Diplomarbeit zum Thema

„Automatische Generierung von Bewegungsmustern für reale Roboter mit Evolutionären Algorithmen“

zur Erlangung des akademischen Grades
Diplom-Informatiker(FH)

vorgelegt dem
Fachbereich Informatik und Medien der Fachhochschule Brandenburg

Maurice Hüllein
29. November 2006

Erstprüfer: Dipl.-Inform. Ingo Boersch
Zweitprüfer: Prof. Dr.-Ing. Jochen Heinsohn

Inhaltsverzeichnis

Kapitel 1.

Einführung

„And as natural selection works solely by and for the good of each being, all corporeal and mental endowments will tend to progress towards perfection.“
(Charles Darwin, Origin of Species)

Eine grundlegende Herausforderung der Informatik ist, Systeme zu schaffen, welche sich selbst an die Gegebenheiten eines Problems anpassen. Gerade in der heutigen technik- und informationsgestützten Zeit bieten sich zunehmend Aufgaben, die nicht mehr manuell mit den Methoden der herkömmlichen Algorithmik gelöst werden können. Dies betrifft insbesondere den Bereich der autonomen und mobilen Systeme, deren Ziel es ist, künstliche Agenten zu schaffen, die sich unabhängig von einer externen Steuerung in einer unbekannten Umwelt bewegen und mit dieser interagieren können. Die Motivation hierfür ist mannigfaltig: Autonome Systeme können Arbeiten übernehmen, die den Menschen überfordern oder für ihn zu gefährlich sind, wie beispielsweise Notfallbergung und Einsätze in Kriegsgebieten. Zudem können angepasste Morphologien von Robotern auch in Bereiche vordringen, die sonst unzugänglich wären (Kanalsystem, U-Boote etc.). Dabei ist es häufig nicht möglich, in direktem Kontakt mit dem Agenten zu stehen, um ihm Weisungen für den weiteren Verlauf des Einsatzes zu geben. Dies betrifft alle Anwendungsgebiete in denen eine Kommunikation zu unsicher ist oder nicht mehr in ausreichender Zeit übermittelt werden kann (Unterwassereinsätze, extraterrestrische Orte).
Bei den mobilen und autonomen Systemen stehen somit zwei Fragen im Vordergund: Wie kann ein Bewegungsmuster erstellt werden, welches auch einer komplizierten Robotermorphologie eine effektive Bewegung ermöglicht? Und wie kann der Agent Lernen, ohne Hilfe in einer Umwelt mit wechselnden Bedingungen seine Aufgabe zu erfüllen?
Wird ein Fokus auf die erste Frage geworfen, lohnt es sich, einen Blick auf die höchst effektiven Lösungen der Natur zu lenken. Im Zuge einer fortlaufenden Evolution sind Lebewesen entstanden, die perfekt auch an schwierigste Lebensbedingungen angepasst sind. Der Technik ist es dabei in den meisten Bereichen nicht annähernd gelungen, die Leistungsfähigkeit dieser Lösungen zu erreichen. Wird beispielsweise die Fortbewegung vieler Säugetiere betrachtet, ist zu erkennen, dass elegante und genaue Bewegungsabläufe

auch bei hoch komplexen Morphologien möglich ist.
Aus diesem Grund sollen in diesem Rahmen die Evolutionären Algorithmen und im Speziellen die Genetische Programmierung untersucht werden, die das Prinzip der Evolution zu imitieren versuchen. Die Entwicklung eines Bewegungsmusters wird hierbei als eine Optimierungssuche in einem Raum aller möglichen Lösungen gesehen. Jede Lösung wird dabei als ein Individuum einer Population von Steuerungsprogrammen dargestellt, welche in einem gegenseitigen Wettstreit um die Belohnung einer Fitnessfunktion stehen.

1.1. Aufgabenstellung

Ziel der Arbeit ist die Untersuchung einer automatischen Erzeugung von Bewegungsmustern für gegebene Robotermorphologien am Beispiel des ELFE-Laufroboters. Hierzu ist der Stand der Forschung auf dem Gebiet des maschinellen Lernens sensomotorischer Rückkopplungen mit Gütefunktional, die möglichen Repräsentationen von Bewegungsmustern und deren Lernalgorithmen im Überblick darzustellen.
Der Schwerpunkt der Arbeit liegt in der Untersuchung der Anwendung von Evolutionären Algorithmen, insbesondere Genetischem Programmieren auf die Problemstellung. Die gewonnen Erkenntnisse sind in ein prototypisches System mit einem realen Roboter umzusetzen. Hierbei ist auf Modularisierung und die Möglichkeit zur Weiterentwicklung Wert zu legen. Wenn möglich, sollten Hypothesen über den Evolutionsverlauf am realen System aus einer Simulation abgeleitet und im Experiment evaluiert werden.

1.1.1. Aufbau der ELFE

Die *ELFE* (Electronic Lifeform Fabricated for Exploration) ist ein elfbeiniger autonomer Schreitroboter, der bewusst für das Testen automatisch generierter Steuerprogramme entworfen wurde. Der Roboter besteht aus zwei übereinanderliegenden, runden Plastikscheiben (Durchmesser: 30 cm), zwischen denen elf Servomotoren radial in gleichmäßigen Abständen angeordnet sind. An jedem Motor befindet sich ein gelenkloses Bein bestehend aus einem 8 cm langen Plastikzylinder. Auf diese Weise ergibt sich für jedes Bein ein Freiheitsgrad mit einem Rotationsvermögen von etwa 180°.
Die Steuerung wird von einem AKSEN-Board (*AktorSensor*, [Boe00]) übernommen, welches auf der Oberseite des Roboters befestigt ist. Dabei handelt es sich um einen an der FH Brandenburg entwickeltes Mikrocontroller-Board der Familie SAB-C515A (8 bit) mit 12 MHz. Da das AKSEN-Board den notwendigen Strom über ein Kabel bezieht, ist es in seiner Laufzeit unbegrenzt. Zur Messung einer Fortbewegung ist am Roboter eine horizontal fixierte Maus angebracht, die ihre Positionsveränderungen über ein weiteres Kabel an den PC überträgt. Ein Datenaustausch zwischen Basisstation und dem AKSEN-Board kann über eine Bluetooth-Verbindung stattfinden.

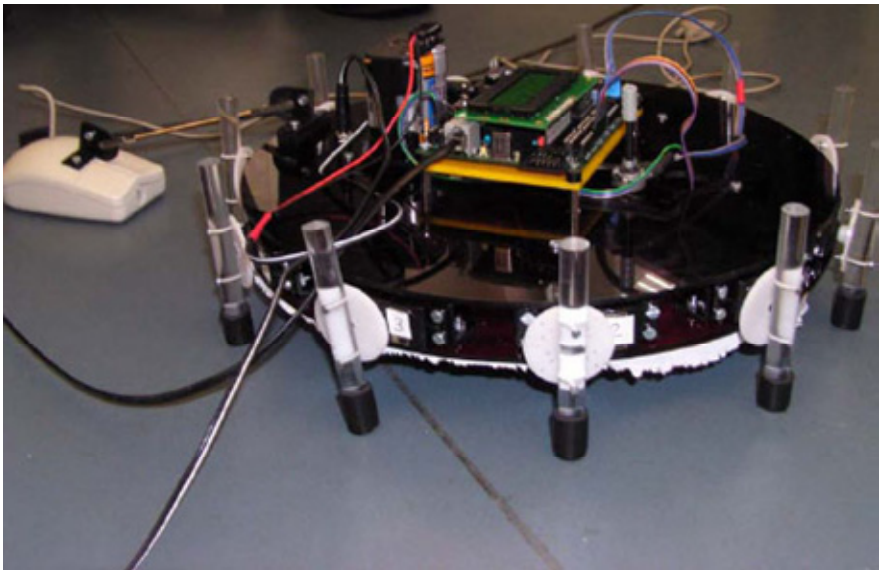

Abbildung 1.1.: Der ELFE-Schreitroboter

1.2. Gliederung der Arbeit

Der erste Teil der Arbeit bildet die theoretischen Grundlagen zum Gesamtverständnis der Arbeit. Dazu widmet sich das zweite Kapitel der Frage, auf welche Weise Bewegungsmuster dargestellt werden können. Es wird Annäherung an den Begriff des Wissens unternommen, um darauf basierend Kriterien zu formulieren, die bei der Auswahl einer geeigneten Repräsentation helfen. Desweiteren werden verschiedene Wege zur Darstellung von Bewegungsmustern besprochen und mit Beispielen aus der Forschung ergänzt. Das Kapitel soll insbesondere die Vielfältigkeit der möglichen Lösungswege andeuten.
Im dritten Kapitel wird in das Thema des Maschinellen Lernens eingeführt. Nach einer Betrachtung einiger Definitionsversuche werden Lernaufgaben anhand ihrer Eigenschaften klassifiziert und näher beschrieben. Um die Motivation zur Verwendung maschineller Lernmethoden zu erläutern, wird ein Vergleich zur klassischen deduktiven Algorithmik angestrebt. Nach einer Vertiefung des Reinforcement Learning wird zur Klasse der Evolutionären Algorithmen übergeleitet. Dort wird ein Überblick der wichtigsten Zweige dieses Forschungsbereiches gegeben und deren Anwendbarkeit auf Problemefälle abgegrenzt.
Das vierte Kapitel stellt die Genetische Programmierung als einen besonderen Vertreter der Evolutionären Algorithmen vor. Dafür wird zunächst die Motivation hinter der Schaffung der Genetischen Programmierung betrachtet.

Mit dem fünften Kapitel beginnt der Hauptteil der Arbeit. Als Einführung wird die Aufgabenstellung analysiert und Indizien zur Anwendbarkeit von Methoden des Maschinellen Lernens, insbesondere der Genetischen Programmierung aufgezeigt. Darüber hinaus wird das Ziel der Arbeit im Detail neudefiniert und ein Überblick über die ge-

schaffene Systemarchitektur gegeben.
Die folgenden Unterkapitel stellen die Komponenten des hybriden Evaluationssystem dar. Als erstes wird in 5.2 der GP-Kernel ECJ vorgestellt. Nach einer Erläuterung der Auswahlkriterien werden die Architektur und Arbeitsweise des Frameworks sowie alle Erweiterungen erklärt, die im Rahmen dieser Arbeit entstanden sind.
Der Abschnitt 5.3 zeigt die verwendete Möglichkeit zur Repräsentation von Genetischen Programmen in einer konkreten Implementation. Zusätzlich werden zwei alternative Wege zur Ausführung dieser Strukturen aufgezeigt und deren Vor- und Nachteile gegeneinander abgewägt.
In Abschnitt 5.4 wird der Schwerpunkt des praktischen Teils besprochen, die Arbeit mit dem Simulationssystem BREVE. Es wird dargelegt, welchen Nutzen ein Simulator bei der Arbeit mit Evolutionären Algorithmen bringen kann und in wie fern BREVE als geeignete Wahl zur Modellierung der ELFE erschien. Nachdem die Funktionsweise des Simulationssystems vorgestellt worden ist, wird die Modellierung der ELFE als 3D-Agent der virtuellen Welt beschrieben. Eine nachfolgende Einführung in die Entwicklung von BREVE-Plugins schafft die Voraussetzungen zum Verständnis der Simulatorerweiterungen. Auf diese Weise wird bei der

Das sechste Kapitel beschreibt die Durchgeführten Experimente mit dem geschaffenen System anhand verschiedener Aufgaben. Es klärt in einem ersten Schritt, ob die Funktionstauglichkeit des Systems gewährleistet ist und ein evolutionärer Zyklus zustande kommt. Darauf folgend wird die Entwicklung eines Fortbewegungsmusters für das ELFE-Modell untersucht.
Abschließend werden die Ergebnisse und Probleme der Arbeit noch einmal zusammengefasst und mit den ursprünglichen Erwartungen verglichen. Zudem wird ein Ausblick gegeben, wie die Arbeit weitergeführt werden könnte.

Kapitel 2.

Wissensrepräsentationen von Bewegungsmustern

Innerhalb der Informationstechnik werden Daten als Sachverhalte, Aussagen und Fakten verstanden, welche insbesondere für die maschinelle Verarbeitung leicht zu strukturieren und zu erfassen sind. Informationen repräsentieren zusätzlich eine Bedeutungsebene, indem sie Daten in einen speziellen Kontext setzen und sie somit einem Zweck zuordnen. So ist beispielsweise eine reine Zeitangabe, wie 11.53 h, ein Datum. Ist jedoch bekannt, dass es sich dabei um die aktuelle Zeit in Tokio handelt, wird dieses Datum zur Information.
Bei einer Vielzahl von Problemstellungen sind Unmengen an verschiedenen Informationen verfügbar. Die Herausforderung besteht darin, unter der Last der Informationsmenge nur diejenigen herauszufiltern, welche für die Betrachtung des Problems relevant sind. Alle anderen, unwichtigen Details müssen ignoriert werden, damit sie die Sicht auf eine Lösung nicht versperren. Aus dieser Einsicht leitet sich die Definition von Wissen ab, wie sie in dieser Arbeit verwendet werden soll. Besonders intuitiv wird sie formuliert in [HSA99, S. 1]:

> „Wissen ist Information, die in einer bestimmten Situation für eine bestimmte Person sinnvoll verwendet werden kann, Information also, die einen Nutzen bringt."

Bezogen auf informationstechnische Fragestellungen bedeute dies genauer, dass Wissen Informationen seien, welche zur Lösung eines Problems genutzt werden könnten.

2.1. Wissensrepräsentation

Soll ein Problem mit Hilfe eines Computers gelöst werden, ist eine der ersten zu beantwortenden Fragen, wie das Wissen um das Problem formal repräsentiert werden kann.

Hierfür stehen meist eine Vielzahl an Möglichkeiten zur Verfügung. Jede von ihnen stellt eine Abstraktion der Anwendungsdomäne dar, indem ausgewählte Bestandteile modelliert, andere hingegen ausgeschlossen werden. Abbildung 2.1 zeigt den prinzipiellen Vorgang bei der Überführung einer Anwendungsdomäne in eine Wissensrepräsentation. Der Kreis deutet das Wissen über die Anwendungsdomäne an. Dieses ist häufig schlecht definiert und nur mit Hilfe eines Experten zu verwenden. Auf der rechten Seite befindet sich das Metamodell. Es verkörpert die formale Sprache, welche für die Repräsentation des Wissens benutzt wird. Beide Elemente verschmelzen in der Phase des Wissenserwerbs, hier wird das Wissen über die Anwendungsdomäne in der Sprache des Metamodells ausgedrückt. Als Resultat entsteht das Modell der Domäne.

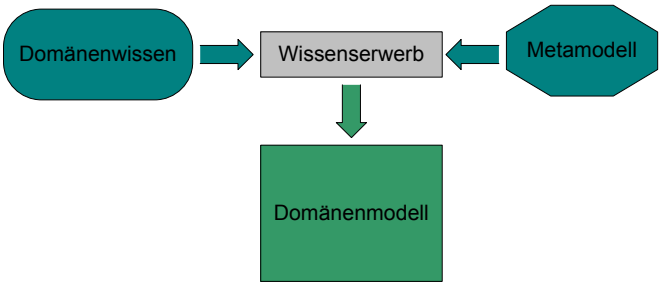

Abbildung 2.1.: Überführung der Anwendungsdomäne in die Wissensrepräsentation

Allerdings eignet sich nicht jede Darstellungsform gleichermaßen zur Lösung eines Problems. Dies kann mit einem einfachen Beispiel verdeutlicht werden (Vgl.: [GRS03, S. 39]). Die folgende Multiplikation ist mit einer einfachen schriftlichen Rechnung zu bewältigen, da die Stellwertigkeit des Dezimalsystems als Hilfe genutzt werden kann:

43 mal 118

Wird die gleiche Aufgabe im Römischen Additionssystem dargestellt, erweist sich die Aufgabe als wesentlich schwieriger:

XLIII mal CXVIII

Ein Algorithmus zur Findung einer optimalen Wissensrepräsentation existiert nicht. Vielmehr ist die Güte einer Darstellungsform abhängig von der Natur des Problems und den individuellen Anforderungen, die an sie gestellt werden. Dennoch können allgemeine Kriterien genannt werden, um losgelöst vom Problem ein Urteil zu erlauben. Eine Möglichkeit bietet [HSA99, S. 3ff]:

Vollständigkeit Modelliert die Wissensrepräsentation genügend Wissen, damit eine Lösung überhaupt gefunden werden kann?

Abstraktion Werden nur so viele Elemente repräsentiert, wie für die Lösung des Problems notwendig sind?

Ökonomie Wird das Wissen im Rahmen der Repräsentation möglichst kompakt ausgedrückt?

Freiheit von Redundanz Werden alle Elemente ausgeschlossen, die abgeleitet werden können oder bereits in anderer Form vorhanden sind?

Transparenz Wird das Wissen in möglichst verständlicher Form ausgedrückt?

Sind über die Anwendungsdomäne, für die eine Wissensrepräsentation gelten soll, Details bekannt, können die Anforderungen erweitert und genauer spezifiziert werden. Bezogen auf die Darstellung von Bewegungsmustern stellen sich so zusätzliche Fragen, wie nach der Art der Fortbewegung, ob die Muster manuell implementiert oder trainiert werden sollen, inwiefern Interaktion durch Sensorik von Interesse ist etc. Das zusätzliche Wissen um die Implementierung kann den Kriterienkatalog um folgende Aspekte erweitern:

Performanz Mit welchem Kostenaufwand ist die Bewegung darstellbar bzw. berechenbar?

Leistung Führt die Darstellung zu einer möglichst effizienten (energiesparenden und direkten) Bewegung?

Erweiterbarkeit Können der Repräsentation leicht neue Elemente hinzugefügt werden?

Interaktion Müssen Umwelteinflüsse (eventuell in Realzeit) verarbeitet werden können?

Entwicklung Werden die Muster manuell erstellt oder liegt ihnen ein Lernalgorithmus zugrunde?

Besonders in realen Umgebungen ist es faktisch unmöglich, allen Kriterien gleichermaßen gerecht zu werden. Vielmehr wird deutlich, dass einige von ihnen sogar im direkten Widerspruch zueinander stehen können: Eine möglichst redundantfreie Darstellung kann schwieriger zu berechnen sein, als eine, in der komplexe Berechnungen bereits mit abgebildet werden und sofort verfügbar sind (Vgl.: [HSA99, S. 4]). In den meisten Fällen wird jedoch abwägbar sein, welche Kriterien in den Hintergrund treten können, um sich auf Kernpunkte der Darstellungsform zu konzentrieren. Beispielsweise kann die Transparenz bei automatisch generierten Mustern nebensächlich sein, solange keine manuellen Korrekturen durch Menschen vorgesehen sind. Auch kann für ein Robotermodell, welches ausschließlich eine schnelle Bewegung ausführen soll, auf Erweiterbarkeit verzichtet werden, während Performanz und Leistung eine übergeordnete Rolle spielen.

Im Folgenden soll eine exemplarische Auswahl an Wissensrepräsentationen vorgestellt werden, mit deren Hilfe Bewegungsmuster dargestellt werden können. Anhand von Beispielen aus der Forschung werden die Grenzen und Möglichkeiten der Repräsentationformen näher betrachtet.

2.2. Endlicher Zustandsautomat

Ein Endlicher Zustandsautomat ist ein mathematisches Modell zur Beschreibung einer Rechenmaschine, bestehend aus einer endlichen Menge von Zuständen, Übergangsrelationen und einem Eingabealphabet. Beginnend bei einem definierten Startzustand liest er schrittweise eine endliche Symbolfolge aus dem Eingabealphabet ein und wechselt anhand der Transitionsbedingungen in einen seiner Folgezustände. Kann ein Endlicher Automat bei jedem Eingabesymbol nur in einen möglichen Folgezustand wechseln, so wird er als deterministisch bezeichnet. Stehen hingegen mehrere Folgezustände zur Auswahl, heißt er nicht-deterministisch.
Grundsätzlich werden Endliche Automaten in zwei Kategegorien unterteilt. Die *Akzeptoren* ordnen ein Eingabewort einer Sprache zu, falls es sie in einen definierten Endzustand überführt. *Transduktoren* hingegen erzeugen Ausgaben anhand von Aktionen. Der sogenannte *Moore*-Automat ist ein Transduktor, welcher seine Ausgabe anhand seines aktuellen Zustandes erzeugt. Eine Alternative ist der *Mealy*-Automat. Seine Ausgabe hängt von dem Zustand in Kombination mit dem Eingabesymbol ab. Moore-Automaten gelten als leichter verständlich, benötigen in vielen Fällen jedoch mehr Zustände als Mealy-Automaten. Da beide Automaten gleichmächtig sind, können sie ineinander überführt werden. Transduktoren werden aufgrund ihrer Fähigkeit, eine Ausgabe zu bestimmen, häufig als Steuerungselemente eingesetzt. Deren Sinn besteht nicht primär im Parsen eines Eingabewortes, sondern in der Erzeugung von Kontrollbefehlen anhand des aktuellen Status.

Ein Beispiel für die Verwendung von Zustandsautomaten zur Realisierung von Bewegungsmustern ist die Arbeit von Rodney Brooks [Bro89]. Brooks entwarf für die Steuerung eines sechsbeinigen Schreitroboters ein Netzwerk aus sogenannten *Augumented Finite State Machines (AFSM)*. Jeder dieser Zustandsautomaten ist eine unabhängig funktionierende Einheit und realisiert genau eine Aufgabe, wie die Steuerung eines Gelenkmotors für die Positionierung eines Beins oder die Koordination von Beinbewegungen untereinander. Damit AFSMs sowohl autark als auch in Zusammenarbeit arbeiten können, wurde die Idee eines Zustandsautomaten um eine interne Uhr, zusätzliche Register und Verbindungsleitungen erweitert. Durch die Uhr ist es möglich, zeitgesteuerte Zustandsänderungen des Automaten zu bewirken. Darüber hinaus können sich AFSM gegenseitig Nachrichten in den Registern hinterlassen oder andere Leitungen blockieren, vorausgesetzt, sie sind untereinander verbunden. Der schematische Aufbau einer AFSM ist in Abb. 2.2 zu sehen.

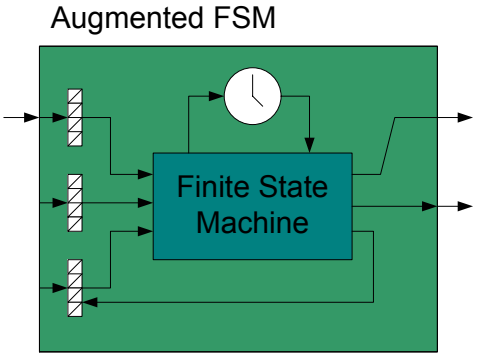

Abbildung 2.2.: Schematischer Aufbau eines AFSM

Brooks betont, ein entscheidender Vorteil dieser Architektur sei, dass Verhaltensweisen eines Roboters schrittweise und modular aufgebaut werden könnten. Als einfachstes Beispiel zeigt er in seiner Arbeit, wie ein übergeordnete Funktion *StandUp* für den Schreitroboter realisiert wurde. Hierzu waren zwei AFSMs pro Bein nötig, eine für die Einstellung der horizontalen Position (*alpha Pos*) und eine für die vertikale (*beta Pos*). Beide Stellwerte wurden initial als Eingabedaten in ein Register geschrieben, der Zustandsautomat war dafür verantwortlich, die Motoren entsprechend zu bewegen. Die aktuelle Position des Motors während der Bewegung stellte die Ausgabe dar.
Die Abb. 2.3 zeigt ein komplexeres Beispiel eines AFSM-Netzwerkes, welche es dem Roboter ermöglichte zu laufen. Jeder Kasten symbolisiert eine AFSM, welche für eine spezielle Aufgabe verwendet wird. Die Zahlen darüber deuten an, dass alle Automaten bis auf die globalen Koordinatoren mehrfach, nämlich einmal pro Bein, vorhanden sind. Der Automat *walk* koordiniert die Abfolge aller Beinbewegungen untereinander, indem er über *up leg trigger* eine Schwingphase einleitet. Befindet sich ein Bein nicht in einer Abwärtsbewegung, versucht *leg down* diese herbeizuführen. Damit gegensätzliche Aktionen nicht gleichzeitig stattfinden können, gibt es Ausgaben, die von anderen Leitungen unterdrückt werden können (in der Abbildung mit einem *s* für *suppress* markiert). Die Automaten *alpha pos* und *beta pos* kalibrieren die Motoren auf einen gegebenen Stellwert.

Zusammenfassend kann gesagt werden, dass Endliche Automaten insbesondere von ihrer Mächtigkeit profitieren, nahezu jeden Prozess schematisch abzubilden. Wie Brooks Experiment zeigt, ist es möglich, hardwarenahe Abläufe (Kalibrierung des Motors etc.) wie auch übergeordnete Verhaltensweisen (Koordination der Beine etc.) gleichermaßen als Automat zu modellieren. Zudem ist der interne Aufbau eines Endlichen Automaten grundsätzlich für den Menschen leicht nachzuvollziehen, da jeder Zustand grafisch und im engen Zusammenhang zum Kontext des Problems dargestellt werden kann.

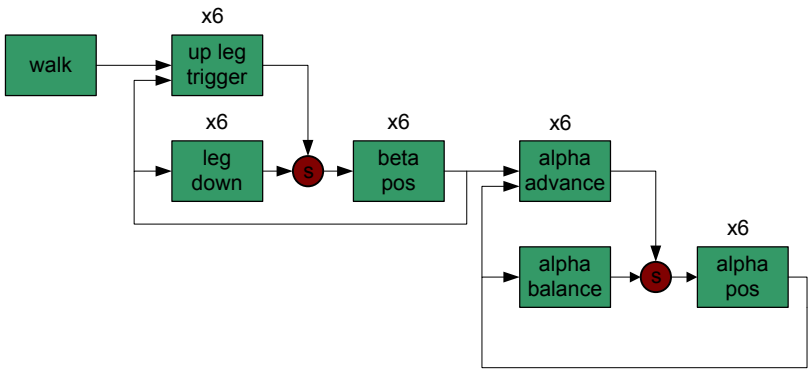

Abbildung 2.3.: AFSM Netzwerk zur Steuerung eines Ganges

2.3. Trigonometrische Funktionen

Eine weitere Möglichkeit, Bewegungsmuster darzustellen, ist die direkte Überführung von Eingabewerten zur gewünschten Bewegungsform über eine funktionale Abhängigkeit. Als Eingabe können verschiedenste Vektoren dienen, etwa ein zeitlicher Takt oder die letzte Variablenbelegung der Funktion. Die Schwierigkeit besteht darin, einen mathematischen Zusammenhang zwischen den Parametern einer Bewegungsform und einem Ausgabevektor zu finden, über den sich die Aktoren des Roboters ansteuern lassen. In einer Vielzahl von Experimenten wurden trigonometrische Funktionen, insbesondere Kosinus und Sinus, benutzt. Sie lassen sich über die geringe Anzahl von drei Parametern einstellen (Frequenz, Amplitude, Phase) und produzieren symmetrische Wellenmuster, wie sie näherungsweise auch in der Natur zu beobachten sind.
Die Arbeit [Stu02] zeigt den Zusammenhang zwischen einfachen Wellenbewegungen und dem Fortbewegungsmuster einer Raupe. Das Tier wurde mehrere Stunden gefilmt, um dessen Bewegungsstrategie für einen computersimulierten Kriechroboter nachahmen zu können. Bei der Auswertung zeigte sich, dass die Raupe eine stabile Vorwärtsbewegung erreicht, indem nacheinander kurz Beinpaare angehoben und die zugehörigen Körpersegmente kontrahiert werden (Eine Vertiefung des Themas bietet [Eps95]). Dieses Schema wird vom hinteren zum vorderen Teil der Raupe propagiert, wodurch eine Wellenbewegung entsteht.
Einen bedeutenden Schritt weiter geht das Experiment von [SSW03]. Es setzte erfolgreich die segmentbasierte Bewegung in einem realen Roboter um. Wie Abb. 2.4a) zeigt, wurden hierfür mehrere sogenannte *CONRO*-Module (Configurable Robot Modul) zu einer Kette verbunden. Jedes Modul ist ein eigenständiges Teilgelenk, welches ein Motorsystem mit zwei Freiheitsgraden (vertikale und horizontale Bewegung) ansteuern kann. Über ein gerichtetes Steckersystem können beliebig viele Module in Serie geschaltet werden (siehe Abb. 2.4b). Eine globale Steuer- oder Synchronisationseinheit wird nicht verwendet, stattdessen kann jedes Modul über Infrarotsensoren Nachrichten empfangen und senden, um Aktionen zu koordinieren.

a) b)

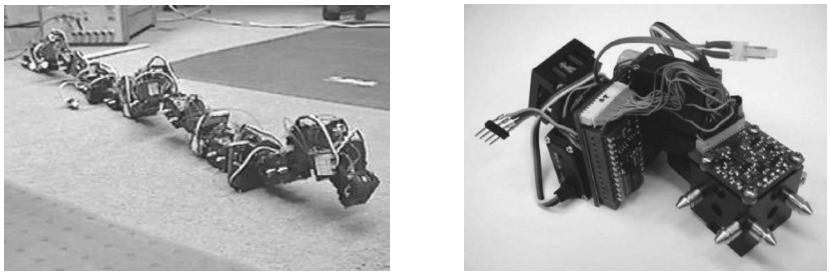

Abbildung 2.4.: a) Roboter aus zusammengeschalteten CONRO-Modulen, b) ein CONRO-Modul in Nahaufnahme

Innerhalb des Experiments wurden drei Arten der Fortbewegung nachgebildet: raupenartiges Kriechen, seitliches Schlängeln[1] und Rollen. Die verschiedenen Bewegungsformen wurden dabei durch manuelle Parametrisierung erreicht: Anhand eines internen Zählers t, der eine maximale Periodenlänge von T erreichen konnte (t = *(t+1) modulo* T), wurden die aktuellen Servokommandos der CONRO Module für die horizontale (yaw) und vertikale ($pitch$) Position berechnet. Die Taktkonstante T war für zwei Aspekte wichtig. Zum einen regelte sie die Größenordnung, um die ein Winkel inkrementiert wurde. Auf diese Weise konnte die Geschwindigkeit der Servomotoren innerhalb der Module gedrosselt werden, um eine weichere Bewegung zu erzeugen. Zum anderen wurde eine Variable d als Bruchteil von T definiert, sie markierte einen Zeitpunkt zum Senden eines Synchronisationssignals. Stimmte d mit dem aktuellen Zeittakt t überein, wurde das nächste Glied der Kette informiert, welches daraufhin seinen Zähler auf null zurücksetzte. Dieses ermöglichte gestaffelte Aktionsverläufe. Die Gleichungen 2.1 zeigen die Einstellungen des Experiments für die seitliche Schlängelbewegung. Die Raupenbewegung hingegen wurde erreicht, indem einfach die horizontalen Servos fixiert ($yaw(t) = 0$) und eine leicht veränderte Amplitude verwendet wurden.

$$\begin{aligned} T &= 180 \\ pitch(t) &= 20° sin(\frac{2\pi}{T}t) \\ yaw(t) &= 50° sin(\frac{2PI}{T}t) \\ d &= \frac{1}{5}T \end{aligned} \tag{2.1}$$

Ähnliche Arbeiten, teilweise ebenso mit polymorphen Robotern, sind unter [GGBoJ], [She01] und [Dow97] zu finden. Ein weiterer, klassischer Ansatz zur Schwingung eines Beins mit trigonometrischen Funktionen kann anhand der Programmierung des Sony Quadruped Robots, dem Prototypen des populären AIBO, nachgelesen werden [HFT+99].

[1] Zu beobachten bei Sidewinder-Klapperschlangen

Die Beispiele zeigen, dass trigonometrische Funktionen benutzt werden können, um auf kompakte Weise Bewegungsmuster darzustellen. Entscheidende Vorteile sind der geringe Modellierungsaufwand durch wenige Parameter und eine schnelle Berechenbarkeit (gegebenenfalls unter Zuhilfenahme einer Lookup-Table). Zusätzlich ist das Funktionsergebnis in den meisten Fällen leicht zu interpretieren, da es grundsätzlich ein Winkel zur Stellung eines Gelenks sein wird. Hier liegt jedoch auch der größte Nachteil: Trigonometrische Funktionen beschränken sich auf die Darstellung von periodischen Wellen. Für sich allein genommen sind sie ungeeignet, um komplexere Bewegungsmuster darzustellen oder sie untereinander zu koordinieren.

2.4. Künstliche Neuronale Netze

Ein Künstliches Neuronales Netz (KNN) ist ein Rechnersystem aus untereinander verbundenen virtuellen Prozessoren zur parallelen und dezentralisierten Informationsverarbeitung. Ähnlich ihrem biologischen Vorbild, den neuronalen Netzstrukturen innerhalb von Gehirnen, werden eigenständige Prozessorknoten, die sog. *Units*, über informationstragende Leitungen beliebig miteinander verbunden. Jeder Verbindungsleitung ist eine Wichtung zugeordnet, die Signale, welche zwischen Units ausgetauscht werden, verstärken oder hemmen kann. Eine Unit ist in der Lage, eingehende numerische Werte zu verarbeiten und an die Ausgänge weiterzureichen. Hierfür werden die gewichteten Eingaben aller vorgeschalteten Units summiert und mit Hilfe einer Transferfunktion verrechnet. Das Ergebnis bestimmt bis zur nächsten Veränderung den Zustand der Unit und wird als Erregung bezeichnet. Obwohl grundsätzlich jede Art von Transferfunktion gewählt werden kann, wird meist eine binäre Ausgabe anhand einer Schwellwertfunktion erzeugt oder die Identität einer sigmoiden Funktion verwendet.
Grundsätzlich können Units anhand ihrer Vernetzung in drei Arten von Schichten gegliedert werden: Die *Input-Units* stellen die erste Schicht dar, sie erhalten ihre Eingabe ungewichtet von außerhalb des Systems und reichen sie an andere Units weiter. Dem gegenüber stehen die *Output-Units* der letzten Schicht. Ihre Eingabe kommt aus dem Inneren des Netzes und wird nach Außen geleitet. In den beliebig vielen Zwischenschichten befinden sich *Hidden-Units*. Sowohl ihre Eingabe- als auch ihre Ausgabeleitungen sind mit anderen Units verbunden. Abb. 2.5 zeigt diese Vernetzung aller Unit-Arten über mehrere Schichten. Eine Ausnahme des Schichtenmodells bilden Netzwerke mit sogenannten *shortcut connections*, welche eine ebenenübergreifende Verknüpfung der Units erlauben.
Handelt es sich bei einem KNN um einen azyklischen, gerichteten Graphen, wird von einem Feedforward-Netz gesprochen. Dieses arbeitet wie ein assoziativer Speicher und erzeugt seinen Ausgabevektor ausschließlich in Abhängigkeit von der Belegung der Input-Schicht. Rekurrente Netze hingegen erlauben Verbindungen einer Unit auf sich selbst (indirekte Rückkopplung) mit einer der vorhergehenden Schichten (direkte Rückkopplung) oder innerhalb einer Ebene (laterale Rückkopplung). Auf diese Weise erreicht das Netz

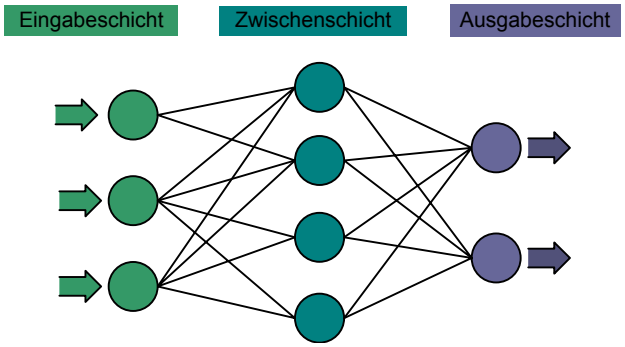

Abbildung 2.5.: Künstliches Neuronales Netz mit drei Schichten

ein wesentlich komplexeres Verhalten, da es in der Lage ist, interne Zustände zu berücksichtigen. Ein KNN modelliert sein Wissen durch die gezielte Wahl der Netzstruktur. Sowohl die Anzahl der Knoten und deren Vernetzung wie auch die Einstellung der Verbindungswichtungen und Transferstrategien bestimmen entscheidend, wie eine Eingabe in einen Ausgabevektor transformiert wird.

Der Versuch [LFB93] zeigt, wie Bewegungsmuster für Roboter anhand Künstlicher Neuronaler Netze umgesetzt werden können. Bei dem Roboter handelte es sich um einen schematischen Nachbau von Rodney Brooks *Genghis* (Vgl.: [Bro89]), einem Hexapoden mit je zwei Freiheitsgraden pro Bein. Ziel des Versuchs war es, ein Künstliches Neuronales Netz zu entwickeln, welches die Beine des Roboters so ansteuert, dass eine vorwärtsgerichtete Bewegung entsteht. Die Wahl der Bewegungsrepräsentation wurde hauptsächlich durch zwei Kernaspekte motiviert. Zum einen könnten Künstliche Neuronale Netze Differenzialgleichungen ebenso mächtig abbilden wie Zustandsautomaten. Zum anderen seien KNN die natürliche Darstellungsform von Bewegungsmustern bei Tieren, dies sei als Beweis für ihre Funktionstauglichkeit zu werten. Insbesondere das biologische Vorbild vom sogenannten *Central Pattern Generator* (CPG) wurde als strukturelle Grundlage dieser Arbeit verwendet. CPGs sind neuronale Systeme, welche rhythmische Bewegungen, ähnlich einem Oszillator, erzeugen und steuern können. Diese kommen beispielsweise bei der Flossenbewegung von Fischen oder dem Flügelschlag von Vögeln zum Ausdruck.
Innerhalb des Experiments wurden je zwei Units wechselseitig miteinander verbunden, um ein Bein zu kontrollieren. Dabei beeinflusste der Zustand der einen Unit die Vorwärtsbewegung, der Zustand der anderen die Höhe eines Beines.

