht ist, eine Roboter-Roboter-Kommunikation über Farben. Eine auf dem Board integrierte Micro SD-Kartenhalterung ermöglicht es, während der Laufzeit mehrere Gigabyte an Daten zu sammeln oder Teile der Steuerungssoftware abzulegen.

Das vierte Board enthält den LM3S1960[5] μC, auf welchen im nächsten Abschnitt genauer eingegangen wird. Es implementiert ferner die IR-basierte Kommunikation und enthält ADCs für die Abfrage des Ladezustandes der einzelnen Zellen. Dieses Board stellt einen logischen Schnitt in den Bussystemen des Roboters dar. Es führen von diesem Board jeweils ein extra TWI-, SPI- und UART-Bus nach unten, sowie einer nach oben, vgl. Abbildung 3.7. Hierdurch ist es möglich, alle Boards, die auf dieses Board gesteckt werden, unabhängig von den für die korrekte Funktion des Roboters wichtigen unteren Boards zu trennen. Die Schnittstelle auf diesem Board ist für Erweiterungsboards gedacht.

Ein Erweiterungsboard ist das schon bei Jasmine verwendete ODeM-Board. Es stellt Schicht fünf dar. Zusätzlich zu dem schon bekannten ODeM-Sensor ist eine Schnittstelle für ein leistungsstarkes Blackfin-Board von Bluetechnix integriert, das den Betrieb eines Dual-Core-Blackfin-Prozessor mit bis zu 500 MHz Taktung, 64 MB SDRAM und 8 MB Flash-Speicher ermöglicht. Dieses Board lässt sich mit uClinux™ betreiben. Um den Strombedarf diese Boards zu decken, ist jedoch eine größere LiPoly-Zelle anzuschließen. Der Blackfin lässt sich bei Bedarf vom Cortex ausschalten.

[4]http://www.onsemi.com

[5]http://www.ti.com/Stellaris

3.3.2 Prozessor

Der Wanda wird mit einem LM3S1960 μC von Luminary Micro (Texas Instruments) betrieben. Der Kern des LM3S1960 ist ein ARM-Cortex-M3-Prozessor mit 50 MHz Taktung, 64 KB SDRAM und 256 KB Flash-Speicher. Der LM3S1960 hat drei UART-Schnittstellen[6], 7-60 GPIOs, zwei SSI/SPI-Schnittstellen, zwei TWI-Schnittstellen, sechs PWM Kanäle und besitzt keinen einzigen ADC. Die ADCs wurden, wenn nötig, durch TWI-basierte ADC-Bausteine implementiert. Dieses hat zum Vorteil, dass die ADCs so konfiguriert werden können, dass sie bei bestimmten Werten einen Interrupt auslösen, was den μC entlastet. Der μC besitzt des Weiteren 24-Bit Systick Zeitgeber und 4×32-Bit bzw. 8×16-Bit Zeitgeber.

Das für den Roboter implementierte Betriebsystem basiert auf dem eingebetteten Echtzeitbetriebssystem FreeRTOS für die echtzeitbasierte Tasksteuerung. Die von Luminary Micro zur Verfügung gestellte Treiberbibliothek ermöglicht einen einfachen Zugriff auf alle Hardwareressourcen des Roboters. Die Verwendung eines Echtzeitbetriebssystems, sowie vieler interruptbasierter Sensoren, ermöglicht es, Sensoren zu pollen, ohne dabei die Echtzeit der *extended Motion Description Language 2 (*MDL**2**ϵ*)*-basierten Steuerung zu unterbrechen, wie es bei dem Jasmine während der Kommunikation der Fall war. Daten fließen unabhängig in das System und werden in einem Weltmodell gespeichert. Das implementierte Betriebsystem ermöglicht es der Steuerung, über Nachrichten auch direkt auf wichtige, eingehende Ereignisse ohne Verzögerung zu reagieren. Dieses System basiert auf sogenannten *Hooks* und wird in [106] eingehend beschrieben.

3.3.3 Sensorik

Die reichhaltige Sensorik des Roboters ist ein Schlüssel zur Anwendung als Roboter für Schwarmexperimente. Im Folgende wird auf die einzelnen Sensoren genauer eingegangen.

Kommunikation

Wie im I-SWARM- und Jasmine-Roboter basiert die Kommunikation des Wanda auf IR-Licht. Er besitzt wie der Jasmine-Roboter sechs um jeweils 60° versetzte Sender und Empfänger. Im Gegensatz zum Jasmine-Roboter ist die Kommunikation unabhängig von ADCs, sie ist ähnlich zum

[6] Es werden nur zwei der drei UART-Schnittstellen verwendet.

I-SWARM-Roboter rein interruptgesteuert. Anders als beim I-SWARM-Roboter findet das Protokoll mit Datenanalyse und Fehlerkorrektur auf dem μC statt [58]. Die an die Empfänger angeschlossene Hardware erzeugt einen Interrupt für eingehende Impulse. Das Besondere daran ist, dass die Dauer des Interrupts mit der Intensität des eingegangenen Impulses korreliert. Als Protokoll für die Bitkodierung wurde der Pulsabstand gewählt. Hiermit können bis zu vier Bits durch zwei aufeinanderfolgende Impulse kodiert werden. Ein einzelnes Nachrichtenpacket enthält 16 Bit Nutzdaten, zwei Bit für den Nachrichtentyp und vier Bit für Fehlererkennung. Die Nachrichtenintensität wird für die Abstandsbestimmung verwendet. Um Interferenzen zwischen kommunizierenden Robotern zu vermeiden, wurde ein Zeitschlitzverfahren implementiert, das jedem Roboter einen von zehn Zeitschlitzen zuweist. Die Zeitschlitze sind jedoch nur lokaler Natur und können auf den sechs Kanälen unterschiedlich sein. Durch den impulsbasierten Kommunikationsansatz ist der Roboter im Gegensatz zum Jasmine-Roboter nicht anfällig auf im Tageslicht enthaltene Infrarotanteile.

Odometrie bzw. Inkrementgeber

Ein großes Manko des Jasmine-Roboters war die fast nicht vorhandene Odometrie. Eine Regelung des Roboters war fast nicht möglich, was zu großen Fehlern im Fahrverhalten der Roboter führte. Diese Manko wurde im Wanda-Roboter durch ein Enkoderrad mit 18 Schritten verringert. Zusammen mit einem PID-Regler lässt sich der Roboter gut regeln. Ein Model zur Wegmessung (Odometrie) basierend auf der Umdrehungsgeschwindigkeit der Räder ist somit möglich.

Grauwertsensoren

Am unteren Board befinden sich drei Sensoren, mit denen sich die Reflektivität des Bodens messen lässt. Mit Hilfe solcher Sensoren lassen sich zum einen verschiedene Bereiche, wie z.B. Nest, Futter und Gefahrenbereiche, in einer Arena markieren und zum anderen einfache Linien-Folge-Verhalten realisieren, mit deren Hilfe das Finden einer Ladestation, bzw. das Ausrichten auf eine Ladestation vereinfacht wird.

RGB-Sensor

Der an der Vorderseite des Roboters befindliche RGB-Sensor ermöglicht, zusammen mit einem Berührungssensor und einer weißen LED, mehrere Messungen zugleich. Zum einen kann die Farbe eines sich direkt vor dem Roboter befindenden Objektes durch anleuchten mit der weißen LED festgestellt werden. Diese Messung wird erst vorgenommen, wenn sich ein Objekt genügend nah vor dem Sensor befindet. Diese Messung wird daher durch den Berührungssensor ausgelöst.

Zum anderen können mit Hilfe des RGB-Sensors die Intensität des Umgebungslichtes als auch die roten, grünen und blauen Anteile des Umgebungslichts gemessen werde. Damit kann der Roboter andere Roboter mit aktivierten RGB-Licht finden oder auch einfach in der Arena platzierte RGB-Quellen identifizieren.

Der Sensor basiert auf dem ADJD-S313[7] Farbsensor, welcher ebenfalls über TWI angeschlossen ist und viele Einstellungsmöglichkeiten, wie die Größe der Sensorfläche und der Integrationszeit jeweils für rot, grün und blau, besitzt.

Energie

Um während des Betriebs den Ladezustand der Akkus zu kontrollieren, wurde ein TWI-12-Bit-ADC eingebaut. Dieser misst kontinuierlich den Ladestand des Roboters und kann so konfiguriert werden, dass er bei einem bestimmten Messwert einen Interrupt auslöst.

Um zu erkennen, ob ein Ladevorgang gestartet wurde oder abgeschlossen ist, stellen die Ladeschaltkreise zwei weitere Interrupts zur Verfügung. Dies ist notwendig, um im Mikrosekundenbereich zu erkennen, ob sich die Schleifkontakte des Roboters auf einer Ladestation befinden. Nur über das Abfragen der ADCs ist dieses nicht möglich, da der Anfang des Ladenzyklus nicht durch ein zügiges Ansteigen der Werte am ADC zu erkennen ist.

Beschleunigungssensor

Der Roboter besitzt einen einfachen dreiachsigen Beschleunigungssensor, der über TWI angeschlossen ist. Die Idee zum Einbau eines Beschleunigungssensor besteht darin, dass es möglich ist Kollisionen mit Hindernissen

[7] http://www.avagotech.com

(Wände oder andere Roboter) zu erkennen, ohne eine Analyse der Abstandsdaten durchzuführen. Dies ist ein wichtiges Feedback für Lernalgorithmen, die auf dem Roboter implementierbar sind. Des Weiteren kann der Roboter hiermit auch seine Lage im Raum feststellen. Zur Unterstützung des odometrischen Systems kann der Sensor nicht herangezogen werden, da die Daten zu verrauscht sind, um die geringe Beschleunigung des Roboters zu messen.

3.3.4 Aktuatorik

Wie der Jasmine-Roboter besitzt Wanda einen differentiellen Antrieb. Geregelt wird er über einen einfachen PID-Regler basierend auf den Daten des Inkrementgebers. Der Geschwindigkeitsbereich ist ähnlich zum Jasmine.

Als weitere Aktuatoren kann man die RGB-LEDs ansehen, die rund um den Roboter angebracht sind. Diese RGB-LEDs lassen sich gleichzeitig in vier verschiedenen Modi betreiben: 100% An/Aus sowie zwei PWM-basierte Zuständen, bei denen die PWM-Frequenz und das Tastverhältnis eingestellt werden können. Dies wird vollständig durch den CAT9532 realisiert und belastet somit nicht den μC.

Das Chassis des Wanda besitzt einen passiven Backenschieber, mit dem sich Objekte von der Größe eines LEGO®-Duplo-Steins schieben lassen.

3.3.5 Besonderheit

Einige der Besonderheiten des Jasmine- und des I-SWARM-Roboters gingen in dem Wanda-Roboter auf. Besonders an dem Wanda ist, dass er, durch seinen sehr modularen Aufbau, mit nur drei der insgesamt fünf Boards betrieben werden kann. So reicht es für die Funktionsfähigkeit des Roboters aus, ihn nur aus Board ①, ② und ③ zusammenzusetzen. Das Chassis folgt ebenfalls diesem modularen Aufbau.

3.4 Zusammenfassung

Diese Kapitel beschreibt die in der Arbeit betrachteten Minatur- und Mikroroboter. Zunächst wird der I-SWARM-Roboter beschrieben, der die minimalen Anforderungen an die Softwarearchitektur definiert. Danach werden der Jasmine- und der Wanda-Roboter beschrieben, die sich zwar äußerlich in einigen Punkten ähneln doch bei genauerer Betrachtung sehr unterschiedliche Systeme darstellen.

Betrachtet man die drei Roboter, so unterscheiden sie sich in vielen Punkten. Unterschiedliche Arten von Sensoren, Aktuatoren und μCs stellen verschiedene Anforderungen, die in die Entwicklung der jeweiligen Betriebssystemen und einer plattformunabhängige Steuersprache mit einfließen müssen.

4. Entwicklungsumgebung

Eine Entwicklungsumgebung für Schwarmrobotikexperimente umfasst diverse Systeme, die in einem System integriert werden müssen, um nahtlos in einander zu greifen. Die im Rahmen dieser Arbeit entwickelte Entwicklungsumgebung besteht aus insgesamt drei Hauptteilen, nämlich die für die verwendeten Roboter entwickelten Roboterbetriebssysteme mit einer einheitlichen Steuersprache, eine interaktive Arena für wiederholbare Experimente und eine Simulationsumgebung. Jedes dieser Teile stellt an den Entwickler unterschiedliche Anforderungen.

In diesem Kapitel betrachten wir jedes der drei Hauptteile des entwickelten Gesamtsystems. In Abschnitt 4.1 wird analysiert, was für Anforderungen an die Steuersoftware, die einer solchen Entwicklungsumgebung zu Grunde liegt, gestellt werden. Danach wird in Abschnitt 4.2 eine aus den Anforderungen resultierende Steuersprache und deren Einbettung in die diversen Roboterbetriebsysteme beschrieben. In Abschnitt 4.3 wird auf die Simulationsumgebung und in Abschnitt 4.4 auf die interaktive Arena eingegangen.

4.1 Anforderungsanalyse für die Steuersoftware

Um Aussagen über die Anforderungen an die Steuersoftware zu machen, wird eine Analyse hinsichtlich mehrerer Kriterien durchgeführt. Zum einem wird betrachtet, in welcher Form Algorithmen für emergentes Verhalten von Schwärmen in der Biologie und der Robotik dargestellt werden. Viele Algorithmen lassen sich zum Beispiel rein mathematisch durch Formeln, aber auch graphisch anhand von Ablaufdiagrammen und Automaten, beschreiben. Die verschiedenen Darstellungen werden anhand ihrer Lesbarkeit und Umsetzbarkeit hinsichtlich der direkten Eingabe zur Steuerung eines Roboters bewertet. Zum anderen fließen Randbedingungen ein, die sich durch die betrachtete Hardware an sich ergeben. Nicht alle bekannten Steuerungen lassen sich eins-zu-eins auf einen Microcontroller umsetzen. Auch spielt die Nutzbarkeit der Darstellung als Genom für Evolutionary Computation in der vorliegenden Arbeit eine wichtige Rolle.

4.1.1 Anforderungen aus der Darstellung von Algorithmen

Betrachten wir die Modellierung des Verhaltens von Tieren in der Biologie, so wird in vielen Fällen ein einfacher Automat herangezogen, um das Verhalten darzustellen [38, 75, 28]. Die Zustände des Automaten geben an, in welchen Zuständen sich das Tier befindet, und die Übergänge geben an, in welcher Situation das Tier den Zustand wechselt. Für einzelne Tiere, Gruppen oder Schwärme werden die Übergänge oft auch mit Wahrscheinlichkeitswerten versehen.

Diese Darstellung von Verhalten als Automaten findet man auch in der Robotik [73]. Verhaltensbasierte Ansätze führen zumeist auf den Hirnforscher und Kybernetiker Braitenberg zurück, der in seiner Arbeit Gedankenexperimente mit fest verdrahtetem Verhalten durchgeführt hat. In seinem Buch [8] beschreibt er, wie mit einfachen Sensoren ausgestattete Vehikel selbstständig auf Umweltreize reagieren können. Es wird beschrieben, wie komplexes Verhalten schon durch sehr einfache Mechanismen zu erreichen ist. Braitenbergs Arbeit wurde von verschiedenen Robotikern aufgegriffen, die diese Ideen in Robotersteuerungen umsetzten. Hierbei nimmt Brooks mit seiner Subsumption-Architektur [14] eine wichtige Rolle ein.

Ein großer Vorteil einer Automaten ähnlichen Beschreibung ist ihre gute Lesbarkeit. Algorithmen, die so verfasst werden, sind leicht nachvollziehbar und verständlich. Dies steht im Gegensatz zu anderen gängigen Robo-

tersteuerungen wie Künstlichen Neuronalen Netzes [92, 93]. Hier wird das Verhalten des Roboters in den Gewichten des KNN versteckt. Es ist nur schwer nachvollziehbar, wie sich die Veränderung eines Netzes auf das Verhalten des Roboters auswirkt. Besonders erschwert wird dies in den heute häufig verwendeten *Rekurrenten Neuronalen Netzen (RNNs)*, welche zusätzliche Rückkoppelungen enthalten. RNNs haben jedoch den Vorteil, dass sie sich gut für die Evolution von Roboterverhalten nutzen lassen.

In der vorliegenden Arbeit ist ein Ziel, dass sich Roboterverhalten auch von einem Designer leicht erstellen lässt. Daher sind in unserem Fall KNNs keine gute Wahl zur Robotersteuerung. Es sollte jedoch immer die Möglichkeit bestehen, das Verhalten basierend auf KNNs einfach in die Steuerung zu integrieren.

4.1.2 Anforderungen durch Evolutionary Computation

Evolutionary Computation ist eine in der Schwarmrobotik weit verbreitete Technik, um Schwarmverhalten zu lernen. Dabei kommen verschiedene Arten aus dem Bereich der EC zur Anwendung. Gerade RNNs sind sehr beliebt und wurden in vielen Projekten als Genom verwendet. Ein Problem, das jedoch alle KNNs haben ist die Lesbarkeit bzw. Analysierbarkeit dieser, da die Wechselwirkungen der Gewichte aufeinander nur sehr schwer zu fassen sind. Die Darstellung als KNN steht somit auch im Widerspruch zu der Darstellung von Schwarmalgorithmen, wie sie im vorherigen Abschnitt beschrieben worden sind.

Eine weitere Unterart der EC ist die GP. Diese hat den Vorteil, dass man sie mit der Darstellung von Algorithmen in der Biologie durch Automaten leicht in Einklang bringen kann. Dabei lassen sich die evolutionär erzeugten Programme zumeist leichter lesen und analysieren. Je nach verwendetem Genom sind einzelne Teile eines Programms eindeutig einem Verhalten des Roboters zuweisbar.

Ein Problem, das durch die Verwendung von GP entstehen kann, ist das Wachstum der Steuerprogramme aufgrund von Programmteilen, die irrelevant für die Steuerung des Roboters sind, aber von Generation zu Generation vererbt und weiter kopiert werden. Dies ist nicht unbedingt von Nachteil für die Evolution, da diese Programmteile in späteren Generationen eventuell aktiv und nutzbar werden können. Dieses Wachstum führt zu einem größeren Verbrauch von Speicher, worauf beim Softwareentwurf geachtet werden muss.

EC und somit auch GP lassen sich in verschiedenen Ausprägungen realisieren, die sich in allen möglichen Kombinationen von online/offline in Kombination mit onboard/offboard ausdrücken lassen. Dabei bedeutet der Unterschied zwischen online und offline, dass der Roboter im Falle des Online-Lernens während der Ausführung einer Aufgabe den Controller anpasst und so lernt. Man spricht hier auch davon, dass die Exploration möglicher Lösungen während der Ausnutzung (engl. *exploitation*) der gefundenen Lösungen stattfindet. Im Offline-Fall wird die Explorations-Phase von der Exploitation-Phase getrennt.

Onboard bedeutet, dass die Evolution auf dem Roboter stattfindet, wohingegen bei offboard die Evolution in einer Simulationsumgebung auf simulierten Robotern passiert.

Das zu entwickelnde System sollte, mit nur minimalen Veränderungen, für alle vier möglichen Kombinationen dieser Spielarten ausgelegt sein. Dies ist nur möglich, wenn die Steuerprogramme zur Laufzeit austauschbar und zwischen den Robotern übertragbar sind.

4.1.3 Anforderungen aus der Hardware

Betrachtet man die in Kapitel 3 vorgestellten Roboter, so geht daraus direkt hervor, dass die Steuersoftware so plattformunabhängig wie möglich sein muss, damit sie sich leicht von einer Architektur auf die andere übertragen lässt. Dies hat zur Folge, dass man die Anforderungen auf den kleinsten gemeinsamen Nenner der Hardware bringen muss. Die größte Beschränkung stellt somit der Speicherbedarf der Software im RAM und im Programmspeicher dar. Der gesamte Speicherbedarf der Software auf dem Roboter darf also acht Kilobyte nicht überschreiten. Um dies zu gewährleisten, ist eine klare Trennung vom Steueralgorithmus und der Informationsakquise, also die Trennung vom *sense*- und *act*-Teil vom *plan*-Teil unabdingbar. Durch diese Trennung erhalten wir zum einen eine Schnittstelle zum Betriebssystem-Teil (*sense, act*) und zum anderen den Controller Teil (*plan*). Eine weitere Bedingung ergibt sich aus den schon am Anfang des Abschnitts 3 beschriebenen, je nach Hardware unterschiedlichen Arten der Informationsakquise, die den zeitlichen Ablauf von *Sense, Plan, Act (SPA)* stark beeinflussen, und die Abtrennung des *plan*-Teils bestärkt.

Durch den I-SWARM-Roboter wird die Zeit als zusätzliche Beschränkung dem System auferlegt, die zum Beschreiben des Programmspeichers benötigt wird. Diese verändert zwar nicht das Verhalten des Roboters, hat

aber eine großen Einfluss auf die Nutzbarkeit des gesamten Systems. Ein Debuggen von Steuerprogrammen auf dem Roboter ist nur begrenzt möglich, da die Übertragung des geänderten Controllers bis zu 45 Minuten benötigt.

4.1.4 Zusammenfassung und Analyse der Anforderungen

Nach der Analyse der Anforderung anhand verschiedener Gesichtspunkt in den vorherigen Abschnitten, lassen sich die Anforderungen an die Software wie folgt zusammenfassen:

1. Darstellung des Controllers als Automat (Biologie, Lesbarkeit),
2. Plattformunabhängigkeit (viele verschiedene Architekturen),
3. Speichereffizienz (μC mit wenig Speicher),
4. Trennung von Betriebsystem und Controller (viele verschiedene Architekturen),
5. Übertragbarkeit von Controllern (GP),
6. Wechsel der Controller während der Laufzeit (großer Schwarm, GP).

Ein interpreterbasierter Ansatz stellt für diese Anforderungen die wohl günstigste Lösung bezüglich der Plattformunabhängigkeit, des Speicherbedarfs, des Wechsels von Controllern während der Laufzeit und der Besonderheiten der GP dar. Die Plattformunabhängigkeit wird dadurch gewährleistet, dass sich ein Interpreter sehr leicht kapseln lässt und man eine einfache Schnittstelle für die Einbindung des Interpreters in den Betriebssystemteil zur Verfügung stellen kann. Das Wechseln eines interpretierten Programms ist in jedem Fall einfacher, da der auf dem Roboter laufende Code bestehen bleibt, und nur die Bereiche des Speichers überschrieben werden müssen, die den zu interpretierenden Anteil beinhalten. Ein Wechseln von Programmen zur Laufzeit ist bei vielen μCs nicht möglich, da der Flash-Speicher nicht während der Ausführung eines Programms beschrieben werden kann, und falls doch, besondere Rücksicht auf die verwendeten Speicherstellen genommen werden muss.

Klassischerweise wird für die GP LISP [61] verwendet. Allerdings ist der Umfang der LISP-Sprache so groß, dass sich kein einfacher plattformunabhängiger Interpreter finden lässt, der zusammen mit dem LISP-Programm

in die vorhandenen acht Kilobyte Programmspeicher des Roboters passt [119]. Daraus ergibt sich die Nutzung einer einfacheren Sprache zur Darstellung von hybriden Automaten als die geeignetste Lösung. MDL2ϵ stellt eine solche einfache Sprache dar. Sie wird in Abschnitt 4.2 eingehend beschrieben. Der Speicherbedarf einer interpreterbasierten Lösung dieser Sprache wird in Abschnitt 4.2.3 genauer betrachtet.

Im Besonderen reduziert sich durch die Verwendung eines Interpreters beim I-SWARM-Roboter auch die Zeit zum Wechseln des Controllers, da nur noch dieser Anteil hochgeladen werden muss. Dies ist nicht unerheblich, da sich die Ladezeit von maximal 45 Minuten auf bis zu 1,4 - 6 Minuten verringert.

Das sich aus der Anforderungsanalyse ergebende Blockdiagramm der Softwarestruktur ist in Abbildung 4.1 dargestellt. Auf unterster Ebene befindet sich die Abstraktion der Hardware, bzw. des simulierten Roboters, dann folgt die Schnittstelle zu SPA, auf der wiederum im Fall des Roboters der Interpreter mit dem Bytecode aufsetzt. Darüber liegt eine für die Simulation und den Roboter vereinigende Ebene, die zum einen den Bytecode erzeugt und zum anderen für den simulierten Roboter die gleiche Aufgabe wie der Interpreter übernimmt. Zuletzt kommt der eigentliche Controller, dessen Darstellung als Textdatei oder *eXtensible Markup Language (XML)*-Datei unabhängig von der unterliegenden Software gehalten wird. Eine klare Trennung zwischen plattformunabhängigen Teilen (oberhalb der zweiten Schicht) und Teilen die auf dem Roboter laufen, und der Teile, die auf einem Rechner laufen, ist deutlich erkennbar.

Diese grundlegende Struktur wird in den nachfolgenden Abschnitten weiter verfeinert und erklärt. Zunächst wollen wir jedoch im folgenden Abschnitte auf die MDL2ϵ und ihren Ablauf genauer eingehen.

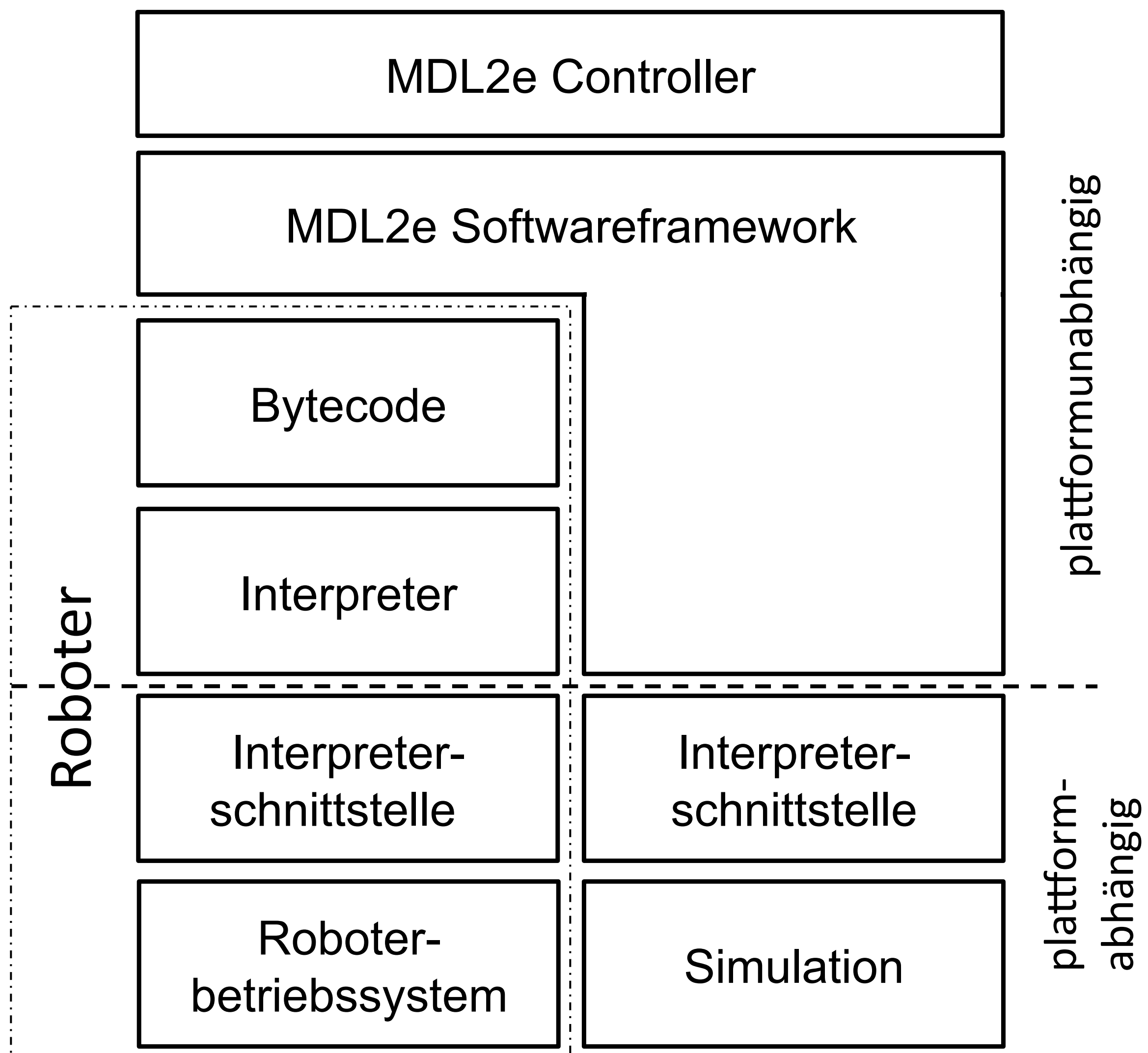


Abbildung 4.1: Blockdiagramm der Softwarestruktur des entwickelten Systems.

4.2 Die extended Motion Description Language 2

Die *extended Motion Description Language 2 (*MDL2ϵ*)* ist eine einfache Sprache zum Beschreiben von reaktiven bzw. verhaltensbasierten Robotersteuerprogrammen. Die extended Motion Description Language 2 geht auf die extended Motion Description Language von Manikonda et al. zurück [70], welche eine Erweiterung der ursprünglichen Motion Description Language von R. W. Brockett ist [11, 12, 13].

Die Sprache verbindet kontinuierliche Steuerung mit interrupt- bzw. event-basierten Ansätzen. Es findet eine direkte Koppelung von Sensoren mit Aktoren statt. Dieser Ansatz ist durch Brooks Sub-Sumption-Architektur [14] inspiriert. Er verbindet einen kontinuierlichen, auf Differentialgleichungen basierenden Regler, wie er aus der Reglungstechnik wohl bekannt ist, mit einem Zustandsautomaten. Diese Art der Regelung wird als hybride Regelung bezeichnet. Das Hinzufügen von sensorbasierten Interruptfunktionen und deren formalen Beschreibung war die grundlegende Ertweiterung von Brocketts *Motion Description Language (MDL)* zu *extended Motion Description Language (MDLe)*.

Die ursprüngliche Verwendung von MDLe bezieht sich auf die reine Beschreibung einer Bewegung. Sie ist hervorragend dazu geeignet, als Ausgabe für einen Pfadplaner zu dienen, der aufgrund der Einbindung von Sensordaten auch mit dynamisch auftretenden Problemen, bzw. Hindernissen zurecht kommt. Mit MDLe versuchen V. Manikonda et al. eine formale Beschreibung bereit zu stellen, um hybride Steueransätze auf mathematische Art fassbar und analysierbar zu machen.

Um MDL2ϵ für die Steuerung von autonomen Schwarmroboter zu verwenden, wurde MDLe im Rahmen der Dissertation erweitert.

Für MDL2ϵ gibt es verschiedene Darstellungsmöglichkeiten. Zum einen eine formale und zum anderen ein XML-basierte Darstellung, die dem Programmierer die Erstellung von Roboterverhalten vereinfacht. Betrachten wir zunächst die formale Darstellung von MDL2ϵ.

4.2.1 Formale Beschreibung der MDL2ϵ

Kinetischer Zustandsautomat

Mit MDL2ϵ steuern wir einen physisch vorhandenen Roboter ausgestattet mit einfachen Sensoren. Der Roboter kann durch einen *Kinetischen Zustandsautomaten (KZA)* modeliert werden.

Definition 4.1. (*Kinetischer Zustansautomat*).
Wir bezeichnen einen **Kinetischen Zustansautomat** (KZA) als das Sextupel $(\mathcal{U},\mathcal{X},\mathcal{Y},\mathcal{S},h,\Xi)$, wobei

$\mathcal{U} = X^{\infty}(\mathbb{R}^{+} \times \mathbb{R}^{p}; \mathbb{R}^{m})$ ist ein Raum von Steuerregeln,
$\mathcal{X} = \mathbb{R}^{N}$ ist der Zustandsraum,
$\mathcal{Y} = \mathbb{R}^{p}$ ist der Ausgaberaum,
$\mathcal{S} \subset \mathbb{R}^{k}$ ist der Raum der Sensordaten,
$h : \mathcal{X} \to \mathcal{Y}$ bildet den Zustandsraum auf den Ausgaberaum ab und
Ξ eine Menge von Interrupts, wobei
$\xi : \mathcal{S} \to \{falsch, wahr\}$ ein Interrupt ist.

Ein KZA wird durch Differentialgleichungen der folgenden Art beschrieben:

$$\dot{x} = \sum_{i=1}^{m} g_i(x)u_i$$

$$y = h(x) \in \mathbb{R}^{p}$$

wobei

$$\begin{aligned} x(\cdot) : \mathbb{R}^{+} = [0,\infty) &\to \mathbb{R}^{N} \\ u_i : \mathbb{R}^{+} \times \mathbb{R}^{p} &\to \mathbb{R} \\ (t,y(t)) &\mapsto u_i(t,y(t)) \end{aligned}$$

Weiter ist jedes g_i, $1 \leq i \leq m$, ein Vektorfeld in $\mathbb{R}^{N}$.

Die Modellierung der Umwelt findet zumeist in einer Simulationsumgebung statt. Eine Computer-Simulation stellt einen interativen Ansatz zur Lösung von Differentialgleichungen dar. Auf die verwendete Simulationsumgebung und die Modellierung der Roboter wird in Kapitel 4.3 eingegangen.

Atom

Betrachten wir die einzelnen Teile des KZA, so lassen sich zunächst zwei zur Steuerung des Roboters wichtige Teile herausziehen. Zum einen haben wir die Steuerregeln $\mathcal{U}$ und zum anderen die Menge Ξ der Interrupts. Diese werden in MDL2ϵ zu einem grundlegende Baustein, dem Atom, zusammengeführt. Ein Atom ist das kleinste unabhängige Ausführungsprimitiv. Mit Hilfe eines Atoms kann der interne sowie der externe Zustand des Roboters verändert werden.

Definition 4.2. (*Atom*).
Wir bezeichnen ein **Atom** von MDL2ϵ als das Tripel der Form $\sigma = (U,\xi^a,T^a)$ wobei

$$U = (u_1, \ldots, u_m)'$$

mit u_j wie vorher definiert, und

$$\begin{array}{rccc} \xi^a : & \mathbb{R}^k & \rightarrow & \{falsch, wahr\} \\ & s(t) & \mapsto & \xi^a(s(t)) \end{array}$$

ist eine Boole'sche Funktion. $T^a \in \mathbb{R}^+$ ist die maximale Ausführungsdauer eines Atoms und $s(\cdot) : [0,T] \rightarrow \mathbb{R}^K$ ist ein k-dimensionales Signal, das die Ausgabe von Sensor k repräsentiert.

ξ^a kann als Interrupt interpretiert werden, der durch eine Veränderung der Umwelt oder des internen Zustandes des Roboters ausgelöst wird. Zur Vollständigkeit der Definition eines Atoms definieren wir nun ein *skaliertes Atom*.

Definition 4.3. (*skaliertes Atom*).
Gegeben sei das Atom, $\sigma = (U,\xi,T^a)$, dann definieren wir

$$(\alpha U,\xi^a,\beta T^a),\ \alpha = (\alpha^1, \ldots, \alpha^m) \in \mathbb{R}^m,\ \beta \in \mathbb{R}^+$$

als das entsprechende **skalierte Atom** und schreiben es als $(\alpha,\beta)(U,\xi^a,T^a) = (\alpha,\beta)\sigma$.

Somit werden α zur Skalierung der Eingabe und β zur Skalierung der Zeit, die ein Atom ausgeführt werden soll, verwendet.

Alphabet

Ein Atom stellt ein Ausführungsprimitiv dar. Dies entspricht einem terminalen Zeichen von Sprachen, die mit MDL2ϵ beschrieben werden können. Somit definieren wir nun allgemein das Alphabet, auf dem MDL2ϵ operiert.

Definition 4.4. (*Alphabet*).
Ein **Alphabet** Σ ist eine endliche Menge von unabhängigen Atomen, das heißt der Tripel (U,ξ^a,T^a). Somit ist $\Sigma = \{\sigma_1,\ldots,\sigma_n\}$ für ein festes n, wobei σ_i ein Triple der Form (U_i,ξ_i^a,T_i^a) ist, so dass $\sigma_i \neq (\alpha,\beta)(\sigma_j)$, für ein $\alpha \in \mathbb{R}^m$, $\beta \in \mathbb{R}^+$ und $1 \leq i \leq n$, $i \leq j \leq n$.

Somit ist ein Alphabet eine Menge von Atomen, die nicht von einander durch Skalierung abgeleitet werden können. Man kann dieses Alphabet auch als minimales Alphabet bezeichnen. Im Gegensatz hierzu steht das *erweiterte Alphabet*, welches die Menge aller möglichen skalierten Atome umschließt.

Definition 4.5. (*erweitertes Alphabet*).
Ein **erweitertes Alphabet** Σ_ϵ ist die unendliche Menge von skalierten Atomen, z.B. der Tripel $(\alpha U_i,\xi_i^a,\beta T_i^a) = (\alpha,\beta)\sigma_i$, $\alpha \in \mathbb{R}^m$, $\beta \in \mathbb{R}^+$, $1 \leq i \leq n$ abgeleitet vom Alphabet Σ.

Sprache

Mit MDL2ϵ können wir Sprachen definieren, die bestimmte Abfolgen von Atomen beschreiben.

Definition 4.6. (*Sprache*).
Eine **Sprache** $L(G)$ ist definiert als eine Menge von Zeichenketten auf einem festen Alphabet Σ, welche durch einen Grammatik $G = (V,\Sigma,P,S)$ abgeleitet werden:

$$L(G) = \{w \in \Sigma^* | S \Rightarrow_G^* w\},$$

wobei

V ist die Mege der Variablen,
Σ ist das Alphabet,
P sind die Produktionsregeln,
und S ist das Startsymbol.

Hierbei ist Σ^* die Menge aller Wörter, die sich durch die Konkatenation von Symbolen aus Σ bilden lassen. $\Rightarrow_G^*$ ist die reflexive und transitive Hülle von $\Rightarrow_G$.

Zur Vollständigkeit definieren wir eine *erweiterte Sprache* $L_\epsilon(G)$ auf dem Alphabet Σ_ϵ als:

Definition 4.7. (*erweiterte Sprache*).

Eine **erweiterte Sprache** $L_\epsilon(G)$ ist eine Sprache, deren Alphabet ein *erweitertes Alphabet* Σ_ϵ ist. Sie wird durch die Grammatik $G = (V, \Sigma_\epsilon, P, S)$ erzeugt.

Für eine solche Sprache $L_\epsilon(G)$ gibt es verschiedene Möglichkeiten die Grammatik G darzustellen. Zum einen durch die Angabe der Produktionregeln, der Form $S \rightarrow \sigma S$, $S \rightarrow S$, ..., durch einen *nichtdeterministischen Automaten* oder durch einen *regulären Ausdruck*.

Zur Angabe der Grammatik in MDL2ϵ wird die Darstellung als regulärer Ausdruck verwendet. Hierbei werden alle bekannten Operatoren von regulären Ausdrücken angewendet. Diese werden jedoch teilweise MDL2ϵ-spezifisch umdefiniert, um den Anforderung an eine Steuersprache gerecht zu werden.

Definition 4.8. (MDL2ϵ-*Ausdruck*).

Ein MDL2ϵ-Ausdruck ist wie folgt induktiv definiert:

1. Die leere Menge $\emptyset$ ist ein MDL2ϵ-Ausdruck.
2. Das leere Wort ϵ ist ein MDL2ϵ-Ausdruck.
3. Für jedes $\sigma \in \Sigma_\epsilon$ ist σ ein MDL2ϵ-Ausdruck.
4. Wenn α und β MDL2ϵ-Ausdrücke sind, dann auch $\alpha\beta$, $(\alpha|\beta)$ sowie $(\alpha)^n$ mit $n \in \mathbb{N}$, für $n = \infty$ schreiben wir wie im herkömmlichen regulären Ausdruck den Kleen-Stern $(\alpha)^*$.
5. Wenn α ein MDL2ϵ-Ausdruck ist, dann auch (α, ξ^b, T^b) mit ξ^b einem Interrupt und T^b einem Timer.
6. Wenn α_i, $1 \leq i \leq n$, $n \in \mathbb{N}$, MDL2ϵ-Ausdrücke sind, dann auch $(\alpha_1|\alpha_2|\ldots|\alpha_n,(p_{\alpha_1},p_{\alpha_2},\ldots,p_{\alpha_n}))$, wobei p_{α_i} die Auswahlwahrscheinlichkeit für α_i ist.

Die Punkte 1. – 4. entsprechen der Definition eines regulären Ausdrucks,

wobei in 4. der Kleen-Stern durch $n \in \mathbb{N}$ ersetzt worden ist, dies entspricht aber ohne Beschränkung der Allgemeinheit einem n-maligen Hintereinanderschreibens des selben Teilausdrucks, bzw. im Fall der unendlichen Wiederholung dem Kleen-Stern. In einem MDL2ϵ-Ausdruck können somit Teilausdrücke n-mal wiederholt werden. Wir bezeichnen diesen Operator als *Multiplicity* oder auch *Mult*.

Die Punkte 5. und 6. in der Definition beschreiben zwei wichtige Eigenschaften von MDL2ϵ. Punkt 5. führt ein sogenanntes Verhalten ein.

Definition 4.9. (*Verhalten*).
Ein **Verhalten** π ist ein Element (d.h. ein Wort) der Erweiterten Sprache $L_\epsilon(G)$, mit einem Timer T^b und einem Interrupt ξ^b.

Somit definiert die Grammatik G eine Verhaltensstrategie $\Pi(G)$ für den Roboter. Ein *atomares Verhalten* ist ein Verhalten, das nur ein einziges Atom mit $\xi^a = \xi^b$ und $T^a = T^b$ hat.

Ausgehend von einem Verhalten lassen sich die Länge und die Dauer eines Verhaltens definiere.

Definition 4.10. (*Länge eines Verhaltens*).
Die Länge eines Verhaltens $|\pi|$ ist die Anzahl an Symbolen aus Σ_ϵ in dem Verhalten.

Definition 4.11. (*Dauer eines Verhaltens*).
Die Dauer $T(\pi)$ eines Verhaltens

$$\pi = (\alpha_1, \beta_1)(U_1, \xi_1, T_1) \ldots (\alpha_l, \beta_l)(U_l, \xi_l, T_l),$$

dessen Ausführung zum Zeitpunkt t_0 beginnt, ist das Minimum über die Summe der Zeitintervalle, für die jedes der Atome in dem Verhalten ausgeführt worden ist und der Zeit die das Verhalten maximal ausgeführt werden kann.

$$T(\pi) = \min[(\min[\hat{T}_1^a, \beta_1, T_1^a] + \ldots + \min[\hat{T}_n^a, \beta_l, T_n^a]), T^b].$$

Punkt 6. in Definition 4.8 beschreibt eine ODER-Beziehung bzw. Auswahl, deren Wahrscheinlichkeitsverteilung zusätzlich angegeben ist und von der Gleichverteilung abweicht. Im Gegensatz dazu ist die ODER-Beziehung in Punkt 4. abhängig von der Reihenfolge der Konkatenation der MDL2ϵ-Ausdrücke.

Aufbauend auf den Definitionen 4.1 – 4.9 lässt sich nun ein MDL2ϵ-Plan definieren.

Definition 4.12. (*Plan*).
Gegeben sei ein KZA und ein Weltmodell. Ein **Plan** Γ ist definiert als die geordnete Sequenz von Verhalten π_i, $1 \leq i \leq n \in \mathbb{N}$, deren Ausführung das gegebene Ziel erfüllt. Die Länge eines Plans Γ ist $|\Gamma| = \sum_i |\pi_i|$ und die Dauer eines Plans ist $T(\Gamma) = \sum_i T(\pi_i)$.

Aktivität und Gültigkeit

Nachdem wir MDL2ϵ formal beschrieben haben, geben wir einige weitere Definitionen, die die Arbeit mit MDL2ϵ und das Verständnis vereinfachen. So ist es sinnvoll, die Begriffe *gültiger* bzw. *aktiver Interrupt*, *gültiges Atom* bzw. *gültiges Verhalten* und *aktives Atom* einzuführen.

Definition 4.13. (*gültiger Interrupt*).
Ein Interrupt ξ^γ, $\gamma \in \{a,b\}$, heißt *gültig* zum Zeitpunkt t, wenn

$$\xi^a(s(t)) \equiv wahr$$

gilt.

Das heißt, es gibt eine Belegung von Sensordaten, so dass die Interruptfunktion *wahr* ist. Die Menge aller gültigen Interrupts bezeichnen wir mit Ξ_v,

$$\forall \xi_i \in \Xi_v \subset \Xi : \xi_i(s(t)) \equiv wahr.$$

Alle Verhalten bzw. Atome mit gültigem Interrupt bezeichnen wir als *gültige Verhalten* bzw. *gültige Atome*.

Im Gegensatz dazu ist ein Interrupt ξ^a *aktiv* genau dann, wenn er gültig ist und das ihm zugehörige skalierte Atom gerade ausgeführt bzw. gesschrieben wird. Das Atom bezeichnen wir dann als *aktives Atom*. Trivialerweise

kann es zu einem Zeitpunkt t nur ein aktives Atom geben.

Definition 4.14. (*aktiver Interrupt bzw. aktives Verhalten*).
Ein *aktives Verhalten* bzw. *aktiver Interrupt* läßt sich wie folgt induktiv definieren:

- sei σ ein aktives Atom und gilt $\sigma \subset \pi_v$, d.h. σ ist ein Teil des gültigen Verhaltens π_v, dann bezeichnen wir auch π_v und dessen Interrupt als *aktiv*.
- sei π_a ein aktives Verhalten und gilt $\pi_a \subset \pi_v$, d.h. π_a ist ein Teilwort des gültigen Verhaltens π_v, dann bezeichnen wir auch π_v und dessen Interrupt als *aktiv*.

Alle anderen MDL2ϵ-Operatoren, wie $(\alpha|\beta)$, $(\alpha|\beta,(p_\alpha,p_\beta))$ und $(\alpha)^n$ gelten, da sie keine Interrupts besitzen, immer als gültig. Als aktive gelten sie jedoch nur, wenn das durch sie produzierte Teilwort ein aktives Atom enthält.

Mit Hilfe dieser Definitionen lässt sich der Oder-Operator mit und ohne Wahrscheinlichkeitsverteilung aus Definintion 4.8 einfach beschreiben. Beim Standard |-Operator, der sogenannten Union, wird der erste[1] gültige MDL2ϵ-Ausdruck aus der Liste der Union ausgewählt.

Der |-Operator mit Angabe einer Wahrscheinlichkeitsverteilung, der sogenannten *Random Union* oder auch *RUnion*, wählt anhand der Wahrscheinlichkeitsverteilung einen MDL2ϵ-Ausdruck aus der Liste. Ist dieser nicht gültig so wird er verworfen. Es findet jedoch keine weitere Auswahl statt.

Spur

Für eine Laufzeitanalyse von Plänen ist es wichtig zu wissen, welche Atome besucht worden sind. Hierdurch lassen sich die zur Laufzeit erzeugten Worte der Sprache statistisch reproduzieren, ohne sie im Gesamten aufzuzeichnen.

[1] Im regulären Ausdruck von links nach rechts gelesen der erste Ausdruck.

Wir erweitern hierfür das skalierte Atom um einen Zähler S^a zu einem *Atom mit Spur.*

Definition 4.15. (*Atom mit Spur*).

Gegeben sei das skalierte Atom $(\alpha,\beta)(U,\xi^a,T^a)$, dann definieren wir

$$(\alpha U,\xi^a,\beta T^a, S^a),\ \alpha = (\alpha^1,\ldots,\alpha^m) \in \mathbb{R}^m,\ \beta \in \mathbb{R}^+,\ S^a \in \mathbb{R}$$

als das **Atom mit Spur** S^a und schreiben es als $(\alpha,\beta)(U,\xi^a,T^a,S^a)$.

Die Anpassungsregel der Spur S^a für ein aktives Atom $\sigma(t)$ zum Zeitpunkt t ist:

$$\sigma(t) = (\alpha,\beta)(U,\xi^a,T^a,S^a(t-1) + \gamma(t)). \tag{4.1}$$

Dabei ist γ ein zeitlich veränderbarer oder auch konstanter Faktor. Basierend auf einem Atom mit Spur kann die Spur eines Plans definiert werden.

Definition 4.16. (*Spur eines Plans*).

Mit der Spur(Γ, δ_{prune}) eines Plans Γ bezeichnen wir einen Plan, in dem alle Atome σ_i *Atome mit Spur* sind und für die gilt, dass

$$S_i^a > \delta_{prune}.$$

Dabei ist $\delta_{prune} \in \mathbb{R}$, eine Schwellwert.

Die Spur eines Plans berechnet man rekursiv mit dem folgenden Algorithmus:

1. Entferne alle Atome für die gilt Spur $S^a \leq \delta_{prune}$.

2. Entferne alle leeren klammernden Operatoren.

3. Wiederhole 2. bis keine leeren klammernden Operatoren übrig sind.

Beispiel

Betrachten wir nun ein paar Beispiele. Zunächst definieren wir einen einfachen KZA der mit dem folgenden Differentialgleichungssystem beschrieben wird:

$$\begin{aligned} \dot{x} &= v_t \cos\theta, \\ \dot{y} &= v_t \sin\theta, \\ \dot{\theta} &= v_r, \end{aligned}$$

wobei $(x,y) \in \mathbb{R}^2$ die Position und θ die Ausrichtung des Roboters im Weltkoordinatensystem ist. v_t und v_r sind die Translations- und die Rotationsgeschwindigkeit des Roboters. Somit gilt für die Steuervariablen $u_1 = v_t$ und $u_2 = v_r$.

Betrachten wir zunächst einen einfachen Interrupt ξ_E, der gültig wird, wenn die Entfernung δ zwischen dem Roboter und einem Hindernis im Dedektionsbereich des Sensors einen bestimmten festen Wert unterschreitet. Der Dedektionsbereich kann durch den Öffnungswinkel α des Sensors in Abhängigkeit zur Sensorhauptaches $\mathbf{x}_s$ im Roboterkoordinatensystem angegeben werden. Zum Beispiel:

$$\xi_E = \begin{cases} falsch & \text{für } \delta > 20 \\ wahr & \text{für } \delta \leq 20 \end{cases} \quad \text{mit } \mathbf{x}_s = (1{,}0),\ \alpha = 180^\circ.$$

Hieraus lassen sich verschiedene Atome ableiten

$$\begin{aligned} \sigma_1 &= ((1{,}0), \neg\xi_E, 1), \\ \sigma_2 &= ((0{,}1), \xi_E, 1) \text{ und} \\ \sigma_3 &= ((0{,}1), \xi_{wahr}, 1), \text{ wobei } \xi_{wahr} \equiv wahr \text{ gilt.} \end{aligned}$$

Diese Atome können zu einem Plan zusammengesetzt werden, um ein Ausweichverhalten des Roboters zu erreichen. Zwei einfache Pläne in MDL2ϵ sind:

$$\Gamma_1 = \{(2,\infty)\sigma_1(1,10)\sigma_2,\infty\},$$

$$\Gamma_2 = \{(2,\infty)\sigma_1(1,10)\sigma_3,\infty\}.$$

Der Plan Γ_1 definiert ein Ausweichverhalten, bei dem der Roboter solange mit der Geschwingikeit $v_t = 2$ vorwärts fährt, bis der negierte Interrupt $\neg\xi_E$ "falsch" wird, $\xi_E = wahr$, was ein Hindernis anzeigt. Danach wird für eine maximale Dauer von zehn Zeitschritten das Atom σ_2 mit einer Geschwindigkeit von $v_r = 1$ aktiv, was eine Drehung bewirkt. Hat sich der Roboter vom Hindernis weggedreht wird wieder das Atom σ_1 aktiv.

Plan Γ_2 bewirkt ein ähnliches Verhalten des Roboters. Allerdings dreht hier der Roboter immer für zehn Zeitschritte mit einer Rotationsgeschwindigkeit von 1.

Ein sogenanntes *Random-Walk-Verhalten* mit Kollisionsvermeidung erhält man mit dem folgenden Plan:

$$\pi_{Zufall} = (((2,\infty)\sigma_1 \mid (1,10)\sigma_2 \mid (-1,10)\sigma_2,(0.\bar{3},0.\bar{3},0.\bar{3})),\neg\xi_E,\infty)$$

$$\Gamma_r = \{\pi_{Zufall}\ (2,10)\sigma_1,\infty\}$$

4.2.2 MDL2ϵ in der eXtensible Markup Language

Die *eXtensible Markup Language (XML)* ist eine Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten in Form von Textdaten. XML wird u. a. für den plattform- und implementationsunabhängigen Austausch von Daten zwischen Computersystemen insbesondere über das Internet eingesetzt [9].

Daher werden zur einfacheren Programmierbarkeit die Pläne im Allgemeinen als XML-Datei angelegt, welche dann von der Hauptapplikation entweder zur Simulation oder zur Steuerung des Roboters verwendet werden.

XML hat den Vorteil, dass sie seit Jahren als Metasprache weltweit akzeptiert ist und viele Parser für diese Sprache existieren, was die Implementierung einer auf XML basierten Applikation vereinfacht. Hinzu kommen Vorteile, wie die Möglichkeit der inhaltlichen und strukturellen Einschränkung von anwendungsspezifischen Sprachen, die anhand von Schemasprachen wie *Document Type Definition (DTD)* oder XML-Schema vorgenommen werden kann. Dies ermöglicht die Validierung eines MDL2ϵ-XML-Plans vor der Ausführung desselben und trägt somit zur Stabilität des gesamten Systems bei.

Das MDL2ϵ-XML-Format folgt der Syntax, die in der formalen Beschreibung angegeben wurde. So wird ein Atom durch die Attribute name, interrupt, duration und arg0 bis arg19 bestimmt. Hierbei definiert das name-Attribut den gewählten Steuervektor U, wobei die Attribute arg0 - arg19 den Skalierungsvektor α darstellen. Das duration-Attribut entspricht dem zeitlichen Skalierungsfaktor β. Ein Atom hat in XML somit die Folgende Form:

```
<ATOM name="<string>" interrupt="<string>" arg0="<const|var>"
   ... arg19="<const|var>" duration="<long>|'infinite'"
   probability="<const|var>"/>
```

Ein Verhalten hat die gleichen Attribute wie das Atom und besteht in der Regel aus einem öffnenden und schließendem XML-Tag. Im Plan an früherer Stelle definierte Verhalten können durch Wiederverwenden der selben Namens-Tags referenziert werden. Bei dieser Art der Referenzierung kann ein neuer Interrupt für das Verhalten angegeben werden. Das Verhalten wird dann wie das Atom als Solo-Tag geschrieben. Der XML-Ausdruck für ein Verhalten ist

```
<BEHAVIOUR name="<string>" interrupt="<string>"
   duration="<const>|'infinite'" probability="<const|var>">
  ...
```

```
</BEHAVIOUR>
```

und für ein referenziertes Verhalten

```
<BEHAVIOUR name="<name>" interrupt="<string>" probability="<
   const|var>"/>
```

Im Gegensatz zur formalen Darstellung werden Unions und Random Unions in der XML-Darstellung auch als Klammern betrachtet. Somit ergibt sich für eine einfache UNION

```
<UNION name="<string>" probability="<const|var>">
  ...
</UNION>
```

und für die RUNION

```
<RUNION name="<string>" probability="<const|var>">
  ...
</RUNION>
```

Die Wahrscheinlichkeitsverteilung für die zufällige Auswahl in der RUNION wird anhand von Wahrscheinlichkeitswerten berechnet, die als probability-Attribute den einzelnen XML-Tags hinzugefügt werden. Die Wahrscheinlichkeiten p_{α_i}, der durch den RUNION-Tag eingeschlossenen XML-Tags (*Kindknoten*), wird durch die Normierung aller Wahrscheinlichkeitswerte v_{α_i} auf eins berechnet.

$$p_{\alpha_i} = \frac{v_{\alpha_i}}{\sum_{k=0}^{N-1} v_{\alpha_k}}, \tag{4.2}$$

wobei N die Anzahl der Kindknoten der RUNION ist, v_{α_i} der im probability-Attribut für den Kindknoten α_i, $0 \leq i \leq N-1$, angegebene Wahrscheinlichkeitswert.

Die Multiplicity wird analog definiert. Die Anzahl der Wiederholungen wird in dem multiplicity-Attribut spezifiziert.

```
<MULT name="<string>" multiplicity="<const>|'infinite'"
   probability="<const|var>">
  ...
</MULT>
```

Wärend der Arbeit mit MDL2ϵ hat sich herausgestellt, dass es von Vorteil ist, wenn Atome und Verhalten aktiv bleiben, auch wenn der ihnen zugeordnete Interrupt nicht mehr gültig ist. Dies wurde in früheren Versionen von MDL2ϵ durch die Einführung von Semaphoren erreicht, die dem eigentlichen Interrupt über ein Oder hinzugefügt wurden und somit das Verhalten/Atom künstlich aktiv hielten. Da dieses Verfahren sehr unübersichtlich

war und leicht zu Programmierfehlern führte, wurden das SBEHAVIOUR und SATOM eingefügt. In einem SATOM/SBEHAVIOUR muss der Interrupt nur während der Aktivierung gültig sein. Danach wird der zugeordnete Interrupt immer als *wahr* angesehen.

```
<SATOM name="<string>" interrupt="<string>" arg0="<const|var>"
   ... arg19="<const|var>" duration="<long>|'infinite'"
   probability="<const|var>"/>

<SBEHAVIOUR name="<string>" interrupt="<string>"
   duration="<long>|'infinite'" probability="<const|var>">
  ...
</SBEHAVIOUR>
```

Eine vollständige Beschreibung mit der gängigen Standardbelegung von XML-MDL2ϵ findet man als DTD-Skript in Anhang A.