Abhängig von der Wahl der Wichtungen und des Schwellwertes einer Unit wird eine Schwingung mit bestimmtem Frequenz- und Phasengang erreicht. Auf diese Weise laufen die Wellensignale in Phase, wenn die Units über positiv gewichtete Leitungen miteinander verbunden werden. Sind beide negativ, findet eine Phasenverschiebung um 180° statt. Die dritte Variante, bei der nur eine Leitung negativ ist, wurde benutzt, um

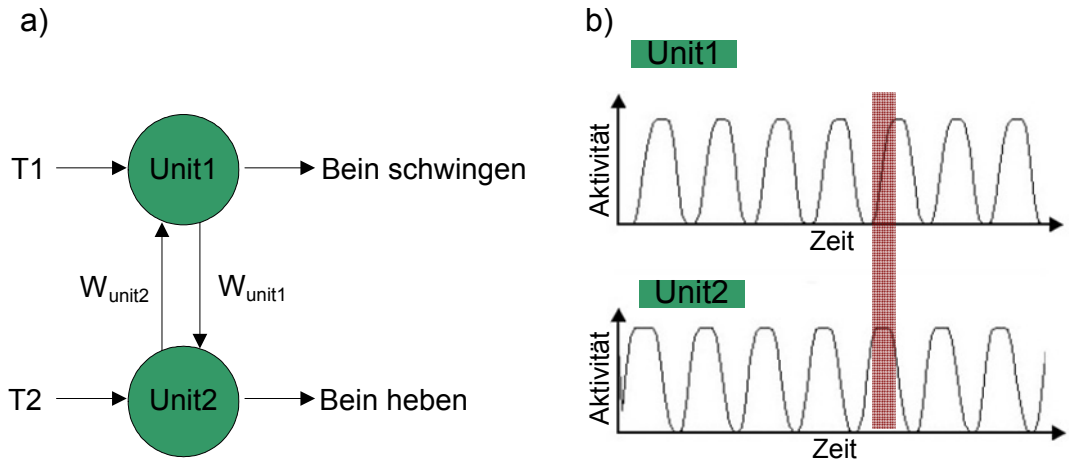

Abbildung 2.6.: a) Central Pattern Generator zur Bewegung eines Beines, b) Neuronale Aktivierungen beider Units über die Zeit

für eine flüssige Schwingbewegung des Beines beide Signale um 90° versetzt laufen zu lassen. Abbildung 2.6a zeigt den Aufbau eines neuronalen Oszillators mit den Verbindungswichtungen W_{Unit1} und W_{Unit2} sowie den Schwellwerten *T1* und *T2*. Die um 90° versetzte Aktivität beider Neuronen wird in den Grafen der Abbildung 2.6b dargestellt. Die eigentliche Bewegung des Roboter kam zustande, indem die Ausgabe des Netzes auf eine Pulsweitenmodulation der Servomotoren umgerechnet wurde. Um die einzelnen Beine untereinander zeitlich koordinieren zu können, mussten die einzelnen Oszillatoren in einen wechselseitigen Kreislauf geschaltet werden.
Da auch in dieser Verwendung von KNN die Findung passender Wichtungen und Schwellwerte ein zentrales Problem darstellte, wurde ein Genetischer Algorithmus eingesetzt, welcher das CPG in zwei Schritten evolvierte: Die erste Phase ermittelte die Konfiguration der einzelnen Oszillatoren, die zweite erzeugt die Parameter, um sie untereinander zu koordinieren.

Der vorgestellte Versuch ist nur eines von zahlreichen Beispielen, um KNN erfolgreich zur Darstellung von Bewegungsmustern zu nutzen (weitere: [IHW98], [KeioJ], [CNB+oJ], [HS02]). Als wichtigste Motivation wird dabei die Adaption von bereits funktionierenden biologischen Systemen genannt (Vgl. [RIP+02]), welche extremes Leistungspotential deutlich machen. Ein besonders interessanter Aspekt ist, dass in den meisten der genannten Versuche die KNN nicht in ihrer traditionellen Verwendung als Klassifizierer, sondern als CPGs gebraucht werden.
Der entscheidende Nachteil zu klassischen Repräsentationsformen ist, dass ein KNN nur in wenigen und besonders einfachen Fällen manuell entworfen werden kann. Meist ist die Abstraktion zwischen dem darzustellenden Wissen und dem komplexen mathematischen Modell des Netzes zu gravierend, als dass eine Topologie intuitiv ersichtlich sein könnte. Daher werden KNN in der Regel nach Erfahrungswerten oder maschinell aufgebaut und über Lernalgorithmen für ihre Aufgabe trainiert.

2.5. Lineare Darstellung

Die Lineare Darstellung ist eine sequentielle Abfolge von Anweisungen, wie sie in den meisten Programmiersprachen wie *C* oder *Assembler* vorkommt. Abhängig von der Mächtigkeit der zugrundeliegenden Sprache werden die Befehle zur Ansteuerung der Bewegungsaktoren nacheinander formuliert und bei der Ausführung von der ersten bis zur letzten Zeile abgearbeitet. Die Eingaben des Systems werden über einen globalen Speicher bereitgestellt. Populäre Vertreter der linearen Datenrepräsentation sind Zweiadress-Registermaschinen (Vgl.: [BNKF97, S. 114ff]). Wie in Abb. 2.7 zu sehen ist, kodieren sie Anweisungen, welche maximal zwei Operanden über eine Funktion verknüpfen dürfen, in binäre Strings. Die Ergebnisse der Berechnung werden zurück in den globalen Registerspeicher geschrieben.

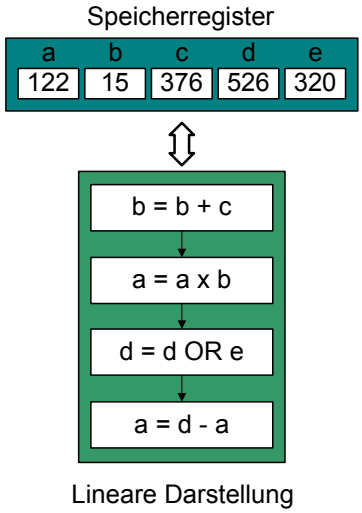

Abbildung 2.7.: Lineare Darstellung mit Registerspeicher (nach [BNKF97])

Eine besondere Anwendung der Linearen Repräsentation ist in der Genetischen Programmierung zu finden. Diese Darstellungsform ist eine Alternative zur klassischen Baumdarstellung im GP (siehe nächster Abschnitt) und wurde erfolgreich in den Experimenten von Ziegler [Zie03], Wolff [WN] und Banzhaf [BN97] eingesetzt. Alle Arbeiten verwendeten das gleiche Grundkonzept: Jede Programmanweisung durfte nur aus elementaren, ungeschachtelten mathematischen Operationen oder dem Aufruf von Steuerungsfunktionen bestehen. Zur Aufnahme der Berechnungsergebnisse wurde ein globaler Speicher genutzt, welcher aus einer begrenzten Anzahl von Registern bestand. Ziegler und Wolff generierten Programme zur Schrittsteuerung eines bipeden Roboters, indem Registerergebnisse als Winkel der Beinmotoren interpretiert wurden. Banzhaf regelte auf die gleiche Weise die Geschwindigkeit des differentiellen Antriebs eines Khepera Roboters. Zusätzlich stattete er den Roboter jedoch mit acht Infrarotsensoren aus, deren Eingabewerte in die Speicherregister weitergeleitet wurden. Dem Roboter war es somit möglich,

seine Richtungsgeschwindigkeit in Abhängigkeit der Sensorinformationen zu bestimmen und ein kollisionsfreies Fahrverhalten zu entwickeln.

2.6. Baumstrukturen

Eine Baumstruktur ist ein gerichteter und azyklischer Graph zur Darstellung von hierarchisch gegliederten Strukturen. Die eigentlichen Daten werden innerhalb des Baums als Knoten dargestellt und über Kanten miteinander vernetzt. Dabei gilt, dass genau ein Knoten ohne Vorgänger existiert (die *Wurzel*) und die übrigen Knoten nur einen Vorgänger besitzen dürfen. So ergibt sich von der Wurzel ausgehend zu jedem Knoten ein eindeutiger Pfad, dessen Länge als *Tiefe* des Knotens bezeichnet wird. Alle Knoten, die innerhalb einer Ebene liegen und einen gemeinsamen Vorgänger besitzen, werden als *Söhne* eines *Vaterknotens* bezeichnet und bilden zusammen *Geschwister*. Das Ende eines Baums wird durch die *Blätter* markiert, sie besitzen keine nachfolgenden Knoten.

In aktuellen Experimenten beschränkt sich der Einsatz von Baumstrukturen auf das Gebiet der Genetischen Programmierung. Durch den Ursprung des GP in der Programmiersprache Lisp werden Syntaxbäume, sog. *S-Expressions* benutzt, um eine Steuerung mathematisch abzubilden. Die Knoten eines Syntaxbaums bestehen aus zwei verschiedenen Knotentypen: Innere Knoten werden aus einem *Funktions-Set* gewählt. Dabei handelt es sich um parameterverarbeitenden Funktionen, wie mathematischen Ausdrücken, Sprung- oder Bedingungssanweisungen oder auch Speichermanipulationen. Die Blätter des Baumes können Variablen und Konstanten bzw. parameterlose Funktionen sein. Als Menge der *Terminale* bilden sie die Eingabe des Systems. Jeder Funktionsknoten ist in der Lage, über sog. *Seiteneffekte* mit dem umgebenden System zu interagieren. Beispielsweise können so Sensordaten ausgelesen oder Aktoren angesteuert werden.
Für die Evaluierung eines Baums wird ein Pfad zunächst bis zum Ende verfolgt. Das jeweilige Blatt wird ausgewertet und sein Ergebnis an den Vaterknoten weitergegeben. Die Berechnung endet, wenn die Wurzel erreicht ist und den Funktionswert des Baumes zurückgegeben hat. Ein Ansatz nach diesem Muster ist [LH01]. Ziel des Experimentes war, einen simulierten Roboter zu entwicklen, der in der Lage sein sollte, einer Wand zu folgen. Die virtuelle Welt bestand aus einem 16×16 Rastergitter, welches aus nicht betretbaren sowie freien Zellen aufgebaut war und eine umzäunende Außenwand besaß. Die Syntaxbäume wurden aus Knoten zur Abfrage der Sensoren, Bewegungsbefehlen sowie einem Satz einfacher mathematischer Operatoren aufgebaut. Durch die Sensoren konnte die Belegung der acht umgebenden Quadranten abgetastet werden, während die Bewegungsbefehle den Roboter um eine Zelle in eine der vier Himmelsrichtungen versetzen konnte. Die inneren Knoten bestanden aus den Funktionen *AND*, *OR*, *NOT* und einem boolschen *IF* zur Entscheidung zwischen zwei Baumpfaden.
Als Ergebnis konnte erfolgreich ein Baum erstellt werden, der eine optimale Steuerung des Roboters innerhalb von fünf Testszenarien leistete. Das Experiment wiederholte ein

älteres und sehr ähnliches Experiment von [Koz92, S. 147ff], bei der eine virtuelle Ameise ebenfalls auf einer Rasterwelt Nahrung finden sollte.

Die gezeigten Experimente zeigen zwar die Möglichkeit, mit Hilfe von Baumstrukturen Bewegungsmuster darzustellen. Dies war jedoch nicht die grundlegende Motivation der Autoren. Vielmehr sollte der erfolgreiche Einsatz von Genetischer Programmierung demonstriert werden, die Wahl der Wissensrepräsentation wurde diesem Ziel untergeordnet.
Als Nachteile sind die hierarchische Gliederung der Organisation der Steuerungselemente zu nennen, welche mit wachsender Baumtiefe zunehmende Abhängigkeiten unter den Knoten erzeugt. Darüber hinaus ist die Mächtigkeit dieser Struktur auf rein reaktive Systeme beschränkt, da jeder Knoten nur lokal die Ergebnisse seiner Kinder speichert. Erst, wenn ein Interpreter mit einem zusätzlichen globalen Speicher die Evaluierung des Baumes übernimmt, ist das System einem Automaten gleichzusetzen.

2.7. Zusammenfassung

Das Kapitel hat eine Einführung in die Repräsentation von Wissen zur computergetützten Lösung von Problemen gegeben. Nach einem Definitionsversuch der grundlegenden Begriffe sind die Schwierigkeiten aufgezeigt worden, die bei der Überführung einer Anwendungsdomäne in eine formale Repräsentation auftauchen können. Zu diesem Zweck ist zunächst ein allgemeiner Kriterienkatalog definiert worden, anhand dessen entschieden werden kann, welche Aspekte der Repräsentation mehr oder weniger von Bedeutung für das Problem sind. Der Katalog ist dann um Fragen der Darstellung von Bewegungsmustern ergänzt worden. Um einen Einblick in die variantenreichen Herausforderungen und Lösungen zu geben, wurden diverse Beispiele aus der Forschung präsentiert, welche als Überblick in Tabelle 2.1 zusammengefasst sind.

Repräsentation	Versuch	Bewegung	Roboter
Zustandsautomat	[Bro89]	Aufstehen, laufen	Genghis Hexapod
Trig. Funktionen	[SSW03]	Kriechen, schlängeln, rollen	Modularer Roboter
	[GGBoJ]	Kriechen, schlängeln, rollen	Modularer Roboter
	[She01]	Schlängeln	Schlangenroboter
	[Dow97]	Schlängeln	Schlangenroboter
	[HFT+99]	Bein schwingen	AIBO Quadruped
Neuronale Netze	[LFB93]	Bein schwingen	Hexapod
	[IHW98]	Bein schwingen, schlängeln	Quadruped (simuliert)
	[KeioJ]	Bein schwingen	Quadruped (simuliert)
	[CNB+oJ]	Bein schwingen	Biped
Lineare Darstellung	[Zie03]	Bein schwingen	Zorc Biped
	[WN]	Bein schwingen	Biped
	[BN97]	Differentialantrieb	Khepera

Tabelle 2.1.: Überblick der Experimente zur Repräsentation von Bewegungsmuster

Kapitel 3.

Maschinelles Lernen von Bewegungsmustern

> „Find a bug in a program, and fix it, and the program will work today. Show the program how to find and fix a bug, and the program will work forever."
>
> Oliver G. Selfridge, in AI's Greatest Trends and Controversies

Soll die Technik des Maschinellen Lernens näher betrachtet werden, ist es zunächst erforderlich, den Begriff des Lernens zu umschreiben. Eine genaue und allgemein anerkannte Definition existiert nicht, da abhängig von der Zielsetzung jeweils andere Kernaspekte des Lernphänomens von Interesse sein können. Beispielsweise könnte die Modellierung menschlicher Kognitionseigenschaften untersucht werden oder die Frage im Vordergrund stehen, welche Grenzen das maschinelle Lernen zeigt (Vgl.: [GRS03, S. 518]). Aus diesem Grund sollen exemplarische Erklärungsversuche zu einem Gesamtverständnis verhelfen. Marvin Minsky [Min86] formulierte eine sehr allgemeine Beschreibung, welche einen Nutzen des Lernens betont:

> „Learning ist making useful changes in our minds."

Bei Ryszard Michalski [Mic86] hingegen spielt in der Definition von Lernen eine Zielsetzung keine Rolle:

> „Learning is constructing or modifying representations of what is being experienced."

Das Maschinelle Lernen ist der Versuch, das Konzept des Lernens zu adaptieren und es in künstlichen Systemen umzusetzen. Wird das Lernen auf diesen Kontext eingeengt, so kann angenommen werden, dass es ein definiertes Ziel gibt, welches durch Veränderungen einer internen Repräsentation erreicht werden soll. Ausgehend von diesen Aspekten bietet sich insbesondere die Definition von Tom Mitchell [Mit97] an:

> „A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.“

Maschinelle Lernalgorithmen werden abhängig von der Art wie sie Wissen erwerben in drei Paradigmen unterteilt:

- Überwachtes Lernen
- Unüberwachtes Lernen
- Bestärkendes Lernen

Beim *Überwachten Lernen* versucht der Lernalgorithmus einen funktionalen Zusammenhang zwischen Ein- und Ausgabedaten zu finden. Hierzu wird er mit einem Satz Trainingsdaten konditioniert, welche Eingaben zur Verfügung stellen, zu denen die korrekten Ergebnisse bekannt sind. Dies ist die Vorgabe, wie der Algorithmus zu reagieren hat. Der Algorithmus kann auf diese Weise seine Prognosen zu einer Eingabe mit der produzierten Ausgabe vergleichen und sich anhand der Abweichung korrigieren. Das Überwachte Lernen wird häufig mit einem Lehrer verglichen, der die Arbeit seines Schülers bewertet und eine Musterlösung bereit stellt. Wenn das System mit genügend Traningsdaten konditioniert wurde, ist es nach der *Inductive Learning Hypothesis* dann nicht nur in der Lage, bekannte Muster, sondern auch bislang unbekannte korrekt zu klassifizieren.
Im genauen Gegensatz dazu steht das *Unüberwachte Lernen*. Hier werden dem System nur Eingaben zur Verfügung gestellt, nicht aber korrespondierende Ausgabewerte oder eine allgemeine Zielfunktion. Die Aufgabe des Lernalgorithmus besteht darin, die Eingabedaten in eine interne Repräsentation zu überführen, um zusätzliches Wissen zu gewinnen. Auf diese Weise können als ähnlich definierte Datensätze gruppiert (Clusterbildung), statistische Modelle zur Vorhersage neuer Eingaben aufgebaut oder redundante Attribute zur Dimensionsreduktion entfernt werden.
Das *Bestärkende Lernen* [SB98] geht von einem System aus, welches eine Umwelt erkundet und mit dieser interagiert. Zu jedem Zeitpunkt befindet sich das System in einen definierten Zustand und wählt anhand einer internen Strategie die nächste Aktion aus. Als Rückkopplung erhält es von der Umwelt eine Belohnung, welche eine positive oder negative Bewertung der Aktion darstellt. Die Bewertung kann verzögert erfolgen und darüber hinaus von Zufallselementen abhängen. Das Ziel des Lernalgorithmus ist es, eine Strategie zu verfolgen, welche über einen längeren Zeitraum die maximale positive Belohnung verspricht. Dabei ist ihm in der Regel weder die Umgebung noch deren Belohnungsfunktionen im Voraus bekannt.

3.1. Motivation des Maschinellen Lernens

Der klassische Ansatz in der Algorithmik ein Problem zu bearbeiten besteht darin, es zunächst möglichst genau zu definieren, zu formalisieren und auf dieser Basis einen Lösungsweg zu beschreiben. Dies ist jedoch nicht immer möglich, in vielen Fällen ist die Anwendungsdomäne zu komplex, um intuitiv einen Lösungsansatz verfolgen zu können. Auch kann das vorhandene Wissen über das Problem zu gering oder potentiell falsch sein. Zwar ist es möglich, einen Experten zu Rate zu ziehen, der eine Betrachtung und Modellierung des Problems mit seinem Fachwissen und seiner Erfahrung ermöglicht. Doch selbst, wenn dieser überhaupt in der Lage ist, sein Wissen klar zu formulieren, kann dies ein zeit- und kostenintensiver Vorgang sein und führt letztendlich nur zu einer subjektiven Meinung. Darüber hinaus ist die entwickelte Lösung sehr eng an die Anwendungsdomäne gekoppelt, dass selbst kleine Veränderungen der Anforderungen den entwickelten Algorithmus unbrauchbar machen können.
Eine andere Herangehensweise ist es, den Prozess der Problemlösung als Suche in einem Raum aller möglichen Lösungen zu betrachten. Die wichtigste Frage hierbei ist, welche Strategie verfolgt wird, um eine passende Lösung zu finden. Der naivste Ansatz, das Austesten *jeder* Möglichkeit, ist in den meisten Fällen unpraktikabel, da der Suchraum unendlich groß sein kann oder selbst bei diskreter Anzahl an Lösungen der Zeitaufwand viel zu hoch wäre. Eine grundlegende Motivation des Forschungsgebiets der Künstlichen Intelligenz ist es daher, Techniken des Maschinellen Lernens anzuwenden und so den Grad an Autonomie in künstlichen Systemen zu erhöhen. Sie sollen in der Lage sein, unvollständiges Wissen zu erweitern, mit fehlerhaften Daten operieren und effizient Lösungsräume erkunden zu können. Dies beschreibt einen völlig entgegengesetztes Verfahren zur eingangs beschriebenen klassischen Algorithmik. Dort wird ausgehend von einer festgelegten und unveränderlichen Wissensbasis auf den Problemfall geschlossen (Deduktion). Das Maschinelle Lernen ist jedoch ein induktiver Prozess, bei dem eine Wissensbasis erst durch Erfahrungen geschaffen wird. Anders ausgedrückt betont das Maschinelle Lernen nicht das Programmieren, sondern das Trainieren eines Systems.

Nach Mitchell [Mit] ergeben sich drei Anwendungsgebiete für Maschinelles Lernen:

Data Mining Analysierung und Klassifizierung von großen Datenbeständen, um zukünftige Entscheidungen zu vereinfachen (z. B. Medizinische Diagnosen)

Selbstkonfigurierende Software Programme, die sich den wechselnden Bedürfnissen des Anwenders anpassen (z. B. Newsreader, welcher sich an den Interessen des Lesers orientiert)

Komplexe Softwareapplikationen Programme, die nicht mehr manuell zu erstellen sind, da die Anwendungsdomäne zu komplex ist (z. B. Autonome Fahrzeugsteuerung, Spracherkennung)

Die vorgestellten Anwendungsgebiete zeigen, dass die Wahl des Lernalgorithmus stark abhängig von der Art des Problems und dem verfügbaren Wissen ist. Grundsätzlich stellen sich die Fragen, ob es sich um eine Klassifizierungs- oder Optimierungsaufgabe handelt und falls Trainingsdaten zur Verfügung stehen, inwieweit diese die Anwendungsdomäne repräsentieren. Abhängig davon können Lernalgorithmen aus den drei Bereichen des Überwachten, Unüberwachten oder Bestärkenden Lernens gewählt werden.

3.2. Bestärkendes Lernen von Bewegungsmustern

> „(...) a robot should have a number of internal reward functions built in, rather than relying on a single external reward function. Animals for instance rely on satiating hunger to learn certain tasks rather than using the single bit of reward from death (...)“
>
> Rodney Brooks in [Bro91]

Die Entwicklung von Bewegungsmustern ist im Allgemeinen eine sehr komplexe Aufgabe. In Abhängigkeit von der physikalischen Beschaffenheit des Roboters ergibt sich die Bewegung aus einer Vielzahl von Parametern, etwa die Orientierung der Gliedmaßen, Geschwindigkeit der Motoren, zeitliche Aktivierung der Aktoren etc. Meist reicht es jedoch nicht, einfach ein Muster zur Fortbewegung zu erzeugen, vielmehr spielen weitere Fragen eine Rolle: Soll sich der Roboter möglichst stabil oder schnell bewegen? Ist es wichtig, dass er möglicht sparsam mit vorhandenen Ressourcen umgeht? Während die steigende Komplexität eines Robotermodells somit den Suchraum ausdehnt, schränkt jede zusätzliche Anforderung die Anzahl der Lösungen weiter ein.
Es gibt verschiedene Ansätze, diesem Problem zu begegnen. In der klassischen Regeltechnik wird versucht, Gleichungssysteme zu formulieren, welche die Bewegungsparameter anhand der inversen Kinematik einem Soll-Zustand angleichen. Hierfür wird der Ist-Zustand des Roboters, also die Gesamtheit der relevanten Parameter, zu einem Zeitpunkt in hohen Intervallen abgefragt und über Lösung der Differentialgleichungen korrigiert. Um solche mathematischen Modelle entwerfen zu können, bedarf es umfangreichen Wissens über das Modell. Bei mangelnder Erfahrung kann versucht werden, Bewegungsschemata aus der Natur zu adaptieren, indem sie aufgezeichnet und analysiert werden (Vgl. Abschnitt 2.3). Dieser Vorgang ist jedoch zeitaufwändig und nur möglich, wenn geeignete Messtechniken zur Verfügung stehen.
Methoden des Bestärkenden Lernens bieten einen Ansatz, um die genannten Probleme zu umgehen. Der Roboter testet sich selbst mit Hilfe des Lernalgorithmus innerhalb der Umwelt, für die er eingesetzt werden soll. Über intelligentes Ausprobieren entwickelt er über die Zeit ein Verhalten, welches seinen Erfolg maximiert. Die Abb. 3.1 zeigt diese Wechselwirkung zwischen Umwelt und Roboter (Agent): Abhängig von der Aktion a zum Zeitpunkt t erhält der Agent einen Zeitschritt später eine Belohnung b von seiner Umwelt. Daraufhin befindet er sich in einem neuen Zustand z, dem er seine nächste Aktion anzupassen versucht.

Es ist beim Bestärkenden Lernen nicht notwendig, spezielle Kenntnisse eines Prozessmodells zu besitzen oder mathematische Abhängigkeiten zu definieren. Die Zielstellung des Roboters wird einzig über das Belohnungssystem der Umwelt ausgedrückt. Als einzige Voraussetzung muss der Agent somit in der Lage sein, Rückkopplungssignale seiner Umwelt wahrnehmen zu können, auf deren Basis er die Suche des Lernalgorithmus in Richtung der gesetzten Anforderungen lenken kann.

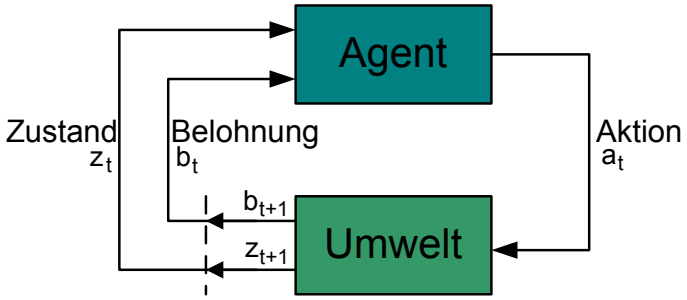

Abbildung 3.1.: Beziehung vom Agenten zur Umwelt beim Bestärkenden Lernen (nach [SB98]

Eine besondere Herausforderung dieser Art von Lernmethoden ist bekannt als das *Exploitation vs Exploration*-Problem. Es beschreibt das Verhältnis zwischen Erforschung (Exploration) des unbekannten Suchraums zur Findung von globalen Optima zum Ausbeuten (Exploitation) von gut erscheinenden und benachbarten Lösungen. Jeder Algorithmus des Bestärkenden Lernens arbeitet mit einer eigenen Strategie, um diese gegensätzliche Verhaltensweisen abzustimmen. Vorausgesetzt die Lösungslandschaft ist nicht völlig zufällig aufgebaut, führt eine Ausbeutung schnell zu besseren Lösungen. Allerdings besteht die Gefahr, in einem lokalen Minimum festzuhängen oder eine zu langwierige Suche einzugehen (Bsp.: *Optimistisches Hill-Climbing*). Die schnelle Erforschung des gesamten Suchraums hingegen wird zunächst schlechte Lösungen anbieten, aber vielleicht ein globales Optimum entdecken (Bsp.: *Monto Carlo*-Verfahren).

3.2.1. Simulated Annealing

Das Verfahren des Simulated Annealing ist ein stochastisches Optimierungsverfahren, welches insbesondere darauf abzielt, globale Maxima zu finden. Als Vorbild dient ein Phänomen aus der Natur, bei dem ein glühendes Stück Metall abgekühlt wird. Hier ist zu beobachten, dass ein langsames Senken der Temperatur den Molekülen des Metalls genügend Zeit einräumt, um eine energiearme und somit stabile Kristallstruktur zu bilden. Ähnlich diesem Prinzip ersetzt das Simulated Annealing zufällig die aktuell gewählte Lösung schrittweise mit einer im Lösungsraum benachbarten. Anders als beim Optimistischen Hillclimbing kann anhand einer sinkenden Wahrscheinlichkeit auch

eine Lösung gewählt werden, die keine sofortige Verbesserung darstellt. Die Wahrscheinlichkeit wird abhängig von der sogenannten Temperatur T gesetzt, die pro Durchlauf reduziert wird. Dies hat zur Folge, dass zu Beginn des Algorithmus der Wert von T am höchsten ist und benachbarte Lösungen nahezu zufällig gewählt werden. Mit fortschreitenden Iterationen werden zunehmend nur noch Verbesserungen akzeptiert, wodurch sich das SA wie Hillclimbing verhält. Es ist bewiesen, dass bei einer unendlich klein gewählten Temperaturveränderung immer das optimale Maximum gefunden wird. (Vgl.: [LA87]).

Das Experiment von [AH06] zeigt die Verwendung von Simulated Annealing zum Lernen von Laufmustern für einen simulierten Roboter. Das Ziel war es, den spinnenähnlichen Roboter zu befähigen, sich möglichst schnell und energiearm innerhalb der physikalisch simulierten Umwelt vorwärts zu bewegen (siehe 3.2a). Der zentrale Körper des Roboters bestand aus einer einfachen Kugel an der kreisförmig acht Beine fixiert waren. Jedes Bein besaß drei Gelenke, die durch sogenannte *Muskeln* bewegt wurden (Vgl. 3.2b). Das *Hip Joint* war die Verbindung des Beins zum Körper und wurde durch einen Muskel für die horizontale und einen für die vertikale Bewegung gesteuert. Die beiden anderen Gelenke, das *Upper-*, sowie das *Lower Leg Joint* besaßen jeweils nur einen Freiheitsgrad und agierten ähnlich einem Kniegelenk.

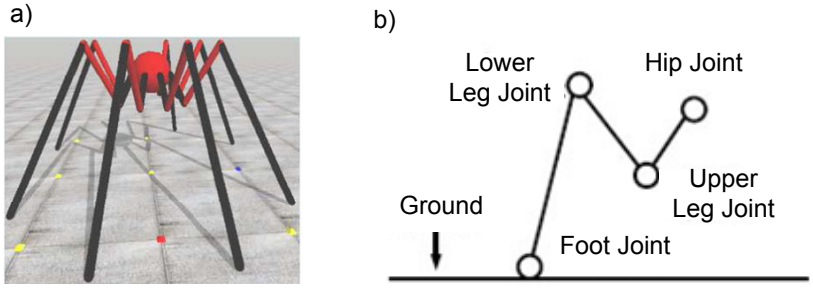

Abbildung 3.2.: a) Simulationmodell der Laufspinne, b) Schematischer Aufbau eines Beines

Jeder Muskel war in der Lage, in einem definierten Maße zu kontrahieren und sich wieder zu entspannen. Die aktuelle Kontraktion konnte in ganzzahligen Prozentwerten angeben werden und beeinflusste über eine einfache trigonometrische Umrechnung die Rotation des zugehörigen Gelenks. Um die Güte eines Bewegungsmusters bestimmen zu können, wurden drei Kriterien definiert: Zunächst sollte sich der Roboter möglichst schnell vorwärtsgerichtet bewegen können. Dabei sollte er jedoch wenig Energie verbrauchen, welche sich aus der summierten Länge aller Muskelaktivierungen berechnete. Zuletzt wurden die Mittelwerte der ersten und zweiten Ableitungen von jeder Muskelkontrollfunktion gebildet, um flüssige Bewegungen bevorzugen zu können. Die Wahrscheinlichkeit, nach der eine schlechtere Lösung für die Expansion des Suchraums gewählt werden kann, wird berechnet durch

$$P = e^{\frac{-\Delta E}{T}}$$

mit ΔE als Differenz zwischen der Güte der aktuellen und neu ausgewählten Lösung und der Temperatur T, welche pro Durchlauf auf 95 % gesenkt wird[1]. Aufgrund dieser Lernstrategie war es dem Roboter bereits nach etwa 700 Iterationen möglich, sich zügig fortzubewegen. Innerhalb der Experimentreihe wurden zudem die einzelnen Gütekriterien verschieden gewichtet, wodurch nachvollziehbare Änderungen des Laufmusters zu beobachten waren: Wurde die Geschmeidigkeit der Bewegung nur gering belohnt, entstanden zwar gerichtete, dafür aber ineffiziente und unnatürliche Bewegungen. Ein Fokus auf dieses Kriterium unter Vernachlässigung der Geschwindigkeit erzielte hingegen die besten Läufe.

3.3. Evolutionäre Algorithmen

> „How can computers learn to solve problems without being explicitly programmed? In other words, how can computers be made to do what is needed to be done, without being told exactly how to do it?“
>
> (zugesprochen Arthur Samuel, 50er Jahre)

Evolutionäre Algorithmen (EA) sind eine Klasse von stochastischen Optimierungsverfahren, welche von den Prinzipien der natürlichen Evolution (Vgl.: [Dar93]) inspiriert werden. Der grundlegende Gedanke sieht eine *Population* von *Individuen* vor, welche in gegenseitiger Konkurrenz um die Ressourcen ihrer Umwelt stehen. Je besser ein Individuum an seine Umwelt angepasst ist, desto größer sind seine Chancen zu überleben und sein genetisches Material an Nachkommen weiterzugeben.
In der Künstlichen Evolution besteht eine Population aus einer Anzahl von Datenstrukturen, die jeweils eine eigene Lösung des Problems kodieren. Die Güte eines Individuums wird bei Optimierungsaufgaben durch eine globale *Fitnessfunktion*[2] bestimmt. Eine nachfolgende Generation wird dabei hauptsächlich aus den Nachfahren fitter Individuen gebildet. Diese Interpretation des „Überleben des Best-Angepassten“ soll sicherstellen, dass schrittweise Populationen entstehen, die zum Optimum des Suchraums konvergieren.

In einer ersten Betrachtung können Evolutionäre Algorithmen als ein Bestärkendes Lernverfahren angesehen werden, bei dem die Fitnessfunktion der Umweltrückkopplung entspricht. In [NF99] wird jedoch zu bedenken gegeben, dass ein wichtiger Unterschied zwischen Evolution und Lernen die Nutzung von Raum und Zeit sei. Evolution setze die Existenz einer auf den Raum verteilten Population vor, deren Individuen sich über

[1] Im Original steht: „(...) decrease the system temperature by 95% (...)“. Der Autor nimmt an, dass es sich dabei um einen Schreibfehler handelt.

[2] Dies entspricht einer *Direkten Evolution*. Beim der *Indirekten Evolution* hingegen wird die Fitness durch freie Interaktion mit der Umwelt beeinflußt (Artificial Life).

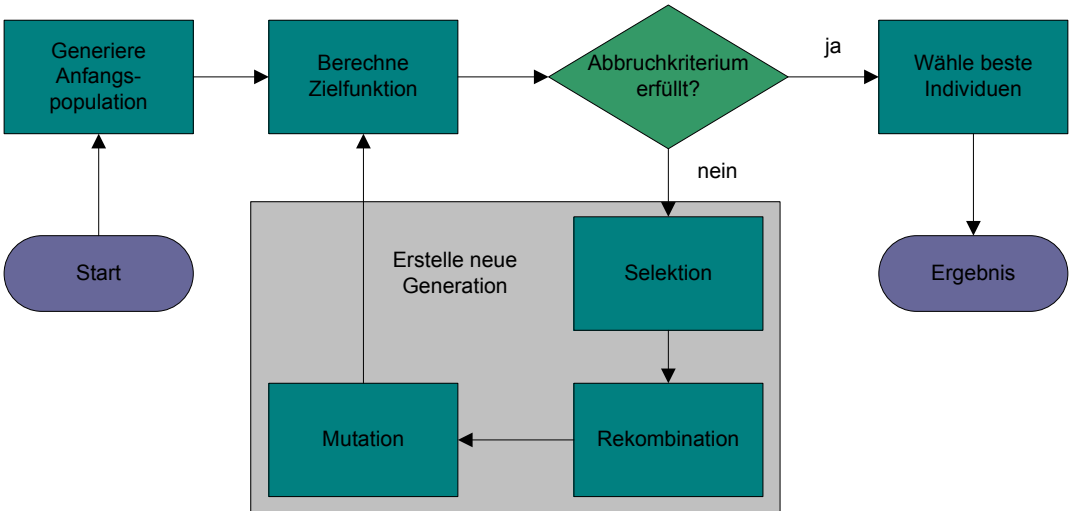

Abbildung 3.3.: Allg. Ablauf eines Evolutionären Algorithmus (nach [Poh99, S. 9])

Generationen langsam an Veränderungen der Umwelt anpassen würden. Ein Individuum sei nicht in der Lage, mehr zu leisten, als in seinem Genmaterial verankert sei. Dem hingegen ermögliche Lernen eine schnelle Veränderung des Verhaltens während des Lebenszyklus. Lernende Individuen könnten sich Fähigkeiten aneigenen, die zum Zeitpunkt ihrer Erschaffung nicht vorgesehen wären. Sie hätten im Gegensatz zur Evolution aber nicht die Möglichkeit, Fähigkeiten zu vererben.

Ablauf Evolutionärer Algorithmen

Obwohl sich im Laufe der Zeit verschiedene Formen der EA gebildet haben (Vgl. Tabelle 3.1), die sich in Strategie, Operatoren und Repräsentationsform unterscheiden, gestaltet sich ihr grundlegender Ablauf gleich. Wie in Abb. 3.3 zu sehen ist, wird zunächst eine Startpopulation erstellt, deren Individuen mit meist zufälligen Variablenwerte belegt werden. Anhand ihrer Fitness werden im Prozess der *Selektion* Individuen gewählt, die sich reproduzieren dürfen, um schlechtere Lösungen zu ersetzen. Dabei werden zusätzlich evolutionäre Operatoren angewandt: Bei der Rekombination entstehen Nachfahren, indem zwei Individuen Teile ihres Genmaterials austauschen. Der Mutationsoperator hingegen verändert mit einer geringen Wahrscheinlichkeit punktuell Geninformationen. Daraufhin werden alle Individuen der neuen Population bewertet und der Algorithmus beginnt von vorn, bis ein Abbruchkriterium erreicht ist. Dies kann beispielsweise eine maximale Laufzeit oder die Entwicklung eines Individuums mit ausreichender Fitness sein. Durch die Wechselwirkung aller Einzelphasen vereinen Evolutionäre Algorithmen die Vorteile von zufälligen und gerichteten Suchstrategien:

Selektion Der maßgebliche Faktor zur Expansion des Lösungsraums ist die Selektion. Sie benutzt die Fitness der Individuen als Heuristik, um die Suche auf ein Optimum zu richten. Je besser die Fitness eines Individuums ist, desto höher ist seine Wahrscheinlichkeit für die Reproduktion gewählt zu werden. Eine starke Bevorzugung der besseren Lösungen beschleunigt zwar die Suche (Exploration) kann aber eine Dominanz weniger Individuen über potentiell nützliche, aber schlechtere Alternativen bewirken (*premature convergence*).

Rekombination Die Rekombination produziert Nachkommen, indem Teile von potentiell guten Individuen untereinander ausgetauscht werden. Dabei entstehen Nachfahren, durch die Teile bewährter Lösungen konserviert werden. Eine Suche über die Rekombination besitzt sowohl exploitativen als auch explorativen Charakter: Die Nutzung bekannter Lösungen betont die Suche in anliegenden Gebieten, allerdings kann durch deren Kombination auch in entferntere Regionen vorgestossen werden.

Mutation Die Mutation wird genutzt, um zufällige Veränderungen in den Genpool der Population zu bringen. Diese spontane Erkundung des Suchraums verhindert ein Festhängen in lokalen Minima und verhindert, dass Genausprägungen völlig aus dem Genpool der Population verschwinden.

3.3.1. Genetische Algorithmen

Die *Genetischen Algorithmen* (GA) wurden gegen 1975 von Holland [Hol75] entwickelt und sind eine besonders populäre Ausprägung der EA. Der Kernaspekt der GA liegt auf einer einfachen Strukturierung der Individuen als gleichlange Stringketten. Jede Variable eines Problems, genannt *Gen*, wird als Abschnitt der Stringkette zunächst binär kodiert und später zur Ermittlung der Fitness wieder dekodiert.
Individuen werden in der Regel proportional zu ihrer Fitness selektiert. Darüber hinaus Varianten, wie die Turnierselektion doer die rangbasierte Selektion, die im Zuge der Genetischen Programmierung in Abschnitt 4.3.2 im Detail beschrieben werden.
Der wichtigste genetische Operator innerhalb der GA ist die Rekombination, welche in drei Formen genutzt wird: Beim *Multi-Point-Crossover* werden bei zwei Individuen mehrere zufällig ausgewählte Schnittpunkte zwischen Genpositionen gewählt. Die daraus resultierenden Segmente werden in Übereinstimmung ihrer Länge unter den Individuen ausgetauscht. Bei der vereinfachten Form des *Single-Point-Crossover* ist hingegen nur eine Schnitstelle vorgesehen. Das *Uniform-Crossover* verzichtet auf Schnittpunkte und entscheidet für jedes einzelne Allel anhand einer vorgegebenen Wahrscheinlichkeit, ob es zwischen den Eltern ausgetauscht werden soll. Bei der Mutation eines Individuums wird die Belegung von Genen verändert, indem entweder Bits gekippt oder durch einen

Jahr	Erfinder	Technik	Individuum
1958	Friedberg	Lernmaschine	Vitueller Assembler
1959	Samuel	Mathematik	Polynom
1965	Fogel, Owens und Walsh	Evolutionäre Programmierung	Automaton
1965	Rechenberg, Schwefel	Evolutionsstrategien	Reellwertiger Vektor
1975	Holland	Genetische Algorithmen	Bit-String mit fester Länge
1978	Holland, Reitmann	Genetische Klassifizierungssysteme	Regeln
1980	Smith	Frühe Genetische Programmierung	Bit-String mit variabler Länger
1985	Cramer	Frühe GP	Baum
1986	Hicklin	Frühe GP	Lisp
1987	Fujiki, Dickinson	Frühe GP	Lisp
1987	Dickmanns, Schmidhuber und Winklhofer	Frühe GP	Assembler
1992	Koza	Genetische Programmierung	Baum

Tabelle 3.1.: Zeitliche Übersicht der wichtigsten Formen der EA (nach [BNKF97, S. 102])

zufälligen Wert ersetzt werden. Eine ausführiche Beschreibung der GA ist in [Mic99] und [SHF94] zu finden.

3.3.2. Evolutionsstrategien

In der Mitte der sechziger Jahre entwickelte Ingo Rechenberg die *Evolutionsstrategie* (ES) [Rec73], welche insbesondere von Hans-Paul Schwefel aufgegriffen und entscheidend erweitert wurde [Sch75], [Sch77]. Die Individuen werden als Vektor reeller Zahlen mit fester Länge dargestellt und können neben den Variablen der Lösung sog. *Strategieparameter* enthalten. Diese können als Metadaten innerhalb des Algorithmus verwendet werden, beispielsweise um die Schrittweiten der Mutation als Teil des Individuums und somit der Fitness zu definieren. Über die Jahre wurde die klassische Art, die (1+1)-Evolutionsstrategie mit nur einem Individuum, vielen Wandlungen unterzogen, wodurch zunehmend komplexere und naturgetreuere Modelle entstanden (u.a.: (1+λ)-ES, (1,λ)-ES, (μ, λ)-ES). Die spezielle Notation beschreibt hierbei die Größe der Population (μ) und die der Nachkommen (λ) sowie die verwendete Selektionsart. Bei der Plusstrategie stehen die Eltern im Wettbewerb mit ihren Kindern während die Kommastrategie die alte Generation vollständig ersetzt. In der Regel werden bei den populationsorientierten ES wesentlich mehr Nachkommen produziert, als die Folgepopulation aufnehmen kann,

wodurch der Selektionsdruck erhöht wird. Eine detaillierte Erläuterung aller Varianten der ES bietet [SHF94].

3.3.3. Evolutionäre Programmierung

Die *Evolutionäre Programmierung* geht auf die Arbeit von Fogel, Owens und Walsh aus der Mitte der sechziger Jahre zurück [FOW66]. Die Population der EP besteht aus Endlichen Automaten, welche anhand einer Sequenz von Eingabesignalen der Umwelt ihren Zustand wechseln und mit der Ausgabe von Symbolen reagieren. In ihrer ursprünglichen Anwendung werden EP-Individuen eingesetzt, um Symbolfolgen vorherzusagen. Ihre Fitness wird dann als Differenz zwischen der Eingabe und der erzeugten Ausgabe gemessen. Die Selektion lässt in der Regel eine kleine Gruppe zufällig gewählter Individuen gegeneinander antreten. Die Gewinner werden entweder in die nächste Generation repliziert oder über Mutationsoperationen variiert, eine Rekombination findet hingegen nicht statt. Der Schwerpunkt der EA liegt auf den verschiedenen Arten der Mutation. Für die unterschiedlichen Elemente des Automaten werden jeweils andere Operatoren gewählt, eine Normalverteilung stellt dabei sicher, dass kleine Änderungen häufiger auftreten, als große. Als Vertiefung kann auf [Nis94, Kapitel 3] verwiesen werden.

3.3.4. Anwendbarkeit Evolutionärer Algorithmen

Evolutionäre Algorithmen bieten aufgrund ihrer Arbeitsweise eine Vielzahl von Vorteilen gegenüber traditionellen Optimierungsmethoden. Es muss über die Anwendungsdomäne nur wenig Wissen im voraus vorhanden sein, da es reicht, ein Repräsentationsschema und eine Gütefunktion für die Lösungen des Problems zu definieren. Zwar ist es zusätzlich notwendig, die Parameter des EA einzustellen, wie etwa die Größe der Population oder die Anwendungswahrscheinlichkeit der evolutionären Operatoren. Hierfür existieren jedoch Standardkonfigurationen, die in den meisten Fällen zu guten Ergebnissen führen. Weiterhin ist es von Vorteil, dass der Algorithmus zu jeder Zeit eine Anzahl an frei wählbaren Lösungen bietet. Auf diese Weise ist ein explizites Abbruchkriterium nicht zwingend notwendig: Sind bereits ausreichend gute Lösungen produziert, kann der Algorithmus gestoppt werden, andernfalls wird er zur weiteren Optimierung fortgesetzt. Zuletzt sind EA durch die Verwendung einer Population an Lösungen hochgradig parallelisierbar. Auf der anderen Seite müssen einige Voraussetzungen getroffen werden, damit ein EA effektiv arbeiten kann. Wie bereits angedeutet, muss jede Lösung durch eine Fitnessfunktion messbar sein. Dabei spielt es eine wichtige Rolle, dass die Bewertung möglichst schnell erfolgen kann, da sich jede Verzögerung der Auswertung aller Populationen summiert. Nicht selten handelt es sich hierbei für komplexere Probleme um dutzende Individuen über hunderte von Generationen. Dieser Aspekt schlägt sich auch in den Anforderungen an die Hardware des evolvierenden Systems aus. Neben dem

hohen Speicherbedarf zur Repräsentation der Individuen kommt der Rechenaufwand für die evolutionären Operatoren und die Auswertung. Aus diesen Gründen nennt Pohlheim [Poh99, S. 12] zwei Indizien, wann EA nicht eingesetzt werden sollten. Zum einen, falls ein spezielles und besser an das Problem angepasstes Verfahren bekannt sei. Zum anderen, wenn die Berechnung der Zielfunktion zu aufwändig sei. Zuletzt muss immer bedacht werden, dass durch EA zwar schnell zu sehr guten Lösungen gefunden werden kann. Bei komplexen Lösungen und einem diskreten Zeittraum ist aufgrund des zufälligen Charakters der Suche die Findung des Optimums jedoch nicht garantiert. Als vertiefende Abwägung der Vor- und Nachteile kann [Nis94, Kapitel 5.3.1] empfohlen werden.

3.4. Zusammenfassung

Das Kapitel hat einen Überblick über die Methodik des Maschinellen Lernen gegeben. Zu diesem Zweck wurden Aufgaben des Lernens in die drei Kategorien des Überwachten, Unüberwachten und Bestärkenden Lernens unterteilt. Dabei wurde der größte Unterschied zur herkömmlichen Algorithmik herausgestellt: Im Bereich des ML wird das induktive Lernen eines Systems angestrebt, nicht die deduktive Programmierung. Zusätzlich wurden Anwendungsbeispiele genannt, die den Einsatz der lernenden Methoden motivieren.
In einer genaueren Betrachtung wurde das Bestärkende Lernen diskutiert, welches Agenten selbständig lernen lässt, indem Signale der Umwelt im Sinne einer Vermehrung von Belohnung angestrebt wird. Als Anwendungsbeispiel wurde ein Experiment der Forschung aus dem Bereich des Simulated Annealing vorgestellt.
Zuletzt wurde in das Thema der Evolutionären Algorithmen eingeführt. Diese bilden eine besondere Klasse von stochastischen Algorithmen, welche sich der Prinzipien der natürlichen Evolution bedienen, um eine besonders effektive Suche durch den Raum der Lösungen zu ermöglichen.

Kapitel 4.

Genetische Programmierung

Anfang der neunziger Jahre entwickelte John Koza eine eigene Form der Evolutionären Algorithmen, das Genetische Programmieren (GP). In seinem Grundlagenwerk [Koz92] kritisiert er zwei wesentliche Eigenschaften aller bisherigen EA: Zum einen die Notwendigkeit, die Lösung eines Problems in eine statische Struktur kodieren zu müssen, zum anderen, die Passivität der entwickelten Individuen.
Bei der Initialisierung der vorgestellten EA ist es stets notwendig, eine Repräsentationsform zu wählen, die eine feste Länge besitzt. Um ein Problem in eine statische Struktur zu überführen, ist jedoch umfangreiches Wissen über die Beschaffenheit seiner Lösung notwendig. Erst, wenn die Anzahl und möglichen Ausprägungen der Variablen bekannt sind kann eine geeignete Kodierungsformel entworfen werden. Koza behauptet, dass in einer Vielzahl der Probleme dieses Vorwissen nicht gegeben sei und durch Annahmen ersetzt werden müsse. Darin bestehe jedoch die Gefahr, ein zu begrenztes Repräsentationsschema zu entwerfen, welches nur ein Teil des Lösungsraumes abdecke und so erfolgreiche Individuen nicht gefunden werden könnten. Aus diesem Grund fordert Koza, die Datenstrukturen dem evolutionären Prozess selbst zu unterwerfen:

> „The size, shape, and structural complexity should be part of the answer produced by a problem solving technique - not part of the question."
>
> [Koz92, S. 2]

Während die EA den Anwender von der Notwendigkeit befreit, umfassendes Wissen über das Prozessmodell der Anwendungsdomäne zu besitzen, geht das Prinzip von Koza somit einen Schritt weiter: Zusätzlich kann die Komplexität der Lösung unbekannt sein. Als weiteren Nachteil sieht Koza, dass es sich bei den klassischen Individuen um rein passive Datenstrukturen handle, die erst mit Hilfe eines zusätzlichen Interpreters zur Ausführung gelangten. Ein Individuum müsse jedoch in der für ein Computersystem natürlichsten Form auftreten - dem Programm (Vgl.: [Koz92, S. 9, S. 76]). Dies erreicht Koza, indem er den genetischen Pool einer Population erweitert: Statt die Individuen auf die Repräsentation von Problemvariablen zu begrenzen, enthalten sie zusätzlich

Steuerungselementen, wie Iterationen, Entscheidungszweige und Aufrufen von Subroutinen. Auf diese Weise werden nicht nur die Parameter eines Problems evolviert, sondern auch die Flusskontrolle eines zugehörigen Programms. Besonders in Bezug auf Regelungstechnik ergibt sich dadurch ein entscheidender Vorteil der GP. Wie Abb. 4.1 zeigt, sind die Programm-Individuen in der Lage, während ihrer Ausführung neue Eingaben zu erhalten (z. B. durch einen Sensor), diese anhand von Verzweigungsoperationen zu bewerten und als Ergebnis Seiteneffekte auf ein umgebendes System zu bewirken (z. B. Motoransteuerung). Werden den Individuen Iterationen zur Verfügung gestellt, wird die Laufzeit als Teil der Evolution bestimmt. Alternativ kann aber auch eine äußere Schleife verwendet werden, die fest definiert das Steuerprogramm mehrfach ausführt.

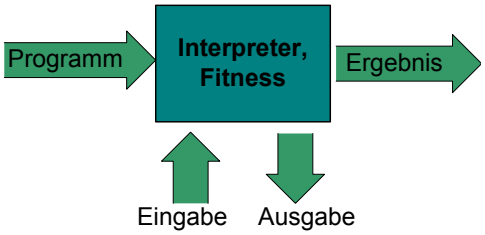

Abbildung 4.1.: Interaktion eines GP-Programms mit dem umgebenden System

In den folgenden Abschnitten wird beschrieben, auf welche Weise Koza seine Forderungen nach einem flexibleren EA in der Genetischen Programmierung umgesetzt hat und wie deren Ablauf im Detail stattfindet. Da es den Rahmen dieser Arbeit überschreiten würde, auf alle Richtungen der Genetischen Programmierung gleichermaßen einzugehen, wird die Betrachtung dabei beschränkt. Als Form wird grundlegend von Baum-GP ausgegangen, die Parameter sowie die Arbeitsweise der Operatoren und Verfahren stützen sich hauptsächlich auf [Koz92] und [BNKF97].

4.1. Struktur der Individuen

Ein GP-Individuum wird aus den Eingabewerten des Systems, den sog. Terminalen und einem Satz von Funktionen zusammengesetzt (Vgl. Kapitel 2.6). Zusätzlich wird eine umschließende Struktur benötigt, die eine Ablaufreihenfolge definiert und die atomaren Einheiten zu einem Programm vereint. Da Koza seine Beispiele anhand der Programmiersprache *Lisp* demonstrierte, wird seitdem der Syntaxbaum als traditionelle Form des GP angesehen. Lisp ermöglicht es aufgrund seiner besonderen Funktionsweise, effektiv mit GP-Individuen zu arbeiten: Alle Programme werden als Baumstrukturen, den S-Expressions, formuliert (Vgl. Kapitel 2.6), welche sowohl den Steuerfluss als auch die Daten vereinen. Operationen, wie die Modifikation des Baumes über genetische Operatoren oder dessen rekursive Abarbeitung werden implizit von der Sprache unterstützt.

Grundsätzlich sind die GP-Strukturen in jeder Programmiersprache und auf verschiedene Weisen modellierbar. So existieren neben den Baumstrukturen auch die in Kapitel 2.5 vorgestellte Lineare Repräsentation, ein Ansatz mit freien Graphen [TV95] sowie hybride Systeme wie dem Linear-GP [KB01], [KB02]. Dabei ist jedoch zu beachten, dass nicht in jeder Sprache das GP-Konzept gleichermaßen einfach umzusetzen ist (siehe Kap. 5.3) und die Wahl der Struktur maßgeblich mitbestimmt, auf welche Weise die genetischen Operatoren zu arbeiten haben.
Da innerhalb des evolutionären Prozesses die Strukturen der Individuen beliebig verändert werden können, muss Sorge getragen werden, dass jede Funktion des Baumes die Ergebnisse beliebiger Kindknoten verarbeiten kann. Dieses als *closure* bekannte Prinzip verhindert ungültige Rechenanweisungen, die zum Programmabsturz führen können (z. B. Nulldivision). Funktionen, die keinen definierten Rückgabewert besitzen (reine *setter* oder ungültige mathematische Resultate) müssen daher mit einem konstanten Standardwert arbeiten.

Initialisierung der Strukturen

Zur Initialisierung der GP-Evolution muss zunächst eine Population von zufälligen Baum-Individuen aus der Menge der Terminale und Funktionen geschaffen werden. Häufig ist es jedoch erwünscht, bestimmte Sonderfälle noch während des Kreationsprozesses auszuschließen. Beispielsweise verhindert eine Restriktion der minimalen und maximalen Knotenanzahl, dass übermäßig große Bäume die Systemressourcen überfordern oder degenerative Individuen aus nur einem Terminal entstehen. Zu diesem Zweck stellt Koza zwei anwendungsunabhängige Strategien vor: *full* und *grow*. Beide Methoden bauen einen Baum bis zu einer gegebenen Maximaltiefe auf und schließen jeden Ast mit einem Terminal. Full wählt als innere Knoten nur Funktionen aus, wodurch sich alle Äste des Baumes über die maximale Tiefe erstrecken (4.2a). Im Gegensatz dazu kann die grow-Methode auch Bäume von irregulärer Form erzeugen, da sie die inneren Knoten rein zufällig bestimmt und so Äste vorzeitig terminiert werden können (4.2b).

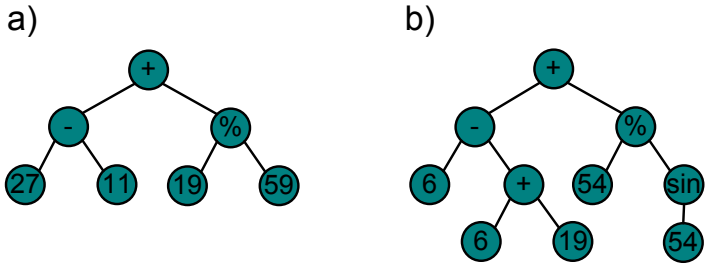

Abbildung 4.2.: Initialisierung der Bäume: a) full-Methode mit $t_{max} = 3$, b) grow-Methode mit $t_{max} = 4$

Um allerdings eine möglichst hohe Diversität der Population zu erreichen, empfiehlt Koza beide Strategien in der *ramped half-and-half* Methode zu vereinen [Koz92, S. 93]. Demnach werden die Bäume in gleichgroße Klassen gemäß ihrer Tiefe von 2 bis t_{max} geordnet. Bei einem Wert von bsp. $t_{max} = 6$ entstehen so fünf Klassen mit den Tiefen 2, 3, 4, 5, 6, die jeweils zur Hälfte über die grow- und zur anderen Hälfte mittels der full-Methode initialisiert werden.

4.2. Operatoren

4.2.1. Reproduktion

Über den asexuellen Operator der Reproduktion wird ein Individuum unverändert in die neue Generation kopiert, wodurch bei gleichbleibender Bewertungsformel eine erneute Evaluierung entfällt. Die Reproduktionsrate wird für gewöhnlich auf 10 % gesetzt. Eine Ausnahme bildet das *Prinzip der Elite*. Hierbei wird eine Auswahl an Individuen mit der höchsten Fitness garantiert in die neue Population übernommen.

4.2.2. Rekombination

Die Rekombination (häufig auch als *Crossover* bezeichnet) wählt zwei Individuen als Eltern zur Erzeugung von zwei Nachkommen. Zunächst wird zufällig bei beiden Individuen ein Knoten als Schnittpunkt ausgewählt. Dieser wird mit dem unterliegenden Subbaum ausgehängt und an den Schnittpunkt des jeweils anderen Individuums angefügt (Abb. 4.3). Aufgrund des closure-Prinzips enstehen stets valide Individuen, unabhängig davon, welche Teilbäume ausgetauscht werden. Koza sieht die Rekombination als treibende Kraft der Evolution und empfiehlt eine Anwendung auf 90 % der Population.

4.2.3. Mutation

Der unäre Mutationsoperator wird in der Regel nach der Rekombination auf jedes Kind mit einer geringen und fitnessproportionalen Wahrscheinlichkeit angewandt. Dazu wird anhand eines zufälligen Punktes der Struktur ein Subbaum ausgewählt und durch einen neu erzeugten ersetzt (Abb. 4.4). Die Parameter zur Erstellung des neuen Subbaums entprechen hierbei denen der Individuen-Initialisierung. Es existieren darüber hinaus diverse Varianten der Mutation, die z. B. nur die Werte eines Terminals ändern oder

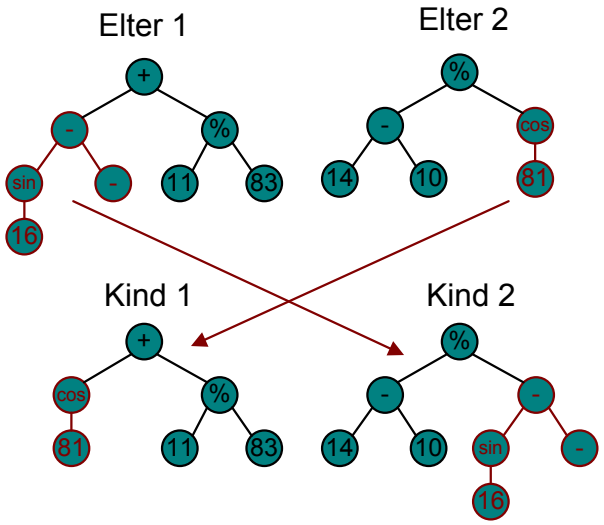

Abbildung 4.3.: Rekombination zweier Individuen (oben), deren Kinder durch Austausch der roten Subbäume entstehen (unten).

Subbäume innerhalb eines Individuums permutieren. Die Mutationsrate wird meist sehr niedrig gewählt, um die Population nicht übermäßig zu destabilisieren. Koza war der Meinung, die Mutation sei in der Genetischen Programmierung ganz zu vernachlässigen, da die Wahrscheinlichkeit, dass ein Symbol aus dem Genpool völlig verschwinden würde, sehr gering sei (Vgl. [Koz92, Kapitel 6.5.1]).

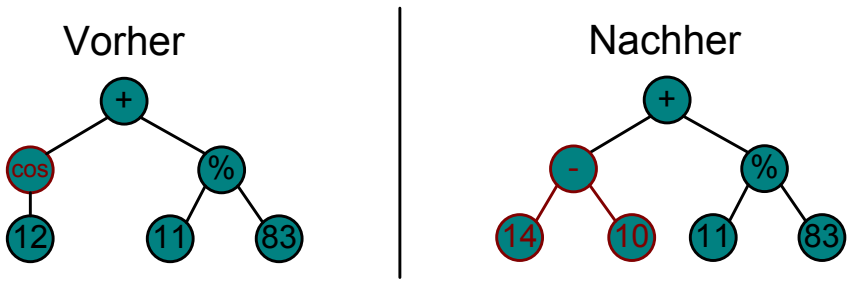

Abbildung 4.4.: Individuum vor und nach der Mutation. Der ausgewählte Knoten bzw. der neue Subbaum ist rot markiert.

4.3. Fitness und Selektion

4.3.1. Fitness

Jedes Individuum wird zunächst auf ein Problem angewandt und abhängig vom seinem Erfolg mit einem Fitnesswert ausgezeichnet. Abhängig von der Art des Problems kann die Fitness unter anderem die Anzahl von Übereinstimmungen oder Fehlern (z. B. Vorhersage, Regression), die Nutzung einer Ressource (z. B. Zeit, Speicher) oder das Ergebnis von Seiteneffekten sein (z. B. Steuerungstechnik). Ebenfalls ist es möglich, mehrere Berwertungskriterien in einer Fitness zu vereinen.
Nicht immer können Fitnesswerte von Individuen dabei direkt vergleichbar sein. Häufig muss ein Individuum Probleme nicht nur unter gleichbleibenden, sondern auch unter wechselnden Bedingungen lösen. In diesem Fall wird das Individuum mehrfach mit wechselnden Eingabewerten über eine Reihe von *fitness cases* ausgeführt. Die finale Güte ergibt sich dann aus der Summe oder dem Durchschnitt aller erreichten Fitnesswerte der Einzelversuche. Beispielsweise können bei Funktionsregressionen die fitness cases eine Anzahl an Koordinaten sein, die berechnet und mit der Zielfunktion verglichen werden. Bei der Steuerung eines Laufroboters hingegen stellt bereits jede neue Startposition auf ungleichmäßigem Grund eine neue Ausgangssituation dar.
Die Fitness kann auf unterschiedliche Weise ausgedrückt werden. Koza stellte hierfür die folgenden Maßeinheiten vor:

Raw Fitness bezeichnet den numerischen Wert, welchen ein Individuum in Anwendung auf das Problem erreicht. Bei Funktionsregression kann die raw fitness beispielsweise die Anzahl der übereinstimmenden Koordinaten sein, bei einem Schreitroboter hingegen die Länge des zurückgelegten Weges.

Standardized Fitness benutzt als Grundlage die Raw Fitness, definiert jedoch für eine höhere Fitness einen geringeren Wert, wobei 0 das Optimum markiert. Falls die Raw Fitness als Summe von Fehlern gemessen wurde, ist diese Bedingung bereits erfüllt. Andernfalls berechnet sich die Standardized Fitness s durch die Differenz zwischen der maximalen (r_{max}) und der erreichten Raw Fitness (r):

$$s_i = r_{max} - r_i$$

Adjusted Fitness skaliert die Fitness auf das Intervall $]0, 1]$ durch die Formel:

$$a_i = \frac{1}{1 + s_i}$$

wobei s die Standardized Fitness des Individuums i ist und ein höherer Wert einer besseren Fitness entspricht. Die Adjusted Fitness eignet sich besonders, um

in späteren Phasen der Evolution ähnliche Individuen deutlich voneinander unterscheiden zu können.

Normalized Fitness verhält sich wie die Adjusted Fitness mit dem Unterschied, dass die Summe aller Fitnesswerte einer Population 1 ergibt. Die Normalized Fitness n des Individuums i ergibt sich wie folgt aus der Adjusted Fitness:

$$n_i = \frac{a_i}{\sum_{j \in P} a_j}$$

4.3.2. Selektion

Die Auswahl der Selektionsstrategie ist ein entscheidender Faktor innerhalb der Genetischen Programmierung. Sie definiert die Chance eines Individuums, bevorzugt für die genetischen Operatoren ausgewählt zu werden und somit den Konkurrenzdruck innerhalb der Population. Unter den vielen verschiedenen Strategien haben sich insbesondere die folgenden Selektionsmethoden durchgesetzt:

Fitness-Proportionale Selektion

Die Selektionswahrscheinlichkeit p berechnet sich bei der Fitness-Proportionalen Selektion aus dem Verhältnis der Fitness des Individuums i zur Fitness der Gesamtpopulation:

$$p_i = f_i \frac{1}{\sum_{j \in P} f_j}$$

Diese Selektion wird häufig mit einem Rouletterad verglichen, auf dem jedes Individuum einen prozentualen Abschnitt gemäß seiner Fitness belegt. Allerdings können bei dieser Methode zwei Effekte die Evolution nachteilig beeinflussen: Auf der einen Seite kann eine Elite von wenigen fitnessstarken Individuen die Evolution dominieren, wodurch sich die Gefahr einer vorzeitigen Lösungskonvergenz erhöht. Auf der anderen Seite ist der Selektionsdruck zu niedrig, wenn die Individuen sich nur unwesentlich voneinander unterscheiden (Vgl.: [Nis94, S. 46f]).

Rangbasierte Selektion

Bei der Rangbasierten Selektion werden alle Individuen anhand ihrer Fitness geordnet. Die Selektionswahrscheinlichkeit p ergibt sich aus der Position r (dem Rang) des Indi-

viduums i innerhalb dieser Ordnung:

$$p_i = f(r_i)$$

Die Funktion f beschreibt hierbei die Abhängigkeit des Ranges zur Auswahlwahrscheinlichkeit (meist lineare oder exponentielle). Diese Methode bewirkt die gleichmäßige Verteilung der Individuen auf der Bewertungsskala und verhindert somit die Probleme der Fitness-Proportionalen Selektion. Individuen mit sehr ähnlichen Fitnesswerten sind anhand ihres Ranges deutlich zu unterscheiden, zudem ist es nicht möglich, dass eine Gruppe von Eliteindividuen entsteht.

Turnierselektion

Im Laufe der Turnierselektion wird eine Untergruppe der Population zufällig ausgewählt, deren Individuen gegeneinander antreten. Die Individuen mit der höchsten Fitness gelten als Gewinner und dürfen Nachkommen bilden, welche die Verlierer in der neuen Generation ersetzen. Als bestimmender Faktor des Selektionsdrucks wird die Größe des Turniers geregelt: Je mehr Individuen teilnehmen, desto höher ist die Wahrscheinlichkeit, dass sich unter ihnen die Elite der Population befindet und das Turnier dominiert - der Selektionsdruck steigt. Bei kleinen Turnieren hingegen werden sich häufig nur schwache Individuen aneinander messen, wodurch der Selektionsdruck sinkt. Die Turnierselektion ist ein Beispiel für die sog. *steady-state*-Methode. Im Gegensatz zu den bislang vorgestellten Selektionsverfahren wird hierbei nicht die gesamte Population, sondern nur ein Teil von ihr durch Nachkommen ersetzt. In diesem Zusammenhang wird von einer neuen Generation gesprochen, wenn durch wiederholte Anwendung des Selektionsverfahrens Nachkommen in Anzahl der Populationsgrösse entstanden sind.

4.4. Ablauf

Bevor der allgemeine Ablauf eines Evolutionslaufes näher beschrieben wird, sollen die Schritte zur Vorbereitung noch einmal zusammengefasst werden:

1. Definition der Terminal- und Funktionsmenge
2. Definition der Fitnessfunktion
3. Konfiguration der Evolutionsparameter
4. Definition eines Abbruchkriteriums

Nachdem die Terminal- und Funktionsmengen sowie die Individuenstruktur bestimmt worden sind, müssen die Bedingungen der Evolution festgelegt werden. Koza beschreibt hierzu in drei Kategorien 19 Parameter[1], von denen einige in den vorangegangenen Abschnitten bereits implizit erwähnt wurden. Da nicht jeder Parameter für diese Arbeit von Interesse ist, soll nur ein Überblick gegeben werden:

Die Hauptparameter bestehen aus der Grösse der Population und der Anzahl der Generationen.

Die Nebenparameter beschreiben hauptsächlich die Wahscheinlichkeiten zur Anwendung der Genetischen Operatoren sowie die maxmiale Baumtiefe.

Die Strategieparameter setzen u. a. die Methoden zur Generierung der Bäume und zur Selektion fest. Darüber hinaus definieren sie das Fitnessmaß.

Da die Genetische Programmierung unabhängig von der Anwendungsdomäne ist, ist es ausreichend, einen allgemeinen Algorithmus zur Lösung zu beschreiben.

1. Initialisierung einer zufälligen Population
2. Wiederholung der folgenden Schritte bis das Abbruchkriterium erfüllt ist:
   a) Evaluierung der Individuen mit anschließender Fitnesszuweisung
   b) Wiederholung, bis eine neue Population von Individuen entstanden ist
      i. Selektion von einen oder mehreren Individuen anhand der eingestellten Selektionsmethode
      ii. Anwendung der Genetischen Operatoren auf das Individuum
      iii. Einfügen des resultierenden Individuums in die neue Population
3. Wähle bestes Individuum aller Generationen als Ergebnis

4.5. Zusammenfassung

Das Kapitel hat einen detaillierten Überblick in die Genetische Programmierung gewährt. Es wurde zunächst die Motivation des geistigen Vaters John Koza dargelegt, eine erwei-

[1] Im Anhang befindet sich unter A.1 die vollständige Auflistung aller Parameter mit den von Koza definierten Standardwerten.

terte Form der Evolutionären Algorithmen zu erschaffen. Die Hauptgründe hierfür sind die Befreiung von einer statischen Datenstruktur und die aktive Rolle der Individuen als Computerprogramme. Durch die Beschreibung der Arbeitsweise der GP konnten diese Vorteile im Kontext einer Robotersteuerung gedeutet werden: GP-Programme erlauben die Ausführung von Seiteneffekten, wodurch ein Individuum gezielt auf Sensorik und Aktorik zurückgreifen kann, um mit dem umgebenden System zu interagieren. Zudem befreit die variable Form der Programme von der Notwendigkeit, Vorkenntnisse über die Beschaffenheit der Lösungsstruktur zu besitzen.

Kapitel 5.

Lösungsansatz mittels Genetischer Programmierung

Die Zielstellung dieser Arbeit ist es, Bewegungsmuster für den Schreitroboter ELFE automatisch erzeugen zu lassen. Als erster Schritt zur Untersuchung der Komplexität dieses Problemes heißt es, festzustellen, auf welche Weise die Architektur des Roboters Bewegung ermöglicht. Die ELFE wird durch ein Mikrocontrollerboard gesteuert, das die Ausführung von beliebigen Programmen des reduzierten C-Compilers ermöglicht. Jedes der elf Beine wird über die Ansteuerung der korrespondierenden Servomotoren ausgerichtet. Zwei Bibliotheksfunktionen übernehmen hierbei die Umrechnung von ganzzahligen Winkeln (0° bis 180°) oder einer Aktivierungszeit in die Pulsweitenmodulation der Motoren. Während die Servokommandos zur unabhängigen Steuerung eines einzelnen Beines genutzt werden, dienen eingeschobene Wartezyklen zur Koordination der Beine untereinander. Die grundlegende Frage ist somit, wie diese Befehle zu einer Sequenz zusammengesetzt werden müssen, um die gewünschte Bewegung zu generieren.
Abhängig von der gewählten Art der Bewegung fällt diese Aufgabe unterschiedlich schwer aus. Es ist zu vermuten, dass der Lösungsraum für einfache Aktionen, wie etwa dem Hinlegen des Körpers, relativ dicht besiedelt ist. Hierzu müssten nur alle Beine weit genug angewinkelt werden, um das Aufliegen des Hauptkörpers auf dem Boden zu ermöglichen. Eine zeitliche Abfolge der Beinaktivierungen spielt dabei keine Rolle.
Im Rahmen dieser Arbeit wurde jedoch ein komplexeres Ziel formuliert: Es soll ein Bewegungsmuster erzeugt werden, welches die ELFE befähigt, sich möglichst schnell vom Ursprungsort zu entfernen. Bei der Betrachtung der physischen Beschreibung der ELFE aus Abschnitt 1.1.1 lässt sich bereits vermuten, dass es schwierig ist, intuitiv solch ein Steuerungsschema zu beschreiben. Durch die radiale Anordnung stehen einige Beine stets quer zur Laufrichtung, so dass keine Orientierung existiert, welche eine geradlinige Fortbewegung unterstützt. Da jedes Bein zudem nur über einen Freiheitsgrad verfügt, kann es weder versetzt noch geschwungen werden. Dies verhindert insbesondere den Ansatz von klassischen Schrittfolgen, wie dem alternierenden Tripodengang[1], der bei Insekten zu beobachten ist.

[1] Ausführlich beschrieben in [Kra02].

In zwei Studentenprojekten der Fachhochschule Brandenburg wurde bereits versucht Fortbewegungsmuster für die ELFE mittels GA zu evolvieren, was aufgrund mangelnder Zeit jedoch nicht vollendet werden konnte. Dennoch wurde gezeigt, dass der Roboter über manuelle Kodierung und ohne Hilfe eines genauen Prozessmodells zur Bewegung gebracht werden kann[2]. Die Fähigkeiten der ELFE wurden für die Experimente in der Simulation erweitert, indem vier Infrarotsensoren an ihrem Unterkörper angebracht wurden. Auf diese Weise ist es dem Roboter möglich, in beschränktem Maße seine Umwelt wahrzunehmen und auf sie zu reagieren. Durch die Abstandsmessung zum Untergrund sind beispielsweise Verhaltensweisen realisierbar, bei denen die ELFE versucht, ein Schleifen ihres Körpers zu verhindern, indem sie rechtzeitig stützende Beine aktiviert. Diese Art der Kollisionsvermeidung wäre zudem ein erster Schritt, um den Gang unabhängig von der Beschaffenheit eines Untergrundes zu entwickeln.
Aufgrund der hohen Anzahl der Parameter durch Sensorik und Aktorik und die ungewöhnliche Struktur der ELFE ist die Aufgabenstellung nach Mitchell in den Bereich der Komplexen Softwareapplikationen einzuordnen (siehe Kapitel 3.1). Selbst wenn es bislang möglich war, das Bewegungsmuster manuell zu erstellen, ist damit ein hoher und personenbezogener Zeitaufwand verbunden. Zudem können bereits geringe Veränderungen an der Morphologie des Roboters oder in der Zielsetzung der Aufgabe erfordern, dass der Steuerungsalgorithmus komplett neu entworfen werden muss. Aus diesem Grund soll im Folgenden demonstriert werden, wie eine Technik des Maschinellen Lernens erfolgreich genutzt werden kann, um die Problemstellung automatisiert und flexibel zu lösen. Die Wahl der ML-Methode fiel aus mehreren Gründen auf das Genetische Programmieren: Es sind weder Testdaten, noch genügend Vorwissen vorhanden, um das System zu trainieren oder den Weg der Lösung näher zu beschreiben. Stattdessen kann nur die Güte einer Lösung im Sinne der Zielerwartung bestimmt und als Rückkopplungssignal verwendet werden. In diesen Rahmenbedingungen spiegeln sich bereits die Indikatoren zur Anwendung der Evolutionären Algorithmen aus Kapitel 3.3.4 wider. Da jedoch die Länge der Steuerungssequenzen unbekannt ist und zudem während der Evaluierung der Lösungen Sensordaten verarbeitet werden müssen, scheiden Algorithmen mit statischen Datenstrukturen aus. Die Kontrolle der ELFE, sowohl über die Sensoren als auch über die Aktoren, wird somit Teil der automatisch entwickelten Programme.