Darstellung der Interruptfunktion ξ_i^a in XML

In XML-MDL2ϵ unterscheiden wir zwischen drei Arten von Interrupts. Zum einen gibt es atomare oder auch einfache Interrupts, die im System fest vorgegeben sind, zum anderen Interrupts, die aus dem Vergleich von Konstanten/Variablen miteinander entstehen (*Vergleichsinterrupt*), sowie zusammengesetzte Interrupts, die durch eine Boole'sche Funktion aus einfachen und Vergleichsinterrupts entstehen. Die Interrupts werden anhand eines Beispiels im Weiteren erläutert.

Betrachten wir den Interrupt ξ_E aus dem vorherigen Abschnitt, so wurde dieser definiert als:

$$\xi_E = \begin{cases} falsch & \text{für } \delta > 20 \\ wahr & \text{für } \delta \leq 20 \end{cases} \text{ mit } \mathbf{x}_s = (1{,}0),\ \alpha = 180^\circ.$$

Der Interrupt ξ_E lässt sich nun zum einen als einfacher Interrupt im System fest implementieren und mit z.B. ITOUCH bezeichnen. Zum anderen kann er auch direkt über den Abstand dargestellt werden, sofern ein Variable VDISTANCE im System existiert, die den Wert δ beschreibt. Der Interrupt wäre dann GE(20,VDISTANCE), wobei GE für größergleich steht. Ein zusammengesetzter Interrupt ist z.B. AND(GE(VDISTANCE, 10), ITOUCH). Dieser Interrupt beschreibt einen Abstandsbereich zwischen 10 und 20. Die Produktionsregel für Interrupts ist in Tabelle 4.1 angegeben. Die angegebene Menge der einfachen Interrupts und der Variablen ist nicht vollständig, was durch "..." angedeutet werden soll. Das Terminal CONSTANT stellt im Gegensatz zu den Variablen eine konstante Zahl dar, die in XML

ausgeschrieben wird, wie es auch im obigen Beispiel zu sehen ist. Auf die Produktionsregeln für eine Zahl wurde zu Gunsten der besseren Lesbarkeit verzichtet.

G	=	(T, N, P, S)
T	=	{CONSTANT, EQ, GEQ, GT, NOT, AND, OR, ITRUE, ITOUCH, ISPACEL, IOBSTACLE, ..., VTOUCH, VENERGY, VSENSOR0, VSENSOR1, ...}
N	=	$\{I_{exp}, B_{int}, B_{val}, R_{op}, L_{op}\}$
P	=	{

S	$\rightarrow$	I_{exp}
I_{exp}	$\rightarrow$	NOT (I_{exp}) | L_{op} (I_{exp}, I_{exp}) | B_{int} | R_{op} (B_{val}, B_{val})
B_{int}	$\rightarrow$	ITRUE | ITOUCH | ISPACEL | ... | IOBSTACLE
B_{val}	$\rightarrow$	VTOUCH | VENERGY | VSENSOR0 | VSENSOR1 | ... | CONSTANT
R_{op}	$\rightarrow$	EQ | GEQ | GT
L_{op}	$\rightarrow$	AND | OR

}

Tabelle 4.1: Produktionsregeln für einen MDL2ϵ-Interrupt mit beispielhaften Variablen und einfachen Interrupts.

Die XML-Repräsentationen der in Abschnitt 4.2.1 beschriebenen Pläne Γ_1, Γ_2 und Γ_r sind in den Listings 4.1, 4.2 und 4.3 aufgeführt. Hierbei wurde der Steuervektor U durch leicht verständliche Namen, wie AMOVE $\hat{=}\ U = (1,0)$, AROT_L $\hat{=}\ U = (0,1)$, mit $\alpha_1 > 0$, und AROT_R $\hat{=}\ U = (0, -1)$, mit $\alpha_1 < 0$, ersetzt.

Listing 4.1: Plan Γ_1 in XML.

```
<PLAN name="Gamma_1" duration="infinite">
  <ATOM name="AMOVE" interrupt="NOT(ITOUCH)" arg0="2"
     duration="infinite"/>
  <ATOM name="AROT_L" interrupt="ITOUCH" arg0="1"
     duration="10"/>
</PLAN>
```

Listing 4.2: Plan Γ_2 in XML.

```
<PLAN name="Gamma_2" duration="infinite">
  <ATOM name="AMOVE" interrupt="NOT(ITOUCH)" arg0="2"
     duration="infinite"/>
  <ATOM name="AROT_L" interrupt="ITRUE" arg0="1"
     duration="10"/>
</PLAN>
```

Listing 4.3: Plan Γ_r in XML.

```
<PLAN name="Gamma_r" duration="infinite">
  <RUNION>
    <ATOM name="AMOVE" interrupt="NOT(ITOUCH)" arg0="2"
       duration="infinite" probability="1"/>
    <ATOM name="AROT_L" interrupt="ITOUCH" arg0="1" duration="
       10" probability="1"/>
    <ATOM name="AROT_R" interrupt="ITOUCH" arg0="1" duration="
       10" probability="1"/>
  </RUNION>
  <ATOM name="AMOVE" interrupt="NOT(ITOUCH)" arg0="2"
     duration="infinite"/>
</PLAN>
```

4.2.3 Entwicklung der MDL2ϵ Software

Nach der Auswahl und Beschreibung von MDL2ϵ als Robotersteuersprache ist die Implementierung einer Anwendung, die MDL2ϵ in einer Simulationsumgebung und auf einem Roboter verwendbar macht, ein Kernpunkt dieser Arbeit. Hierbei wurde aufgrund der Analyse in Abschnitt 4.1 von Anfang an Wert auf einfache Übertragbarkeit, Plattformunabhängigkeit und einen möglichst geringen Speicherbedarf auf dem verwendeten Roboter gelegt.

Die entwickelte Anwendung besteht aus zwei Teilen, die sich schon logisch aus den zu Grunde liegenden, unterschiedlichen Hardwarearchitekturen ergeben. Der eine Teil läuft auf einem herkömmlichen PC (Linux, Mac OS X, MS Windows) und enthält die Anwendungsteile, die für Ausführung von MDL2ϵ in Simulationsumgebungen und auf Robotern gedacht sind, die mit einem Linux ausgestattet sind. Dieser Teil beinhaltet einen XML-Parser zum Einlesen der XML-MDL2ϵ-Pläne, sowie einen Scheduler zum Ausführen der Pläne. Werkzeuge, wie das Framework for Learning and Self-Organisation, oder Werkzeuge zur graphischen Darstellung von MDL2ϵ-Plänen sind hier vorhanden. Diesen Teil der Software werden wir im Weiteren mit *MDL2ϵ-Applikation (MDL2ϵApp)* bezeichnen.

Der erste Teil dient auch als Schnittstelle zum zweiten Teil, der sich auf dem Roboter befindet und einen Scheduler zur Ausführung von MDL2ϵ auf dem Roboter implementiert. Der zweite Teil besteht, je nach vorhandenem Speicher, aus einem MDL2ϵ-Bytecode Interpreter mit Scheduler oder nur aus einem Scheduler. Der Scheduler wird zusammen mit einem von der Anwendung auf dem PC erzeugten MDL2ϵ-Plan erzeugt und kompiliert. Kommt auf dem Roboter der Interpreter zur Anwendung, wird der XML-MDL2ϵ-Plan in einen Bytecode umgewandelt, welcher dann auf dem Roboter interpretiert wird. Der Interpreter wurde auf den I-SWARM-Roboter, den Jasmine-Roboter, den Symbrion/Replicator Roboter sowie auf den Wanda-Roboter portiert. Ferner lässt er sich auch für einen normalen PC kompilieren.

In beiden Teilen wird das gleiche Konzept für die Schnittstelle in Ebene zwei (vgl. Abbildung 4.1) verwendet. Diese Schnittstellen besteht aus der durch MDL2ϵ vorgegebenen Einteilung in Atome, Interrupts und Variablen. In MDL2ϵApp werden diese durch einzelne Objekte realisiert, die von der MDL2eFactory-Klasse anhand der eindeutigen Namen aus der XML-Datei erzeugt werden und durch den MDL2eScriptParser in einen Objektbaum geschrieben werden. Die MDL2eFactory-Klasse folgt dem Entwurfsmuster der *Abstrakten Fabrik* [40]. Hierdurch wird die Applikation von der Erzeugung entkoppelt. Dies ermöglicht das einfache Einbinden verschiedener Simulation, da hier nur die konkrete Fabrikklasse implementiert werden muss, die die simulationsabhängigen Objekte erzeugt. Der MDLeScheduler kann dann direkt auf dem Objektbaum arbeiten und den MDL2ϵ-Plan abarbeiten. Abbildung 4.2 zeigt die Schnittstelle bestehend aus MDLeExecuteable (Atom), MDLeVariable und MDLeInterrupt.

Auf dem Roboter wird das gleiche Verhalten durch Funktionszeiger erreicht, die anhand eines Preprozessor-Makros in den Ablauf eingebunden werden. Hier existiert eine einfache Header-Datei, die die Namen aus der XML-Datei über das #define-Makro abbildet. MDL2ϵApp analysiert den aus der XML-Datei erzeugten Objektbaum und wandelt diesen in eine Bytecodedarstellung um, welche in Abschnitt 4.2.5 beschrieben wird.

Implementiert man auf dem Roboter ein neues Atom (Variable, Interrupt) mit einem eindeutigen Namen, so kann man dieses auch in der Simulationsumgebung implementieren und ihm in der Fabrikklasse denselben Namen zuweisen. Somit lässt sich derselbe Plan sowohl in der Simulation, als auch auf dem Roboter nutzen.

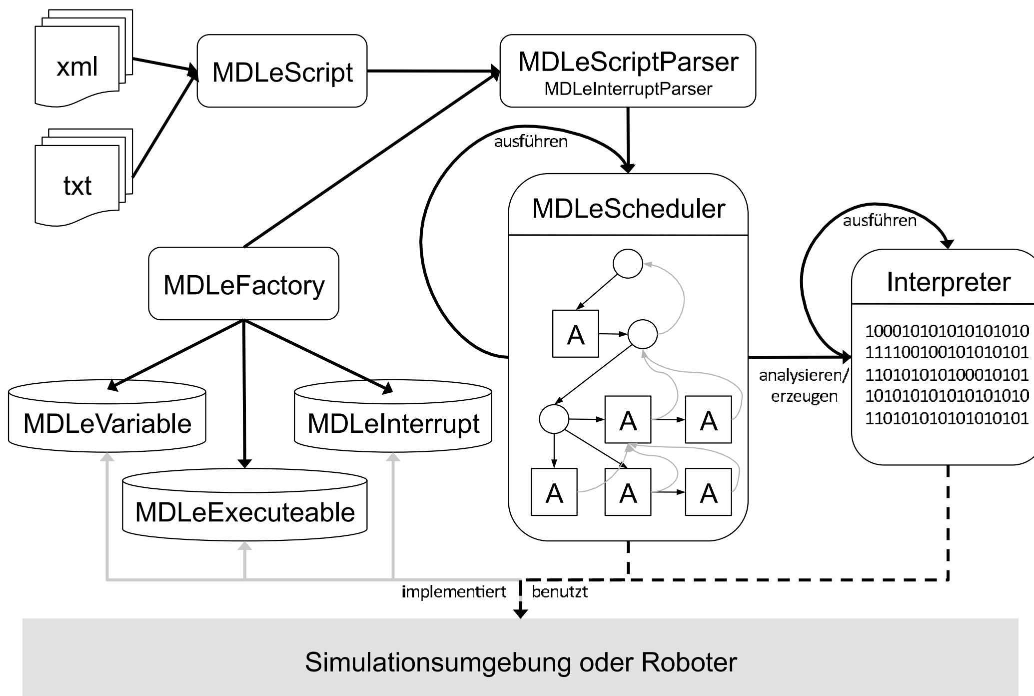


Abbildung 4.2: Schnittstelle zu MDL2ϵApp.

4.2.4 MDL2ϵ Ausführungsstapel

Die Ausführung von MDL2ϵ ist stapelbasiert. Während der Ausführung eines MDL2ϵ-Plans werden Zeiger auf die aktiven Timer und Interrupts der Operatoren auf einem Stapel (engl. *Stack*) abgelegt. Dies erleichtert die Auswertung der aktiven Interrupts/Timer und ermöglicht eine schnelle Suche im Plan, falls ein beliebiger Interrupt/Timer inaktiv wird. Wird ein Interrupt/Timer inaktiv, so wird von dieser Ebene aus das nächste aktive Atom gesucht.

In Abbildung 4.3 ist als Beispiel ein Plan aufgeführt, der ein *Random-Walk-Verhalten* erzeugt. Wird z.B. das Atom AROT_L ausgeführt, so befinden sich alle aktiven Operatoren auf dem Stack, vgl. Abbildung 4.4(a). Für Operatoren, die keine Timer oder Interrupts besitzen, werden leere Timer und Interrupts eingeführt, die immer gültig sind. Abbildung 4.4(b) zeigt den Ausführungsstapel nachdem das Atom AROT_L inaktiv geworden ist. In diesem Fall wurde das Atom AROT_R von der RUNION ausgewählt und ist nun das aktive Atom.

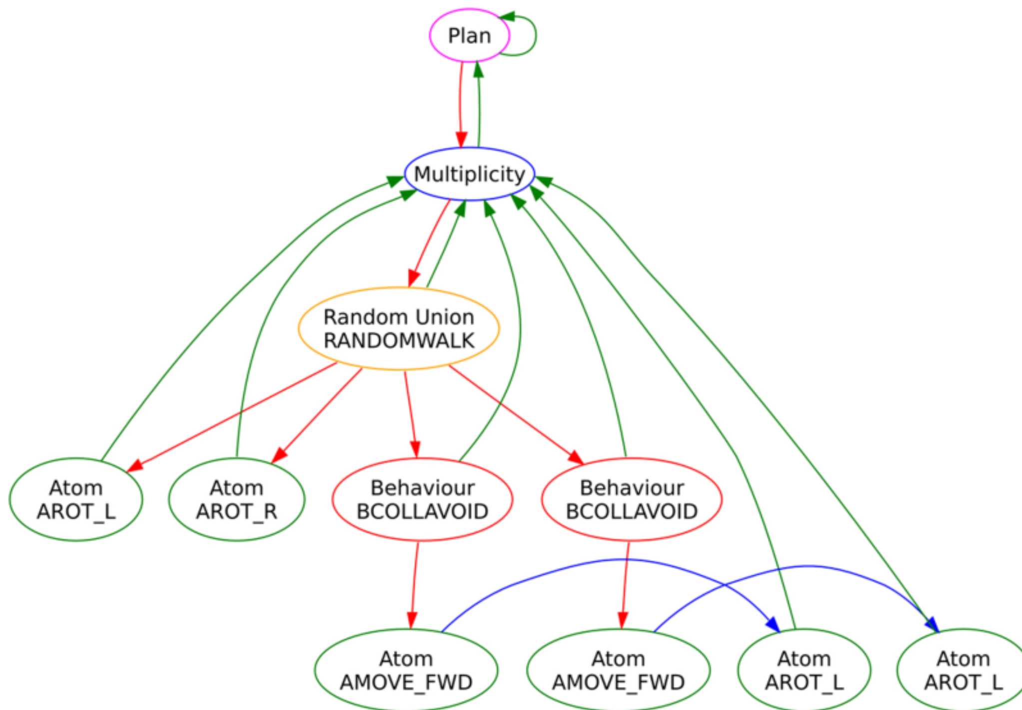
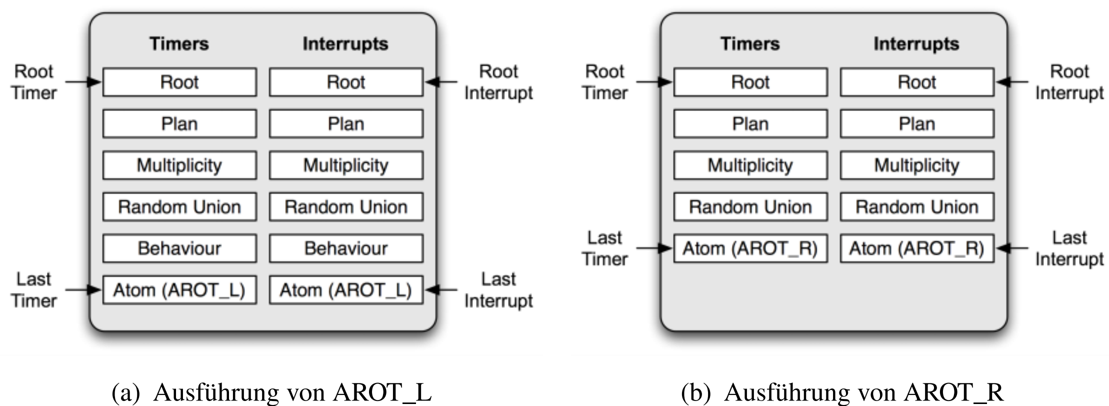


Abbildung 4.3: *Random-Walk*-Verhalten als Beispiel für den MDL2ϵ-Objektbaum.



(a) Ausführung von AROT_L

(b) Ausführung von AROT_R

Abbildung 4.4: Möglicher MDL2ϵ-Ausführungsstapel für den Plan aus Abbildung 4.3.

4.2.5 Darstellung von MDL2ϵ als C- und als Bytecode

Generierter C-Code

Zu Beginn der Arbeit wurden die MDL2ϵ-Pläne direkt in C-Code umgewandelt. Hierbei wurde die Struktur des MDL2ϵ-Plans in die Programmiersprache C über MDL2ϵApp umgeschrieben. Somit konnte schon am Anfang untersucht werden, in wie weit sich MDL2ϵ auf einem echten Roboter anwenden lässt. Allerdings hat der generierte C-Code bei wachsender Größe beträchtliche Nachteile bezogen auf die Größe des verbrauchten Speichers, da alle Ablaufstrukturen kopiert wurden und zusätzlich ihre Daten, wie z.B. die Ablaufzeit eines Atoms, mit in das Programm eingefügt wurden.

Optimierter C-Code

Um die Speichernachteile der ersten Version von generiertem C-Code zu umgehen, wurden einige Optimierungen an der C-Code Erzeugung vorgenommen [29]. Sich wiederholende Programmstrukturen, wie ATOM, BEHAVIOUR, MULT, UNION und RUNION, werden in eigenen Funktionen gekapselt, die nur noch die benötigten Daten übergeben bekommen.

Dieser Ansatz verbessert zwar die Ausnutzung des Flash-Speichers, hat aber dennoch den Nachteil, dass die Programmgröße bei größeren MDL2ϵ-Plänen schnell den verfügbaren Programmspeicher überschreiten kann. Da die Argumente dieser Funktionen zumeist Funktionszeiger sind, belegen sie zwei Bytes im Programmspeicher. Auch die meisten vom Compiler verwendeten Befehlscodes benötigen zwei Bytes, d.h. die Darstellung eines solchen Funktionsaufrufes in Befehlscode ist nicht gerade klein und kann mehrere Bytes belegen.

Des Weiteren wird mit dem Ansatz der Übertragung eines MDL2ϵ-Plans in C-Cod e dem System eine zusätzliche Beschränkung auferlegt. Jeder MDL2ϵ-Plan muss für das System kompiliert werden, und ist somit weder übertragbar noch während der Laufzeit leicht austauschbar.

Bytecode und Interpreter

Wie schon in Abschnitt 4.1 beschrieben bringt der Einsatz eines Interpreters erhebliche Vorteile bezogen auf die Übertragbarkeit und Plattformunabhängigkeit. Aber auch in Bezug auf die Speicherausnutzung ist ein Interpreter von Vorteil. So kann man mit Hilfe eines Bytecodes eine komprimierte Darstellung des Steuerprogramms erzeugen.

Dies erreicht man dadurch, dass innerhalb des Bytecodes die vorher verwendeten Funktionszeiger für Atom, einfache Interrupts und Variablen durchnummeriert werden. D.h. bei 16 verschiedenen verwendeten Atomen benötigt die Darstellung innerhalb des Bytecodes nur vier Bits anstelle der vorher verwendeten 16 Bits für den Funktionszeiger. Die Funktionszeiger müssen somit nur einmal in einem Array abgelegt werden, dessen Indizierung mit der Indizierung der Funktionszeiger übereinstimmt. Ähnlich wird zum Aufbau des Bytecodes mit den verwendeten MDL2ϵ-Operatoren vorgegangen, die aufgrund der festen Anzahl mit maximal drei Bits kodiert werden müssen. Der Anteil an Daten bleibt in seiner Originalgröße erhalten. Der gesamte MDL2ϵ-Plan kann somit in einer Bitsequenz dargestellt werden, die über Bytegrenzen hinaus gespeichert werden kann.

Der Aufbau des Bytecodes folgt einem einfachen Schema. So werden Verhalten im Plan als ganzes in einem Block im Bytecode abgelegt. Dies hat den Vorteil, dass die Verhalten durch einen einfachen Aufruf über einen Zeiger im Bytecode wiederverwendet werden können, wobei der verwendete Interrupt überschrieben werden kann. Um einen Zeiger zu verhindern, der jedes Bit im Plan einzeln adressiert, werden die Verhalten immer an Bytegrenzen abgelegt.

Innerhalb eines Verhaltens werden die MDL2ϵ-Operatoren sequentiell abgespeichert und bestehen immer aus den Identifizierungsbits, die darstellen um welchen Operator es sich handelt. Darauf folgen die zur Ausführung notwendigen Daten.

Beispiel der Codierung anhand eines Atoms

Die Codierung von MDL2ϵ als Bytecode wollen wir beispielhaft anhand eines Atoms betrachten. Ein Atom wird auf folgende Art codiert:

Funktion	max Bits	variabel
der Typ des MDL2ϵ-Elementes (Atom)	2	nein
Interrupt-Ausdruck	N	ja
Argumentanzahl der Exec-Funktion	16	ja
Ausführungsdauer der Exec-Funktion	16	nein
Argumente der Exec-Funktion	$\#args \cdot 8$	nein
Name der Exec-Funktion	16	ja

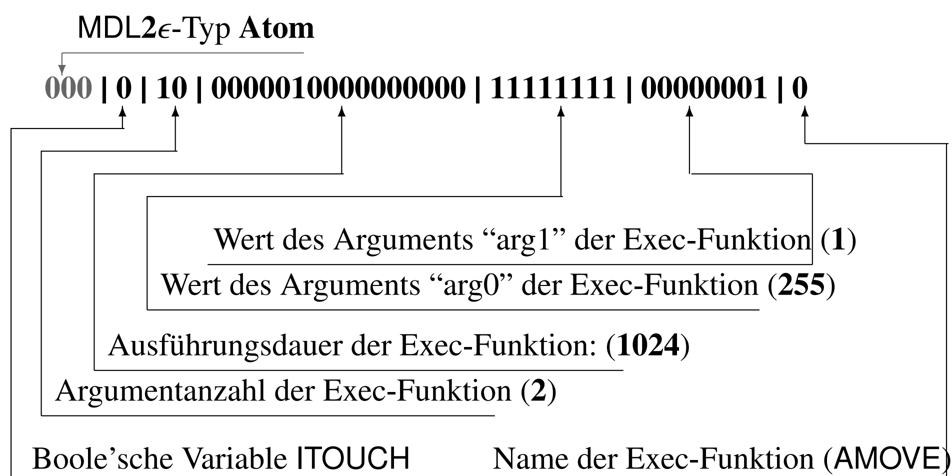
Betrachten wir folgenden MDL2ϵ-Plan

```
<PLAN duration="infinite">
  <ATOM name="ASTOP" interrupt="ITOUCH" arg0="255" arg1="1"
     duration="1024"/>
  <ATOM name="ASTOP" interrupt="ITRUE" duration="32"/>
</PLAN>
```

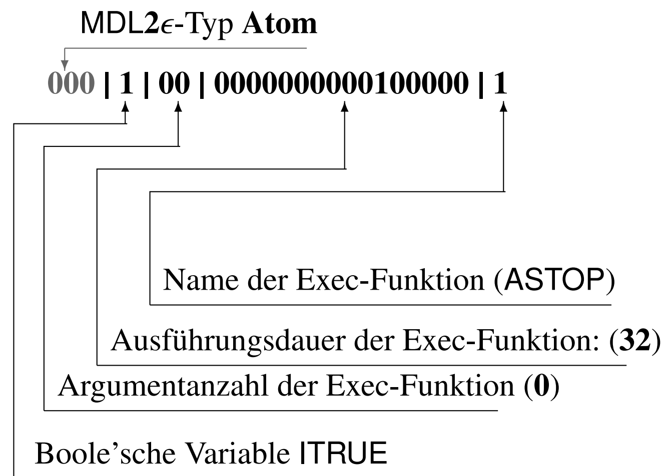
Nach der Analyse des Plans werden den MDL2ϵ-Daten die folgenden Werte zugeordnet:

MDL2ϵ-Typen	Codierungswert	binäre Darstellung
Atom	0	000
Boole'sche Variablen	Codierungswert	binäre Darstellung
ITOUCH	0	0
ITRUE	1	1
Exec-Namen	Codierungswert	binäre Darstellung
AMOVE	0	0
ASTOP	1	1

Die maximale Argumentanzahl der Exec-Funktionen ist zwei, deswegen ist die Codierungslänge der Argumentanzahl auch zwei. Der sich ergebende Bit-String des Atoms AMOVE ist damit:



Für den Bit-String des Atoms ASTOP ergibt sich:



Auf gleiche Art und Weise werden alle MDL2ϵ-Operatoren in eine Bit-Repräsentation übertragen. Eine vollständige Darstellung der Umformungen findet sich in [29].

Größenvergleich C-Code mit Bytecode

Um eine Aussage über den benötigten Programmspeicher der drei beschriebenen Ansätze zu machen, wurden Versuche mit verschiedene Plänen durchgeführt und miteinander verglichen. Beispielhaft werden die Ergebnisse für einen gemischten Plan dargestellt. Der Plan besteht aus einer Behaviour-Deklaration und mehreren MDL2ϵ-Sätzen. Jeder Satz besteht aus einer UNION, RUNION, MULT und einem BEHAVIOUR. In [29] findet sich eine ausführlichere Diskussion aller gemachten Versuche und deren Ergebnisse.

Es ist deutlich zu erkennen, dass bei einer Vervielfachung der Pläne, der verbrauchte Speicher bei den C-Code Varianten schnell ansteigt. Die Bytecode-Variante jedoch nur moderat steigt. Man kann leicht berechnen, das die Größe eines einzelnen verwendeten MDL2ϵ-Satzes bei den C-Code-Varianten zwischen 585 Bytes (optimiert) und 606 Bytes (nicht optimiert) liegt. Die Bytecode-Variante benötigt jedoch nur 35,5 Bytes pro Satz. Trotz des höheren Speicherbedarfs, durch den zusätzlichen Code für den Interpreter, ist dieser Ansatz bei schon mittleren Plänen den C-Code-Varianten vorzuziehen.

Abschließend ist zu sagen, dass der Bytecode-Interpreter viele Forderungen aus der Analyse in Abschnitt 4.1 erfüllt. Ein interpreterbasierter Ansatz ist in diesem Fall:

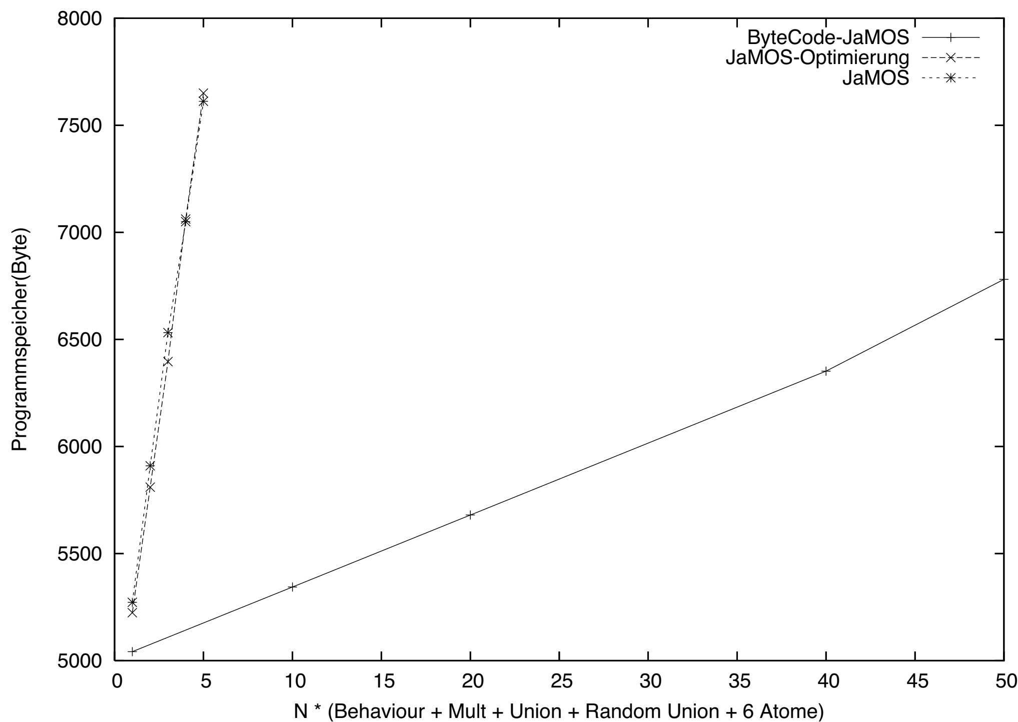


Abbildung 4.5: Ergebnisse für einen Plan mit einem BEHAVIOUR, einer MULT, einer UNION, einer RUNION und sechs Atomen.

- speichereffizient,
- plattformunabhängig (Trennung von Interpreter und Plattform),
- übertragbar (Controller lassen sich während der Laufzeit wechseln).

Ausgehend hiervon, wurden alle weiteren Experimente und Ergebnisse in der vorliegenden Arbeit mit einem interpreterbasierten Controller durchgeführt.

4.2.6 MDL2ϵ in der Robotersteuerung

Die Steuerung der Roboter basierend auf MDL2ϵ ist ein wichtiger Kernpunkt der vorliegenden Dissertation. Sie beinhaltet nicht nur die sukzessive Ausführung des MDL2ϵ-Planes sondern auch die Akquise der Daten, auf die der MDL2ϵ-Plan angewendet wird. Diese Datenakquise findet in den drei Robotern, I-SWARM, Jasmine und Wanda, aufgrund ihrer verschiedenen Hardwarearchitekturen auf drei verschiedenen Weisen statt. Dabei ist der vollständige Steuerprozess in den MDL2ϵ-Steuerzyklus eingebettet. Dieser Zyklus bestehen normalerweise aus den folgenden drei Schritten:

1. Überprüfen, ob das gegenwärtige Verhalten oder das Atom gültig ist:

   ja: fahre fort, das gegenwärtige Atom auszuführen,

   nein: wähle das nächste gültige Atom aus dem Plan und führen es aus.

2. Erfassen der Sensor-Informationen und Anpassen der MDL2ϵ-Interrupts.

3. Verringern der Timer der Verhalten und der Atome.

Abhängig von der Hardware des Roboters können einige Schritte der Schleife mit verschiedenen Frequenzen erfolgen. Zum Beispiel kann beim Jasmine-Roboter, Schritt drei mit einer niedrigeren Frequenz als Schritt eins und zwei ausgeführt werden, da diese für die Reaktivität des Roboters wichtig sind. Hierbei müssen alle Sensoren zum Erhalt neuer Information aktiv abgefragt werden. Dieser Pollingmechanismus ist sehr energieverbrauchend. Auch benötigen einige Atome die CPU während der Durchführung.

Im I-SWARM-Roboter jedoch müssen die Sensoren nicht ständig abgefragt werden. Alle Aktuatoren und Sensoren informieren die CPU durch einen Hardware-Interrupt, dass sie in einen neuen Zustand übergehen. Zum Beispiel, dass eine Nachricht empfangen bzw. gesendet wurde, oder eine Bewegung beendet wurde. Dieses erlaubt eine sehr energiesparende Integration des MDL2ϵ-Steuerparadigmas, das alle Eigenschaften der im I-SWARM-Roboter vorhandenen *Power Management Unit (PMU)* ausnutzt.

Beim Wanda-Roboter werden wie im Jasmine-Roboter einige Sensoren mit einer vorgegebenen Frequenz abgefragt, aber es ist auch möglich, dass die Sensoren neue Daten durch Interrupts der CPU wie beim I-SWARM-Roboter anzeigen. Im Gegensatz zum Jasmine-Roboter fällt beim Wanda-Roboter das Abfragen der Sensordaten nicht so stark ins Gewicht, bzw. hat keinen negativen Einfluss auf den zeitlichen Ablauf des MDL2ϵ-Plans, da die Ausführung der einzelnen Tasks durch das verwendete Echtzeitbetriebssystem geregelt wird.

Im Folgenden gehen wir genauer auf diese drei Arten ein.

Jasmine MDL2ϵ Operating System

Das *Jasmine* MDL2ϵ *Operating System (JaMOS)* [108] ist ein monolithisch aufgebautes, einfaches Betriebssystem geschrieben in C. Zugriffe auf Ressourcen des μC finden in der Regel durch Aufrufe von Methoden statt. Dies erlaubt nur eine rein sequentielle Abarbeitung. Somit sind die Abfrage

von Sensoren und die Kommunikation in den MDL2ϵ-Zyklus eingebettet. Nur der MDL2ϵ-Takt und die Berechnung der MDL2ϵ-Dauer findet in der Interrupt-Service-Routine eines Timers, wie in Abbildung 4.6 dargestellt, statt. Dies ermöglicht zum größten Teil die Einhaltung des Taktes. Allerdings findet die Prüfung, ob eine MDL2ϵ-Dauer abgelaufen ist, innerhalb des MDL2ϵ-Zyklus statt. Somit kann durch eine besonders lang dauernde Abfrage von Sensoren, wie es bei der Inter-Roboter-Kommunikation häufig der Fall ist, die MDL2ϵ-Dauer überlaufen, was eine zeitliche Schwankung zur Folge hat.

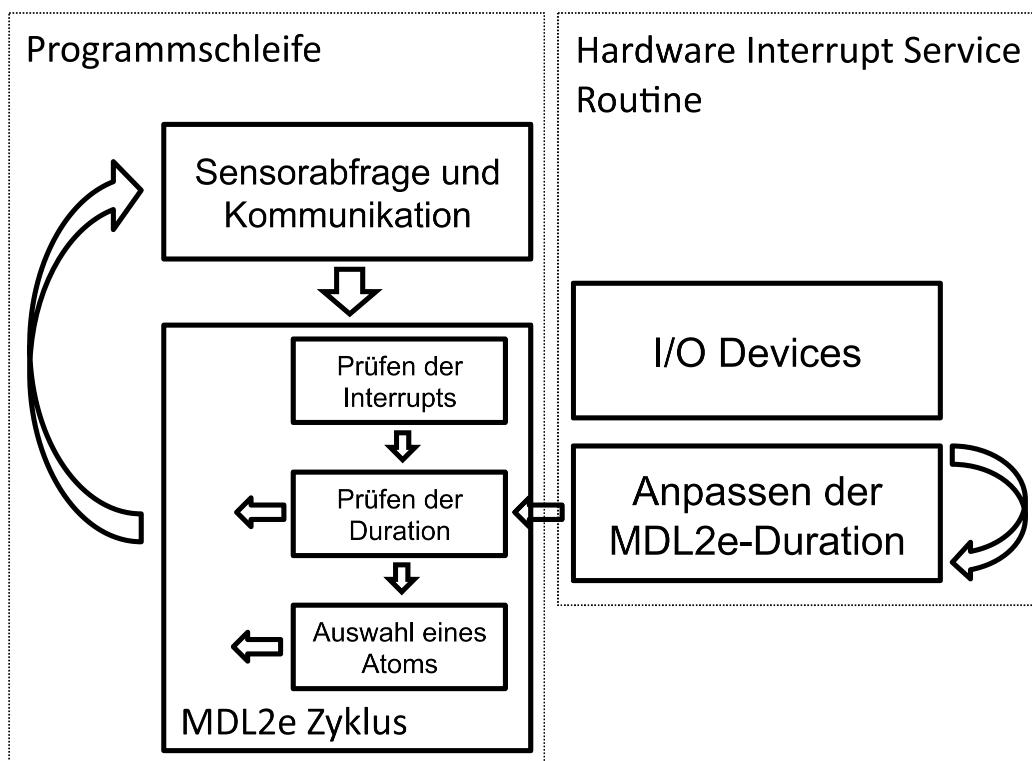


Abbildung 4.6: Ablaufdiagramm der MDL2ϵ-Steuerung im JaMOS.

Diese zeitliche Verzögerung kann dazu führen, dass der Roboter sich zu lange dreht oder bewegt, und somit sein Ziel verfehlt oder auch mit einem Hindernis kollidiert. Dieses Verhalten ist sehr indeterministisch und kann auch nur schwer in einer Simulation abgebildet werden. Es wirkt sich ebenfalls negativ auf die in einer Simulation evolvierten Pläne aus.

I-SWARM MDL2ϵ Operating System

Die oben angesprochene Power Management Unit ermöglicht dem Software Entwickler, alle Submodule des ASIC unabhängig abzuschalten [96, 97].

Dieses spart auf sehr einfache Art Energie aus Sicht von MDL2ϵ. Verantwortliche Module wie die Bewegungs-Steuereinheit können durch ein Atom eingeschalten werden und wenn erforderlich in einer *Interrupt-Service-Routine (ISR)* oder durch ein anderes Atom wieder ausgeschaltet werden. Das Ausschalten der *Optical Communication Unit (OCU)* oder der vibrierenden Nadel ist aus der Sicht des Roboters nicht sinnvoll, da es die einzigen Sensoren sind, die den Roboter auf andere Roboter oder Hindernisse aufmerksam machen. Wenn jedoch Energie knapp ist, können jene Sensormodule durch passende Atome auch automatisch abgestellt werden. Zum Beispiel kann es sein, dass das Senden und das Empfangen einer Mitteilung während einer Bewegung nicht möglich ist, dann kann das AMOVE Atom auch die OCU abstellen.

Die meiste Energie wird durch den DW8051 μC verbraucht. Hier versieht uns die PMU mit einer sehr wichtigen Eigenschaft. Der DW8051 kann in einen energiesparenden Schlafmodus versetzt werden, indem er vom Taktsignal für einen gegebenen Zeit getrennt wird. Diese Zeit wird im IsMOS als MDL2ϵ-Takt verwendet. Es verringert erheblich die Frequenz von Schritt eins des Ausführungszyklus.

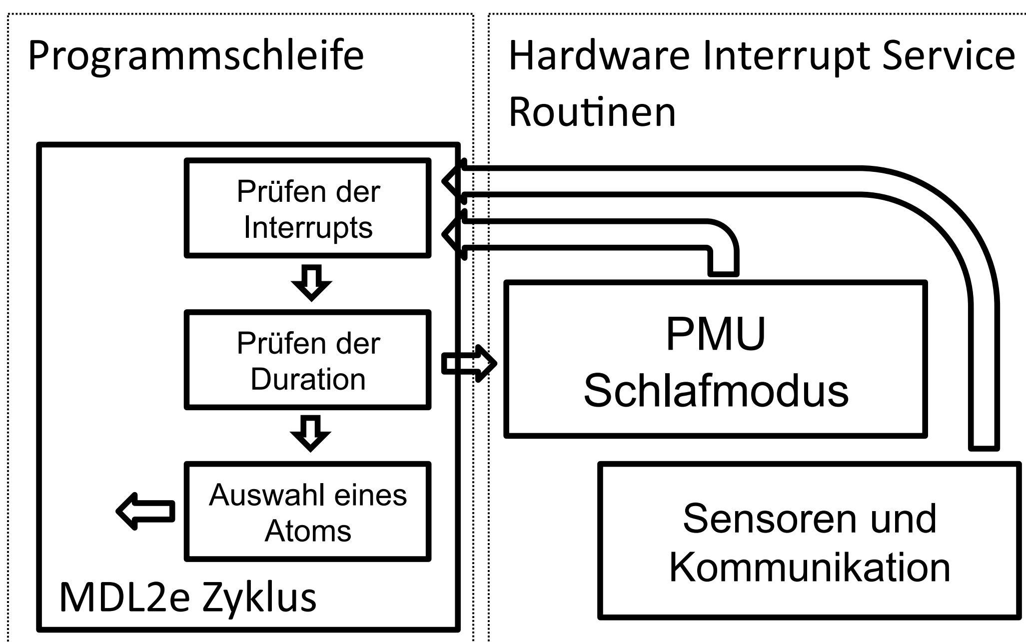


Abbildung 4.7: Ablaufdiagramm der MDL2ϵ-Steuerung im IsMOS.

Der DW8051 kann durch alle Interrupts der Peripherie aus dem Schlafmodus geweckt werden und diese in der ISR behandeln. Einige Hardware-Unterbrechungen beeinflussen die MDL2ϵ-Interrupts.

Abbildung 4.7 zeigt das Ablaufdiagramm der ausgeführten Schleife. Der DW8051 wird dabei ununterbrochen in Schlafmodus gesetzt. Wurde er aufgeweckt, wird überprüft, ob er durch den Schlaf-Timer oder irgendeinen anderen Interrupt geweckt worden ist. Wenn es der Schlaf-Timer war, werden alle MDL2ϵ-Timer (MDL2ϵ-Dauer) verringert und der μC wird in den Schlafmodus zurückgeschickt. Wenn irgendeine andere Unterbrechung oder ein Überlauf bei einem MDL2ϵ-Timer auftritt, wird überprüft, ob das gegenwärtig aktive Verhalten und das Atom noch gültig sind. Wenn nicht, wird ein neues Atom aus dem Plan ausgewählt.

Abbildung 4.8 zeigt den Verlauf von IsMOS während eines Experimentes mit einem Kollisions-Vermeidungs-Plan mit zufälliger Bewegung. Dabei sieht man in den Zeilen D14 und D13 die an die Beinchen des Roboters angelegten Frequenzen für die Bewegung, sowie in Zeile D8 den MDL2ϵ-Takt und in Zeile D7 den Takt des DW8051. Man kann deutlich anhand der Stellen B, D und F erkennen, wann ein neues Atom im Plan herausgesucht wurde. In den Bereichen A, C und E wird der DW8051 nur aufgeweckt, um den MDL2ϵ-Zähler zu erniedrigen, bzw. die MDL2ϵ-Interrupts zu prüfen. Der Fehler, der im MDL2ϵ-Takt durch die benötigte Rechenzeit entsteht, wurde durch eine Messung der benötigten Zeit minimiert und fließt so in die nächste Schlafperiode mit ein. Es ist nicht möglich, eine zu kurze Schlafperiode, die durch einen eintretenden Hardware-Interrupt entsteht, auszugleichen, da es nicht möglich ist die Dauer auszulesen, bzw. zu messen, die die CPU schlief. Alle internen Zeitgeber des DW8051 werden mit dem Anhalten des Taktes ebenfalls gestoppt.

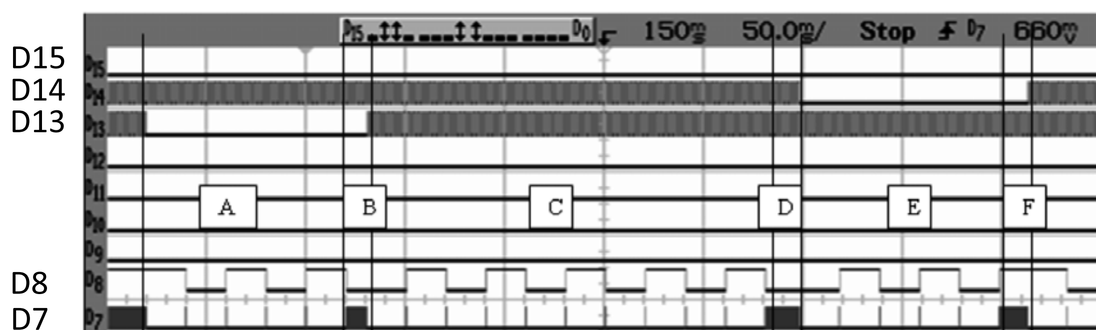


Abbildung 4.8: Ablauf von IsMOS auf dem I-SWARM-Roboter für einen einfachen Kollisions-Vermeidungs-Plan mit zufälliger Bewegung.

SymbricatorRTOS

Im Rahmen des Symbrion- und des Replicator-Projektes [57] wurde ein auf FreeRTOS[2] basierendes Betriebsystem zur Robotersteuerung entwickelt: das SymbricatorRTOS. Dieses Roboterbetriebssystem abstrahiert die C-basierte FreeRTOS API durch eine C++-basierte API und erweitert das *Real-Time Operating System (RTOS)* durch Funktionen, die für den Betrieb des Roboters notwendig sind. So stellt das SymbricatorRTOS Schnittstellen für alle verwendeten I/O-Geräte und Sensoren zu Verfügung. Diese sind als Tasks implementiert. Eine Shell vereinfacht den Zugriff auf die Roboterfunktionen und erleichtert das Debuggen und Testen des Roboters.

Basierend auf dem Warteschlangensystem von FreeRTOS wurde ein Inter-Task-Kommunikationssystem, die sogenannten *Hooks*, implementiert, mit dem sich einzelne Tasks an andere Task einhaken können, um über Veränderungen zeitnah informiert zu werden.

Des Weiteren stellt das SymbricatorRTOS ein einfaches Weltmodell zu Verfügung, über das die Steuerung auf den Zustand des Roboter zugreifen kann. Um inkonsistente Daten zu vermeiden, kann immer nur ein Task schreiben, wohingegen alle Tasks lesen können. Eine genauere Darstellung des SymbricatorRTOS findet sich in [106].

Der Ablauf aller Teile des Betriebssystems und somit auch der MDL2ϵ-Zyklus finden nebenläufig von einander statt und werden durch den prioritätenbasierten Scheduler des RTOS verwaltet. Abbildung 4.9 zeigt, dass jeder Sensor und jedes Gerät (engl. *Device*) in einem eigenen Task gekapselt ist. Diese fragen entweder periodisch Sensordaten direkt ab oder warten auf Nachrichten von den ISRs.

Der MDL2ϵ-Takt wird hier durch das RTOS und nicht durch einen externen Zeitgeber, wie im Fall vom IsMOS und JaMOS, verwaltet. Dies ermöglicht einen Zeitverlauf des MDL2ϵ-Taktes ohne signifikante Abweichungen. Eine starke Veränderung des Zeitverlaufes oder sogar das Überschreiten einer MDL2ϵ-Dauer, wie in JaMOS durch das Auslesen von Sensoren oder durch Kommunikation mit anderen Robotern, findet nicht statt. Die oben erwähnten Hooks ermöglichen zusätzlich, wie im IsMOS, ein direktes Eingreifen der MDL2ϵ-Steuerung bei einer Veränderung von Daten. Dafür muss der MDL2ϵ-Task an den entsprechenden Task gehängt werden.

Es ist sogar wie im IsMOS möglich, den μC während der Dauer des Idle-Tasks schlafen zu legen. Dieser wird dann wieder durch den SysTick-

[2]http://www.freertos.org

oder andere Interrupts der Peripherie aufgeweckt.

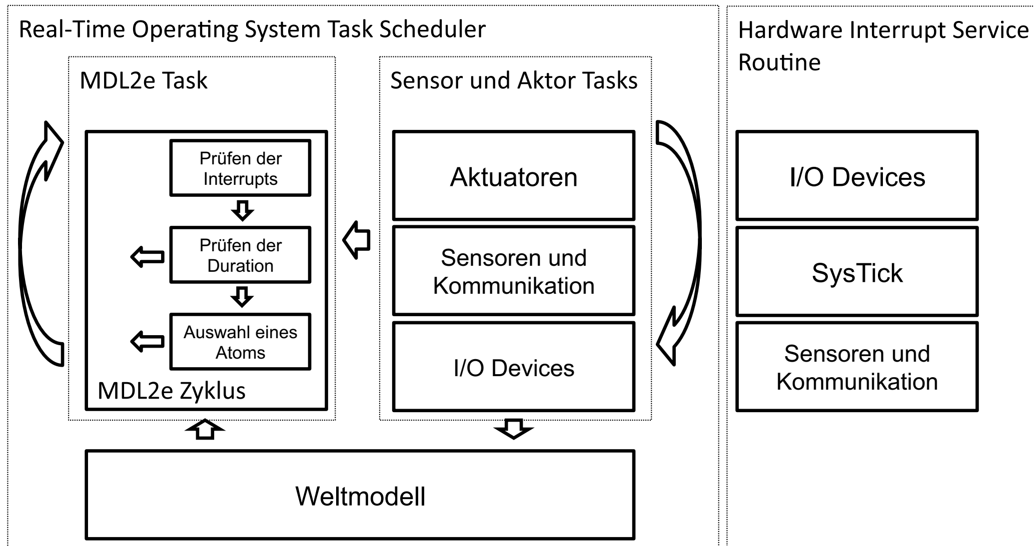


Abbildung 4.9: Ablaufdiagramm der MDL2ϵ-Steuerung im WandaRTOS.

4.3 Simulationsumgebung

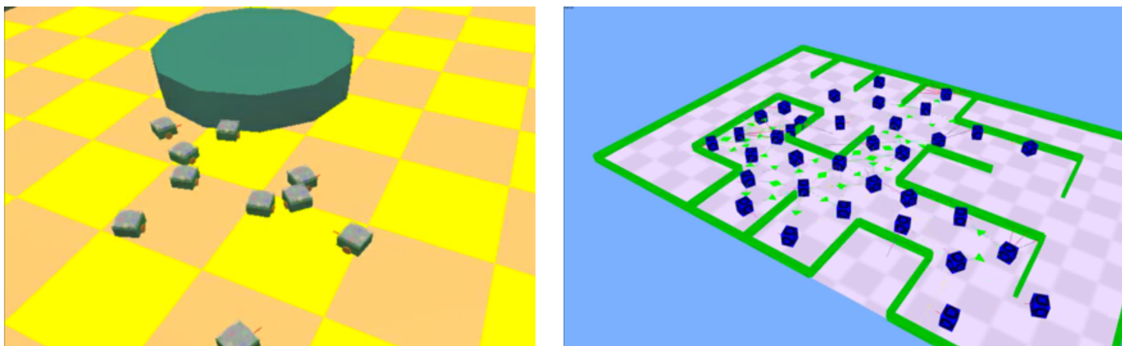
Kinetischen Zustandsautomaten (Eine Simulationsumgebung ist ein wichtiges Werkzeug für den Entwurf von Algorithmen zur Steuerung von Robotern. Sie ermöglicht eine kürzere Entwurfsphase, schützt den Roboter vor Fehlern in den Algorithmen und ist eine wichtige Vorraussetzung für offline-offboard EC. Simulationsumgebungen reichen von sehr einfachen Darstellungen von Robotern als einfache Punkte mit sehr einfach simulierter Sensorik, wie in NetLogo [116], bis zu komplexen Simulationen wie Symbricator3D [118], WeBots, usw., in denen die Physik des Roboters genauer simuliert werden kann.

Im Rahmen der Dissertation wurden drei Simulationsumgebungen an MDL2ϵ angebunden und getestet. So gibt es Implementierungen für WeBots, Player/Stage und Breve. MDL2ϵ wurde auch in Symbricator3D eingebunden und in [90] verwendet, was in dieser Dissertation jedoch nicht weiter betrachtet wird.

Von den drei verwendeten Simulationsumgebungen wurde die Simulationsumgebung Breve für die Simulation des Jasmine-Roboters verwendet, siehe Abbildung 4.10(b). Ihre Vorteile liegen in der einfachen 3D-Darstellung und der Verwendung von OpenSource, was eine Anpassung der Simulationsumgebung an die Bedürfnisse des gegebenen Systems vereinfacht.

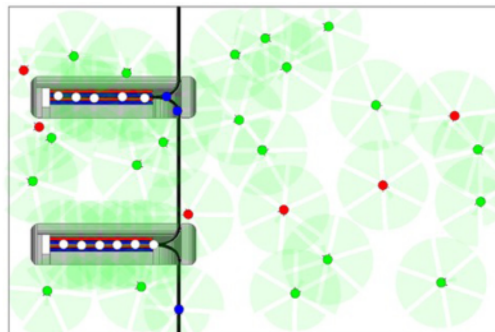
Webots dagegen ist durch seine geschlossene Form als kommerzielle Simulationsumgebung nur schwer zu erweitern. Nach der Erstellung einer MDL2ϵ-Schnittstelle mit einem Modell für eine frühe Version des I-SWARM-Roboters, vgl. Abbildung 4.10(a), wurde Webots nicht weiter betrachtet.

Player/Stage wurde zur Simulation des Wanda-Roboters eingesetzt, siehe Abbildung 4.10(c).



(a) I-SWARM-Roboter in Webots

(b) Jasmine-Roboter in Breve



(c) Wanda-Roboter in Stage

Abbildung 4.10: Roboter in den verschieden Simulationsumgebungen.

4.3.1 Integration von Breve in die MDL2ϵ-Applikation

Breve ist eine von J. Klein entwickelte OpenGL basierte Simulationsumgebung für Artifical Life und Multiagenten-Systeme [60]. Als Physik-Engine wird wie in anderen Simulationen die *Open Dynamics Engine (ODE)* verwendet. Viel Wert hat der Entwickler auf eine objektorientierte, interpretierte Simulationssprache *Steve* gelegt, in der die Simulationsumgebung programmiert wird. Aufrufe von externen Funktionen sind durch eine einfache auf dynamischen Bibliotheken basierende Schnittstelle möglich. Dies vereinfachte das Einbinden von MDL2ϵ und auch des GP-Frameworks (FLSO)

wie in Kapitel 5 beschrieben. Eine Trennung von MDL2ϵ und dem simulierten Roboter war wie in Abbildung 4.1 dargestellt problemlos möglich.

Für die Integration von Breve in MDL2ϵApp wurde die in Abbildung 4.2 dargestellte Schnittstelle in Breve implementiert. Für jede Klasse gibt es ein Gegenstück implementiert in Steve. Die Fabrikklasse auf der Seite von Breve gibt Zeiger auf Steve-Objekte zurück, welche von MDL2ϵApp in den Objektbaum eingebunden werden. In jeder Iteration wird beim Durchlauf der Roboterklasse der Scheduler von MDL2ϵApp aufgerufen und der MDL2ϵ-Plan abgearbeitet.

Abbildung 4.11 zeigt die Anbindung des Robotermodells in Breve. Die obere Schicht stellt die von Breve zu Verfügung gestellte Schnittstelle zur Simulation von Agenten dar. Die mittlere Schicht zeigt die Implementierung der Fabrikklasse (IsFactory) und die der MDL2ϵ-Schnittstelle (Executeable/ Atom, Interrupt, Variable). Die unterste Schicht ist anwendungsabhängig und enthält z.B. die Fitnessfunktion für GP.

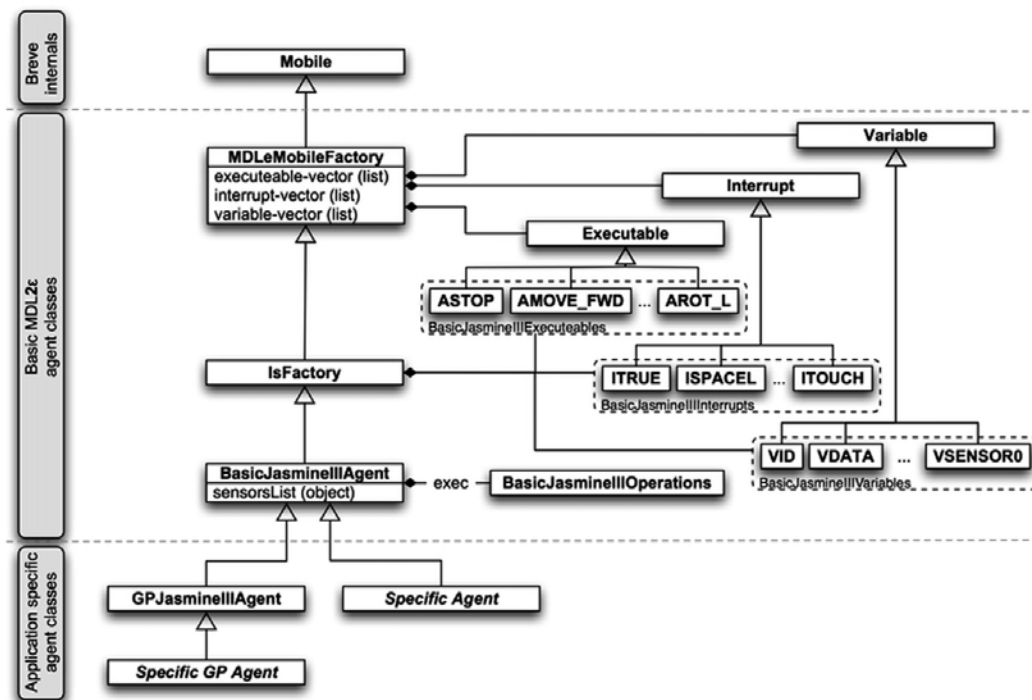


Abbildung 4.11: UML-Diagramm des Robotermodells in der Simulationsumgebung Breve [33].