5.1. Orientierung des Lösungsansatzes

Zur Lösung des Problems wurde ein hybrider Versuchsaufbau angestrebt, welches die Entwicklung von Steuerprogrammen sowohl auf dem Realen Roboter wie auch auf einem simulierten System ermöglichen sollte. Die Absicht war, die Anfangsphase der Evolution auf einem Modell der ELFE virtuell zu testen und erfolgreiche Individuen mit dem realen Roboter weiterzuentwickeln. Dadurch sollte Zeit eingespart und das Material des

[2]Siehe: http://ots.fh-brandenburg.de.

Roboters während der langwierigen Versuche geschont werden. Es stellte sich jedoch heraus, dass sich die Schaffung eines flexiblen hybriden Evaluationssystems zunehmend zeitintensiver gestaltete als angenommen. Besonders der Umgang mit dem Simulator *BREVE* zeigte unvorhersehbare Hürden, die einer zügigen Entwicklung entgegenwirkten. Das Ziel der Arbeit bleibt vom Kernaspekt gleich, es sollen Individuen zur Steuerung der ELFE-Morphologie auf Basis der Genetischen Programmierung entwickelt werden. Allerdings verschiebt sich der Schwerpunkt auf die Umsetzung der Problemlösung mit dem Simulator. Zwar wurde das implementierte Evaluationsprogramm auch für den Zweck entwickelt, auf dem AKSEN-Board eingesetzt zu werden, für einen ausführlichen Test mit dem realen Roboter verblieb jedoch keine Zeit.

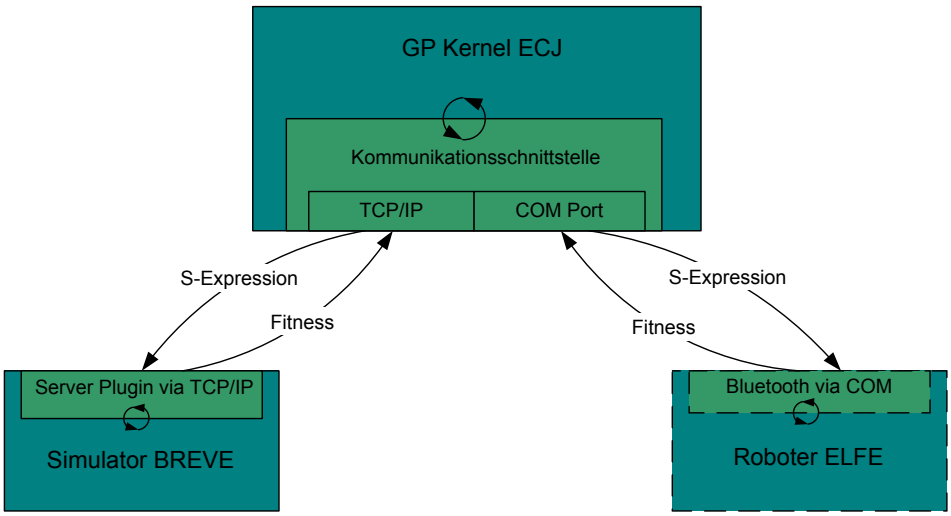

Abbildung 5.1.: Schematischer Aufbau des Hybriden Evolutionssystems

Wie Abb. 5.1 zeigt, wurde der Lösungsansatz dieser Arbeit in drei Komponenten gegliedert. Auf einem PC lief der GP-Kernel *ECJ*, der für die Initialisierung einer baumbasierten Population von Laufsteuerprogrammen verantwortlich war und die Individuen dem evolutionären Prozess unterwarf. Über eine einheitliche Verbindungsschnittstelle wurden die Individuen mit den unterschiedlichen Zielsystemen ausgetauscht. Im Falle des Simulators BREVE wurde ein TCP/IP -Server als Zusatzkomponente (*Plugin*) angebunden, für die ELFE hingegen wurde ein eigenes Protokoll entwickelt, um über eine serielle Bluetooth-Schnittstelle zu kommunizieren.

5.2. Evolvierung der Individuen in ECJ

Die zentrale Komponente des Versuchssystems bildet der GP-Kernel. Er ist verantwortlich für die Steuerung und Organisation des gesamten evolutionären Prozesses, angefangen bei der Darstellung der Individuen als interne Kodierung, über die Initialisierung der Startpopulation, bis hin zur Selektion und Anwendung genetischer Operatoren. Inzwischen sind einige Softwaresysteme zur Berechnung von künstlichen Evolutionen frei erhältlich, die sich in Umfang und Zielplattform voneinander unterscheiden. Bei der Wahl des GP-Kernels ist darauf zu achten, dass Kriterien wie Performanz, Konfigurierbarkeit der Parameter und Erweiterungsmöglichkeiten den Anforderungen des Versuchsaufbaues entsprechen.
Für die Experimente mit der ELFE war insbesondere die Flexibilität des Kernels von Interesse. Es musste möglich sein, eine Kommunikationsschnittstelle anbinden zu können, um die Individuen extern über den Simulator bzw. den Roboter auswerten lassen zu können. Darüber hinaus wurde Wert auf umfangreiche Optionen zur Kontrolle und Protokollierung der Evolution gelegt. Als Entscheidungshilfe wurde die Arbeit von [GP06] herangezogen. Die Autoren untersuchten sechs der populärsten GP-Frameworks und verglichen sie mit einem Fokus auf die Flexibilität der gesamten Systemarchitektur. Die besten Bewertungen wurden dabei für das *C++* Projekt *OpenBEAGLE* [GP], welches von den Autoren selbst entwickelt wurde sowie für das auf Java basierenden *ECJ* [LPB$^+$] vergeben. Beide Kernel wurden für die Verwendung in dieser Arbeit getestet und zeigten in Umfang, Erweiterbarkeit und Bedienbarkeit vergleichbare Eigenschaften. Letztendlich wurde aufgrund der persönlichen Neigung des Autors das ECJ gewählt.

5.2.1. Architektur des ECJ

Das ECJ (*Evolutionary Computation Research System for Java, Version 15*) ist ein in Java umgesetzter Evolutionskernel, der für die Arbeit mit baumbasierter GP wie auch für ES und GA entwickelt wurde. Dabei ist beim Entwurf der Klassenbibliothek auf maximale Flexibilität und Erweiterbarkeit Wert gelegt worden: Alle Komponenten, wie Individuen, Brüter oder Operatoren sind in einer strengen Klassenhierarchie organisiert und können bei Bedarf über Schnittstellendefinitionen ausgetauscht werden. Änderungen des Systems und die Parameter des Evolutionslaufs werden über umfangreiche Konfigurationsdateien festgelegt.
Der Eintrittspunkt des Systems bildet die Konsolenapplikation der Klasse `Evolve`. Sie wird mit einer Hauptkonfigurationsdatei geladen, die alle nutzerspezifischen Änderungen und die Bestandteile des GP-Problems angibt. Nach dem Prinzip des *dependency injection* werden die Klassen automatisch instanziert und in die Architektur eingebunden. Die Abb. 5.2 zeigt anhand eines vereinfachten Klassendiagramms, welche Komponenten vom Nutzer zu definieren und in das Framework zu integrieren sind, um einen minimalen Evolutionslauf vorzubereiten:

Das Problem definiert das Ziel der Evolution. Als Subklasse von `GPProblem` wird es durch das Framework aufgerufen, um die Individuen auszuführen und ihnen Fitnesswerte zuzuordnen.

Ein Datenobjekt muss von der Klasse `GPData` erben. Es kapselt einen beliebigen Datentypen innerhalb einer standardisierten Schnittstelle und wird während der Evaluationsphase als einheitlicher Träger der Funktionsergebnisse genutzt.

Funktionen werden durch eine eigene Klasse vom Typ `GPNode` repräsentiert. Sie enthalten die Ausführungsvorschrift zur Berechnung des Funktionsergebnisses sowie eine formale Beschreibung. Darüber hinaus können zusätzliche Beschränkungen für das Anhängen von Kindknoten formuliert werden.

Terminale werden entweder über Variablen oder *ERC*-Knoten eingebunden. Im Falle der Variablen wird eine Subklasse von `GPNode` gebildet, sie verhält sich wie eine Funktion ohne Kindknoten und liefert als Ergebnis den Wert einer Klassenglobalen Variable zurück. Konstanten hingegen werden von der abstrakten Klasse ERC (Ephemeral Random Constants, Vgl. [Koz92]) abgeleitet.

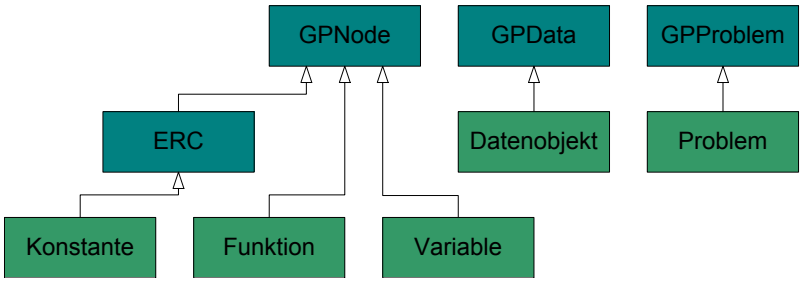

Abbildung 5.2.: Integration problemspezifischer Klassen (grün) in das ECJ Framework (blau)

Die hohe Modularisierung der Komponenten erlaubt zwar ein hohes Maß an Flexibilität, fordert aber gleichzeitig verhältnismäßig viel Einarbeitungszeit. Zusätzlich erschwert die Entscheidung des Entwicklers, zu Gunsten der Performanz auf Grundprinzipien der Objektorientierung zu verzichten, die Orientierung innerhalb der Klassenbibliothek: Variablen sind generell ohne Zugriffsschutz deklariert, darüber hinaus wird auf viele Komponenten statisch oder per Singleton-Muster zugegriffen.
Um die weitere Funktionsweise des Kernels zu verdeutlichen, wird in vereinfachter Form der Aufbau eines Individuums näher betrachtet. Wie in Abb. 5.3 zu sehen ist, besitzt jedes GP-Individuum eine Referenz auf ein `GPTree`-Objekt, welches den Syntaxbaum verwaltet. Grundsätzlich kann in ECJ einem Individuum mehrere Bäume von verschiedenartiger Struktur zugewiesen werden, diese Möglichkeit wird in dieser Arbeit jedoch nicht verwendet. Die Konstruktionsanleitung eines Syntaxbaumes enthalten die `GPTreeConstraints`, sie definieren über `GPFunctionSet`, welche Funktionen und Terminale verwendet werden dürfen und welche Arität diese besitzen. Zusätzlich gehören

sie einem `GPType` an, der zur Unterscheidung verschiedenartiger Bäume dienen kann. Die Knoten des Baumes werden über eine doppelt verkettete Liste des Typus `GPNode` realisiert, dessen Wurzel über das `GPTree`-Objekt zugänglich ist. Jedem Individuum ist ein Fitnessmodul zugeordnet, welches dessen Raw-Fitness speichert und sie je nach Ausprägung in verschiedenen Metriken wiedergeben kann. Für GP-Probleme ist die Klasse `KozaFitness` vorgesehen, welche Standardized und Adjusted Fitness unterstützt.

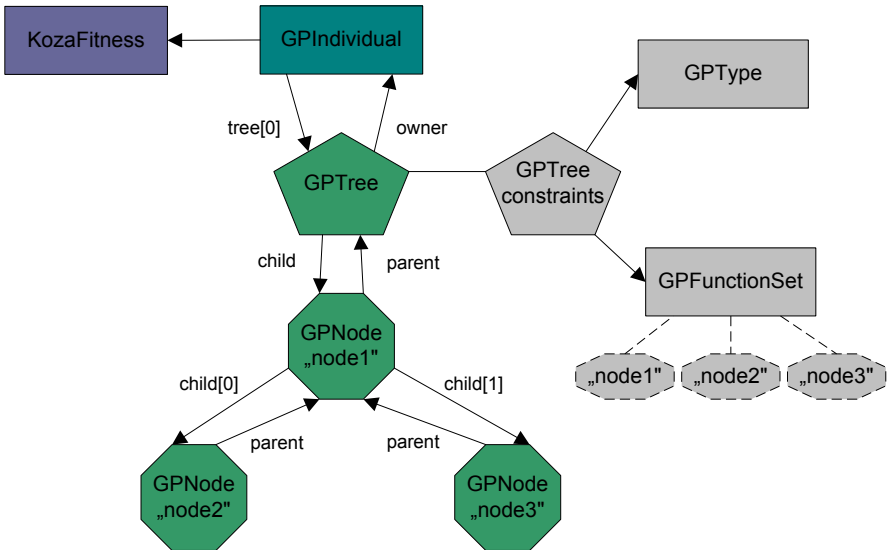

Abbildung 5.3.: Vereinfachter Aufbau eines GP-Individuums in ECJ

Die Auswertung eines Individuums während des Evaluationslaufes zeigt das Sequenzdiagramm in Abb. 5.4. Nahezu alle Objekte sind zu diesem Zeitpunkt bereits vom Framework über die Angaben der Konfigurationsdateien initialisiert worden. Nach der Erstellung einer GP-Population werden nacheinander die Individuen an die Evaluierungsmethode der Problemklasse zur Bewertung weitergereicht. Dort wird zunächst ein Datenobjekt der Klasse `GPData` erstellt, welches die Rückgabewerte der Funktionsknoten zwischenspeichert. Daraufhin wird die Baumstruktur des aktuellen Individuums angefordert, über dessen Wurzelknoten (Typ: `GPNode`) die Abarbeitung gestartet wird. Zur anschliessenden Bewertung wird die Güte des Individuums in dessen Fitnessobjekt vom Typus `KozaFitness` festgelegt. Im letzten Schritt wird das Individuum markiert, um bei doppelter Selektion (z. B. Turnierselektion) erkennen zu können, dass es bereits ausgewertet worden ist.

Solange die Individuen lokal im GP-Kernel ausgeführt und bewertet werden, ist der beschriebene Ablauf allgemein für jede GP-Evolution, die in ECJ beschrieben wird, gültig. Abhängig von der Art des Problems stammt der Fitnesswert dabei aus verschiedenen Quellen. Ist nur ein einziger Funktionswert von Interesse, kann allein der Rückgabewert des Wurzelknotens die Lösung darstellen. Bei mehreren Problemparametern müssen hingegen Variablenknoten in den Baum eingefügt werden, die als Seiteneffekt in klassen-

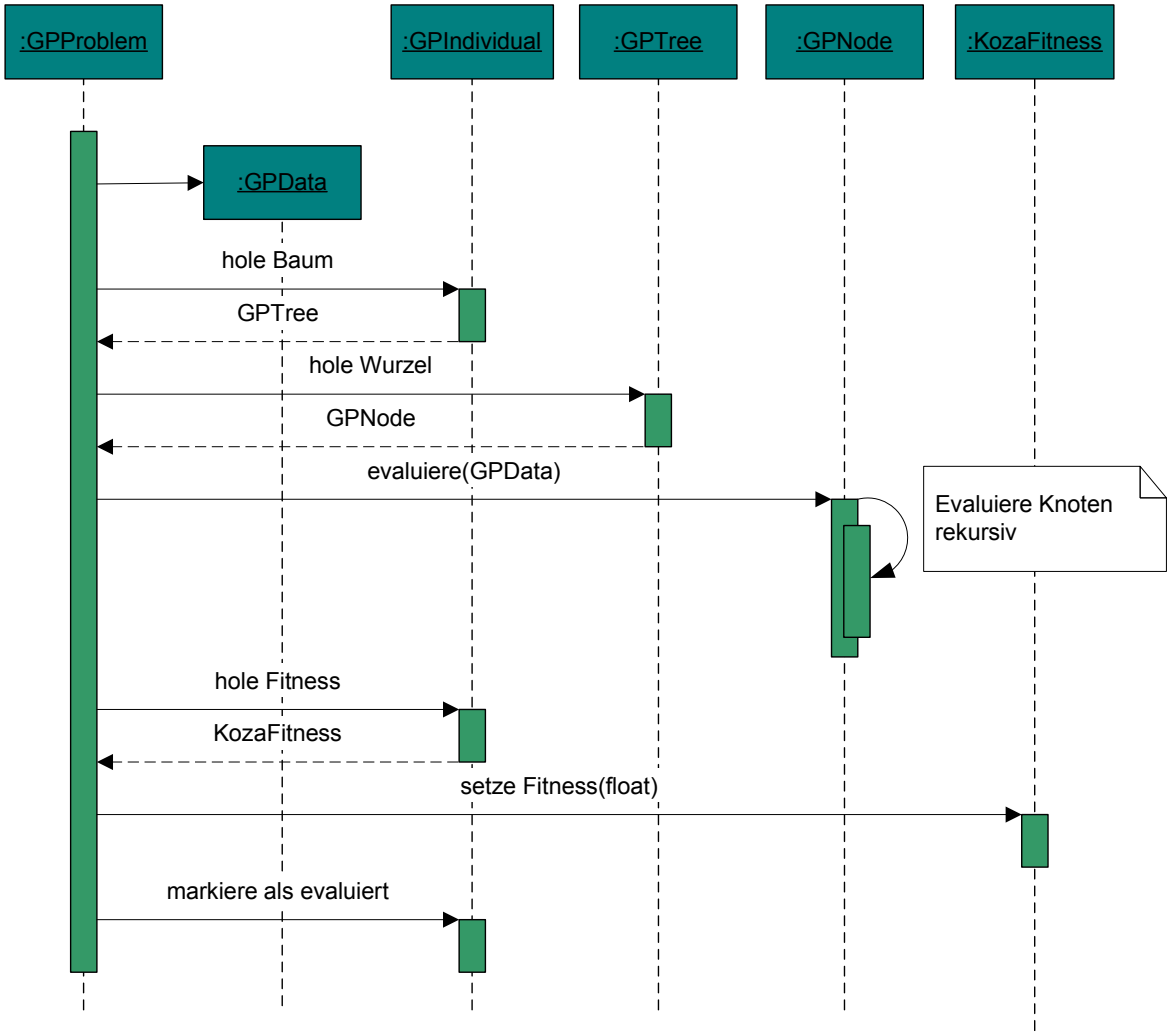

Abbildung 5.4.: Evaluierung der Individuen in ECJ

globale Variablen bsp. der Problemklasse schreiben. Durch die Nutzung eines solchen globalen Speichers erweitert ECJ die Möglichkeiten der Individuen im Gegensatz zu klassischen Lisp-Strukturen, wie bereits in Kapitel 2.6 besprochen worden ist. Als dritte Möglichkeit kann das Resultat von den Seiteneffekte auf ein externes System abhängen, wie es bei der ELFE der Fall ist. Dabei wird nicht die Berechnung des Syntaxbaumes, sondern der Erfolg des Steuerprogrammes in Zusammenhang mit dem Roboter bewertet.

5.2.2. Erweiterungen des ECJ

Um das ECJ-Framework für das ELFE-Problem vorzubereiten, mussten diverse Änderungen und Erweiterungen vorgenommen werden. Dies umfasst neben den notwendigen Klassen zur Integration des Problems eine Erweiterung der GP-Grundtypen, eine Kommunikationsschnittstelle zum Austausch der Individuen mit der ELFE und Modifikationen des experimentellen GUI.

Funktionen und Terminale

Bei der Konfiguration des Kernels muss eine der ersten und wichtigsten Entscheidungen zur Lösung eines GP-Problems getroffen werden, die Festlegung auf eine Funktions- und Terminalmenge. Sie bestimmen die Möglichkeiten, die dem interpretierenden System offen stehen: Ist die Menge der Operatoren und Terminale zu restriktiv gewählt, können nicht alle Lösungen erschlossen werden, eine zu große Auswahl hingegen erweitert den Suchraum unnötig und verlangsamt den Evoluationsprozess. Da es ohne umfangreiches Vorwissen keine sicheren Auswahlkriterien gibt, wurde der allgemeinen Empfehlung gefolgt, den Wertebereich so klein wie möglich zu halten und nur wenige, einfache mathematische Operatoren zu wählen (Vgl. [BNKF97]).
Da die Steuerprogramme letztendlich auf dem AKSEN-Board ausführbar sein sollten, wurde die Terminal- und Funktionsmenge an dessen Möglichkeiten angepasst. Die meisten Funktionen, die für die Steuerung der ELFE benötigt werden, arbeiten mit einem Wertebereich von 0 bis 255. Als grundlegender Datentyp für die Syntaxbäume wurde daher ein vorzeichenloses Byte gewählt. Eine Ausnahme bildet der `sleep` Aufruf der AKSEN-API, welcher die Anzahl an Millisekunden zum Pausieren als vorzeichenlosen long erwartet. Diese erfährt durch den vereinbarten Wertebereich eine Einschränkung, so dass max 255 ms ununterbrochen pausiert werden könnten. Insbesondere für die rekursive Abarbeitung der Individuen auf dem Mikrocontroller-Board ist es jedoch von Vorteil, kleine Datentypen zu wählen, die nur geringe Stacklast erzeugen. Um dennoch längere Unterbrechungen zu ermöglichen, wird daher jeder Sleep-Knoten um den Faktor zehn skaliert. Als Funktionen wurden im GP-Kernel die vier Grundrechenarten, Befehle zur

Ansteuerung der Servomotoren und Infrarotsensoren sowie eine Verzweigungsanweisung implementiert. Darüber hinaus wurde eine Funktion in Anlehnung an Kozas `PROGN`-Funktion aufgenommen, welche die Abarbeitung einer Befehlssequenz erlaubt, die nicht hierarchisch gegliedert sein muss (Vgl. [Koz92, S. 148]).
Weil der Syntaxbaum im Kernel nur aufgebaut, aber nicht ausgewertet wird, ist es nicht möglich, ohne zusätzliche Protokolle zu gewährleisten, dass verschiedene Evaluationssysteme die Implementierung der Funktionen auf gleiche Weise vornehmen. Für die hybride Auswertung von Simulator und Roboter müssen die Individuen jedoch exakt gleich interpretiert werden, um das Ergebnis miteinander vergleichen zu können. Daher wurde eine Vereinbarung entworfen, welche für jedes auswertende System verbindlich ist und vom Evaluator der ELFE befolgt wird. Sie beschreibt die vom Kernel zur Verfügung gestellten Operatoren und die Art, wie diese zu berechnen sind.
Ebenso muss ein einheitliches *closure*-Prinzip verfolgt werden. Da bei den arithmetischen Berechnungen die Grenzen des erlaubten Wertebereichs überschritten werden können, wird jedes Ergebnis definiert als: $ergebnis = |ergebnis\%256|$.

Funktion	**Parameter**	**Rückgabe**	**Name**
Addition	arg0, arg1	$arg0 + arg1$	+
Subtraktion	arg0, arg1	$arg0 - arg1$	-
Multiplikation	arg0, arg1	$arg0 \times arg1$	*
Geschützte Division	arg0, arg1	$arg1 = 0\ ?\ 1 : arg0/arg1$	/
Sequenz	arg0..arg3	$arg3$	prog4
Servostellung	servoNr, stellung	$stellung$	setservo
IR Sensor	sensorNr	$sensorNr$	ir
Verzweigung	arg0.. arg3	$arg0 >= arg1\ ?\ arg0 : arg1$	ifgt
Pause	laenge	$laenge * 10$	sleep

Tabelle 5.1.: Definition der implementierten Funktionen

Das Ergebnis ist in Tabelle 5.1 zu sehen. Neben einer Kurzbeschreibung der Funktion ist der ersten Spalte ist daneben deren Arität (*Parameter*) und die Formel zur Berechnung des Rückgabewerts zu sehen. Die letzte Spalte enthält den Namen der Funktion, welcher als Zeichenkette vom GP-Kernel verwendet wird, um die S-Expression für die Übertragung des Individuums zu konstruieren.

Das ECJ bietet zwar mehrere Möglichkeiten zur Darstellung von Individuen als Datenstrom, unter anderem auch die für das ELFE-Protokoll anvisierten Lisp-Ausdrücke. Allerdings wird diese Form nur intern zur Ausgabe auf die Protokoll-Dateien benutzt und ist nicht über eine Zugriffsmethode abrufbar. Wie in Abb. 5.2.2 zu sehen ist, wurden daher die Klassen der Individuen-Komponente abgeleitet und um die Methode `getAsLispExpression` erweitert. Die durchwandert rekursiv den Baum, sammelt über die `toString()`-Methode die Bezeichner der Knoten ein und fügt sie zu einer S-

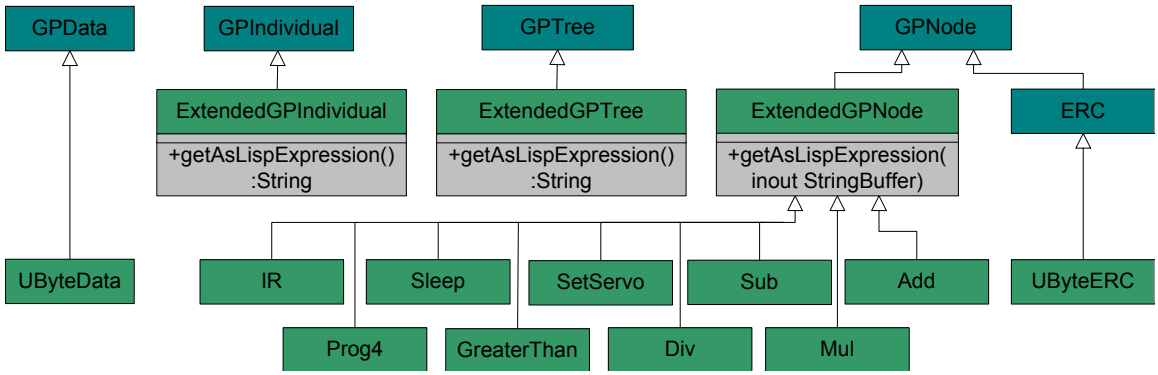

Abbildung 5.5.: Ableitung der Funktions- und Terminalklassen (grün) aus den Datentypen des Frameworks (blau)

Expression zusammen. Um zu gewährleisten, dass diese zusätzliche Funktionalität innerhalb des Frameworks verfügbar ist, müssen die modifizierten Klassen `ExtendedGPIndividual` und `ExtendedGPTree` in die Konfigurationsdatei des Problems eingetragen und alle Knoten von `ExtendedGPNode` abgeleitet werden.

Kommunikationsschnittstelle

Da der Kernel die Ausführung der Individuen nicht selbst übernehmen kann, wurde eine Kommunikationsschnittstelle in das Framework integriert. Bei der Wahl des Kommunikationsprotokolls zum Austausch eines Individuums gegen seinen Fitnesswert wurde besonders auf eine unkomplizierte und systemunabhängige Implementierung geachtet. Jede Übertragung geschieht in Form einer Zeichenkette, wobei Individuen als S-Expressions und Fitnesswerte als stringbasierte Fließkommazahl gesendet werden. Die Wahl zur Verwendung von S-Expressions für die Individuen begründet sich auf zwei Vorteile. Zum einen bieten sie eine kompakte Möglichkeit zur Repräsentation einer Baumstruktur, zum anderen sind sie auch für den Menschen leicht zu interpretieren.
Auf zusätzliche Metainformationen, wie Handshakes, Sequenznummern oder Konfigurationsvereinbarungen wurde gänzlich verzichtet. Die Abfolge der Kommunikation verhält sich folgendermaßen: Der GP-Kernel sendet ein Individuum und wechselt daraufhin in den Lesemodus, um die Fitness zu empfangen. Das auswertende System empfängt das Individuum, führt es aus und sendet den Fitnesswert zurück. Werden in der Zeit der Auswertung durch einen Fehler weitere Individuen empfangen, so werden diese ohne Benachrichtigung verworfen. Erhält der GP-Kernel hingegen nach einer ausreichend lang gewählten Zeit keine Fitness, so sieht er die Transmission als gescheitert an und sendet das Individuum erneut. Dieses Protokoll erlaubt zwar keine Propagierung eines Fehlers durch das Netzwerk, dafür können die Systeme unabhängig voneinander neu gestartet werden, ohne einen alten Zustand wiederaufnehmen zu müssen.

Anwenderoberfläche

Das ECJ-Framework stellt seit Version 15 eine experimentelle Anwenderoberfläche zur Verfügung. Die Autoren betonen zwar, dass für deren Funktionalität und Sicherheit noch keine Garantie gegeben werden kann, allerdings stellte sich die Oberfläche als nützliches Werkzeug für die Versuche dieser Arbeit heraus. Da die Projektseite von ECJ noch keine Dokumentation zu diesem Thema bereitstellt, wird deren Aufbau kurz erläutert.

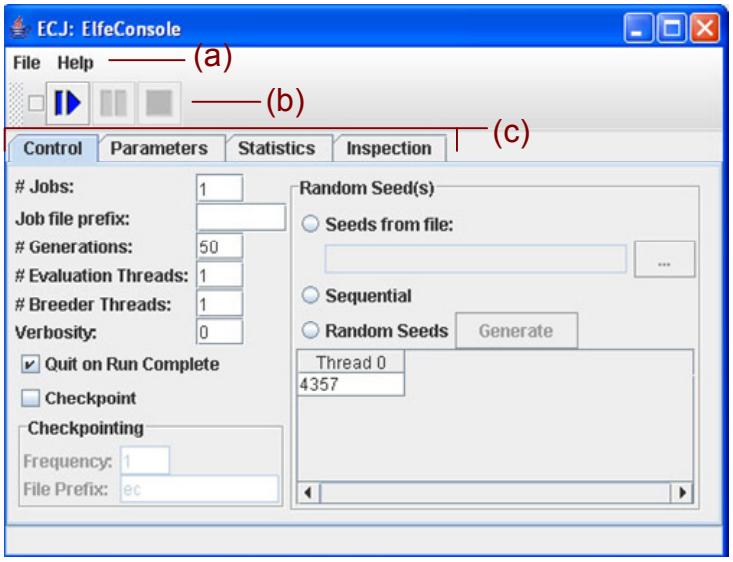

Abbildung 5.6.: Die Anwenderoberfläche des ECJ

Wie die Abb. 5.6 zeigt, besteht die Anwenderoberfläche aus einer Menuleiste zum Laden von Parameterdateien und Evolutionsständen (a), einer Kontrollleiste zum Starten, Pausieren und Stoppen der aktuellen Evolution (b) sowie vier Karteireitern (c), deren Aufgabe sich wie folgt gliedert:

Control erlaubt die Einstellung der wichtigsten Parameter, ohne die Konfigurationsdateien zu verändern. Unter anderem handelt es sich dabei um die Anzahl der Generationen, die Möglichkeit, checkpoint-Dateien zu erstellen oder die Aktivierung von zusätzlichen Threads. Zudem kann unter verschiedenen Varianten zur Generierung von Zufallszahlen gewählt werden.

Parameters zeigt als Baumstruktur alle geladenen Variablen der Konfigurationsdateien an. Falls mehrmals ein Parameter mit gleichem Namen geladen wurde, lässt sich zwischen ihnen über ein Menu wählen. Ansonsten aktiviert ECJ den stets zuletzt definierten Parameter für die Evolution. In diesem Reiter sollten jedoch keine Veränderungen vorgenommen werden, da häufig Parameter aus anderen Bereichen unwiederruflich in die ausgewählten Felder kopiert werden.

Statistics ist für die statistische Darstellung der Evolution über Grafen gedacht. Im Paket `ec.app.gui` befindet sich ein einfaches Beispiel zur Anzeige des besten Individuums einer Generation, welches aber nicht in die Oberfläche integriert wurde. Hierfür werden zwei zusätzliche Frameworks benötigt: *JFreeChart*[3] zur Erstellung des Funktionsplots sowie *IText*[4], um den Grafen in eine pdf-Datei zu wandeln.

Inspection stellt alle Parameter der laufenden Evolution dar. Zusätzlich können die Fitnesswerte und Syntaxfolgen aller Individuen der aktuellen Population betrachtet werden.

Damit die Experimente visualisiert werden können, wurde das Statisics-Beispiel unter Verwendung der erwähnten Frameworks erweitert und in die Anwenderoberfläche integriert. Hierbei zeigte sich das frühe Entwicklungsstadium der grafischen Komponenten: Die Klassen sind wenig flexibel organisiert und daher nur schwierig zu erweitern.
Jede Klasse, die als Plot aufgenommen werden soll, muss von `XYSeriesChartStatistics` erben, um während des dynamischen Ladens korrekt instanziert zu werden (Vgl. Abb. 5.7). Für das ELFE-Projekt wurden drei Klassen zur statistischen Auswertung umgesetzt:

AverageBestChart zeigt die jeweils durchschnittliche und beste Fitness der Population über alle Generationen hinweg. Sie arbeitet nur mit der Klasse `KozaStatistics` des ECJ Frameworks zusammen, wodurch Fitness entweder als standard- oder adjusted fitness interpretiert wird.

SimpleAverageBestChart zeigt ebenfalls die durchschnittliche und beste Fitness der Populationen an. Die Klasse wertet allerdings nur Daten der statistischen Klasse `SimpleStatistics` des Frameworks aus, um raw fitness darstellen zu können, bei der höhere Werte einer besseren Güte entsprechen.

AverageNodesChart berechnet die durchschnittliche Anzahl der Knoten einer Population und zeigt den Verlauf über die Generationen hinweg.

Zur Integration eines Graphen ist es ausreichend, die abstrakte Methode `postEvolutionStatistics` zu überschreiben. In dieser wird definiert, welche Daten in den Plot über `addDataPoint` eingetragen werden. Sollen hingegen mehrere Kurven gleichzeitig angezeigt werden, müssen zusätzlich die Methoden `setup` und `makeChart` neu definiert werden. Über `setup` werden die neue Datensätze angemeldet, während über `makeChart` die Erscheinungsform des Plots konfiguriert werden kann.

[3]Kostenfrei erhältlich unter: http://www.jfree.org/jfreechart
[4]Kostenfrei erhältlich unter: http://www.lowagie.com/iText

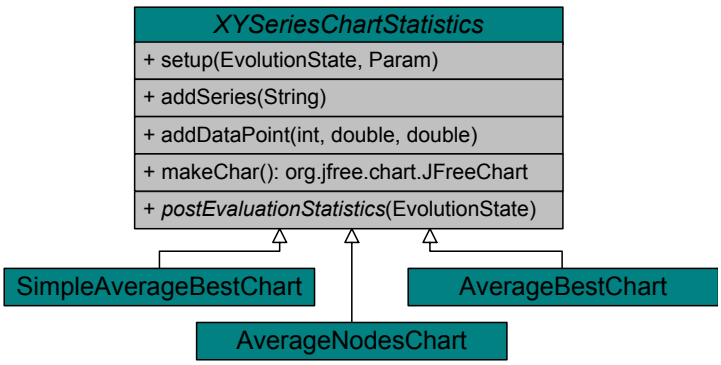

Abbildung 5.7.: Integration der Klassen zur Visualisierung der Evolution

5.2.3. Bewertung der Lösung

Die Arbeit mit dem ECJ hat sich als äußerst produktiv erwiesen. Das Framework zeigt sich durch die hohe Modularisierung als flexibel für eigene Erweiterungen, bietet jedoch genügend vorgefertigte Klassen, um die meisten GP-Probleme mit minimalem Implementationsaufwand umzusetzen. Insbesondere durch die Möglichkeit, alle Aspekte des Evolutionslaufs über Konfigurationsskripte zu bestimmen, beschleunigt die Entwicklung erheblich. Die Mächtigkeit des Frameworks hat jedoch auch den Preis einer verhältnismäßig hohen Einarbeitung, um die Struktur der vielfältigen Komponenten zu durchschauen und aus den hunderten Parametern die relevanten herauszusuchen. Ebenfalls gewöhnungsbedürftig sind viele Designentscheidungen, die aufgrund von Performanzgründen getroffen wurden. Der häufige Verzicht auf das Kapselungskonzept und die Verwendung von statischen Objekten lässt am Anfang die Frage offen, wie auf wichtige Teile des Systems zugegriffen werden kann (z. B. Evaluierung eines Individuums).
Durch die Flexibilität des ECJ konnten alle Klassen des ELFE-Problems als Erweiterungen umgesetzt, ohne den Kernel selbst verändern zu müssen, so dass die Kompatibilität auch mit neuen Versionen des ECJ gewährleistet ist. Von der Norm der GP-Probleme in ECJ weichen nur zwei Konzepte ab: Es wurden nicht die vorgesehenen Grundtypen (GPIndividual etc.) benutzt, sondern abgeleitete Klassen, um die protokollkonformen S-Expressions erzeugen zu können. Zudem wurde die Auswertung der Individuen über ein externes System vorgenommen, wodurch eine zusätzliche Kommunikationsschnittstelle integriert werden musste. Der Zugriff auf die Schnittstelle ist dabei im Stil des Frameworks realisiert worden, um auch ohne Referenzübergabe an dessen Funktionalität nutzen zu können. Dabei wurde mit der Realisierung eines seriellen und eines internetbasierten Protokolls gezeigt, dass die zugrundeliegende Technik problemlos ausgetauscht werden kann.
Insbesondere die experimentelle Anwenderoberfläche gewährt eine komfortable Bedienung des Programmablaufs. Dabei ist die Entscheidung der Entwickler hervorzuheben, für die Visualisierung der Evolution die Verwendung der Frameworks iText und JFreeChart zu demonstrieren. Auf diese Weise konnte ohne großen Aufwand eine eigene Visualisierung entworfen werden. Da die grafischen Komponenten noch nicht als fester

Bestandteil des Frameworks gesehen werden, ist allerdings zu erwarten, dass zukünftige Versionen von ECJ Veränderungen an den Schnittstellen vornehmen werden.

5.3. Reräsentation der Individuen im Evaluator

Jede Repräsentationsform eines Problems ist zunächst nur eine Vereinbarung zur Strukturierung und Abarbeitung von Wissen. In den meisten Fällen unterscheidet sich jedoch die vom Anwender modellierte Datenstruktur von der natürlichen Darstellungsweise des umgebenden Computersystems. Wie in Kapitel 4.1 beschrieben, wird für die Genetische Programmierung vorzugsweise die Sprache Lisp eingesetzt, da sie die Abarbeitung von Baumstrukturen ohne Zwischenschritt erlaubt. Soll hingegen eine Repräsentationsform interpretiert werden, welches keine native Unterstützung durch das ausführende System erfährt, so müssen die notwendigen Details manuell implementiert werden.
Im Folgenden wird eine Datenstruktur beschrieben, mit der es möglich ist, S-Expressions von einem C-Programm auf einfache Weise rekursiv abarbeiten zu lassen. Dieser Ansatz basiert weitgehend auf den Ideen, wie sie in [KM94] vorgestellt wurden. Zudem wird alternativ ein Algorithmus zur iterativen Ausführung der Strukturen vorgestellt.

5.3.1. Struktur der Individuen in C

Eine S-Expression ist in der Verwendung mit einem C-Compiler nur ein beliebiger String ohne Ausführungsmöglichkeit. Um den Syntaxbaum dennoch interpretieren zu können, muss der Ausdruck zunächst in seine Bestandteile der Funktionen und Terminale geparst werden. Die Token werden dann als Knoten in einer einfachen verketteten Liste angeordnet, bei der jede Funktion Zeiger auf seine Operanden[5] besitzt. Listing 5.1 zeigt die Datenstruktur eines Knotens: Die Variable `type` gibt Auskunft darüber, ob es sich um einen Terminal- oder einen Funktionsknoten handelt, abhängig davon ist `value` mit einem Wert belegt oder bleibt ungenutzt. Um die Verknüpfung zu einem Baum zu schaffen, enthält das Array `args` Zeiger zu jedem Kindknoten. Über den Funktionszeiger `evalFunc` wird eine Evaluationsfunktion referenziert, welche nur für die Berechnung dieses Knotentypes verantwortlich ist.

```
struct Node
{
        unsigned char type;
        unsigned char value;
        struct Node *args[NUM_MAX_ARGS];
```

[5] Diese Top-Down Referenzierung entspricht den S-Expressions in Präfixnotation. Besitzen hingegen die Kindknoten Zeiger auf ihren Vater, muss der Baum Bottom-Up in Postfixnotation interpretiert werden.

```
        unsigned char (*evalFunc)(struct Node* n);
};
```

Listing 5.1: C Implementierung eines Baumknotens

Insbesondere durch die Verwendung von objektorientierten Sprachen könnten einige Aspekte der Struktur flexibler gelöst werden. Beispielsweise wäre es möglich, variable Argumentlisten zu verwenden oder die verschiedenen Knotentypen in einer gemeinsamen Klassenhierarchie zu modellieren. Dieser Ansatz ist jedoch auf dem AKSEN-Board nicht zu verwirklichen, da nur ein reduzierter C-Compiler eingesetzt wird. Zudem verfügt es nicht über eine dynamische Speicherverwaltung, Reservierungen für Variablen müssen somit zur Kompilierungszeit angekündigt worden sein. Die Knoten der verketteten Liste werden aus diesem Grund nicht über den *new*-Operator alloziert, sondern in Zellen eines Arrays mit definierter Länge abgelegt. Die Abbildung 5.8 zeigt, wie die S-Expression `(+ (- 20 10) 7)` in einem verketteten Array dargestellt wird.

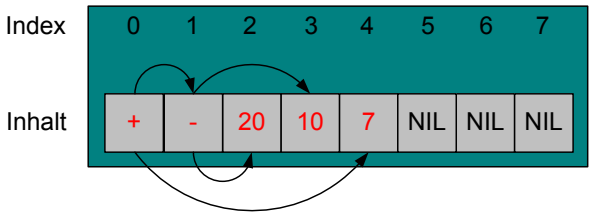

Abbildung 5.8.: S-Expression als verkettetes Array

Wenn die Knoten wie bei der vorliegenden Implementation linear im Speicher angeordnet werden, ist es zunächst nicht notwendig, sie über Zeiger miteinander zu verketten. Bei der Ausführung könnte über einen einzigen Funktionszeiger nacheinander jeder Knoten referenziert und ausgeführt werden. Mit Hilfe der verketteten Liste können jedoch gezielt einzelne Subbäume ohne zusätzliche Berechnung übersprungen werden, was beim Verzweigungsknoten von Bedeutung ist, welcher die Auswahl zwischen zwei Pfaden des Baumes ermöglicht.

5.3.2. Rekursive Methode

Für die Abarbeitung des Individuums navigiert die rekursive Methode durch die verkettete Struktur, indem jeder Knoten innerhalb seiner Evaluationsfunktion die Rückgabewerte seiner Kinder anfordert. Dies wiederholt sich rekursiv für die Kindknoten, bis ein Terminal erreicht wird und somit die Aufrufkette für diesen Zweig aufgelöst werden kann. Wie in Listing 5.2 anhand der Evaluationsfunktion für einen Additionsknoten gezeigt wird, ist die einfache Umsetzung einer der Vorteile der Rekursion.

```
unsigned char evaluateAdd(struct Node* n)
{
        return (unsigned char)
        (
                n->args[0]->evalFunc( n->args[0] ) +
                n->args[1]->evalFunc( n->args[1] )
        );
}
```

Listing 5.2: C Rekursive Evaluierung eines Additions-Knotens

Ein gravierendes Problem der rekursiven Abarbeitung ist die intensive Beanspruchung des Stack-Speichers. Bei jedem Aufruf werden die lokalen Variablen sowie die Rücksprungadressen der Funktion auf dem Stack zwischengespeichert. Während herkömmliche PCs in der Lage sind, tausende rekursive Schritte zu verarbeiten, ist der Stack des AKSEN-Boards auf 255 Bytes je Prozess beschränkt.

5.3.3. Iterative Methode

Das Stack-Problem wird durch die *iterative Methode* umgangen. Hierbei wird das rekursive Verhalten nachgebildet, indem ein Stackspeicher im Heap angelegt wird. Jedesmal, wenn ein Funktionsknoten ausgewertet werden muss, wird seine Adresse auf dem Stack abgelegt und zu seinem ersten Kind gewechselt. Handelt es sich dabei um ein Terminal wird auch dieses auf Stack zwischengespeichert und die Adresse des zuletzt ausgeführten Funktionsknotens geholt. Dieser kann berechnet werden, wenn die Anzahl der Terminale direkt über seiner Stackposition seiner Funktionsstelligkeit entspricht. Sowohl die Funktionsadresse als auch die Terminalwerte werden dann vom Stack entfernt und durch den Rückgabewert ersetzt. Die Auswertung ist beendet, wenn alle Elemente der S-Expression ausgewählt wurden bzw. nur noch ein Terminal auf dem Stack liegt.

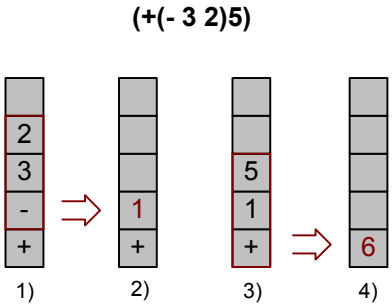

Abbildung 5.9.: Beispiel einer Stackevaluation

Dies soll an einem einfachen Beispiel demonstriert werden: Die Abbildung 5.9 zeigt

den Zustand des Stacks während der Verarbeitung der S-Expression `(+(- 3 2)5)` über mehrere Phasen, welche sich wie folgt ergeben:

1. Die ersten vier Elemente wurden auf den Stack gelegt. Da der zuletzt abgelegte Funktionsknoten vorgemerkt wurde (Subtraktion), ist bekannt, dass die zwei Terminale '3' und '2' reichen, um erstmals einen Teil des Stacks aufzulösen.

2. Die Subtraktion wurde mit den zugehörigen Parametern vom Stack entfernt und durch ihren Rückgabewert ersetzt. Das resultierende Terminal reicht jedoch nicht, um die aktuelle Funktion, den Additionsknoten aufzulösen.

3. Ein weiteres Element, das Terminal '5' wurde aus von der S-Expression auf den Stack gelegt. Die Anzahl der direkt aufliegenden Terminale stimmt nun mit der Arität des Additionsknoten überein.

4. Der Additionsknoten wurde ausgewertet und durch sein Funktionsergebnis ersetzt.

Eine Schwierigkeit bei diesem Algorithmus ist es, nach jeder Auflösung jenen Funktionsknoten innerhalb des Stacks zu referenzieren, welcher als nächstes berechnet werden muss. Dies kann auf verschiedene Weisen erfolgen: Entweder der Stack wird jedesmal vom oberen Ende ausgehend nach dem nächsten Funktionsknoten durchsucht oder alle Funktionen werden auf einem zusätzlichen Stack zwischengespeichert.
Eine elegantere Lösung ist in Abb. 5.10 zu sehen, bei der ein Stack mit zwei Enden benutzt wird, die sich beim Befüllen entgegenlaufen. Die eine Seite wird für die Funktionen, die andere für die Terminale benutzt. Auf diese Weise stehen der letzte geöffnete Funktionsknoten und die zugehörigen Operatoren immer an aktuellster Stelle des jeweiligen Stacks. Der gezeigte Evaluationsstand entspricht dabei der ersten Phase aus Abb. 5.9.

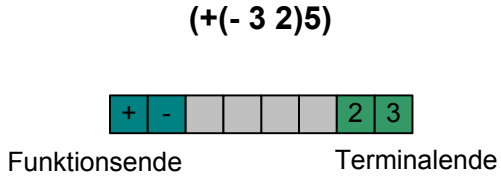

Abbildung 5.10.: Evaluation mit einem doppelseitigen Stack

Problem des Verzweigungsknotens

Bei dem beschriebenen Verfahren wird ein Problem durch den Verzweigungsknoten aufgeworfen. Der Verzweigungsknoten ist meist ein vierstelliger Operator, der es ermöglicht,

zwischen zwei Ästen eines Syntaxbaumes zu wählen. Er vergleicht die ersten beiden Operanden und wählt anhand dessen entweder den dritten oder letzten Zweig zur Berechnung aus.
Daher darf er nicht aufgelöst werden, wenn bereits alle vier Parameter auf den Stack gelegt worden sind, ansonsten würde Algorithmus automatisch beide alternativen Subbäume ausgeführen. Im geringsten Fall entstünden so nur zusätzliche Berechnungen des Prozessors. Werden hingegen auch Knoten mit Seiteneffekten ausgewertet, verändert sich das Ergebnis des Individuums.
Aus diesem Grund ist es notwendig, zunächst die ersten beiden Parameter auf das Funktionsergebnis zu reduzieren und anhand derer zu entscheiden, ob mit Hilfe der Knotenreferenzierung der dritte oder der letzte Parameter gewählt werden soll. Die Abb. 5.11 veranschaulicht dieses Problem. Der rot umrahmte Subbaum in a) stellt einen Verzweigungsknoten dar, der vergleicht ob der erste Parameter größer oder gleich dem zweiten ist. Da es in diesem Beispiel der Fall ist, muss der dritte Subbaum bei Index 4 (angedeutet durch Auslassungszeichen) ausgewertet werden. Dieser kann, wie in b) gezeigt, wiederum ein Verzweigungsknoten sein. Für jede dieser geöffneten Ebenen muss auf einem Stack vermerkt werden, ob der vierte Parameter später ausgewertet werden muss oder nicht.
Darüber hinaus kann sich eine weitere Schwierigkeit ergeben. Angenommen, in a) wurde der gewählte Subbaum ausgeführt, so dass die Alternative, der `setservo`-Befehl bei $Zeiger_{alt}$ übersprungen werden muß. Dann ist durch die vorwärtsverkettete Listenstruktur nicht direkt erkennbar, bei welchem Index das Ende des Befehls ist (angedeutet durch $Zeiger_{neu}$). Dies erfordert wieder die rekursive Zählung aller verschachtelten Unterfunktionen und deren Argumente, bis das Ende des Motorbefehls erreicht ist.

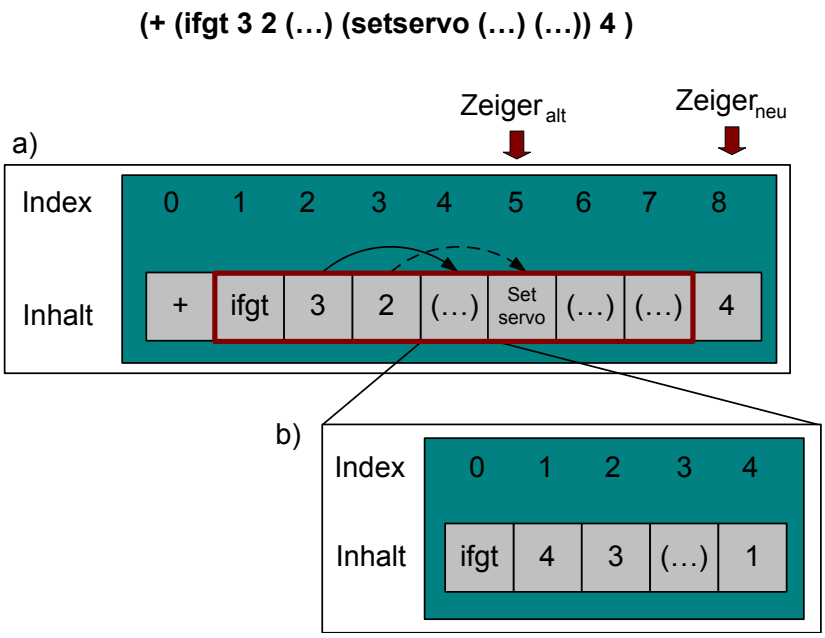

Abbildung 5.11.: Problem des Verzweigungsknotens

5.3.4. Bewertung der Lösungen

5.4. Evaluation der Individuen im Simulator BREVE

Bei der Vorstellung der Evolutionären Algorithmen wurde bereits eines der schwerwiegensten Probleme angesprochen - der zeitliche Faktor. Während die Darstellung und Manipulation der Individuen direkt von der Rechenleistung des Computersystems abhängt, wird die Fitnessbewertung von der Geschwindigkeit des Rückkopplungssignals bestimmt. In vielen Fällen kann die Bewertung jedoch nicht durch steigende Rechenkraft beschleunigt werden: Der Zeitfaktor eines realen Bewerters wird durch die Umwelt begrenzt und ist nicht skalierbar. Bezogen auf die ELFE bedeutet dies, dass jedes Steuerungsprogramm zumindest einmal ausgeführt werden muss, bevor die Fitness gemessen werden kann. Die Auswertung von großen Populationen über vielfache Generationen wird somit schnell zum zeitaufwändigsten Teil des Algorithmus.
Ein weiterer Aspekt ist die mechanische Belastung der Bauteile. Die handelsüblichen Komponenten der ELFE sind nicht dafür ausgelegt, den andauernden Testreihen und der wiederholten Beanspruchung im Grenzbereich standzuhalten. Insbesondere die Servomotoren der Beine bilden eine Schwachstelle, da sie die gesamte mechanische Last der Fortbewegung tragen müssen. Da die Evaluationsreihen ohne Aufsicht stattfinden, hätte ein technischer Ausfall, beispielsweise eines Servomotors weitreichende Auswirkungen auf den weiteren Verlauf. Der GP-Algorithmus wäre zwar in der Lage, weiterhin Steuerungsprogramme zu erzeugen, es würden jedoch wahrscheinlich Lösungen konvergieren, welche die Nutzung des Beines vermeiden.
Die genannten Probleme können durch den Einsatz eines Simulators vermieden werden. Der Roboter wird innerhalb einer virtuellen Umwelt modelliert, die möglicht exakt den Bedingungen des realen Versuches angenähert ist. Dies erfordert bei einem Schreitroboter die Nachahmung einer physikalischen Atmosphäre. Denn nur wenn Faktoren wie Gravitation, Reibung und Plastizität simuliert werden, ist es möglich, die Dynamik eines Körpers als Fortbewegung nachzuempfinden. Die Leitidee ist, den groben Beginn einer Evolution zunächst zu simulieren, um erfolgversprechende Individuen auf dem Roboter zu verwenden oder weiterzuentwickeln. Dies reduziert die mechanische Belastung deutlich, ist zeitlich skalierbar und erlaubt die volle Kontrolle über den Versuchsaufbau. Auf diese Weise können Veränderungen an der Morphologie des Roboters erst am Modell getestet und bei Erfolg übernommen werden. Zudem kann der virtuelle Roboter nach jeder Evaluation in exakt die gleiche Ausgangsposition gebracht werden, um einen direkten Vergleich der Individuen zu erlauben.
Letztendlich bleibt jedoch die Frage nach der Qualität der Simulationsergebnisse. Jede Modellierung ist stets nur eine stark vereinfachte Abstraktion der Anwendungsdomäne. Viele Faktoren, wie der Verschleiß von Bauteilen, Störsignale durch Sonne, elektromagnetische Felder oder die Beschaffenheit von Materialien, werden häufig nicht oder nur unvollständig nachgebildet. In den folgenden Abschnitten wird daher beschrieben, auf welche Weise die ELFE innerhalb der Experimente simuliert wurde, welche Probleme

dabei auftraten und inwiefern eine Übereinstimmung mit dem realen Roboter existiert.

Für die Simulation der ELFE wurden mehrere kostenfreie Systeme in Betracht gezogen. Die Grundanforderungen waren, dass ein 3D-Modell entworfen werden kann, welches in einer physikalischen simulierten Welt frei kontrollierbar ist. Darüber hinaus musste es möglich sein, den Datenaustausch mit dem GP-Kernel über eine Programmierschnittstelle zu realisieren. Die Wahl fiel auf die Simulationsumgebung *BREVE*[6] [Kle03], da nur sie alle Kriterien erfüllte und zugleich von einer aktiven Entwicklergemeinschaft unterstützt wird (siehe Anhang B.1, B.2).

5.4.1. Architektur von BREVE

Das Simulationssystem BREVE ist ein Open Source Projekt, welches im Rahmen einer Master-These [Kle02] entstand und seit 2002 kontinuierlich von seinem Autor Jon Klein weitergepflegt wird. Wie die Abb. 5.12 zeigt, besteht die Entwicklungsversion von BREVE (*breveIDE*) aus drei Komponenten: Über die objektorientierte Skriptsprache *steve* werden die 3D-Modelle zusammengesetzt, die Simulationsparameter definiert sowie der gesamte Programmablauf gesteuert. Wird zusätzlich die integrierte Physik-Engine *ODE*[7] aktiviert, erweitert sich das Modell um Kollisionserkennung, Kinematik und physikalische Parameter, wie Masse und Reibung. Das Ergebnis wird in der letzten Komponente, einem grafischen Ausgabefenster, als dreidimensionale Ansicht über *OpenGL* visualisiert. Zusätzliche Funktionalitäten können in Form von C/C++ -Bibliotheken importiert werden, den sog. *Plugins*. Die Abb. 5.12 zeigt den schematischen Aufbau der Breve-Komponenten.

Im den folgenden Abschnitten werden die Konzepte von BREVE genauer dargestellt und die Lösungsansätze zur Schaffung einer Simulationsumgebung für die ELFE besprochen. Die Abb. 5.4.1 zeigt ein Schema der entwickelten Komponenten: Der sog. Controller markiert den Ausgangspunkt der Simulation und ist für die Initlialisierung des 3D-Modells und das Laden zusätzlicher Plugins verantwortlich. Da Funktionalitäten der ELFE umgesetzt werden mussten, die nicht in der Bibliothek der Skriptsprache enthalten sind, wurden zusätzlich Klassen zur Simulation von Infrarotsensoren und Servomotoren entworfen. Die Kommunikationsschnittstelle mit dem GP-Kernel wurde durch ein Server-Plugin realisiert, welches Individuen als S-Expression empfängt und dem Controller bereitstellt. Ebenso wurde die Abarbeitung der Steuerungsprogramme in ein eigenes Plugin ausgelagert. Dieser Evaluator nimmt die S-Expression entgegen, wandelt sie in eine ausführbare Programmstruktur und delegiert Befehle zum Stellen eines Servos

[6] Zu Beginn wurde die Version 2.3 benutzt, in den tatsächlichen Experimenten hingegen Version 2.5b
[7] Der quelloffene Physikkernel Open Dynamics Engine (ODE) ist unter http://www.ode.org zu finden

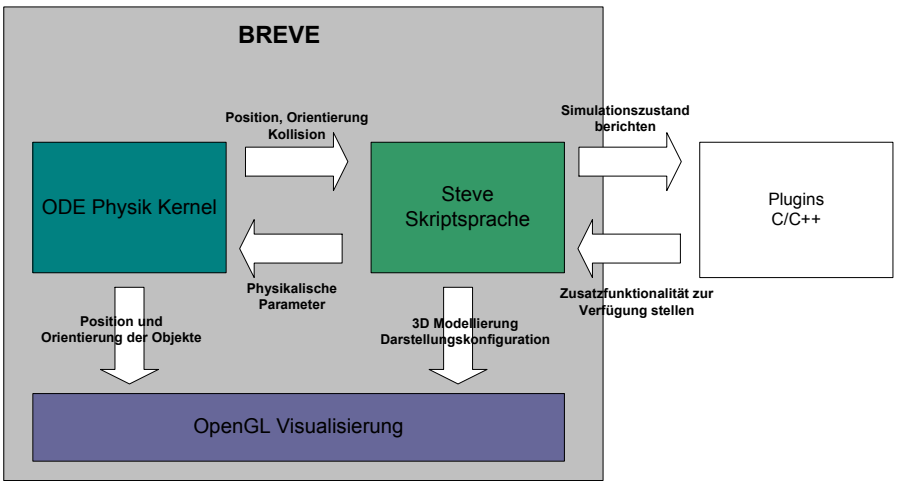

Abbildung 5.12.: Aufbau des BREVE Simulatorsystems

oder zum Auslesen eines Sensors an den Controller weiter.

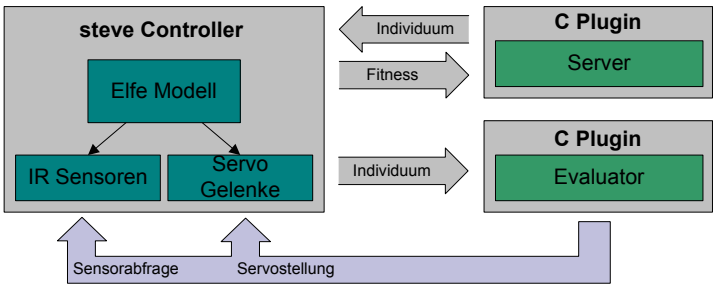

Abbildung 5.13.: Schematischer Aufbau der Simulationskomponenten

5.4.2. Das ELFE-Modell

Zur Erschaffung der 3D-Agenten für die Simulation bietet steve eine kleine Bibliothek an einfachen geometrischen Körpern, die über Verbundelemente, den sog. `Joints`, zu komplexe Modellen zusammengefügt werden können. Joints sind rein virtuelle Objekte, die keine Körperlichkeit besitzen und nur die Art der Verbindung definieren. In BREVE können auf diese Weise Teleskop- und Drehgelenke mit bis zu drei Freiheitsgraden geschaffen werden. Der Zentrale Körper eines Agenten besteht aus einem `MultiBody`, der als Ausgangspunkt für die Verknüpfung weiterer Elemente, den `Links` gedacht ist. Sowohl die Instanzen von MultiBody als auch die von Link definieren ihre physikalische Präsenz über ihre Körperform, die sich aus der Klasse `Shape` ableitet. Obwohl es auf diese Weise möglich ist, nahezu jedes reale Objekt nachzubilden, wird empfohlen,

die Strukturen des Modells möglichst einfach zu halten, solange die physikalische Simulation eingesetzt werden soll. Mit jeder Form steigt exponentiell der Aufwand zur Erkennung von Kollisionen, insbesondere Gelenke können in verschiedenen Situationen zu stark abnormalem Verhalten führen (z. B. bei Verkeilungen, extreme Gewichtsbelastung etc.). Die Herausforderung besteht in der Kreation eines Modells, welches den wichtigsten Eigenschaften des realen Roboters entspricht, ohne durch dessen Komplexität die Möglichkeiten des Simulationssystems zu überfordern.

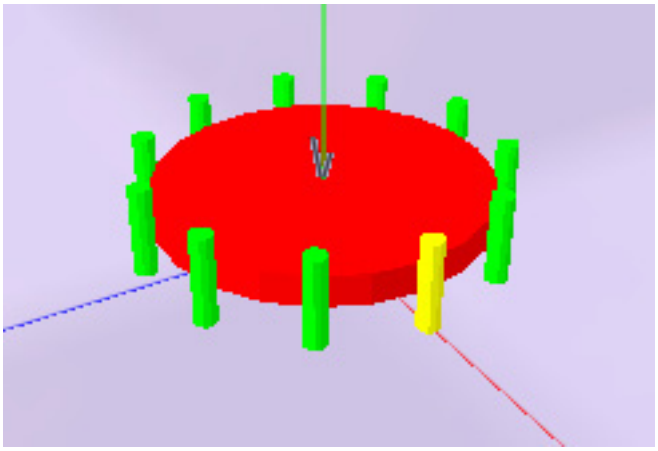

Abbildung 5.14.: Modell der ELFE in der Simulation

Die einfache Architektur der ELFE erlaubte es, ein ausreichendes Simulationsmodell aus einfachen geometrischen Figuren aufzubauen. Der Hauptkörper des Modells besteht aus einem MultiBody in Form einer elfeckigen Polygonscheibe, an deren Rand die Beine als Zylinderform mit 22 Ecken angebracht sind. Die Verknüpfung dieser Elemente erfolgt über ein erweitertes Rotationsgelenk mit einem Freiheitsgrad, welches im folgenden Abschnitt genauer betrachtet wird. Zusätzlich sind am Unterkörper der ELFE vier Abstandssensoren durch statische Gelenke (*FixedJoint*) symmetrisch angebracht. Als Größenmaße wurden die realen Abmessungen in der Standardeinheit des Simulators proportional um den Faktor zehn reduziert. Die Abb. 5.14 zeigt zum Vergleich die Simulation in BREVE mit umgebenden Drahtgitter und lokalem Koordinatensystemen.[8] Bei der Modellierung der ELFE im Simulator konnte nur sehr begrenzt auf eine Übereinstimmung zum realen Roboter geachtet werden. Faktoren, wie die genaue Geschwindigkeit und Drehmomente der Servomotoren, die Reibungskraft der Körpermaterialien oder das Spektrum der Infrarotsensoren konnten innerhalb der gegebenen Zeit mit den zur Verfügung stehenden Ressourcen nicht näher betrachtet werden. Zum anderen gestaltet es sich zunächst schwierig, diese Parameter in BREVE einzustellen, da in der Dokumentation keine Angaben zu der Dimensionierung der physikalischen Einheiten gemacht wird. Eine Anfrage beim Autor ergab lediglich, dass die Simulationsmodelle in

[8] Auf der Grafik sind die Infrarotsensoren nicht zu sehen, da sie sich an der Unterseite befinden.

BREVE meist durch Experimentieren abgestimmt würden. Auf den FAQ Seiten von ODE hingegen ist die Empfehlung zu finden, sich an den Maßen Meter, Kilogramm und Sekunde zu orientieren und bei Bedarf proportional zu skalieren [Unb]. Während der ersten Versuche zeigte sich bereits ein realistischen Verhalten bei einfachen, handerstellten Bewegungen. Allerdings erfuhr die Simulation durch die Verwendung der elf Beine massive Geschwindigkeitseinbrüche. Mit Hilfe des Autors von BREVE konnte der Grund ermittelt werden: Anhand eines Profilers wurde deutlich, dass die Physiksimulation große Probleme bei der Kollisionsberechnung von Objekten hat, die viele Kontaktpunkte zum Boden besitzen. Erst durch die Aktivierung der vereinfachten Physiksimulation, welche laut einer Forumsdiskussion mit Klein den Berechnungsaufwand von $O(n^3)$ auf $O(n)$ reduziere, wurde das Problem behoben. Obwohl dadurch ungenauere und schnellere Verfahren für die Kollisionserkennung angewandt wurden, war keine Beeinträchtigung des Modellverhaltens zu beobachten. Um weitere Fehler zu vermeiden, wurde ebenso die Kollisionsabfrage der Körpersegmente untereinander deaktiviert. Da die Architektur der ELFE verhindert, dass sich Beine überschränken oder in den Körper drehen, wird das Verhalten des Modells dadurch nicht beeinflusst.

Servomotoren

Um das Bewegungsverhalten der Roboters nachbilden zu können, wurde BREVE um einen neuen Typ von Gelenk erweitert. Die Programmbibliothek von steve bietet zwar ein Rotationsgelenk, ähnlich den Servomotoren der ELFE. Dieser sog. *RevoluteJoint* kann jedoch nur über die Drehgeschwindigkeit, nicht aber über einen Stellwinkel gesteuert werden. Zu diesem Zweck wurde der *ServoRevoluteJoint* als Subklasse des RevoluteJoint implementiert. Die Abb. 5.15 zeigt die Funktionalität des Servomotors, wie er in der steve-Sprache als Klasse umgesetzt wurde. Dieser verhält sich wie sein Pendant bei

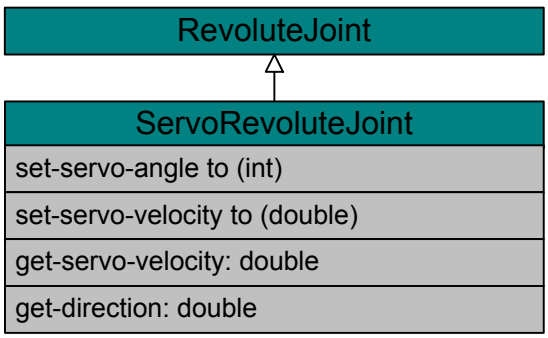

Abbildung 5.15.: Klassendiagramm des ServoRevoluteJoint

der realen ELFE und erlaubt über die methode `set-servo-angle` ganzzahlige Stellungen von 0° bis 180°. Bei Aktivierung des Servos wird zunächst die Drehgeschwindigkeit in Richtung der Winkelposition gesetzt, wodurch sich das Gelenk zu bewegen beginnt.

Um zu erkennen, wann der Motor gestoppt werden muss, wird die aktuelle Winkelposition in radians ausgelesen und als gerundeter Wert im Gradmaß mit der Zielstellung verglichen.

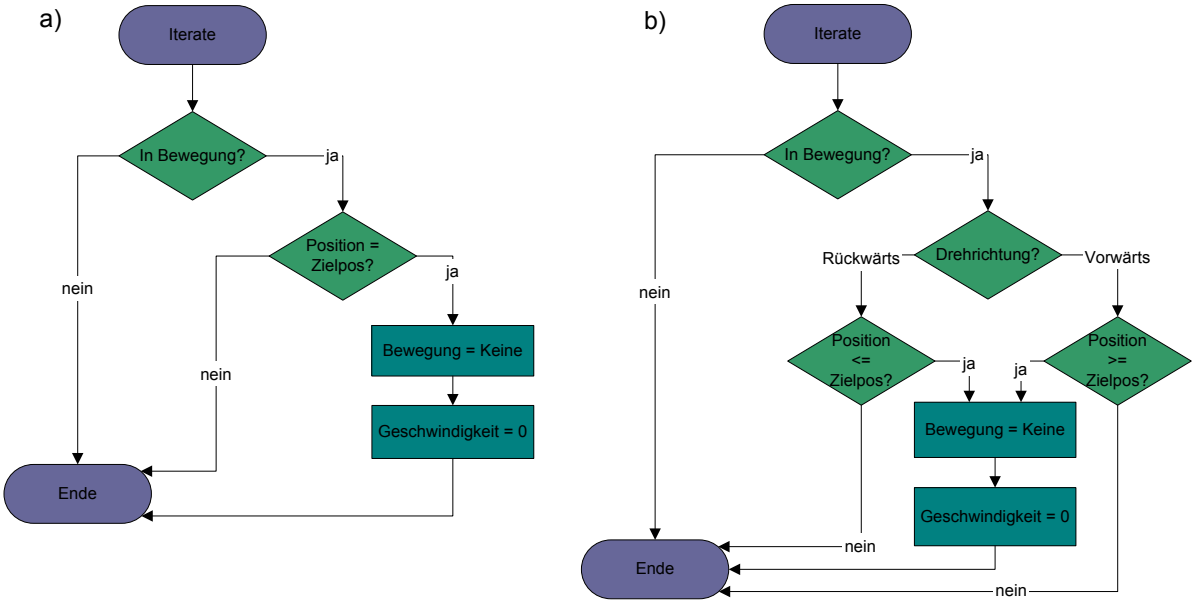

Abbildung 5.16.: Algorithmen zur Überprüfung der Winkelstellung einer Servobewegung: a) mit genauem Zielvergleich, b) mit tolerantem Zielvergleich

Je nach Drehgeschwindigkeit des Servogelenks funktioniert diese Methode jedoch nicht. Um dies näher zu erklären, muss zunächst betrachtet werden, auf welche Weise BREVE Zeit abbildet. Grundsätzlich muss jeder Simulator reale, analoge Zeitverhältnisse auf ein internes Maß diskretisieren. Die kleinste Einheit, die durch den Simulator abgearbeitet wird ist der *Tick*. Er beschreibt die Zustände aller Objekte innerhalb der Simulation zu einem definierten Zeitpunkt. Je höher die Tick-Frequenz ist, desto schneller kann ein Agent auf Veränderungen reagieren. Dieser Parameter wird in BREVE durch den sog. *iteration step* bestimmt, er beschreibt, wie viele Simulatorsekunden bis zur Zählung eines neuen Ticks vergehen müssen. Jeder Agent wird über seine `iterate`-Methode über den Wechsel eines Ticks informiert, damit er seinen Zustand aktualisieren kann.
Innerhalb der iterate-Methode wurde der Algorithmus implementiert, um die Stellung des Servogelenks zu überwachen. Dies ist in 5.16 in zwei Varianten zu sehen: Algorithmus a) vergleicht die aktuelle Winkelposition direkt mit der Endstellung, während b) überprüft, ob die Endstellung bereits überschritten ist. Beide Methoden erzeugen jedoch ein Fehlverhalten, das an einem Beispiel verdeutlicht werden soll. Angenommen, ein Servogelenk bewegt sich ähnlich dem realen Motor mit 180° pro Simulatorsekunde, während der Iterationsschritt auf der Standardeinstellung auf 0.05 gesetzt ist. Der Servoalgorithmus wird somit 20 mal pro Simulatorsekunde aufgerufen, was einer Überprüfung des Winkels alle 9° entspricht. Durch Algorithmus a) würde eine Zielstellung von 102° übersprungen werden, ohne, dass die Abbruchbedingung einträte. Algorithmus b) hinge-

gen würde die Bewegung terminieren, die Winkelstellung wiese aber eine Differenz zum vorgegebenen Wert auf, welche mit zunehmender Drehgeschwindigkeit wachsen würde. Die Abb. 5.4.2 zeigt diese Bewegung des Servos über den Zeitraum einer Simulatorsekunde. Eine vermeintliche Lösung des Problems wäre, die Reaktionsrate des Servomotors

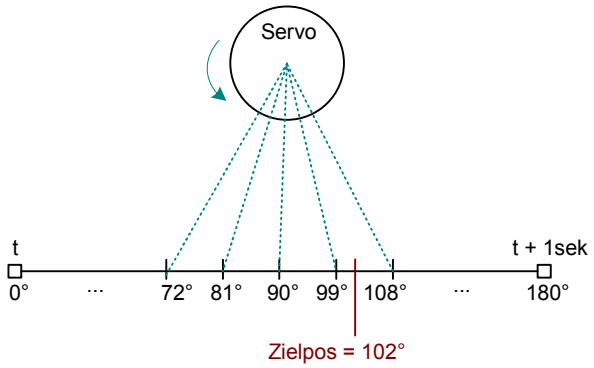

Abbildung 5.17.: Drehung des Servogelenks mit 180° pro Sekunde in Richtung *ZielPos* bei einer Tickfrequenz von 20Hz.

durch die iterations steps so zu erhöhen, dass selbst schnelle Bewegungen genau abgetastet werden können. Jeder zusätzliche Tick bedeutet allerdings die Verarbeitung der `iteration`-Methoden aller Objekte der Simulationsumgebung und erzeugt somit Rechenlast.
Für die Experimente war es somit von entscheidender Bedeutung, einen Kompromiss zwischen der Simulationsgeschwindigkeit und der Genauigkeit der Servomotoren zu finden. Da die Zeitskalierung eines der Hauptargumente für den Einsatz einer Simulation war, muss diese nach Meinung des Autors zumindest in doppelter Realgeschwindigkeit ablaufen. Versuche auf dem Testrechner (siehe Anhang C.1) haben ergeben, dass die Frequenz der iteration steps nicht höher als 30 Hz betragen darf, um diese Bedingung zu erfüllen. Da die virtuellen Gelenke ähnlich den realen Motoren mit einer Rotationsgeschwindigkeit von 180° die Sekunde arbeiten und somit eine Winkelabweichung von bis zu 5° möglich ist, wurde für die Versuche dieser Arbeit der fehlertolerante Algorithmus b) gewählt.

Auf diese Weise verbleibt ein zusätzlicher Ungenauigkeitsfaktor in dem ELFE-Modell. Obwohl es technisch ohne weiteres möglich wäre, die Stellung der Servos zu kontrollieren, lassen es die Möglichkeiten des Simulators aufgrund fehlender Zugriffsmethoden nicht zu. Grundsätzlich bieten alle mobilen und interaktiven Objekte der Simulation für eine Positionsänderung je zwei Methoden. Die eine versetzt ein Objekt unter Berücksichtigung von Kollisionen und der physikalischen Parameter, wie Kollision oder Beschleunigung, so dass die Aktion Teil der simulierten Welt wird. Die andere hingegen führt eine Veränderung sofort und ohne Nebeneffekte durch. Im Falle des `RevoluteJoint` scheint das schlicht vergessen worden zu sein. Das Drehgelenk kann zwar beschleunigt werden, eine unbedingte Korrektur des Stellwinkels ist aber nicht möglich.

Da der Winkelfehler bei den gewählten Einstellungen recht klein bleibt und auch die Servomotoren der realen ELFE nur in einem Toleranzbereich genau arbeiten, ist zu erwarten, dass die Simulation nicht volkommen entwertet wird. Dennoch entfällt ein eingangs erwähnter Vorteil der Simulation: Das Robotermodell kann nicht mehr in jede beliebige Haltung gebracht werden, z. B., um die Startbedingung für jedes Individuum gleich zu gestalten.