Modellierung der Roboter

Die Modellierung der Roboter wurde so entworfen, dass verschiedene Roboter leicht zu erzeugen und Sensorenmodelle durch andere Sensormodelle einfach austauschbar sind. Eine Klasse beschreibt den Roboter und enthält die Geometrie des Roboters, die Geometrie der Räder, die Texture sowie eine Liste der verwendeten Sensoren mit Typ und ihrer relativer Position und Ausrichtung zum Mittelpunkt des Roboters. Diese Klasse wird zu Beginn der Simulation benutzt, um mit Hilfe einer Sensor-Fabrikklasse ein entsprechendes Robotermodell zu erzeugen.

Modelliert wurden alle Roboter, auch der I-SWARM-Roboter, mit einem differentiellen Antrieb und der entsprechenden Anzahl von IR-Sensoren. Um eine dem Roboter hinreichend ähnliche Simulation der IR-Sensoren zu erreichen, ohne die Simulationsgeschwindigkeit stark zu reduzieren, wurde in [113] ein Modell mit zugehörigem Kalibrierungsverfahren entwickelt. Dieses Modell wurde für die Simulation der Jasmine verwendet.

4.3.2 Integration von Stage mit MDL2ϵ

Für die Integration von MDL2ϵ in Stage wurde im Gegensatz zur Integration von MDL2ϵ in Breve kein direkter Anschluss von MDL2ϵApp in Stage gewählt [67]. Es wurde der auch auf den Robotern verwendete Interpreter direkt integriert. Dies ermöglicht die Ausführung desselben Bytecodes in der Simulation wie auf dem Roboter. Dafür wurde das in SymbricatorRTOS integrierte Weltmodell auf Stage übertragen, was die Integration von Variablen und Sensorwerten vereinfacht.

4.4 Interaktive Dynamische Arena

Um Experimente mit Roboterschwärmen durchführen zu können, ist es wichtig, eine geeignete Experimentierumgebung zu haben. Diese Umgebung zusammen mit dem verwendeten Roboter bestimmt in hohem Maße, was für Experimente durchgeführt werden können. Mit Hilfe einer guten *interaktiven Arena* können Sensoren simuliert werden, die aufgrund ihrer Größe, ihres Energieverbrauchs oder auch aufgrund der Kosten nicht direkt in den Roboter integriert werden können. Dies ist in den Augen des Autors ein probates Mittel, auf sehr flexible Weise viele verschieden Szenarien zu simulieren ohne dabei den Aspekt der Benutzung von echter Hardware

zu übergehen.

Die in der vorliegende Arbeit entwickelte Arena hat den Anspruch, sowohl die gleichen Möglichkeiten zur Verfügung zu stellen, wie die I-SWARM-Arena in Abbildung 3.3, als auch diese um weitere zu erweitern. Wie schon in Abschnitt 4.1 angesprochen ist es wichtig, auch die Arena so zu gestalten, dass GP mit den Robotern möglich ist. Auswertungen von Experimenten sollten ähnlich wie in der Simulationsumgebung durchgeführt werden können. Für die Forschung an autonomen Roboter spielen *dynamische Umgebungen* eine große Rolle, was beim Entwurf der Arena mit berücksichtigt wird.

Die entwickelte Arena bezeichnen wir als *interaktive, dynamische Arena*, da eine direkte Interaktion Roboter-Arena-Roboter und Mensch-Arena-Roboter möglich ist.

4.4.1 Konzept

Die entwickelte Arena ist 240 cm × 160 cm groß und basiert auf drei Konzepten, die zusammen eine interaktive, dynamische Arena möglich machen. Diese drei Konzepte sind:

1. Projektion von Daten in die Arena, die vom Roboter über zwei Photosensoren wahrgenommen werden,

2. Abgabe von Lichtsignalen von den Robotern an die Arena, die mit Hilfe einer Kamera aufgenommen werden, und

3. drahtlose Kommunikation mit den Robotern.

Abbildung 4.12 zeigt das Gesamtkonzept der Arena und die drei möglichen Arten von Datentypen. Die projizierbaren Daten sind die *Position* eines Pixels in XY-Koordinaten kodiert als Graycode, sowie *8-Bit-Datenfelder* im normalen Binärcode und *Pheromone* kodiert als Grauwert. Diese Daten werden mit Hilfe des ODeM-Sensors aufgenommen und vorverarbeitet und dann an den Roboter übermittelt. Der ODeM-Sensor ist mit einer Schnittstelle für drahtlose Kommunikation ausgestattet und wird in Abschnitt 4.4.2 genauer beschrieben. Abbildung 4.13 zeigt die Arena.

Die projizierten Daten bestehen immer aus einem Kopf, der aus Synchronisations-Bits, sowie zwei Bits zur Unterscheidung der jeweiligen Datenart besteht. Die Projektion findet mit Einzelbildern statt, die mit 60 Hz gesendet werden. Die Bilder werden mit Hilfe der Arena-Steuerung erstellt,

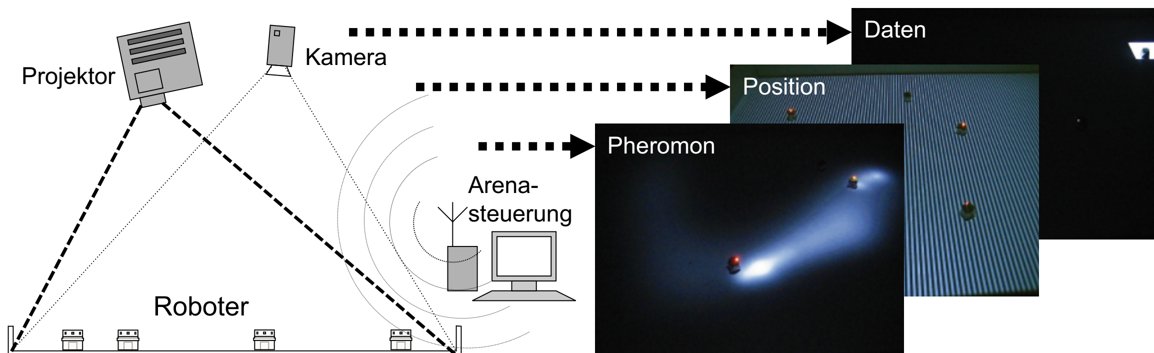


Abbildung 4.12: Schematische Darstellung des Arena Konzeptes mit den möglichen projizierbaren Daten (Position, Pheromone und Datenfelder).

welche in Abschnitt 4.4.3 genauer beschrieben wird. Zunächst werden die verschieden Datenarten erklären.

Positionsbestimmung

Die Arena ermöglicht dem Roboter, seine absolute Position im Arbeitsbereich zu bestimmen. Hierfür wird die X und Y Position als Graycode auf den Arbeitsbereich projiziert. Ausgestattet mit zwei Photosensoren kann der Roboter so seine Position und Orientierung bestimmen. Die Auflösung der Position hat eine Genauigkeit von 1,5625 mm und die Orientierung eine Genauigkeit von 5°. Es wurde ein Graycode gewählt, da nebeneinander liegende X- und Y-Werte sich nur um ein Bit unterscheiden. Dies vermindert Fehler bei der Messung der Position während einer Bewegung. Um den Fehler, der durch die Bewegung des Roboters entsteht, so gering wie möglich zu halten, werden zusätzlich die einzelnen Positionsbits für X und Y abwechselnd projiziert, wobei mit den höchstwertigen Bits angefangen wird. Die Umrechnung von dem gewählten Gray- zu Binärcode geht mit folgender Funktion[3] :

```
1 long grayToBin( gray, numberOfBits) {
2   bitPosition = numberOfBits - 1;
3   bin = 0;
4   while (bitPosition >= 0) {
5     bin   = (((gray >> bitPosition) & 1) ^ (bin & 1))
6             | (bin << 1);
7     bitPosition--;
8   }
9   return bin;
```

[3]Mit den bitweisen Operatoren: & und, | oder, ^ xor, << links shift und >> rechts shift

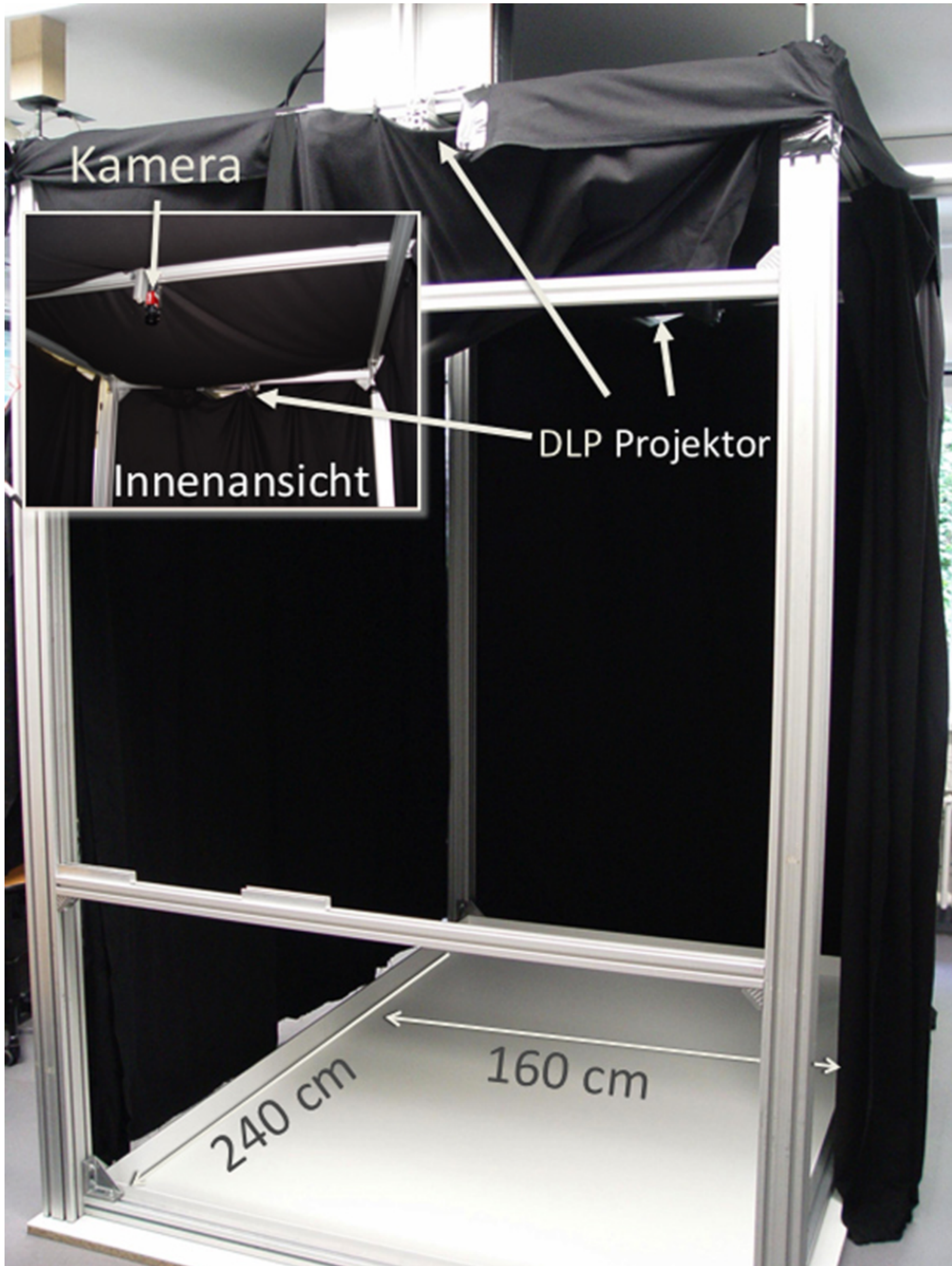


Abbildung 4.13: Die interaktive Arena für Schwarmroboterversuche mit Projektoren, Kamera und Arbeitsbereich.

```
10 }
```

Der Ablauf der Messung ist in Abbildung 4.14 dargestellt. Die schwarzen Flächen stellen in der Bitsequenz eine ’0’ und die weißen eine ’1’ dar. Die vom Roboter im Beispiel wahrgenommene Bitsequenz für den linken Sensor ist dann “00110001” bzw. “11110010” für den rechten Sensor. Dies ergibt in dem Beispiel $(0100{,}0101)_{graycode} = (7{,}6)_{10}$ als Position für den linken Sensor und $(1101{,}1100)_{graycode} = (9{,}8)_{10}$ für den rechten Sensor. Hieraus lässt sich abhängig von der Lage der Sensoren zum Mittelpunkt des Roboters, die Position und Orientierung des Roboters berechnen [91].

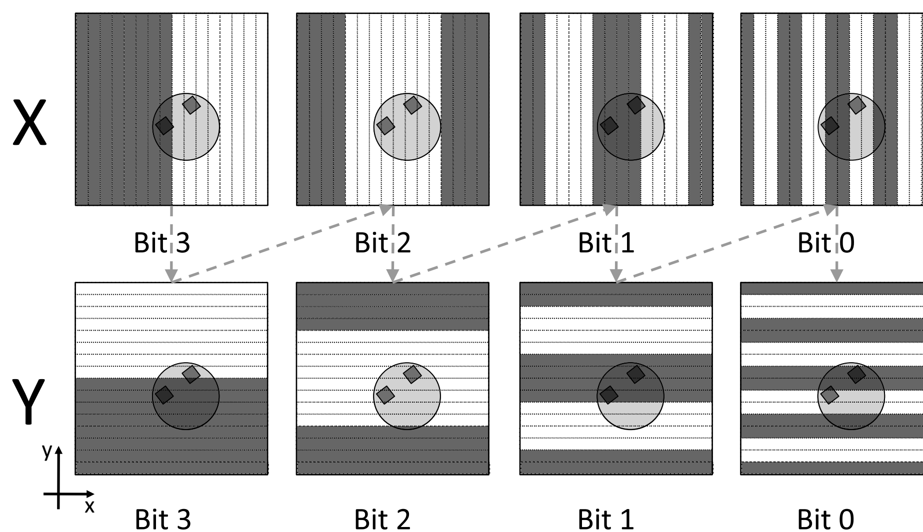


Abbildung 4.14: Positionsbestimmung anhand eines Graycodes.

Mit Hilfe der Position kann der Roboter verschiedenste Sensoren simulieren, die sonst nicht zu Verfügung stehen. So kann er sich zum Beispiel eine Position merken, um diese wieder zu finden. Dies simuliert z.B. die Sensoren von Bienen und Ameisen, die anhand des Sonnenstandes und der Polarisation des Lichtes, die Position ihres Nestes oder im Fall von Bienen auch die Richtung und Entfernung zu einer Futterquelle bestimmen können. So lassen sich die Verhalten sozialer Insekten anhand von Robotern leichter studieren, ohne diese mit zusätzlichen Sensoren und zusätzlicher Software zu belasten.

Beliebige Datenfelder

Eine weitere Art von Daten, die mit dem System dargestellt werden kann, sind 8-Bit-Binärdaten, die in beliebiger Form und an beliebiger Stelle projiziert werden können. Diese Felder können so programmiert werden, dass

sie sich während eines Versuchs verändern. So können sie während eines Versuchs ihre Größe, Position oder auch den Wert ändern.

Solche Datenfelder sind sehr nützlich, um das Verhalten von Algorithmen auf eine dynamische Umgebung zu untersuchen. Dynamische Umgebungen, die sich wiederholbar genau verhalten, sind normalerweise nur sehr schwer zu realisieren, was einen Vergleich verschiedener Algorithmen mit echten Robotern normalerweise sehr schwierig macht.

Anwendungen solcher Datenfelder sind z.B. Reinigungsszenarien, in denen die Roboter ein Gebiet von Verunreinigungen jeglicher Art finden und bereinigen müssen (z.B. Ölteppiche von Gewässeroberflächen). In Sammelszenarien müssen Roboter verschiedene Ressourcen finden und einsammeln. Hierbei könne Ressourcen erschöpfen oder verschiedene Wertigkeiten haben. Das Schwarmrobotersystem muss sich darauf dynamisch einstellen.

Virtuelle Pheromone

Virtuelle Pheromone sollen das Verhalten biologischer Pheromone simulieren, wie sie in sozialen Schwärmen in der Natur zur Anwendung kommen. Pheromone stellen ein Kommunikationsmedium des Schwarms mit sich selbst über seine Umgebung dar. Sie sind in der Natur ein wichtiges Mittel zur Organisation der Arbeitsteilung und Optimierung. Das Experiment in Abschnitt 6.1.1 zeigt dies. Virtuelle Pheromone basierend auf Licht geben die Möglichkeit, Parameter wie Diffusions- und Evaporationsgeschwindigkeit des Pheromons zu untersuchen, ohne den Roboter mit verschieden chemischen Sensoren auszustatten.

Im Fall der Arena werden die lokalen Pheromonekonzentrationen anhand eines projizierten Grauwertbildes (Pheromonkarte) dargestellt. Die Evaporation und die Diffusion werden über einen veränderbaren Gaussfilter realisiert, den man je nach gewünschter Evaporations-/Diffusionsgeschwindigkeit vor der Projektion des Bildes über die Pheromonkarte laufen lässt. Die Pheromonkarte kann dann von dem Roboter über seine Photosensoren wahrgenommen werden. Je nach Stärke des Gradienten kann der Roboter den Gradienten auf seinen beiden Sensoren direkt wahrnehmen und sich danach ausrichten.

Die Roboter legen Pheromone ab, indem sie eine Leuchtdiode auf dem ODeM-Board anschalten, welche dann über eine Kamera der Arena-Steuerung wahrgenommen wird. Die abgegebenen Pheromone werden in das Pheromonbild addiert und zurück in die Arena projiziert.

Da die Helligkeitsverteilung der Projektoren ungleichmäßig ist, wird vor

jedem Pheromonbild ein Normierungsbild gesendet, das einen bekannten Grauwert zeigt, anhand dessen der Pheromonwert normiert wird.

4.4.2 ODeM

Das Optical Data transmission system for Micro robots ist das Herzstück der gesamten Arena [91]. Mit seiner Hilfe kann der Roboter die Daten wahrnehmen und verarbeiten. Das ODeM hat zur Verarbeitung der Daten einen AtMega168 μC. An den μC sind zwei Photodioden (BPW34) über ADCs, bzw. beim Wanda-Roboter auch über einen Schmitt-Trigger angeschlossen, welche die sequentiell projizierten Daten aufnehmen. Da die verwendeten *Digital Light Processing (DLP)*-Projektoren ein sehr verrauschtes Signal senden, ist eine Vorfilterung der Daten zum Entfernen hoher Frequenzen nötig. Dies wird über einen RC-Filter implementiert.

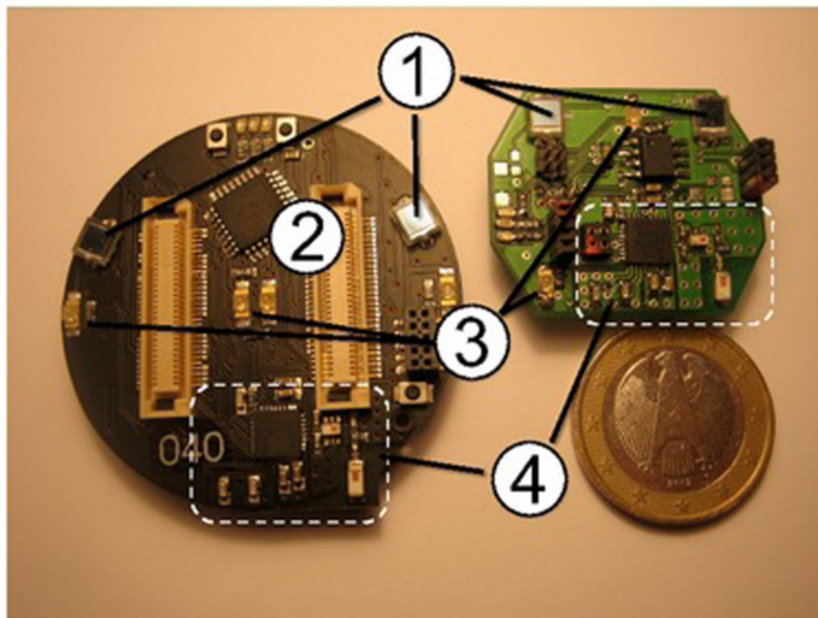


Abbildung 4.15: Das ODeM-Board für Wanda (links) und Jasmine (rechts) mit seinen funktionalen Elementen. 1. Photodiode; 2. AtMega168; 3. LEDs; 4. ZigBee

Zusätzlich zu den Photosensoren besitzt der Roboter mehrere LEDs, die zum einen den Status des Roboters anzeigen, zum anderen zur Abgabe von Pheromonen benutzt werden. Abbildung 4.15 zeigt das ODeM-Board mit seinen verschiedenen funktionalen Teilen:

①	Photodiode (BPW34)
②	AtMega168
③	Pheromon-LEDs
④	ZigBee

Der Roboter kann die Funktionen des ODeM-Boards über TWI steuern. Über TWI erhält er auch die projizierten Daten vorverarbeitet als Position, Daten und Pheromone.

Ein weiterer wichtiger Teil des ODeM-Boards ist die auf einem ZigBee-Chip basierende drahtlose Kommunikationsschnittstelle, auf die in Abschnitt 4.4.4 eingegangen wird.

4.4.3 Arena-Steuerung

Die Arena-Steuerung besteht aus einem Rechner, an dem sowohl die zwei Projektoren, die Kamera als auch eine ZigBee-Basisstation (Host Controller) angeschlossen sind. Die Steuerung ist für die Vorverarbeitung und Analyse der Kamerabilder und der Erzeugung und Darstellung der projizierten Bildsequenzen verantwortlich. Die Arena-Steuerung stellt eine Schnittstelle für ECMAScript zu Verfügung [67, 50]. Hiermit werden Experimente, die dynamisch auf das Experiment Einfluss nehmen und auch auf den Ablauf reagieren können, programmiert.

Projektor-Kamera Kalibrierung

Um die vom Roboter abgegebenen Pheromone an die Stelle in der Arena zu projizieren, an der der Roboter diese abgegeben hat, wird eine Projektor-Kamera-Kalibrierung benötigt. Hierbei handelt es sich nicht um eine klassische Tsai-Kalibrierung der Kamera, da diese zwar eine Entzerrung des Bildes zulässt, aber dann das eigentlich Problem, die Abbildung vom wahrgenommenem Roboter auf einen projizierten Punkt, noch zusätzlich durch eine Koordinaten Registrierung gelöst werden muss.

Ein direkterer Ansatz wurde gewählt, um dies zu vereinfachen. Hierzu wird der in Abschnitt 4.4.1 beschriebene Graycode Bild für Bild auf die, um die Höhe des Roboters erhöhte, Arena projiziert und mit der Kamera aufgenommen. Danach wird anhand des Graycodes die in jedem Pixel der Kamera wahrgenommene Position in der Arena bestimmt. Hierdurch erhält

man eine direkte Abbildung von dem Pixel in der Kamera zu der Position in der Arena/Projektor. Eine in der Kamera wahrgenommene LED kann nun direkt über eine Tabelle in Projektor-/Arenakoordinaten umgewandelt und projiziert werden.

Diese Art der Kamera-Beamer-Kalibrierung ist sehr schnell (wenige Sekunden), genau und rotations- und projektionsunabhängig.

4.4.4 Drahtlose Kommunikation basierend auf ZigBee Hardware

Drahtlose Kommunikation ist ein wichtiges Werkzeug für die automatische Durchführung von Schwarmexperimenten. Zusammen mit MDL2ϵ ermöglicht sie das Aufspielen von Steuerplänen auf alle Roboter zur gleichen Zeit, das Starten und Initialisieren des Experimentes und die Überwachung des Experimentes in Echtzeit [105]. Auch im Vorfeld von Experimenten ist die drahtlose Kommunikation wichtig zum Debuggen von Algorithmen.

Bei drahtloser Kommunikation für mobile Systeme steht zum einen Bluetooth und zum anderen ZigBee zu Verfügung. In dem ODeM kommt ZigBee zur Anwendung, da es im Vergleich zu Bluetooth einen deutlich geringeren Energieverbrauch hat, was sich positiv auf die Dauer der Experimente auswirkt, und sich deutlich mehr ZigBee-Geräte in einer Umgebung befinden können als bei Bluetooth.

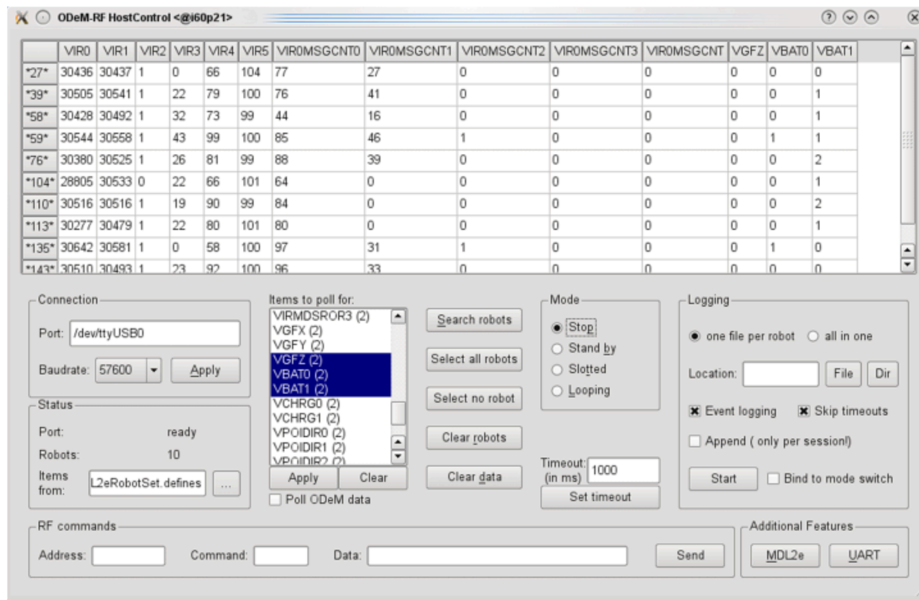
Bei der verwendeten ZigBee-Hardware handelt es sich um einen AT86RF230 ZigBee-Chip von Atmel. Dieser wird von dem schon vorhanden ATMega168 angesteuert. Abbildung 4.15 zeigt die ZigBee Hardware auf der ODeM-Platine für Wanda und Jasmine. Der Chip an sich stellt nur die physikalische Ebene von ZigBee dar, wobei mit ZigBee eigentlich auch eine Protokollimplementierung gemeint ist. Die Implementierung des ZigBee-Protokolls benötigt jedoch mehr Speicher als der ATMega168 zur Verfügung stellt. Daher wurde ein proprietäres Protokoll implementiert, das auf der ZigBee-Hardware aufsetzt. Dieses Protokoll ermöglicht die Kommunikation der Arenasteuerung über den Host Controller mit den Robotern und den Zugriff auf alle notwendigen Daten. Ein übersichtliches *Graphical User Interface (GUI)* unterstützt den Benutzer [66] beim Abfragen von Daten und Hochladen der Pläne, vgl. Abbildung 4.16.

4.5 Zusammenfassung

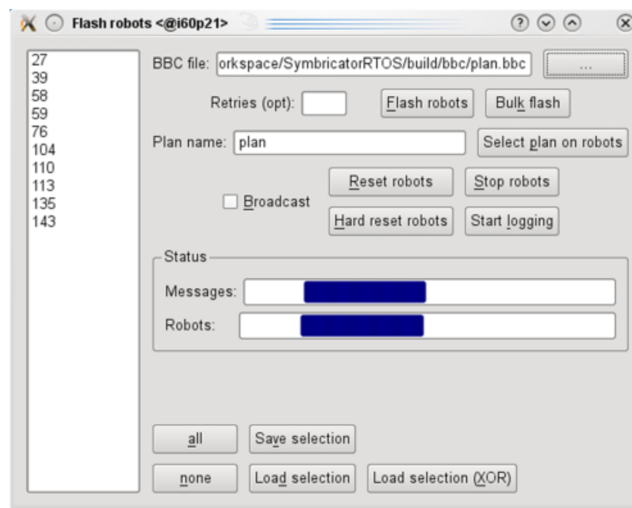
In diesem Kapitel wurde die *Entwicklungsumgebung für Roboterschwärme* beschrieben. Diese umfasst alle Komponenten, die für eine zielgerichtete und effiziente Forschung und Entwicklung notwendig sind. Die einzelnen Komponenten reichen von den implementierten Betriebssystemen, der Simulation der Roboter bis hin zu einer dynamischen, interaktiven Arena. Nach einer ausführlichen Analyse der Anforderungen, wurde eine an die Anforderungen angepasste Beschreibungssprache (MDL$\mathbf{2}\epsilon$) für Robotersteuerungen formal beschrieben und deren Einbettung in die jeweiligen Roboterbetriebssysteme dargestellt.

Die Beschreibung des ODeM-Sensors zusammen mit der dynamischen, interaktiven Arena für Schwarmroboterexperimente vervollständigt das Kapitel.

Die in diesem Kapitel beschriebene Entwicklungsumgebung stellt in ihrer Gesamtheit ein einzigartiges Mittel zum Entwurf und zur Implementierung und Analyse von Schwarmverhalten im Bereich autonomer Roboter dar. Sie unterstützt den Forscher und Entwickler in jeder Phase von der Konzeption bis hin zur Realisierung im Experiment.



(a) Abfragen von Daten



(b) Hochladen von MDL2ϵ-Plänen

Abbildung 4.16: GUI zum Abfragen/Speichern von Daten und zum Hochladen von MDL2ϵ-Plänen via ZigBee.

5. Genetische Programmierung

Genetische Programmierung ist eine Art der Evolutionary Computation, bei der die Genome der Individuen in der Population aus ausführbaren Programmen bestehen. In dem hier betrachteten Fall werden diese Programme in MDL2ϵ dargestellt. Die Verwendung von GP soll den Entwickler bei dem Design von Steueralgorithmen unterstützen. Das zu diesem Zweck entwickelte Rahmenwerk wird *Framework for Learning and Self-Organisation (FLSO)* genannt und ist in MDL2ϵApp eingebunden.

Ziel der vorliegenden Arbeit ist deshalb grundlegend zu untersuchen, ob GP auf MDL2ϵ für diesen Zweck eingesetzt werden kann. Eine Optimierung bestehender GP-Verfahren steht daher nicht im Vordergrund. In der vorliegenden Arbeit wird sich deshalb stark auf die von Koza [61] vorgestellte Art der GP bezogen. Dieser beschreibt den generellen Ablauf von GP wie in Abbildung 5.1 in zwei Phasen.

Initialisierungsphase In der Initialisierung wird die erste Population von Individuen erzeugt. Die richtige Initialisierung ist ein wichtiger Kernpunkt von GP und wirkt sich stark auf den Verlauf der Evolution aus. Eine möglichst ausgewogene Initialisierung über dem Suchraum verbessert die Konvergenz des Verfahrens hin zu einem globalen Extremum. Schlecht initialisierte Populationen können leicht in einem lokalem Extremum stecken blei-

ben. Es gibt verschiedene Verfahren der Initialisierung, die in Abschnitt 5.2 beschrieben werden.

Evolutionsphase Die eigentliche Evolution besteht aus:

- der *Evaluation der Fitness* der Individuen,
- dem *Abbruchtest* (Zeit abgelaufen, hinreichende Fitness erreicht),
- der *Erzeugung* einer neuen Population aus der vorherigen.

Die Fitness f_i eines Individuums i stellt eine relatives Maß dar, mit dem die Güte der Lösungen in einer Population zueinander bewertet werden können. Nach der Berechnung der Fitness der einzelnen Individuen werden bei der Erzeugung der neuen Population verschieden Selektionsstrategien angewendet, um die geeignetsten Individuen aus der alten Population zu selektieren. Auf die in dieser Arbeit untersuchten Selektionsstrategien wird in Abschnitt 5.4 genauer eingegangen. Nach der Selektion wird eine neue Population aus den selektierten Individuen, durch Anwendung der genetischen Operatoren Kopieren, Mutieren und Rekombinieren erzeugt. Die implementierten Operatoren werden in Abschnitt 5.3 beschrieben.

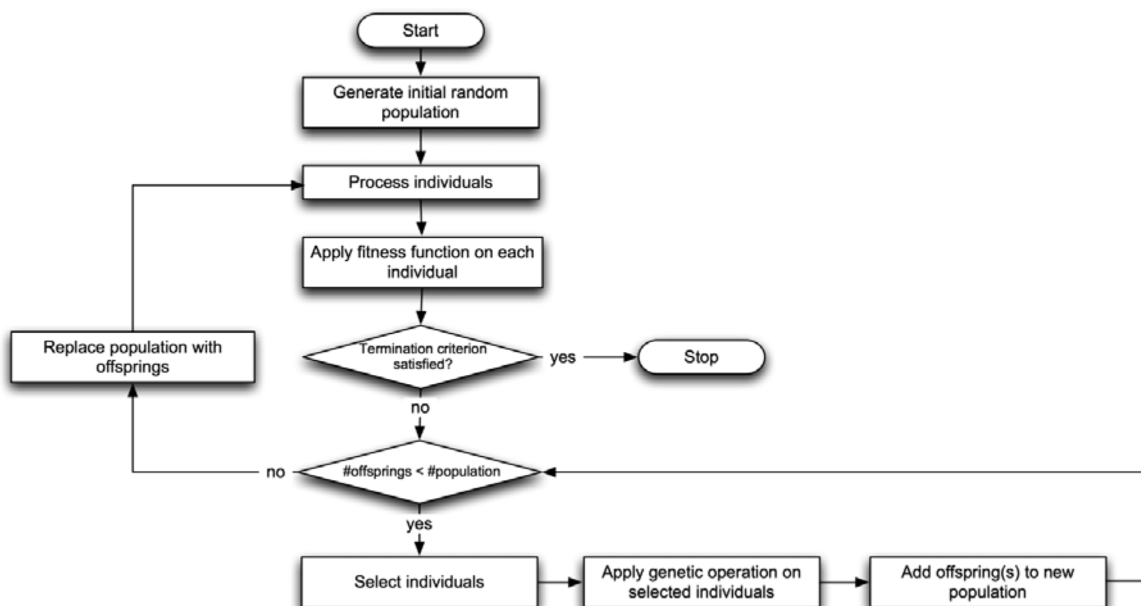


Abbildung 5.1: Genereller Ablauf des GP Meta-Algorithmus [33].

GP auf MDL2ϵ in der Schwarmrobotik

Um den Anforderungen der Schwarmrobotik gerecht zu werden, wurden zwei verschiedene Betrachtungsweisen von Populationen in GP identifiziert und in das System eingebaut. Es wird zwischen *konkurrierenden* (engl. *competing*) und *nicht konkurrierenden* (engl. *non-competing*) Individuen unterschieden. Dabei stellt bei den *konkurrierenden* Individuen jeder Roboter ein Individuum der Population dar. Jeder Roboter in der Simulation hat daher einen anderen MDL2ϵ-Plan zur Steuerung. Wir bezeichnen dies als konkurrierend, da die Fitness der Roboter/Individuen in der gleichen Arena evaluiert wird. Bei *nicht konkurrierenden* Individuen stellen alle Roboter in einer Arena ein Individuum der Population dar. D.h. jeder Roboter in einer Arena hat denselben MDL2ϵ-Plan und folgt somit den gleichen Regeln.

Diese Unterscheidung ist im Sinne der Schwarmrobotik sinnvoll, da kooperatives Verhalten oft auf gleiches Verhalten zurück geht. Verhalten, die einen nur geringen bis gar keinen Anteil an Kooperation voraussetzen, wie z.B. Kollisionsvermeidung, können somit konkurrierend laufen. Dies erhöht zum einen die Verarbeitungsgeschwindigkeit, da viele Individuen gleichzeitig getestet werden können, und erhöht zum anderen die Robustheit der gefundenen Lösungen, da schlechtere Individuen eine Störgrösse auf besseren Individuen ausüben. Verhalten, die einen erhöhten Anteil an Kooperation voraussetzen, werden bevorzugt in der nicht konkurrierenden Variante evolviert.

Da der Rechenaufwand für nicht konkurrierende Individuen recht gross ist (viele Individuen aus Schwärmen mit Dutzenden von Robotern müssen evaluiert werden) ist es sinnvoll, die Berechnung für beide Fälle zu parallelisieren. Dies geschieht in dem implementierten Rahmenwerk über eine Master-Slave-Architektur, mit der die Evaluation der Individuen auf mehrere Rechner ausgelagert wird. Der Ablauf dieser verteilten Art der GP mit konkurrierenden und nicht konkurrierenden Individuen wird in Abbildung 5.2 dargestellt. Die Anzahl der gleichzeitig verwendete Arenen ist direkt abhängig von der Leistung der verwendeten Rechner. Abschnitt 5.1.2 beschreibt die hierfür entwickelte Softwarearchitektur.

5.1 Softwarearchitektur

Die Software für GP gliedert sich in zwei Kernteile. Ein Teil verwaltet den Ablauf der GP wie in Abbildung 5.1. Der andere Teil kümmert sich um die

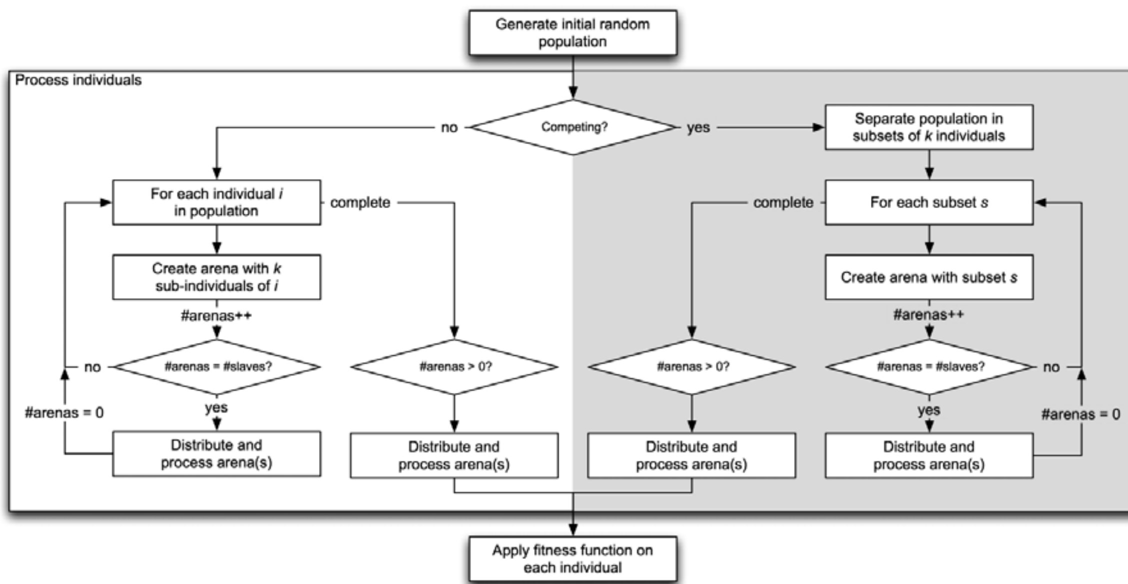


Abbildung 5.2: Verteilter Ablauf von GP unter der Berücksichtigung von konkurrierenden und nicht konkurrierenden Robotern [33].

Verteilung der Individuen auf die entsprechenden Arenen, die auf verschiedenen Rechnern gestartet worden sind.

5.1.1 Anbindung von GP an MDL2ϵ

Ziel der Softwarearchitektur der GP-Schnittstelle ist es, sich nahtlos in die Architektur des bestehenden Systems einzugliedern. Daher wurden zu den bestehenden Schnittstellen von MDL2ϵApp nur zwei Klassen (GP_MDLeTournament, GP_MDLeArena) hinzugefügt, die die Schnittstelle zur Simulation und den Ablauf der GP implementieren. Dabei ist GP_MDLeTournament für den Ablauf zuständig. Ein XML-basiertes Konfigurationsskript setzt alle wichtigen Parameter. Eine Auswahl von Parametern aus dem Konfigurationsskript ist in Tabelle 5.1 aufgelistet. GP_MDLeArena ist die Schnittstelle zu der Simulationsumgebung und wird durch GP_MDLeBreveArena implementiert. Abbildung 5.3 zeigt die Verbindung zwischen den Schnittstellen und die Funktionalität, die diese im FLSO übernehmen.

5.1.2 Verteilte Master-Slave GP Architektur

Betrachtet man eine Anfangspopulation von 1000 Individuen, die über 50 Runden evolvieren sollen, so enthält ein vollständiger Durchlauf der Evo-

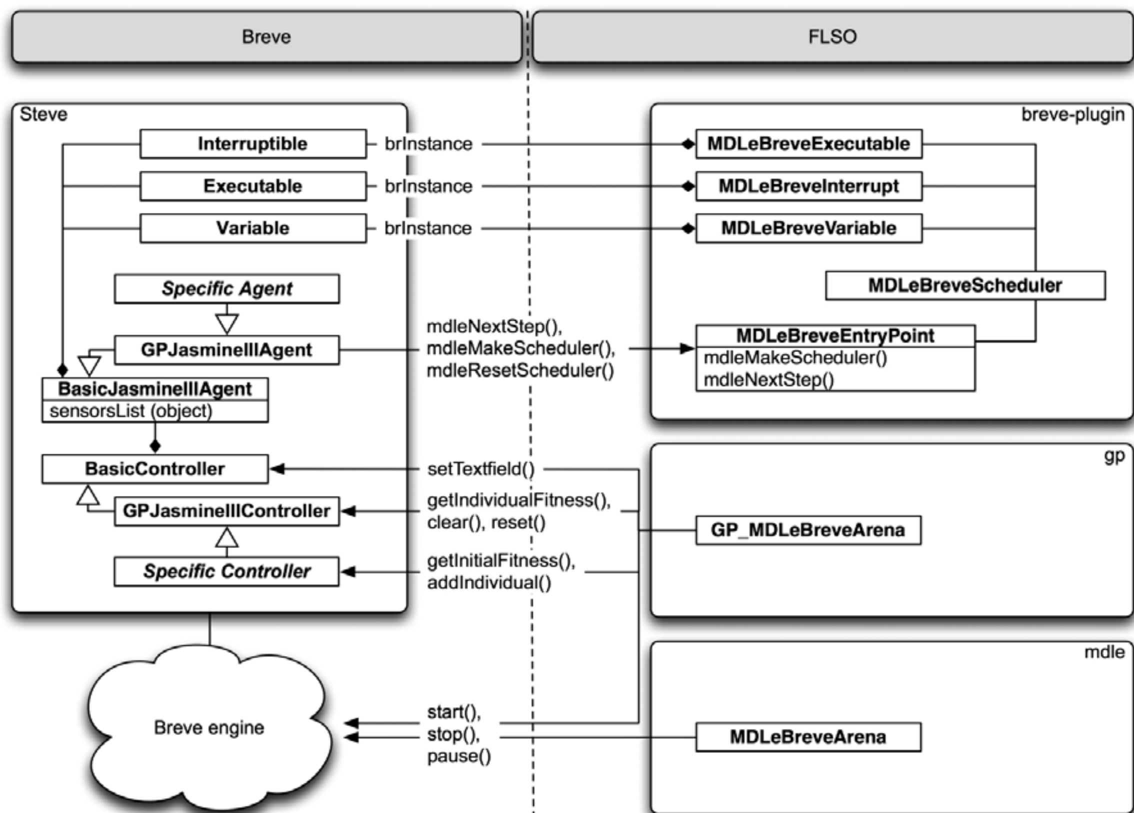


Abbildung 5.3: Funktionale Schnittstelle zwischen der Simulation in Breve und dem Rahmenwerk für GP [33].

Name	Typ	Wert	Beschreibung
competing	int	1	konkurrierender/nicht konkurrierender Modus (1/0)
populationPerArenaComp	int	15	Anzahl an Individuen in einer konkurrierenden Arena
arenasComp	int	63	Anzahl an konkurrierender Arenen pro Generation
totalPopulationComp	-	-	Gesamtanzahl an Individuen pro Generation = $populationPerArenaComp \cdot arenasComp$.
totalPopulationNonComp	int	100	Anzahl an Individuen pro Generation im nicht konkurrierenden Modus
agentsPerArenaNonComp	int	30	Anzahl an Robotern in einer nicht konkurrierenden Arena
rounds	int	35	Anzahl der Generationen
duration	double	600	simulierte Zeit in Sekunden
replacement	string	elitist	die Erseztungsstrategie
eliteNumber	int	20	Anzahl an Elitisten
selection	string	fps	Selektionsstrategie (“fps”, “rbs” oder “tms”)
crossover	double	0,8	Kreuzungsrate
mutation	-	-	Mutationsrate (ist immer $1 - crossover$).
neighbourMutRate	double	0,1	Mutationsrate für Neighbour Mutation
subtreeMutRate	double	0,1	Mutationsrate für Subtree Mutation.
save_to_xml	int	1	speichern der Individuen in XML (0/1)

Tabelle 5.1: Standardparameter für GP.

lution $50 \cdot 100 = 50.000$ Individuen, deren Fitness bestimmt werden muss. Betrachten wir die Laufzeit einer einzelnen Runde von 600 simulierten Zeitschritten, die ungefähr 6 Minuten Rechenzeit benötigen, so braucht ein vollständiger Durchlauf bei 20 Individuen pro Arena $\frac{50.000}{20} = 2500$ Arenen. Dies entspricht $2500 \cdot 6 = 15.000$ Minuten oder ungefähr 10,4 Tage auf einem einzelnen Rechner. Daher ist es wichtig diesen Prozess möglichst zu parallelisieren. Dies erreichen wir durch eine Master-Slave-Architektur, wie sie in Abbildung 5.4 dargestellt ist.

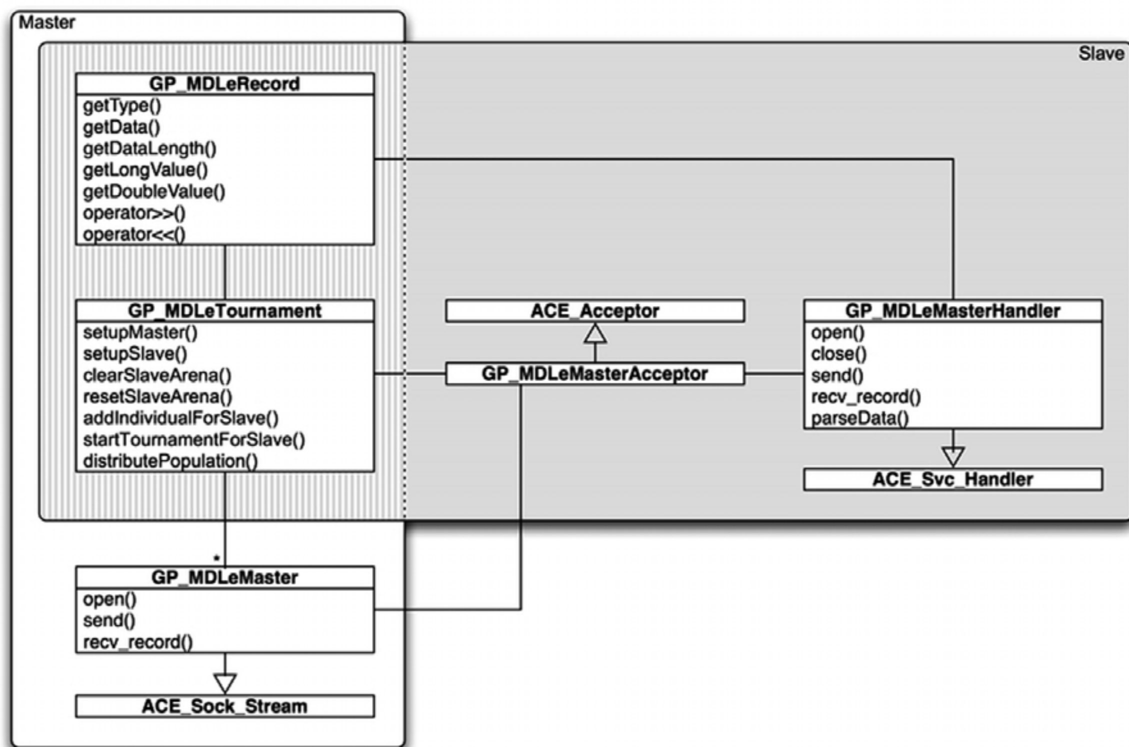


Abbildung 5.4: Darstellung der Master-Slave-Architektur [33].

Der *Master* steuert den gesamten evolutionären Prozess und verteilt die Individuen als XML-MDL2ϵ-Plan auf die *Slaves*. Diese berechnen die Fitness der erhaltenen Individuen und schicken diese zurück an den Master. Um zu verhindern, dass durch einen defekten Slave ein gesamtes Experiment gestört wird, wartet der Master nur eine gewisse Zeit auf seine Slaves. Schicken diese keine Antwort, so verteilt er die Individuen der ausgefallenen Slaves an einen anderen. Der Master selbst wird in die Berechnung der Fitness auch miteinbezogen.

Diese Architektur hat sich im universitären Betrieb, in dem mehrere Benutzer auf Rechner Zugriff haben, die als Slave konfiguriert wurden, als sehr robust und gut skalierbar erwiesen. Gegen einen Defekt beim Master ist das System nicht geschützt.

5.2 Initiale Population

Die Auswahl der initialen Population ist wichtig für den Verlauf der GP. Sie sollte den Lösungsraum möglichst gut abdecken. Bei GP ist es jedoch nur schwer zu bewerten, ob eine Population den Lösungsraum abdeckt, da dies nicht aus den Programmen ablesbar ist. Daher versucht man bei GP möglichst viele verschiedene Programmbäume zu generieren. Die gebräuchlichsten drei Methoden sind *Full, Grow* und *Ramped Half-and-Half.*

Full Erzeugt einen Programmbaum, bei dem jeder Pfad zwischen der Wurzel und den Blättern die gleiche Länge l hat.

Grow Erlaubt Bäume mit unterschiedlich langen Pfaden, aber einer maximalen Länge l.

Ramped Half-and-Half Ist eine Mischung aus der Full und Grow-Methode. Ein Parameter $l > 2$ legt Grenzen für das Wachstum fest, die von 2 bis l gehen. Basierend auf diese Grenzen werden jeweils für jeden Grenzparameter eine Hälfte der Bäume mit der Grow und die andere Hälfte der Bäume mit der Full-Methode erzeugt.

Die Erzeugung von MDL2ϵ-Plänen wird mit einer angepassten Grow-Methode realisiert, die mehr Einflussmöglichkeiten auf die erzeugten Bäume als die ursprüngliche Methode zulässt. Zusätzlich zu dem Parameter für die Tiefe, *maxDepth*, werden Parameter für die maximale Länge einer Konkatenation, *maxWidth*, und für die maximale Anzahl von Knoten, *maxNodes*, eingeführt. Übertragen auf einen regulären Ausdruck bestimmen dabei *maxDepth* die maximale Verschachtelung von klammernden Operatoren, *maxWidth* die maximale Anzahl an konkatenierten Operatoren und Atomen in einer Klammer und *maxNodes* die maximale Gesamtanzahl an Operatoren und Atomen, die in dem regulären Ausdruck verwendet werden.

Zusätzlich wird eine Wahrscheinlichkeitsverteilung für die Erzeugung der einzelnen Bausteine und die Maximalwerte der Attribute von MDL2ϵ angegeben, vgl. Tabelle 5.2. Interrupts werden hier nur aus einfachen Inter-

Name	**Typ**	**Wert**	**Beschreibung**
maxDepth	int	5	maximale Tiefe eines Plans
maxWidth	int	10	maximale Anzahl an Konkatenationen
maxNodes	int	40	maximale Anzahl an Knoten insgesamt
maxDuration	long	100	maximale Dauer eines ATOM/BEHAVIOUR
maxMultiplicity	int	100	maximale Anzahl an Wiederholungen einer Multiplizität
maxProbability	double	255	maximaler Wahrscheinlichkeitswert
probInfMult	double	0,1	Wahrscheinlichkeit für unendlichen MULT-Knoten
probInfDur	double	0,1	Wahrscheinlichkeit für unendliche Dauer eines ATOM/BEHAVIOUR
probBeh	double	0,3	Wahrscheinlichkeit für die Erzeugung eines BEHAVIOUR-Knotens
probAtom	double	0,4	Wahrscheinlichkeit für die Erzeugung eines ATOM-Knotens
probMult	double	0,1	Wahrscheinlichkeit für die Erzeugung eines MULT-Knotens
probUnion	double	0,1	Wahrscheinlichkeit für die Erzeugung eines UNION-Knotens
probRunion	double	0,1	Wahrscheinlichkeit für die Erzeugung eines RUNION-Knotens

Tabelle 5.2: Parameter für die Erzeugung der initialen Population.

rupts und deren Negation erzeugt. Dies schränkt die Möglichkeiten jedoch nicht ein, da komplexere Interrupts durch den MDL2ϵ-Ausdruck selbst erzeugt werden. Eine UND-Verknüpfung kann z.B. durch mehrere aufeinander folgende Verhalten und eine ODER-Verknüpfung durch einen Konkatenation ausgedrückt werden.

5.3 Genetische Operatoren

Die *Genetischen Operatoren: Reproduktion, Mutation* und *Crossover* sind die Grundlage der künstlichen Evolution. Durch die Anwendung dieser Operatoren wird eine neue Population erzeugt, die gewährleisten soll, dass sich die Genome mit besserer Fitness durchsetzen, und ihre bessere Fitness auch an ihre Nachfahren vererben. Die Genetischen Operatoren müssen jeweils an das zugrunde liegende Genom angepasst werden. Im Besonderen gilt dies für den Crossover und die Mutation.

5.3.1 Reproduktion

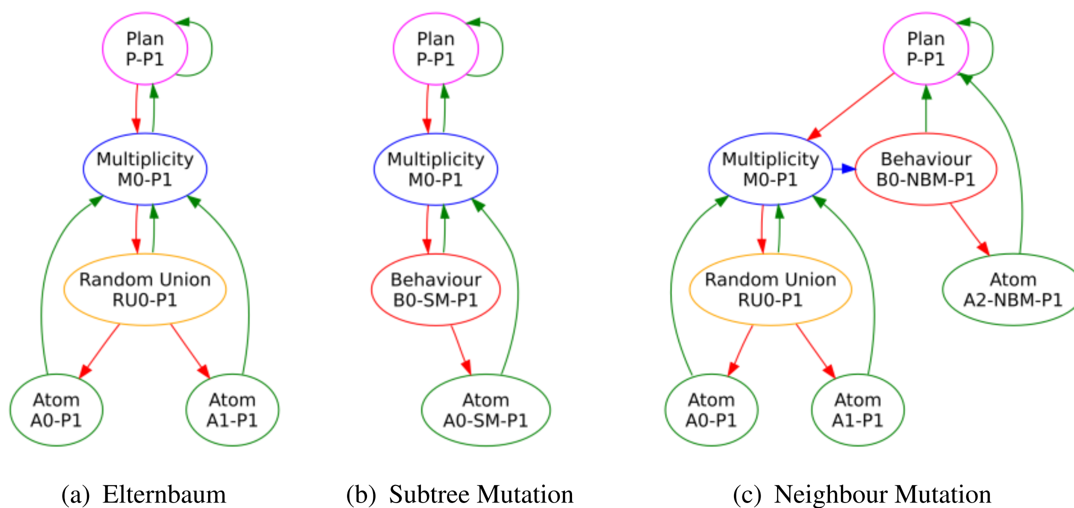
Die *Reproduktion* ist ein einfacher unärer Operator, der eine Kopie/Klon des Elternteils erzeugt. Innerhalb des entwickelten Systems wird er für die *Elitist*-Methode verwendet, bei der die besten n Individuen direkt in die neue Population übernommen werden.

5.3.2 Mutation

Die *Mutation* ist ebenfalls ein unärer Operator, der eine veränderte/mutierte Version des ursprünglichen Genoms erzeugt. Für MDL2ϵ wurden zwei verschiedene Mutationsoperatoren implementiert, die *Subtree* und *Neighbour Mutation*.

Subtree Mutation Bei dieser Art der Mutation wird ein beliebiger Unterbaum (engl. *subtree*) durch einen neuen zufällig erzeugten Unterbaum ersetzt. Die Erzeugung des Unterbaums findet, wie bei der Erzeugung der initialen Population, nur mit anderer Parametrisierung statt. Die Erzeugung der Unterbäume wird hier mit $maxDepth = 3$, $maxWidth = 3$ und $maxNodes = 9$ parametrisiert. Dadurch erhält man deutlich kleinere Bäume als bei der Erzeugung einer initialen Population. Es ermöglicht das Wachstum bestehender Bäume in die Tiefe. Abbildung 5.5(b) zeigt die *Subtree Mutation* anhand eines Beispielbaumes.

Neighbour Mutation Hier wird kein Teil des bestehenden Programmbaumes ersetzt, sondern ein Nachbar (engl. *Neighbour*) zu einem bestehenden Knoten hinzugefügt. Als Nachbarn werden hier Knoten bezeichnet, die mit einander konkateniert sind. Diese Mutation erlaubt ein Wachstum des Baums in die Breite. Abbildung 5.5(c) zeigt die *Neighbour Mutation* anhand eines Beispielbaumes.