Infrarotsensoren

Eine Unterstützung von Sensorik ist innerhalb von BREVE 2.5b kaum vorhanden. Die Umgebung bietet mit der Klasse *BraitenbergSensor* nur einen Empfänger für Licht, dessen Quelle über die zugehörige Komponente *BraitenbergLight* simuliert werden muss und damit ungeeignet als IR-Sensor ist. Um dennoch die Abstandssensoren der ELFE nachbilden zu können, wurde die Klasse *IRSensor* implementiert, welche von `Link` erbt und somit über ein Gelenk an jedem Körper in BREVE befestigt werden kann. Ein IR-Sensor sollte zur optischen Hilfe in der Simulation als kleines, pyramidenförmiges Objekt dargestellt werden, dessen Spitze die Richtung des Strahles zeigt. Allerdings unterstützt BREVE bei festen Verbindungen von Körpern über den `FixedJoint` keine vorherige Rotation. Daher ist jeder Sensor initial eine kleine Platte mit einer extrem kleinen Höhe, wodurch die Richtung des Sensors nur rechnerisch erkennbar ist.
Der IR-Sensor ist als Prototyp anzusehen, welcher auf das ELFE-Experiment zugeschnitten ist und eine wichtige Einschränkung besitzt: Es wird nur der Abstand zum Standarduntergrund (in BREVE die Klasse `Floor`) berechnet. Der Abstand ergibt sich somit aus der Länge des Vektors ausgehend vom Sensorkörper bis zum Schnittpunkt des IR-Strahles mit der XZ-Ebene. Zur Berechnung wird daher zunächst die Gleichung der Ebene benötig:

$$E : \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} = a \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + b \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{5.1}$$

Über die Rotationsmatrix R des Sensorkörpers und der ursprüngliche Richtungsvektor des IR-Strahles $\overrightarrow{heading}$ kann die aktuelle Orientierung $\overrightarrow{dir}$ des Sensors berechnet werden:

$$\overrightarrow{dir} = \overrightarrow{heading} \times R \tag{5.2}$$

Wird $\overrightarrow{dir}$ als Gerade interpretiert (5.3), die durch den Ortsvektor $\overrightarrow{op}$ des Sensors läuft, kann der Skalar berechnet werden, welcher den Durchstoßpunkt der Ebene markiert (5.4). Die Distanz des Durchstoßpunktes zum Ortsvektor des Sensor wird als Ergebnis von der Methode zurückgeliefert. Zur Veranschaulichung sind die Vektoren in Abb. 5.4.2 zusammen mit dem Modell des Sensors gezeigt. Der Sensorkörper ist hierfür als Pyramide dargestellt, um dessen Ausrichtung zum Durchstosspunkt (blaue Markierung)

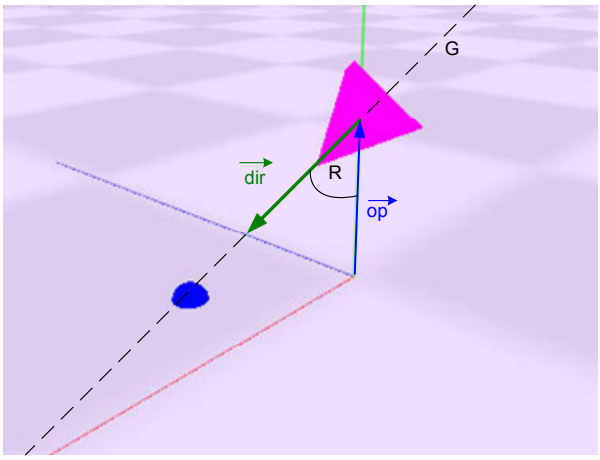

Abbildung 5.18.: IRSensor als Pyramidenform im Simulator. Die blaue Markierung stellt den berechneten Durchstosspunkt zur Ebene dar.

anzudeuten.

$$G : \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = \begin{pmatrix} op_1 \\ op_2 \\ op_3 \end{pmatrix} + t \begin{pmatrix} dir_1 \\ dir_2 \\ dir_3 \end{pmatrix} \tag{5.3}$$

$$op_2 + t \cdot dir_2 = 0 \Rightarrow t = -\frac{op_2}{dir_2} \tag{5.4}$$

Für sehr einfache Simulationen, bei denen nur der Abstand zum ebenen Boden von Interesse ist, mag der Sensor genügen. Dennoch müssen die starken Simplifizierungen des IR-Modells bedacht werden: Stehen Objekte zwischen Sensor und Untergrund, werden diese nicht registriert. Dieser Aspekt ist für diese Arbeit jedoch zu vernachlässigen, da der Untergrund für die ELFE-Simulation plan ist und keine weiteren Objekte enthält. Zudem sind die Sensoren orthogonal am Körper des Roboters befestigt, so dass sie unabhängig von dessen Bewegung stets nur auf den Untergrund ausgerichtet sind.

Wie das Klassendiagramm in Abb. 5.19 alternativ zur Abfrage der absoluten Distanz über `get-distance` kann zusätzlich ein Wert abgefragt werden, welcher beispielhaft zeigt, wie das Verhaltens eines realen IR-Sensor am AKSEN-Board nachzuempfunden werden kann. Durch Aufruf der Methode `get-sensor-value` wird der Abstand zum Untergrund berechnet und als normierter, ganzzahliger Wert im Bereich 0 bis 255 zurückgegeben. Analog zum AKSEN-Board bedeutet hierbei 0 die maximale Sensorerregung, während bei 255 kein Signal erkannt wird. Die Standardreichweite des ELFE-Modells beträgt genau den Abstand vom Untergrund bis zur unteren Körperfläche.

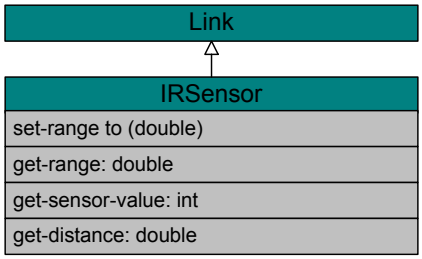

Abbildung 5.19.: Funktionalität der IRSensor-Klasse

5.4.3. Entwicklung von Plugins

Die Möglichkeit, plugins in Form von dynamischen Bibliotheken schreiben zu können, ist eine der herausragendsten Eigenschaften der Simulationsumgebung. Auf diese Weise können beliebige Zusatzfunktionen, wie Kommunikationsschnittstellen oder externe Frameworks angebunden und in den steve-Skripten genutzt werden. Ursprünglich war nur geplant, die Kommunikationsschnittstelle zwischen dem Simulator und dem GP-Kernel als Plugin umzusetzen. Durch die Unzulänglichkeiten des Simulators war es später jedoch auch notwendig, den Evaluator zur Ausführung der Individuen als C-Programm anzubinden. Auf diese Weise ist der Plugin-Entwicklung innerhalb dieser Arbeit eine wesentliche Bedeutung zugekommen. Da die Entwicklung von Zusatzkomponenten nur schwach im BREVE-Leitfaden beschrieben ist und innerhalb dieser Arbeit immer wieder massive Probleme bereitet hat, wird auf sie im Folgenden detailliert eingegangen.

Plugins werden mit Hilfe einer *C/C++*-Bibliothek entwickelt, die von BREVE zur Verfügung gestellt wird. Sie enthält Schnittstellen und Datentypen zur Erstellung von Proxyfunktionen, welche eine Transformation zwischen der Skriptsprache und dem nativen C-Programm ermöglichen. Die Plugins selbst werden unter Windows als dynamische Bibliotheken (dll) kompiliert und innerhalb einer steve-Klasse über das Schlüsselwort `@plugin` geladen. Zusätzlich muss eine Eintrittsfunktion angegeben werden, welche das Plugin initialisiert und weitere Funktionen im Simulationskern anmeldet. Dies soll anhand eines Beispiels demonstriert werden. Innerhalb der Controller-Klasse wird ein Plugin namens `MathPlugin` eingebunden, welches BREVE um die fehlende Funktionalität zum Runden einer Fließkommazahl erweitern soll. Hierzu wird eine C-Programm formuliert, welches zunächst die Eintrittsfunktion `loadMathFunction` beschreibt:

```
/** MathPlugin.c */

__declspec(dllexport) void loadMathFunction(void* data)
{
   brNewBreveCall(data, "roundValue", roundValueProxy,
     AT_INT, AT_DOUBLE, 0);
}
```

Der Funktionsparameter `data` ist undokumentiert und von der Bedeutung her unklar. Auf Anfrage beschrieb der Autor Jon Klein den Parameter als ein *Namespace-Objekt*, welches nur bei der Registrierung der Funktionen weitergereicht werden müsse, ansonsten jedoch ohne Nutzen für den Plugin-Entwickler sei. Über den Aufruf der Bibliotheksfunktion `newBreveCall` wird die Verbindung zwischen einem steve-Aufruf und einer C-Funktion hergestellt. Der erste Parameter ist das angesprochene Namespace-Objekt und wird unverändert weitergereicht. An zweiter Stelle wird festgelegt, unter welchem Methodennamen die zusätzliche Funktionalität in steve registriert wird, in diesem Fall `roundValue`. Darauf folgt in der Parameterliste der Zeiger auf die C-Funktion, welche den Algorithmus zum Runden der Zahl enthält (`roundValueProxy`). Zusätzlich müssen der Rückgabetyp der Zielfunktion im vierten Parameter, wie auch nachfolgend deren Argumente deklariert werden. BREVE benutzt hierfür Makros, welche die erlaubten Datentypen beschreiben. In dem gezeigten Fall wird ein ganzzahliger Rückgabewert als `AT_INT` und ein Fließkommaparameter über `AT_DOUBLE` angekündigt. Das letzte Argument ist immer eine 0, sie terminiert die Deklaration der Parameter für die Zielfunktion. Als nächstes muss die Zielfunktion `roundValueProxy` implementiert werden. Sie benötigt ebenfalls eine definierte Signatur, um von der Skriptsprache direkt aufgerufen werden zu können. Dabei werden die Parameter `args` als Array des Typus `brEval` übergeben, welcher von der Simulation zur einheitlichen Speicherung aller primitiven Datentypen gebraucht wird. Der zweite Parameter dient zur Rückgabe eines Berechnungsergebnisses und ist daher ebenfalls ein `brEval`-Objekt. Zuletzt beinhaltet der Aufruf einen Zeiger auf den Kernel der Simulation, dieser findet jedoch nur im Zusammenhang mit der callback-Schnittstelle Verwendung, auf die später eingegangen wird.

```
int roundValueProxy(brEval args[], brEval* result, void* instance)
{
   int roundedValue = (int) round( BRDOUBLE(&args[0]) );
   result->set(roundedValue);

   return EC_OK;
}
```

Wie in der ersten Zeile des Funktionkörpers zu sehen ist, wird das Makro `BRDOUBLE` angewendet, um die zu rundende Zahl als Fließkommawert aus der Parameterliste `args` zu extrahieren. Für das Setzen des Ergebnisses in der nächsten Zeile bietet das Objekt `result` hingegen setter-Methoden, die eine korrekte Speicherung des Wertes übernehmen. Jede Plugin-Funktion muss mit der Rückgabe des Wertes `EC_OK` abschließen und so die erfolgreiche Bearbeitung signalisieren. Obwohl alternativ bei Fehlersituationen der Wert `EC_ERROR` benutzt werden kann, ist davon abzuraten, da die Simulation dadurch gestoppt wird. Sinnvoller wäre es gewesen, eine strukturierte Anzahl von Fehlercodes zur Verfügung zu stellen, um der Simulation detailliert die Art der Ausnahme mitteilen zu können.

Generell gestaltet es sich schwierig einen Fehler, der im Plugin ausgelöst wird, im Simulationsprogramm zu erkennen. Steve selbst bietet hierfür keine Mechanismen wie Exception Handling. Ungültige Anweisungen, die nicht automatisch zum Programmabsturz führen, werden über den numerischen Wert 0 angedeutet. Das Debuggen des Plugins ist dabei ebenfalls ausgeschlossen. Obwohl BREVE quelloffen angeboten wird, noch keinem Entwickler der Gemeinschaft gelungen, das Projekt anhand der Konfigurationsskripte zu kompilieren. In den unübersichtlichen makefiles werden Teile der Headerdateien abhängig von den genutzten Bibliotheken erst während der Ausführung kreiert, so dass ein Verzicht auf die Skripte nicht möglich ist.
Zudem wird die Entwicklung unter Windows durch die Tatsache erschwert, dass die Plugin-Bibliothek nur im Unix-Format beigefügt wird. Da BREVE für diese Arbeit unter Windows genutzt wurde, war es somit notwendig, die Plugins plattformübergreifend mit Hilfe von *Cygwin*[9] und dem *gcc-Compiler*[10] zu erstellen. Selbst mehrfache Anfragen beim Autor Jon Klein, die Bibliothek für Windows zu kompilieren und eine detaillierte Anleitung der makefiles bereitzustellen, blieben unbeantwortet.

Ein besonderes Problem bei der Entwicklung von Plugins ergibt sich durch den Umstand, dass der Simulationskernel und die Anwenderoberfläche von BREVE in dem selben Thread laufen. Die Applikation wird nicht beeinflusst, solange ein Plugin nur kurze Berechnungen durchführt. Bei blockierenden Operationen hingegen, wie langen Schleifen, Lesevorgängen aus dem Netzwerk oder `sleep`-Anweisungen, friert der Rest des Programmes ein. Für die Implementation des ServerPlugins wäre dies zunächst kein Hindernis gewesen, da der Vorgang der Individuenbewertung rein seriell abläuft: Der Controller hätte zum Empfang eines Individuums einen blockierenden Lesevorgang ausgelöst, wodurch die Simulation und die Anwenderoberfläche nicht weitergearbeitet hätten. Nach Vollendung der Netzwerkkommunikation wäre der Thread wieder freigegeben worden und die Abarbeitung des Steuerprogrammes durch die Simulation hätte beginnen können. Da BREVE nach einer Blockierung allerdings dazu neigt, abzustürzen, müssen zeitintensive Pluginfunktionen in einen eigenen Thread ausgelagert werden.
Auf diese Weise wird eine zusätzliche Problematik aufgeworfen, die Sicherung der Synchronität zwischen dem Simulator und den Plugins. Durch die Nutzung von multiplen Threads ist keine direkte Kommunikation zwischen den Komponenten mehr möglich. Anstelle von einfachen Funktionsaufrufen mit Rückgabewert müssen globale Speicher eingeführt werden, die über Mutexe, kritische Sektionen oder Ereignissysteme geschützt werden. Die Synchronisationmethode für die Plugins dieser Arbeit mit Hilfe von *Win32-Events* wird in den Kapiteln 5.4.4 und 5.4.5 näher erläutert.
Häufig ist es notwendig, dass ein Plugin direkt eine Methode des steve-Skriptes aufruft, um beispielsweise eine Statusmeldung auszugeben oder im Fall der ELFE-Simulation direkten Einfluss auf das Robotermodell zu nehmen. Bis zu den ersten Versionen von BREVE 2.4 enthielt die Plugin-Bibliothek nur eine Funktion zur Anzeige einer Nachricht auf der Debug-Konsole der Anwenderoberfläche. Auf besonderen Wunsch schaltete der

[9] Linux-Emulation für Windows. Internetseite des Projekts: http://www.cygwin.com
[10] Freie Compiler-Kollektion unter GNU Lizenz. Internetseite des Projekts: http://gcc.gnu.org

Autor von BREVE zusätzliche Funktionen der sog. *callback-API* frei, welche Zugriff auf alle Methoden einer steve-Klasse über deren Namen bietet. Da diese Funktionen bislang nicht dokumentiert sind, wird ein Beispiel gegeben. Die folgenden Zeilen bereiten den Aufruf einer Methode `set-value` vor, die in der steve-Klasse implementiert sein muss, welche das Plugin eingebunden hat:

```
1. brEval result;
2. brEval paramWrapper;
3. paramWrapper.set(10);
4. brEval* params[1];
5. params[0] = &paramWrapper;
6. brMethodCallByNameWithArgs(breveInstance, "set-value", params,
7.                            1, &result);
8. int returnValue = BRINT(&result);
```

In den ersten beiden Zeilen werden zwei Variablen vom internen Datentyp `brEval` deklariert, um einen Rückgabewert des Aufrufes auffangen (`result`) und einen Parameter übergeben zu können (`paramWrapper`). Nachdem `paramWrapper` mit einem Wert initialisiert wurde (Zeile 3), muss ein `brEval`-Array erzeugt werden, welches die Adresse des Parameters aufnimmt (Zeile 5). Über die sechste und siebte Zeile erstreckt sich der eigentliche Methodenaufruf. Er benötigt einen Zeiger auf den Simulationskernel (`breveInstance`), den Namen der Methode als String, das Parameterarray und dessen Länge und zuletzt die Adresse von `result` zur Speicherung des Rückgabewertes. Es ist notwendig, immer eine gültige `brEval`-Variable anzugeben, selbst, wenn kein Rückgabewert gesetzt wird, ansonsten stürzt die Applikation ab.
Das Emulator Plugin, welches die Abarbeitung der Individuen übernimmt, nutzt die callback-API, um die Seiteneffekte des GP-Individuums auf die virtuelle ELFE zu übertragen. Da das Plugin in einem eigenen Thread ausgeführt wird und die Sprache steve keine Sicherung von parallel genutzten Programmabschnitten bietet, ergibt sich erneut ein Synchronisationsproblem mit dem Simulator. Auf Anfrage bei Jon Klein kam dieser der Bitte nach, einen synchronisierten Aufruf der Methoden zu ermöglichen und erweiterte die Plugin-API. Ein gesicherter callback solle demnach wie folgt aussehen:

```
brEngineLock(breveInstance->engine);
brMethodCallByNameWithArgs(breveInstance, "set-value", params,
                           1, &result);
brEngineUnlock(breveInstance->engine);
```

Der bereits besprochene Aufruf der Methode `set-value` wird umschlossen durch das Paar `brEngineLock` und `brEngineUnlock`, die jeweils das nicht weiter bestimmte `engine`-Element aus der Kernelstruktur des Simulators benötigen. Aus Gründen, die weder dem

Autor noch Jon Klein bekannt sind, scheint die Applikation bei dem Synchronisationsversuch jedoch in einem Deadlock festzuhängen.

5.4.4. Der Kommunikationsserver

Der Simulator benötigte zur Kommunikation mit dem GP-Kernel eine TCP/IP-Schnittstelle, um Individuen empfangen und deren Fitnesswert zurücksenden zu können. Zu diesem Zweck wurden mehrere Implementationsmöglichkeiten untersucht und gegeneinander abgewogen.
BREVE selbst bietet eine Möglichkeit, um mit externen Systemen über eine Netzwerkverbindung kommunizieren zu können. Über die Klasse `Network` kann der Simulator einen Webserver starten, der entweder HTML-Seiten auf Anfrage versendet oder einfache Methodenaufrufe in Form einer URL-Kodierung akzeptiert. Die folgenden Beispiele sind der Klassendokumentation [Kle] entnommen und demonstrieren den Aufruf der steve-Methoden `turn-agent-blue` und `set-agent-color`:

```
1) http://myserver:33333/turn-agent-blue
2) http://myserver:33333/set-agent-color_.2_.4_.6
```

Beide Anfragen richten sich an den von BREVE aufgesetzten Server `myserver`, der auf Port `3333` lauscht. Soll wie in 1) eine parameterlose Methode ausgeführt werden, reicht es, den Methodennamen nach einem Schrägstrich an die URL anzuhängen. Zusätzliche Parameter, wie die drei Fließkommawerte `0.2`, `0.4` und `0.6` in 2) werden durch Unterstrich getrennt voneinander dargestellt. Jon Klein beabsichtigt mit dieser Option, dass jeder Simulationsentwickler dem Anwender eine HTML-Seite mit vorgefertigten Steuerlinks zur Verfügung stellen kann, die aus dem Webbrowser bedienbar sind.
Für die Übertragung der Individuen zum Simulator eignete sich diese Art der Übertragung jedoch nicht. Zum einen ist es nur möglich, Integer oder Double Datentypen zu übertragen, nicht aber Strings. Zwar hätte ein Individuum in eine Zahlenkodierung überführt werden können, da aber die Anzahl der Methoden-Parameter statisch ist, wäre es notwendig gewesen, die genaue Länge eines Individuums im Voraus zu kennen. Zum anderen befand sich die Network-Klasse zum Stand dieser Arbeit in einem sehr frühen Entwicklungsstadium und wies zahlreiche bekannte Fehler, wie Speicherverletzungen und Aufrufverzögerungen auf.
Um dennoch eine unbeschränkte Übertragungsschnittstelle zu realisieren, wurde ein Plugin namens *ServerPlugin* in prozeduralem *C* unter Verwendung der *Winsock*-Bibliothek geschrieben. Wahlweise über eine synchrone oder asynchrone TCP/IP-Verbindung kann der Server genau eine Verbindung aufnehmen, Individuen in Form von Strings annehmen und an den Kern des Simulators weiterleiten. Sobald ein Individuum von der Simulation ausgewertet worden ist, wird dessen Fitness ebenfalls im Stringformat an den GP-

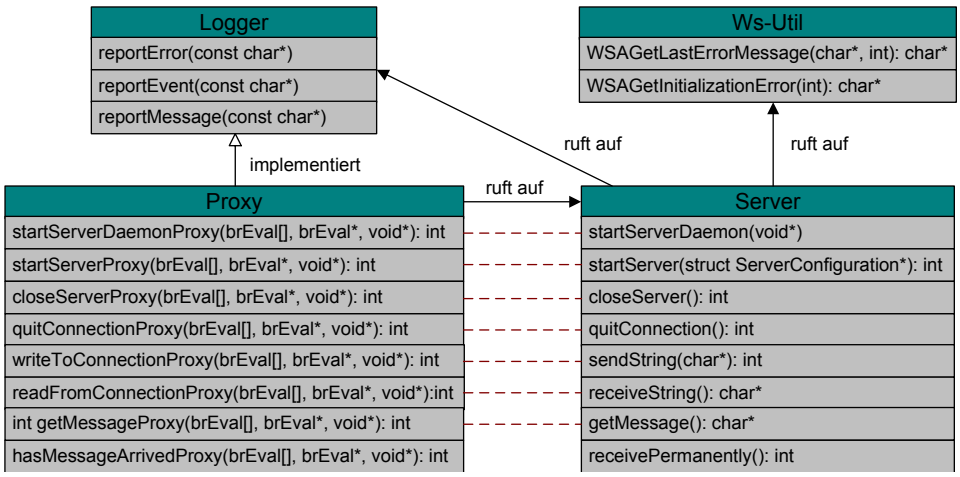

Abbildung 5.20.: Aufbau der Komponenten des ServerPlugins

Kernel zurückgeschickt. Die Abb. 5.20 zeigt den schematischen Aufbau der ServerPlugin-Komponenten: Der `Proxy` ist die Pluginschnittstelle und registriert alle Serverfunktionen im Simulator. Dabei werden Aufrufe nicht direkt bearbeitet, sondern an die Server-Komponente weitergeleitet, welche für die eigentliche Kommunikation zuständig ist.
Die Trennung der Komponenten ist wichtig, um die Entwicklung der Pluginfunktionalität von den Schnittstellendefinitionen der BREVE-Bibliothek zu entkoppeln. Der Server kann auf diese Weise extern als eigenständige Applikation getestet oder bei Bedarf ausgetauscht werden. Dieses Prinzip wird durch den Header `Logger` unterstützt. Da es häufig notwendig ist, eine Statusmeldung auszugeben, wurden die drei Funktionen `reportMessage`, `reportError` und `reportEvent` definiert. Obwohl die Server-Komponente den Header einbindet, werden die Funktionen im Proxy implementiert, da nur dort Zugriff auf die Pluginbibliothek und somit auf den Kernel des Simulators möglich ist. Es war angedacht, die drei Reportfunktionen in unterschiedliche Quellen schreiben zu lassen, so hätten beispielsweise informative Nachrichten auf der Debugkonsole, Fehler hingegen in einer Datei ausgegeben werden können. In der momentanen Implementierung werden die Meldungen jedoch nur mit verschiedenen Präfixen auf die Debugkonsole umgeleitet. Die letzte Komponente, `Ws-Util`, wird bei Bedarf vom Server aufgerufen und übernimmt die Transformation von Fehlercodes der Winsock-API in aussagekräftige Statusmeldungen für den Endbenutzer.

Ursprünglich war es geplant, den Server auf serielle Weise in die Simulation einzubinden, wie es in Abb. 5.21a) zu betrachten ist. Bei diesem Szenario wird zunächst der Server anhand eines Ports und einer IP-Adresse über die Funktion `startServer` gestartet. Das Plugin wartet daraufhin auf die Verbindung des GP-Kernels als Klienten, wodurch die Simulation, die im selben Thread läuft, blockiert. Nachdem eine Verbindung aufgebaut wurde, löst die Simulation durch `readFromConnection` einen Lesevorgang aus und blockiert ein weiteres Mal, bis als Ergebnis ein Individuum vom Server zurückgeliefert wird. Nach der Ausführung des Individuums wird der Fitnesswert über `writeToConnection`

in eine Zeichenkette konvertiert und über das Plugin an den GP-Kernel geschickt.
Wie in Abschnitt 5.4.3 bereits beschrieben wurde, versetzen blockierende Aufrufe den Simulator jedoch in einen instabilen Zustand. Aus diesem Grund wurde das ServerPlugin erweitert, um bei Bedarf in einem eigenen Thread als Daemon gestartet werden zu können. Auf diese Weise kann für zukünftige Projekte oder überarbeitete Versionen von BREVE zwischen den beiden Betriebsmodi gewählt werden.
Die Nutzung des Server-Daemons unterscheidet sich in einem wesentlichen Punkt von dem synchronen Modus: Die Simulation kann eingehende Nachrichten nicht über den Rückgabewert einer Leseoperation erhalten, sondern muss über gesichertes Polling den Status des Servers überprüfen, wie Abb. 5.21b) zeigt. Hierfür wird der Server über die Funktion `startServerDaemon` parallel zur Simulation gestartet. Sobald eine Verbindung vom GP-Kernel aufgebaut wurde, wechselt der Server automatisch über die Funktion `receivePermanently` in eine Leseschleife. Eingehende Individuen werden zwischengespeichert, woraufhin eine interne Zustandsvariable aktiviert wird. Dabei handelt es sich um ein Win32 Kernel Event, welches implizit synchronisiert ist und andeutet, dass die Simulation über `getMessage` ein Individuum abholen kann. Hierfür muss das Simulationsprogramm jedoch ständig den Status des Servers durch die Funktion `hasMessageArrived` abfragen, was in der momentanen Implementierung zweimal pro Sekunde geschieht. Nach Auswertung des Individuums wird in gleicher Weise wie beim synchronen Server der Fitnesswert über `writeToConnection` direkt geschrieben. Da Sockets der Winsock-API synchronisiert sind und schreibende Operationen nicht blockieren, kann dieser Aufruf direkt aus dem Simulatorthread erfolgen. Die Leseschleife des Servers wird erst verlassen, wenn entweder `quitConnection` zum Schliessen der aktuellen Verbindung oder `closeServer` zum vollständigen Herunterfahren aufgerufen wird.

Das ServerPlugin ist als Prototyp anzusehen, da in vielen Fällen Fehler produziert werden können, die einen Evolutionslauf gefährden: Wird der Server asynchron als Daemon gestartet, kann keine überprüfbare Rückmeldung an den Simulator gegeben werden, ob das Erstellen des TCP/IP-Socket erfolgreich gewesen ist. Zwar wird in diesem Fall, wie bei anderen Netzwerkfehlern auch, eine Meldung auf der BREVE-Konsole ausgegeben, eine Auswirkung auf den Programmablauf hat dies jedoch nicht. Generell wäre es möglich, über die Callback-Schnittstelle direkt in die Simulation einzugreifen, da jedoch keine Möglichkeiten zur Synchronisation bestehen, ist in diesem Fall von dieser Option abzuraten. Aufgrund des einfachen Übertragungsprotokolls können auch Fehler in der Übertragungsreihenfolge nicht erkannt werden. Sendet der GP-Kernel beispielsweise zwei Individuen hintereinander, ohne vorher als Antwort einen Fitnesswert entgegengenommen zu haben, wird die zweite Übertragung vom ServerPlugin verworfen. Der Kernel wird darüber nicht informiert, da kein Handshake zwischen den Kommunikationspartnern stattfindet. Dies ist jedoch ein Fehler, der aus falscher Anwendung resultiert. Bei Einhaltung des Protokolls hat sich der Server als zuverlässig erwiesen.

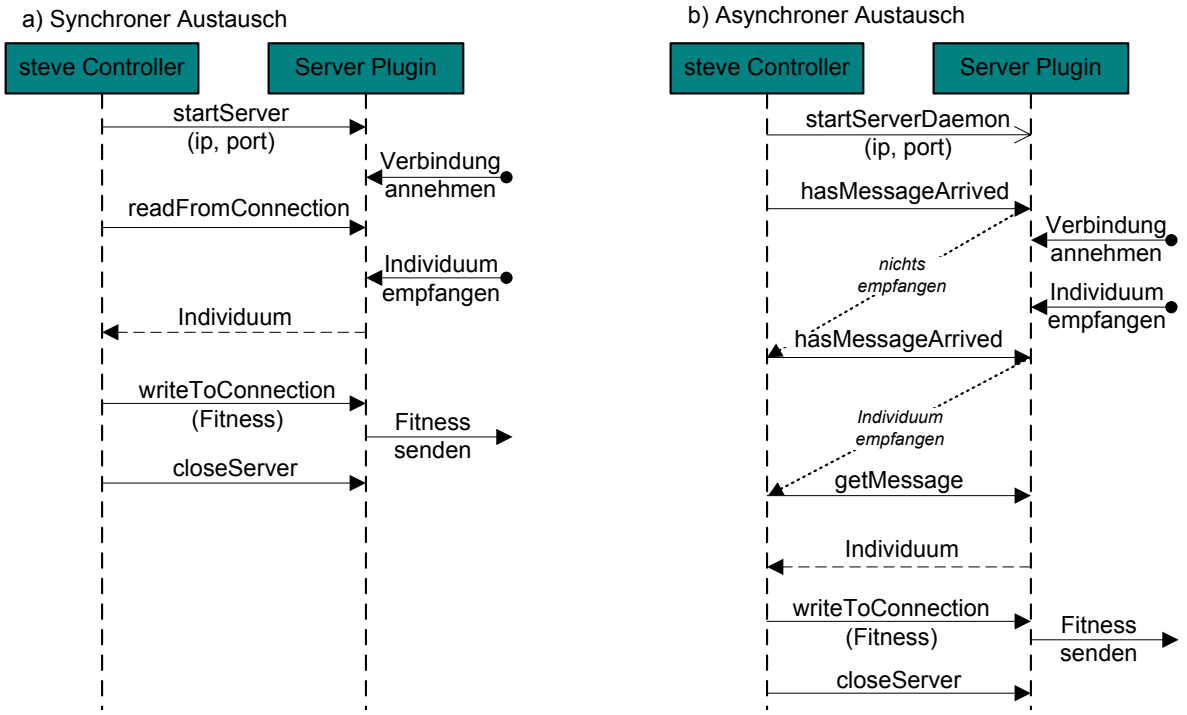

Abbildung 5.21.: Synchroner und asynchroner Nachrichtenaustausch zwischen Simulator und ServerPlugin

5.4.5. Der Evaluator

Nachdem die Individuen über das ServerPlugin empfangen worden sind, müssen sie als Steuerprogramm für das ELFE-Modell interpretiert und ausgeführt werden. Ursprünglich sollte dieser Evaluationsalgorithmus in der Skriptsprache steve verwirklicht werden. Aufgrund von Implementationsschwierigkeiten wurde er jedoch in einem eigenen Plugin, dem *AksenPlugin*, verwirklicht. Im Folgenden werden beide Ansätze näher beschrieben und deren Vorteile und Probleme gegeneinander abgewägt.

Rekursive Umsetzung in steve

Die einfachste Möglichkeit scheint zunächst, die Abarbeitung der Individuen rekursiv über das Simulationsskript zu realisieren. Der Implementationsaufwand ist gering, da in steve eine Baumstruktur als verkettete Liste von Objekten einfach nachgebildet werden kann. Auch kann der Algorithmus zur Abarbeitung des Individuums aus Abschnitt 5.3.2 mit nur wenigen Veränderungen übernommen werden: Einzig die Funktionszeiger, zu denen in steve kein Äquivalent existiert, müssen durch eine Kodierung ersetzt werden, welche den Knoten der Evaluationsfunktion zuordnet.

Letztendlich scheitert es jedoch an dem Seiteneffekt des `Sleep`-Knotens, der die Abarbeitung des Individuums pausieren soll, ohne gleichzeitig die Simulation zu stoppen. Die steve-Syntax bietet zwar als Äquivalent selbst einen `sleep`-Befehl in der Klasse `Control`. Dieser blockiert jedoch die gesamte Applikation: Eventuell aktivierte Motoren laufen über diese Zeit nicht weiter, ebenso wenig wird die interne Simulationszeit weitergemessen. Darüber hinaus reagiert die Anwenderoberfläche nicht mehr und die Applikation wird instabil.

Iterative Umsetzung in steve

Das Problem des Pausierens kann umgangen werden, indem das Individuum iterativ abgearbeitet wird, wie es in Abschnitt 5.3.3 vorgestellt wurde. Im Gegensatz zur rekursiven Methode kann hierbei die Ausführung des Steuerprogramms jederzeit unterbrochen und wieder aufgenommen werden. Dies ermöglicht es, den Methoden-Scheduler von BREVE zum Pausieren des Evaluationslaufes zu nutzen, während die Simulation ungestört weiterläuft.
Der schematische Ablauf ist wie folgt: Der `Controller` übernimmt in diesem Beispiel die Flusssteuerung der Abarbeitung, der Algorithmus zur iterativen Berechnung der Knoten eines Individuums ist hingegen in der Klasse `StackMachine` realisiert. Nachdem der Evaluator das Individuum erhalten hat, führt der Controller wiederholt dessen Methode `evalNode` aus, um nacheinander jeden Knoten einzeln zu berechnen. Handelt es sich dabei um einen `sleep`-Knoten, erhält der Controller als Rückgabewert die Länge der zu pausierenden Zeit. Über die Methode `schedule` ist der Controller dann in der Lage, die nächste Ausführung von `evalNext` um diesen Wert zu verzögern. Da BREVE den zeitaktivierten Methodenaufruf übernimmt, läuft die gesamte Simulation über diese Zeit weiter.
Bei dieser Art der Abarbeitung zeigt sich jedoch wieder das Problem des Verzweigungsknotens, welches nur über erhöhten Implementierungsaufwand zu lösen ist.

Rekursive Umsetzung als Plugin

Für diese Arbeit wurde die Auswertung der Individuen daher nicht in der steve-Sprache, sondern als eigenes C-Programm in Form eines Plugins verwirklicht. Das *AksenEmulator* genannte Plugin arbeitet Individuen rekursiv in einem eigenen Thread ab. Auf diese Weise ist der Implementationsaufwand geringer, als bei der Iterativen Methode und zudem kann die Ausführung des Steuerprogramms über die `sleep`-Funktion des C-Compilers unterbrochen werden, ohne die Simulation zu blockieren.
Seiteneffekte anderer Knoten, wie die Abfrage eines IR-Sensors oder das Stellen eines Servos werden über die callback-Schnittstelle der Plugin-API an den Simulator weitergelei-

tet. Der asynchrone Zugriff des Plugins auf die Simulation erfordert zwar zusätzliche Synchronisationsmaßnahmen, allerdings konnten die Techniken, welche bereits beim Kommunikationsserver erfolgreich eingesetzt wurden, wieder verwendet werden.

Durch die asynchrone Architektur verhält sich das Plugin ähnlich dem AKSEN-Board, bei dem Aktorik und Programmausführung vollkommen parallel ablaufen. Um diesen Aspekt weiter zu unterstreichen wurde daher ein besonderes Konzept verfolgt: Das Programm zur Ausführung der Individuen sollte nur einmal entwickelt und mit möglichst wenig Portierungsaufwand für beide Systeme übernommen werden können. Dies bedeutete einen drastischen Rückgang der Entwicklungszeit, da Änderungen am Evaluationssystem nur einmalig durchgeführt werden müssen.
Das Konzept basiert auf dem Gedanken, ausgewählte Teile der Bibliothek des AKSEN-Boards für die Pluginentwicklung zu emulieren. Um zu gewährleisten, dass das Programm auf beiden Systemen lauffähig sein wird, muss somit Code erzeugt werden, der sich nur den Möglichkeiten bedient, die vom Mikrocontroller unterstützt werden. Dies bedeutet nicht nur die Beschränkung auf Funktionsaufrufe der AKSEN-API sondern auch der Verzicht auf eine dynamische Speicherverwaltung und weite Teile der C-Standardbibliothek.

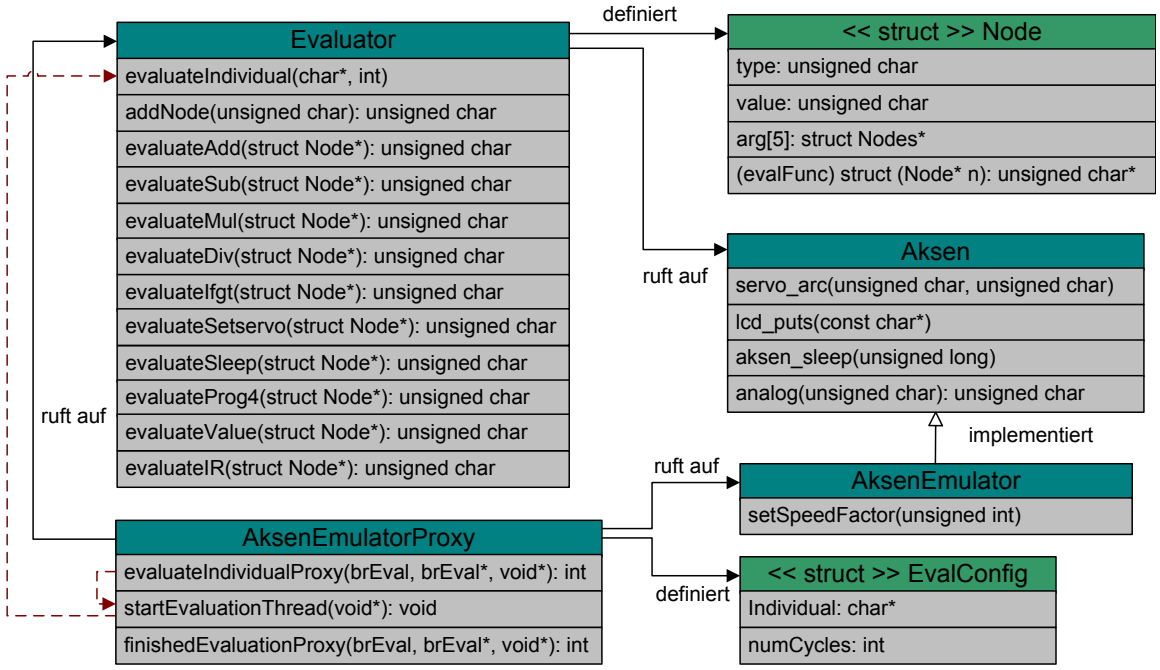

Abbildung 5.22.: Aufbau der Komponenten des AksenEmulator Plugins

Die Abb. 5.22 zeigt den Aufbau der Komponenten des AksenEmulators. Der Kern des Plugins ist der `Evaluator`, er ist für den Aufbau sowie die Ausführung des Syntaxbaumes verantwortlich. Hierfür definiert er in genauer Entsprechung zu Abschnitt 5.3.1

eine Struktur `Node`, die den Rahmen für jeden ausführbaren Knoten bildet. Anhand der `evaluate`-Funktionen des Evaluators ist zu erkennen, auf welche Weise die Baumstruktur als aktives Programm interpretiert wird. Jeder Knoten besitzt einen Zeiger `evalFunc` auf eine entsprechende Funktion des Evaluators, durch die er ausgewertet wird.
Die Seiteneffekte werden durch den Evaluator über die `AKSEN API` ausgelöst. Dabei handelt es sich um die Header der Funktionen, die vom Mikrocontroller-Board unterstützt und im Falle des Plugins durch den `AKSEN Emulator` nachgebildet werden. Zu sehen ist nur die Untermenge der Funktionen, die für das ELFE-Problem benötigt werden, wie z. B. Zugriffe auf die Infrarot-Ports, Steuerung der Motoren, Ausgabe auf dem Display und der Sleep-Befehl.
Auf diese Weise werden z. B. Nachrichten, die im realen AKSEN-System auf der LCD-Anzeige der Boards ausgegeben würden, auf die Debug-Konsole des Simulators umgeleitet. Während zu diesem Zweck in der callback-API von BREVE eine eigene Funktion existiert (`slMessage`(int, char*)), müssen Aufrufe, die Einfluss auf Aktorik und Sensorik des ELFE-Modells nehmen, explizit im Controller-Skript der Simulation definiert werden.
Der `AksenEmulatorProxy` stellt die Schnittstelle von steve-Skript und Plugin dar. Der Simulation wird die Funktion `evaluateIndividual` (in der Abbildung nur als Proxy zu sehen) exportiert und bei Aufruf über `startEvaluationThread` in einem eigenen Thread ausgeführt. Die Verkettung dieses Aufrufs wird durch die roten Pfeile angedeutet. Da der Thread-Funktion zwei Parameter übergeben werden müssen, die S-Expression und die Anzahl, wie häufig das Individuum hintereinander ausgeführt werden soll, wurde die Struktur EvalConfig definiert. Ob die Ausführung des Steuerungsprogrammes abgeschlossen wurde, kann innerhalb von steve über `finishedEvaluation` abfragt werden. Ähnlich wie beim ServerPlugin wird hierbei der aktuelle Status der Ausführung über ein Win32-Event markiert.
Zusätzlich wird durch den AksenEmulatorProxy die Funktion `setSpeedFactor` zur Verfügung gestellt, die Zeit zu skalieren, mit der ein Sleep-Knoten die Abarbeitung eines Individuums pausiert. Dies ist notwendig, um die Ausführungszeit der Individuen auf dem Simulator und dem Realsystem annähernd proportional und somit vergleichbar zu halten.

Ein beispielhafter Ablauf zur Ausführung eines Individuums ist in Abb. 5.23 zu sehen. Zunächst wird innerhalb vom steve-Controllers das Individuum als S-Expression und die Anzahl der gewünschten Iterationen über die exportierte Funktion `evaluateIndividual` an den Proxy des Plugins, dem `EmulatorProxy`, übergeben. Die Ausführung des Individuums wird vom Proxy in einen neuen Thread ausgelagert und über `evaluate` gestartet. Im Beispiel ist zu sehen, wie der Controller den Status des Evaluators überprüft, während dieser die Baumstruktur aus der S-Expression rekursiv aufbaut. Die Funktion `finishedEvaluation` liefert den Wert 0 zurück, da der Thread noch aktiv und das zugehörige Win32-Event deaktiviert ist. Sobald das Individuum in eine Baumstruktur transformiert worden ist, wird jeder Knoten rekursiv über seinen gespeicherten Funktionszeiger `evalFunc` ausgewertet. Das Beispiel zeigt hierbei zwei Seiteneffekte, die sich als Aufrufe von emulierten AKSEN-Funktionen äußern. Über `servo_arc` wird ein Motor

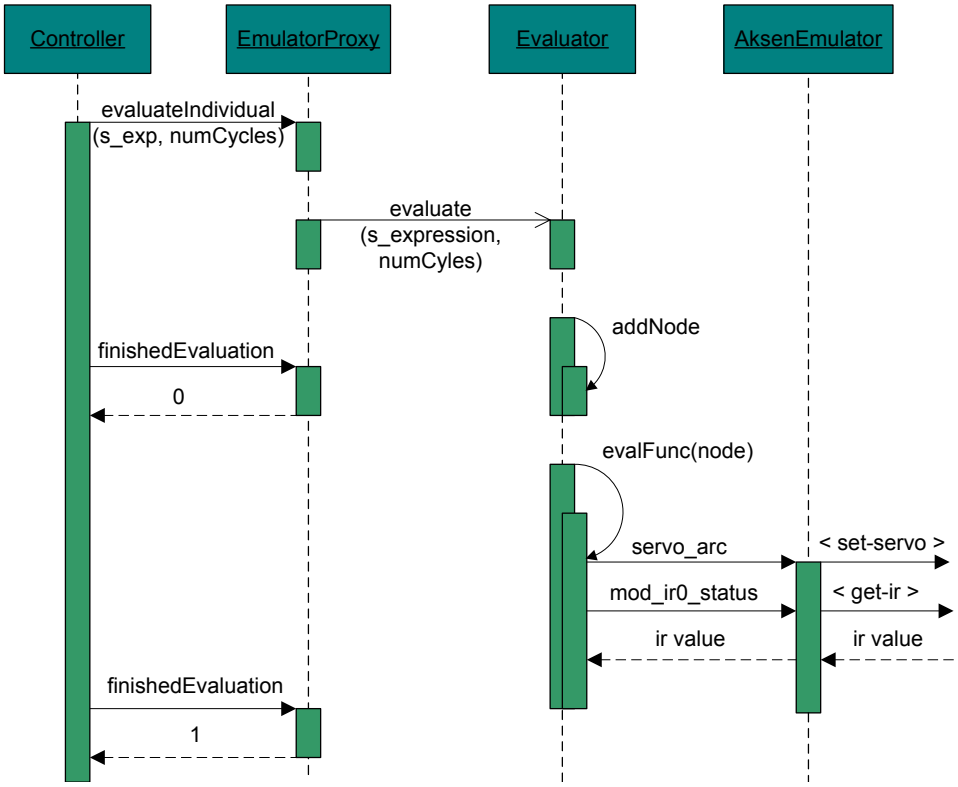

Abbildung 5.23.: Beispiel einer Individuenauswertung über das AksenEmulatorPlugin

auf einen gegebenen Winkel gedreht, während `get-ir` den Wert eines Infrarotsensors des ELFE-Modells anfordert. Der AksenEmulator nutzt intern jedoch die callback-API von BREVE, welche festgelegte Methoden der Controller-Klasse des steve-Skriptes aufruft. In der Abbildung wird dies durch die ausgehenden Pfeile mit umklammerter Beschriftung dargestellt. Den Abschluss der Evaluierung erfährt der Controller, wenn die Funktion `finishedEvaluation` ein positives Ergebnis zurückgibt. Obwohl in der Abbildung nur zwei dieser Aufrufe zu sehen sind, werden sie in der Praxis in einer Frequenz von 2 Hz wiederholt.

5.4.6. Der Controller

Der Ausgangspunkt jeder BREVE-Simulation ist der Controller, welcher von der Klasse `Control`, bzw. bei physikalischen Simulationen von `PhysicalControl` abgeleitet wird. Für gewöhnlich erledigt der Controller maximal die folgenden Aufgaben:

- Laden von Plugins
- Initialisierung der physikalischen Parameter

- Instanzierung der Agenten
- einfache Ablaufsteuerung

Im Falle der ELFE-Simulation war die Ablaufsteuerung jedoch nur über erheblichen Zusatzaufwand zu realisieren. Dies resultiert aus der asynchronen Kommunikation des Skriptes mit den Plugins und der Abstinenz eines blockierungsfreien Pausierungsbefehls in steve.
Um ein Individuum vollständig auszuwerten, müssen mehrere Schritte sequentiell abgearbeitet werden: Zunächst wird der Server gestartet, um Transmissionen vom GP-Kernel entgegennehmen zu können. Sobald ein Individuum angekommen ist, muss die ELFE in eine vorher definierte Grundposition gebracht werden, um Koevolutionseffekte zu vermeiden. Danach kann die Ausführung des Steuerprogrammes über den AksenEmulator beginnen. Zuletzt berechnet der Controller die Fitness des Laufs und sendet sie über den Server zurück an den GP-Kernel.

Das Problem bei diesem Ablauf ist, dass der Controller zu den beiden Plugins wie auch zum ELFE-Modell parallel arbeitet. Bevor die jeweils nächste Sequenz ausgeführt werden darf, muss sichergestellt werden, dass deren vorhergehende abgeschlossen wurde. Wie in den Abschnitten über die Plugins bereits dargelegt worden ist, wurde die Synchronisation über Polling einer Zustandsvariablen realisiert: Der Controller bleibt so lange in einem Wartezustand, bis ein neues Ereignis signalisiert wird.
Dieses Verfahren musste auch für das Zurücksetzen der ELFE verwendet werden. Da es nicht möglich ist, die Beine ohne Verzögerung in die gewünschte Stellung zu bringen, müssen sie über normale Steuerungsbefehle bewegt werden. Dabei ist der Abschluss der Bewegung gegeben, wenn der Evaluator nicht mehr arbeitet und sich keiner der Servomotoren mehr dreht.
Insbesondere bei der Arbeit mit BREVE ist es wichtig, das Polling künstlich auszubremsen, um nicht eine zu hohe Rechenlast zu erzeugen und die Stabilität der Applikation zu gefährden. Der implementierte `sleep`-Befehl ist, wie bereits in Abschnitt 5.4.5 erklärt, für diese Aufgabe nicht zu gebrauchen. Daher wurde auf den Methoden-Scheduler zurückgegriffen: Jede einzelne Sequenz wird in eine eigene Methode ausgelagert, die im Falle eines Wartezustands sich selbst mit einer Verzögerung von 0.5 sek aufruft oder beim Wechsel des Zustands die nächste Sequenz aktiviert. Eine weitere Aufgabe des Controllers ist die zeitliche Synchronisation mit dem AksenEmulator. Während der gesamten Simulation läuft in der `iterate`-Methode des Controller eine Zeitmessung zur Bestimmung der Simulationsgeschwindigkeit. Das Ergebnis wird dem AksenEmulator über die `setSpeedFactor`-Methode mitgeteilt, damit die Länge der Pausen angepasst werden kann, welche durch die sleep-Knoten ausgelöst werden.

5.4.7. Bewertung der Lösung

Es konnte gezeigt werden, dass es möglich ist, ein Modell der ELFE in BREVE zu schaffen und an ein externes Evolutionssystem anzubinden. Wie beschrieben worden ist, war dies jedoch mit vielen Schwierigkeiten verbunden. Obwohl BREVE großes Potential durch die einfach zu erlernende Skriptsprache steve und die Anbindungsmöglichkeit von Plugins zeigt, verhindern noch viele Fehler und Schwächen des Systems eine schnelle und problemlose Entwicklung. Insbesondere die daraus resultierende Ungenauigkeit der Servomotoren bedeutet einen massiven Nachteil für die Experimente: Sie können maximal in doppelter Realgeschwidigkeit ablaufen, um nicht zu stark vom Verhalten des realen Roboters abzuweichen.
Die beiden Zusatzkomponenten, das ServerPlugin und der AksenEmulator, funktionieren zuverlässig und erfüllen den angedachten Zweck. Der Server ist problemunabhängig entworfen worden und ist für jedes Projekt zu verwenden, welches eine rein zeichenkettenbasierte Kommunikation benötigt. Falls in einer folgenden Version von BREVE das Problem mit der Anwenderoberfläche gelöst werden sollte, ist zudem der synchrone Modus des Servers interessant. Der Emulator kann ebenso für, die nicht spezifisch mit dem ELFE-Modell arbeiten. Als Ausblick wäre auch eine Erweiterung denkbar, welche alle wesentlichen Bestandteile der AKSEN-Bibliothelk im Emulator nachbildet, um eine vollständige Simulation des Mikrocontroller-Boards zu ermöglichen.
Während die C-Programme selbst ohne größere Schwierigkeiten entwickelt werden konnte, erforderte deren Integration in BREVE viel Zeit und mehrfache Neukonzipierung.

Kapitel 6.

Experimente

Mit dem entwickelten System wurde eine Reihe von Experimenten durchgeführt, die insbesondere Aufschluss über folgende drei Fragen geben sollen:

1. Funktioniert das System, so dass der GP-Kernel Individuen erzeugt, die korrekt über den Evaluator ausgeführt werden?

2. Ist ein Vorteil in der Evolution zu beachten, wenn Sensorik eingesetzt wird?

3. Kann mit dem System ein erfolgreiches Laufmuster für das ELFE-Modell erzeugt werden?

Da innerhalb dieser Arbeit der ELFE-Roboter nicht derart modifiziert werden konnte, dass eine Messung der Fitness über Bewegungssensoren möglich war, fand der Evolutionszyklus ausschließlich zwischen dem GP-Kernel und dem Simulator statt.
Die Bedingungen innerhalb der Simulation konnten teilweise idealisiert werden: Das Robotermodell befindet sich zu Anfang eines jeden Versuches auf dem Zentrum einer barrierefreien Ebene mit $1000\,m^2$ Fläche. Abhängig von der Konfiguration des Experiments wird das Individuum einmal oder mehrfach ausgeführt und das Robotermodell danach in eine stehende oder liegende Haltung gebracht. Dabei ist zu beachten, dass je nach Geschwindigkeit der Simulation die Servomotoren nicht genau die angesteuerten Winkel erreichen. Individuen werden somit meist nicht getreu ihrem Steuerungsprogramm ausgeführt, auch wird der Roboter beim Rücksetzen nicht in eine exakt definierte Haltung gebracht.

Die Versuchsreihe wurde in zwei Experimente gegliedert. Zur Demonstration der Funktionstauglichkeit des evolutionären Zyklus, sollte die ELFE als erste Aufgabe aus einer liegenden Position zum Aufstehen gebracht werden. Im zweiten Experiment wurde untersucht, ob das Modell ein ungerichtetes Laufmuster zur Flucht vom urprünglichen Ort erlernen kann. Die vollständige Dokumentation aller Experimente befindet sich auf der

zugehörigen CD dieser Arbeit. Dies umfasst Protokolldateien welche jeweils das beste Individuum jeder Generation als S-Expression mit zugehöriger Fitness aufführen, wie auch alle Graphen zur Visualisierung der Fitness und durchschnittlichen Knotenanzahl. Zudem können alle Versuche entweder über die Konfigurationsdateien des ECJ-Frameworks oder die einzelnen checkpoints eines Evolutionslaufes wiederholt werden.

6.1. Aufsteh-Problem

Als erste Aufgabe sollte die ELFE versuchen, aus einer liegenden Position aufzustehen. Zu Beginn jeder Evaluierung sind alle Beine des Roboters auf 0° angewinkelt, so dass sie parallel zum Untergrund ausgerichtet sind. Ein Individuum muss somit eine Steuersequenz finden, die alle elf Servomotoren auf 90° bewegt, um den Körper aufzurichten. Die Fitness wurde als summierte Differenz der Beinstellungen zum Optimum 90° definiert. Um in der Anfangsphase der Evolution bessere Individuen zu bevorzugen, wurde die Fitness zusätzlich quadriert. Sie ergibt sich somit nach der Formel

$$f_i = (\sum_{b=0}^{10} |rot_b - 90|)^2$$

mit der Fitness f des Individuums i und der Rotation im Gradmaß rot des Beines b. Der maximale Fehler und somit die schlechteste Fitness beläuft sich demnach auf 980100.

Vorüberlegung

Das Problem wurde bewusst in der Annahme gewählt, dass es schnell und leicht zu lösen sei, da ein erfolgreiches Individuum auch manuell sehr einfach zu entwerfen ist. Eine Möglichkeit wäre, einen Servo auf 90° zu drehen und diesen Befehl als Winkelparameter eines jeweils vorgeschalteten Servo-Befehls zu nehmen:

```
(setservo(0, setservo(1, setservo(2, setservo(3, setservo(4,
   setservo(5, setservo(6, setservo(7, setservo(8, setservo(9,
      setservo(10, 90))))))))))))
```

Insgesamt wurden drei Versuche jeweils drei mal mit verschieden initialisierten Zufallsgeneratoren durchgeführt. Die Größe der Population von 100 Individuen sowie die Anzahl der Generationen (30) wurden intuitiv als ausreichend erachtet, um eine Lösung zu finden, welche zumindest annähernd optimal wäre. Dabei war zu bedenken, dass aufgrund der Servoungenauigkeit eventuell ein optimales Individuum entwickelt, aber nicht entsprechend gut bewertet worden wäre.

Da im kompaktesten Fall zur Lösung des Problems nur Servo-Befehle und Konstanten gebraucht werden, eine zeitliche Abfolge der Sequenz jedoch nicht von Bedeutung ist, wurden Sleep-Befehle zum Vergleich nur in einem Versuch integriert. Auf diese Weise konnte der Ablauf der Evolution erheblich beschleunigt werden. Zusätzlich galt, dass wenig erfolgserscheinende Individuen, die nicht zumindest drei Servo-Befehle enthielten, mit der schlechtesten Fitness bestraft und nicht ausgeführt wurden. Die Tabelle 6.1 zeigt die Auflistung der wichtigsten Parameter und Bedingungen sowie das Funktionsset, welche als Basis für diese Versuchsreihe benutzt wurde.

Hauptparameter	
Populationsgrösse	100
Anzahl Generationen	30
Nebenparameter	
Wahrscheinlichkeit Rekombination	0.9
Wahrscheinlichkeit Reproduktion	0.1
Wahrscheinlichkeit Mutation	0.1
Maximale Baumtiefe während des Laufs	10
Maximale Baumtiefe bei Initialisierung	5
Qualitative Parameter	
Initialisierungsmethode	ramped half-and-half (50% grow, 50% full)
Selektionsmethode	rangbasiertes Turnier der Grösse 7
Fitnessmaß	Raw Fitness
weitere Einstellungen	
Anzahl Servos, um ausgewertet zu werden	3
Funktionen	ADD, SUB, MUL, DIV, PROG4, SETSERVO
Terminale	Konstante [0..255]
Anzahl Ausführungen	1

Tabelle 6.1.: Basiskonfiguration des Aufsteh-Problems

6.1.1. Fehlgeschlagene Experimente

Die ersten Versuchreihen bestanden aus drei verschiedenen Experimenten zum Aufsteh-Problem, die sich in Verwendung von Sensorik unterschieden. Jeder Versuch wurde dabei dreimal ausgeführt, um zufällige Ergebnisse erkennen und dementsprechend bewerten zu können. Allerdings zeichnete sich in allen Evolutionsläufen nach den ersten erfolgreichen Generationen ein Verharren in einem lokalen Optimum ab, wie in der Entwicklung der durchschnittlichen Fitnesswerte in Abb. 6.1 zu sehen ist. Sie entstammt dem erste Experiment über 30 Generationen ohne Sensorik.

Bei einer Überprüfung der Evolutionsparameter des GP-Kernels wurden zwei Einstellungen als Grund der Stagnation näher betrachtet: Zum einen wurde vermutet, dass die

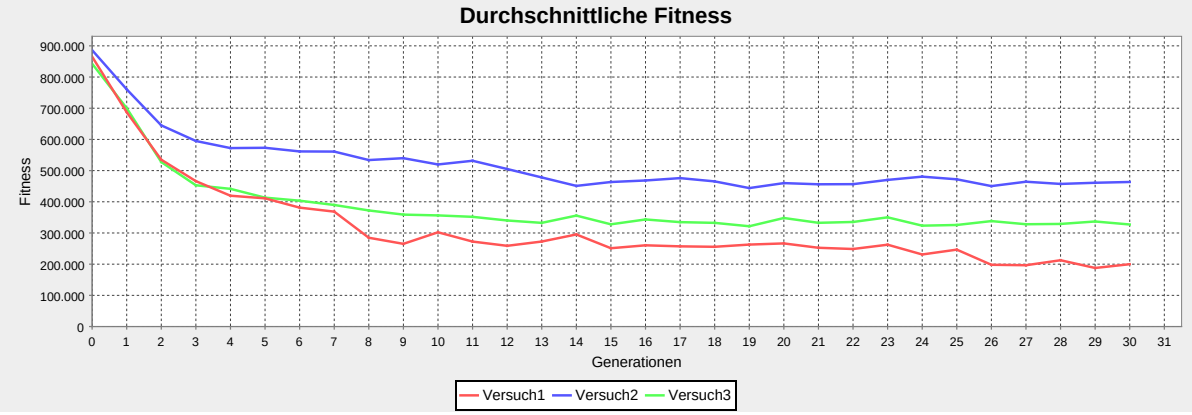

Abbildung 6.1.: Durchschn. Fitnesswerte aus Experiment 1 zum Aufsteh-Experiment

Mutationsrate von 0.01 nicht ausreiche, um neues Genmaterial zu schaffen. Zum anderen fiel auf, dass die maximale Baumtiefe bei der Rekombination auf den Wert 5 eingestellt war, obwohl die Bäume insgesamt bis zur Tiefe 10 wachsen durften.
Durch drei Experimente konnte herausgefunden werden, dass die Rekombinationsbeschränkung für eine erfolgreiche Evolution auf die maximale Baumtiefe gesetzt werden musste, die Mutationsrate hingegen keinen übermäßigen Einfluss zu haben scheint. Die Abb. 6.2 zeigt den Fitnessverlauf des wiederholten Aufstehexperiments ohne Sensorik und mit identisch initialisiertem Zufallsgenerator:

Versuch1 Rekombinationstiefe max. 5, Mutation = 0.02: Erwartete Stagnation ab Generation 9 ungefähr bei der Hälfte des erreichbaren Fitness.

Versuch2 Rekombinationstiefe max. 10, Mutation = 0.2: Stetige Verbesserung Fitness mit Erreichen eines approximierten Optimums bei Generation 24.

Versuch3 Rekmbinationstiefe max. 10, Mutation = 0.02: Ähnlicher Verlauf wie Versuch 2 mit steter Annäherung an Optimum.

Die Experimente wurde daraufhin mit der veränderten Basiskonfiguration in je zwei Versuchen wiederholt. Zudem wurde die Erfahrung aus den erfolgreichen Versuchen 1 und 2 genutzt, um anzunehmen, dass eine Verkürzung der Evolution auf 20 Generationen ausreicht, um ein nahezu optimales Individuum finden zu können.

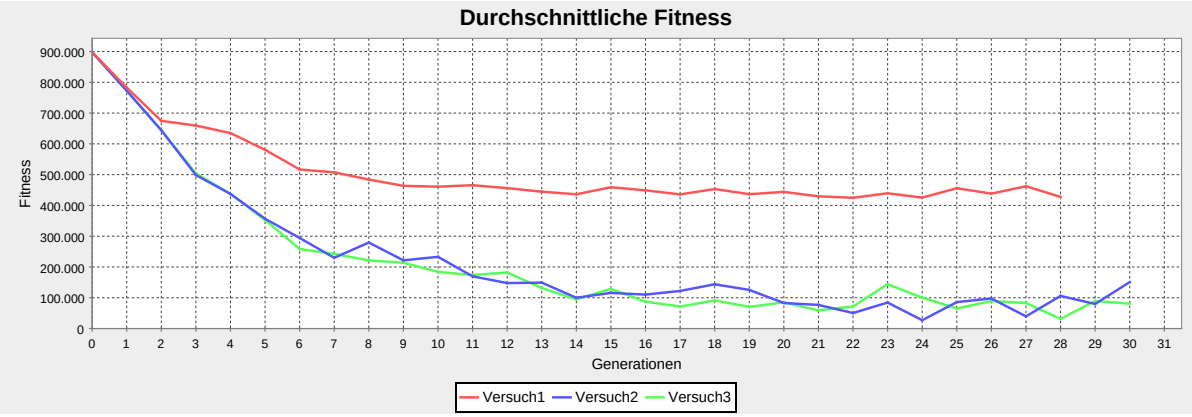

Abbildung 6.2.: Vergleich der Fitness des Aufsteh-Experiments mit verschiedener Konfiguration

6.1.2. Experiment 1

Im ersten Experiment wurden die Basisparameter unverändert übernommen, auf Sensorik wurde verzichtet. Ebenfalls wurde der Verzweigungsknoten nicht in die Funktionsmenge aufgenommen, da er nur von zusätzlichem Nutzen ist, wenn Sensoreingaben verarbeitet werden.

Die wichtigsten Beobachtungen der zwei Versuche sind wie folgt:

- Rapide Verbesserung der durchschnittlichen Fitness bereits in den ersten Generationen (Abb. 6.3)
- Erreichen eines approximierten Optimums in der durchschnittlichen Fitness zum Ende des Laufs (6.3, Generation 20).
- Exemplarisch an der Fitness der besten Individuen aus Versuch 2 zu erkennen: Erreichen eine approximierten Optimums in Generation 15.

6.1.3. Experiment 2

Das zweite Experiment wurde um IR-Sensorik und den Verzweigungsknoten erweitert. Auf diese Weise wurde neben der flexiblen Struktur der Individuen der zweite entscheidende Vorteil der GP eingebracht, die Möglichkeit eines Individuums, auf Umwelteinflüsse zu reagieren. Es wurde erwartet, dass Lösungen entwickelt würden, welche die Motoren teilweise in Abhängigkeit der Sensorik steuern. Dies könnte im Sinne einer

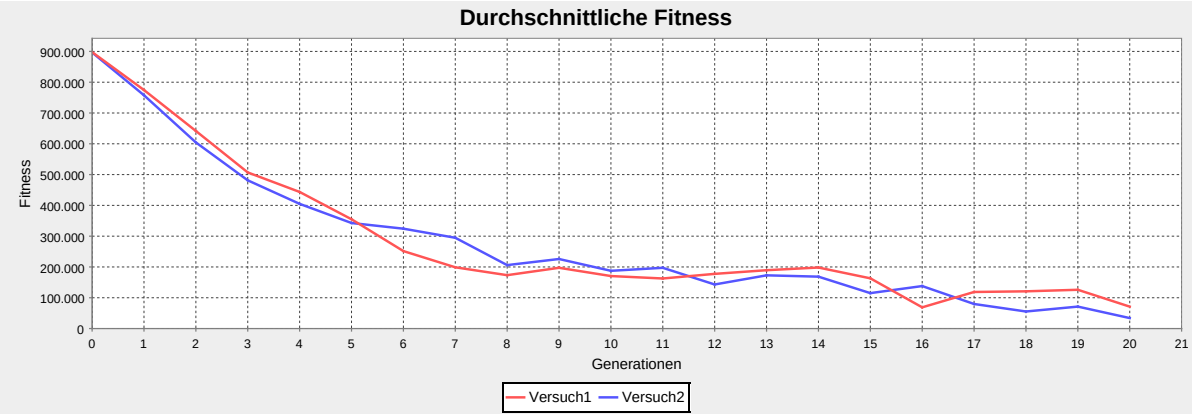

Abbildung 6.3.: Durchschn. Fitness beider Versuche aus Experiment 1

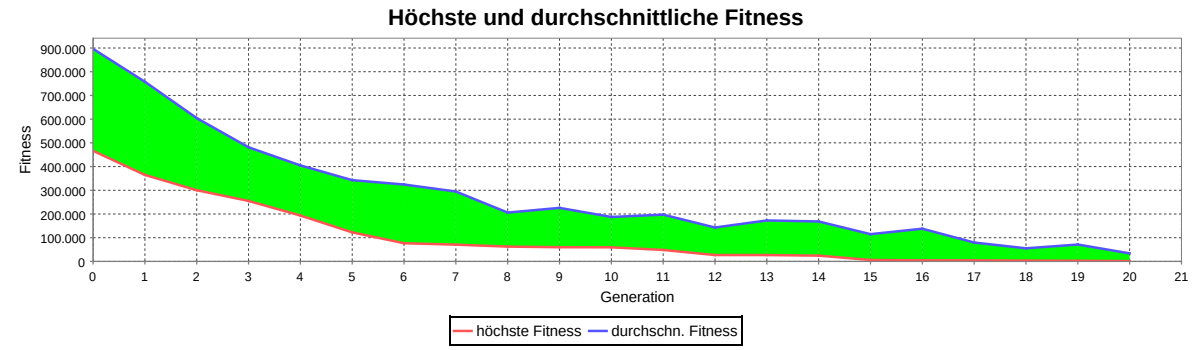

Abbildung 6.4.: Fitness des zweiten Versuches aus Experiment 1

flüssigen und effektiven Bewegung von Nutzen sein, um beispielsweise ein Bein aufrechtzustellen, bevor der Körper des Roboters zu nahe an den Untergrund gelangt.

Als Beispiel soll der erste Versuch des Experiments näher betrachtet werden (6.5):

- Rapide Entwicklung der Fitness in der Initialphase der Evolution (bis Gen. 4).
- Gesamter Verlauf der Fitness etwas schlechter als beim ersten Experiment ohne Sensorik
- Protokolldateien zeigen, dass Sensorknoten regelmäßig in den Gewinnerindividuen auftauchen

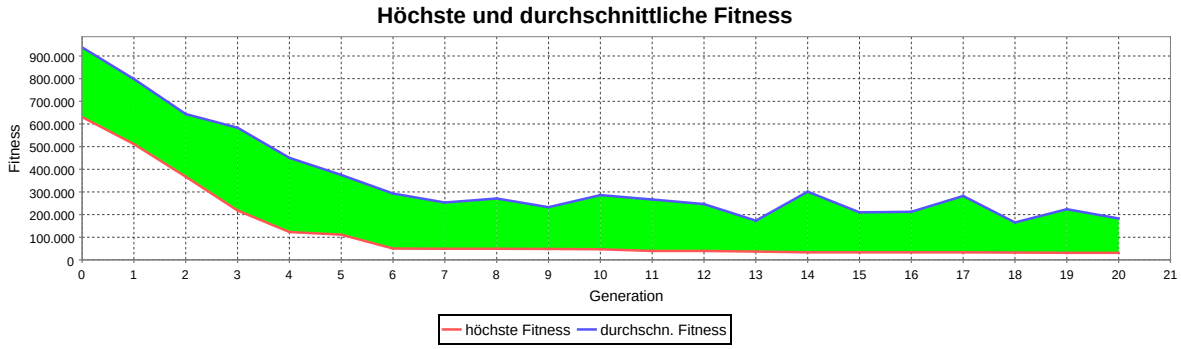

Abbildung 6.5.: Fitnessverlauf ersten Versuches von Experiment 2

6.1.4. Experiment 3

Um eine stärkere Auswirkung der Sensorik auf die Aufgabenstellung zu testen, wurden alle Individuen mit weniger als zwei IR-Knoten bestraft. Sie erhielten 20 % des schlechtesten Fitnesswertes addiert, ohne, dass dieser überschritten werden konnte.

6.1.5. Zusammenfassung

Die Experimente haben die Funktionstauglichkeit des Evaluationssystem demonstrieren können. Es wurden automatisch Steuerungsprogramme entwickelt, die den Roboter zum Aufstehen befähigten. Zwar wurde keine optimale Lösung entdeckt, das Gewinnerindividuum aller Versuche (aus Experiment 1 Versuch 2) schaffte nur eine Minimierung auf den Fitnesswert von 1369. Dies bedeutet allerdings nur einen summierten Winkelfehler von 37°.

```
(/ (setservo (setservo 129 103) (setservo
     (setservo 84 62) (setservo 241 85))) (prog4
     (* (/ (setservo (+ 59 243) (setservo (/ 204
         154) (+ 76 10))) (setservo 241 85)) (prog4
         (+ (setservo 245 (setservo (setservo 85 (+
             76 10)) (setservo 241 85))) 102) (setservo
         (/ 204 154) (+ 76 10)) (prog4 (+ 33 140)
         (+ 33 140) (setservo 149 84) (- 61 207))
         (* (+ 140 136) (+ (setservo 187 97) 245))))
     (setservo (+ 27 66) (+ 76 10)) (setservo
     241 85) 61))
```

6.2. Flucht Problem

In einer zweiten Aufgabe sollte die ELFE versuchen, sich aus der liegenden Position heraus möglicht weit vom ursprünglichen Ort zu entfernen. Eine gerichtete Bewegung des Fluchtverhaltens spielte keine Rolle, als Fitnessfunktion f des Individuums i wurde nur die Distanz zwischen dem Startpunkt ($\overrightarrow{endPos}$) und dem Endpunkt ($\overrightarrow{endPos}$) gemessen:

$$f_i = |\overrightarrow{endPos} - \overrightarrow{startPos}|$$

Der y-Abschnitt der dreidimensionalen Vektoren wurde hierbei auf null gesetzt, um nur die Strecke in der Ebene des Untergrundes zu messen. Für jedes Individuum wurde die Ausführung der Steuersequenz drei mal wiederholt. In der Tabelle 6.2 ist die Grundkonfiguration des Problems aufgeüfhrt.

Hauptparameter	
Populationsgrösse	150
Anzahl Generationen	zeitliche Begrenzung
Nebenparameter	
Wahrscheinlichkeit Rekombination	0.9
Wahrscheinlichkeit Reproduktion	0.1
Wahrscheinlichkeit Mutation	0.1
Maximale Baumtiefe während des Laufs	17
Maximale Baumtiefe bei Initialisierung	5
Qualitative Parameter	
Initialisierungsmethode	ramped half-and-half (50% grow, 50% full)
Selektionsmethode	rangbasiertes Turnier der Grösse 7
Fitnessmaß	Raw Fitness
weitere Einstellungen	
Anzahl Servos, um ausgewertet zu werden	3
Funktionen	ADD,SUB,MUL,DIV,PROG4,SERVO,IR,SLEEP
Terminale	Konstante [0..255]
Anzahl Ausführungen	3

Tabelle 6.2.: Basiskonfiguration des Flucht-Problems

Vorüberlegung

Die Flucht-Aufgabe stellt ein Problem dar, welches im Gegensatz zum ersten Experiment nicht mehr auf einfache Weise intuitiv zu lösen ist. Es ist weder bekannt, welche Länge eine effektive Steuersequenz haben muss noch ob der Einsatz von allen oder nur einem Teil der Beine einen Vorteil bringt.
Da eine Fortbewegung grundsätzlich aus einer rhythmischen Wiederholung einer kürzeren

Sequenz besteht, wurde der Roboter nur zu Beginn in die liegende Ausgangsposition gebracht und das Individuum dreimal hintereinander ausgeführt. Auf diese Weise war zu erwarten, dass die Knotenanzahl der Individuen mit der Fitness steigen würde, wodurch mehr Aktionen pro Lauf mögliche wären. Demnach könnten theoretisch unendlich lange Steuerungssequenzen generiert werden, was letztendlich aber durch die Begrenzung der maximalen Baumtiefe eines Inviduums verhindert wurde.
Desweiteren wurde angenommen, dass eine erfolgreiche Fortbewegung nur entwickelt werden kann, wenn neben den Befehlen zur Ansteuerung des Motors auch die sleep-Funktion zur Verfügung gestellt würde, um eine Staffelung der Einzelbewegungen aller Beine zu ermöglichen.

Fehlgeschlagene Versuche

Wie auch bei den Experimenten zum Aufsteh-Problem wurden die ersten Versuche mit einer zu geringen Rekombinationstiefe durchgeführt. In der Abb. 6.6 ist dabei ein ähnliches Schema zu beobachten: Die durchschnittliche Fitness steigt in den ersten Generationen zunächst sprunghaft an, verläuft aber etwa ab Generation 18 in einem lokalen Optimum. Dabei wurden 150 Individuen zunächst ohne Verwendung von Sensorik über 50 Generationen evolviert.