(a) Elternbaum (b) Subtree Mutation (c) Neighbour Mutation

Abbildung 5.5: Beispiele für Subtree und Neighbour Mutation.

Eine Mutation, die die Attribute einzelner Knoten oder den dort beschriebenen Operator ändert, wurde aufgrund der Komplexität und der dadurch dazu kommenden Anzahl zusätzlicher Parameter verworfen.

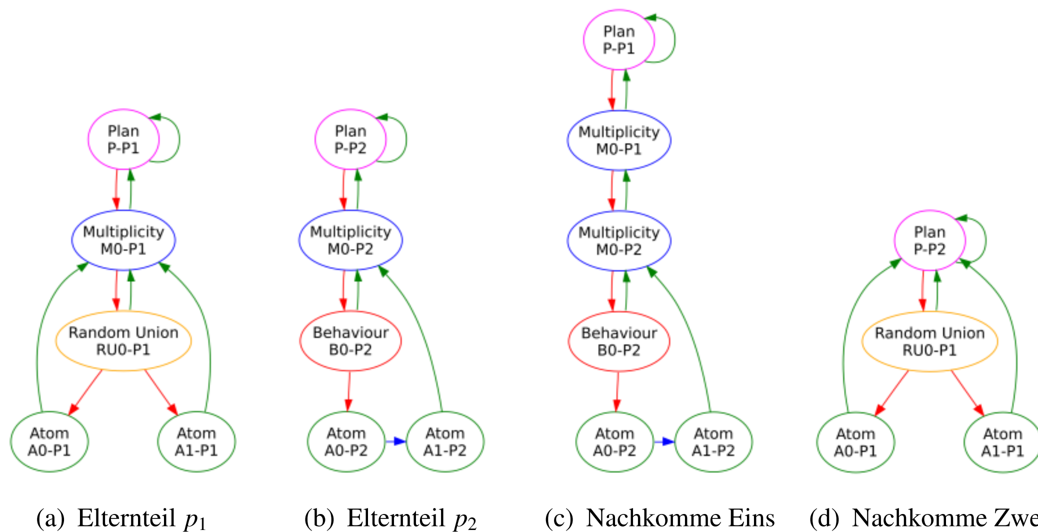
5.3.3 Crossover

Der *Crossover*-Operator ist ein binärer Operator, der zwei Genome miteinander kreuzt, indem er zufällig ausgewählte Teilbäume kreuzweise austauscht. Hierdurch sollen fitte Individuen ihre guten Teilgenome vererben, was zu einer Konvergenz von fitten Verhalten führen soll. Abbildung 5.6 zeigt die Anwendung des Crossover-Operators auf zwei Eltern. Da die Auswahl auf einen Kreuzungspunkt beschränkt wird, spricht man auch vom *Single-Point-Crossover*. In der Literatur findet man aber auch Varianten mit *N* Kreuzungspunkten und bezeichnet diese daher als *N-Point-Crossover*.

Der Crossover-Operator trägt zu einem schnellen Wachstum von Plänen während der Evolution bei. Doch zum Ende der Evolution stagniert das Verfahren, da Mutation und Crossover mit einer wachsenden Wahrscheinlichkeit auf Teilbäumen durchgeführt werden, die nicht zur Änderung

des Verhaltens des Roboters beitragen. Eine Möglichkeit dies zu verringern, ist das Abschneiden (engl. *prune*) nicht verwendeter Teilbäume vor der Anwendung der genetischen Operatoren. Dieses Verfahren wird in Abschnitt 5.5 beschrieben.

Dieses “ungezügelte” Wachstum von Plänen während der Evolution ist ein Grund für die Verwendung eines interpreterbasierten Ansatzes, wie in der Analyse in Abschnitt 4.1 beschrieben.



(a) Elternteil p_1 (b) Elternteil p_2 (c) Nachkomme Eins (d) Nachkomme Zwei

Abbildung 5.6: Anwendung des Crossover-Operators auf zwei Elternteile.

5.4 Selektionsstrategien

Die Selektionsstrategie beschreibt die Auswahl von Individuen einer Population zur Erzeugung einer neuen. Selektionsstrategien sollten daher fittere Individuen bevorzugen, aber auch hinreichend über die gesamte Population streuen, um die Diversität nicht zu stark zu verringern. Eine Selektion, die zu stark die besten Individuen bevorzugt, kann zu einer zu frühen und daher unerwünschten Konvergenz der Genome in einem lokalen Extremum führen. Daher wurden drei, in der Literatur häufig verwendete, Selektionsstrategien implementiert und analysiert. Zu diesem Zweck wurden die *rangbasierte Selektion (RBS)*, die *fitnessproportionale Selektion (FPS)* und die *Turnierselektion (TMS)* ausgewählt.

5.4.1 Fitnessproportionale Selektion

Die fitnessproportionale Selektion wurde von John Holland [48] vorgeschlagen. Sie setzt die Selektionswahrscheinlichkeit mit der Fitness in Proportion zu einander. Die Selektionswahrscheinlichkeit p_i eines Individuums i aus der Population Pop berechnet sich anhand der Fitness f_i als:

$$p_i = \frac{f_i}{\sum_{j \in Pop} f_j} \tag{5.1}$$

Ein Nachteil, der sich aus Gleichung (5.1) direkt ablesen lässt, ist, dass die Selektionswahrscheinlichkeit stark durch Ausreisser in der Fitness beeinflusst wird. Dies kann eine frühzeitige Abnahme der Diversität der Population zur Folge haben, und dadurch zu einer unerwünschten frühzeitigen Konvergenz in einem lokalen Extremum führen.

5.4.2 Rangbasierte Selektion

Die rangbasierte Selektion [3] versucht den Nachteil der FPS zu umgehen. Dabei wird jedem Individuum i anhand seiner Fitness f_i ein Rang r_i zu gewiesen, der über eine Funktion $g(r_i)$ mit einem Wert verbunden ist. Dieser Wert ersetzt dann die Fitness f_i in Gleichung (5.1). Der Rang r_i ist eine Ordnung der Population basierend auf der Fitness, die $r_i \geq r_j \Leftrightarrow f_i \geq f_j$ erfüllt.

Für die Funktion $g(r_i)$ sind zwei verschieden Funktionen realisiert:

Negativ Exponentielle Verteilungsfunktion

$$g(r_i) = a \cdot e^{-(b \cdot r_i + c)}, \text{ mit } a,b,c \in \mathbb{R} \tag{5.2}$$

Lineare Verteilungsfunktion

$$g(r_i) = a \cdot r_i - a \cdot |Pop|, \text{ mit } a \in \mathbb{R} \tag{5.3}$$

Die RBS verstärkt, in Fällen von nur schwach ausgeprägten Unterschieden in der Fitness, leichte Fitnessunterschiede und unterdrückt die Dominanz eventuell vorhandener Individuen mit sehr hoher Fitness. Dies führt nach [114] zu besseren Ergebnissen.

5.4.3 Turnierselektion

Bei der Turnierselektion werden n zufällig ausgewählte Individuen in einem Turnier zusammengefasst. Der Gewinner des Turniers ist das Individuum mit der besten Fitness und wird für die Fortpflanzung ausgewählt.

Dieser zweistufige Selektionsprozess kann sich negativ auf die Diversität der nächsten Generation auswirken. Manche Individuen werden unabhängig von ihrer Fitness gar nicht in ein Turnier gewählt, und üben somit keinen Einfluss auf die nächste Generation aus. Es gibt Bemühungen, den hierdurch entstandenen Verlust an Diversität aufzuheben [102].

Ein Vorteil der TMS ist jedoch, dass der Selektionsdruck leicht über die Turniergröße n angepasst werden kann. Kleine n verringern dabei den Selektionsdruck und ermöglichen dadurch auch weniger fitten Individuen, ein Turnier zu gewinnen.

5.5 Pruning

Während der Evolution kann es, durch den Crossover mit großen Unterbäumen, zu einem nicht unerheblichen Wachstum der Steuerprogramme kommen. Dies lähmt zunehmend den Fortschritt der Evolution. Die Lähmung ist darauf zurückzuführen, dass die Operatoren auf Teilen der Steuerung arbeiten, die nicht besucht werden können, da z.B. ineinander geschachtelte Verhalten widersprüchliche Interrupts aufweisen, die die Ausführung des Unterbaums verhindern. Bei zunehmender Größe steigt die Anzahl solcher toten Abschnitte in der Steuerung. Es steigt die Wahrscheinlichkeit, dass die genetischen Operatoren auf solchen toten Bäumen arbeiten und damit keinen Einfluss auf das Verhalten der neu erzeugten Individuen ausüben. Um dies zu verhindern, verwendet man das so genannte *Pruning*, bei dem Teilbäume aus der Steuerung herausgeschnitten werden.

Die Auswahl der zu entfernenden Teilbäume richtet sich nach dem Einfluss eines Teilbaums auf die Fitness des Individuums. Den Einfluss berechnet man dadurch, dass während der Evaluation der Fitness allen MDL2ϵ-Atomen ein zusätzlicher Wert zugewiesen wird, der bei dem aktiven Atom zu jedem Zeitschritt erhöht wird. Das Pruning entspricht somit der Berechnung der Spur eines Planes aus Definition 4.15.

Bei der Anwendung des Pruning wird angenommen, dass Teile, die lange aktiv waren, auch einen grösseren Einfluss auf die Fitness haben. Dies ist je nach Beschaffenheit der Fitnessfunktion nicht immer der Fall, kann aber

im Allgemeinen als grobe Annäherung herangezogen werde. Der Faktor γ in Gleichung (4.1) kann verwendet werden, um die in diesem Schritt erhaltene Belohnung in die Spur einzuarbeiten, damit die Spur die enthaltene Fitness besser abbildet. Dies ist eine Anlehnung an *Reinforcement Learning* bzw. *Reinforced Genetic Programming*.

5.6 Diversität von MDL2ϵ-Plänen

Wie schon angesprochen, hat die Diversität einer Population einen Einfluss auf die Güte der Evolution und deren Konvergenzgeschwindigkeit. Daher ist die Diversität ein wichtiges Maß, die Auswirkung von Selektionsstrategien und Pruning auf die Evolution zu analysieren [107].

Es gibt verschiedene Arten der Diversität, die sich zumeist auf zwei verschiedene Aspekte beziehen: die *strukturelle Diversität* und die *Verhaltensdiversität*. Die strukturelle Diversität bezieht sich auf die Struktur der Programmbäume, die Verhaltensdiversität auf die durch die Steuerung erzeugten Verhalten. Dies spiegelt sich oft auch in der strukturellen Diversität wider. Eine detaillierte Analyse von verschieden Diversitätsmaßen wurde von Burke et al. in [17] durchgeführt. Einige Quellen in der Literatur unterstützen die Annahme, dass eine ausreichend hohe Diversität eine zu frühe Konvergenz in einem lokalem Extremum verhindern kann [31, 74].

Um ein geeignetes Diversitätsmaß für MDL2ϵ anzugeben, betrachten wir nur die Struktur der ausgeführten Teile eines MDL2ϵ-Plans Γ, also die Spur$(\Gamma, 0)$. Hierdurch wird sichergestellt, dass nur die Teile, die zu dem Verhalten des Plans beitragen, zu der Diversitätsberechnung herangezogen werden. Die strukturelle Diversität, wird in Richtung der Verhaltensdiversität verschoben.

5.6.1 Paarweise Ähnlichkeit

Die Berechnung der Diversität einer Population geht auf die *paarweise Ähnlichkeit*, *sim*(*a*,*b*), von je zwei Individuen *a* und *b* zurück. Die Berechnung der Ähnlichkeit findet auf der in Abschnitt 4.2.5 beschriebenen Bytecode-Darstellung statt. Diese eignet sich sehr gut für eine Korrelationsanalyse. Hierfür muss jedoch der Bytecode angepasst werden, um die Redundanz zu verringern. Bei dem angepassten Bytecode werden z.B. die 16-Bit-Adressen der Verhalten entfernt, da sie keine Aussage über das Verhalten des Roboters enthalten.

$byteCode_a(k)$ und $byteCode_b(j)$ sind die Werte im Bytecode an den Stellen k und j von Individuum a und b. Ohne Beschränkung der Allgemeinheit wird im Folgenden angenommen, dass der Bytecode von Individuum b kürzer oder gleich lang ist wie der von Individuum a. Eine Kreuzkorrelation mit dem logischen Operator XNOR ($\overline{\oplus}$) ist gegeben durch die folgende Gleichung (5.4).

$$\begin{aligned} cc_{a,b}(k) &= \sum_j d_j \\ j &= 1, 2, \ldots, |byteCode_b| \\ k &= 1, 2, \ldots, |byteCode_a| \\ d_0 &= 0 \\ d_j &= \begin{cases} 0 & \text{, wenn } cmp(a,b,j,k) = 0 \\ 1 & \text{, wenn } cmp(a,b,j,k) = 1 \\ & \ \wedge d_{j-1} = 0 \\ (1+\alpha) \cdot d_{j-1} & \text{, wenn } cmp(a,b,j,k) = 1 \\ & \ \wedge d_{j-1} \neq 0 \end{cases} \\ \alpha &\in (0,1] \end{aligned} \tag{5.4}$$

$$cmp(a,b,j,k) = \overline{byteCode_b(j) \oplus byteCode_a((j+k) \mod |byteCode_a|)} \tag{5.5}$$

Hierbei ergibt Gleichung (5.5) nur dann 1, wenn $byteCode_b(j)$ und $byteCode_a((j+k) \mod |byteCode_a|)$ gleich sind, wobei $|byteCode_a|$ die Anzahl der Bits in dem Bytecode ist. Ohne den Gewichtungsfaktor $(1+\alpha)$ beschreibt $cc_{a,b}(k)$ die Anzahl der übereinstimmenden Bits, wenn der Bytecode von b mit dem von a an der Stelle $byteCode_a(k)$ übereinander gelegt werden. Durch den Modulo-Operator werden überlappende Bits abgeschnitten und mit dem Anfang von a überlagert. Der Gewichtungsfaktor $(1+\alpha)$ führt zusammen mit d_j zu einem exponentiellen Wachstum von $cc_{a,b}(k)$, wenn eine lange Bitsequenz übereinstimmt. Hierfür wird d_j mit 0 initialisiert und bei jeder Übereinstimmung exponentiell erhöht. Stimmen zwei Bits nicht überein, so wird wieder $d_j = 0$ gesetzt.

Eine obere Grenze für $cc_{a,b}(k)$ kann mit Gleichung (5.6) berechnet werden.

$$
\begin{aligned}
cc_{a,b}^{max} &= (1+\alpha)^0 + (1+\alpha)^1 + \cdots + (1+\alpha)^{|byteCode_b|-1} \\
&= \frac{1-(1+\alpha)^{|byteCode_b|}}{1-(1+\alpha)} \\
&= \frac{(1+\alpha)^{|byteCode_b|}-1}{\alpha} \\
\alpha &\in (0,1]
\end{aligned}
\tag{5.6}
$$

In Gleichung (5.6) wird für die Berechnung die n-te Teilsumme der geometrischen Reihe

$$\sum_{k=0}^{n} a_0 q^k = a_0 \frac{1-q^{n+1}}{1-q} \tag{5.7}$$

verwendet. Die obere Grenze $cc_{a,b}^{max}$ wird erreicht, wenn für jede Position k der Bytecode von b mit dem a vollständig übereinstimmt. $cc_{a,b}$ ist ein Maß für die Ähnlichkeit der Individuen a und b der Population Pop.

Da sich die Bytecodes der Individuen in der Länge unterscheiden, diese aber ein wichtiger Faktor in der Berechnung der Ähnlichkeit zweier Individuen darstellt, ist es notwendig einer Normierung der Ähnlichkeit $cc_{a,b}$ durchzuführen.

Eine einfache Normierung wäre $\frac{cc_{a,b}(k)}{cc_{a,b}^{max}}$. Dies hätte jedoch, aufgrund des exponentiellen Wachstums von d_j und des daraus resultierenden großen Wertes für $cc_{a,b}^{max}$, einige Nachteile. Wählt man α zu groß so kann dies zur Unterdrückung signifikanter Information führen. Die längste Übereinstimmung in $cc_{a,b}$ würde alle kürzeren Übereinstimmungen dominieren, was deren Anteil am Ähnlichkeitsmaß vernachlässigen würde. Des Weiteren wäre es auch nicht möglich, die paarweise Ähnlichkeit zur Berechnung des Mittelwertes der gesamten Population heranzuziehen. Die normierten Werte von $cc_{a,b}(k)$ wären aussagelos, wenn sich die Länge des Bytecodes der korrespondierenden Individuen unterscheiden würde.

Aufgrund dieser Überlegungen wurde eine andere Normierung gewählt. Zunächst wird die theoretische Anzahl der übereinstimmenden Bits $\overline{cc}_{a,b}(k)$ berechnet. Hierbei wird keine Rücksicht darauf genommen, ob es sich dabei um mehrere kurze oder eine lange Übereinstimmung handelt. Dafür wird die Geometrischen Reihe aus Gleichung (5.7) nach n aufgelöst, wobei $n = \overline{cc}_{a,b}(k)$, $q = (1 + \alpha)$ und $a_0 = 1$ gesetzt wird.

$$\begin{aligned} cc_{a,b}(k) &= \frac{(1+\alpha)^{\overline{cc}_{a,b}(k)} - 1}{\alpha} \\ \Leftrightarrow \quad \alpha \cdot cc_{a,b}(k) + 1 &= (1+\alpha)^{\overline{cc}_{a,b}(k)} \\ \Leftrightarrow \quad \overline{cc}_{a,b}(k) &= \log_{(1+\alpha)}(\alpha \cdot cc_{a,b}(k) + 1) \\ \Leftrightarrow \quad \overline{cc}_{a,b}(k) &= \frac{\ln(\alpha \cdot cc_{a,b}(k) + 1)}{\ln(1+\alpha)} \\ \alpha &\in (0{,}1] \end{aligned} \tag{5.8}$$

Hierdurch werden die Filtereigenschaften von α mit in die Normierung einbezogen. Je nach Wahl von α können so die Auswirkung von kurzen und langen Übereinstimmungen beeinflusst werden. Die Wahl von α wirkt hierbei wie ein Tiefpassfilter. Verschiebt man α in Richtung $\alpha \to 1$, desto stärker wird die Filterwirkung. Die Gleichungen (5.9) veranschaulichen diese Verhalten.

$$\begin{aligned} \lim_{\alpha \to 0} \overline{cc}_{a,b}(k) &= match_{all}(k) \leq |byteCode_b| \\ \lim_{\alpha \to \infty} \overline{cc}_{a,b}(k) &= match_{longest}(k) \geq 0 \end{aligned} \tag{5.9}$$

Hierbei ist $match_{all}(k)$ die Summe der Längen aller Übereinstimmungen und $match_{longest}(k)$ die Länge der längsten Übereinstimmung an der Stelle k. $match_{all}(k)$ und $match_{longest}(k)$ beschreiben daher eine obere und untere Grenze für $\overline{cc}_{a,b}(k)$.

$$0 \leq \lim_{\alpha \to \infty} \overline{cc}_{a,b}(k) \leq \overline{cc}_{a,b}(k) \leq \lim_{\alpha \to 0} \overline{cc}_{a,b}(k) \leq |byteCode_b| \tag{5.10}$$

$\overline{cc}_{a,b}$ ist abhängig von der Länge des Bytecodes von a und b. Wie man aus Gleichung (5.10) entnehmen kann, ist $|byteCode_b|$ eine obere Grenze für $\overline{cc}_{a,b}$. Eine Normierung ist daher durch $\frac{\overline{cc}_{a,b}}{|byteCode_b|}$ gegeben.

Ein weiterer Faktor, der eine Einfluss auf $\frac{\overline{cc}_{a,b}}{|byteCode_b|}$ hat, ist das Verhältnis $\frac{|byteCode_b|}{|byteCode_a|}$. Je kürzer der Bytecode von b im Verhältnis zu a ist, desto wahrscheinlicher ist eine vollständige Übereinstimmung von b in a und um

so größer wird $\frac{\overline{cc}_{a,b}}{|byteCode_b|}$ sein. Daher wird $\frac{|byteCode_b|}{|byteCode_a|}$ als zusätzlicher Gewichtungsfaktor hinzugefügt. Die endgültige Normierung ergibt sich nun als:

$$\begin{aligned} \overline{cc}_{a,b}^{norm}(k) &= \frac{\overline{cc}_{a,b}(k)}{|byteCode_b|} \cdot \frac{|byteCode_b|}{|byteCode_a|} \\ &= \frac{\overline{cc}_{a,b}(k)}{|byteCode_a|} \end{aligned} \tag{5.11}$$

Die Ähnlichkeit zweier Individuen a und b wird durch die Summe aller $\overline{cc}_{a,b}^{norm}$ berechnet, die eine Schwellwert ϵ überschreiten. Der Schwellwert wird eingeführt, um zufällige kleine Übereinstimmungen nicht in die Berechnung der Ähnlichkeit zu übernehmen. Zur Berechnung des Schwellwertes wird eine Standardmethode für die Bestimmung von Ausreißern verwendet. Dazu wird m-mal die Standardabweichung σ von $\overline{cc}_{a,b}^{norm}$ zum Mittelwert μ addiert, vlg. Gl. (5.12). Über die Stufenfunktion Θ_ϵ gehen nur die vermeidlichen Ausreisser in die Berechnung der paarweisen Ähnlichkeit $sim(a,b)$ mit ein. Dadurch wird das durch zufällige Übereinstimmungen erzeugte Rauschen entfernt.

$$\begin{aligned} sim(a,b) &= \sum_k \Theta_\epsilon(cc_{a,b}^{norm}(k))\overline{cc}_{a,b}^{norm}(k) \\ k &= 1,2,\ldots,|byteCode_a| \\ \Theta_\epsilon(x) &= \begin{cases} 1 & \text{, wenn } x > \epsilon \\ 0 & \text{, anderenfalls} \end{cases} \\ \epsilon &= m \cdot \sigma(\overline{cc}_{a,b}^{norm}) + \mu(\overline{cc}_{a,b}^{norm}) \\ sim(a,b) &:= sim(b,a) \text{ falls } |byteCode_a| < |byteCode_b| \end{aligned} \tag{5.12}$$

5.6.2 Ähnlichkeit der Population

Nach der Bestimmung der paarweisen Ähnlichkeit wird diese zur Bestimmung der Ähnlichkeit sim_{Pop} einer Population Pop herangezogen. Diese berechnen wir als das arithmetische Mittel aller paarweisen Ähnlichkeiten.

$$sim_{Pop} = \frac{1}{2n(n-1)} \sum_{\forall a,b \in Pop\,:\, a \neq b} sim(a,b), \text{ mit } n = |Pop| \tag{5.13}$$

5.6.3 Diversität der Population

Gleichung (5.13) beschreibt eine standardisierte Form der Ähnlichkeit. Ein hypothetischer Ähnlichkeitswert von null beschreibt eine vollständig diverse Population. Größere Ähnlichkeitswerte beschreiben weniger diverse Populationen. Um die Lesbarkeit zu verbessern, formen wir die Ähnlichkeit in die Diversität um, indem wir die Ähnlichkeit von eins subtrahieren. Dabei werden Ähnlichkeitswerte, die größer als eins sind, was theoretisch möglich ist, abgeschnitten. Ein Ähnlichkeitswert größer als eins stellt eine extreme Form der Selbstähnlichkeit dar und gibt keine weitere Aussage über die Diversität der Population.

$$div_{Pop} = \begin{cases} 1 - sim_{Pop} & \text{, wenn } sim_{Pop} < 1 \\ 0 & \text{, anderenfalls} \end{cases} \quad (5.14)$$

Durch Gleichung (5.14) erhalten wir also ein Diversitätsmaß, das für eine geringe Diversität nahe bei null und für eine hohe Diversität nahe bei eins liegt. Das so definierte Diversitätsmaß kann zur Analyse und Optimierung des evolutionären Prozesses herangezogen werden.

5.7 Zusammenfassung

Evolution von Steuerprogrammen für Schwarmroboter ist ein kaum betrachteter Bereich in der Evolutionären Robotik. Die Synthese von Steuerungen basiert zumeist auf der Evolution von KNNs oder ähnlichem. Das in diesem Kapitel beschriebene Rahmenwerk ermöglicht die Synthese von Steuerprogrammen basierend auf GP. In diesem Kapitel wurde der Ablauf von GP auf MDL2ϵ und dessen Einbettung in die Entwicklungsumgebung für Roboterschwärme beschrieben. Die dazu verwendeten Selektionsmechanismen, und die für MDL2ϵ entwickelten genetischen Operatoren werden aufgeführt und erläutert. Angepasst an die Bedürfnisse der Schwarmrobotik wird eine Aufteilung in einen konkurrierenden und nicht-konkurrierenden Modus beim Ablauf der Evolution vorgeschlagen und erklärt.

Zur Optimierung des Ablaufs der Evolution wird ein Pruningverfahren für MDL2ϵ-Pläne beschrieben.

Ein neues Diversitätsmaß, welches versucht die strukturelle Diversität in Richtung der Verhaltensdiversität zu verschieben, wird am Ende des Kapitels für die Bytecode-Repräsentation von MDL2ϵ mathematisch hergeleitet.

6. Experimente und Bewertung

Wie in Abschnitt 1.3 beschrieben, wurden über den gesamte Entwicklungszeitraum immer wieder Experimente durchgeführt, um die Teilprojekte der Arbeit zu überprüfen und zu evaluieren. Dieses Kapitel zeigt einige ausgewählte Experimente. Es gliedert sich in einen Abschnitt für die Entwicklungsumgebung sowie einen für das Rahmenwerk für GP (Framework for Learning and Self-Organisation).

6.1 Entwicklungsumgebung

Dieser Abschnitt zeigt zwei ausgewählte Experimente, die im Rahmen der Evaluation der Entwicklungsumgebung durchgeführt worden sind. Tabelle 6.1 gibt einen Überblick über die in dem jeweiligen Experiment verwendeten Teilkomponenten.

Das erste Experiment (I) steht ganz im Kontext der entwickelten Arena. Innerhalb des Experimentes sollen Roboter mit Hilfe virtueller Pheromone den kürzesten Weg zwischen einer virtuellen Ressource und ihrer Basisstation finden. In dem Experiment wird zur Steuerung der Roboter MDL2ϵ verwendet. Das gesamte Arenasystem zur Projektion von Position, Daten und Pheromonen kommt in diesem Experiment zur Anwendung.

Tabelle 6.1: Übersicht über die in den Experimenten verwendeten Teile der Hard- und Software.

Nr.	Experiment	MDL2ϵ	Simulation	Roboter	Arena
I	Stigmergie	ja	–	Jasmine/ Wanda	Kamera/ Projektor/ ZigBee
II.a	Sammeln	ja	Stage	Wanda	Projektor/ ZigBee
II.b	Selbst-Laden	ja	Stage	Wanda	–

Die Aufgabenstellung des zweiten Experiments (II) ist es, ebenfalls Ressourcen zu sammeln (II.a), wobei hier jedoch der Fokus auf die dafür benötigte Energie und deren Infrastruktur gesetzt wird (II.b).

Es wurden weitere Experimente mit dem gesamten System durchgeführt, die aber in dieser Arbeit nicht weiter betrachtet werden. In [47] wird ein Experiment mit dem Jasmine-Roboter beschrieben, welches sowohl die Arena als auch MDL2ϵ mit einschließt.

6.1.1 Stigmergie

Stigmergie ist ein von P.P. Grassé geschaffener Begriff [45], der sich aus den zwei Worten Stigma (Ansporn) und Ergon (Arbeit, Produkt der Arbeit) zusammensetzt. Er bezeichnet somit den *Ansporn aus dem Produkt der Arbeit.* Er beschreibt damit das Verhalten sozialer Insekten, bei denen die Veränderung der Umgebung während der Arbeit zu einer Stimulierung weitere Verhaltensweisen führt. Diese Stimmulierung kann durch bestimmte Konfigurationen eines Bauwerks oder aber auch durch die kummulierte Ablage von Pheromonen ausgelöst werden. Stigmergie beschreibt somit eine Art der Kommunikation, in der die Umgebung als Speichermedium dient.

In dem hier betrachteten Experiment beschränken wir uns auf die Veränderung der Umgebung durch Pheromone. Jean-Louis Deneubourg beschreibt in [44] das Verhalten von argentinischen Ameisen, die durch die Ablage von Pheromonen über die Zeit den kürzesten Weg zwischen Futter und Nest finden. Hierbei wird den Ameisen ein aus Brücken bestehendes Wegenetz präsentiert, das nur vier mögliche Wege zulässt.

Das Verhalten der Ameisen basiert dabei auf der Verstärkung der Phero-

monspur, die jede Ameise hinterlässt. Ameisen entscheiden sich mit höherer Wahrscheinlichkeit für einen Weg mit höherer Pheromonkonzentration. Auf kürzeren Wegen können schneller höhere Pheromonkonzentrationenen aufgebaut werden als auf langen, was zu einem selbstverstärkenden Effekt führt, bei dem sich der kürzeste Weg herausbildet. Garnier beschreibt in [41] ein Experiment, in dem er das von Deneubourg beschriebene Experiment auf eine Schwarm von Alice-Roboter übertrug.

Kazama et al. [54] beschreiben eine Versuch mit drei Robotern. In diesem legen die Roboter virtuelle, optische Pheromone ab, während sie zu ihrem *Nest* zurückkehren, wobei die Position des Nests nicht bekannt ist. Die Roboter, die auf der Suche nach Futter sind, versuchen den abgelegten Pheromonen zu folgen.

Um die Eigenschaften der Arena zu testen, wurde zeitgleich zu Garnier und Kazama, ein Versuchsaufbau gewählt, der von dem ursprünglichen Versuchsaufbau von Deneubourg und Garnier abweicht und dem von Kazama ähnelt. Der Versuch orientiert sich an dem Ameisenmodell aus NetLogo [116, 115]. Hier suchen die Ameisen in einer freien Arena nach Futterquellen. Haben sie diese gefunden, so kehren sie zu ihrem Nest zurück und legen dabei Pheromone ab, die andere Ameisen anziehen. Die Roboter haben sich zuvor die Postion des Nestes gemerkt.

Dieser Versuchsaufbau eignet sich besonders gut für die Evaluation der vorgestellten Arena, da alle wichtigen Teile getestet werden können. Während des Experimentes werden die Futterquellen und das Nest durch Datenfelder und die Pheromone, wie in Abschnitt 4.4 beschrieben, durch virtuelle Pheromone dargestellt. Damit die Roboter den Weg zurück zum Nest finden, verwenden sie das dort beschriebene Positionssystem.

Um die Effizienz des Pheromonansatzes zu bewerten, werden vier verschiedenen Versuche durchgeführt. Es wird eine zufallsbasierte Suche (*Random-Walk-Verhalten*) verwendet, bei der die Roboter jedoch das Positionssystem für die Rückkehr zum Nest verwenden. Beim zweiten Versuch dürfen die Roboter zusätzlich untereinander kommunizieren. Die Kommunikation beschränkt sich dabei darauf, dass die, die gerade Futter gefunden haben, dies den anderen Robotern mitteilen. Aus der Nachricht kann der Roboter die ungefähre Richtung zur Quelle ableiten. In dem dritten Versuch legen die Roboter auf dem Rückweg zum Nest virtuelle Pheromone ab. Der vierte Versuch kombiniert die letzten beiden. Hier werden Pheromone abgelegt und zusätzlich die Richtung zur Quelle von den Robotern übermittelt.

Experimente mit Jasmine

Es wurde jeweils ein Experiment mit zehn Jasmine-Robotern für jede der vier Strategien über eine Dauer von 40 Minuten durchgeführt. Während des Experiments wechselte nach 20 Minuten automatisch die Position der Futterquelle, um die dynamische Reaktion des Schwarms auf diese Störung zu untersuchen. Die Versuche wurden in einer nur halb so großen Arena ($160 \times 120 cm^2$) durchgeführt. Während dieser Versuche stand keine ODeM mit ZigBee zu Verfügung. Abbildung 6.2(a) zeigt den Versuchsaufbau mit dem Nest und den zwei Futterquellen A und B.

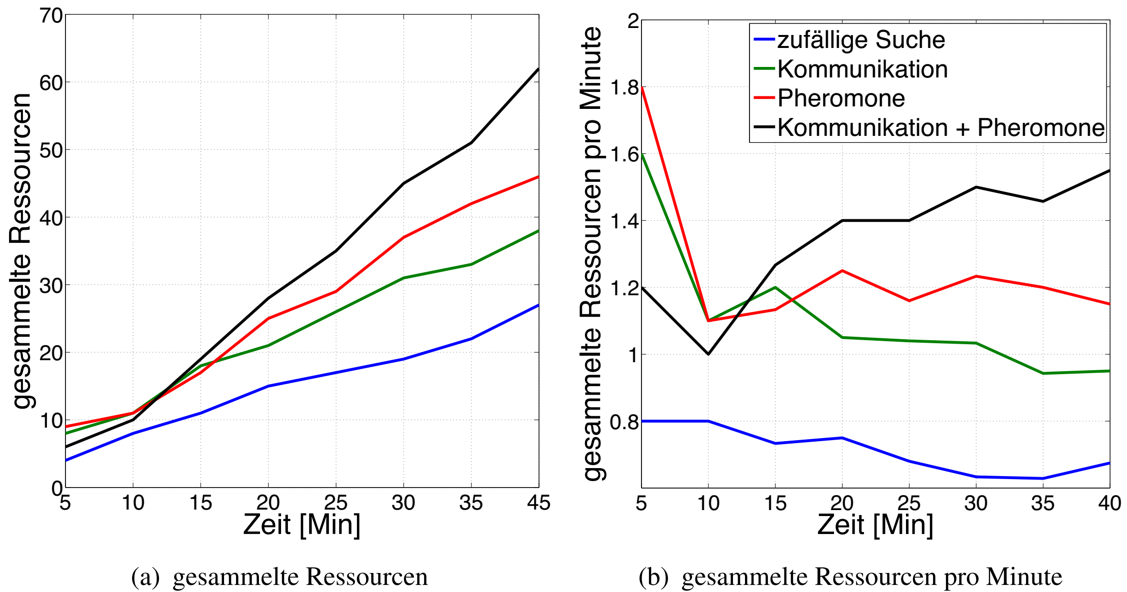
Abbildung 6.1 zeigt die Anzahl der im Verlauf des Versuchs gefundenen Ressourcen und die gefundenen Ressourcen pro Minute. Es ist eine klare Ordnung der Strategien zu erkennen. So ist die auf dem Zufall basierende Strategie die schlechteste, gefolgt von der auf Kommunikation basierenden, dann der Pheromon-Strategie und der Kombination von Pheromon und Roboter-Roboter Kommunikation. Gerade nach dem Wechsel der Quelle nach 20 Minuten können sich die auf Pheromonen basierenden Strategien schneller wieder fangen und einen hohen Durchsatz pro Minute erreichen.

Die Verwendung der Pheromone ermöglicht eine bessere Attraktion der Roboter zur Futterquelle. Dabei haben sich die Roboter in der rein pheromonbasierten Strategie für eine Richtung zu entscheiden. Mit einer Wahrscheinlichkeit von jeweils 50% wählen sie den Weg zur Quelle oder zum Nest. Die Wahrscheinlichkeit zur Wahl der richtigen Richtung wird durch die Hinzunahme der Kommunikation signifikant erhöht. Weitere Experimente mit verschiedenen Schwarmgrößen und mit Hindernissen zwischen der Futterquelle und dem Nest sind in [117] beschrieben.

Abbildungen 6.2(b) – (f) zeigen die projizierte Pheromonkarte in den verschiedenen Phasen des Experimentes. Man erkennt, wie eine Pheromonspur von Quelle A zum Nest aufgebaut wird. Nach dem Wechsel von Quelle A zu B zerfällt diese wieder, und es wird eine Pheromonspur zwischen der neuen Quelle B und dem Nest aufgebaut.

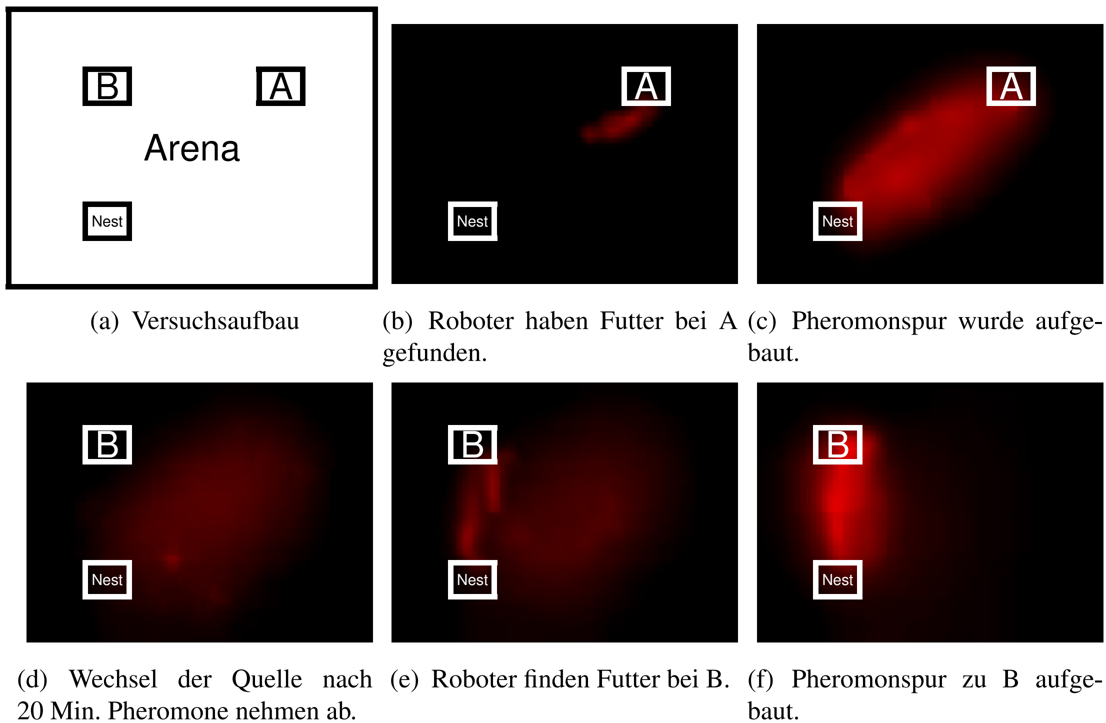
Experimente mit Wanda

Nach der Fertigstellung der Entwicklungsumgebung, der Arena und dem Wanda-Roboter wurden das Experiment als Testszenario mit zwanzig Wanda-Robotern in der doppelt so großen Arena ($160 \times 240 cm^2$) wiederholt. Im Gegensatz zum vorangegangenen Experiment mit Jasmine-Robotern, ist im Experiment mit Wanda-Robotern eine Quelle (A) immer aktiv. Die zweite



(a) gesammelte Ressourcen (b) gesammelte Ressourcen pro Minute

Abbildung 6.1: Während des Versuchs mit zehn Wanda-Robotern gesammelte Ressourcen und Ressource pro Minute.



(a) Versuchsaufbau (b) Roboter haben Futter bei A gefunden. (c) Pheromonspur wurde aufgebaut.

(d) Wechsel der Quelle nach 20 Min. Pheromone nehmen ab. (e) Roboter finden Futter bei B. (f) Pheromonspur zu B aufgebaut.

Abbildung 6.2: Pheromonkarte in den verschiedenen Phasen des Versuchs mit dem Jasmine-Roboter. Ein Wechsel von Quelle A zu B findet automatisch nach 20 Minuten statt.

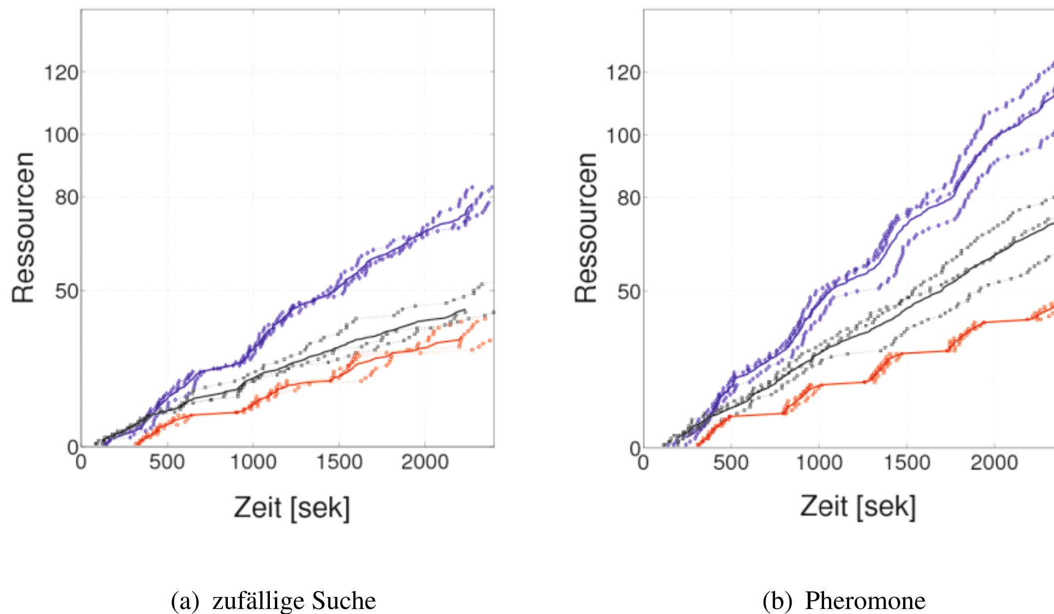
Quelle (B) wird nach fünf Minuten hinzugeschaltet. Sie stellt dann insgesamt zehn Ressourcen bereit. Wurden diese von Robotern eingesammelt, wird Quelle B wieder ausgeblendet und erscheint erst dann wieder nach einer Regenerationszeit von fünf Minuten. Die Geschwindigkeit, mit der der Schwarm Ressourcen von Quelle B einsammelt bestimmt, wie häufig sich die Quelle während des Versuches regeneriert. Dieser Versuchsaufbau soll verdeutlichen, wie die Arena dynamisch auf das Verhalten der Roboter reagieren kann. Zusätzlich wurde darauf geachtet, dass die Position des Nestes den Robotern nicht bekannt ist. Die Roboter speichern ihre Position erst, wenn sie im Nest "*geboren*" werden. Dies ermöglicht eine flexible Positionierung des Nestes.

Es wurden je drei Versuche mit dem Random-Walk-Verhalten (zufällige Suche) und mit dem pheromonbasierten Verhalten durchgeführt. Die verwendeten Pläne findet man in Anhang B.1. Abbildung 6.3 zeigt wie viele Ressourcen gefunden, bzw. beim Nest abgelegt wurden. Mit der pheromonbasierten Suche werden ca. 1,5 mal so viele Ressourcen gesammelt wie mit die Zufallsbasierten. Das pheromonbasierte Verhalten kann auch deutlich schneller die pulsierende Quelle B abernten. In allen drei Versuchen kann sich Quelle B fünf mal regenerieren. In den Versuchen mit dem zufallsbasierten Verhalten regeneriert sich diese nur vier mal. Auch lässt das pheromonbasierte Verhalten eine deutlich konstanteren Strom an Ressourcen von Quelle A zu als das zufallsbasierte Verhalten. Wie in Abbildung 6.3(a) und (b) zu sehen sind die Verläufe der drei Versuche jeweils sehr ähnlich. Hieraus und aus den Versuchen mit Jasmine lässt sich ableiten, dass eine Hinzunahme von Kommunikation die Sammelgeschwindigkeit erhöhen kann.

In Abbildung 6.4 wird der zeitliche Verlauf eines Versuchs mit pheromonbasiertem Verhalten gezeigt. Es ist gut zu erkennen, dass sich die Roboter zunächst verteilen und dann die Pheromone aufbauen. Dadurch, dass die Pheromone in der Nähe von Quelle B sich nicht vollständig auflösen, bleibt der Ort dieser pulsierenden Quelle im kollektiven Gedächtnis. Dies ermöglicht ein schnelles Ernten der Ressourcen bei erneutem Auftauchen der Quelle. Bei längerer Abwesenheit der Quelle würde sie langsam aus dem kollektiven Gedächtnis verschwinden.

Bewertung

Das Experiment zeigt, dass virtuelle ortsgebundene Pheromone auch für die Koordination von Roboterschärmen sinnvoll eingesetzt werden können. Es



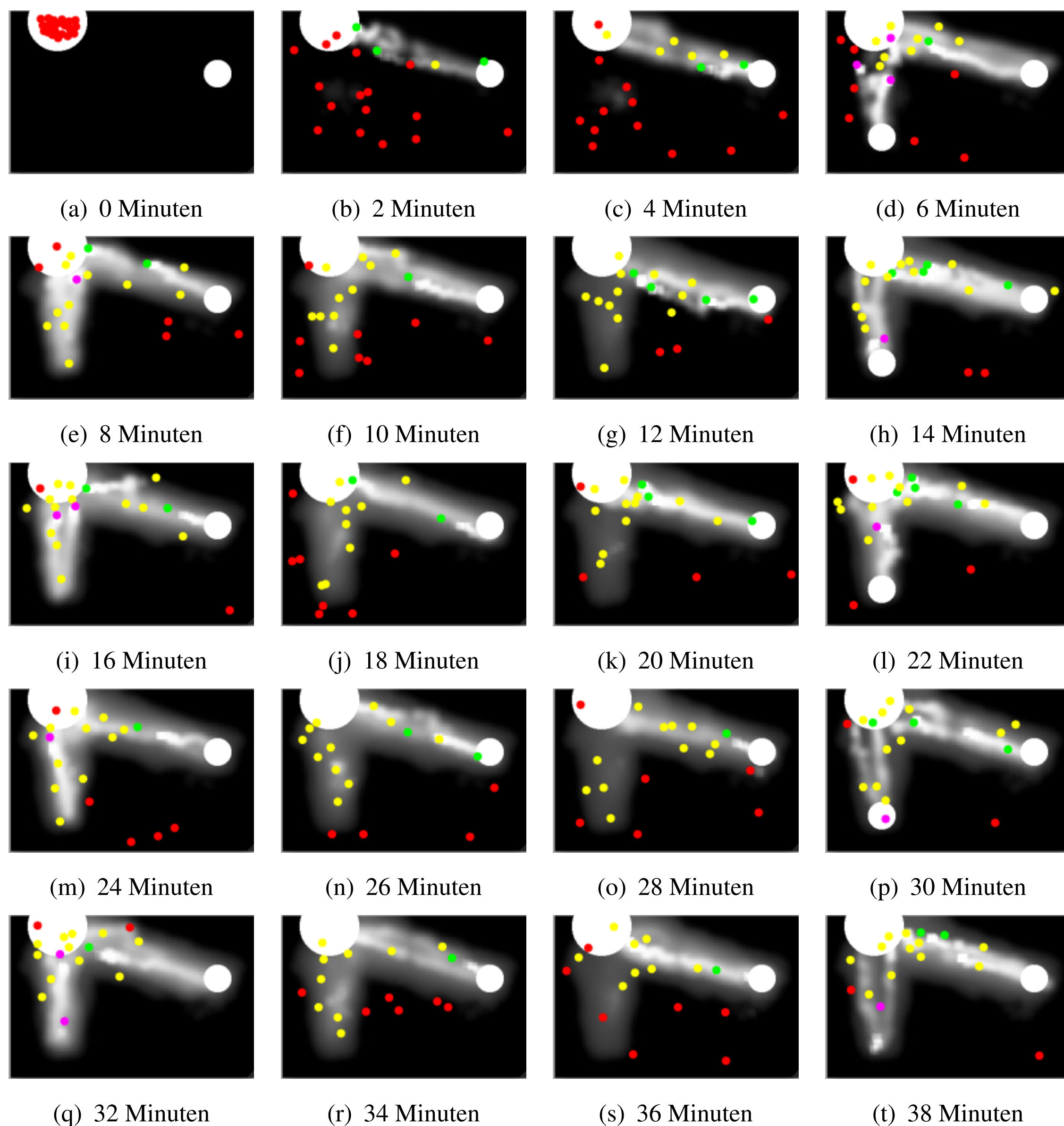
(a) zufällige Suche (b) Pheromone

Abbildung 6.3: Gesammelte und gefundenen Ressourcen für je drei Versuche mit dem Wanda-Roboter und deren Mittelwert aufgeteilt nach den gesammelte Ressourcen (blau), gefundene Ressourcen Quelle A (schwarz) und gefundene Ressourcen Quelle B (rot).

ist klar, dass die Umsetzung mit virtuellen projizierten Pheromonen keine Lösung für Roboterschwärme in unüberwachten, unstrukturierten Arbeitsbereichen ist. Es ist aber möglich mit Hilfe des vorgestellten Systems Untersuchungen durchzuführen, in denen die Eigenschaften von Substanzen modelliert werden können, für die schon Geruchssensoren existieren, um diese dann in ähnliche Robotersysteme zu übertragen.

Das Experiment zeigt auch, wie die Arena für dynamische interaktive Experimente eingesetzt werden kann. Durch die Experimente mit Jasmine wird deutlich, dass es im Zuge der Automatisierung von Experimenten notwendig ist, dass das ODeM mit einer drahtlosen Kommunikationsschnittstelle für das Sammeln von Daten ausgestattet wird.

Die konsequente Umsetzung dieser Schlussfolgerung sieht man im Experiment mit dem Wanda-Roboter. Hier steht die vollständige Arena mit ZigBee zur Datenerfassung zur Verfügung. Es konnten deutlich mehr Experimente durchgeführt werden und es können mehr Daten gesammelt werden, die sich aufgrund ihrer Vollständigkeit leichter analysieren lassen.



(a) 0 Minuten (b) 2 Minuten (c) 4 Minuten (d) 6 Minuten
(e) 8 Minuten (f) 10 Minuten (g) 12 Minuten (h) 14 Minuten
(i) 16 Minuten (j) 18 Minuten (k) 20 Minuten (l) 22 Minuten
(m) 24 Minuten (n) 26 Minuten (o) 28 Minuten (p) 30 Minuten
(q) 32 Minuten (r) 34 Minuten (s) 36 Minuten (t) 38 Minuten

Abbildung 6.4: Entwicklung der Pheromonkarte bei einem Versuch mit 20 Wanda-Robotern. Es werden von den Robotern nur die Pheromone verwendet, Kommunikation findet nicht statt. Das große weisse Feld ist das Nest (links oben). Die kleinen weissen Felder sind die Futterquellen (Quelle A (rechts oben), Quelle B (links unten in den Abbildungen (d), (h), (l) und (p)). Dabei ist Quelle A immer an. Quelle B hat zehn Ressourcen. Sind die zehn Ressourcen verbraucht, geht sie erst nach fünf Minuten wieder an. Die Farben der Roboter stellen deren Zustand dar: Random-Walk (rot), Folge-Pheromon (gelb), Rückkehr-Quelle-A (grün) und Rückkehr-Quelle-B (magenta). Die Darstellung wurde aus den Daten rekonstruiert, die während des Versuches aufgezeichnet worden sind.

6.1.2 Sammeln von Ressourcen unter energetischen Gesichtspunkten

Betrachtet man das Sammeln von Ressourcen, wie im letzten Experiment dargestellt, so interessiert nicht nur die Frage nach einer optimalen Arbeitsverteilung und einem ausgewogenen Verhältnis zwischen Exploration der Umwelt und Ausnutzung (engl. *exploitation*) der Ressourcen, sondern auch nach dem energetischen Nutzen. Das heißt: *Gibt es eine positive Energiebilanz für den Schwarm?* Eine positive Energiebilanz ist wichtig, um den Schwarm und seinen Mitglieder am "Leben" zu halten.

Die Frage muss betont werden, da sich in Zukunft der Nutzen von Schwarmrobotern an seiner Wirtschaftlichkeit beim Sammeln von Ressourcen in für den Menschen schwer zugänglichen Bereichen messen lassen muss, wie z.B. dem Abbau von Manganknollen in der Tiefsee. Hierbei kann die Energiebilanz auch direkt auf die Wirtschaftlichkeit angewendet werden. Der Roboterschwarm arbeitet nur dann wirtschaftlich, wenn es dem Schwarm möglich ist, deutlich mehr Ressourcen zu sammeln, als er benötigt, um sich, seinen Energieverbrauch und seine Wartung zu bezahlen.

Im letzen Fall gehen wir nicht davon aus, dass die Roboter die gesammelten Ressourcen selbst verwerten, bzw. in für sie nutzbare Energie umwandeln müssen. Dies vereinfacht die Betrachtungsweise und lässt die Fragen zu:

> *Wie viele Ladestationen werden benötigt, um einen Schwarm in einem optimalen Zustand zu halten?*
>
> *Wie groß muss der Schwarm gewählt sein, um ein ausgewogenes Verhältnis zwischen ruhenden (Roboter die sich aufladen) und arbeitenden Robotern zu erreichen?*

Dies ist beim Betrieb von Robotern in für den Menschen unzugänglichen Gebieten besonders wichtig, wie z.B. bei der Exploration von Planetenoberflächen.

Um diesen Fragen nachzugehen, wurden zwei Experimente durchgeführt, die zusammen eine Analyse der Fragen zulassen. Einige Forscher konnten im Vorfeld zeigen, dass ein Hauptpunkt, der die Effizienz eines Schwarms bestimmt, der Wettbewerb um freien Bewegungsraum ist. Anhand von Wahrscheinlichkeitsmodellen analysierte Lerman [64], welchen Effekt die Größe eines Schwarms auf dessen Performanz hat. Sie konnte zeigen, dass es eine optimale Anzahl von Robotern gibt, eine Sammel-

aufgabe durchzuführen, die zu einer optimalen Schwarm-Performanz führt. Daher wird im ersten Experiment für eine einfache Sammelstrategie die optimale Schwarmgröße bzw. Schwarmdichte für einen Schwarm aus Wanda-Robotern bestimmt.

Im zweiten Experiment wird das *Selbst-Lade-Verhalten* von einem Roboterschwarm untersucht. Ein wichtiger Punkt ist hierbei die Verteilung von Ruhe- zu Arbeitsphasen. Krieger und Billeter [62] verwenden dazu einen schwellwertbasierten Ansatz, der zuerst von Théraulaz et al. [109] zur Untersuchung der Arbeitsverteilung bei sozialen Insekten eingesetzt wurde, um die Verteilung zwischen Ruhe- und Arbeitsphasen ihrer Robotern auszugleichen. In ihrem Experiment hat jeder Roboter einen unterschiedlichen, zufällig gewählten Schwellwert, um die Aktivitäten im Schwarm zu regulieren. Labella et al. [63] führen einen einfachen adaptiven Mechanismus ein, um die Rate zwischen sammelnden und ruhenden Robotern und damit die gesamte Performanz des Systems zu verbessern. Sie belohnen erfolgreiches Futtersammeln und bestrafen Fehler, um die Wahrscheinlichkeit zum Verlassen des Nestes anzupassen. Jones und Matarić [51] beschreiben einen adaptiven Ansatz zur Arbeitsverteilung, der auf der Beobachtung des Verhaltens der Mitglieder des Schwarms ohne direkte Kommunikation basiert. Liu et al. [65] beschreiben ein adaptives Verfahren, das nur auf lokalen Informationen basiert, um den Energiehaushalt des Schwarms während eines Sammelszenarios zu optimieren. Sie setzen einen breitgefächerten Satz an Regeln bzw. Nachrichten für den einzelnen Roboter ein: interne Nachrichten (erfolgreiches Futtersammeln), Umgebungsnachrichten (Kollisionen mit anderen Robotern während des Sammelns) und pheromonartige soziale Nachrichten (der Erfolg eines anderen Roboters beim Futtersammeln). Hierbei gehen sie davon aus, dass die gesammelte Energie direkt in für den Schwarm nutzbare Energie umgesezt werden kann. Ruhephasen werden nur verwendet, um den Roboter in einen Modus zu versetzen, der weniger Energie verbraucht, und um den Durchsatz beim Sammeln von Futter zu optimieren. Dies entspricht aber nicht der Realität von Roboterschwärmen, da nur wenige Roboter die gesammelte Ressourcen, wie bei EcoBot I und II [49, 76] der Fall, in Energie umwandeln können. Selbst diese Roboter müssen ruhen um zu laden und nicht nur um Energie zu sparen. Sie erzeugen also beim Ruhen eine positive Energiebilanz und nicht beim Sammeln von Ressourcen.

Daher steht in dem hier durchgeführten Experiment die Optimierung zwischen echter Ruhe- und Arbeitszeit, bzw. Lade- und Entladephase, im Vordergrund. Anschließend werden die Ergebnisse beider Experimente theo-

retisch zusammengeführt, um eine optimale Anfangskonfiguration und Parametrisierung basierend auf der Lade- und Entladezeit zu berechnen, und um eine Aussage über die Performanz des Schwarms bei gleichzeitiger Ausführung beider Experimente zu machen.

Experiment: Sammeln von Ressourcen

Im Gegensatz zum Experiment aus Abschnitt 6.1.1 werden zum Finden der Ressourcen und des Ablagebereichs nicht die vollen Möglichkeiten der Arena ausgeschöpft (virtuelle Pheromone, Ego-Positioning). Vielmehr ist es ein Ziel, dass die Roboter nur über lokale Kommunikation ihre Arbeit koordinieren. Die Arena wird in diesem Experiment zur Darstellung der Ressource, und zum Sammeln von Informationen über den Verlauf der Experimente verwendet. Der Ablagebereich ist durch eine für die Bodensensoren des Wanda-Roboters wahrnehmbaren, schwarzen Linie markiert, die den Ablagebereich vom Arbeitsbereich trennt. Zu Beginn eines jeden Versuchs sind die Roboter zufällig über den Arbeitsbereich verteilt, und auch die Initialisierungen der ersten Ziele eines Roboters, Ressource oder Ablagebereich, sind zufällig. Während eines Versuchs werden die auftretenden Wechsel zwischen diesen zwei Zielen pro Roboter gezählt. Die Summe über alle Roboter ergibt eine Punktzahl (engl. *score*) für den Versuch.

Es wurden vier Experimente mit Versuchen á 20 Minuten durchgeführt. In zwei von vier Experimenten wurde in der Mitte der Arena zusätzlich eine Wand eingezogen, die die Arena auf einem Viertel der Breite abtrennt. Dabei bleiben zwei Öffnungen im Verhältnis eins zu zwei übrig. Eine Darstellung des Aufbaus ohne Wand sieht man in Abbildung 6.5.

Die Experimente mit und ohne Wand wurden anhand von zwei Strategien evaluiert. Die eine Strategie basiert, wie im Stigmergie-Experiment, auf einer *Random-Walk-Strategie (RWS)*. Das heißt, die Roboter können die Ressource und den Ablagebereich nur zufällig finden. Die andere basiert auf einer *Hop-Count-Strategie (HCS)*.

Bei der HCS wird die Entfernung zum Ziel anhand der Anzahl (engl. *count*) der Weiterleitungen einer Nachricht (engl. *hops*) dargestellt. Befinden sich die Roboter im Zielbereich, so senden sie die Nachricht mit Wert null über alle sechs Kommunikationskanäle. Befinden sie sich nicht im Zielbereich, senden sie das empfangene Minimum über alle eingehenden Nachrichten plus eins weiter, wobei ein vorgegebener Maximalwert –hier fünf– nicht überschritten werden darf. Dies erzeugt einen Gradienten in Richtung des Zielbereichs. Die HCS wird sowohl für das Auffinden der Ressourcen,

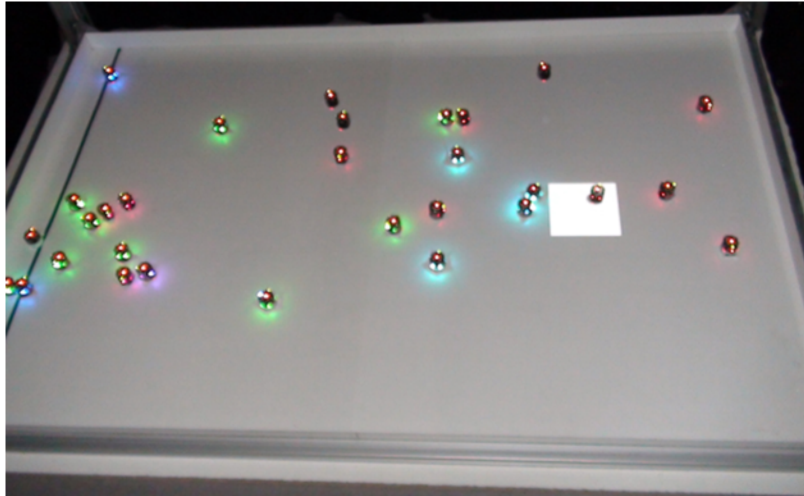
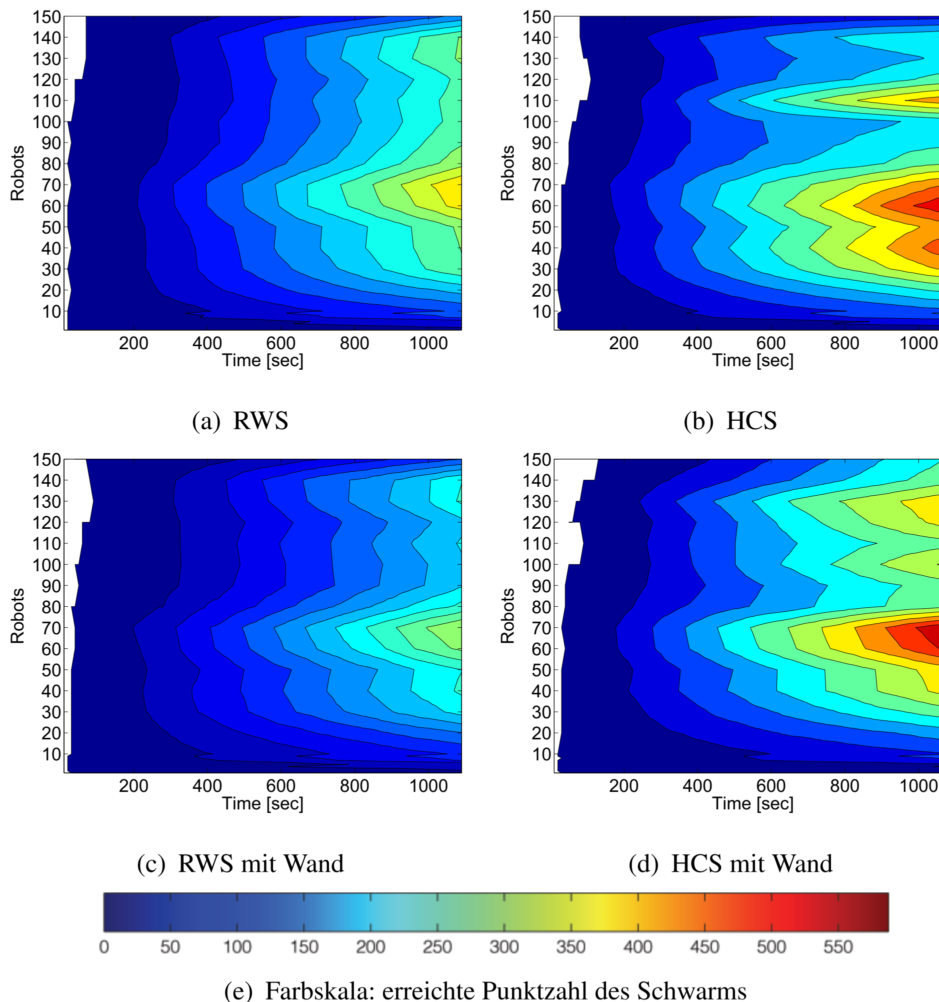


Abbildung 6.5: Aufbau der Arena ohne Wand: links am Rand ist der schwarze Streifen für den Ablagebereich, rechts sieht man die mit dem Projektor projizierte Ressource als helles Rechteck. Die verschiedenen Farben der Roboter stellen verschieden Zustände im Verhalten der Roboter dar [67].

als auch für das Auffinden des Ablagebereichs angewendet. Der in der Simulation und auf dem Roboter verwendet MDL2ϵ-Plan findet sich in Anhang B.2.

Es wurden in der Simulation jeweils 50 Versuche mit einer Schwarmgröße von eins bis neun Robotern und jeweils zehn Versuche mit Schwärmen zu 10, 20, ..., 150 Robotern durchgeführt. Mit den Wanda-Robotern wurden jeweils zehn Versuche mit 1, 5, 10, 20 und 30 Robotern durchgeführt. Alle Experimente in der Simulation und auf dem Roboter wurden mit und ohne Wand gemacht.

Abbildung 6.6 stellt die durchschnittlich erreichte Punktzahl des Schwarms für die vier Experimente in der Simulation dar. Man erkennt deutlich, dass in allen Fällen die HCS der einfacheren RWS überlegen ist. Betrachtet man die Leistung eines einzelnen Roboters im Schwarm in Abbildung 6.7, so sieht man eine deutliche Abnahme der Leistung pro Roboter mit zunehmender Schwarmdichte bei beiden Strategien, was bei der RWS deutlich stärker ausgeprägt ist. Dies wird, wie schon von Lerman [64] beschrieben, durch die zunehmende Anzahl von Kollisionen in einem dichteren Schwarm verursacht . Die Roboter benötigen immer mehr Zeit zum Ausweichen, die ihnen dann für das Finden und Transportieren von Ressourcen nicht mehr zur Verfügung steht. Bei der HCS erreicht man einen akzeptablen Wert bis zu einer Schwarmgröße von 40 Robotern. Bei einer



(a) RWS

(b) HCS

(c) RWS mit Wand

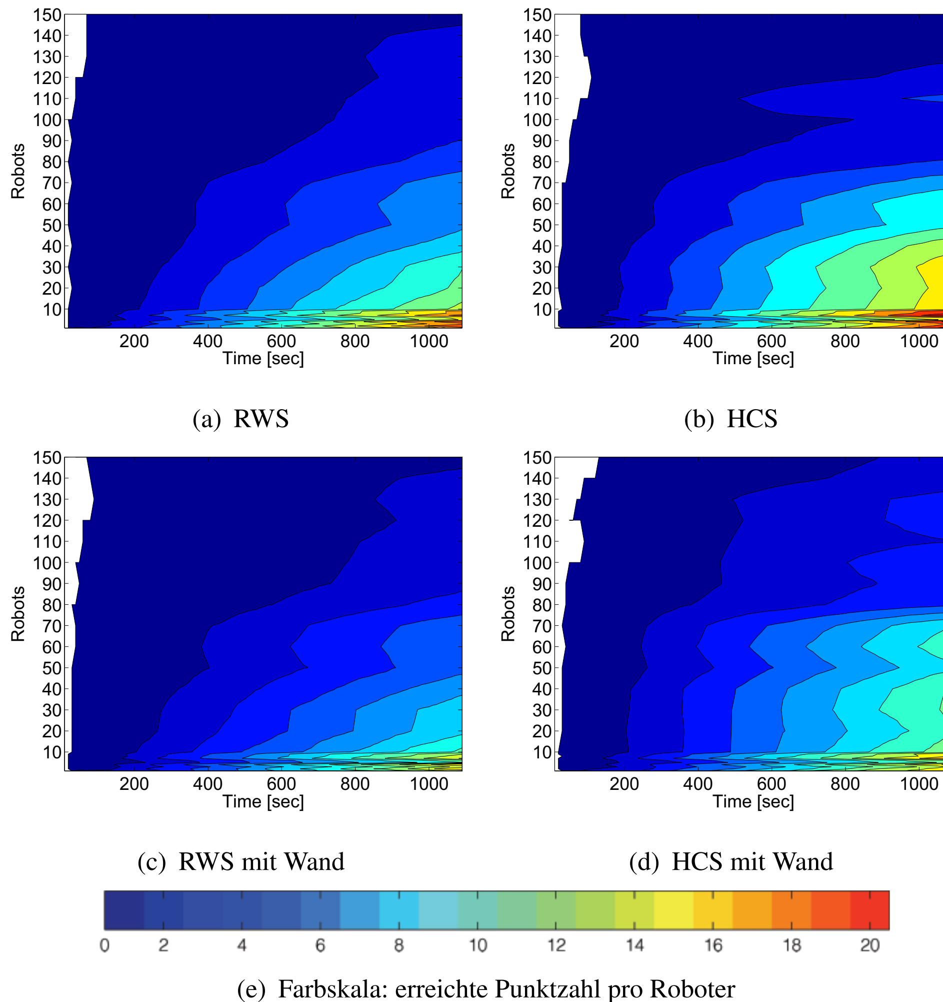
(d) HCS mit Wand

(e) Farbskala: erreichte Punktzahl des Schwarms

Abbildung 6.6: Durchschnittlich erreichte Punktzahl für den gesamten simulierten Schwarm beim Sammeln von Ressourcen.

Schwarmgröße von mehr als 70 Robotern ist die Dichte so groß, dass es zu einem deutlichen Einbruch in der Gesamtleistung des Schwarms kommt. Über 70 Robotern gibt es eine weitere Spitze bei 110 Robotern. Betrachtet man hier aber zusätzlich die Standardabweichung, so steigt diese ab 70 Robotern sprunghaft an. Versuche mit mehr als 70 Robotern haben einen stark unterschiedlichen Ausgang. Dies liegt daran, dass sich in manchen Versuchen große Cluster vor der Ablagestelle bilden, die ein Abfließen von Robotern, die schon ihre Ressource abgelegt haben, verhindern. Dies verringert den Durchsatz.

Ein nahezu gleiches Verhalten beobachtet man bei den Versuchen mit dem Wanda-Roboter, vgl. Abbildungen 6.8 und 6.9. Diese unterscheiden sich nur um einen Faktor von den Ergebnissen aus der Simulation. Der Faktor ergibt sich durch Unterschiede zwischen den echten Robotern und dem



(a) RWS

(b) HCS

(c) RWS mit Wand

(d) HCS mit Wand

(e) Farbskala: erreichte Punktzahl pro Roboter

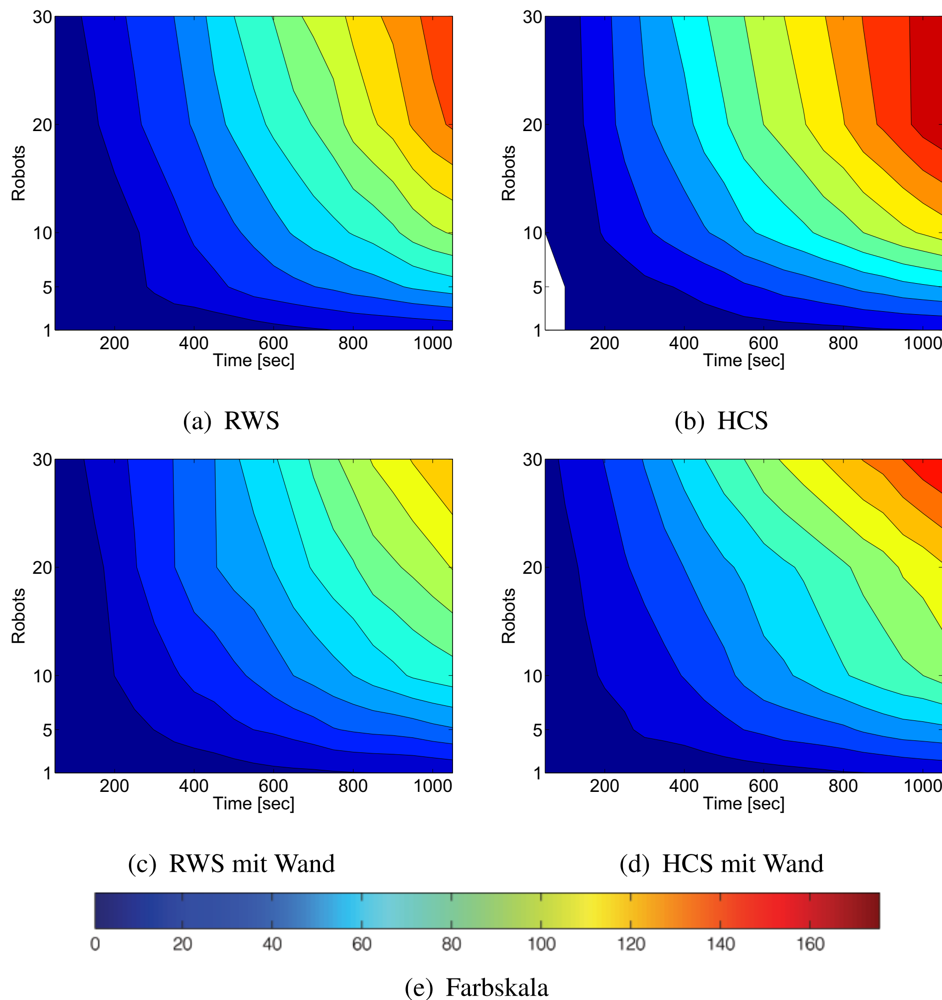
Abbildung 6.7: Durchschnittlich erreichte Punktzahl pro simulierten Roboter beim Sammeln von Ressourcen.

Modell des Roboters in der Simulation. Daher lässt die Simulation Schlüsse für das Verhalten von Schwärmen mit bis zu 150 Robotern zu.

Experiment: Selbst-Lade-Verhalten

Im Experiment zum *Selbst-Lade-Verhalten* wird untersucht, in wieweit sich ein Schwarm mit einer gegebenen Anzahl an Ladestationen am Leben erhalten kann. Das heisst wie groß kann ein Schwarm sein, dem m-Ladeplätze zur Verfügung stehen, und wie sieht die Verteilung von arbeitenden zu ruhenden/ladenden Robotern aus.

Hierzu wurden Eigenschaften wie durchschnittlicher Energieverbrauch und Ladezeit des Wanda-Roboters vermessen und in die Simulation übertragen. Ein Modell des Ladestands über die aktuell anliegenden Spannungen



(a) RWS

(b) HCS

(c) RWS mit Wand

(d) HCS mit Wand

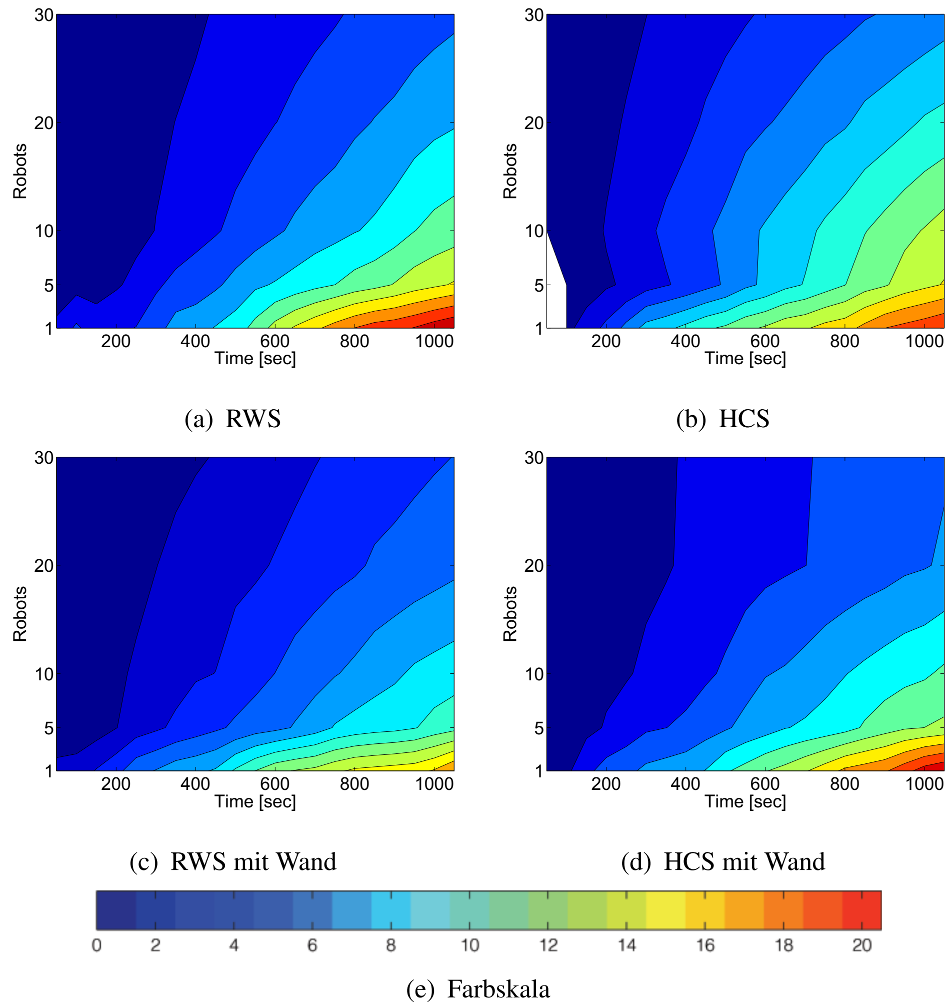
(e) Farbskala

Abbildung 6.8: Durchschnittliche erreichte Punktzahl für den gesamten realen Schwarm beim Sammeln von Ressourcen.

der Akkus sichert dieselbe Sicht des simulierten und echten Roboters auf den aktuellen Ladestand. Die für den Roboter ermittelten Werte für die Entladezeit t_e beträgt $t_e = 2{,}5h$ und für die Ladezeit $t_l = 2h$.

Für den Versuch wurde eine Ladestation entwickelt, siehe Abbildung 6.10, die das gleichzeitige Laden von bis zu 2×8 Robotern unterstützt. Schwarze Streifen auf dem Arenaboden leiten die Roboter zu der Ladestation. Der grüne Bereiche um die kupfernen Ladekontaktstreifen verhindert das Eindringen von nicht ladenden Robotern.

Während des Experimentes wechselt der Roboter zwischen drei Verhalten: dem *Arbeits*-, *Lade*- und *Social-Starving-Verhalten*. Das *Arbeitsverhalten* ist ein einfaches Random-Walk-Verhalten. Es simuliert den Energieverbrauch einer Tätigkeit wie dem oben beschriebenen Sammeln von Ressourcen. Das *Ladeverhalten* beinhaltet das Suchen und Folgen der Leitlinie zur



(a) RWS

(b) HCS

(c) RWS mit Wand

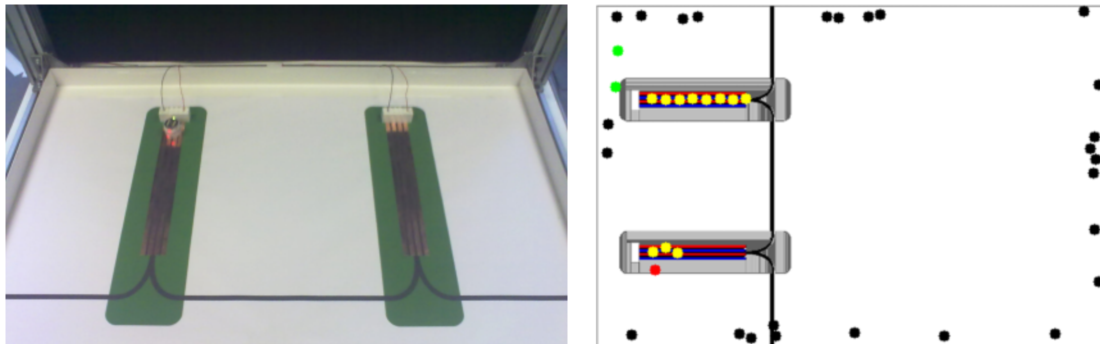
(d) HCS mit Wand

(e) Farbskala

Abbildung 6.9: Durchschnittliche erreichte Punktzahl pro realem Roboter beim Sammeln von Ressourcen.

Ladestation, dem Nachrücken in der Ladeschlange und dem Herausfahren aus der Ladestation. Das *Social-Starving-Verhalten* soll ein “*verhungern*” (engl. *starving*) der Roboter auf der Leitlinie verhindern. Ist dieses Verhalten aktiv, so fahren die Roboter geradeaus und bleiben an Hindernissen stehen, bis sie dort verhungert sind, vgl. Abbildung 6.10(b). Dies verhindert eine Blockade der Leitlinie, und damit der Ladestation, durch “*tote*” Roboter. Die drei Verhalten werden durch verschiedene Schwellwerte des Ladestandes ausgelöst. Der Schwellwert für den Übergang zum Social-Starving-Verhalten, θ_s, liegt bei 2,5% des maximalen Ladestandes ($\theta_s = 0.025$). Die anderen zwei Schwellwerte sind ein Teil des Experiments und werden im Folgenden beschrieben.

Es wurden Experimente durchgeführt, die drei Parameter untersuchen, die einen Einfluss auf die Anzahl der am Ende “*lebendigen*” Roboter haben.



(a) Ladestation mit Leitlinie und Kupferstreifen als Ladekontakte (b) Ladeverhalten mit Sozial-Starving. grün: arbeiten, gelb: laden, rot: suchen, schwarz: "*tot*"

Abbildung 6.10: Echte und simulierte Ladestation [67].

Nr.	Experiment	Schwellwert	20	40	60	80	100	∅
I.	100% Ladestand, Laden bis 100%	25%	15,0	11,0	12,5	9,0	12,0	11,9
		50%	8,5	13,5	10,5	14,0	10,5	11,4
		75%	19,0	20,0	22,5	21,0	14,5	19,4
II.	Zufälliger Ladestand, Laden bis 100%	25%	14,0	15,0	11,0	10,5	14,0	12,9
		50%	20,0	20,5	21,0	26,0	21,5	21,8
		75%	19,5	22,5	18,0	20,5	21,0	20,3
III.	100% Ladestand, max. 10 Min, Ladezeit	25%	18,5	25,5	18,5	22,5	11,0	19,2
		50%	19,5	21,5	20,0	20,5	15,5	19,3
		75%	20,0	21,0	16,0	17,0	9,0	16,6
IV.	Zufälliger Ladestand, max. 10 Min, Ladezeit	25%	19,5	23,0	23,5	17,5	21,0	20,9
		50%	20,0	25,5	25,0	25,0	25,0	24,1
		75%	19,0	16,5	20,0	21,0	13,5	18,0

Tabelle 6.2: Durchschnittliche Anzahl an Robotern, die am Ende des Versuchs noch "*lebten*".

Diese sind: der Schwellwert θ_l zum Umschalten von Arbeits- auf Ladeverhalten (25%, 50%, 75% Ladestand), der initiale Ladestand E_0 (100% oder zufälliger Ladestand zwischen 50% - 100%) und die maximale Ladedauer $\hat{t}_l$ (maximal 10 Minuten oder bis 100% Ladestand erreicht). Hierzu wurden in der Simulation zu jeder möglichen Kombination der oben genannten Einflussfaktoren je zwei Versuche mit 20, 40, 60, 80 und 100 Robotern durchgeführt. Jedes der Experimente hatte eine Laufzeit von zehn simulierten Stunden. Damit war sichergestellt, dass die Roboter mehrere Ladezyklen durchlaufen. Die Ergebnisse finden sich in Tabellen 6.2 und 6.3.

Die Ergebnisse zeigen, dass alle drei Faktoren die Überlebenswahrscheinlichkeit des Schwarms bestimmen. Zum einen ist es wichtig, dass die Robo-

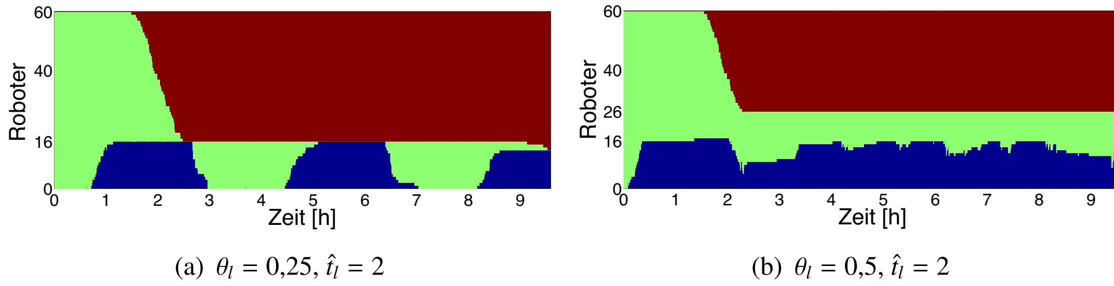
Nr.	Experiment	Schwellwert	20	40	60	80	100	∅
I.	100% Ladestand, Laden bis 100%	25%	6,4	5,1	5,5	5,1	5,7	5,56
		50%	7,1	8,5	7,3	8,3	7,3	7,72
		75%	10,4	11,4	12,4	12,1	9,1	11,08
II.	zufälliger Ladestand, Laden bis 100%	25%	7,2	8,2	7,4	6,6	7,6	7,4
		50%	11,4	11,5	12,7	13,5	12,4	12,3
		75%	11,9	13,4	11,0	12,7	12,6	12,32
III.	100% Ladestand, max. 10 Min, Ladezeit	25%	6,8	9,6	7,1	8,8	5,2	7,5
		50%	10,0	10,5	9,7	10,0	7,3	9,5
		75%	10,6	11,3	9,5	10,7	7,1	9,84
VI.	zufälliger Ladestand, max. 10 Min, Ladezeit	25%	7,5	10,0	10,4	7,9	10,7	9,3
		50%	10,9	11,9	13,0	12,4	12,6	12,16
		75%	10,6	10,9	12,2	12,0	9,7	11,08

Tabelle 6.3: Durchschnittliche Auslastung der Ladestation.

ter nicht zu früh und nicht zu spät laden gehen. In drei der vier Experimente erreicht man eine Maximum in der Überlebensrate ab einem Schwellwert von 50%. In Experiment I. gilt dies jedoch nicht. Dies liegt daran, dass die Roboter bei gleichmäßig vollem Ladestand zur gleichen Zeit die Ladestation aufsuchen. Diese kann dann maximal 16 Roboter aufnehmen. Bei einem Schwellwert von unter 75% sterben viele Roboter so schon vor Beginn des 1. Ladezyklus aus. Bei zufälligem Ladestand kann schon zu Beginn die Ladestation besser ausgenutzt werden, da manche Roboter einfach früher laden gehen. Dieses Verhalten erkennt man deutlich in Abbildung 6.11, in der der zeitliche Verlauf von Experiment II. mit 60 Robotern und Schwellwerten von 25% und 50% dargestellt ist. Beschränkt man zusätzlich die Ladezeit auf maximal 10 Minuten, erreichen wir eine besonders hohe durchschnittliche Auslastung der Ladestation, was direkt zu einem größeren Anteil an überlebenden Robotern führt. Anhand der Tabelle lässt sich die beste Parametrisierung des Versuchs für den Wanda-Roboter ablesen:

- zufällige Initialisierung des Ladestands,
- maximal 10 Minuten Laden und
- Wechsel von Arbeits- zu Ladeverhalten bei einem Schwellwert von 50%.

Insgesamt wird mit der sich aus den Experimenten ergebenden besten Parametrisierung ein maximaler Wert von durchschnittlich 25,5 überlebenden Robotern erreicht.



(a) $\theta_l = 0{,}25$, $\hat{t}_l = 2$ (b) $\theta_l = 0{,}5$, $\hat{t}_l = 2$

Abbildung 6.11: Zeitlicher Verlauf von ladenden (blau), arbeitenden (grün) und toten Robotern (rot) bei Versuchen mit 60 Robotern, zufällig initialisiertem Ladestand und verschiedenen Schwellwerten.

Energetische Betrachtung

In der folgenden energetischen Betrachtung werden die beiden Experimente auf theoretischer Basis zusammengeführt. Dazu werden die Ergebnissen von Selbst-Lade-Experiment in Gleichungen und Modelle umgeschrieben, die eine Erklärung und Optimierung des Experimentes zulassen.

Optimale Wahl der Anfangsparameter für eine große Überlebenswahrscheinlichkeit Betrachtet man das Selbst-Lade-Experiment, so kann man die gesamt Energie E des Schwarms darstellen als:

$$E(t) = \frac{E_l}{t_l} \sum_{i=1}^{t/\Delta t} l(i\Delta t) - \frac{E_l}{t_e} \left(\sum_{i=1}^{t/\Delta t} r(i\Delta t) - \sum_{i=1}^{t/\Delta t} l(i\Delta t) \right) + E_0 \qquad (6.1)$$

Hierbei ist $r(t)$, mit $r(0) = n$, die Anzahl der funktionierenden Roboter im Schwarm zum Zeitpunkt t, $l(t) \leq m$ ist die Anzahl der Roboter, die zum Zeitpunkt t laden, mit maximal m Ladestationen, E_l ist die maximale Ladung eines Roboters und $E_0 = \sum_{j=1}^{n} E_j^0$ die Summe der Anfangsladungen.

Es ist trivial, dass $E(t) > 0$ zu jedem Zeitpunkt t gelten muss, damit der Schwarm überlebt. Hieraus lässt sich ableiten, wie viele Ladestationen benötigt werden, um einen Schwarm der Größe n überleben zu lassen. Hierfür nehmen wir an, es gilt für zu jedem Zeitpunkt t: $r(t) = n$ (alle Roboter überleben) und $l(t) = m$ (die Ladestation ist immer voll ausgelastet). Mit $\Delta t = 1$ und $E_l = 1$ ergibt sich Gleichung 6.1 zu:

$$\begin{aligned} 0 &< \frac{1}{t_l}\sum_{i=1}^{t} m - \frac{1}{t_e}\left(\sum_{i=1}^{t} n - \sum_{i=1}^{t} m\right) + E_0 \\ \Leftrightarrow m &> \frac{t_l}{t_l + t_e} n - \frac{E_0 t_l t_e}{(t_l + t_e)t} \qquad (6.2) \\ \Rightarrow m &> \frac{t_l}{t_l + t_e} n \qquad (6.3) \end{aligned}$$

Für eine Abschätzung von m kann man den Term $\frac{E_0 t_l t_e}{(t_l+t_e)t}$ vernachlässigen, da $\lim_{t\to\infty} \frac{E_0 t_l t_e}{(t_l+t_e)t} = 0$. Man erhält eine Untergrenze für die Anzahl der Ladestationen. Dies gilt aber nur unter der Annahme, dass die Ladestation immer gut ausgelastet ist ($l(t) = m$). Die Simulationen zeigen, dass die Wahl der Parameter θ_l, E_0 und $\hat{t}_l \leq t_l$ einen signifikanten Einfluss auf die Auslastung der Ladestation hat. Denn obwohl die Anzahl der Ladestationen für 20 Roboter nach Gleichung (6.3) mit $m = 16$ überproportioniert ist ($m > 8{,}\bar{8}$) können in Experiment I. (25%/50%) und II. (25%) nicht alle Roboter geladen werden. Dies liegt daran, dass die Parameter so gewählt sind, dass die Roboter zu spät laden, und somit noch alle Ladestationen belegt sind, wenn die letzten vier noch nicht geladenen Roboter sterben. Dies lässt sich durch eine optimale Wahl von $\hat{t}_l$ und θ_l verhindern.

Bei der Berechnung von θ_l ist die Startphase des Schwarms ausschlaggebend. Die Parameter müssen so gewählt werden, dass auch der letzte Roboter überlebt, bis er laden kann. Hierfür betrachten wir in Gleichung (6.5) die Zeit, die abläuft, bis die ersten $n - 1$ Roboter geladen sind. Diese muss kleiner sein als die Zeit, die der letzte Roboter bis zum Start des Social-Starving-Verhaltens hat. Zur Vereinfachung gehen wir von der mittleren Energie $E_m = \frac{E_0}{n}$ als Anfangsenergie für einen Roboter aus.

$$((E_m - \theta_l)\, t_e + \hat{t}_l) + \left\lfloor \frac{n-1-m}{m} \right\rfloor \hat{t}_l < (E_m - \theta_s)\, t_e \qquad (6.4)$$

Durch Umformung nach θ_l und $\hat{t}_l$ erhalten wir:

$$\theta_l \geq \theta_s + \left\lfloor \frac{n-1}{m} \right\rfloor \frac{\hat{t}_l}{t_e} \quad (6.5)$$

$$\hat{t}_l \leq \left\lceil \frac{m}{n-1} \right\rceil (\theta_l - \theta_s) t_e \quad (6.6)$$

Optimal für unsere Roboter sind Werte von $\hat{t}_l$ und θ_l, die eine möglichst lange Laufzeit der Roboter garantieren, also $\hat{t}_l$ möglichst gross und θ_l möglichst klein. Beide hängen jedoch linear voneinander ab. Wächst $\hat{t}_l$ so wächst auch θ_l. Es ist aber möglich eine obere Schranke anzugeben. Diese ist die Summe der beim Laden zugewonnenen Energie mit dem Ladeschwellwert:

$$\frac{\hat{t}_l}{t_l} + \theta_l = e, \text{ mit } e \approx 1. \quad (6.7)$$

Diese Schranke erlaubt eine maximale Ausbeute der zugewonnenen Energie bei maximaler Stabilität des Schwarms. Ist e größer eins, so kann die maximale Ladezeit nicht ausgeschöpft werden, da schon vorher die maximale Ladung des Akkus erreicht wird. Ist sie kleiner, so sinken $\hat{t}_l$ und damit auch θ_l. Setzt man Gleichung (6.7) in Gleichung (6.5) ein, so erhält man einen optimalen Wert $\hat{t}_l^*$ bei möglichst kleinem θ_l.

$$\hat{t}_l^* = \frac{(e - \theta_s) t_e t_l}{\left(t_e + \left\lfloor \frac{n-1}{m} \right\rfloor t_l\right)} \quad (6.8)$$

Zur Überprüfung der Formeln wurde das in Algorithmus 6.1.1 aufgeführte diskrete, stochastische Modell aufgestellt. Dabei ist $\mathbf{S}_t = (S_t^0, \ldots, S_t^n)$ ein Vektor von Zeitzählern, E_t^r die Energie des Roboters r, $\mathcal{D}_t$ die Menge aller toten Roboter, $\mathcal{W}_t$ die Menge aller arbeitenden Roboter, C_t die Menge aller Roboter, die laden wollen, $\mathcal{L}_t$ die Menge aller ladenden Roboter und $\mathcal{N}_t$ die zufällige Menge aller Roboter, die anfangen zu laden. Die *Blockzeit* t_b wurde in das Modell eingeführt, um die Zeit zu modellieren, in der ein Roboter zwar auf der Ladestation steht aber noch nicht lädt, diese also blockiert. Dies geschieht z.B. beim Herauf- und Herunterfahren der Roboter.

Algorithmus 6.1.1: RECHARGEMODEL(n,m,t_e,t_l,$\hat{t}_l$,θ_s,θ_l,T,Δt,t_b,E_{min})

$$\textbf{init}\begin{cases}\mathbf{S}_0 = \hat{t}_l \cdot \mathbf{1} + \Delta t \cdot \mathbf{1}, \mathbf{S} \in \mathbb{R}^n \\ \mathbf{E}_0 = \textbf{random}(\mathcal{U}([E_{min},1])), \mathbf{E} \in \mathbb{R}^n\end{cases}$$

for $t \leftarrow 0$ **to** $\frac{T}{\Delta t}$

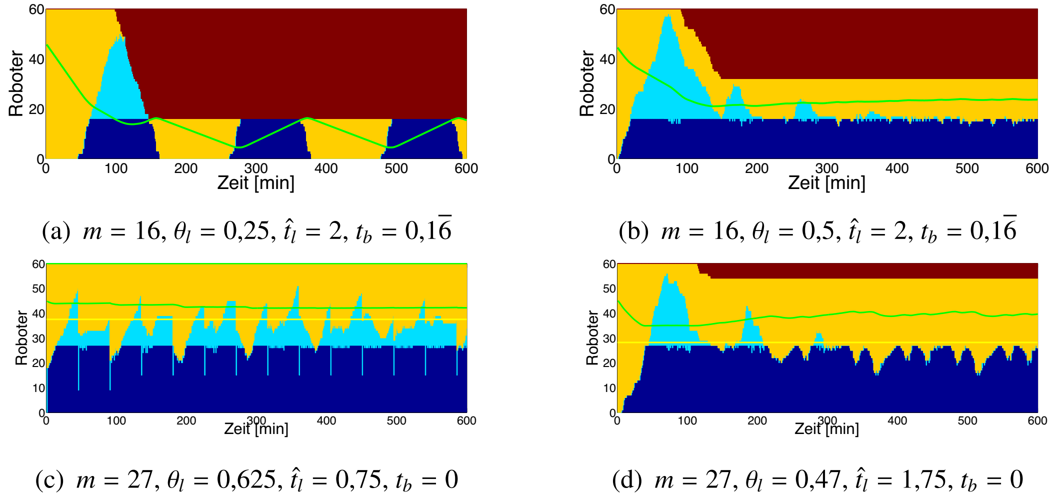
$$\textbf{do}\begin{cases}\mathbf{S}_t \leftarrow \mathbf{S}_{t-1} + \mathbf{1}\Delta t \\ \mathcal{D}_t \leftarrow \{r \in \mathcal{R} \mid E^r_{t-1} < \theta_s\} \\ \mathcal{W}_t \leftarrow \{r \in \mathcal{R}/\mathcal{D}_t \mid S^r_t \geq \hat{t}_l\} \\ \mathcal{C}_t \leftarrow \{r \in \mathcal{W}_t \mid (E^i_{t-1} < \theta_l)\} \\ \mathcal{L}_t \leftarrow \{r \in \mathcal{R}/\mathcal{D}_t \mid (E^r_{t-1} < 1) \ \wedge \ (S^r_t < \hat{t}^*_l)\} \\ \mathcal{N}_t \leftarrow \emptyset \\ \textbf{while } |\mathcal{N}_t| < m - |\mathcal{L}_t| \textbf{ or } |\mathcal{C}_t| = 0 \\ \quad \textbf{do}\begin{cases}\mathcal{N}_t \leftarrow \mathcal{N}_t \cup \textbf{random}\,(\mathcal{U}(\mathcal{C}_t)) \\ \mathcal{C}_t \leftarrow \mathcal{C}_t/\mathcal{N}_t\end{cases} \\ S^r_t \leftarrow -t_b\Delta t, \forall r \in \mathcal{N}_t \\ E^r_t \leftarrow E^r_{t-1} - (t_e)^{-1}\Delta t, \ \forall r \in \mathcal{W}_t \\ E^r_t \leftarrow E^r_{t-1} + (t_l)^{-1}\Delta t, \ \forall r \in \mathcal{L}_t \cup \mathcal{N}_t\end{cases}$$

return $(\mathcal{D}_t, \mathcal{W}_t, \mathcal{C}_t, \mathcal{L}_t)$

Abbildungen 6.12(a) und 6.12(b) zeigen die Ausgabe des Modells bei gleicher Parametrisierung wie bei den Versuchen aus Abbildung 6.11. Der Verlauf des Modells gleicht diesem Versuch. In Abbildung 6.12(c) wird der Verlauf des Modells beim selben Versuch mit optimierten Parametern gezeigt. Durch die Optimierung der Startparameter müssen keine Roboter sterben.

Durch die Formeln (6.3), (6.5) und (6.8) können Werte für m, θ_l und $\hat{t}^*_l$ in einem schwellwertbasierten System allein anhand der Kenntnisse über die durchschnittlichen Lade- und Entladezeit t_l, t_e festgelegt werden, die die Überlebenswahrscheinlichkeit des Schwarms optimiert.

Optimale Wahl der Anfangsparameter für eine maximale Gesamtarbeitszeit Betrachtet man Abbildung 6.12(c), so erkennt man, dass zwar keine Roboter sterben, aber eine große Anzahl von Robotern nicht arbeitet, sondern vor der Ladestation wartet (hellblau). In Abbildung 6.12(d) hingegen sterben ein paar Roboter, dafür können deutlich mehr Roboter arbeiten,



(a) $m = 16, \theta_l = 0{,}25, \hat{t}_l = 2, t_b = 0{,}1\overline{6}$

(b) $m = 16, \theta_l = 0{,}5, \hat{t}_l = 2, t_b = 0{,}1\overline{6}$

(c) $m = 27, \theta_l = 0{,}625, \hat{t}_l = 0{,}75, t_b = 0$

(d) $m = 27, \theta_l = 0{,}47, \hat{t}_l = 1{,}75, t_b = 0$

Abbildung 6.12: Zeitlicher Verlauf von $\mathcal{L}$ (blau), $\mathcal{W}$ (gelb), $\mathcal{D}$ (rot) und C (hellblau), bei Versuchen mit 60 Robotern, zufällig initialisiertem Ladestand und verschiedenen Startparametern. Die grüne Kurve beschreibt die gesamte Energie im Schwarm und die gelbe Linie stellt den Schwellwert $\theta_l \cdot n$ dar.

und haben aufgrund des niedrigeren Schwellwerts θ_l sogar mehr Energie für die Verrichtung ihrer Arbeit zur Verfügung. Energetisch gesehen arbeitet der Schwarm in Abbildung 6.12(d) besser, da hier die Gesamtarbeitszeit größer ist. Diese ergibt sich aus einer möglichst langen Arbeitszeit des Individuums, sowie einer möglichst großen Anzahl an lebenden Individuen im Schwarm. Eine Optimierung bezüglich der maximalen Gesamtarbeitszeit ist somit sinnvoll.

Gleichung (6.9) beschreibt die maximale Energie e_a, die ein Roboter beim Laden hinzubekommen kann.

$$e_a(\theta_l, \hat{t}_l) = \min\left(1 - \theta_l, \frac{\hat{t}_l}{t_l}\right) \tag{6.9}$$

e_a entspricht somit auch der maximalen Energie, die ein Roboter verbrauchen kann, bis er wieder laden muss. Das heisst

$$T_a(\theta_l, \hat{t}_l) = \mathcal{W}(\theta_l, \hat{t}_l) \cdot \alpha e_a(\theta_l, \hat{t}_l) \cdot t_e, \text{ mit } \alpha < 1, \tag{6.10}$$

ist die maximale Arbeitszeit des Schwarms. αe_a stellt mit $\alpha = 0{,}5$ die mittlere Energie eines Roboters dar. T_a gilt es zu maximieren, um Parameter für einen Schwarm mit vielen lebenden Individuen mit möglichst langer Arbeitszeit zu erhalten.

Hierzu berechnen wir die erwartete Anzahl an arbeitenden Robotern $\widehat{\mathcal{W}}(\theta_l,\hat{t}_l)$. Für $\widehat{\mathcal{W}}(\theta_l,\hat{t}_l)$ gibt es eine obere (l_{max}) und untere Grenze (l_{min}).

$$l_{max} = n \cdot \left(1 - \frac{t_e}{t_l + t_e}\right) \approx n - m \tag{6.11}$$

$$l_{min} = m \cdot \left(\frac{t_e}{t_l + t_e}\right) \approx (n - m)\frac{m}{n} \tag{6.12}$$

Die obere Grenze l_{max} beschreibt die maximale Anzahl an arbeitenden Robotern. l_{min} berechnet die durchschnittliche Anzahl an arbeitenden Robotern für den Fall, dass nur noch m Roboter leben.

Eine Approximation von $\widehat{\mathcal{W}}(\theta_l,\hat{t}_l)$ lässt sich aus drei Teilen zusammensetzen.

$$\widehat{\mathcal{W}}(\theta_l,\hat{t}_l) = \max\left(\min\left(\widehat{\mathcal{W}}_A(\theta_l, \hat{t}_l), \widehat{\mathcal{W}}_B(\theta_l,\hat{t}_l)\right), l_{min}\right) \tag{6.13}$$

Dabei sind $\widehat{\mathcal{W}}_A(\theta_l,\hat{t}_l)$ und $\widehat{\mathcal{W}}_B(\theta_l,\hat{t}_l)$:

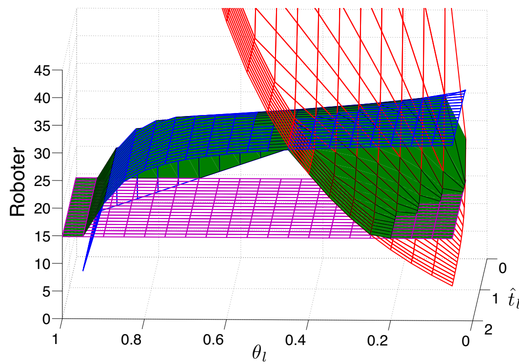
$$\widehat{\mathcal{W}}_A(\theta_l,\hat{t}_l) = \left(2 \cdot \frac{n - 2m}{n} + (\theta_l - \theta_s)\frac{t_e}{e_a(\theta_l, \hat{t}_l) \cdot t_l}\right) \cdot m \tag{6.14}$$

$$\widehat{\mathcal{W}}_B(\theta_l,\hat{t}_l) = l_{max} - (e_a(\theta_l, \hat{t}_l) + \theta_l - \theta_s)\frac{t_e}{e_a(\theta_l, \hat{t}_l) \cdot t_l} \tag{6.15}$$

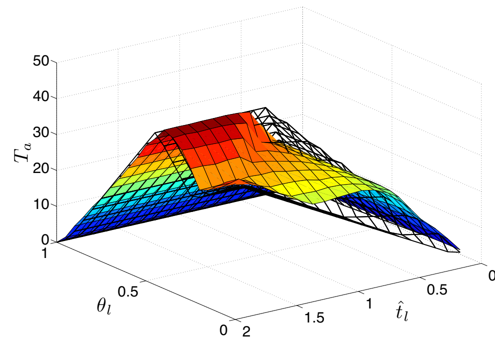
$\widehat{\mathcal{W}}_A(\theta_l,\hat{t}_l)$ ist die erwartete Anzahl an Robotern, die die Ladezeit der anderen Roboter überleben. $\widehat{\mathcal{W}}_B(\theta_l,\hat{t}_l)$ bestimmt die erwartete Anzahl der Roboter, die arbeiten können, ohne vor der Ladestation warten zu müssen.

Abbildung 6.13 zeigt die durch Gleichung (6.13) abgeschätzte Funktion $\widehat{\mathcal{W}}$ sowie die daraus berechnete Funktion T_a. Ein globales Maximum von T_a kann mit Hilfe eines einfachen iterativen Optimierungsverfahrens berechnet werden[1]. Zum Vergleich wurden hundert Parameterdurchläufe mit Algorithmus 6.1.1 über hundert simulierte Stunden ($T = 100$, $\Delta t = 1$) mit (θ_l, $\hat{t}_l$), $0{,}1 \leq \theta_l \leq 1$, $0{,}1 \leq \hat{t}_l \leq t_l$, zur Berechnung von T_a^m und $\mathcal{D}$ durchgeführt und über diese hundert Durchläufe an jedem Punktepaar (θ_l, $\hat{t}_l$) gemittelt ($\overline{T_a^m}$, $\overline{\mathcal{D}}$). Die anhand der Maxima von $\overline{T_a^m}$ und T_a berechneten Werte für $\hat{t}_l$, θ_l, T_a und $\mathcal{D}$ stimmen gut überein. Die Abweichungen zwischen $\overline{T_a^m}$ und T_a aus Gleichung (6.10) sind gering. Die Ergebnisse sind in Tabelle 6.4 dargestellt.

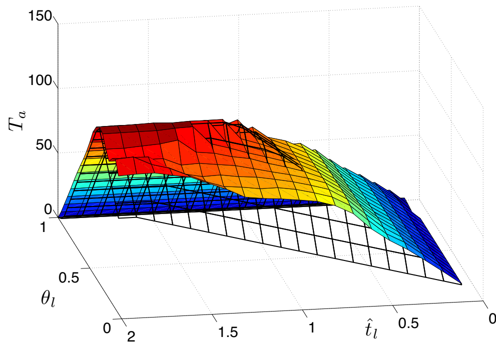
[1] Hier wurde das Programm *Mathematica* verwendet, um den Maximalwert zu bestimmen.



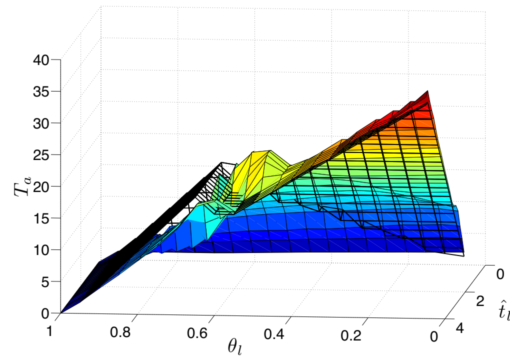
(a) Funktion $\widehat{W}$ (grün) zusammengesetzt aus $\widehat{W}_A$ (rot), $\widehat{W}_B$ (blau) und l_{min} (magenta).



(b) $t_l = 2$, $t_e = 2{,}5$, $n = 60$ und $m = 27$.



(c) $t_l = 2$, $t_e = 5$, $n = 60$ und $m = 18$.



(d) $t_l = 4$, $t_e = 2{,}5$, $n = 60$ und $m = 37$.

Abbildung 6.13: Abbildung (a) zeigt die anhand von Gleichung (6.13) abgeschätzte Funktion $\widehat{W}$ und deren Komponenten. Abbildungen (b) – (c) zeigen die Funktionen T_a (schwarz) und $\overline{T_a^m}$ (bunt) für verschiedene Startparameter.

Energie-/Ressourcenausbeute bei optimaler Parametrisierung unter Verwendung der Hop-Count-Strategie Mit Hilfe der Ergebnisse aus dem vorherigen Abschnitt und den Ergebnissen der Hop-Count-Strategie kann nun berechnet werden, wie viel Energie eine gesammelte Fuhre einer Ressource haben muss, um den Schwarm am Leben zu erhalten.

Der Wanda-Roboter ist mit zwei LiPoly-Zellen mit je $250mAh@3{,}7V$ ausgestattet, was eine Energie von $E_r = 2 \cdot 250mAh \cdot 3{,}7V = 1{,}85Wh$ entspricht. Basierend auf Messungen benötigt der Wanda-Roboter beim Laden ca. $107mA@3{,}7V$ was einer Leistung von $L_l = 0{,}396W$ entspricht. Zum Laden von m Robotern benötigen wir eine Leistung von

$$L_s(m) = m \cdot \left(\frac{E_r}{t_l} + L_l \right) = m \cdot \left(\frac{1{,}85Wh}{2h} + 0{,}396W \right) = m \cdot 1{,}321W.$$

t_l	t_e	m	Gleichungen (6.8) und (6.5)			
			$\hat{t}_l$	θ_l	$\overline{T_a^m}$	$\overline{\mathcal{D}^m}$
2	2,5	27	0,75	0,625	23,84	0,18
2	5	18	0,89	0,56	90,41	0
4	2,5	37	0,67	0,83	14,02	4,31
			Gleichung (6.10)			
			$\hat{t}_l$	θ_l	$T_a/\overline{T_a^m}$	$\overline{\mathcal{D}^m}$
2	2,5	27	1,75	0,47	40,91/39,64	5,26
2	5	18	1,68	0,42	109,69/114,86	3,21
4	2,5	37	4	$7{,}3 \cdot 10^{-9}$	31,5/34,91	22,49
			Algorithmus 6.1.1			
			$\hat{t}_l$	θ_l	$\overline{T_a^m}$	$\overline{\mathcal{D}^m}$
2	2,5	27	1,5	0,4	44,30	6,26
2	5	18	1,5	0,35	122,24	6,79
4	2,5	37	3,8	0,05	35,55	22,49

Tabelle 6.4: Für verschieden Lade- und Entladezeiten mit 60 Robotern berechnete Werte für $\hat{t}_l$, θ_l, T_a und $\mathcal{D}$. Die Werte $\overline{T_a^m}$ und $\overline{\mathcal{D}^m}$ ergeben sich aus dem Durchschnitt über 100 Durchläufe von 100 simulierten Stunden des Modells (Algorithmus 6.1.1).

Diese ist durch den Schwarm konstant zu erbringen. Im letzten Abschnitt konnten wir zeigen, dass wir den Schwarm so parametrisieren können, dass ungefähr l_{max} Roboter arbeiten und diese genügend Zeit zur Verfügung haben, den Weg von der Lagerstätte zum Sammelplatz zurückzulegen.

Daraus folgt, dass der Schwarm mindestens eine Größe von $n = m + a$ Robotern haben muss, wobei a die Anzahl der arbeitenden Roboter ist. Durch Einsetzen in Gleichung (6.3) erhalten wir dann,

$$m(a) = a \cdot \frac{t_l}{t_e}. \tag{6.16}$$

Das heißt, die zu erbringende Leistung der Ladestation in Abhängigkeit der arbeitenden Roboter a ist

$$L_s(m(a)) = a \cdot \frac{t_l}{t_e} \cdot \left(\frac{E_r}{t_l} + L_l\right) = a \cdot \frac{2h}{2{,}5h} \cdot 1{,}321W = a \cdot 1{,}0568W.$$

Betrachtet man die Anzahl an Fuhren $F(a,t_a)$, die a Roboter in einer Zeit t_a sammeln können, und die Energie einer einzelnen Fuhre E_F, so ist die Gesamtleistung des Schwarms

$$L_s(a) = \frac{F(a,t_a) \cdot E_F}{t_a}.$$

Unter der Annahme, dass zum Überleben die Leistung der Ladestation kleiner oder gleich der Leistung des Schwarms seien muss gilt für die minimale Energie einer Fuhre

$$E_f = \frac{a \cdot 1{,}0568W \cdot t_a}{F(a,t_a)}.$$

In dem durchgeführten Sammel-Experiment entspricht

$$F(a,t_a) \approx \left\lfloor \frac{S(a,t_a)}{2} \right\rfloor,$$

wobei $S(a,t_a)$ die in der Zeit t_a erreichte Punktzahl bei a sammelnden Robotern ist.

Daraus folgt, um HCS mit $a = 60$ Robotern, also einer Schwarmgröße von insgesamt $n = 108$ Robotern, zu betreiben wird für den Wanda eine Energie pro Fuhre von

$$E_f = \frac{60 \cdot 1{,}0568W \cdot 1200s}{\left\lfloor \frac{663{,}7}{2} \right\rfloor} = 229{,}88J$$

benötigt, da nach $t_a = 1200s$ eine Punktzahl von $S(a,t_a) = 663{,}7$ erreicht wurde. Aus der Optimierung der Parameter für eine Schwarmgröße von $n = 108$ durch die Bestimmung des Maximums von Gleichung (6.10) ergeben sich $\theta_l = 0{,}48$ und $\hat{t}_l = 1{,}66$. Daraus folgt, dass ein einzelner Roboter nach dem Laden $e_a(0{,}48\ ,\ 1{,}66) \cdot t_e = 0{,}52 \cdot 2{,}5h = 1{,}3h$ hat um Ressourcen zu sammeln, bis er sich wieder aufladen muss.