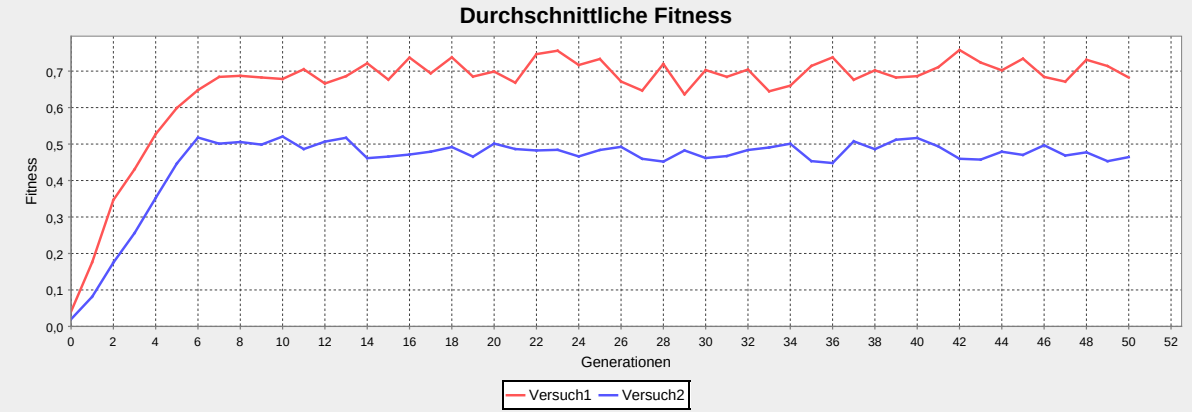

Abbildung 6.6.: Durchschn. Fitness der zwei Versuche des fehlgeschlagenen Flucht-Experiments

In einem zweiten Experiment wurde das Funktionsset um Sensorik und den Verzweigungsknoten erweitert. Zusätzlich musste jedes Individuum zumindest zwei Sensorknoten enthalten, um nicht als Strafe 50 % Abzug der Fitness zu erhalten. Es wurde vermutet, die Komplexität dieses Versuches wäre zu hoch, um in einer Zeit von 50 Generationen die stagnierende Initialphase zu überwinden. Daher wurde die Anzahl der Generationen auf 200 erhöht. Wie in der Abb. 6.7 im Verlauf der Fitness zu sehen ist, wurde ab Generation 20 weder eine Verbesserung noch eine Verschlechterung der Eliteindividuen

bewirkt. Ebenfalls war durch die nahezu regelmäßigen Schwankungen der durchschnittlichen Fitness keine weitere Änderung zu erwarten.

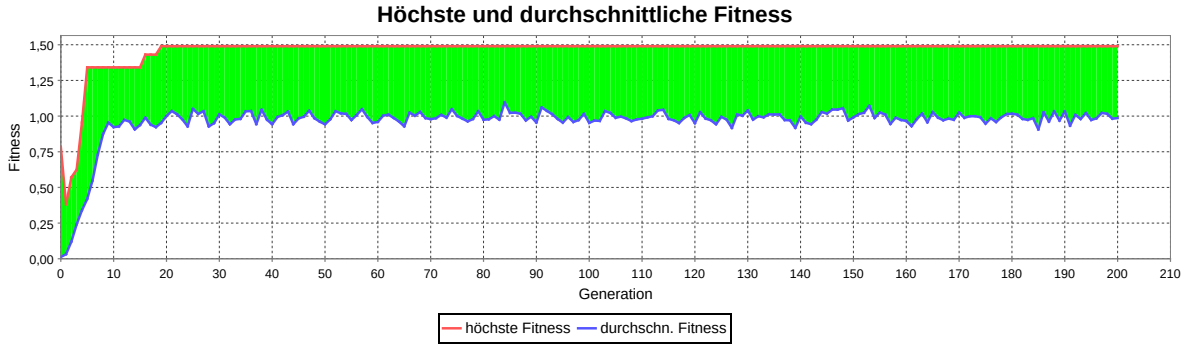

Abbildung 6.7.: Fitness des ersten Versuches von Experiment 2

6.2.1. Experiment 1

Durch die zeitintensiven vorangegangenen Versuche konnte mit der korrigierten Rekombinationstiefe nur noch ein Langzeitversuch zur Flucht-Aufgabe durchgeführt werden. Daher wurden das gesamte Funktionsset genutzt und der Einsatz von Sensorik über zusätzliche Bestrafung forciert.
Diesmal wurde die Versuchsdauer jedoch nicht durch eine maximale Anzahl von Generationen begrenzt, sondern nach etwa 40 Stunden abgebrochen. Der Erfolg des Experiments konnte dabei nicht nur in Verlauf der Fitness sondern auch im Simulator beobachtet werden. Eine Betrachtung der Abb. 6.8 zeigt die kontinueriliche Verbesserung der Fitnesswerte bis zum Erreichen eines vorzeitigen Optimums in Generation 58. Besonders in der ersten Hälfte der Evolution ist gut zu erkennen, wie sich Individuen sprunghaft in der Population durchsetzen (besonders in Gen. 4, 22) und trotz der teilweise ebenso großen Einbrüche in Richtung einer Optimierung weiterentwickeln.

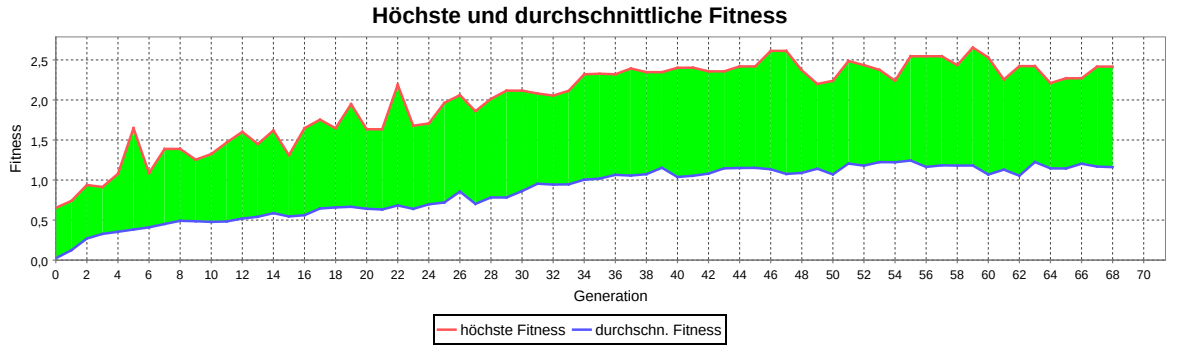

Abbildung 6.8.: Fitnessverlauf des Flucht-Experiments

Anhand des Graphen stellt sich zunächst die Frage, ob das langsame Einschwingen der Fitness (ab Generation 40 etwa knapp unter dem Wert $2,5$) eine Grenze der Evolution darstellt. Zu diesem Zweck wird auch die Anzahl der Knoten mit in die Überlegung eingezogen. Die Abb. 6.9 zeigt über den wesentlichen Teil der Evolution eine durchschnittliche Anzahl von etwa 100 Knoten, während die höchsten Werte bis knapp über 225 ausschlagen. Anhand der unregelmäßigen Arität der verwendeten Funktionen lässt sich aus der eingestellten Baumtiefe keine genaue Aussage über maximalen Anzahl der Knoten folgern.
Allerdings fällt ein Zusammenhang zur Konfiguration des AksenEmulators auf: Der Autor wählte intuitiv eine Begrenzung von 200 Knoten pro Individuum aus, um bei ersten Versuchen mit dem realen Mikrocontroller ein Überlaufen des Stacks zu verhindern. Längere Steuersequenzen wurden nicht ausgeführt und somit mit der schlechtesten Fitness bewertet. Es ist daher anzunehmen, dass die Anzahl an Knoten über den Lauf etwa bei diesem Wert konvergiert.

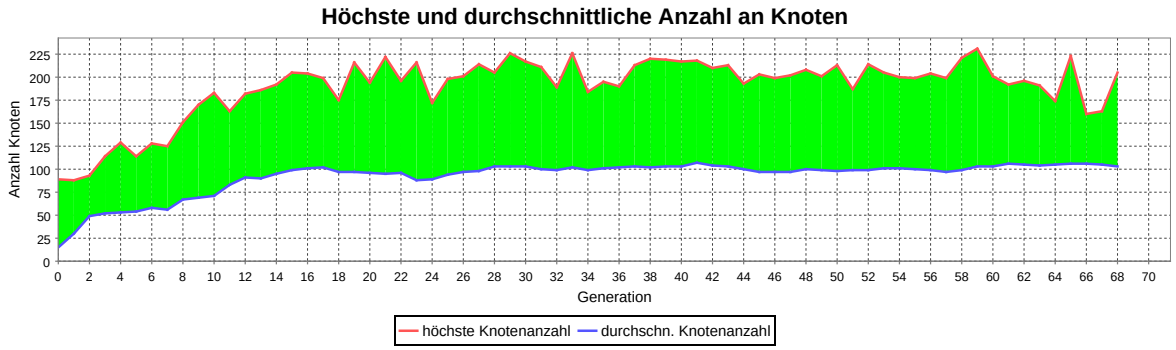

Abbildung 6.9.: Entwicklung der Knotenanzahl des Flucht-Experiments

Bei der näheren Analyse der Gewinnerindividuen aus der Protokolldatei des Versuchs können folgende Beobachtung festgehalten werden:

- Die Steuersequenzen bestehen hauptsächlich aus verschachtelten und seriellen Servobefehlen.
- Mathematische Ausdrücke verhältnismäßig rar eingesetzt.
- Die erfolgreichsten Individuen bleibt weit unter der möglichen Länge bei etwa 100 bis 150 Knoten.
- Sleep-Befehle werden regelmäßig, aber sparsam genutzt

Das Gewinnerindividuum aus Generation 58 entwickelte eine Art „Froschhüpfen“. Dabei wurden fast alle Beine waagerecht angewinkelt, um nicht im Weg zu stehen. Zwei sich gegenüberliegende Beine werden aufgestützt und in einer Bewegung nach hinten abgestoßen. Dabei dient ein drittes Bein als Sicherung, um nicht nach vorne zu kippen. Der

interessanteste Aspekt ist, dass das Individuum nach jedem Sprung den Roboter neu ausrichtet: Es wird kurz pausiert und mit alleiniger Bewegung des einen Sprungbeines die Richtung korrigiert. Auf diese Weise wählt der Roboter einen möglichst geraden Pfad vom Ursprung zurück, welches die optimale Strategie im Sinne der Fitnessfunktion ist. Ein Video der Ausführung ist auf der CD zu finden.
Das Gewinnerindividuum ist im Folgenden als S-Expression aufgelistet, es erreichte eine Geschwindigkeit von etwa 54 cm/s:

```
(setservo (/ (- (setservo (/ (/ (ir 89) 236)
     236) (ir 157)) (* (ir 207) 136)) (ifgt (/
     (setservo 1 238) (setservo 6 154)) (sleep
     (ir 89)) (setservo 209 (setservo 127 39))
     (/ (/ (ir 89) (setservo 127 39)) 49))) (prog4
     (prog4 (setservo 6 154) (ir 104) 6 (/ (+
         (sleep 150) 161) (setservo 6 154))) (ir 89)
     249 (ifgt (/ 225 189) (setservo 127 39) (/
     (sleep 243) (setservo 6 154)) (* 79 135))))
```

6.2.2. Zusammenfassung

Es konnte erfolgreich ein Fortbewegungsmuster automatisch generiert werden, welche der ELFE eine Flucht vom letzten Standpunkt ermöglicht. Die Lösung ist wahrscheinlich nur suboptimal, deutet jedoch das Potential an, sollte die Evolution fortgeführt und verfeinert werden. Es ist zu betonen, dass eine sehr einfache Fitness für das Experiment gewählt wurde, welche nur im Ansatz und im zusammenhang mit der Sensorik gezeigt hat, wie multiple Ziele vereint werden können.

Kapitel 7.

Zusammenfassung und Ausblick

7.1. Zusammenfassung

Im Mittelpunkt dieser Arbeit war die automatische Generierung von Bewegungsmustern für die Morphologie des Schreitroboters ELFE. Hierzu wurde das Problem als eine Optimierungsaufgabe begriffen, welche aufgrund der hohen Komplexität ein geeignetes Beispiel zur Demonstration von Methoden des Maschinellen Lernens darstellt. In einer vergleichenden Untersuchung von Lernverfahren mit bestärkender Rückkopplung wurden insbesondere die Evolutionären Algorithmen näher untersucht. Die Indizien zur deren Anwendbarkeit (Vgl. [Boe00]) konnten auf das Problem zur Erzeugung eines Laufmusters für die ELFE bezogen werden:

Fehlendes Prozessmodell Es existiert kein Prozessmodell. Zudem wurde der Roboter nicht für eine optimale Fortbewegung konstruiert, eine Lösung ist daher nur intuitiv und unter hohem Aufwand zu finden.

Testbarkeit des Suchraums Es ist möglich, beliebig viele Steuerungsvarianten zu testen.

Wahrnehmbarkeit Nach Ablauf der Steuerprogrammen ist sofort eine Fitness in Form des Weges berechenbar. Die Sequenz der Steuerprogramme ist ausreichend kurz, da sie für eine zyklische Ausführung ausgelegt ist.

Komplexe Lösungseigenschaften Es wurde durch den Einsatz von Sensorik und deren Integration in die Fitnessfunktion versucht, ein Bewegungsmuster zu erzeugen, welches nicht nur effektiv ist, sondern zusätzlich ein Schleifen des Körpers verhindert.

Zur Bearbeitung des Problems wurde auf die Methode des Genetischen Programmierens zurückgegriffen. Sie ermöglicht die automatische Erzeugung von flexiblen und aktiven Programmstrukturen, wodurch zwei Vorteile gegenüber den herkömmlichen EA genutzt werden konnten:

- Die Länge der Steuerungssequenz muß nicht im Voraus bekannt sein. Sie können ohne Einschränkung wachsen, wenn es im Sinne der Evolution von Vorteil ist.

- Das aktive Individuum kann Sensorik benutzen und somit die Umwelt wahrnehmen. Die Steuerungssequenz ist in der Lage, passt sich der aktuellen Situation anzupassen.

Im technischen Teil der Implementation konnte die Aufgabenstellung der Arbeit nicht im vollen Umfang erfüllt werden. Ursprünglich war angedacht, die Steuerprogramme primär mit dem realen Roboter zu entwickeln und sekundär den Einsatz eines Simulators zu erwägen. Der Schwerpunkt der Arbeit verlagerte sich jedoch auf die Integration des Simulationssystems BREVE in den evolutionären Prozess. Der Autor versprach sich davon folgende Vorteile:

- Beschleunigung der Evolution durch zeitliche Skalierung der Simulation

- Schonung der Roboterkomponenten bzw. Vermeidung von Defekten während der Versuche

- Schaffung einer idealisierten Umwelt ohne Einflüsse von Störfaktoren (Sensorrauschen etc.)

- Erweiterte Möglichkeiten zur schnellen und genauen Messung der Fitness (Motorstellung, zurückgelegter Weg)

- Identische Startbedingungen für jedes Individuum

Die Arbeit mit dem Simulationssystem BREVE hat letztendlich aber nicht die Vorteile gebracht, die angestrebt worden sind. Die diversen Hürden bei der Entwicklung haben mehr Zeit in Anspruch genommen, als durch die Versuche mit dem Simulator gewonnen werden konnte. Besondere Schwierigkeiten bereitete der Umgang mit den Plugins:

- Bis zur Version 2.3 wurden Plugins vom Simulator nicht geladen. Der ausgegebene Fehler suggerierte ein Fehlen der entsprechenden DLL.

- Auch in der zuletzt genutzten Version 2.5b existieren diverse Fehler in den Standardfunktionalitäten. Unter anderem fehlt in der Plugin-Bibliothek die Implementation der deklarierten Listenstruktur und die Funktionen zur Synchronisation der

callback-Aufrufe verursachen Deadlocks.

- Die Plugin-Bibliothek wird auch in der Distribution für Windows nur im Unix-Format beigelegt. Auf diese Weise müssen die Plugins plattformübergreifend kompiliert werden.

- Aus der Entwicklergemeinschaft ist es noch niemandem gelungen, den Quellcode von BREVE selbst zu kompilieren. Die notwendigen makefiles sind umfangreich und unübersichtlich aufgebaut, können jedoch nicht umgangen werden, da sie die korrekte Verknüpfung einzelner Komponenten übernehmen. Auf diese Weise ist die Nutzung eines Debuggers oder Profilers ausgeschlossen.

- Es existiert keine Art der Fehlerbehandlung. Meist ist die einzige Möglichkeit, eine Ausnahme festzustellen, der Absturz des Systems. Auch ist für den Entwickler keine sichere Möglichkeit gegeben, um Fehler an die laufende Simulation zu propagieren und zu behandeln.

Darüber hinaus braucht es Zeit, die Eigenheiten und Fehler der steve-Skriptsprache zu berücksichtigen. Neben den vielen Schwächen der Programmumgebung, welche dem Quelltext häufig nicht korrekt parst oder die Dateien nicht speichert, wirkten sich aber besonders die folgenden Misskonzepte auf die Simulation der ELFE aus:

- Simulationskernel und Anwenderoberfläche in einem Thread. Blockierende Aufrufe führen so meist zum Absturz und nehmen dem Benutzer jegliche Interaktionsmöglichkeit. Dieser Fehler führte zu der Notwendigkeit, den Server in einer aufwändigeren asynchronen Version zu betreiben.

- Da es versäumt wurde, in der Skriptsprache steve eine Möglichkeit zur unbedingten Kalibrierung eines Gelenkwinkels zu implementieren, ist es nicht möglich, die virtuellen Servomotoren akkurat anzusteuern oder die Simulationsgeschwindigkeit zu erhöhen. Auf diese Weise erhöht sich die Diskrepanz zwischen dem Modell und dem realen Roboter.

- Es ist nicht möglich, den Ablauf eines steve-Skriptes zu pausieren, ohne auch die Simulation zu unterbrechen und somit die Stabilität der gesamten Applikation zu gefährden. Eine sinnvolle Implementation des sleep-Befehls hätte zugelassen, den GP-Evaluator wie geplant in als Skript umzusetzen.

Eine der Hauptmotivationen zur Wahl von BREVE war die Nutzergemeinschaft, welche in einem aktiven Forum alle Themen des Simulators diskutiert. Allerdings stößt auch diese Hilfe schnell an ihre Grenzen, da von den wenigen hundert registrierten Nutzern schätzungsweise ein dutzend regelmäßig mit BREVE arbeiten. Die Sparte der Plugin-entwicklung hingegen zeigt kaum Aktivität. Auch die Korrespondenz mit dem Autor

Jon Klein ist von schwankender Qualität. Bei einigen Problemen kam eine Hilfe innerhalb von wenigen Tagen, etwa Korrekturen der Skriptsprache oder die Analyse eines 3D-Modells, während in Bereichen der Pluginentwicklung oft nicht einmal eine Antwort zu erwarten war.
Dennoch sieht der Autor in BREVE großes Potential. Trotz einer gewissen Unstätigkeit in der Pflege der Systems war dessen Weiterentwicklung deutlich zu erkennen. Werden in zukünftigen Versionen die genannten Mängel behoben, ist BREVE als ein mächtiger und flexibler Agentensimulator zu betrachten.

Die meisten Probleme waren nicht vorhersehbar und mussten durch Kompromisslösungen umgangen werden, wodurch die Entwicklung der Simulation wesentlich mehr Zeit beanspruchte, als angenommen. Im Sinne der Aufgabenstellung wäre es sicherlich sinnvoll gewesen, ab einem gewissen Punkt die Entwicklung des Simulators einzustellen und den Fokus auf den realen Roboter zu richten. Der Autor war jedoch bestrebt, die Ergebnisse der bisherigen Bemühungen zu verwenden und auf die Vorteile eines hybriden Evaluationssystems hinauszuarbeiten.
Die Experimente unterstreichen dabei einen Teilerfolg der Arbeit. Die Funktionstauglichkeit des verteilten Evolutionssystems, bestehend aus dem GP-Kernel und der Simulation, konnte bewiesen und in mehreren Versuchen demonstriert werden. Dabei wurde gezeigt, wie auch komplexe und intuitiv schwer zu beschreibende Steuerungssequenzen automatisch und über eine verhältnismäßig geringe Zeit generiert werden konnten. Um alle Anforderungen der Aufgabenstellung zu erfüllen, wären allerdings noch weitere Schritte notwendig gewesen. Es verbleiben somit folgende Fragen:

- Funktioniert der Evaluator zuverlässig auf dem AKSEN-Board trotz der zusätzlichen Beschränkungen (Speicher etc.)?

- In wie fern ist das Simulationsmodell auf den realen Roboter zu übertragen? Verhalten sich die Steuerprogramme ähnlich genug, um die Initialphase der Evolution zunächst simulieren lassen zu können?

- Wie kann die Fitnessmessung auf der ELFE mittels der angeschlossenen Maus realisiert werden? Ist zusätzliche Sensorik notwendig?

- Ergeben sich zusätzliche Probleme, die gelöst werden müssten, aber nicht nicht im idealisierten Simulator auftreten, wie beispielsweise eine Überbelastung der Komponenten, stark verrauschte Sensordaten oder die Begrenzung des Testgeländes?

Tabellenverzeichnis

Literaturverzeichnis

[AH06] Alshurafa, Nabil I. und Justin T. Harmon: *Artificial spider: eight-legged arachnid and autonomous learning of locomotion.* Technischer Bericht, Department of Computer Science/ University of California, 2006.

[BN97] Banzhaf, Wolfgang und Peter Nordin: *Real Time Control of a Khepera Robot using Genetic Programming.* Technischer Bericht, Universität Dortmund, 1997.

[BNKF97] Banzhaf, Wolfgang, Peter Nordin, Robert E. Keller und Frank D. Francone: *Genetic Programming : An Introduction : On the Automatic Evolution of Computer Programs and Its Applications (The Morgan Kaufmann Series in Artificial Intelligence).* Morgan Kaufmann Publishers, November 1997.

[Boe] Boersch, Ingo: *Internetseite des AKSEN-Boards.* Online im Internet: http://ots.fh-brandenburg.de/index.php (27.11.2006).

[Boe00] Boersch, Ingo: *Genetisches Programmieren einfacher Roboterfähigkeiten.* In: *4. Brandenburger Workshop Mechatronik.*, 2000.

[Bro89] Brooks, Rodney: *A Robot that Walks; Emergent Behaviors from a Carefully Evolved Network.* Technischer Bericht, Massachusetts Institute of Technology, 1989.

[Bro91] Brooks, Rodney A.: *The role of learning in autonomous robots.* In: *COLT '91: Proceedings of the fourth annual workshop on Computational learning theory*, Seiten 5–10, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[CNB+oJ] Capi, G., Y. Nasu, L. Barolli, M. Yamano, K. Mitobe und K. Takeda: *A Neural Network Implementation of Biped Robot Optimal Gait during Walking Generated by Genetic Algorithm*, o.J.

[Dar93] Darwin, Charles: *The origin of species.* Random House, 1993.

[Dow97] Dowling, Kevin J.: *Limbless Locomotion: Learning to Crawl with a Snake Robot.* Dissertation, The Robotics Institute/ Carnegie Mellon University, 1997.

[Eps95] Epstein, Marc E.: *Evolution of locomotion in slug caterpillars (Lepidoptera: Zygaenoidea: Limacodid group).* Journal of Research on the Lepidoptera, 34:1–13, 1995.

[FOW66] Fogel, L. J., A. J. Owens und M. J. Walsh: *Artificial Intelligence through Simulated Evolution.* John Wiley, New York, USA, 1966.

[GGBoJ] Gonzalez-Gomez, Juan und Eduardo Boemo: *Motion of Minimal Configurations of a Modular Robot: Sinusoidal, Lateral Rolling and Lateral Shift.* Technischer Bericht, School of Engineering. Universidad Autonoma de Madrid, o.J.

[GP] Gagné, Christian und Marc Parizeau: *OpenBEAGLE Projektseite.* Online im Internet: http://beagle.gel.ulaval.ca/ (15.11.2006).

[GP06] Gagné, Christian und Marc Parizeau: *Genericity in Evolutionary Computation Software Tools: Principles and Case Study.* International Journal on Artificial Intelligence Tools, 15(2):173–194, 2006.

[GRS03] Görz, Günther, Claus-Rainer Rollinger und Josef Schneeberger: *Handbuch der Künstlichen Intelligenz.* Oldenbourg, München, 2003.

[HFT+99] Hornby, G. S., M. Fujita, S. Takamura, T. Yamamoto und O. Hanagata: *Autonomous Evolution of Gaits with the Sony Quadruped Robot.* Technischer Bericht, Sony Corporation, Orlando, Florida, USA, 13-17 1999.

[Hol75] Holland, John: *Adaption in natural and artificial systems.* University of Michigan Press, Ann Arbor, 1975.

[HS02] Harbick, Kale und Gaurav S. Sukhatme: *Speed Control of a Pneumatic Monopod using a Neural Network.* Technischer Bericht, Institute for Robotics and Intelligent Systems Technical Report IRIS-02-413, 2002.

[HSA99] Heinsohn, Jochen und Rolf Socher-Ambrosius: *Wissensverarbeitung. Eine Einführung.* Spektrum, Berlin, 1999.

[IHW98] Ijspeert, Auke Jan, John Hallam und David Willshaw: *Evolution of a Central Pattern Generator for the Swimming and Trotting Gaits of the Salamander*, 1998.

[KB01] Kantschik, Wolfgang und Wolfgang Banzhaf: *Linear-Tree GP and Its Comparison with Other GP Structures.* In: *EuroGP '01: Proceedings of the 4th European Conference on Genetic Programming*, Seiten 302–312, London, UK, 2001. Springer-Verlag.

[KB02] Kantschik, Wolfgang und Wolfgang Banzhaf: *Linear-Graph GP. A new GP Structure.* In: Lutton, Evelyne, James A. Foster, Julian Miller, Conor Ryan und Andrea G. B. Tettamanzi (Herausgeber): *Proceedings of the 4th European Conference on Genetic Programming, EuroGP 2002*, Band 2278, Seiten 83–92, Kinsale, Ireland, 3-5 2002. Springer-Verlag.

[KeioJ] Keith Wiley: *Observations on the Evolution of Neural Networks for the Control of a Simulated Quadruped Robot.* Technischer Bericht, University of New Mexico, o.J.

[Kle] Klein, Jon: *Breve Class Documentation.* Online im Internet: http://www.spiderland.org/breve/documentation.php (14.11.2006).

[Kle02] Klein, Jon: *breve : An Environment for Simulation of Decentralized Systems and Artificial Life.* Diplomarbeit, Chalmers University, Göteborg, Sweden, 2002.

[Kle03] Klein, Jon: *Breve: a 3D environment for the simulation of decentralized systems and artificial life.* In: *ICAL 2003: Proceedings of the eighth international conference on Artificial life*, Seiten 329–334, Cambridge, MA, USA, 2003. MIT Press.

[KM94] Keith, Mike J. und Martin C. Martin: *Genetic Programming in C++: Implementation Issues.* In: Kinnear, Jr., Kenneth E. (Herausgeber): *Advances in Genetic Programming*, Seiten 285–310. MIT Press, 1994.

[Kom00] Komosinski, Maciej: *The World of Framsticks: Simulation, Evolution, Interaction.* In: *Proceedings of 2nd International Conference on Virtual Worlds.* Springer, 2000.

[Koz92] Koza, John R.: *Genetic programming: on the programming of computers by means of natural selection.* MIT Press, Cambridge, MA, USA, 1992.

[Kra02] Krause, Norman: *Untersuchung von Elementarbewegungen des Schreitroboter SimengDolores (Entwurf, Programmierung und Test).* Diplomarbeit, FH Brandenburg/ Fachbereich Informatik und Medien, 2002.

[LA87] Laarhoven, P. J. M. und E. H. L. Aarts: *Simulated annealing: theory and applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[Leg99] Leger, Chris: *Automated Synthesis and Optimization of Robot Configurations: An Evolutionary Approach*. Dissertation, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, December 1999.

[LFB93] Lewis, M., A. Fagg und G. Bekey: *Genetic algorithms for gait synthesis in a hexapod robot*, 1993.

[LH01] Lazarus, C. und H. Hu: *Using Genetic Programming to Evolve Robot Behaviours*. In: *Proceedings of the 3 rd British Workshop on Towards Intelligent Mobile Robots (TIMR) '01, Manchester*, 2001.

[LP00] Lipson und J. B. Pollack: *Automatic design and Manufacture of Robotic Lifeforms*. Nature, 406:974–978, 2000.

[LPB+] Luke, Sean, Liviu Panait, Gabriel Balan, Sean Paus, Zbigniew Skolicki, Jeff Bassett, Robert Hubley und Alexander Chircop: *ECJ Projektseite*. Online im Internet: http://cs.gmu.edu/ eclab/projects/ecj/ (15.11.2006).

[Mar05] Markelic, Irene: *Evolving a Neurocontroller for a Fast QuadrupedalWalking Behavior*. Diplomarbeit, Institut für Computervisualistik/ Universität Koblenz-Landau, 2005.

[MHA98] Miyashita, T., K. Hosoda und M. Asada: *Hybrid Structure of Reflective Gait Control and Visual Servoing for Walking*. In: *International Conference on Intelligent Robotics and Systems*, Seiten 229–234, 1998.

[Mic86] Michalski, R. S.: *Understanding the nature of learning: issues and research directions*. In: Michalski, R., J. Carbonnel und T. Mitchell (Herausgeber): *Machine Learning: An Artificial Intelligence Approach*, Band 2, Seiten 3–25. Kaufmann, Los Altos, CA, 1986.

[Mic99] Michalewicz, Zbigniew: *Genetic Algorithms + Datastructures = Evolution Programs*. Springer, 1999.

[Min86] Minsky, Marvin: *Society of Mind*. Simon & Schuster, 1986.

[Mit] Mitchell, Tom: *Lecture slides for the textbook Machine Learning*. Online im Internet: http://www.cs.cmu.edu/ tom/mlbook-chapter-slides.html (22.11.2006).

[Mit97] Mitchell, Tom M.: *Machine Learning.* McGraw-Hill, New York, 1997.

[NF99] Nolfi, Stefano und Dario Floreano: *Learning and Evolution.* Auton. Robots, 7(1):89–113, 1999.

[Nis94] Nissen, Volker: *Evolutionäre Algorithmen. Darstellung, Beispiele, betriebswirtschaftliche Anwendungsmöglichkeiten.* Deutscher Universitäts-Verlag, 1994.

[PJRWoJ] Pettersson, L., K. Jansson, H. Rehbinder und J. Wikander: *Behavior-based Control of a Four Legged Walking Robot*, o.J.

[Poh99] Pohlheim, Hartmut: *Evolutionäre Algorithmen. Verfahren, Operatoren und Hinweise für die Praxis.* Springer, 1999.

[Rec73] Rechenberg, Ingo: *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der Biologischen Evolution.* Frommann-Holzbog, Stuttgart, 1973.

[RIP+02] Rybak, Ilya A., Dmitry G. Ivashko, Boris I. Prilutsky, M. Anthony Lewis und John K. Chapin: *Modeling Neural Control of Locomotion: Integration of Reflex Circuits with CPG.* In: *ICANN '02: Proceedings of the International Conference on Artificial Neural Networks*, Seiten 99–104, London, UK, 2002. Springer-Verlag.

[SB98] Sutton, R.S. und A.G. Barto: *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.

[Sch75] Schwefel, Hans-Paul: *Evolutionsstrategie und numerische Optimierung.* Dissertation, Technische Universitat Berlin, 1975.

[Sch77] Schwefel, Hans-Paul: *Numerische Optimierung von Computer-Modellen mittels Evolutionsstrategie.* Birkhäuser, Basel, 1977.

[She01] Sherman, Mark W.: *Sine-Wave Locomotion in a Robotic Snake Model Form and Programming.* Technischer Bericht, American Association for Artificial Intelligence, 2001.

[SHF94] Schöneburg, Erberhard, Frank Heinzmann und Sven Feddersen: *Genetische Algorithmen und Evolutionsstrategien.* Addison-Wesley, 1994.

[Spe01] Spector, Lee: *Autoconstructive Evolution: Push, PushGP, and Pushpop.* In: Spector, L., E. Goodman, A. Wu, W.B. Langdon, H.-M. Voigt, M. Gen, M. Dorigo S. Sen, S. Pezeshk, M. Garzon und E. Burke.

(Herausgeber): *GECCO-2001: Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann, 2001.

[SSW03] Stoy, K., W.-M. Shen und P.M. Will: *A simple approach to the control of locomotion in self-reconfigurable robots*. Robotics and Autonomous Systems, 44:191–199, 2003.

[Ste06] Stellmann, Patrik: *Simulator Bob. 3D Simulation Environment for Mobile Robots*, 2006.

[Stu02] Stulce, John R.: *Conceptual Design and Simulation of a Multibody Passive-Legged Crawling Vehicle*. Dissertation, Faculty of the Virginia Polytechnic Institute and State University, 2002.

[TV95] Teller, Astro und Manuela Veloso: *PADO: Learning Tree Structured Algorithms for Orchestration into an Object Recognition System*. Technischer Bericht CMU-CS-95-101, Pittsburgh, PA, USA, 1995.

[Unb] Unbekannt: *ODE FAQ. What units should I use with ODE?* Online im Internet: http://opende.sourceforge.net/wiki/index.php/FAQ (15.10.2006).

[WN] Wolff, Krister und Peter Nordin: *Learning Biped Locomotion from First Principles on a Simulated Humanoid Robot using Linear Genetic Programming*. Technischer Bericht, Chalmers University of Technology.

[Zie03] Ziegler, Jens: *Evolution von Laufrobotersteuerung mit Genetischer Programmierung*. Dissertation, Universität Dortmund, 2003.

Anhang A.

Genetische Programmierung

A.1. Koza Standard Parameter

Hauptparameter	
Populationsgrösse	M = 500
Anzahl Generationen	51
Nebenparameter	
Wahrscheinlichkeit Rekombination	90%
Wahrscheinlichkeit Reproduktion	10%
Wahrscheinlichkeit Interne Rekombinationspunkte zu wählen	90%
Maximale Baumtiefe während des Laufs	17
Maximale Baumtiefe bei Initialisierung	6
Wahrscheinlichkeit Mutation	0%
Wahrscheinlichkeit Permutation	0%
Wahrscheinlichkeit Editing	0%
Wahrscheinlichkeit Encapsulation	0%
Bedingung für Dezimation	-
Überlebende der Dezimation in Prozent	0%
Strategische Parameter	
Initialisierungsmethode	ramped half-and-half
Grundlegende Selektionsmethode	fitness proportional
Selektionsmethode für Eltern (Rekombination, Reproduktion)	fitness proportional
Fitnessmaß	adjusted fitness
Überselektion	ab M >= 100
Elitist Strategy	nein

Tabelle A.1.: Standard-Kontrollparameter für GP nach [Koz92, S. 116]

A.2. Koza GP Ablauf

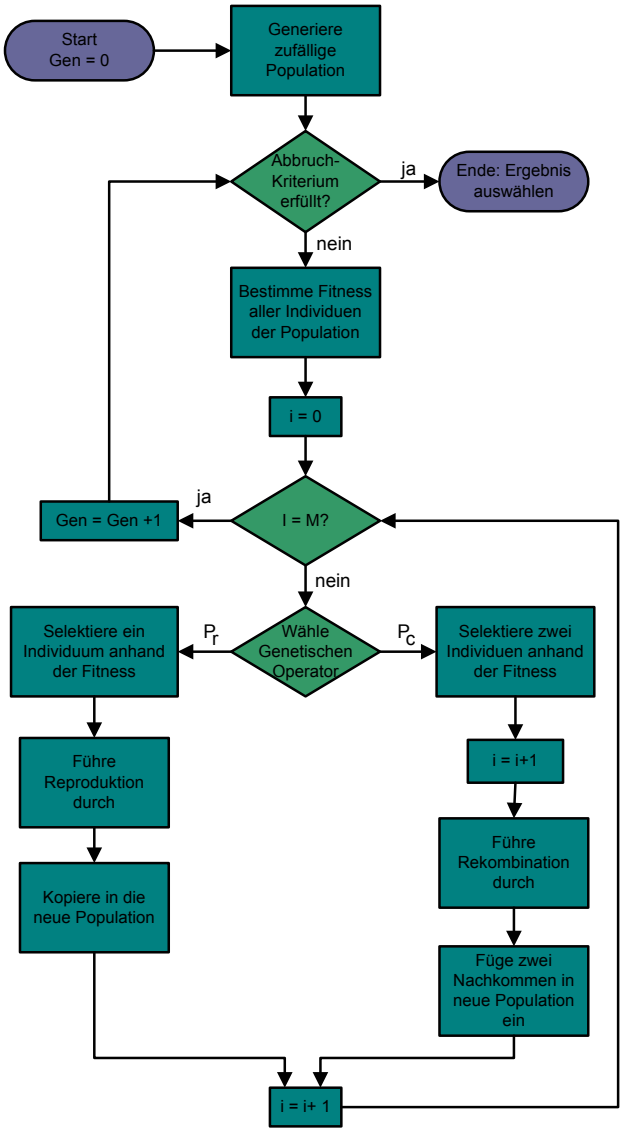

Abbildung A.1.: Ablauf der Genetischen Programmierung (nach [Koz92, S. 76])

Anhang B.

Simulationssysteme

System	**BREVE**	**Framsticks**	**SimulatorBob**
Version	2.4b1	2.10	1.4
Visualisierung	OpenGL	OpenGL	OpenGL
Physiksimulation	ODE	Mechastick	ODE
Erweiterbarkeit	Skriptsprache, Plugins	Skriptsprache	Rekompilierung
Quellfreiheit	GNU Lizenz	Nicht offen	GNU Lizenz
Dokumentation	Ausführlich	Ausführlich	Teilweise
Entwicklung	Aktiv, wachsende Entwicklergemeinschaft	Entwicklergemeinschaft	Aktiv, keine Gemeinschaft
Verweis	[Kle02]	[Kom00]	[Ste06]

Tabelle B.1.: Vergleich der untersuchten Simulatorsysteme I. Stand: Februar 2006

System	**Sigel**	**Darwin2k**
Version	1.0	0.91
Visualisierung	OpenGL	OpenGL/ MesaGL
Physiksimulation	Dynamo, DynaMechs	Eigene
Erweiterbarkeit	Rekompilierung	Rekompilierung
Quellfreiheit	GNU Lizenz	GNU Lizenz
Dokumentation	Ausführlich	Kaum
Entwicklung	Inaktiv seit 09/2001, keine Gemeinschaft	Inaktiv seit 04/2002
Verweis	[Zie03]	[Leg99]

Tabelle B.2.: Vergleich der untersuchten Simulatorsysteme II. Stand: Februar 2006

Anhang C.

Experimente

C.1. Testrechner

Prozessor	AMD Athlon XP 2400+
RAM	1 GB
Grafik	ATI Radeon 9800 Pro
Betriebssystem	Microsoft Windows XP SP2

Tabelle C.1.: Ausstattung des Testsystems

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Brandenburg, 29. November 2006 _____

Unterschrift