Tabelle 6.5 zeigt, wie viel Brennstoff in g pro Fuhre ein einzelner Roboter transportieren muss, um den Schwarm am Leben zu erhalten. Selbst bei einem sehr schlechten Wirkungsgrad von gerade mal 1% benötigt man gerade mal $2{,}74g$ frisches Holz pro Fuhre um den Schwarm am Leben zu erhalten. Der Wanda-Roboter kann ohne Probleme ein vielfaches dieser Menge schieben.

Brennstoff	Heizwert (in kJ/g)	g/Fuhre bei einem Wirkungsgrad von 25% (Kohlekraftwerk)	1%
Dieselöl	41,69	0,022056	0,5514
Heizöl aus Erdöl	42,95	0,021409	0,53523
Petroleum	40,85	0,02251	0,56274
Holz			
frisch	8,38	0,10973	2,7432
lufttrocken	15,08	0,060976	1,5244
Torf, lufttrocken	14,67	0,05516	1,379
Hartbraunkohle	16,76	0,054864	1,3716
Steinkohle			
Gasflammkohle	27,24	0,033756	0,84391
Fettkohle	31	0,029662	0,74155
Anthrazit	31	0,029662	0,74155
Braunkohlebriketts	19,7	0,046676	1,1669
Zechenkoks	27,23	0,033769	0,84422

Tabelle 6.5: Benötigte Masse in g pro Fuhre ($229{,}88J$) bei der Anwendung von HCS und Selbst-Laden mit $n = 108$ und $a = 60$ Robotern bei verschiedenen Brennstoffen und Wirkungsgraden. Heizwerte übernommen aus [53].

Bewertung

Das Experiment zeigt, wie ein in der Simulation entwickelter MDL2ϵ-Plan ohne Veränderungen auf den Roboter übertragen werden kann. Es stellt dabei anhand von Versuchen mit Robotern die Verbindung zwischen Simulation und realem Roboter her. Für die Analyse der Versuche mit echten Robotern übernimmt die entwickelte Arena eine wichtige Rolle. Zum einen wird die Arena verwendet, um den Bereich der Ressourcen für die Roboter zu markieren. Zum anderen werden die Versuche automatisch durchgeführt und Daten gesammelt.

Die in der formalen Analyse aufgestellten Formeln sind ein wichtiges Werkzeug bei der Entwicklung von Robotern, die in Schwärmen, Gruppen oder Teams zusammenarbeiten sollen, um abzuschätzen, wie das Verhältnis zwischen Lade- zu Entladezeit gewählt sein muss, bzw. wie groß die Autonomie der Roboter sein muss, um die gegebene Aufgabe zu erfüllen.

Die am Ende durchgeführte Analyse, bei der die Ergebnisse der formalen Analyse unter Verwendung der Ergebnisse der Hop-Count-Strategie zusammengeführt werden, zeigt beispielhaft anhand des realen Verbrauchs eines Wanda-Roboters, wie eine Berechnung der benötigten Ressourcen durchgeführt werden kann.

6.2 Genetische Programmierung

Zur Evaluation des Framework for Learning and Self-Organisation werden zwei Experimente in der Simulation auf Jasmine-Robotern und einer in der Arena auf den Wanda-Robotern durchgeführt. Ein Experiment verwendet dabei den konkurrierenden und das andere den nicht konkurrierenden Ansatz. Im ersten Experiment wird eine einfache *Kollisionsvermeidung* evolviert. Innerhalb dieses Experimentes werden die verschiedenen Selektionsstrategien sowie das Pruning evaluiert. Da die Kollisionsvermeidung ein sehr individuelles Verhalten ist, kommt hier der konkurrierende Ansatz zur Anwendung.

Im zweiten Experiment soll ein einfaches Schwarmverhalten, die *Aggregation* von Robotern, evolviert werden. Mit diesem Experiment wird der nicht konkurrierende Ansatz evaluiert. Es wird im Anschluss auf den Wanda-Roboter übertragen, um die Übertragbarkeit von evolviertem Verhalten auf echte Roboter zu testen. Tabelle 6.6 gibt eine Übersicht der hier verwendeten Ressourcen

Tabelle 6.6: Übersicht über die in den Experimenten verwendeten Teile der Hard- und Software.

Nr.	Experiment	MDL2ϵ	Simulation	Roboter	Arena
I	Kollisions-vermeidung	ja	Breve	Jasmine	–
II	Aggregation	ja	Breve	Jasmine/ Wanda	Projektor/ ZigBee

6.2.1 Kollisionsvermeidung

Kollisionsvermeidung ist ein, von vielen Wissenschaftlern unterschätztes, zentrales Thema der Schwarmrobotik. Die jeweils verwendete Kollisionsvermeidungsstrategie kann sich bei steigender Schwarmdichte stark auf die Leistungsfähigkeit des Schwarmes auswirken.

In dem folgenden Experiment wollen wir anhand der Kollisionsvermeidung die verschiedenen Teilbereiche des FLSO testen und auswerten. Die untersuchten Aspekte sind die verschiedenen Selektionstrategien FPS, RBS und TMS, sowie die Auswirkung von Pruning. Zunächst aber wird die verwendete Fitnessfunktion sowie der dazugehörige Aufbau der Arena beschrieben.

Fitnessfunktion

Die Fitnessfunktion ist einer der elementarsten und auch schwierigsten Teile der Evolutionary Computation. Eine schlecht gewählte Fitnessfunktion kann zu nicht gewünschten Verhalten führen oder aber auch die Evolution früh zu lokalen Extrema hin konvergieren lassen. Eine Fitnessfunktion zur Synthese eines Verhaltens zur Vermeidung von Kollisionen, die zwar schnell zu Evolution eines einfachen Verhaltens zur Kollisionsvermeidung führt, aber wohl nicht im Sinne des Designers ist, ist die Belohnung des Individuums für jeden Zeitschritt, in dem es keine Kollision gibt. Dies evolviert zu einem Verhalten, bei dem der Roboter entweder einfach nur auf der Stelle rotiert oder aber gleich einfach stehen bleibt. –Was sich nicht bewegt, kann nicht kollidieren!–

Um solche Probleme zu vermeiden, wurde eine Dekomposition des gewünschten Verhaltens durchgeführt und eine Fitnessfunktion entworfen, die sich aus mehreren Teilen zusammensetzt, wobei jeder Teil ein bestimmtes

Verhalten belohnt.

Die Fitnessfunktion (6.17) setzt sich daher aus drei Teilen zusammen, die mit unterschiedlicher Gewichtung in die individuelle Fitness eingehen. Als vierten Teil betrachten wir die Gestaltung der Arena selbst, sie enthält verschiedene Bereiche, die wichtig für die Entwicklung des Verhaltens sind.

$$f_{total}(t,x) = (0{,}2 f_{move} + 0{,}2 f_{duration}(t) + 0{,}6 f_{distance}(x))\, f_{initial} \qquad (6.17)$$

Im Falle der Fitnessfunktion (6.17) wird die Fitness mit einem initialen Faktor $f_{initial} = 1.000$ skaliert. Die Fitness eines Individuums bewegt sich zwischen null, der besten erreichbaren Fitness, und 1.000 der schlechtesten Fitness. Diese Art der Darstellung geht auf Details in der Implementierung des FLSO zurück.

Die während der Evolution verwendeten Arena ist in Abbildung 6.14 dargestellt. Sie besteht aus einem Start- und Zielbereich (engl. *take-off/target area*), sowie einem Bereich dazwischen. Der Bereich zwischen dem Start- und Zielbereich ist so entworfen, dass verschiedene Verhalten verstärkt werden.

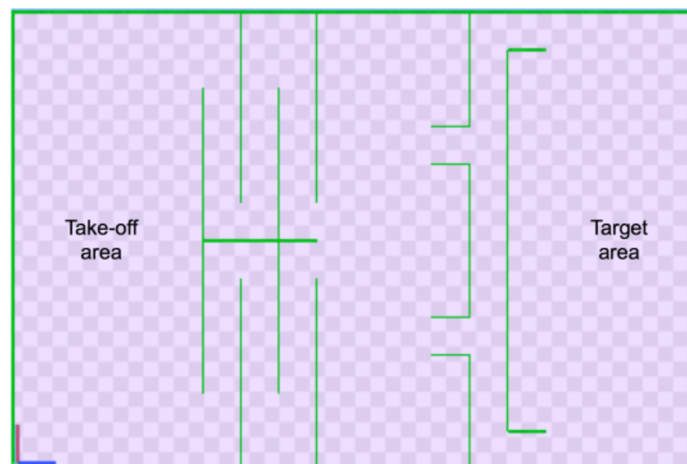


Abbildung 6.14: Aufbau der Arena [33].

Betrachten wir nun die drei Teile der Fitnessfunktion und deren Aufgaben.

Exploration (move) – 20% Um am Anfang der Evolution exploratives Verhalten, Bewegung durch die Arena, zu fördern, belohnt ein Teil der Fitnessfunktion Verhalten, in denen sich der Roboter vorwärts bewegt. Hierbei wird das zeitliche Verhältnis zwischen der Ausführung des

Atoms AMOVE_FWD (Vorwärtsbewegung) und allen anderen ausgeführten Atomen bestimmt. Je höher der Anteil an AMOVE_FWD desto besser ist die Fitness des einzelnen Individuums.

$$f_{move} = 1 - \frac{a_{move_fwd}}{a_{all}} \tag{6.18}$$

Zeit (duration) – 20% Um am Ende der Evolution Individuen zu belohnen, die den Zielbereich schnell erreicht haben, fließt die Zeit, die zum Erreichen des Zielbereiches benötigt worden ist, mit in die Fitness ein. Dabei muss der Zielbereich innerhalb einer gewissen Zeit $t_{threshold}$ erreicht werden, um eine Verbesserung der Fitness zu erzielen. Der verwendete Standardwert ist $t_{threshold} = 300$. t_{max} ist die maximale Zeit, die den Individuen zur Evaluation der Fitness gegeben wird (hier $t_{max} = 600$).

$$f_{duration}(t) = \begin{cases} \frac{t - t_{threshold}}{t_{max} - t_{threshold}} & \text{, wenn } t \geq t_{threshold} \\ 0 & \text{, wenn } t < t_{threshold} \\ 1 & \text{, Zielbereich wird nicht erreicht} \end{cases} \tag{6.19}$$

Distanz (distance) – 60% Der größte Anteil an der Fitness wird für das Durchqueren der Arena vom Start zum Zielbereich vergeben. Die Arena wird dafür, wie in Abbildung 6.15 dargestellt, in sechs Bereiche unterteilt. Die verschiedenen Bereiche haben einen unterschiedlich starken Einfluss auf die Fitness. Die in Gleichung (6.20) verwendeten Brüche beziehen sich auf eine Normierung der Länge der Arena auf eins.

$$f_{distance}(t,x) = \begin{cases} 1 & \text{, wenn } \max(x(t)) < \frac{5}{18} \\ 1 - \frac{18}{10}(x - \frac{5}{18}) & \text{, wenn } \frac{5}{18} \leq \max(x(t)) < \frac{6}{18} \\ \frac{9}{10} - \frac{3 \cdot 18}{10 \cdot 2}(x - \frac{6}{18}) & \text{, wenn } \frac{6}{18} \leq \max(x(t)) < \frac{8}{18} \\ \frac{6}{10} - \frac{2 \cdot 18}{10 \cdot 4}(x - \frac{8}{18}) & \text{, wenn } \frac{8}{18} \leq \max(x(t)) < \frac{12}{18} \\ \frac{4}{10} - \frac{4 \cdot 18}{10 \cdot 2}(x - \frac{12}{18}) & \text{, wenn } \frac{12}{18} \leq \max(x(t)) < \frac{14}{18} \\ 0 & \text{, wenn } \max(x(t)) \geq \frac{14}{18} \end{cases} \tag{6.20}$$

Bereich ① in Abbildung 6.15, $\max(x(t)) \geq \frac{14}{18}$, ist der Startbereich, in dem die Roboter mit einer zufälligen Position starten. Hier hat die zurückgelegte Entfernung keinen Einfluss auf die Fitness. Erreichen die

Roboter Bereich ② werden sie leicht belohnt. Roboter die es hierher schaffen, haben zumeist schon eine einfache Kollisionsvermeidungsstrategie gelernt, welche zumeist aus dem Wegdrehen in der Nähe eines Hindernisses besteht. Um eine Kollisionsvermeidungsstratgie zu erhalten, bei der die Roboter in beide Richtungen abdrehen können, wird Bereich ③ hinzugefügt. Die sich aus Bereich ② und ③ zusammensetzende siphonartige Struktur belohnt Verhalten, bei denen die Roboter nach links und rechts ausweichen. Die Bereiche ④ und ⑤ erhöhen die Explorationsfähigkeit des Roboters. Gelangt der Roboter in Bereich ⑥, so wird er gestoppt. Roboter die einander oder eine Wand berühren werden angehalten und aus der Arena entfernt. Sie können innerhalb dieser Runde ihre Fitness nicht weiter verbessern.

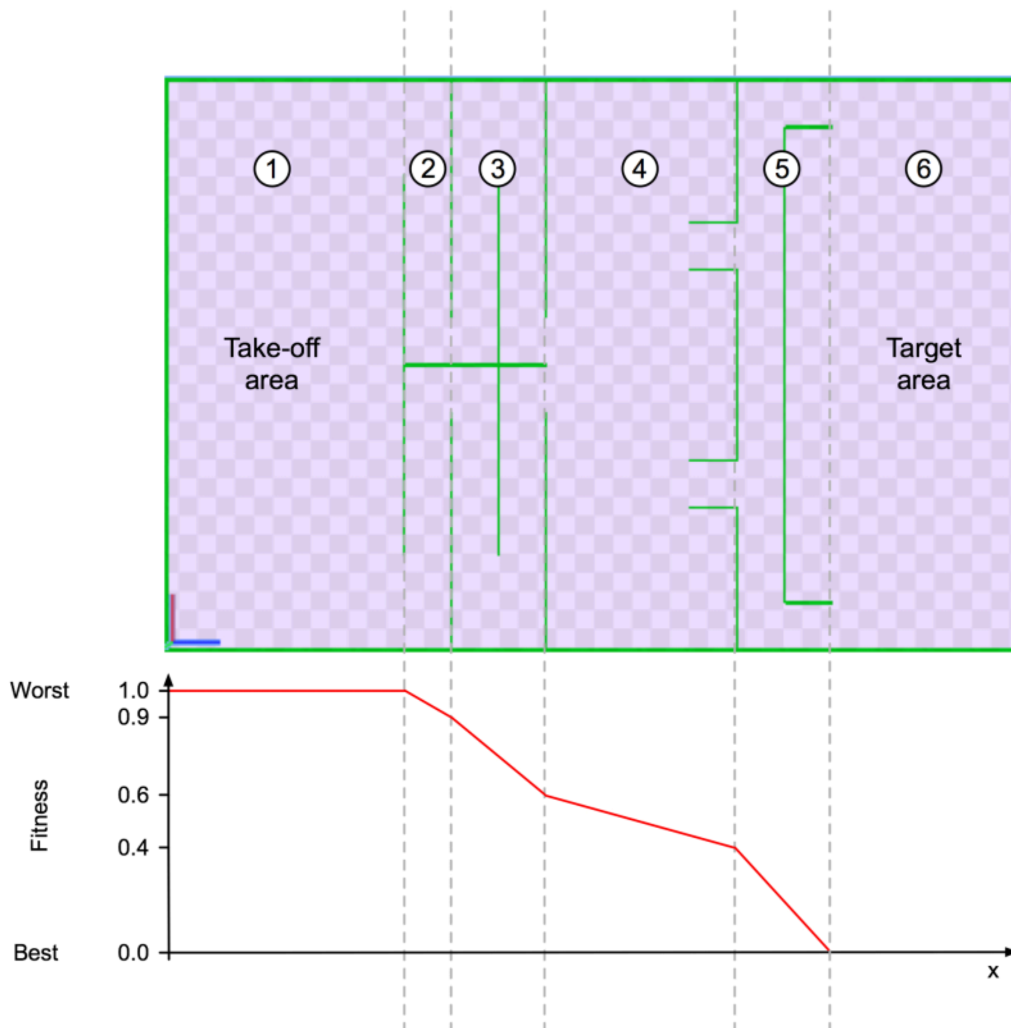


Abbildung 6.15: Abhängigkeit von Position und Fitness [33].

Parametrisierung

Die Parametrisierung aller Experimente zur Evolution von Kollisionsvermeidung ist in Tabelle 6.8 zusammengefasst. Es kommen dabei vier Atome zur Anwendung, die die Bewegung des Roboters steuern. Drei Interrupts regeln den Ablauf der Pläne. ITRUE ist immer wahr, ISPACEL ist wahr, wenn links vorne mehr Abstand zu einem Hindernis ist als rechts vorne und ICRITCOLLISION gibt das Vorhandensein eines Hindernisses innerhalb einer Roboterlänge auf einem der drei vorderen Sensoren an.

Ein Versuch geht über 31 Generation mit insgesamt 945 Individuen verteilt auf 63 Arenen mit jeweils 15 Individuen. Jedes Experiment setzte sich aus zehn Versuchen zusammen.

Tabelle 6.7: Parametrisierung der Evolution von Kollisionsvermeidung.

Atome	ASTOP, AMOVE_FWD, AROT_R und AROT_L
Interrupts	ITRUE, ISPACEL und ICRITCOLLISION
Generationen	31
Simulierte Zeit	600 Zeitschritte
Individuen	945
Roboter pro Arena	15
Arenen pro Generation	63
Selektionsmechanismen	RBS, FPS und TMS
Crossover Rate	70%
Mutationsrate	30%, davon: 50% Subtree Mutation, 50% Neighbour Mutation
Elitist Strategie	25

Experimente mit verschiedenen Selektionsstrategien in der Simulation

Es wird zu jeder Selektionsstrategie ein Experiment durchgeführt und dabei die Auswirkung der Selektiosstrategie auf die Fitness und Diversität analysiert.

Für die RBS Selektionsstrategie wird die Negativ-Exponentielle-Rangfunktion, siehe Gleichung (5.2), mit $a = 1$, $b = 0{,}01$ und $c = 1$ gewählt. Dies Entspricht einer Selektionswahrscheinlichkeit eines Individuums aus

den ersten 100 bzw. 25 Rängen von 60% bzw. 20%. Bei der TMS wird eine Tourniergröße von 25 festgelegt, was einer Selektionswahrscheinlichkeit eines Individuums aus den ersten 100/50/25 Rängen von 93%/72%/47% entspricht. FPS ist frei von Parametern.

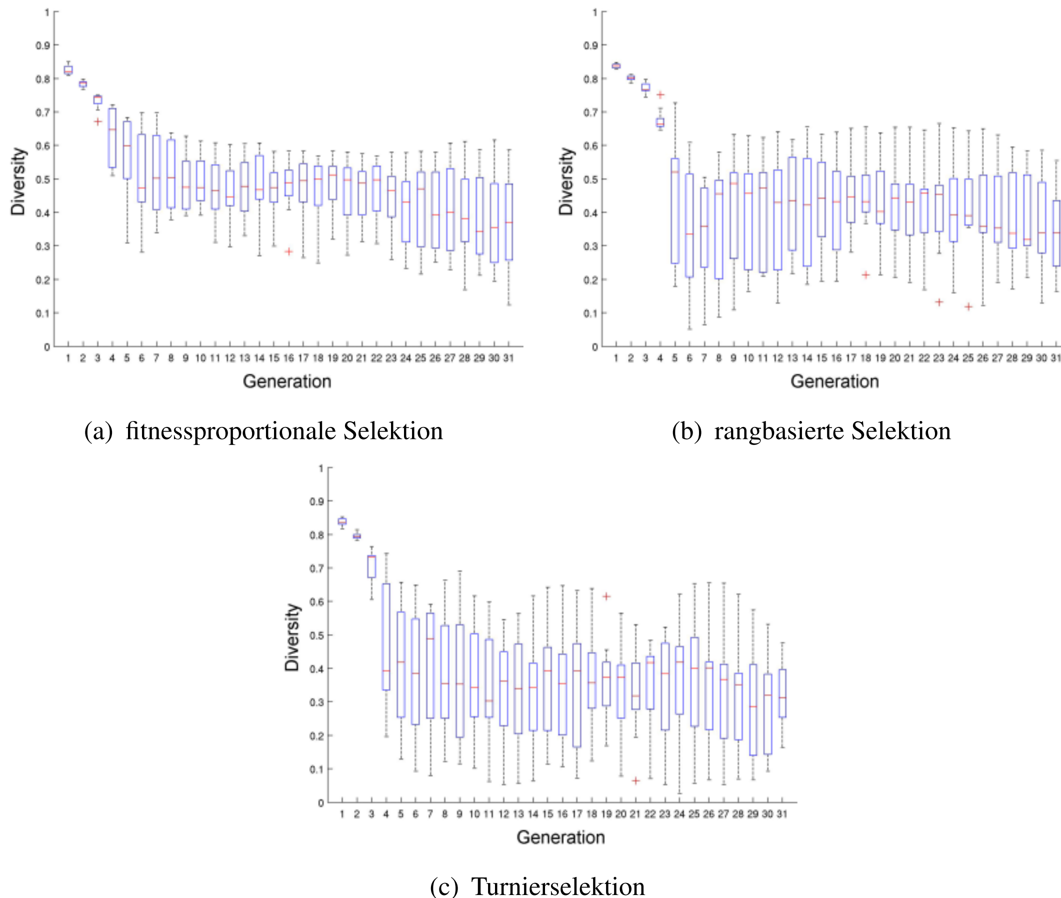
Die Darstellung der Experimente in den folgenden Abbildungen findet mit einer Kastengrafik (Box-Whisker-Plot) statt. Es fasst verschiedene robuste Streuungs- und Lagemaße in einer Darstellung zusammen. Es werden der Median (roter Strich), die zwei Quartile 75% und 25% (Balkenenden), die beiden Extremwerte ("Whisker"), die keine Ausreisser sind, und die Ausreisser (Sterne) dargestellt.

Die Entwicklung der Diversität während des Experimentes, wie in Abbildung 6.16 gezeigt, verhält sich bei den verschiedenen Selektionsmechanismen recht ähnlich. Sie startet auf einem Wert um 0,84 mit einer recht geringen Abweichung, was für die Güte des entwickelten Verfahrens für die Initialisierung der Population spricht. Danach fällt sie innerhalb der ersten fünf Generationen auf einen Wert um 0,4, auf den sich die Diversität dann mit leichter Abwärtstendenz einpendelt. Je nach der Ausprägung des Selektionsdrucks (TMS > RBS > FPS) sieht man leichte Schwankungen eher oberhalb bzw. unterhalb der Diversitätsmarke von 0,4.

Die Entwicklung der Fitness während des Experimentes, wird anhand von zwei Werten in Abbildung 6.17 illustriert. Diese sind die Entwicklung der Fitness des besten Individuums, sowie die durchschnittlichen Fitness einer Generation.

Die durchschnittliche Fitness verhält sich bei allen drei Experimenten ähnlich. Sie fällt langsam von 1.000 auf ca. 750 ab. Die rangbasierte Selektion liegt im Durchschnitt etwas darunter und die Turnierselektion etwas darüber.

Interessanter sind jedoch die Verläufe der Fitness der besten Individuen einer Generation. Es ist zu beobachten, dass es zwei Verhalten gibt, zu denen die Evolution konvergiert, ein lokales Minimum um einen Fitness Wert von 600 und dem globalen Minimum unter 50. Hier zeigt die RBS ein stabiles Konvergenzverhalten zu einer guten Lösung unter 50 mit einer nur sehr geringen Standardabweichung. Selbst Ausreisser, die noch zwischen der 12. und 24. Generation vorkommen, konvergieren am Ende zu einer optimalen Lösung. Die TMS zeigt, dass sie nicht immer im Stande ist gegen die optimale Lösung zu konvergieren, sondern vorzeitig im lokalen Minimum



(a) fitnessproportionale Selektion

(b) rangbasierte Selektion

(c) Turnierselektion

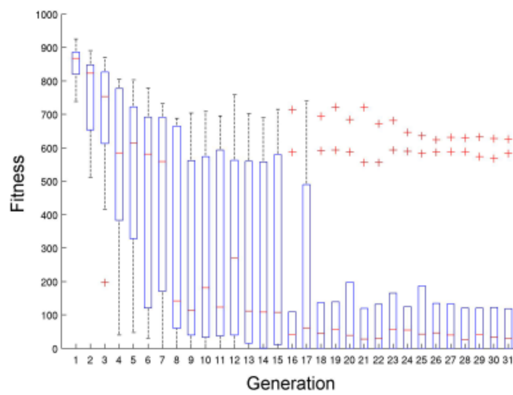
Abbildung 6.16: Auswirkung der Selektionsstrategie auf die Diversität. Experiment mit jeweils zehn Versuchen pro Selektionsstrategie [33].

bei 600 verharrt. Die fitnessproportionale Selektion konvergiert zwar in den meisten Fällen zu einer guten Lösung, zeigt aber auch, dass sie im lokalen Minimum stecken bleibt.

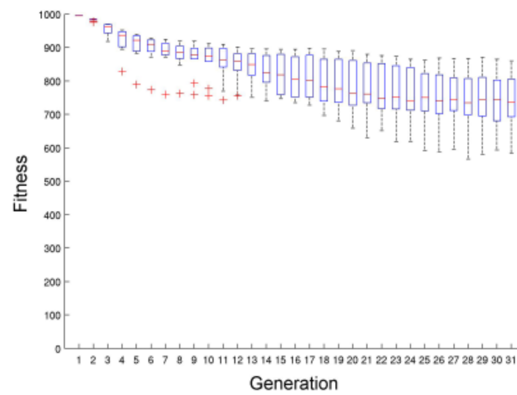
Man sieht hier deutlich, dass die Selektionsmechanismen, die eine höheren Diversität erhalten, stabiler gegen das globale Optimum konvergieren. Hierbei zeigt die RBS bei der gegebenen Parametrisierung das wohl beste Verhalten.

Analyse der Fitnesslandschaft anhand der mittlere Fitnessverteilung innerhalb verschiedener Generationen. Hierzu werden die Individuen jeder fünften Generation in Cluster mit einer Auflösung von jeweils 50 Fitnesspunkten aufgeteilt und die Größe der Cluster bestimmt. Die Graphen in der linken Spalte von Abbildung 6.18 zeigen den Verlauf über die gesamte Clustergröße, rechts wird die Clustergröße auf 50 beschränkt.

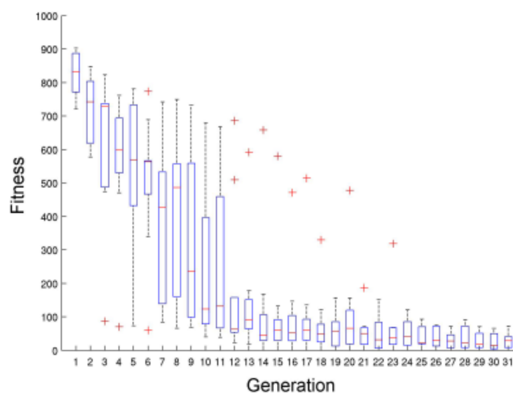
Mit zunehmender Generation flachen die Kurven ab, da sich die gleich-



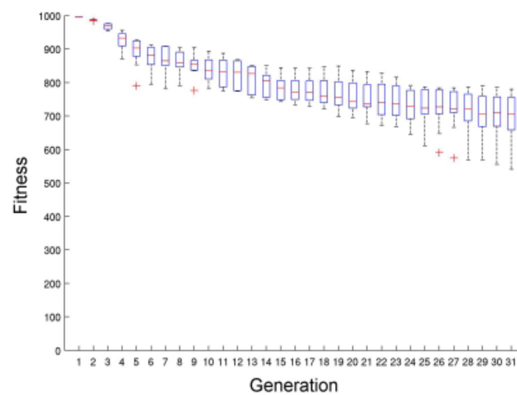
(a) Verteilung der besten Individuen, FPS



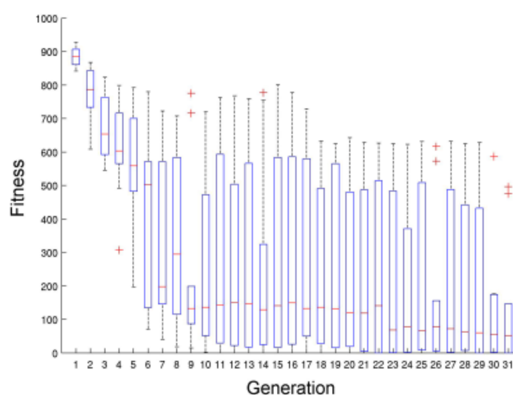
(b) Verteilung der mittleren Fitness, FPS



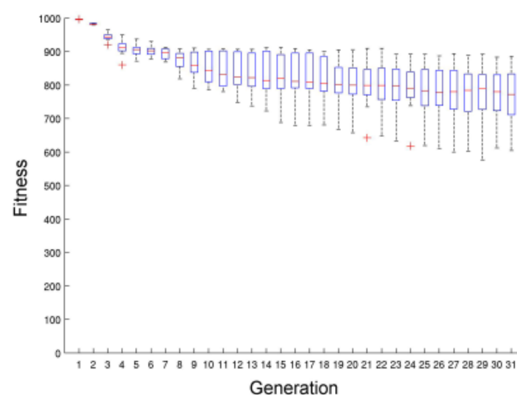
(c) Verteilung der besten Individuen, RBS



(d) Verteilung der mittleren Fitness, RBS



(e) Verteilung der besten Individuen, TMS



(f) Verteilung der mittleren Fitness, TMS

Abbildung 6.17: Verteilung der jeweils besten (links) sowie der mittleren (rechts) Fitness aller Selektionsstrategien über ein Experiment mit jeweils zehn Versuchen [33].

bleibende Anzahl von 945 Individuen über den gesamten Fitnessbereich verteilt. Es bilden sich jedoch drei lokale Maxima um die Fitnesswerte 100, 600 und 850, und ein fast leerer Bereich zwischen 200 und 400 Fitnesspunkten heraus. Die Maxima lassen sich anhand der Dekomposition der Fitnessfunktion aus Gleichung (6.17) ablesen.

Betrachten wir zunächst das Maximum zwischen 825 und 875, so entspricht dies Robotern, die sich im Bereich ① aufhalten und ihre Fitness daher nur von f_{move} abhängt. Setzen wir $f_{distance} = f_{duration} = 1$, so erhalten wir nach Gleichung (6.17) einen Basisfitnesswert von 800. Die weiteren 25 – 75 Fitnesspunkte gehen auf f_{move} zurück und besagen, dass die Individuen 60% – 90% ihrer Lebenszeit AMOVE_FWD ausgeführt haben. Sie sind somit schon relativ mobil, bzw. haben es gelernt auszuweichen. Dies kann man auch durch die während der Evolution entstandenen Pläne veranschaulichen. Betrachtet man die folgenden bereinigten[2] Listings 6.1 und 6.2 nach der ersten bzw. sechsten Runde, so stellen die Zahlen vor den einzelnen Operatoren die Ausführungsdauer dar. Bei beiden Plänen wird AMOVE_FWD zu 65% bzw. 54% ausgeführt. Die Pläne erreichten eine Fitness von 853 bzw. 871 Fitnesspunkten. Sie erzeugen ein Verhalten, bei dem das Individuum immer nur in eine Richtung nach links bzw. rechts ausweichen kann. Dies verhindert das Durchqueren des Bereichs ③.

Listing 6.1: Runde: 1, Individuum: 1125, Elternteil A: 91, Elternteil B: 56, Fitness: 853,567, Selektion: FPS

```
1 10504 <PLAN duration="infinite">
2 06579  <ATOM duration="10" interrupt="NOT(ICRITCOLLISION)"
3         name="AMOVE_FWD"/>
4 03925  <ATOM duration="11" interrupt="ICRITCOLLISION"
5         name="AROT_L"/>
6       </PLAN>
```

Listing 6.2: Runde: 6, Individuum: 5824, Elternteil A: 5349, Elternteil B: 5226, Fitness: 871,698, Selektion: RBS

```
1 10673 <PLAN duration="infinite">
2 10673  <BEHAVIOUR duration="36" interrupt="ITRUE">
3 04323   <ATOM duration="33" interrupt="ICRITCOLLISION"
4          name="AROT_R"/>
5 05354   <ATOM duration="10" interrupt="NOT(ICRITCOLLISION)"
6          name="AMOVE_FWD"/>
7 00996   <ATOM duration="81" interrupt="ISPACEL"
8          name="AROT_R"/>
```

[2]Zur besseren Lesbarkeit wird nur die Spur des Plans dargestellt.

```
9          </BEHAVIOUR>
10       </PLAN>
```

Das Maximum bei 600 kommt durch die Hinzunahme von $f_{distance}$. Diese Roboter haben ihren Fitnesswert um circa 300 Fitnesspunkte erniedrigt, was $f_{distance} = 0{,}5$ entspricht. Dieser Wert wird in Bereich ④ erreicht. Individuen, die diesen Bereich erreichen, haben es gelernt nach links und nach rechts auszuweichen, um die siphonartige Struktur im Bereich davor zu durchqueren. Betrachtet man den Plan aus Listing 6.3, so ist hier der Übergang in Runde sieben gut zu erkennen. Er ergibt sich durch die Kreuzung von Individuum 5824 aus Listing 6.2 mit Individuum 6573 an dem Kreuzungspunkt in Zeile 7. Hier wird eine Drehung nach links eingeführt, was zu einem Verhalten führt, dem es möglich ist, Bereich ④ zu erreichen. Das Individuum erreicht eine Fitness von 628 in der Nähe des angesprochenen Maximums.

Listing 6.3: Runde: 7, Individuum: 6770, Elternteil A: 5824, ElternteilB: 6573, Fitness: 628,664, Selektion: RBS

```
1  10714 <PLAN duration="infinite">
2  10714  <BEHAVIOUR duration="36" interrupt="ITRUE">
3  03671    <ATOM duration="33" interrupt="ICRITCOLLISION"
4             name="AROT_R"/>
5  05706    <ATOM duration="10" interrupt="NOT(ICRITCOLLISION)"
6             name="AMOVE_FWD"/>
7  01337    <ATOM duration="88" interrupt="ISPACEL"
8             name="AROT_L"/>
9          </BEHAVIOUR>
10       </PLAN>
```

Zwischen einem Fitnesswert von 400 und 200 beginnt ein Bereich, in dem man fast keine Individuen findet. Die Grenze von 400 entspricht ungefähr einem Individuum, welches bis in den Bereich ⑤ bzw. an dessen Grenzen gekommen ist, aber nicht den Bereich ⑥ erreicht hat. Denn aus dem Sprung von 200 Fitnesspunkten kann man erkennen, dass Individuen, die den Bereich ⑥ erreicht haben, dies bis auf wenige Ausnahmen auch innerhalb von $t_{threshold}$ geschafft haben. Listing 6.4 zeigt ein Individuum aus Generation 30, welches den Bereich ⑥ erreicht hat. Dieses Individuum ist mit einer Fitness von 26 das Beste seiner Generation.

Listing 6.4: Runde: 30, Individuum: 28366, Elternteil A: 28253, Elternteil B: – (*Mutation*), Fitness: 26,307, Selektion: RBS

```
1  6050 <PLAN duration="infinite">
2  6050  <UNION>
3  4210    <ATOM duration="54" interrupt="NOT(ICRITCOLLISION)"
```

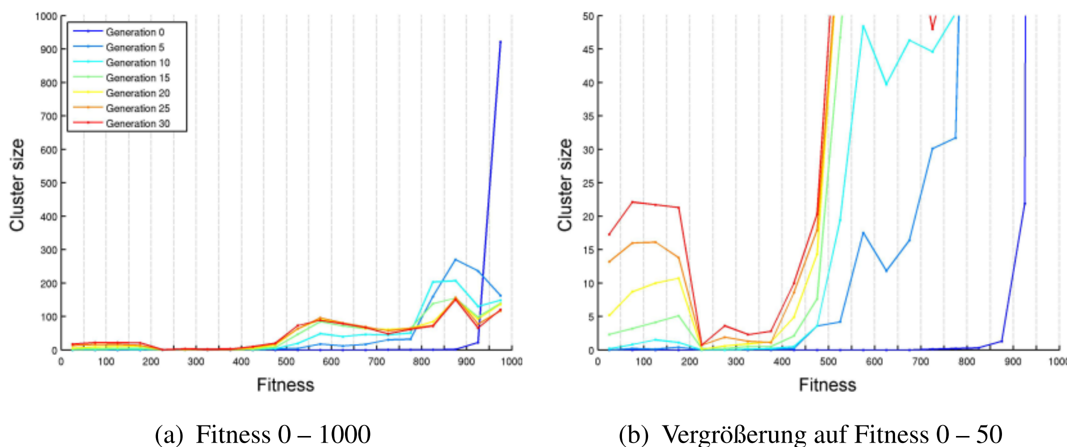
```
4              name="AMOVE_FWD" />
5   0230       <ATOM duration="4" interrupt="NOT(ISPACEL)"
6              name="AROT_R"/>
7   1610       <ATOM duration="32" interrupt="ICRITCOLLISION"
8              name="AROT_L"/>
9            </UNION>
10         </PLAN>
```

Ausgehend von dieser Argumentation kann man, basierend auf dem letzen lokalen Maximum in dem Bereich um 100, wiederum die durchschnittliche Ausführungszeit von AMOVE_FWD berechnen. Diese ergibt dann nach Gleichungen (6.17) und (6.18)

$$\frac{a_{move_fwd}}{a_{all}} = 1 - \frac{f_{total}}{f_{initial} \cdot 0{,}2} \rightarrow \frac{a_{move_fwd}}{a_{all}} = 1 - \frac{100}{1000 \cdot 0{,}2} = 0{,}5.$$

D.h. die Roboter mussten ca. die Hälfte der Zeit Hindernissen ausweichen.

Diese Analyse der Fitnesslandschaft ist unabhängig von der gewählten Selektionsstrategie und spiegelt sich daher in den Graphen aller Selektionsmechanismen wider. In Abbildung 6.18 wird deshalb stellvertretend für alle die RBS ausgewählt.



(a) Fitness 0 – 1000

(b) Vergrößerung auf Fitness 0 – 50

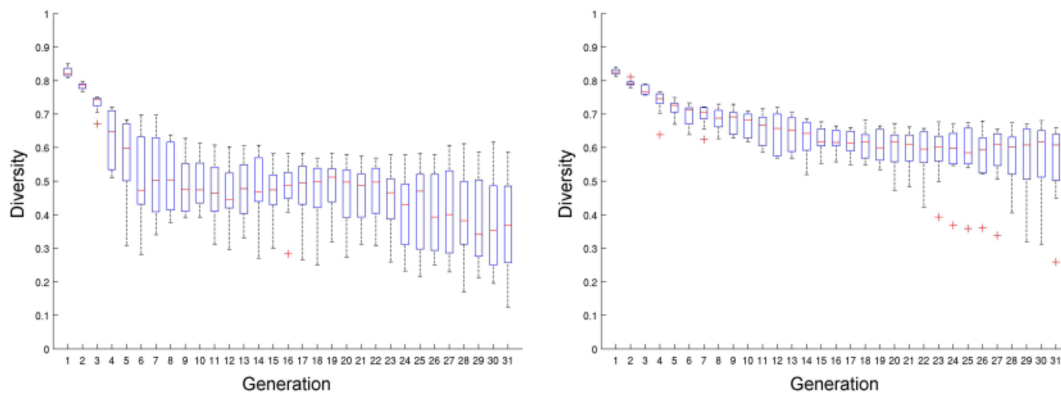
Abbildung 6.18: Histogramm der durchschnittlichen Fitnessverteilung für Generationen 0, 5, 10, . . ., 30 über die zehn Versuche der rangbasierte Selektion aus den Daten von Abbildung 6.17(d) [33].

Experimente mit Pruning

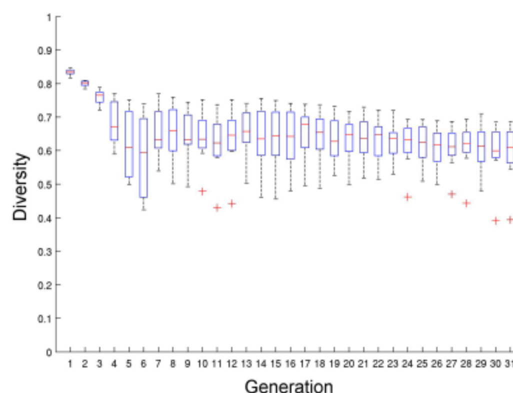
Pruning ist, wie in Abschnitt 5.5 beschrieben, eine Möglichkeit, Einfluss auf die Entwicklung der Diversität und damit auch auf den Verlauf der Fitnessentwicklung während der Evolution zu nehmen. Hierfür wurden weitere

Experimente mit FPS durchgeführt, bei denen gar nicht, auf jede Generation oder auf jede fünfte Generation Pruning auf die Pläne vor der Erstellung einer neuen Population angewendet worden ist. Die Parametrisierung des Pruningverfahrens ist dabei $m = 3$ mit $\alpha = 0{,}1$. Die Wahl der Parameter bezieht nur Ergebnisse der Kreuzkorrelation in die Berechnung der Diversität mit ein, die nicht zu den 99,73% der Punkte gehören, die nahe am Mittelwert liegen. Der Wert $alpha$ ist experimentell motiviert und erlaubt auch den kürzeren Übereinstimmungen einen Einfluss auf das Diversitätsmaß.

Beobachten wir im vorherigen Abschnitt eine Abnahme der Diversität auf ungefähr 0,4 bei allen Selektionsstrategien, so wirkt sich das Pruning, unerheblich ob bei jeder oder nur jeder fünften Generation, positiv auf die Diversität aus. Abbildung 6.19 zeigt deutlich, wie durch das Pruning die mittlere Diversität über zehn Versuche zwischen 0,6 und 0,7 gehalten werden kann. Auch verringert sich die Varianz der Diversität deutlich. Pruning verhindert somit erfolgreich den Einbruch der Diversität, der sonst nach den ersten fünf Generationen auftritt.



(a) kein Pruning (b) Pruning nach jeder Generation



(c) Pruning nach jeder 5. Generation

Abbildung 6.19: Der Einfluss von Pruning auf die Diversität [33].

Dies hat einen direkten Einfluss auf die Fitness, wie in Abbildung 6.20 zu sehen. Bei der Anwendung von Pruning nach jeder Generation konvergiert die Fitness des besten Individuums, im Durchschnitt nach nur acht Generationen, auf eine Fitnesswert unter 200. Sie sinkt dann in den darauffolgenden Generationen auf Werte unter 50. Eine Stagnation in einem lokalen Optimum fand nicht statt. Gutes Verhalten kann sich hier wohl schneller durchsetzen, da die genetischen Operatoren auf der Spur des Plans arbeiten, und so einen direkteren Einfluss auf die neue Generation ausüben.

Findet Pruning alle fünf Generationen statt, so zeigen sich keine Verbesserungen gegenüber dem Experiment ohne Pruning. Dieses Experiment scheint im Mittel sogar langsamer zum globalen Optimum zu konvergieren. Allerdings werden auch hier im Mittel leicht bessere Werte erreicht als bei keinem Pruning. Dies lässt sich wiederum darauf zurückzuführen, dass die genetischen Operatoren jede fünfte Generation auf wirklich ausgeführten Programmteilen arbeiten.

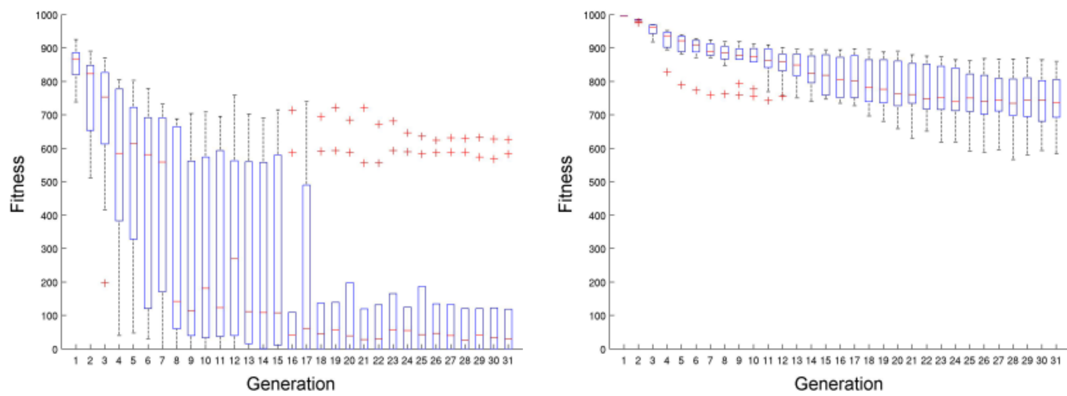
Betrachtet man die Entwicklung der mittleren Fitness, so verläuft sie in allen drei Fällen recht ähnlich und unterscheidet sich eher in der Varianz. Man erkennt allerdings beim Pruning jeder fünften Generation, leichte Verschlechterungen in der mittleren Fitness. Dies kommt offensichtlich daher, dass die Wahrscheinlichkeit, gutes Verhalten durch die genetischen Operatoren zu zerstören, nach dem Pruning sehr hoch ist. Dies wird in Abbildung 6.21 deutlich sichtbar, da hier die mittlere Anzahl der Programmzeilen (*Lines of Code (LoC)*) dargestellt ist.

Ohne Pruning wächst diese linear an, vgl. Abbildung 6.21(b), was die Wahrscheinlichkeit eines einzelnen Knotens, von einem genetischen Operator ausgewählt zu werden, verringert. Der Anteil an nicht ausgeführtem Code steigt hingegen nur leicht, vgl. Abbildung 6.21(a). Wird Pruning angewendet, so wird in allen Fällen sowohl die LoC klein gehalten, als auch der Anteil an nicht ausgeführtem Code stark reduziert, siehe Abbildung 6.21(e)–(d).

Das schnelle Kovergenzverhalten beim Pruning nach jeder Generation ist mehr auf diesen Sachverhalt zurückzuführen, als auf eine Verbesserung der Diversität.

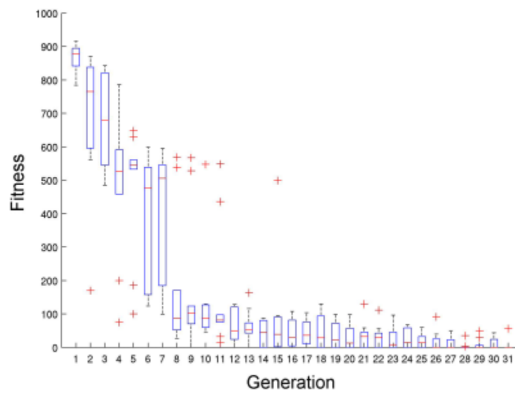
6.2.2 Aggregation

Aggregation ist ein Schwarmverhalten, bei dem sich die Roboter in einem möglichst großen Cluster aggregieren sollen. Dieses Verhalten kann genutzt



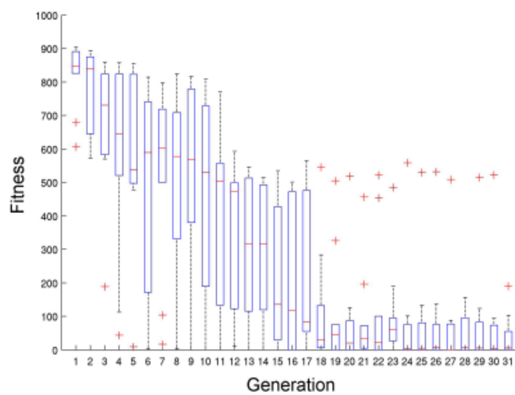
(a) kein Pruning

(b) kein Pruning



(c) Pruning nach jeder Generation

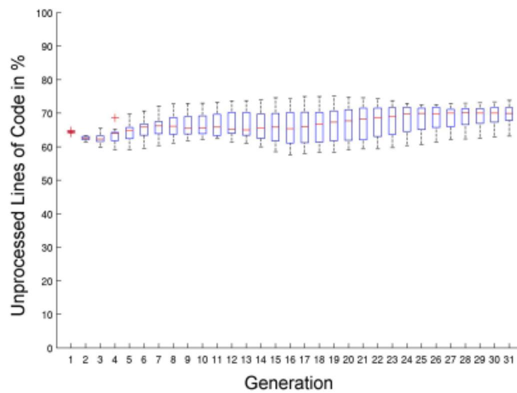
(d) Pruning nach jeder Generation



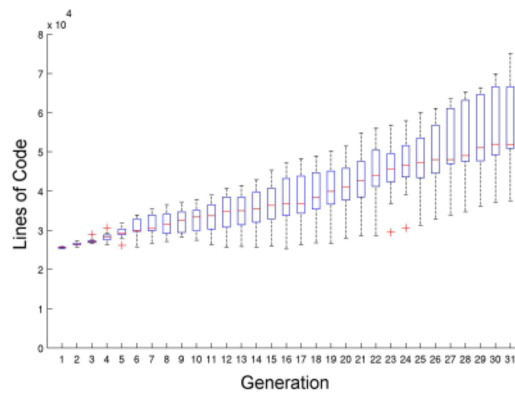
(e) Pruning jede 5. Generation

(f) Pruning jede 5. Generation

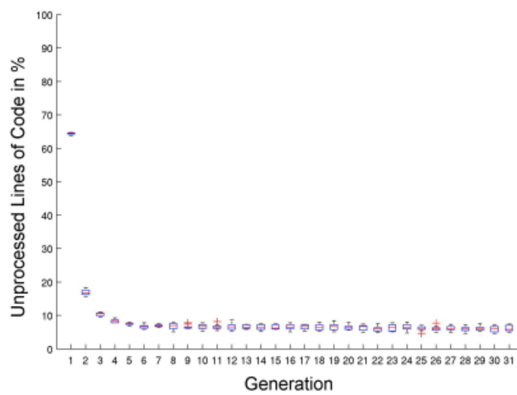
Abbildung 6.20: Betrachtung der Fitnessentwicklung bei der Anwendung von Pruning. In der linken Spalte wird die Verteilung der besten Individuen und in der rechten Spalte die Verteilung der mittleren Fitness dargestellt [33].



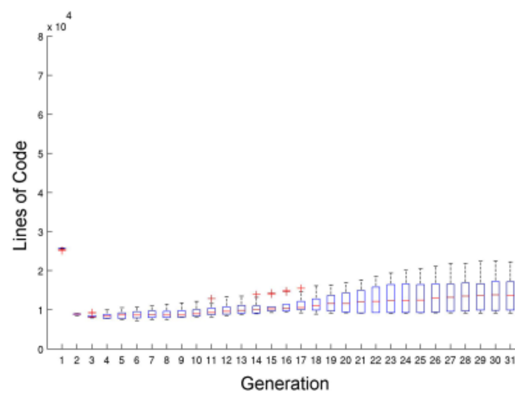
(a) kein Pruning



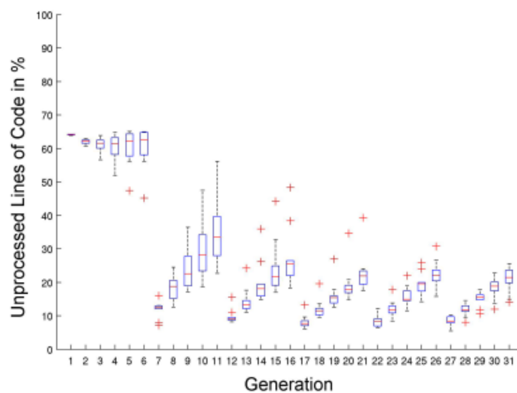
(b) kein Pruning



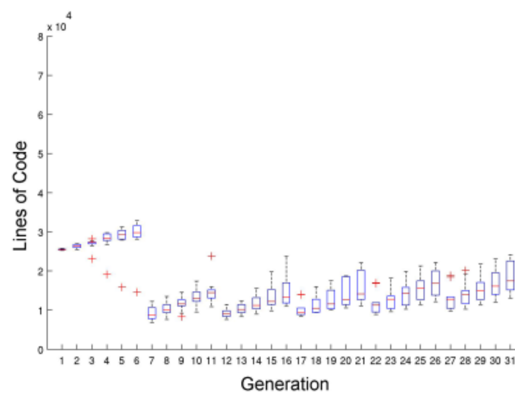
(c) Pruning nach jeder Generation



(d) Pruning nach jeder Generation



(e) Pruning jede 5. Generation



(f) Pruning jede 5. Generation

Abbildung 6.21: Entwicklung des prozentualen Anteils ungenutzter Zeilen in einem Plan (linke Spalte) und der gesamten Anzahl der Zeilen eines Plans bei unterschiedlich oft angewendetem Pruning [33].

werden, um Roboter an interessanten Stellen, wie Bereiche mit wichtigen Ressourcen, im Arbeitsbereich zu sammeln, um dort komplexere Verhalten mit einer höheren Schwarmdichte auszulösen. Da dies ein sehr grundlegendes Schwarmverhalten ist, ist es als Szenario für den nicht konkurrierenden Ansatz gut geeignet.

Fitnessfunktion

Die Fitnessfunktion für die Aggregation weist ebenso wie bei Kollisionsvermeidung jedem Roboter eine Fitness zwischen null (sehr gut) und 1.000 (sehr schlecht) zu. Aus der Fitness für einen Roboter wird dann die Fitness des betrachteten Individuums, das im nicht konkurrierenden Modus dem gesamten Schwarm entspricht, berechnet.

Um den verwendeten einfachen Sensoren gerecht zu werden, wird eine Fitnessfunktion gewählt, die jeder Roboter lokal für sich aus rein lokalem Wissen berechnen kann. Globales Wissen, wie die Position des Roboters, fließen in die Fitnessfunktion nicht mit ein. Dies ist ein wichtiger Unterschied zu üblichen Ansätzen wie in [111] und [30] dargestellt. Dies berücksichtigt die Möglichkeit von späterer online, onboard Evolution, wenn, wie z.B. beim Wanda-Roboter mit Blackfin, ausreichend Speicherplatz und Rechenleistung für die Durchführung zu Verfügung steht.

Gleichung (6.21) beschreibt hierbei die Fitness f_{nb}, die sich aus der Nachbarschaft anderer Roboter ergibt. $d^s_{nb}(j)$ bezeichnet die Entfernung zum nächsten Nachbarn von Roboter $j \in I$, des Individuums I, innerhalb der Reichweite von Sensor s mit einer maximalen Reichweite von d_{max}.

$$\begin{aligned} f_{nb}(j) &= 1 - \frac{1}{6 \cdot d_{max}} \sum_{s=1}^{6} (d_{max} - \overline{d}^s_{nb}(j)) \\ \overline{d}^s_{nb}(j) &= \begin{cases} d^s_{nb}(j) & \text{, wenn } d^s_{nb}(j) < d_{max} \\ d_{max} & \text{, anderenfalls} \end{cases} \end{aligned} \tag{6.21}$$

Der Term $(d_{max} - \overline{d}^s_{nb}(j))$ führt zusammen mit $\overline{d}^s_{nb}(j)$ eine Umrechnung von Entfernung in Signalstärke durch, wie sie auf dem Jasmine- und Wanda-Robotern als Ausgabe der Entfernungsmessung bzw. Kommunikation üblich ist. Die Summe aller Sensoren wird danach mit $\frac{1}{6 \cdot d_{max}}$ normiert. f_{nb} sinkt mit zunehmender, um den Roboter wahrgenommener Schwarmdichte.

Gleichung (6.22) fasst die Fitness der einzelnen Roboter des Individuums I zusammen.

$$f_{total} = \frac{1}{|I|} \sum_{j \in I} f_{nb}(j) \cdot f_{initial} \tag{6.22}$$

Parametrisierung

Die Parametrisierung des Experimentes ist wie bei der Kollisionsvermeidung. Allerdings werden zusätzliche Interrupts und Atome hinzugefügt. Die Interrupts I1NEIGHBOURS, ... ,I6NEIGHBOURS sind gültig, wenn die Anzahl der wahrgenommenen Nachbarn größer gleich 1, 2,. . ., 6 ist. Dabei kann pro Sensor maximal ein Nachbar wahrgenommen werden.

Das zusätzliche Atom ACHOOSE_LEADER bewirkt die zufällige Auswahl eines Roboters in der Nähe als Führenden, AFOLLOW versucht dann, dem Führenden zu folgen, indem der Roboter sich auf den Führenden ausrichtet und sich auf ihn zubewegt. Falls kein Führer ausgewählt werden kann, oder ACHOOSE_LEADER nicht ausgeführt wurde, bewirkt AFOLLOW, dass der Roboter anhält.

Die Arena hat dieselbe Größe wie bei der Evolution der Kollisionsvermeidung, allerdings enthält sie keine Wände. Tabelle 6.8 fasst die wichtigsten Parameter zusammen.

Experimente in der Simulation

Die aus der Evolution der Kollisionsvermeidungsstrategie gewonnenen Ergebnisse werden in die Evolution einer Aggregationsstrategie mit einbezogen. Da sich RBS mit Pruning nach jeder Generation als die beste Variante herausgestellt hat, wird diese mit ihren Parametern übernommen.

Zur Evolution der Aggregation wurden fünf Durchläufe durchgeführt. Jeder Durchlauf benötigte vier bis fünf Tage. Dabei wurden 18 Individuen parallel auf fünf Dual Core Rechnern (Intel® Core™2 CPU 6400 @ 2.13 GHz) und einem Dual Quad-Core Rechner (Intel® Xeon® CPU X5365 @ 3.00 GHz) mit insgesamt 18 Kernen gerechnet.

Abbildung 6.22 stellt den Verlauf der Fitness während des Experiments dar. Bedingt durch die nicht gestufte, lineare Fitnessfunktion und die Mittelung der Fitness über alle Roboter in einem Schwarm, ist die Entwicklung der Fitness glatter als im Kollisionsvermeidungs-Experiment. Eine deutliche Verbesserung der Fitness ist bis zum Ende des zweiten Drittels zu erkennen. Danach flacht die Kurve ab, was die Konvergenz zu einer hinreichend guten Lösung anzeigt.

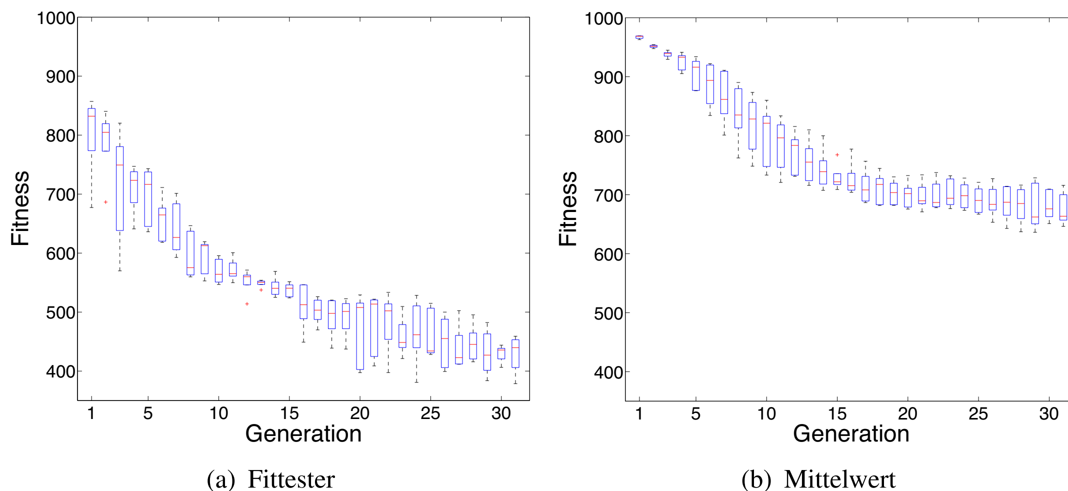
Tabelle 6.8: Parametrisierung der Evolution von Aggregation.

Atome	ASTOP, AMOVE_FWD, AROT_R, AROT_L, AFOLLOW und ACHOOSE_LEADER
Interrupts	ITRUE, ISPACEL, ICRITCOLLISION, I1NEIGHBOURS, I2NEIGHBOURS, I3NEIGHBOURS, I4NEIGHBOURS, I5NEIGHBOURS und I6NEIGHBOURS,
Generationen	31
Simulierte Zeit	500 Zeitschritte
Individuen	300
Roboter pro Arena	30
Arenen pro Generation	300
Selektionsmechanismen	RBS
Crossover Rate	70%
Mutationsrate	30%, davon: 50% Subtree Mutation, 50% Neighbour Mutation
Elitist Strategie	15

Listing 6.5 zeigt den besten Plan eines Versuchs aus Generation 30. Er unterscheidet sich von dem besten Individuum aus Generation 14 in Listing 6 nur im Verhalten in Zeile 6. Hier findet jedoch nur eine Wiederholung des vorherigen Ausweichverhaltens statt. Die bessere Fitness ergibt sich eher zufällig durch eine günstigere Anfangsposition der Roboter. Solche Effekte könnten durch mehrmalige Evaluation des gleichen Individuums egalisiert werden. Da dies aber anscheinend zu keiner signifikanten Beeinflussung der Evolution führt, wird darauf zu Gunsten der eingesparten Rechenzeit verzichtet. Evolution auf der durchschnittlichen Fitness eines Individuums ist im Rahmenwerk vorhanden und benutzbar.

Listing 6.5: Runde: 30, Individuum: 9240, Elternteil A: 8712, Elternteil B: 8956, Fitness: 379

```
1 271657 <PLAN duration="infinite">
2 102973   <MULT multiplicity="infinite">
3 102973     <MULT multiplicity="infinite">
4 102973       <ATOM duration="36" interrupt="NOT(ICRITCOLLISION)"
5                name="AMOVE_FWD"/>
6            </MULT>
7          </MULT>
8 018756   <BEHAVIOUR duration="37" interrupt="NOT(I3NEIGHBOURS)">
```



(a) Fittester

(b) Mittelwert

Abbildung 6.22: Fitnessentwicklung bei der Evolution von Aggregationsverhalten. Die Daten setzen sich aus fünf Versuchen zusammen.

```
 9  018756    <MULT multiplicity="39">
10  013479     <ATOM duration="28" interrupt="ICRITCOLLISION"
11               name="AROT_R"/>
12  005277     <BEHAVIOUR duration="61"
13               interrupt="NOT(I5NEIGHBOURS)">
14  002043      <ATOM duration="infinite"
15                interrupt="NOT(ICRITCOLLISION)"
16                name="AMOVE_FWD"/>
17  003234      <BEHAVIOUR duration="15"
18                interrupt="NOT(I2NEIGHBOURS)">
19  003234       <MULT multiplicity="39">
20  003234        <BEHAVIOUR duration="37"
21                  interrupt="NOT(I3NEIGHBOURS)">
22  003234         <MULT multiplicity="39">
23  002844          <ATOM duration="28" interrupt="ICRITCOLLISION"
24                    name="AROT_R"/>
25  000390          <BEHAVIOUR duration="96"
26                    interrupt="NOT(ISPACEL)">
27  000390           <BEHAVIOUR duration="15"
28                     interrupt="NOT(I2NEIGHBOURS)">
29  000390            <ATOM duration="36"
30                      interrupt="NOT(ICRITCOLLISION)"
31                      name="AMOVE_FWD"/>
32                   </BEHAVIOUR>
33                  </BEHAVIOUR>
34                 </MULT>
35                </BEHAVIOUR>
36               </MULT>
37              </BEHAVIOUR>
```

```
38                </BEHAVIOUR>
39              </MULT>
40            </BEHAVIOUR>
41 149928   <ATOM duration="94" interrupt="I1NEIGHBOURS"
42                name="AFOLLOW"/>
43          </PLAN>
```

Analysiert man das während der Evolution entstandene Aggregationsverhalten aus Listing 6.6, so basiert es auf zwei einfachen Verhalten zur Kollisionsvermeidung, die zusammen zu einer Aggregation führen. Grundlage für beide ist zunächst, dass die Roboter sich vorwärts bewegen, solange es keine kritische Kollision gibt. Dies bewirkt die Multiplizität in den Zeilen 2 – 5. Das erste ist ein Ausweichverhalten. Es wird durch das Verhalten in Zeilen 6 – 15 beschrieben. Es bewirkt, dass sich die Roboter, bei weniger als zwei Robotern in ihrer Umgebung, bei einer kritischen Kollision nach rechts drehen, bzw. falls es keine kritische Kollision gibt, geradeaus fahren. Diese führt dazu, dass die Roboter zum einen ausweichen und zum anderen dichter in schon vorhandene Cluster hineinfahren.

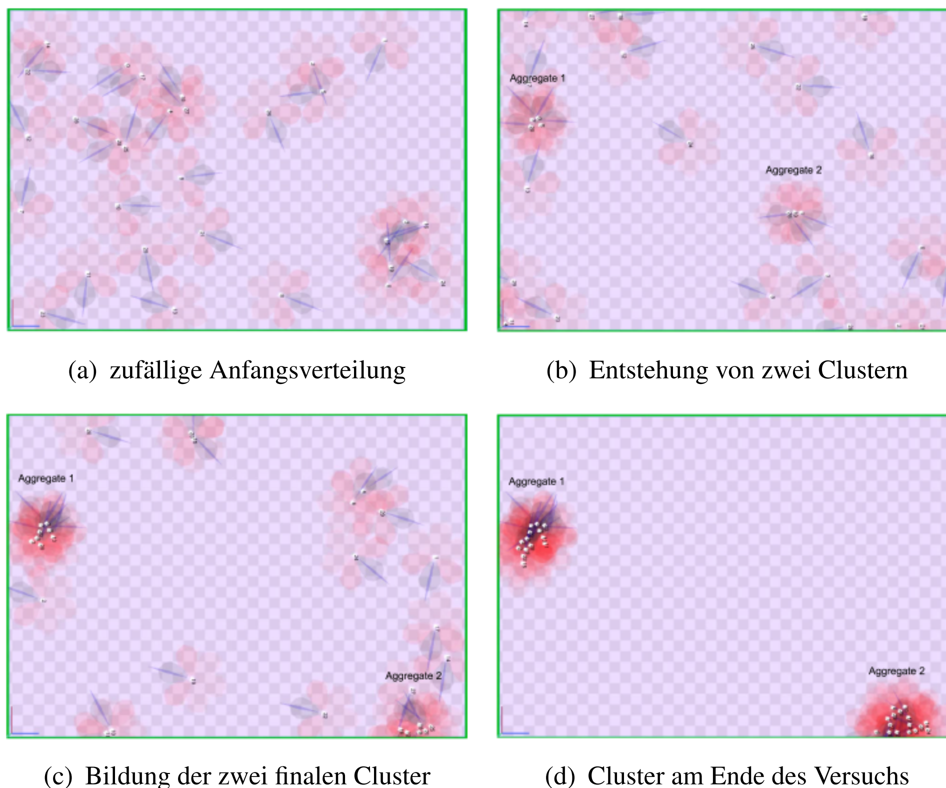
Das zweite Verhalten in Zeile 16 besteht nur aus dem Atom AFOLLOW, welches in dem Fall, dass nicht ACHOOSE_NEIGHBOUR aufgerufen wird, einem einfachen Stop entspricht. Es bewirkt, dass die Roboter für 94 Zeitschritte anhalten, wenn sie mindestens einen Roboter in ihrer Umgebung wahrnehmen. Diese Situation tritt aber wegen des vorgelagerten Bewegungsverhaltens in Zeilen 2 – 5 nur dann auf, wenn eine kritische Kollision vorliegt. Dieses verursacht eine Clusterbildung an Stellen in der Arena, an denen Roboter länger verweilen, wie z.B. in Ecken. Hier ist die Wahrscheinlichkeit groß, dass sich viele Roboter beim Ausweichen begegnen.

Listing 6.6: Runde: 14, Individuum: 4361, Elternteil A: 4191 Elternteil B: 3953, Fitness: 589

```
1  271650 <PLAN duration="infinite">
2  271650   <MULT multiplicity="infinite">
3  109416     <ATOM duration="36" interrupt="NOT(ICRITCOLLISION)"
4                name="AMOVE_FWD"/>
5           </MULT>
6  001957   <BEHAVIOUR duration="37" interrupt="NOT(I3NEIGHBOURS)">
7  001957     <MULT multiplicity="39">
8  016987       <ATOM duration="28" interrupt="ICRITCOLLISION"
9                  name="AROT_R"/>
10 002583       <MULT multiplicity="73">
11 002583         <ATOM duration="36" interrupt="NOT(ICRITCOLLISION)"
12                   name="AMOVE_FWD"/>
```

```
13              </MULT>
14            </MULT>
15          </BEHAVIOUR>
16 142664   <ATOM duration="94" interrupt="I1NEIGHBOURS"
17            name="AFOLLOW"/>
18        </PLAN>
```

Versuche mit diesem Verhalten bestätigen diese Analyse. Abbildung 6.23 zeigt, wie sich die Roboter im Verlauf eines Versuches aggregieren. Nach der Bildung zufälliger Cluster in Abbildung 6.23(a) verstärkt sich Cluster Eins (Aggregate 1), wohingegen Cluster Zwei (Aggregate 2) sich auflöst. Am Ende haben sich zwei Cluster gebildet. Die Bildung mehrerer Cluster ergibt sich durch die Verwendung rein lokaler Information und einem Cluster-Effekt, der nur auf der Nutzung dieser Informationen basiert. Aggregation anhand von Umweltmustern, wie in [99] beschrieben, führt zu größeren Clustern, soweit durch das Umweltmuster vorgegeben. Für die Roboter in unserem Versuch ist es nur bedingt möglich, die Größe der Cluster zu unterscheiden. Individuen in früheren Generationen der Evolution haben jedoch zu deutlich kleineren Clustern geführt.



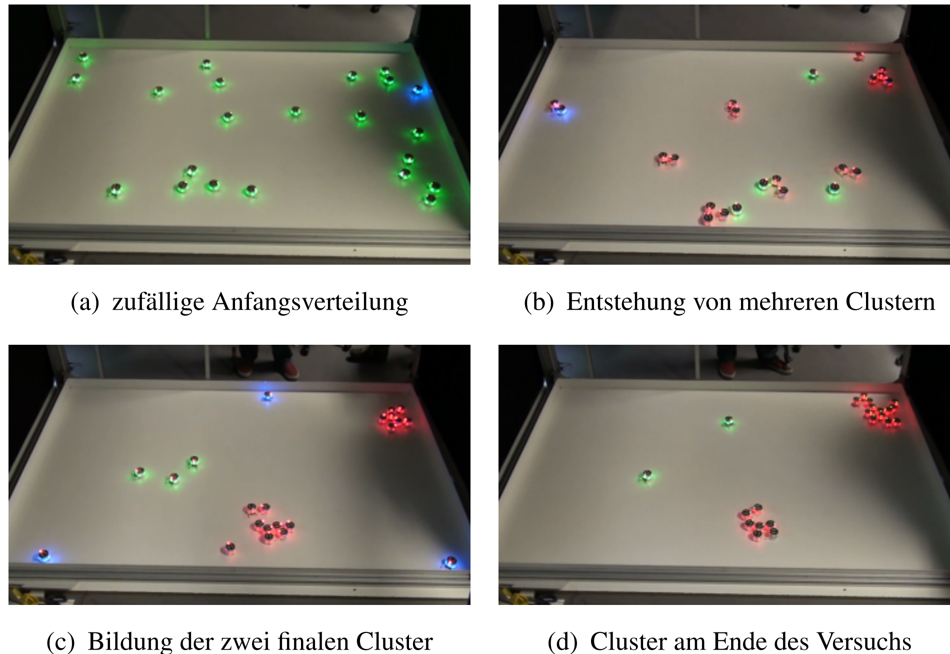
(a) zufällige Anfangsverteilung

(b) Entstehung von zwei Clustern

(c) Bildung der zwei finalen Cluster

(d) Cluster am Ende des Versuchs

Abbildung 6.23: Simulation des Verhaltens des besten Individuums [33].



(a) zufällige Anfangsverteilung (b) Entstehung von mehreren Clustern

(c) Bildung der zwei finalen Cluster (d) Cluster am Ende des Versuchs

Abbildung 6.24: Auf Wanda-Roboter übertragenes Verhaltens des besten Individuums. Dabei sind blaue Roboter in Zeile 2, grüne in Zeile 6 und rote Roboter in Zeile 16 des Plans aus Listing 6.6.

6.2.3 Experimente mit Wanda

Zur Evaluation der Aggregationsstrategie wurden Versuche mit dem Wanda-Roboter durchgeführt. Diese Versuche veranschaulichen mehrere Konzepte, des in dieser Arbeit entwickelten Systems. Das Experiment zeigt die Übertragbarkeit der durch GP erzeugten MDL2ϵ-Pläne auf echte Roboter, sowie die Durchführung automatisierter Versuche in der Arena.

Für das Experiment wird der Plan aus Listing 6.6 auf den Wanda-Roboter übertragen. Da während der Evolution Pläne für einen Jasmine-Roboter erzeugt wurden, werden die verwendeten Atome und Interrupts, sowie die Größe der Arena jeweils mit einem Faktor $f_{WJ} = 3{,}5/5 = 0{,}7$ skaliert, was eine Skalierung auf das Verhältnis der Roboterlängen entspricht. Hiernach wird der synthetisierte Plan eins-zu-eins auf den Roboter übertragen.

Versuchsaufbau

Die Durchführung der Versuche geht in folgenden Schritten:

1. Platzieren der Roboter in der Arena anhand eines projizierten Rasters

2. Roboter speichern ihre aktuelle Position (ODeM-Positionierungssystems)

3. Durchführen des Versuchs mit dem Verhalten aus Listing 6.6 (MDL2ϵ/ GP)

4. Roboter kehren zu ihrer gespeicherten Position zurück (ODeM-Positionierungssystems)

5. System beginnt automatisch wieder bei 3. bis die gewünschte Anzahl an Versuchen durchgeführt ist (Arenasteuerung/ZigBee)

Im Idealfall platziert der Experimentator nur die Roboter und aktiviert Punkte 2. und 3. von der Arenasteuerung aus. In einigen wenigen Fällen mussten Versuche von der Arenasteuerung aus neu angestossen werden.

Während des Versuchs werden über ZigBee die Positionsdaten der einzelnen Roboter und die Intensität der von anderen Robotern empfangenen Nachrichten zur Abstandsmessung aufgezeichnet.

Um eine quantitative Analyse für das Verhalten der Roboter zwischen Simulation und Realität durchzuführen, werden zwanzig Versuchen á zehn Minuten (plus fünf Minuten Repositionierung) mit den Wanda-Robotern und hundert Versuche in der Simulation durchgeführt.

Analyse

Abbildung 6.24 zeigt die Entstehung von Clustern während des Experimentes mit Wanda-Robotern, wie sie auch in Abbildung 6.23 während der Simulation zu sehen ist. Zur besseren Darstellung, der in der obigen Analyse der Aggregationsstrategie aufgezählten drei Teilverhalten, werden Atome zur Aktivierung der Farben in den Plan eingefügt. Es ist sehr gut zu erkennen, dass sich die Roboter entsprechend der zuvor durchgeführten Analyse des Plans verhalten.

Diese Beobachtung wird durch die Diagramme in Abbildung 6.25 quantitativ bestätigt. Abbildung 6.25(a) und (b) zeigen die Entwicklung der mittleren Anzahl an Clustern während der Experimente mit Wanda und mit der Simulation. Eine Abnahme der Clusteranzahl auf zehn Cluster ist bei beiden bis zu Minute 2 – 3 des Experiments zu erkennen. Danach bleibt die Anzahl der Cluster bei Experimenten mit Wanda in diesem Bereich. In der Simulation sinkt die Anzahl der Cluster weiter bis zur 6. Minute auf im Mittel fünf Cluster und bleibt dann konstant. Dies wird dadurch bewirkt, dass

die Roboter in der Simulation sich viel seltener aus großen, schon bestehenden Clustern lösen. Nach sechs Minuten sind in der Simulation zumeist alle Roboter in Clustern gefangen. Die realen Roboter können sich jedoch mit höherer Wahrscheinlichkeit aus entstandenen Clustern wieder lösen. Dies wird durch das Sensorrauschen bewirkt, was bei den Robotern zufällig das Ausweichverhalten auslöst. Sensorrauschen wird in der Jasmine-Simulation nicht modelliert.

Betrachtet man Abbildungen 6.25(c) und 6.25(d) so bestätigen sie diese Annahmen. Die Abbildungen zeigen die mittlere Anzahl an Robotern (Größe eines Bereichs gleicher Farbe auf der Y-Achse), die in einem Cluster der Größe N (Farbdarstellung) zum Zeitpunkt t (X-Achse) gebunden sind. Zu Beginn des Versuchs sind die meisten Roboter in Clustern der Größe eins (dunkel blau). Diese Roboter sind frei beweglich und gruppieren sich dann im Verlauf des Experimentes zu immer größeren Clustern. In der Simulation werden Cluster mit bis zu 15 Robotern erreicht. Ca. 2/3 der Cluster haben eine Größe von mehr als fünf Robotern. Im Versuch mit Wanda erreichen nur 1/3 der Cluster eine Größe von mehr als fünf Robotern. Die größten erreichten Cluster bestehen aus maximal elf Robotern.

Dies ist nicht nur auf die größere Tendenz zur Auflösung der Cluster zurückzuführen, sondern auch durch die unterschiedliche Konstruktion der Roboter bedingt. Die Wanda-Roboter verhaken sich während des Versuchs mit ihren Schiebern zu Clustern von zwei bis drei Robotern, die sich nur sehr schwer auflösen, obwohl sich die Roboter im Ausweichverhalten befinden. Dies führt zu einer deutlich höheren Anzahl an kleinen und damit weniger großen Clustern, da diese Roboter in den großen Clustern fehlen. Hier wäre zur Lösung des Problems eine manuelle Anpassung der Ausweichstrategie durch die Hinzunahme von Rückwärtsfahren möglich.

Abbildung 6.25(f) zeigt die aus den aufgezeichneten Intensitätswerten der Kommunikation über Gleichungen (6.21) und (6.22) mit $d_{max} = 33$ zurückgerechnete Fitness der Wanda-Roboter. Sie lässt sich eigentlich nicht direkt mit den Werten der simulierten Fitness aus Abbildung 6.25(e) vergleichen, da hier die Modellierung der Jasmine-Sensoren zu Grunde liegt, welche sich in einigen Eigenschaften von denen des Wanda unterscheiden. Trotz dieses systematischen Unterschieds ist die Ähnlichkeit der erreichten Fitnesswerte bei ähnlichen Clustergrößen sehr stark. Beide Kurven stimmen mit den jeweiligen Abbildungen 6.25(a) und (b) in dem zeitlichen Verlauf grob überein. In beiden Experimenten kann man eine leichte Verdichtung der Cluster zum Ende der Versuche hin erkennen, wobei die Anzahl der

Cluster gleich bleibt.

Die starke Ähnlichkeit der Verläufe zwischen Simulation und Realität spricht für die Durchführung von GP direkt auf den Robotern. Dies bestätigt die Annahme, dass die lokale Fitnessfunktion ohne Änderungen für online on-/offboard GP eingesetzt werden kann.

6.2.4 Bewertung

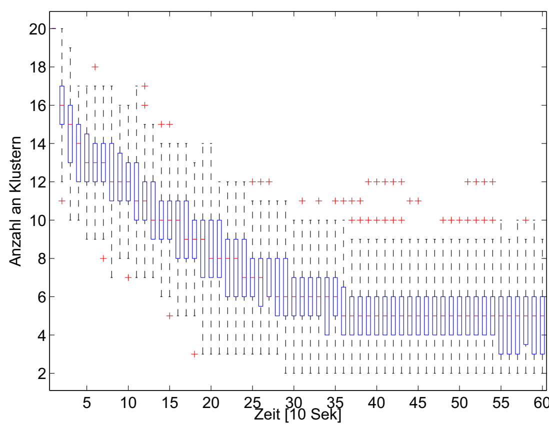
Die mit dem FLSO durchgeführten Experimente und deren Übertragung auf echte Roboter belegen die in Abschnitt 1.2.2 aufgestellte These, dass die Methoden der Genetischen Programmierung, angewendet auf die Steuersprache MDL2ϵ, für die halbautomatische Erzeugung von Schwarmrobotersteuerungen verwendet werden kann, und dabei gut interpretierbare Lösungen entstehen.

Die mit Wanda durchgeführten Experimente zeigen eine gute Generalisierbarkeit der gefundenen Lösungen, was in der Analyse der gesammelten Daten durch ein ähnliches Verhalten der simulierten und echten Roboter gezeigt worden ist.

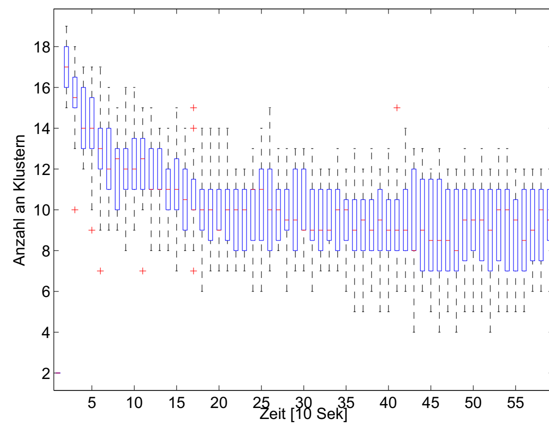
In der Analyse der Experimente werden Wege aufgezeigt, wie Fitnessfunktionen aufgestellt, analysiert und verbessert werden können. So zeigt die Analyse der Fitnesslandschaft in Abschnitt 6.2.1, dass eine Optimierung der Fitnessfunktion möglich ist. In der Anwendung sollte aber immer der Aufwand gegen den Nutzen abgewogen werden. Eine optimierte Fitnessfunktion kann zwar die Evolution beschleunigen, dies kann aber mit geringerem Aufwand auch durch zusätzliche parallele Rechenzeit erreicht werden.

Das in der vorliegenden Arbeit entwickelte Konzept einer Entwicklungsumgebung für Schwarmroboter bestehend aus einer Steuersprache mit Betriebssystem und Interpreter, einer interaktiven Arena und dem Framework for Learning and Self-Organisation wurde während des letzten Experiments durchgeführt und umgesetzt.

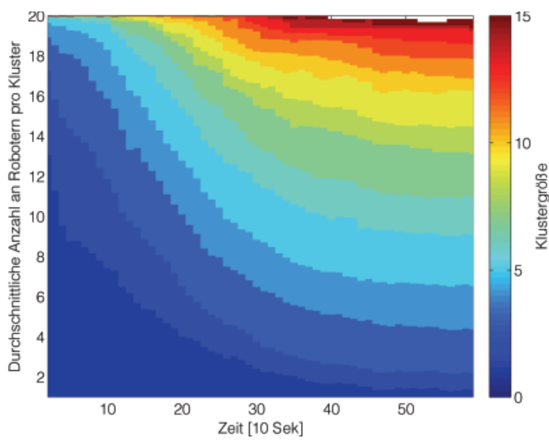
Die Nutzung der durch die Arena und das ODeM bereitgestellen Funktionalitäten für die automatische Durchführung von Schwarmroboterexperimenten ermöglicht einen schnellen, akuraten und wiederholbaren Ablauf der Experimente auf den realen Robotern. Alle für die Auswertung des Experiments benötigten Daten können ohne Nachbearbeitung mit dem entwickelten System während der Laufzeit abgefragt werden. Der Experimenta-



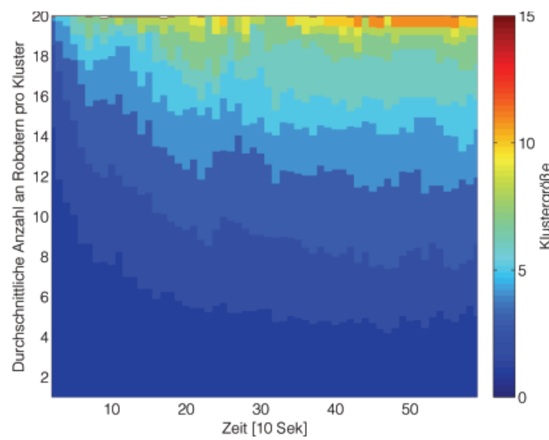
(a) Mittlere Clusteranzahl



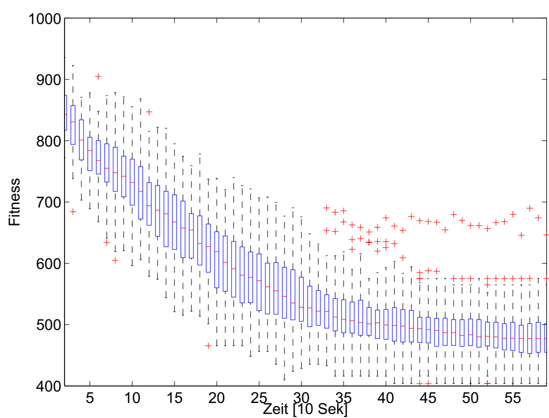
(b) Mittlere Clusteranzahl



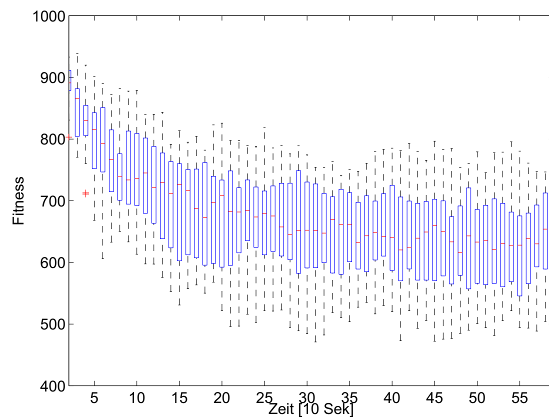
(c) Mittlere Anzahl an Robotern in einem Cluster der Größe N



(d) Mittlere Anzahl an Robotern in einem Cluster der Größe N



(e) Mittlere Fitness des Schwarms



(f) Mittlere Fitness des Schwarms

Abbildung 6.25: Entstehung von Clustern über den Verlauf eines Experimentes. Links Daten aus 100 Simulationen; Rechts Daten aus 20 automatisierten Versuchen mit dem Wanda-Roboter.

tor muss nur zum Aufstellen und Laden der Roboter im Labor sein. Danach können die Experimente von einem anderen Ort aus fernüberwacht werden.

Bei der Vorbereitung der Experimente kamen die Vorteile des Konzepts einer plattformunabhängigen Steuersprache basierend auf Atomen, Interrupts und Variablen, sowie eines platformunabhängigen Interpreters voll zur Geltung. Die Anpassungen, die beim Übergang von dem simulierten auf die realen Roboter, gemacht werden mussten, beschränkten sich auf die Justierung und Implementierung der in der Evolution verwendeten Interrupts und Atome. Es mussten keine Veränderungen an dem synthetisierten MDL**2**ϵ-Plan durchgeführt werden.

6.3 Zusammenfassung

Zur Evaluation der Entwicklungsumgebung für Roboterschwärme wurde eine Vielzahl von Experimenten in Simulationsumgebungen sowie auf echten Robotern durchgeführt. In diesem Kapitel wurden die Experimente und die aus den Experimenten abgeleiteten Ergebnisse beschrieben.

Im ersten Abschnitt des Kapitels wurde auf Experimente eingegangen, die eine evaluation der Arena und der Entwicklungsumgebung ermöglichen. Es wurde ein Experiment zur Stigmergie beschrieben, bei dem die Roboter mit Hilfe eines optischen, virtuellen Pheromons ihren Durchsatz beim Sammeln von Ressourcen optimieren. Das Experiment zeigt die vollen Fähigkeiten der entwickelten dynamischen, interaktiven Arena. Das Experiment wurde sowohl auf dem Jasmine- als auch auf Wanda-Robotern durchgeführt.

Ein zweites Experiment im ersten Abschnitt betrachtet die energetische Effizienz von Roboterschwärmen beim Sammeln von Ressourcen. Hier wurden grundlegende Formeln und Modelle für die Optimierung von Lade- und Arbeitsphasen in Roboterschwärmen hergeleitet. Die Modelle ergeben sich aus einer Kombination von Versuchen in der Simulation als auch auf echten Robotern sowie theoretischen Überlegungen.

Das Experiment zeigt die Übertragbarkeit von in der Simulation durch einen Designer entwickeltem Verhalten auf echte Roboter. Die durch MDL**2**ϵ erreichte Abstraktion von der Hardware stellt sich dabei als sehr erfolgreiches Konzept heraus.

Im zweiten Abschnitt wurden zwei Experimente mit dem Framework for Learning and Self-Organisation beschrieben. In dem ersten Experiment, welches nur in der Simulation vom Jasmine-Roboter durchgeführt wurde, wur-

den die verschiedenen Selektionsstrategien und das Pruningverfahren anhand der erreichten Fitness und der Diversität evaluiert. Bei dem Experiment handelt es sich um die Synthese eines einfachen Kollisionsvermeidungs-Verhaltens. Die hieraus erhaltenen Parameter wurden in das zweite Experiment übernommen.

Im Zweiten Experiment wurde ein grundlegendes Schwarmverhalten, die Aggregation, evolviert. Anhand des Verhaltens wurde gezeigt, wie leicht sich automatisch erzeugte MDL2ϵ-Pläne interpretieren lassen. Dies unterstreicht die in der vorliegenden Arbeit vorgenommene Abgrenzung von Evolution auf KNNs.

Das in der Simulation von Jasmine-Robotern synthetisierte Verhalten wurde dann, ohne Veränderung des MDL2ϵ-Plans, auf Wanda-Roboter übertragen. Ein Vergleich des Verhaltens der Roboter in der Simulation und der echten Wanda-Roboter beendet den Abschnitt. Die Experimente mit Wanda zeigen die Qualitäten der Arena zur automatischen Aufnahme von Daten und der Wiederholbarkeit von Experimenten.

7. Zusammenfassung und Ausblick

Sehr große Schwärme von Mikrorobotern waren bisher ein nicht betrachteter Teil der Schwarmrobotik. Schwärme von drei bis zehn Robotern waren der Standard. Durch Arbeiten in diesem Bereich, die die Entwicklung eines Schwarms von bis zu 1.000 $3 \times 3 \times 3mm^3$ großen Roboter zum Ziel hatten, wurde ersichtlich, dass die Grundlagen zur Erforschung solch großer Systeme, aber auch schon kleinerer Schwärme ab 20 Roboter, noch nicht vorhanden sind. Bestehende Ansätze lassen sich nicht auf die sehr eingeschränkten Fähigkeiten der Mikroroboter und der sehr großen Anzahl hiervon übertragen. Es wurde ersichtlich, dass schon alleine der Umgang mit solchen Systemen nicht hinreichend erforscht ist.

Ziel dieser Arbeit war daher, die Handhabung zur Erforschung und Entwicklung solcher Systeme zu vereinfachen und Wege aufzuzeigen, wie sich die Entwicklung von Algorithmen teilweise automatisieren lässt. Das in dieser Arbeit entwickelte System zeigt daher einen vollständigen Ansatz zur Erforschung und Entwicklung großer Mikro- und Miniaturroboterschwärme und dessen Übertragbarkeit auf verschieden Roboterplattformen.

In der Arbeit wurden daher zunächst die Anforderungen aus Sicht der verschiedenen Benutzer und Robotersysteme analysiert und beschrieben. Die Liste der Anforderungen wurde in Teilziele übertragen. Die Arbeit teilt sich daher in zwei Bereiche, die Entwicklungsumgebung für Schwarmrobo-

tik und das *Framework for Learning and Self-Organisation (FLSO)*.

Innerhalb der Entwicklungsumgebung für Schwarmrobotik wurde anhand der Anforderungsanalyse die Steuersprache *extended Motion Description Language 2 (*MDL2ϵ*)* definiert und formal beschrieben. Verschiedene Ansätze zur Umsetzung der Steuersprache wurden implementiert und untersucht. Dabei wurde auch die Einbindung der Steuersprache in die verschiedenen Hardwarestrukturen und in die speziell entwickelten Betriebssysteme der drei verwendeten Robotersysteme betrachtet. Dies war notwendig da die Roboter unterschiedliche Arten der Informationsakquise und -bearbeitung aufweisen.

Eine automatisierte Experimentierarena ist Teil der Entwicklungsumgebung. Sie erweitert zum einen die Fähigkeiten der Roboter und ermöglicht zum anderen einen wiederholbaren automatisierten Ablauf von Experimenten, was zur wissenschaftlichen Auswertung sehr entscheidend ist.

Das entwickelte FLSO basierend auf Genetischer Programmierung zeigt eine Möglichkeit zur halb-automatischen Erzeugung von Schwarmrobotersteuerungen auf. Es bietet somit ein wichtiges Werkzeug für Entwickler und Forscher. Innerhalb des FLSO wird beschrieben, wie sich *Genetische Programmierung (GP)* auf MDL2ϵ umsetzen lässt. Mit einem innerhalb der Arbeit entwickelten, neuen Diversitätsmaß lassen sich Analysen des Verlaufs der Evolution durchführen und Parameter optimieren. Es wird auch ein Pruning-Verfahren beschrieben, welches für MDL2ϵ optimiert wurde und sich in die formale Beschreibung der Sprache mit eingliedert.

Am Ende der Arbeit wurden verschieden Experimente beschrieben, die einen wissenschaftlichen Beitrag zur Schwarmrobotik geben, und anhand derer die einzelnen Teilziele des gesamten Systems beleuchtet und bewertet wurden.

7.1 Ergebnis und Beitrag der Arbeit

Die Arbeit liefert eine wichtigen Beitrag zur Erforschung und Entwicklung von emergenten Schwarmverhalten. Innerhalb der Arbeit wurde ein vollständiges System, von der Benutzerschnittstelle bis hin zur Implementierung und dem Design einzelner Schwarmrobotern, entwickelt. Die Entwicklungsumgebung für Schwarmrobotik mit ihrer angepassten Sprache für die Steuerung von Schwarmroboter und der automatisierten Arena für Schwarmroboterexperimente, wird schon jetzt von anderen Institutionen, die in den

Bereichen der Schwarmrobotik und der Kollektiven Robotik for-schen, übernommen.

Eine Reihe von Experimenten liefert wissenschaftliche Beiträge zur Forschung in der Schwarmrobotik. Die wissenschaftlichen Ergebnisse reichen von der Evolution von emergenten Verhalten mit Methoden der GP und deren Übertragung auf echte Roboterschwärme, über die Betrachtung eines neuartigen Verfahrens zur Koordination von Roboterschwärmen basierend auf ortsgebundenen virtuellen Pheromonen, bis hin zur energetischen Betrachtung von Roboterschwärmen. Dies unterstreicht die hohe Flexibilität und Anwendbarkeit des vorgestellten Systems.

Es konnte gezeigt werden, dass eine Entwicklungsumgebung für Roboterschwärme die Entwicklung beschleunigt und unabhängiger von Ergebnissen macht, die durch Simulationen gewonnen werden. Ferner wurde gezeigt, dass Methoden der GP eingesetzt werden können um Teile einer Schwarmrobotersteurung halbautomatisch zu erzeugen und eine einfachere Analyse der gefundenen Lösung im Gegensatz zu *Evolutionary Computation (EC)* basierend auf KNNs ermöglichen.

7.2 Ausblick

Die Arbeit gibt ein wichtiges Werkzeug für weiterführende Forschung in der Kollektiven Robotik und Schwarmrobotik. Sie ermöglicht eine schnelle Evaluierung von Algorithmen auf echten Robotern und kann in vielen weiteren Forschungsprojekten zielführend eingesetzt werden.

Weitere wissenschaftliche Fragen, die sich direkt aus den durchgeführten Experimenten ergeben, und die sich anhand des in der Arbeit beschriebenen Systems untersuchen lassen, sind:

- *Wie müssen Substanzen und Sensoren für diese Substanzen beschaffen sein, um sie zur Koordination von großen Roboterschwärmen in einer unstrukturierten Umgebung einzusetzen?*

- *Wie kann ein Schwarm energieeffizient Ressourcen in schwer zugänglichen Gebieten abbauen?*

- *Wie ist dabei seine Effizienz in Bezug auf Zuverlässigkeit, Kosten und Ertrag zu herkömmlichen Ansätzen?*

- *Welche technischen Mittel, die die in der Natur vorhandenen Mittel übersteigen (wie z.B. drahtlose Kommunikation), können eingesetzt werden, um die Effizienz eines künstlichen Schwarms zu erhöhen, ohne dabei auf Fähigkeiten wie Robustheit und Skalierbarkeit zu verzichten?*

Das in der Arbeit entwickelte System für GP auf MDL2ϵ wird in weiteren Forschungsprojekten, wie dem Symbrion und Replicator-Projekt, weitergeführt. Es ist ohne viel Aufwand, aufgrund seiner geringen Größe und seines plattformunabhängigen Designs, möglich, das gesamte System auf ein eingebettetes Linux, z.B. uClinux, zu übertragen. Dann kann on-/offline onboard GP auf den Robotern durchgeführt werden. Im Besonderen bietet es sich für diese Projekte an, da sich die Parallelisierung von GP, wie in der Arbeit beschrieben, gut mit dem Konzept von einem Organismus mit vielen vorhandenen CPUs in Einklang bringen lässt. Es wurde schon in [90] gezeigt, dass MDL2ϵ für die Steuerung von selbstkonfigurierenden Robotern verwendet werden kann.

Ferner können mit dem beschrieben System komplexere Schwarmverhalten erzeugt werden. Es ist dann interessant, wie sich unabhängig von einem Designer die Entwicklung von emergenten Verhalten vollzieht. Solche Verhalten sind nur durch die Fitnessfunktion und das vorgegeben Alphabet beschränkt. Die Analyse der entstanden Verhalten kann wissenschaftlich interessante Einblicke in die Grundlagen von emergentem Verhalten zulassen.

A. MDL2ϵ XML Beschreibung in DTD

```
1  <?xml version="1.0" encoding="UTF8"?>
2
3  <!ELEMENT MDLeScript (PLAN)>
4  <!ELEMENT PLAN ((ATOM|SATOM|BEHAVIOUR|SBEHAVIOUR|UNION|RUNION|
      MULT)*)>
5  <!ATTLIST PLAN
6  duration CDATA "infinite"
7  name CDATA "plan"
8  >
9
10 <!ELEMENT UNION ((ATOM|SATOM|BEHAVIOUR|SBEHAVIOUR|UNION|RUNION|
      MULT)*)>
11 <!ATTLIST UNION
12 name CDATA "union"
13 probability CDATA "1.0"
14 >
15
16 <!ELEMENT RUNION ((ATOM|SATOM|BEHAVIOUR|SBEHAVIOUR|UNION|RUNION
      |MULT)*)>
17 <!ATTLIST RUNION
18 name CDATA "runion"
19 probability CDATA "1.0"
20 >
```

```
21
22 <!ELEMENT ATOM EMPTY>
23 <!ATTLIST ATOM
24 duration CDATA "1.0"
25 name CDATA #REQUIRED
26 arg0 CDATA #IMPLIED
27 arg1 CDATA #IMPLIED
28 arg2 CDATA #IMPLIED
29 arg3 CDATA #IMPLIED
30 arg4 CDATA #IMPLIED
31 arg5 CDATA #IMPLIED
32 arg6 CDATA #IMPLIED
33 arg7 CDATA #IMPLIED
34 arg8 CDATA #IMPLIED
35 arg9 CDATA #IMPLIED
36 arg10 CDATA #IMPLIED
37 arg11 CDATA #IMPLIED
38 arg12 CDATA #IMPLIED
39 arg13 CDATA #IMPLIED
40 arg14 CDATA #IMPLIED
41 arg15 CDATA #IMPLIED
42 arg16 CDATA #IMPLIED
43 arg17 CDATA #IMPLIED
44 arg18 CDATA #IMPLIED
45 arg19 CDATA #IMPLIED
46 probability CDATA "1.0"
47 interrupt CDATA "ITRUE"
48 >
49
50 <!ELEMENT SATOM EMPTY>
51 <!ATTLIST SATOM
52 duration CDATA "1.0"
53 name CDATA #REQUIRED
54 arg0 CDATA #IMPLIED
55 arg1 CDATA #IMPLIED
56 arg2 CDATA #IMPLIED
57 arg3 CDATA #IMPLIED
58 arg4 CDATA #IMPLIED
59 arg5 CDATA #IMPLIED
60 arg6 CDATA #IMPLIED
61 arg7 CDATA #IMPLIED
62 arg8 CDATA #IMPLIED
63 arg9 CDATA #IMPLIED
64 arg10 CDATA #IMPLIED
65 arg11 CDATA #IMPLIED
66 arg12 CDATA #IMPLIED
```

```
67 arg13 CDATA #IMPLIED
68 arg14 CDATA #IMPLIED
69 arg15 CDATA #IMPLIED
70 arg16 CDATA #IMPLIED
71 arg17 CDATA #IMPLIED
72 arg18 CDATA #IMPLIED
73 arg19 CDATA #IMPLIED
74 probability CDATA "1.0"
75 interrupt CDATA "ITRUE"
76 >
77
78 <!ELEMENT BEHAVIOUR ((ATOM|SATOM|BEHAVIOUR|SBEHAVIOUR|UNION|
     RUNION|MULT)*)>
79 <!ATTLIST BEHAVIOUR
80 duration CDATA "infinite"
81 name CDATA #REQUIRED
82 probability CDATA "1.0"
83 interrupt CDATA "ITRUE"
84 >
85
86 <!ELEMENT SBEHAVIOUR ((ATOM|SATOM|BEHAVIOUR|SBEHAVIOUR|UNION|
     RUNION|MULT)*)>
87 <!ATTLIST SBEHAVIOUR
88 duration CDATA "infinite"
89 name CDATA #REQUIRED
90 probability CDATA "1.0"
91 interrupt CDATA "ITRUE"
92 >
93
94 <!ELEMENT MULT ((ATOM|SATOM|BEHAVIOUR|SBEHAVIOUR|UNION|RUNION|
     MULT)*)>
95 <!ATTLIST MULT
96 name CDATA "multiplicity"
97 probability CDATA "1.0"
98 multiplicity CDATA "infinite"
99 >
```

B. MDL2ϵ-Pläne zu den durchgeführten Experimenten

B.1 Stigmergie Experiment

B.1.1 Zufällige Suche von Ressourcen

Listing B.1: MDL2ϵ-Plan für das Hop-Count-Experiment.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2
3  <!DOCTYPE MDLeScript SYSTEM "MDLeScriptSchema.dtd">
4
5  <MDLeScript>
6    <PLAN name="PLAN" duration="infinite">
7
8      <ATOM name="ASETSBLED" arg0="2" arg1="1" duration="0"/>
9      <ATOM name="ASTOP" interrupt="NOT(IODEMPOS)" duration="infinite"/>
10     <ATOM name="ASETPOI" arg0="VOPOSX" arg1="VOPOSY" arg2="0" duration="
          0"/>
11     <ATOM name="ASETSBLED" arg0="2" arg1="0" duration="0"/>
12     <ATOM name="ASETSBLED" arg0="3" arg1="0" duration="0"/>
13
14     <!--
```

```
15      VACCUM0 holds all temporary calculations
16      VACCUM1 holds the direction of the last rotation
17      VACCUM2 holds the number of collisions
18      VACCUM3 holds the current state: 0 – random, 1 – pheromones, 2 – from food1,
          3 – from food2
19      VACCUM4 holds the pheromone threshold
20      VACCUM5 holds the IR threshold
21      -->
22      <ATOM name="ASTORE" arg0="40" arg1="4" duration="0"/>
23      <ATOM name="ASTORE" arg0="20" arg1="5" duration="0"/>
24
25      <MULT>
26        <BEHAVIOUR name="foraging" interrupt="AND(NOT(GEQ(VACCUM3,2)),NOT
            (OR(OR(EQ(VODATAL,21845),EQ(VODATAR,21845)),OR(EQ(VODATAL
            ,29127),EQ(VODATAR,29127)))))">
27          <MULT>
28
29            <SBEHAVIOUR name="avoid␣front" interrupt="GT(VIR0,VACCUM5)">
30              <ATOM name="AADD" arg0="VACCUM2" arg1="1" arg2="2" interrupt="
                  GT(VIRMSGCNT0,0)" duration="0"/>
31              <UNION>
32                <ATOM name="AROTL" interrupt="ISPACEL" arg0="50" duration="10
                    "/>
33                <ATOM name="AROTR" interrupt="NOT(ISPACEL)" arg0="50" duration
                    ="10"/>
34              </UNION>
35              <UNION>
36                <ATOM name="AMOVE" arg0="50" interrupt="NOT(GT(VIR0,VACCUM5)
                    )" duration="10"/>
37                <ATOM name="AMOVE" arg0="-50" interrupt="AND(GT(VIR0,40),NOT
                    (IBACK))" duration="10"/>
38              </UNION>
39            </SBEHAVIOUR>
40
41            <!-- wait for data and calculate absolute difference -->
42            <ATOM name="ASTOP" duration="5"/>
43            <ATOM name="ASTOP" interrupt="NOT(IODEMGREY)" duration="10"/>
44
45            <UNION>
46              <SBEHAVIOUR name="randomWalk" interrupt="NOT(GT(VIR0,VACCUM5
                  ))">
47                <ATOM name="ASTORE" arg0="0" arg1="3" duration="0"/>
48                <BEHAVIOUR name="walk" interrupt="NOT(GT(VIR0,VACCUM5))">
49                  <RUNION>
50                    <ATOM name="AMOVE" arg0="50" duration="10" probability="70"
                        />
```

```
51                  <BEHAVIOUR name="turn␣right" probability="15">
52                    <ATOM name="AROTDEG" arg0="-60" duration="0"/>
53                    <ATOM name="ANOP" interrupt="IMOTION" duration="10"/>
54                    <ATOM name="ASTOP" duration="0"/>
55                  </BEHAVIOUR>
56                  <BEHAVIOUR name="turn␣left" probability="15">
57                    <ATOM name="AROTDEG" arg0="60" duration="0"/>
58                    <ATOM name="ANOP" interrupt="IMOTION" duration="10"/>
59                    <ATOM name="ASTOP" duration="0"/>
60                  </BEHAVIOUR>
61                </RUNION>
62              </BEHAVIOUR>
63            </SBEHAVIOUR>
64          </UNION>
65        </MULT>
66      </BEHAVIOUR>
67
68      <SBEHAVIOUR name="got␣food" interrupt="AND(NOT(GEQ(VACCUM3,2)),
          OR(OR(EQ(VODATAL,21845),EQ(VODATAR,21845)),OR(EQ(VODATAL
          ,29127),EQ(VODATAR,29127))))">
69        <UNION>
70          <ATOM name="ASTORE" arg0="2" arg1="3" interrupt="OR(EQ(VODATAL
              ,21845),EQ(VODATAR,21845))" duration="0"/>
71          <ATOM name="ASTORE" arg0="3" arg1="3" interrupt="OR(EQ(VODATAL
              ,29127),EQ(VODATAR,29127))" duration="0"/>
72        </UNION>
73        <ATOM name="ASENDRADIO" arg0="104" arg1="VODATAL" arg2="VODATAR
            " duration="0"/>
74        <ATOM name="ASETSBLED" arg0="3" arg1="1" duration="0"/>
75        <ATOM name="AMOVE" arg0="50" interrupt="NOT(GT(VIR0,VACCUM5))"
            duration="10"/>
76      </SBEHAVIOUR>
77
78      <BEHAVIOUR name="homing" interrupt="AND(GEQ(VACCUM3,2),NOT(OR(EQ(
          VODATAL,13107),EQ(VODATAR,13107))))">
79        <MULT>
80          <ATOM name="ASTOP" duration="5"/>
81          <ATOM name="ASTOP" interrupt="NOT(IPOI)" duration="10"/>
82          <ATOM name="AROTDEG" arg0="VPOIDIR0" interrupt="IPOI" duration="0
              "/>
83          <ATOM name="ANOP" interrupt="IMOTION" duration="1000"/>
84          <ATOM name="ASTOP" duration="0"/>
85          <ATOM name="AMOVE" arg0="50" interrupt="NOT(GT(VIR0,VACCUM5))"
              duration="10"/>
86          <ATOM name="ASTOP" interrupt="GT(VIR0,VACCUM5)" duration="10"/>
87          <SBEHAVIOUR name="avoid␣and␣move" interrupt="GT(VIR0,VACCUM5)
```

```
          ">
88          <BEHAVIOUR name="avoid␣front" interrupt="GT(VIR0,VACCUM5)"/>
89          <ATOM name="AMOVE" arg0="50" interrupt="NOT(GT(VIR0,VACCUM5))
              " duration="5"/>
90        </SBEHAVIOUR>
91      </MULT>
92    </BEHAVIOUR>
93
94    <SBEHAVIOUR name="home␣sweet␣home" interrupt="AND(GEQ(VACCUM3,2)
        ,OR(EQ(VODATAL,13107),EQ(VODATAR,13107)))">
95      <ATOM name="ASENDRADIO" arg0="104" arg1="VODATAL" arg2="VODATAR
          " duration="0"/>
96      <ATOM name="ASETSBLED" arg0="3" arg1="0" duration="0"/>
97      <RUNION>
98        <ATOM name="AROTDEG" arg0="-180" duration="0"/>
99        <ATOM name="AROTDEG" arg0="180" duration="0"/>
100     </RUNION>
101     <ATOM name="ANOP" interrupt="IMOTION" duration="10"/>
102     <ATOM name="ASTORE" arg0="0" arg1="3" duration="0"/>
103   </SBEHAVIOUR>
104
105  </MULT>
106 </PLAN>
107 </MDLeScript>
```

B.1.2 Pheromonbasierte Suche von Ressourcen

Listing B.2: MDL2ϵ-Plan für das Hop-Count-Experiment.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2
3  <!DOCTYPE MDLeScript SYSTEM "MDLeScriptSchema.dtd">
4
5  <MDLeScript>
6    <PLAN name="PLAN" duration="infinite">
7
8      <ATOM name="ASETSBLED" arg0="2" arg1="1" duration="0"/>
9      <ATOM name="ASTOP" interrupt="NOT(IODEMPOS)" duration="infinite"/>
10       <ATOM name="ASETPOI" arg0="VOPOSX" arg1="VOPOSY" arg2="0"
           duration="0"/>
11     <ATOM name="ASETSBLED" arg0="2" arg1="0" duration="0"/>
12     <ATOM name="ASETSBLED" arg0="3" arg1="0" duration="0"/>
13
14     <!--
15     VACCUM0 holds all temporary calculations
16     VACCUM1 holds the direction of the last rotation
17     VACCUM2 holds the number of collisions
```

```
18      VACCUM3 holds the current state: 0 – random, 1 – pheromones, 2 – from food1,
            3 – from food2
19      VACCUM4 holds the pheromone threshold
20      VACCUM5 holds the IR threshold
21      -->
22      <ATOM name="ASTORE" arg0="40" arg1="4" duration="0"/>
23      <ATOM name="ASTORE" arg0="20" arg1="5" duration="0"/>
24
25      <MULT>
26        <BEHAVIOUR name="foraging" interrupt="AND(NOT(GEQ(VACCUM3,2)),NOT
            (␣OR(␣OR(EQ(VODATAL,21845),EQ(VODATAR,21845))␣,OR(EQ(VODATAL
            ,29127),EQ(VODATAR,29127))␣)␣))">
27          <MULT>
28
29            <SBEHAVIOUR name="avoid␣front" interrupt="GT(VIR0,VACCUM5)">
30              <ATOM name="AADD" arg0="VACCUM2" arg1="1" arg2="2" interrupt="
                  GT(VIRMSGCNT0,0)" duration="0"/>
31              <UNION>
32                <ATOM name="AROTL" interrupt="ISPACEL" arg0="50" duration="10
                    "/>
33                <ATOM name="AROTR" interrupt="NOT(ISPACEL)" arg0="50" duration
                    ="10"/>
34              </UNION>
35              <UNION>
36                <ATOM name="AMOVE" arg0="50" interrupt="NOT(GT(VIR0,VACCUM5)
                    )" duration="10"/>
37                <ATOM name="AMOVE" arg0="-50" interrupt="AND(GT(VIR0,40),NOT
                    (IBACK))" duration="10"/>
38              </UNION>
39            </SBEHAVIOUR>
40
41            <!-- wait for  data and calculate  absolute difference  -->
42            <ATOM name="ASTOP" duration="5"/>
43            <ATOM name="ASTOP" interrupt="NOT(IODEMGREY)" duration="10"/>
44
45            <UNION>
46              <SBEHAVIOUR name="followTrack" interrupt="OR(GT(VOGREYR,
                  VACCUM4),GT(VOGREYL,VACCUM4))">
47                <ATOM name="ASTORE" arg0="1" arg1="3" duration="0"/>
48                <ATOM name="ASUB" arg0="VOGREYL" arg1="VOGREYR" duration="0
                    "/>
49
50                <SBEHAVIOUR name="downhill" interrupt="AND(GT(VOLASTGREYR,
                    VOGREYR),GT(VOLASTGREYL,VOGREYL))">
51                  <UNION>
52                    <BEHAVIOUR name="alternating␣rotation" interrupt="AND(
```

```
                  GEQ(VACCUM0,-3),GEQ(3,VACCUM0))">
53              <ATOM name="AROTDEG" arg0="80" interrupt="EQ(VACCUM1,0)
                  " duration="0"/>
54              <ATOM name="AROTDEG" arg0="-80" interrupt="EQ(VACCUM1
                  ,1)" duration="0"/>
55              <UNION>
56                <ATOM name="ASTORE" arg0="1" arg1="1" interrupt="EQ(
                    VACCUM1,0)" duration="0"/>
57                <ATOM name="ASTORE" arg0="0" arg1="1" interrupt="EQ(
                    VACCUM1,1)" duration="0"/>
58              </UNION>
59            </BEHAVIOUR>
60            <ATOM name="AROTDEG" interrupt="GT(VACCUM0,3)" arg0="30"
                duration="0"/>
61            <ATOM name="AROTDEG" interrupt="GT(-3,VACCUM0)" arg0="
                -30" duration="0"/>
62          </UNION>
63          <ATOM name="ANOP" interrupt="IMOTION" duration="10"/>
64          <ATOM name="ASTOP" duration="0"/>
65        </SBEHAVIOUR>
66        <ATOM name="AMOVE" arg0="50" interrupt="NOT(GT(VIR0,VACCUM5)
            )" duration="10"/>
67
68      </SBEHAVIOUR>
69
70      <SBEHAVIOUR name="randomWalk" interrupt="NOT(GT(VIR0,VACCUM5
          ))">
71        <ATOM name="ASTORE" arg0="0" arg1="3" duration="0"/>
72        <BEHAVIOUR name="walk" interrupt="NOT(GT(VIR0,VACCUM5))">
73          <RUNION>
74            <ATOM name="AMOVE" arg0="50" duration="10" probability="70"
                />
75            <BEHAVIOUR name="turn␣right" probability="15">
76              <ATOM name="AROTDEG" arg0="-60" duration="0"/>
77              <ATOM name="ANOP" interrupt="IMOTION" duration="10"/>
78              <ATOM name="ASTOP" duration="0"/>
79            </BEHAVIOUR>
80            <BEHAVIOUR name="turn␣left" probability="15">
81              <ATOM name="AROTDEG" arg0="60" duration="0"/>
82              <ATOM name="ANOP" interrupt="IMOTION" duration="10"/>
83              <ATOM name="ASTOP" duration="0"/>
84            </BEHAVIOUR>
85          </RUNION>
86        </BEHAVIOUR>
87      </SBEHAVIOUR>
88    </UNION>
```

```
89        </MULT>
90      </BEHAVIOUR>
91
92      <SBEHAVIOUR name="got␣food" interrupt="AND(NOT(GEQ(VACCUM3,2)),␣
            OR(␣OR(EQ(VODATAL,21845),EQ(VODATAR,21845))␣,OR(EQ(VODATAL
            ,29127),EQ(VODATAR,29127))␣)␣)">
93        <UNION>
94          <ATOM name="ASTORE" arg0="2" arg1="3" interrupt="OR(EQ(VODATAL
                ,21845),EQ(VODATAR,21845))" duration="0"/>
95          <ATOM name="ASTORE" arg0="3" arg1="3" interrupt="OR(EQ(VODATAL
                ,29127),EQ(VODATAR,29127))" duration="0"/>
96        </UNION>
97        <ATOM name="ASENDRADIO" arg0="104" arg1="VODATAL" arg2="VODATAR
              " duration="0"/>
98        <ATOM name="ASETSBLED" arg0="3" arg1="1" duration="0"/>
99        <ATOM name="AMOVE" arg0="50" interrupt="NOT(GT(VIR0,VACCUM5))"
              duration="10"/>
100     </SBEHAVIOUR>
101
102     <BEHAVIOUR name="homing" interrupt="AND(GEQ(VACCUM3,2),NOT(OR(EQ(
            VODATAL,13107),EQ(VODATAR,13107))))">
103       <MULT>
104         <ATOM name="ASTOP" duration="5"/>
105         <ATOM name="ASTOP" interrupt="NOT(IPOI)" duration="10"/>
106         <ATOM name="AROTDEG" arg0="VPOIDIR0" interrupt="IPOI" duration="0
                "/>
107         <ATOM name="ANOP" interrupt="IMOTION" duration="1000"/>
108         <ATOM name="ASTOP" duration="0"/>
109         <ATOM name="AMOVE" arg0="50" interrupt="NOT(GT(VIR0,VACCUM5))"
                duration="10"/>
110         <ATOM name="ASTOP" interrupt="GT(VIR0,VACCUM5)" duration="10"/>
111         <SBEHAVIOUR name="avoid␣and␣move" interrupt="GT(VIR0,VACCUM5)
                ">
112           <BEHAVIOUR name="avoid␣front" interrupt="GT(VIR0,VACCUM5)"/>
113           <ATOM name="AMOVE" arg0="50" interrupt="NOT(GT(VIR0,VACCUM5))
                  " duration="5"/>
114         </SBEHAVIOUR>
115       </MULT>
116     </BEHAVIOUR>
117
118     <SBEHAVIOUR name="home␣sweet␣home" interrupt="AND(GEQ(VACCUM3,2)
            ,OR(EQ(VODATAL,13107),EQ(VODATAR,13107)))">
119       <ATOM name="ASENDRADIO" arg0="104" arg1="VODATAL" arg2="VODATAR
              " duration="0"/>
120       <ATOM name="ASETSBLED" arg0="3" arg1="0" duration="0"/>
121       <RUNION>
```

```
122             <ATOM name="AROTDEG" arg0="-180" duration="0"/>
123             <ATOM name="AROTDEG" arg0="180" duration="0"/>
124           </RUNION>
125           <ATOM name="ANOP" interrupt="IMOTION" duration="10"/>
126           <ATOM name="ASTORE" arg0="0" arg1="3" duration="0"/>
127         </SBEHAVIOUR>
128 
129     </MULT>
130   </PLAN>
131 </MDLeScript>
```

B.2 Sammeln von Ressourcen unter energetischen Gesichtspunkten

Listing B.3: MDL2ϵ-Plan für das Hop-Count-Experiment.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE MDLeScript SYSTEM "MDLeScriptSchema.dtd">
3  
4  <MDLeScript>
5    <PLAN name="PLAN" duration="infinite">
6      <RUNION> <!-- choose first target randomly -->
7        <ATOM name="ASETTARGET" arg0="1" duration="0"/>
8        <ATOM name="ASETTARGET" arg0="2" duration="0"/>
9      </RUNION>
10 
11     <ATOM name="AHOPCTL" arg0="1" duration="0"/>
12     <ATOM name="ARAWDATACTL" arg0="50" duration="0"/>
13     <ATOM name="ASETSBLED" arg0="3" arg1="1" duration="0"/>
14 
15     <MULT> <!-- main loop -->
16       <BEHAVIOUR name="no␣line" interrupt="NOT(ILINE)">
17         <MULT>
18 
19           <BEHAVIOUR name="to␣line" interrupt="EQ(VTARGET,1)">
20             <MULT>
21 
22               <ATOM name="ACOLOR" arg0="255" arg1="0" arg2="0" duration="0
                   "/>
23               <BEHAVIOUR name="random␣walk" interrupt="OR(GT(1,VHOPS0),
                   EQ(VBC0,-1))">
24                 <MULT>
25 
26                   <BEHAVIOUR name="avoid␣front" interrupt="NOT(IFREE)">
27                     <MULT multiplicity="5">
28                       <UNION>
```

```
29              <SATOM name="AROTL" interrupt="ISPACEL" arg0="50"/>
30              <SATOM name="AROTR" interrupt="NOT(ISPACEL)" arg0="
                    50"/>
31            </UNION>
32          </MULT>
33        </BEHAVIOUR>
34
35        <UNION>
36          <ATOM name="AMOVE" arg0="-50" interrupt="AND(NOT(IFREE)
                ,NOT(IBACK))" duration="10"/>
37          <ATOM name="AMOVE" arg0="80" probability="8" interrupt="
                IFREE" duration="10"/>
38        </UNION>
39        <!-- mark light -->
40        <ATOM name="AMARKZONE" arg0="1" arg1="5" interrupt="AND(
              GT(VORAWL,150),GT(VORAWR,150))" duration="0"/>
41      </MULT>
42    </BEHAVIOUR>
43
44    <SBEHAVIOUR name="go␣to␣line" interrupt="NOT(OR(GT(1,
          VHOPS0),EQ(VBC0,-1)))">
45      <RUNION>
46
47        <SBEHAVIOUR probability="90" name="proceed␣to␣line">
48  <!-- keep direction static -->
49          <ATOM name="AADD" arg0="0" arg1="VHOPDIR0" duration = "0
                "/>
50          <ATOM name="ACOLOR" arg0="255" arg1="0" arg2="255"
                duration="0"/>
51          <SBEHAVIOUR name="turn␣to␣accum">
52            <ATOM name="AROTDEG" arg0="VACCUM" duration="5"
                  interrupt="OR(GT(-5,VACCUM),GT(VACCUM,5))"/>
53          </SBEHAVIOUR>
54
55          <ATOM name="AMARKZONE" arg0="1" arg1="5" interrupt="AND
                (GT(VORAWL,150),GT(VORAWR,150))" duration="0"/>
56          <ATOM name="AMOVE" interrupt="IFREE" duration="5"/>
57          <BEHAVIOUR name="avoid␣front" interrupt="NOT(IFREE)"/>
58        </SBEHAVIOUR>
59
60        <SBEHAVIOUR probability="10" name="do␣not␣proceed␣to␣
              line" duration="50">
61          <ATOM name="ACOLOR" arg0="255" arg1="0" arg2="128"
                duration="0"/>
62          <BEHAVIOUR name="random␣walk"/>
63        </SBEHAVIOUR>
```

```
64
65                  </RUNION>
66
67                </SBEHAVIOUR>
68
69              </MULT>
70            </BEHAVIOUR>
71
72            <BEHAVIOUR name="to data" interrupt="EQ(VTARGET,2)">
73              <MULT>
74
75                <BEHAVIOUR name="no data" interrupt="NOT(AND(GT(VORAWL
                      ,150),GT(VORAWR,150)))">
76                  <MULT>
77
78                    <ATOM name="ACOLOR" arg0="0" arg1="255" arg2="0" duration
                        ="0"/>
79                    <BEHAVIOUR name="random walk" interrupt="OR(GT(1,VHOPS1
                        ),EQ(VBC1,-1))"/>
80
81                    <SBEHAVIOUR name="go to data" interrupt="NOT(OR(GT(1,
                        VHOPS1),EQ(VBC1,-1)))">
82                      <RUNION>
83
84                        <SBEHAVIOUR probability="90" name="proceed to light"
                            >
85                          <ATOM name="AADD" arg0="0" arg1="VHOPDIR1" duration
                              = "0"/>
86                          <ATOM name="ACOLOR" arg0="0" arg1="255" arg2="255"
                              duration="0"/>
87                          <SBEHAVIOUR name="turn to accum"/>
88                          <ATOM name="AMOVE" interrupt="IFREE" duration="5"/>
89                          <BEHAVIOUR name="avoid front" interrupt="NOT(IFREE
                              )"/>
90                        </SBEHAVIOUR>
91
92                        <SBEHAVIOUR probability="10" name="do not proceed to
                             light" duration="50">
93                           <ATOM name="ACOLOR" arg0="0" arg1="255" arg2="128
                               " duration="0"/>
94                          <BEHAVIOUR name="random walk"/>
95                        </SBEHAVIOUR>
96
97                      </RUNION>
98
99                    </SBEHAVIOUR>
```

```
100
101                 </MULT>
102               </BEHAVIOUR>
103
104               <SBEHAVIOUR name="found␣light" interrupt="AND(GT(VORAWL
                      ,150),GT(VORAWR,150))">
105                 <ATOM name="AMARKZONE" arg0="1" arg1="5" duration="0"/>
106                 <ATOM name="ASETTARGET" arg0="1" duration="0"/> <!-- to line
                        -->
107               </SBEHAVIOUR>
108
109             </MULT>
110           </BEHAVIOUR>
111
112         </MULT>
113       </BEHAVIOUR>
114
115       <!-- stop to prevent false detections -->
116       <SATOM name="ASTOP" interrupt="ILINE" duration="5"/>
117       <SBEHAVIOUR name="found␣line" interrupt="ILINE">
118         <ATOM name="ACOLOR" arg0="0" arg2="255" arg1="0" duration="0"/>
119         <RUNION> <!-- turn away -->
120           <SATOM name="AROTDEG" arg0="180" duration="10"/>
121           <SATOM name="AROTDEG" arg0="-180" duration="10"/>
122         </RUNION>
123
124         <ATOM name="AMARKZONE" arg0="0" arg1="5" duration="0"/>
125         <ATOM name="ASETTARGET" arg0="2" interrupt="EQ(VTARGET,1)"/> <!--
               to data -->
126
127         <BEHAVIOUR name="still␣not␣on␣white" interrupt="NOT(IWHITEALL)"
               duration="100">
128           <MULT>
129             <ATOM name="AMOVE" interrupt="IFREE" duration="10"/>
130             <ATOM name="ASTOP" interrupt="NOT(IFREE)" duration="10"/>
131           </MULT>
132         </BEHAVIOUR>
133
134       </SBEHAVIOUR>
135     </MULT>
136   </PLAN>
137 </MDLeScript>
```

Listing B.4: MDL2ϵ-Plan für das Selbst-Lade-Experiment.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE MDLeScript SYSTEM "MDLeScriptSchema.dtd">
```

```
3
4  <MDLeScript>
5    <PLAN name="PLAN" duration="infinite">
6      <ATOM name="ANEEDPARAM" arg0="0" duration="0"/> <!-- param 0 is bat0
           threshold -->
7      <ATOM name="ANEEDPARAM" arg0="1" duration="0"/> <!-- param 1 is bat1
           threshold -->
8
9      <!-- set  initial  target  -->
10     <UNION>
11       <ATOM name="ASETTARGET" arg0="1" interrupt="IMUSTCHARGE"/> <!--
             charge station -->
12       <ATOM name="ASETTARGET" arg0="0"/> <!-- no target -->
13     </UNION>
14     <ATOM name="ACOLOR" arg0="0" arg1="0" arg2="0" duration = "0"/>
15
16     <BEHAVIOUR name="watchdog" interrupt="GT(VBAT0,24100)">
17       <MULT>
18
19   <BEHAVIOUR name="battery␣is␣ok" interrupt="OR(EQ(VTARGET,1),NOT(
       IMUSTCHARGE))">
20           <MULT>
21
22             <ATOM name="ACOLOR" arg0="0" arg1="255" arg2="0" duration = "0"
                 interrupt="EQ(VTARGET,0)"/>
23             <BEHAVIOUR name="no␣target" interrupt="EQ(VTARGET,0)">
24               <MULT>
25
26                 <BEHAVIOUR name="random␣walk" interrupt="OR(IWHITEONE,
                     ILINE)">
27                   <MULT>
28
29                     <BEHAVIOUR name="avoid" interrupt="IFRONT">
30                       <UNION>
31                         <SATOM name="AROTL" interrupt="ISPACEL" arg0="50"
                             duration="10"/>
32                         <ATOM name="AROTR" arg0="30" duration="10"/>
33                       </UNION>
34                     </BEHAVIOUR>
35
36                     <BEHAVIOUR name="walk" interrupt="NOT(IFRONT)">
37                       <RUNION>
38                         <ATOM name="AMOVE" arg0="80" probability="8" duration=
                             "10"/>
39                         <ATOM name="AROTL" arg0="30"/>
40                         <ATOM name="AROTR" arg0="30"/>
```

```
41                </RUNION>
42              </BEHAVIOUR>
43
44            </MULT>
45          </BEHAVIOUR>
46
47          <SBEHAVIOUR name="get␣away␣from␣grey" interrupt="
              IGREYONE">
48  <!-- leave grey area -->
49            <BEHAVIOUR name="grey␣go␣go␣go" interrupt="OR(IGREYONE,
                ILINE)" duration="100">
50              <MULT>
51                <ATOM name="AMOVE" arg0="-10" interrupt="NOT(IBACK)"
                    duration="10"/>
52                <ATOM name="ASTOP" interrupt="IBACK" duration="10"/>
53              </MULT>
54            </BEHAVIOUR>
55
56            <RUNION> <!-- turn away -->
57              <ATOM name="AROTDEG" arg0="170" duration="10"/>
58              <ATOM name="AROTDEG" arg0="-170" duration="10"/>
59            </RUNION>
60            <ATOM name="AMOVE" interrupt="NOT(IFRONTNEAR)" duration="
                5"/>
61          </SBEHAVIOUR>
62
63        </MULT>
64      </BEHAVIOUR>
65
66      <BEHAVIOUR name="go␣to␣charge␣station" interrupt="EQ(VTARGET
          ,1)">
67        <MULT>
68
69          <ATOM name="ACOLOR" arg0="255" arg1="0" arg2="0" duration =
              "0"/>
70          <BEHAVIOUR name="no␣target" interrupt="AND(NOT(ICHARGE),
              NOT(ILINE))"/>
71
72          <ATOM name="ASTOP" />
73          <BEHAVIOUR name="charge␣line␣guard" interrupt="AND(NOT(
              IFRONTNEAR),NOT(IWHITEALL))">
74
75            <ATOM name="ACOLOR" arg0="0" arg1="0" arg2="255" duration
                = "0"/>
76            <ATOM name="AMOVE" arg0="10" interrupt="ILINE" duration="5
                "/>
```

```
77
78            <BEHAVIOUR name="align␣to␣line" interrupt="NOT(IONLINE)
                 ">
79              <RUNION>
80                <ATOM name="AROTL" arg0="10" duration="10"/>
81                <ATOM name="AROTR" arg0="10" duration="10"/>
82              </RUNION>
83            </BEHAVIOUR>
84
85            <BEHAVIOUR name="search␣charge" interrupt="NOT(ICHARGE)
                 ">
86              <MULT>
87
88                <ATOM name="AMOVE" arg0="30" duration="1"/>
89                <ATOM name="AMOVE" arg0="30" interrupt="IONLINE"
                     duration="10"/>
90                <BEHAVIOUR name="not␣on␣line" interrupt="NOT(IONLINE
                     )">
91                  <UNION>
92                    <ATOM name="AROTL" arg0="10" interrupt="ILINELEFT"
                         duration="10"/>
93                    <ATOM name="AROTR" arg0="10" interrupt="ILINERIGHT
                         " duration="10"/>
94                  </UNION>
95                </BEHAVIOUR>
96
97              </MULT>
98            </BEHAVIOUR>
99
100         </BEHAVIOUR>
101
102         <SBEHAVIOUR name="robot␣collision" interrupt="GT(
               VNEIGHBORSCOUNT0,0)">
103           <BEHAVIOUR name="still␣colliding" interrupt="IFRONTNEAR
                 ">
104             <ATOM name="ASTOP" duration="10"/>
105             <ATOM name="ARANDOM" arg0="40" duration="0"/>
106
107             <BEHAVIOUR name="count␣down" interrupt="GT(VACCUM,0)"
                   >
108               <MULT>
109                 <ATOM name="AADD" arg0="VACCUM" arg1="-1"/>
110               </MULT>
111             </BEHAVIOUR>
112           </BEHAVIOUR>
113         </SBEHAVIOUR>
```

```
114
115                 <SBEHAVIOUR name="rotate␣180␣degrees" interrupt="
                       IFRONTNEAR">
116                   <RUNION>
117                     <ATOM name="AROTDEG" arg0="180" duration="10"/>
118                     <ATOM name="AROTDEG" arg0="-180" duration="10"/>
119                   </RUNION>
120                 </SBEHAVIOUR>
121
122                 <SBEHAVIOUR name="charge␣station␣guard" interrupt="
                       ICHARGE">
123                   <ATOM name="ACOLOR" arg0="255" arg1="255" arg2="255"
                         duration= "0"/>
124                   <BEHAVIOUR name="move␣to␣end" interrupt="NOT(IGREYALL)"
                         >
125                     <MULT>
126
127                       <BEHAVIOUR name="can␣move␣on␣charge␣line" interrupt=
                             "NOT(IFRONTNEAR)">
128                         <MULT>
129
130                           <ATOM name="AMOVE" arg0="10" interrupt="ICHARGE"
                                 duration="5"/>
131                           <BEHAVIOUR name="cannot␣charge" interrupt="NOT(
                                 ICHARGE)">
132                             <UNION>
133                               <ATOM name="AROTL" arg0="5" interrupt="GT(70,
                                     VBRIGHT)" duration="1"/>
134                               <ATOM name="AROTR" arg0="5" interrupt="GT(70,
                                     VBLEFT)" duration="1"/>
135                               <ATOM name="AMOVE" arg0="5" duration="1"/>
136                             </UNION>
137                           </BEHAVIOUR>
138                           <ATOM name="ASETTARGET" arg0="2" interrupt="AND(
                                 NOT(ICHARGE),NOT(IWHITEONE))" duration="0"/>
139
140                         </MULT>
141                       </BEHAVIOUR>
142
143                       <BEHAVIOUR name="at␣charge␣end" interrupt="
                             IFRONTNEAR">
144
145           <!-- leave if not charging and battery are full -->
146                         <ATOM name="ASETTARGET" arg0="2" interrupt="AND(NOT
                               (ICHARGE),AND(NOT(GT(30500,VBAT0)),NOT(GT
                               (30500,VBAT1))))" duration="0"/>
```

```
147
148                     <BEHAVIOUR name="cannot␣charge␣at␣end" interrupt="
                          NOT(ICHARGE)">
149     <!-- move backwards with alternating directions -->
150                         <MULT multiplicity="10">
151                           <UNION>
152                             <SBEHAVIOUR name="rotate␣left␣this␣time"
                                  interrupt="EQ(VACCUM,0)">
153                               <ATOM name="AADD" arg0="0" arg1="1"
                                    duration="0"/>
154                               <ATOM name="AWHEELS" arg0="-10" arg1="-5"
                                    duration="10" interrupt="NOT(IBACK)"/>
155                             </SBEHAVIOUR>
156                             <SBEHAVIOUR name="rotate␣right␣this␣time
                                  " interrupt="NOT(EQ(VACCUM,0))">
157                               <ATOM name="AADD" arg0="0" arg1="0"
                                    duration="0"/>
158                               <ATOM name="AWHEELS" arg0="-5" arg1="-10"
                                    duration="10" interrupt="NOT(IBACK)"/>
159                             </SBEHAVIOUR>
160                           </UNION>
161                         </MULT>
162                         <!-- leave if  still  not charging -->
163                         <ATOM name="ASETTARGET" arg0="2" duration="0"/>
164                     </BEHAVIOUR>
165
166                     <ATOM name="ASTOP"/>
167
168                     <SBEHAVIOUR name="is␣charging" interrupt="ICHARGE"
                          >
169                       <ATOM name="ASTOP" interrupt="ICHARGE" duration="
                            infinite"/>
170                      <ATOM name="ASETTARGET" arg0="2" duration="0"/> <!
                           -- should leave -->
171                     </SBEHAVIOUR>
172
173                   </BEHAVIOUR>
174
175               </MULT>
176             </BEHAVIOUR>
177             <ATOM name="ASETTARGET" arg0="2" duration="0"/>
178            </SBEHAVIOUR>
179
180         </MULT>
181       </BEHAVIOUR>
182
```

```
183             <BEHAVIOUR name="leave␣charge␣station" interrupt="EQ(VTARGET
                    ,2)">
184               <UNION>
185                 <SATOM name="AROTDEG" arg0="90" interrupt="IRIGHT" duration=
                        "10"/>
186                 <SATOM name="AROTDEG" arg0="-90" interrupt="ILEFT" duration=
                        "10"/>
187                 <RUNION> <!-- no blocking, random choice -->
188                   <SATOM name="AROTDEG" arg0="90" duration="10"/>
189                   <SATOM name="AROTDEG" arg0="-90" duration="10"/>
190                 </RUNION>
191               </UNION>
192 
193               <BEHAVIOUR name="still␣on␣white" interrupt="NOT(IGREYALL)"
                      >
194                 <MULT>
195                   <ATOM name="AMOVE" interrupt="NOT(IFRONTNEAR)" duration="
                          10"/>
196                   <ATOM name="ASTOP" interrupt="IFRONTNEAR" duration="10"/>
197                 </MULT>
198               </BEHAVIOUR>
199 
200               <BEHAVIOUR name="still␣on␣grey" interrupt="IGREYONE">
201                 <MULT>
202                   <ATOM name="AMOVE" interrupt="NOT(IFRONTNEAR)" duration="
                          10"/>
203                   <ATOM name="ASTOP" interrupt="IFRONTNEAR" duration="10"/>
204                 </MULT>
205               </BEHAVIOUR>
206 
207               <ATOM name="AMOVE" interrupt="NOT(IFRONTNEAR)" duration="5"/>
208               <ATOM name="ASTOP"/>
209               <ATOM name="ASETTARGET" arg0="0"/> <!-- no target -->
210             </BEHAVIOUR>
211 
212           </MULT>
213         </BEHAVIOUR>
214         <ATOM name="ASETTARGET" arg0="1" interrupt="IMUSTCHARGE"/> <!--
                charge station -->
215       </MULT>
216     </BEHAVIOUR>
217 
218     <MULT> <!-- energy watchdog stopped plan -->
219       <ATOM name="AMOVE" interrupt="NOT(IFRONTNEAR)" duration="10"/>
220       <UNION>
221         <SBEHAVIOUR name="avoid␣robot" interrupt="GT(VNEIGHBORSCOUNT0
```

```
                ,0)">
222             <BEHAVIOUR name="avoid" interrupt="IFRONTNEAR"/>
223           </SBEHAVIOUR>
224           <ATOM name="ASTOP" interrupt="IFRONTNEAR" duration="10"/>
225         </UNION>
226       </MULT>
227     </PLAN>
228 </MDLeScript>
```

Akronyme

ADC	Analog Digital Converter	34
ASIC	Application Specific Integrated Circuit	27
ASL	Autonomous Systems Lab	15
CoMiRo	Collective and Micro Robotics	
DLP	Digital Light Processing	91
DTD	Document Type Definition	64
EC	Evolutionary Computation	8
EEPROM	Electrically Erasable Programmable Read-Only Memory	33
EK	Europäische Komission	18
EPFL	Ecole Polytechnique Federale de Lausanne	14
FET	Future and Emerging Technologies	23
FLSO	Framework for Learning and Self-Organisation	12
FPGA	Field Programmable Gate Array	32
FPS	fitnessproportionale Selektion	108
GP	Genetische Programmierung	3

GPIO General Purpose Input Output Port . . . 34

GUI Graphical User Interface . . . 93

HCS Hop-Count-Strategie . . . 127

IPVS Institut für Parallele und Verteilte Systeme der Universität Stuttgart . . . 24

IBMT Fraunhofer-Institut für Biomedizinische Technik . . . 32

IPR Institut für Prozessrechentechnik, Automation und Robotik . 1

IR Infrarotlicht . . . 14

IsMOS I-SWARM MDL2ϵ Operating System . . . 33

ISR Interrupt-Service-Routine . . . 79

I-SWARM Intelligent Small World Autonomous Robots for Micro Manipulation . . . 1

JaMOS Jasmine MDL2ϵ Operating System . . . 77

JTAG Joint Test Action Group . . . 40

KIT Karlsruher Institut für Technologie . . . 1

KITCoRoL KIT Collective Robotics Lab . . . 37

KNN Künstlichen Neuronalen Netzes . . . 8

KZA Kinetischer Zustandsautomat . . . 55

LAMI Microcomputing Laboratory . . . 14

LED Licht Emittierende Diode . . . 16

LiPoly Lithium Polymer . . . 36

LoC Lines of Code . . . 158

MDL2ϵ**App** MDL2ϵ-Applikation . . . 68

MDL Motion Description Language . . . 54

MDLe extended Motion Description Language . . . 54

MDL2ϵ extended Motion Description Language 2 . . . 42

MIPS Millionen Instruktionen pro Sekunde . . . 34

MRS Multi-Roboter-Systeme . . . 6

MST Mikrosystemtechnik . . . 4

OCU Optical Communication Unit 79
ODE Open Dynamics Engine 83
ODeM Optical Data transmission system for Micro robots 33
PCB Printed Circuit Board 37
PMU Power Management Unit 77
PWM Pulsweitenmodulation 34
RAM Random Access Memory 29
RISC Reduced Instruction Set Computing 33
RBS rangbasierte Selektion 108
RWS Random-Walk-Strategie 127
RNN Rekurrentes Neuronales Netz 49
RTOS Real-Time Operating System 81
SFR Special Function Registers
SPA Sense, Plan, Act 50
SPI Serial Peripheral Interface 34
SRAM Static Random Access Memory 29
TMS Turnierselektion 108
TWI Two-Wire Interface 34
UART Universal Asynchronous Receiver Transmitter 34
μC Microcontroller 15
UML Unified Modelling Language
USB Universal Serial Bus 40
XML eXtensible Markup Language 52

Literaturverzeichnis

[1] C. Ampatzis, E. Tuci, V. Trianni, and M. Dorigo, “Evolution of Signalling in a Group of Robots Controlled by Dynamic Neural Networks,” in *Proc. of the Int. Conf. on Simulation of Adaptive Behaviour (SAB), Workshop on Swarm Robotics*, ser. Lecture Notes in Computer Science, S. Nolfi, G. Baldassarre, R. Calabretta, J. C. T. Hallam, D. Marocco, J.-A. Meyer, O. Miglino, and D. Parisi, Eds., vol. 4433. Rome, Italy: Springer, Sep. 2006.

[2] J. Baillie, “URBI: Towards a Universal Robotic Low-Level Programming Language,” in *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*. IEEE Press, 2005, pp. 3219–3224.

[3] J. E. Baker, “Adaptive selection methods for genetic algorithms,” in *Proc. of the 1st International Conference on Genetic Algorithms*. Mahwah, NJ, USA: Lawrence Erlbaum Associates, Inc., 1985, pp. 101–111.

[4] G. Beni and J. Wang, “Swarm Intelligence in Cellular Robotic Systems,” in *Proc. of the NATO Advanced Workshop on Robots and Biological Systems, Il Ciocco, Italy*, Jun. 1989.

[5] A. Birk, “Fast Robot Prototyping with the Cubesystem,” in *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*. IEEE Press, 2004, pp. 5177–7182.

[6] D. Blank, D. Kumar, L. Meeden, and H. Yanco, “Pyro: A Python-based Versatile Programming Environment for Teaching Robotics,” *J. Educ. Resour. Comput.*, vol. 3, no. 4, p. 1, 2003.

[7] A. Boletis, W. Driesen, J.-M. Breguet, and A. Brunete, “Solar Cell Powering with Integrated Global Positioning System for mm^3 Size Robots,” in *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*. IEEE Press, Oct. 2006, pp. 5528–5533.

[8] V. Braitenberg, *Vehicles: experiments in synthetic psychology*. Cambridge, MA: MIT Press, 1984.

[9] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, “Extensible markup language (xml) 1.0 (fourth edition),” Aug. 2006.

[10] J.-M. Breguet, S. Johansson, W. Driesen, and U. Simu, “A Review on Actuation Principles for Few Cubic Millimeter Sized Mobile Micro-Robots,” in *Proc. of the Int. Conf. on New Actuators (ACTUATOR)*, Bremen, Germany, 2006, pp. 374 – 381.

[11] R. W. Brockett, “On the computer control of movement,” in *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*. IEEE Press, Apr. 1988, pp. 534–540.

[12] ——, “Formal languages for motion description and map making,” *Robotics*, pp. 181–93, 1990.

[13] ——, “Hybrid models for motion control systems,” *Perspectives in Control*, pp. 29–54, 1993.

[14] R. A. Brooks, “A robust layered control system for a mobile robot,” *IEEE Journal of Robotics and Automation*, vol. 2, no. 1, pp. 14–23, Mar. 1986.

[15] J. Brufau, M. Puig-Vidal, J. Lopez-Sanchez, J. Samitier, W. Driesen, J.-M. Breguet, N. Snis, U. Simu, S. Johansson, J. Gao, T. Velten, J. Seyfried, R. Estana, and H. Wörn, “MICRON: Small Autonomous

Robot for Cell Manipulation Applications," in *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*. IEEE Press, 2005.

[16] H. Bruyninckx, "Open Robot Control Software: the OROCOS Project," in *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, vol. 3. Seoul, Korea: IEEE Press, May 2001, pp. 2523–2528.

[17] E. K. Burke, S. Gustafson, and G. Kendall, "Diversity in genetic programming: an analysis of measures and correlation with fitness," *IEEE Journal of Trans. Evolutionary Computation*, vol. 8, no. 1, pp. 47–62, 2004.

[18] G. Caprari, K. Arras, and R. Siegwart, "The Autonomous Miniature Robot Alice: From Prototypes to Applications," in *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*. IEEE Press, 2000.

[19] G. Caprari, P. Balmer, R. Piguet, and R. Siegwart, "The Autonomous Micro Robot ALICE: A platform for Scientific and Commercial Applications," in *Proc. of the International Symposium on Micromechatronics and Human Science (MHS)*, Nagoya, Japan, 1998.

[20] G. Caprari, T. Estier, and R. Siegwart, "Fascination of Down Scaling – Alice the Sugar Cube Robot," *Journal of Micro-Mechatronics*, vol. 1, pp. 177–189, 2002.

[21] G. Caprari, "Autonomous micro-robots," Ph.D. dissertation, EPFL, Lausanne, Switzerland, 2003.

[22] R. Casanova, A. Dieguez, A. Sanuy, A. Arbat, O. Alonso, J. Canals, and J. Samitier, "An Ultra Low Power IC for an Autonomous mm^3-Sized Microrobot," in *Proc. of the Int. Solid-State Circuits Conference*, San Francisco, CA, USA, 2007, pp. 55 – 58.

[23] R. Casanova, A. Dieguez, A. Sanuy, A. Arbat, O. Alonso, J. Canals, M. Puig, and J. Samitier, "Enabling Swarm Behavior in mm^3-sized Robots with Specific Designed Integrated Electronics." in *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*. IEEE Press, 2007, pp. 3797–3802.

[24] A. L. Christensen and M. Dorigo, “Evolving an integrated phototaxis and hole-avoidance behavior for a swarm-bot,” in *Proc. of the 10th International Conference on the Simulation and Synthesis of Living Systems (Alife X)*. MIT Press, 2006, pp. 248–254.

[25] P. Corradi, L. Ranzani, O. Scholz, A. Dieguez, Menciassi, C. A., Laschi, M. Martinelli, , and P. Dario, “Free-Space Optical Communication in a Swarm of Microrobots,” in *Proc. of the 33rd European Conference and Exhibition on Optical Communication (ECOC’07)*, VDE. Berlin, Germany: VDE Verlag, Sep. 2007.

[26] N. Correll and A. Martinoli, “Collective Inspection of Regular Structures using a Swarm of Miniature Robots,” in *Proc. of the Int. Symp. on Experimental Robotics (ISER)*, ser. Springer Tracts in Advanced Robotics, M. Ang and O. Khatib, Eds. Berlin/Heidelberg, Germany: Springer, 2006, vol. 21, pp. 375–386.

[27] N. Correll, C. M. Cianci, X. Raemy, and A. Martinoli, “Self-Organized Embedded Sensor/Actuator Networks for ”Smart” Turbines,” in *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS), Workshop on Network Robot System: Toward intelligent robotic systems integrated with environments*, 2006.

[28] A. Depickère, D. Fresneau, and J.-L. Deneubourg, “A basis for spatial and social patterns in ant species: dynamics and mechanisms of aggregation,” *Journal of Insect Behavior*, vol. 17, pp. 81–97, Jan. 2004.

[29] V. Dikovski, “Optimierung des MDL2e basierten eingebetteten Betriebssystems JaMOS,” Diplomarbeit, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, Oct. 2006.

[30] M. Dorigo, V. Trianni, E. Sahin, R. Groß, T. H. Labella, G. Baldassarre, S. Nolfi, F. Mondada, J.-L. Deneubourg, D. Floreano, and L. M. Gambardella, “Evolving Self-Organizing Behaviors for a Swarmbot,” *Autonomous Robots, special Issue on Swarm Robotics*, vol. 17, no. 2-3, pp. 223–245, 2004.

[31] A. Ekárt and S. Z. Németh, “Maintaining the diversity of genetic programs,” in *Proc. of the 5th European Conference on Genetic Programming*. London, UK: Springer-Verlag, 2002, pp. 162–171.

[32] S. Fatikow and U. Rembold, *Microsystem Technology and Microrobotics*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1997.

[33] J. Fischer, "A MDL2e based Genetic Programming Approach for Automated Generation of Swarm Behaviour," Diplomarbeit, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, May 2008.

[34] D. Floreano and F. Mondada, "Automatic Creation of an Autonomous Agent: Genetic Evolution of a Neural Network Driven Robot," in *Proc. of the Int. Conf. on Simulation of Adaptive Behaviour (SAB)*, D. Cliff, P. Husbands, J.-A. Meyer, and S. Wilson, Eds. MA: MIT Press, 1994, d. Cliff, P. Husbands, J.-A. Meyer, and S. Wilson (eds.).

[35] ——, "Evolution of Homing Navigation in a Real Mobile Robot," *IEEE Transactions on Systems, Man and Cybernetics Part B : Cybernetics*, vol. 26, no. 3, pp. 396–407, 1996.

[36] ——, "Evolution of Plastic Neurocontrollers for Situated Agents," in *Proc. of the Int. Conf. on Simulation of Adaptive Behaviour (SAB)*, P. Maes, M. Mataric, J.-A. Meyer, J. Pollack, and S. Wilson, Eds. MA: MIT Press, 1996, p. Maes, M. Mataric, J-A. Meyer, J. Pollack, and S. Wilson (eds.).

[37] D. Floreano, S. Nolfi, and F. Mondada, "Competitive Co-Evolutionary Robotics: From Theory to Practice," in *Proc. of the Int. Conf. on Simulation of Adaptive Behaviour (SAB)*, R. Pfeifer, Ed. MA: MIT Press, 1998, r. Pfeifer (eds).

[38] N. R. Franks, S. C. Pratt, E. B. Mallon, N. F. Britton, and D. J. Sumpter, "Information flow, opinion polling and collective intelligence in house-hunting social insects." *Philosophical transactions of the Royal Society of London. Series B, Biological sciences*, vol. 357, no. 1427, pp. 1567–1583, Nov. 2002.

[39] G. Fricke, D. Milutinovic, and D. P. Garg, "Sensing and estimation on a modular testbed for swarm robotics," in *Proc. of the 2nd Annual Dynamic Systems and Control Conference*, Hollywood, CA, USA, Oct. 2009.

[40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[41] S. Garnier, F. Tache, M. Combe, A. Grimal, and G. Theraulaz, “Alice in pheromone land: An experimental setup for the study of ant-like robots,” in *Proc. of the IEEE Swarm Intelligence Symposium (SIS 2007)*. Honolulu, HI, USA: IEEE Press, Apr. 2007, pp. 37 –44.

[42] S. Garnier, C. Jost, J. Gautrais, M. Asadpour, G. Caprari, R. Jeanson, A. Grimal, and G. Theraulaz, “The Embodiment of Cockroach Aggregation Behavior in a Group of Micro-Robots,” *Artificial Life*, vol. 14, no. 4, pp. 387–408, 2008.

[43] B. P. Gerkey, R. T. Vaughan, and A. Howard, “The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems,” in *Proc. of the Int. Conf. on Advanced Robotics*, Coimbra, Portugal, Jun. 2003, pp. 317–323.

[44] S. Goss, S. Aron, J. Deneubourg, and J. Pasteels, “Self-organized shortcuts in the argentine ant,” *Naturwissenschaften*, vol. 76, no. 12, pp. 579–581, Dec. 1989.

[45] P.-P. Grassé, “La reconstruction du nid et les coordinations interindividuelles chez bellicositermes natalensis et cubitermes sp. la théorie de la stigmergie: Essai d’interprétation du comportement des termites construcieurs,” *Insectes Sociaux*, vol. 6, pp. 41–84, 1959.

[46] R. Groß, F. Mondada, and M. Dorigo, “Transport of an object by six pre-attached robots interacting via physical links,” in *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*. Los Alamitos, CA, USA: IEEE Press, 2006, pp. 1317–1323.

[47] H. Hamann, M. Szymanski, and H. Wörn, “Orientation in a trail network by exploiting its geometry for swarm robotics,” in *Proc. of the IEEE Swarm Intelligence Symposium (SIS 2007)*. Honolulu, HI, USA: IEEE Press, Apr. 2007, pp. 310–315.

[48] J. H. Holland, *Adaptation in natural and artificial systems*. Cambridge, MA, USA: MIT Press, 1992.

[49] I. Ieropoulos, C. Melhuish, and J. Greenman, “Artificial metabolism: Towards true energetic autonomy in artificial life,” in *Advances in Artificial Life*, ser. Lecture Notes in Computer Science, W. Banzhaf, T. Christaller, P. Dittrich, J. T. Kim, and J. Ziegler, Eds. Dortmund,

Germany: Springer Berlin / Heidelberg, Sep. 2003, vol. 2801, pp. 792–799.

[50] E. C. M. A. International, *ECMA-262: ECMAScript Language Specification*, 3rd ed. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), Dec. 1999.

[51] C. Jones and M. J. Matarić, "Adaptive division of labor in large-scale minimalist multi-robot systems," in *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*. IEEE Press, 2003, pp. 27–31.

[52] C. Jost, S. Garnier, R. Jeanson, M. Asadpour, J. Gautrais, and G. Theraulaz, "The embodiment of cockroach behaviour in a micro-robot," in *Proc. of the 35th Int. Symposium on Robotics*, Paris, France, Mar. 2004.

[53] R. Kaltofen, Ed., *Tabellenbuch Chemie*, 8th ed. Leipzig: Deutscher Verlag für Grundstoffindustrie, 1980.

[54] T. Kazama, K. Sugawara, and T. Watanabe, "Collecting Behavior of Interacting Robots with Virtual Pheromone," in *Proc. of the Int. Symp. on Distributed Autonomous Robotic Systems (DARS)*, R. Alami, R. Chatila, and H. Asama, Eds. Springer Japan, 2007, pp. 347–356.

[55] S. Kernbach, L. Ricotti, J. Liedke, P. Corradi, and M. Rothermel, "Study of macroscopic morphological features of symbiotic robotic organisms," in *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS), Workshop on Self-Reconfigurable Robots*. Nice, Italy: IEEE Press, 2008, pp. 18–25.

[56] S. Kernbach, E. Meister, F. Schlachter, K. Jebens, M. Szymanski, J. Liedke, D. Laneri, L. Winkler, T. Schmickl, R. Thenius, P. Corradi, and L. Ricotti, "Symbiotic robot organisms: Replicator and symbrion projects," in *Proc. of the PerMIS 08*, Gaithersburg, MD, USA, 2008.

[57] ——, "Symbiotic robot organisms: Replicator and symbrion projects," in *Proc. of the 8th Workshop on Performance Metrics for Intelligent Systems (PerMIS)*. New York, NY, USA: ACM, 2008, pp. 62–69.

[58] A. Kettler, Szymanski, M., Liedke, J., and H. Wörn, “Introducing Wanda – A New Robot for Research, Education, and Arts,” in *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, IEEE. Taipei, Taiwan: IEEE Press, Oct 2010.

[59] F. Klassner, “A case study of lego mindstorms’™suitability for artificial intelligence and robotics courses at the college level,” in *Proc. of the 33rd SIGCSE technical symposium on Computer science education*. New York, NY, USA: ACM, 2002, pp. 8–12.

[60] J. Klein, “breve: a 3d environment for the simulation of decentralized systems and artificial life,” in *Proc. of the 8th Int. Conf on Artificial Life (ICAL*. Cambridge, MA, USA: MIT Press, 2003, pp. 329–334.

[61] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.

[62] M. J. B. Krieger and J.-B. Billeter, “The call of duty: Self-organised task allocation in a population of up to twelve mobile robots,” *Robotics and Autonomous Systems*, vol. 30, no. 1-2, pp. 65–84, Jan. 2000.

[63] T. H. Labella, M. Dorigo, and J.-L. Deneubourg, “Efficiency and task allocation in prey retrieval,” in *Biologically Inspired Approaches to Advanced Information Technology*, 2004, pp. 274–289.

[64] K. Lerman and A. Galstyan, “Mathematical Model of Foraging in a Group of Robots: Effect of Interference,” *Auton. Robots*, vol. 13, no. 2, pp. 127–141, 2002.

[65] W. Liu, A. F. T. Winfield, J. Sa, J. Chen, and L. Dou, “Towards energy optimization: Emergent task allocation in a swarm of foraging robots,” *Adaptive Behavior - Animals, Animats, Software Agents, Robots, Adaptive Systems*, vol. 15, no. 3, pp. 289–305, 2007.

[66] M. Lüdtke, “Entwurf und implementierung einer funkbasierten schnittstelle for automatisierte schwarmroboterexperiemente,” Studienarbeit, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, Jun. 2009.

[67] ——, "Design and Analysis of Bio-Inspired Strategies for Harvesting Swarm Robots under Energetic Constraints," Diplomarbeit, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, Jun. 2010.

[68] S. Magnenat, P. Retornaz, M. Bonani, V. Longchamp, and F. Mondada, "ASEBA: A Modular Architecture for Event-Based Control of Complex Robots," *Mechatronics, IEEE/ASME Transactions on*, vol. PP, no. 99, pp. 1–9, 2010.

[69] S. Magnenat, P. Rétornaz, B. Noris, and F. Mondada, "Scripting the swarm: event-based control of microcontroller-based robots." in *Proc. of the Int. Conf. on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR), TERECoP Workshop*, E. Menegatti, Ed., 2008, workshop Proc. ISBN: 978-88-95872-01-8.

[70] V. Manikonda, P. S. Krishnaprasad, and J. Hendler, "Languages, behaviors, hybrid architectures, and motion control," *Mathematical control theory*, pp. 200–226, 1999.

[71] L. Marcolino and L. Chaimowicz, "No robot left behind: Coordination to overcome local minima in swarm navigation," in *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*. IEEE Press, May. 2008, pp. 1904–1909.

[72] S. Martel, "Fundamental principles and issues of high-speed piezoactuated three-legged motion for miniature robots designed for nanometer-scale operations," *Int. J. Rob. Res.*, vol. 24, no. 7, pp. 575–588, 2005.

[73] A. Martinoli and K. Easton, "Modeling Swarm Robotic Systems," in *Proc. of the Eight Int. Symp. on Experimental Robotics (ISER)*, 2002, pp. 297–306, in B. Siciliano and P. Dario, editors, Springer Tracts in Advanced Robotics (2003), Vol. 5, pp. 297-306.

[74] D. Mawhinney, "Preventing early convergence in genetic programming by replacing similar programs," in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*. San Diego, CA, USA: IEEE Press, 2000, pp. 67–72.

[75] A. McLellan and C. Rowland, "A honeybee colony swarming model," *Ecological Modelling*, vol. 33, no. 2-4, pp. 137 – 148, 1986, terrestrial Ecosystems.

[76] C. Melhuish, I. Ieropoulos, J. Greenman, and I. Horsfield, "Energetically autonomous robots: Food for thought," *Autonomous Robots*, vol. 21, no. 3, pp. 187–198, Nov 2006.

[77] G. Mermoud, L. Matthey, W. C. Evans, and A. Martinoli, "Aggregation-mediated Collective Perception and Action in a Group of Miniature Robots," in *Proc. of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS-2010)*, W. van der Hoek, G. A. Kaminka, M. Luck, and S. Sen, Eds., Toronto, Canada, 2010, pp. 599–606.

[78] G. Metta, P. Fitzpatrick, and L. Natale, "YARP: Yet Another Robot Platform," *International Journal of Advanced Robotics Systems, special issue on Software Development and Integration in Robotics*, vol. 3, no. 1, 2006.

[79] F. Mondada and D. Floreano, "Evolution and Mobile Autonomous Robotics," in *Towards Evolvable Hardware, The Evolutionary Engineering Approach*, ser. Lecture Notes in Computer Science. Berlin: Springer-Verlag, 1995, pp. 221 – 249.

[80] F. Mondada, E. Franzi, and A. Guignard, "The Development of Khepera," in *Experiments with the Mini-Robot Khepera*, ser. HNI-Verlagsschriftenreihe, Heinz Nixdorf Institut, 1999, pp. 7–14.

[81] F. Mondada, L. Gambardella, D. Floreano, and M. Dorigo, "The cooperation of swarm-bots: physical interactions in collective robotics," *Robotics & Automation Magazine*, vol. 12, no. 2, pp. 21–28, 2005.

[82] F. Mondada, A. Guignard, M. Bonani, D. Bär, M. Lauria, and D. Floreano, "Swarm-bot: From concept to implementation," in *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*. Las Vegas, Nevada, US: IEEE Press, Oct. 2003, pp. 1626–1631.

[83] F. Mondada, G. C. Pettinaro, A. Guignard, I. Kwee, D. Floreano, J.-L. Deneubourg, S. Nolfi, L. Gambardella, and M. Dorigo, "SWARM-BOT: a New Distributed Robotic Concept," *Autonomous Robots*, vol. 17, no. 2–3, pp. 193–221, 2004.

[84] F. Mondada, M. Bonani, X. Raemy, J. Pugh, C. Cianci, A. Klaptocz, S. Magnenat, J.-C. Zufferey, D. Floreano, and A. Martinoli, "The e-puck, a Robot Designed for Education in Engineering," in *Proc. of*

the 9th Conference on Autonomous Robot Systems and Competitions, vol. 1. Portugal: IPCB: Instituto Politécnico de Castelo Branco, 2009, pp. 59–65.

[85] S. Nolfi and D. Floreano, "Co-evolving predator and prey robots: Do 'arm races' arise in artificial evolution?" *Artificial Life*, vol. 4, no. 4, pp. 311–335, 1998.

[86] S. Nolfi, D. Floreano, O. Miglino, and F. Mondada, "How to Evolve Autonomous Robots: Different Approaches in Evolutionary Robotics," in *Proc. of the 4th International Workshop on Artificial Life*, R. A. Brooks and P. Maes, Eds. MA: MIT Press, 1994, r. A. Brooks and P. Maes (eds.).

[87] P. Nordin and W. Banzhaf, "Genetic programming controlling a miniature robot," in *WORKING NOTES FOR THE AAAI SYMPOSIUM ON GENETIC PROGRAMMING*. AAAI, 1995, pp. 61–67.

[88] ——, "Real time control of a khepera robot using genetic programming," *CYBERNETICS AND CONTROL*, vol. 26, pp. 533–561, 1997.

[89] S. Nouyan, R. Groß, M. Dorigo, M. Bonani, and F. Mondada, "Group Transport along a Robot Chain in a Self-Organised Robot Colony," in *Proc. of the 9th Int. Conf. on Intelligent Autonomous Systems (IOS2005)*. IOS Press, 2005, pp. 433–442.

[90] A. Ranganath, "Distributed control algorithm for a multicellular robotic organism," Master's thesis, School of Informatics at the University of Edingburgh; Institute of Perception, Action and Behaviour, Edingburgh, UK, 2009.

[91] P.-Y. Rault, "ODeM – Optisches Datenübertragungs System für Mikroroboter Schwärme," Diplomarbeit, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, Jul. 2006.

[92] H. Ritter, T. Martinetz, and K. Schulten, *Neuronale Netze*, 2nd ed. Bonn [u.a.]: Addison-Wesley, 1991.

[93] R. Rojas, *Theorie der neuronalen Netze*. Berlin/Heidelberg, Germany: Springer, 1993.

[94] E. Sahin, “Swarm Robotics: From Sources of Inspiration to Domains of Application,” in *Swarm Robotics*, ser. Lecture Notes in Computer Science, E. Sahin and W. M. Spears, Eds., vol. 3342. Santa Monica, CA, USA: Springer, Jul. 2004, pp. 10–20.

[95] E. Sahin and W. M. Spears, Eds., *Swarm Robotics, Proc. of the Int. Conf. on Simulation of Adaptive Behaviour (SAB), Workshop on Swarm Robotics, Selected Papers*, ser. Lecture Notes in Computer Science, vol. 3342. Santa Monica, CA, USA: Springer, Jul. 2004.

[96] A. Sanuy, R. Casanova, M. Szymanski, H. Wörn, J. Samitier, and A. Dieguez, “Energy aware hw/sw integration in an autonomous microrobot,” in *Proc. of the 6th WSEAS International Conference on COMPUTATIONAL INTELLIGENCE, MAN-MACHINE SYSTEMS and CYBERNETICS*, 2007, pp. 226–231.

[97] A. Sanuy, M. Szymanski, R. M. Casanova, A. Dieguez, J. Samitier, and H. Wörn, “Energy efficient hw/sw integration in an autonomous microrobot,” in *Proc. of the 12th World Multi-Conference on Systemics, Cybernetics and Informatics*, 2008.

[98] F. Schlachter, E. Meister, S. Kernbach, and P. Levi, “Evolve-ability of the robot platform in the symbrion project,” in *Proc. of the 2nd IEEE Int. Conf. on Self-Adaptive and Self-Organizing Systems (SASO), Workshop on Pervasive Adaptive Systems*. Venice, Italy: IEEE Press, 2008.

[99] T. Schmickl, R. Thenius, C. Möslinger, G. Radspieler, S. Kernbach, M. Szymanski, and K. Crailsheim, “Get in touch: Cooperative decision making based on robot-to-robot collisions,” *Autonomous Agents and Multi-Agent Systems*, Aug. 2008.

[100] J. Seyfried, “Planungs- und Steuersysteme für die Mikromontage mit Mikrorobotern,” Ph.D. dissertation, Universität Karlsruhe (TH), 2003.

[101] J. Seyfried, M. Szymanski, N. Bender, R. Estana, M. Thiel, and H. Wörn, “The I-SWARM Project: Intelligent Small World Autonomous Robots for Micro-manipulation,” in *Swarm Robotics*, ser. Lecture Notes in Computer Science, E. Sahin and W. M. Spears, Eds., vol. 3342. Santa Monica, CA, USA: Springer, Jul. 2004.

[102] A. Sokolov and D. Whitley, “Unbiased Tournament Selection,” in *Proc. of the Conference on Genetic and Evolutionary Computation (GECCO).* New York, NY, USA: ACM, 2005, pp. 1131–1138.

[103] V. Sperati, V. Trianni, and S. Nolfi, “Evolving coordinated group behaviours through maximisation of mean mutual information,” *Swarm Intelligence*, vol. 2, pp. 73–95, 2008.

[104] Stephan Fahlbusch, Sergej Fatikow, Jörg Seyfried, and Axel Bürkle, “Flexible Microrobotic System MINIMAN: Design, Actuation Principle and Control,” *Proc. of the 1999 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM’99), Atlanta, GA, USA*, 1999.

[105] P. Stürzel, “Entwurf und Implementierung einer ZigBee basierten Roboter-Rechner Kommunikationssoftware,” Studienarbeit, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, Oct. 2007.

[106] M. Szymanski, L. Winkler, D. Laneri, F. Schlachter, A. C. van Rossum, T. Schmickl, and R. Thenius, “SymbricatorRTOS: A Flexible and Dynamic Framework for Bio-Inspired Robot Control Systems and Evolution,” in *Proc. of the IEEE Congress on Evolutionary Computation (CEC).* Trondheim, Norway: IEEE Press, May 2009.

[107] M. Szymanski, J. Fischer, and W. Heinz, “Investigating the Effect of Pruning on the Diversity and Fitness of Robot Controllers based on MDL2ϵ during Genetic Programming,” in *Proc. of the IEEE Congress on Evolutionary Computation (CEC)*, 2009.

[108] M. Szymanski and H. Wörn, “JaMOS – A MDL2ϵ based Operating System for Swarm Micro Robotics,” in *Proc. of the IEEE Swarm Intelligence Symposium (SIS 2007).* Honolulu, HI, USA: IEEE Press, 2007, pp. 324–331.

[109] G. Theraulaz, E. Bonabeau, and J.-L. Deneubourg, “Response threshold reinforcement and division of labour in insect societies,” in *Proc. of the Royal Society B: Biological Sciences*, vol. 265, 1998, pp. 327–332.

[110] J. Timmis, P. Andrews, N. Owens, and E. Clark, “An interdisciplinary perspective on artificial immune systems,” *Evolutionary Intelligence*, vol. 1, no. 1, pp. 5–26, Mar. 2008.

[111] V. Trianni, R. Gross, T. Labella, E. Sahin, P. Rasse, J. Deneubourg, and M. Dorigo, “Evolving Aggregation Behaviors in a Swarm of Robots,” in *Proc. of the European Conference on Artificial Life*, ser. Lecture Notes in Artificial Intelligence. Berlin, Germany: Springer Berlin/Heidelberg, 2003.

[112] V. Trianni, S. Nolfi, and M. Dorigo, “Cooperative hole-avoidance in a swarm-bot,” *Robotics and Autonomous Systems*, vol. 54, no. 2, pp. 97–103, Feb. 2006.

[113] M. Waldenberger, “Simulation von Infrarot Sensoren unter Breve,” Studienarbeit, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, Dec. 2006.

[114] D. Whitley, “The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best,” in *Proc. of the Third International Conference on Genetic Algorithms*, J. D. Schaffer, Ed. San Mateo, CA: Morgan Kaufman, 1989.

[115] U. Wilensky, “NetLogo Ants Model,” Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL, http://ccl.northwestern.edu/netlogo/models/Ants, 1997.

[116] ——, “NetLogo,” Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL, Tech. Rep., 1999.

[117] L. Winkler, “Stigmergie und Lokale Kommunikation in einem Roboterschwarm,” Diplomarbeit, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, Mar. 2007.

[118] L. Winkler and H. Wörn, “Symbricator3D – A Distributed Simulation Environment for Modular Robots.” in *Proc. of the Int. Conf. on Intelligen Robotics and Applications*, ser. Lecture Notes in Computer Science, M. Xie, Y. Xiong, C. Xiong, H. Liu, and Z. Hu, Eds., vol. 5928. Singapore: Springer, 2009, pp. 1266–1277.

[119] L. Wirzenius and K. Oksanen, “Hedgehog: A Tiny LISP for Embedded Applications,” in *Free and Open Source Developer’s European Meeting*, Brussels, Belgium, Feb. 2005.

[120] H. Wörn, F. Schmoeckel, A. Bürkle, J. Samitier, M. Puig-Vidal, S. Johansson, U. Simu, J.-U. Meyer, and M. Biehl, “From decimeter- to centimeter-sized mobile microrobots: the development of the MINIMAN system,” *SPIE’s Int. Symp. on Intelligent Systems & Advanced Manufacturing, Conference on Microrobotics and Microassembly, Boston, MA, USA, October 28 - November 2, 2001*, 2001.

[121] H. Wörn and J. Seyfried, “Distributed autonomous micro robots: from small clusters to a real swarm,” *Proc. of the 7th International Symposium on Distributed Autonomous Robotic Systems (DARS)*, pp. 314–350, 2004.

MARC SZYMANSKI

ENTWICKLUNGSUMGEBUNG FÜR ROBOTERSCHWÄRME

ISBN 978-3-86644-619-9